



Leandro Rabindranath León

Programador, ingeniero de Sistemas, mención Sistemas de Control, magíster en Informática Teórica, Cálculo y Programación y doctor en Sistemas Distribuidos, ambos títulos obtenidos en la Universidad Pierre et Marie Curie, París, Francia, Leandro Rabindranath León es profesor del departamento de Computación de la Universidad de Los Andes (ULA) y miembro fundador del Centro Nacional de Desarrollo e

Investigación en Tecnologías Libres, el cual presidió. Participante en proyectos de ingeniería en el ámbito nacional e internacional, investigador en Sistemas Distribuidos y Algoritmos y propulsor del software libre en Venezuela, ha sido ponente en diversos congresos y conferencias sobre soberanía tecnológica.

Tomo I

Este tomo presenta los fundamentos de abstracción y análisis de algoritmos y estructuras de datos. Por cada tema se cubren los tres aspectos esenciales de ingeniería: el propósito y diseño, el análisis mostrativo matemático y la implementación cabal de la idea o concepto. Este primer tomo puede emplearse como texto o referencia en cursos medianos de programación de carreras en ciencias, ingeniería y educación que tengan vínculos con la computación.

ISBN: 978-980-11-1391-1



97 8980 1113911



Leandro Rabindranath León



Tejiendo algoritmos

Abstracción, crítica, secuencias y árboles

Tomo I



Leandro Rabindranath León

Tejiendo algoritmos

Abstracción, crítica, secuencias y árboles

Tomo I

Universidad de Los Andes
Consejo de Publicaciones
Postgrado en Computación
Departamento de Computación

Tejiendo algoritmos

v 1.0b

Leandro Rabindranath León

lrleon@ula.ve

Centro de Estudios en Microelectrónica y Sistemas Distribuidos
Universidad de Los Andes

1 de septiembre de 2010

Prefacio

Quisiera que este texto se considerase como un “vademecum” en el diseño efectivo y en la programación de algoritmos y de estructuras de datos que soporten programas computacionales. *Vademécum* es una palabra latina construida mediante el imperativo latín *vade* que connota “ven”, “anda” o “camina”, y *mecum*, que significa “conmigo”. Aprender es parecido a transitar un camino; enseñar es parecido a mostrárselo a un peregrino. Enseñar proviene del latín “*insignāre*”, verbo compuesto de “*in*” (en) y “*signare*” (señalar); la expresión aún se emplea cuando se señala con el dedo una senda a seguir. Así, un vademecum pretende ser una guía de “tránsito” por una senda de aprendizaje; en este caso, la de programación de computadoras.

Modo y medios

Permitásemelme introducir el modo de enseñanza de este texto, expresado por este antiquísimo proverbio oriental:

“Cuando escucho, olvido,
Cuando veo, recuerdo,
Cuando hago, entiendo.”

La apropiación efectiva de un conocimiento ocurre cuando éste se “entiende”. El entendimiento se torna realidad cuando un aprendiz consuma la hechura de cosas características de su práctica haciéndolas él mismo. Dicho de otra manera, el modo de enseñanza es mostrar enteramente cómo se elabora una estructura de datos y un algoritmo. Bajo este modo intentaré guiar a un potencial aprendiz por el camino de la práctica de la programación avanzada. Para ello empleo algunos medios.

C++

Para representar los algoritmos y las especificaciones de estructuras de datos empleo el lenguaje de programación C++.

Esta decisión no se tomó sin dignas objeciones, las cuales, resumidamente, se pueden clasificar en dos grupos. El primero cuestiona el eventual desconocimiento del lector sobre el lenguaje C++. A esto debo replicar que no sólo C++ es uno de los lenguajes más populares e importantes de la programación de sistemas, sino que su sintaxis y semántica son reminiscentes a la mayoría de los otros lenguajes procedurales y a objetos. De hecho, en el quórum actual de los lenguajes de programación procedurales no sólo domina C++, sino que el resto de los lenguajes se inspiran en él o en su precursor, el lenguaje C; por instancias, java, python, perl, C# y D.

El segundo tipo de objeción denuncia que la comprensión se torna más difícil, a lo cual replico con dos argumentos. El primero es que los lenguajes de programación se pensaron también en un estilo “coloquial”¹ respecto a la programación. Consideremos, por ejemplo, el siguiente pseudoprograma en un pseudolenguaje castellano:

```
Repita mientras m > 0 y m < n
    si a[m] < a[x] entonces
        m = m/2;
    de lo contrario
        m = 2*m
    fin si
Fin repita mientras
```

el cual es equivalente a la traducción literal, “coloquial”, del siguiente bloque en C++:

```
while (m > 0 and m < n)
{
    if (a[m] < a[x])
        m = m/2;
    else
        m = 2*m;
}
```

Aunque un ejemplo no basta para generalizar mi defensa sobre el uso del lenguaje C++, el hecho es que cualquiera que considere seriamente la programación tendrá que programar y esto deberá hacerlo en un lenguaje de programación, el cual, para bien o mal, fue pensado en lengua inglesa. Por tanto, inevitablemente, un aprendiz hispanoparlante deberá programar en un lenguaje de programación “anglizado”. Hecha la acotación anterior, también podemos decir que el programa en castellano y su equivalente en C++ tienen el mismo significado. Así pues, es dudoso, cuando menos, afirmar que la comprensión se torna más difícil.

Una bondad que se atribuye al uso de un pseudolenguaje es que éste es independiente de la implementación. En mi criterio, esto es parcialmente correcto. Por ejemplo, las plantillas en C++ plantean un problema de “portabilidad” en otros lenguajes que no las tienen. Alguien podría aducir que el uso de plantillas son crípticas para el aprendiz. Pero esta objeción sólo tiene sentido si no se comprende el concepto y fin de una plantilla, cual no es otro que la genericidad. Entendido esto, un programa con plantillas es tanto o más genérico que uno expresado en un pseudolenguaje; y resulta que éste es el principal argumento de quienes defienden la enseñanza de programación en un pseudolenguaje!. De todos modos, si se usase un pseudolenguaje, entonces también habría que plantearse el problema de traducir a un lenguaje de programación real². Mi segundo argumento estriba en que, habida cuenta de la popularidad y transcendencia del C++, creo que es más fácil traducir un programa bien estructurado en C++ hacia otro lenguaje, por ejemplo, java, que uno realizado en un pseudolenguaje³.

¹Es imposible ser completamente coloquial en un lenguaje de programación.

²Una traducción con más esfuerzo si se trata de un pseudolenguaje en castellano.

³De hecho, muchos textos de programación aparecen en diferentes ediciones para distintos lenguajes. En muchos de estos casos, el autor escribe los programas en un solo lenguaje y luego emplea traductores hacia el otro lenguaje. Tales son los casos de Sedgewick o Weiss y sus series en C, C++ y java.

Biblioteca *ALEPH*

Este texto contiene en sí mismo una implementación concreta de una biblioteca, libre, llamada *ALEPH*, contentiva de todas las estructuras de datos y algoritmos tratados en este texto.

La lectura posibilita la del fuente de la biblioteca y revela aspectos de su instrumentación. Salvo errores aún no descubiertos o mejoras que cualquiera desee proponer, el lector tiene la posibilidad de apoyarse en una implantación concreta que le facilite la apropiación de sus conocimientos.

Los códigos fuentes de *ALEPH* están disponibles en:

<http://webdelprofesor.ula.ve/ingenieria/lrleon/aleph.tbz>

Ellos fueron automáticamente indentados con bcpp [17].

A lo largo de mi experiencia como enseñante he intentado recompensar a los descubridores de errores y proponentes de mejoras con una ponderación en su calificación. No veo ningún impedimento a que un tercero aplique lo mismo si este texto se usa como material instruccional. Si algún lector fuera de mi círculo de enseñanza desea reportar algún error o hacer alguna mejora, entonces le agradezco que lo reporte al email lrleon@ula.ve.

La documentación de la biblioteca está disponible en el enlace:

<http://webdelprofesor.ula.ve/ingenieria/lrleon/aleph/html>

Ésta fue escrita para procesarse con el excelente programa doxygen [38].

Programación literaria

En la elaboración de este texto se utilizó un sistema llamado noweb [147, 86], el cual es un sistema programado que genera este texto y los fuentes en C++ a partir de un solo archivo “fuente”. El “fuente” entrelaza prosa explicativa con “bloques de código” de la implantación de la biblioteca.

Este estilo de escritura de programas se denomina “programación literaria” y fue propuesto por Knuth [96]. La idea es presentar un estilo más coloquial para escribir programas que el mero estilo deformado de un lenguaje de programación, junto con la ganancia de que la documentación y la última versión del programa (presumiblemente correcta) residan en un sólo fuente.

A parte de código en C++, los bloques pueden contener referencias a otros bloques. Una definición de bloque comienza por su nombre entre paréntesis angulares. Por ejemplo, consideremos determinar si un número n es o no primo. Para ello, podemos definir el siguiente bloque:

```
iii <Calcular si n es primo iii>≡
    if (n <= 2)
        // n es un número primo
    const int raíz_de_n = static_cast<int>(ceil(sqrt(n)));
    for (int i = 3; i < raíz_de_n; i += 2)
        if (n % i == 0)
            // n no es primo
    // n es un número primo
```

Los bloques se enumeran según el número de la página en que se definen por primera vez. Si hay más de un bloque en una página, entonces a éste se le añade una letra que, en el orden alfabético, se corresponde con su orden de aparición. Algunos bloques referencian a otros bloques o variables.

Los bloques están escritos en un orden que -“se cree”- es preferible para la comprensión que el mero listado de código. Así, agradeceré toda crítica que se me pueda hacer sobre el estilo y orden de presentación de una estructura de dato o algoritmo en programación literaria, pero solicito al crítico un esfuerzo primigenio por comparar el estilo noweb con el listado plano de código y, entonces, bajo esa consideración, hacer su crítica.

Un instrumento que podría ser útil es el índice de identificadores de código (pag. ??), el cual que se encuentra en el apéndice y contiene los identificadores de clases y métodos definidos a lo largo del texto. Para cada identificador, se ubica en subrayado el número de página dónde fue definido y los números de las páginas de segmentos de código que le hacen referencia.

Puede decirse que este texto contiene la implantación de cuanta estructura de datos o algoritmo se estudie. Empero, tomarse esto al pie de la letra acarrearía bastante papel; además de que haría este texto más voluminoso y repetitivo. En ese sentido, en pro del espacio, se han hecho “cortes” de dos tipos:

1. Manejo de excepciones: aunque este aspecto constituye una de las bondades más elegantes de C++, este texto casi no los presenta.
2. Métodos de clases repetitivos o muy simples.

En ambos casos, el código de la biblioteca, directamente generado a partir del fuente noweb de este texto, está completo; es decir, la biblioteca contiene los manejadores de excepciones y métodos omitidos en este texto.

Ejercicios

Cualquiera que sea la práctica, no hay otra manera de que un practicante consume sus conocimientos que no sea “haciendo práctica”. Por más esfuerzo al orientarle y enseñarle, el aprendiz debe, preferiblemente guiado por un maestro, ejercitarse por sí solo lo aprendido.

Hay tres maneras, que no deben ser excluyentes, de llevar a cabo lo anterior:

1. Resolución de ejercicios propuestos: en este sentido, al final de cada capítulo se presenta un conjunto de ejercicios destinados primordialmente a la resolución en solitario por parte del aprendiz.

Algunos ejercicios son “teóricos” en el sentido de que no necesariamente requieren escribir un programa compilable. Por supuesto, no está prohibido sentarse frente al computador e intentar concretar el ejercicio mediante un programa. Otros ejercicios son prácticos y consisten en extensiones a la biblioteca. Estos ejercicios son distinguibles de los teóricos porque enuncian su resultado en términos de un objeto de la biblioteca.

Los ejercicios están clasificados según una dificultad subjetiva juzgada por mí. No es fácil ponderar el tiempo de resolución de un ejercicio, pues éste depende del estudiante, pero he aquí mi clasificación:

(a) Ninguna cruz denota a un ejercicio considerado sencillo. Los teóricos deberían de resolverse en el orden de cinco minutos, mientras que los prácticos en el de un día.

(b) Una cruz (+) expresa un tiempo estimado de una a dos horas en el caso de un ejercicio teórico y de dos días en el práctico.

(c) Dos cruces (++) expresan ya una dificultad mucho mayor. Por lo general, esta clase de ejercicios plantean al aprendiz un descubrimiento o revelación de un “truco” o “técnica” que, una vez revelada, debe reducir la complejidad del ejercicio a ninguna cruz.

El tiempo de un ejercicio teórico debe ser al menos de un día, mientras que el de uno práctico será de tres (3).

(d) Tres cruces (+++) representan una ejercicio de élite, difícil aún para un maestro, cuyo tiempo de resolución no está delimitado.

2. Ejecución de ejercicios guiados en laboratorio: uno de los grandes obstáculos que lo abstracto plantea al aprendiz -uno diría que ello ocurre en cualquier práctica-, es que su condición novicia le dificulta aceptar que lo que para él puede ser en principio abstracto se convertirá, a través del ejercicio, en concreto.

Dicho lo anterior, puede decirse que el sentido de un ejercicio en laboratorio es permitirle al estudiante “iniciar” la concreción de sus conocimientos. A tal fin, en el laboratorio, pueden llevarse a cabo tres actividades:

(a) Contemplación de un problema computacional, junto con su solución, en el cual se haga uso de alguna estructura de datos o algoritmo de estudio.

El guía de laboratorio puede enunciar el problema y presentar la instrumentación de su solución. Luego, invitar al estudiante a revisar los fuentes e inferir, antes de su ejecución, las técnicas y estilos utilizados en la instrumentación.

(b) Contemplación de la ejecución del programa para diversas combinaciones de entrada. Esta es la fase en la cual el aprendiz comienza a sentir concreción; es decir, que los conceptos abstractos desemboquen en soluciones concretas.

Durante esta actividad puede ser recomendable la utilización de un “depurador”.

(c) Finalmente, resolución de una variante del problema a partir de la solución primigenia. En este punto es muy importante que se trabaje sobre el mismo problema y fuentes, pero con alguna variante; una ampliación para resolver otras clases de entradas, por ejemplo.

3. Trabajos prácticos: indudablemente, en el espíritu del proverbio citado, esta es la fase que encaja con el “*hacer para entender*”.

¿Cómo debe ser el proyecto? Debe corresponder a un problema original para y con sentido al contexto en el cual se enseñe, es decir, formulado por y para usarse en la comunidad en donde se imparta este curso. Por ejemplo, si aledaño al lugar de enseñanza se padecen de problemas de tráfico, entonces puede plantearse un conjunto amplísimo de proyectos en torno a la simulación del tráfico y al estudio de diversas técnicas para evitar su congestión. Un proyecto de este tipo requeriría, por ejemplo, la modelización con grafos de las vías de circulación, de mecanismos de control como los semáforos y de agentes circulantes como los automóviles.

So riesgo de ser excesivamente repetitivo, debo insistir en la importancia de los ejercicios. Esto ya debe ser obvio desde la perspectiva del estudiante, pues es el único medio de ejercitarse. Al instructor, por otra parte, le permite enriquecer la enseñanza cuando se plantea ejercicios que completan o complementan el conocimiento impartido, así como, mediante la corrección, supervisar el nivel de sus estudiantes.

Estructura del texto

Por razones didácticas y de espacio, este texto está dividido en dos tomos.

El primero de ellos es fundamental y comprende un curso mediano y riguroso de algoritmos y estructuras de datos, así como técnicas para criticar su efectividad y eficiencia. Este tomo puede emplearse como libro texto o de referencia para un curso de estructuras de datos, diseño de algoritmos o programación mediana.

El segundo tomo es ya un texto avanzado y se consagra a dos temas principales: técnicas avanzadas de recuperación de información en memoria primaria y grafos. Los grafos conforman uno de los mundos más complejos de la computación y de la optimización, y su aplicabilidad es vasta para otros dominios que transcinden las ciencias computacionales. En pos del mejor rendimiento de algoritmos sobre grafos, a menudo es necesario emplear técnicas de alto rendimiento para recuperar información en memoria primaria, pero estas técnicas también son requeridas en otros mundos algorítmicos. De ahí el porqué de su inclusión.

La comprensión de tomo II exige un nivel de programación avanzado y buenas cualidades en el diseño y análisis de algoritmos y estructuras de datos. Este tomo puede usarse como texto o referencia de un curso avanzado de algoritmos o de grafos, tanto en el ámbito de pregrado como en el de postgrado.

Ambos tomos pueden fungir de referencia para el quehacer ingenieril.

Estructura de este tomo

El presente tomo comienza por abordar en el capítulo 1 los fundamentos de abstracción empleados en el resto del texto.

El capítulo 2 se adentra profundamente en la idea de secuencia, ubicua e indispensable en la programación.

Una vez dominada e instrumentada cabalmente la idea de secuencia, el capítulo 3 estudia las técnicas conocidas para criticar un algoritmo o estructuras de datos, tanto desde la perspectiva de la eficiencia (análisis) como desde la de la efectividad (correctitud).

Finalmente, el capítulo 4 se consagra a estudiar la estructura de datos “árbol”, cual también es ampliamente usada en la computación.

Historia

Comencé la elaboración de la biblioteca *ALÉPH* en mayo de 1998. En aquel entonces me consagré a escribir clases de objetos para representar listas (las jerarquías Slink y Dlink), árboles binarios (las jerarquías BinNode<Key> y Avl_Tree<Key>) y tablas hash (la clase LhashTable<Key>).

Partes de este texto comenzaron a aparecer a mediados de 1999, luego de que algunos estudiantes me observasen que les sería útil leer algoritmos en código y que éstos fuesen bien comentados. Fue entonces cuando apelé a noweb, cuya magistral efectividad ya había conocido cuando, estudiando generación dinámica de código, me fue oportunísimo leer el libro de compiladores de Hanson y Fraser [69]. En ese tiempo escribí parte de lo que hoy es el capítulo 2, concerniente a secuencias. A mediados del 2000 comencé el capítulo 4 sobre árboles y parte del 5, referente a las tablas hash. El capítulo 4 ha tenido bastantes modificaciones a su contenido original y añadiduras, ocurridas en su mayor parte en el largo período comprendido entre 2001 y 2004.

En noviembre del 2001 redacté casi enteramente lo que actualmente es el capítulo 6 sobre equilibrio de árboles.

A mediados del 2004 inicié la extensión de la biblioteca hacia grafos, tema presentado en el capítulo 7. Las estructuras y algoritmos en torno a este dominio han variado bastante en forma y no ha sido hasta agosto del 2007 cuando han tomado una versión estable. Creo que en este dominio es donde este texto plasma sus principales aportes.

Desde febrero de 2006 me planteé el esfuerzo de revisar y unificar los capítulos bajo lo que hoy conforma este texto. Escribí enteramente los capítulos 1, sobre abstracción de datos, y el 3, concerniente a la crítica de algoritmos y estructuras de datos.

El septiembre de 2007 culminé el capítulo 5 referente a las tablas hash.

La mayoría de las figuras de este libro fue generada automáticamente mediante programas elaborados con la propia biblioteca *ALCEPH*. En ese sentido, hay tres programas:

1. btreepic para dibujar árboles binarios
2. ntreetopic para dibujar arborescencias y árboles en general
3. graphpic para dibujar grafos

El primer programa, btreepic, fue producto de mi insatisfacción con los dibujos de árboles binarios generados con programas especiales tales como Xfig y dia. A esto se aunó la imposibilidad material de dibujar árboles enormes -de cientos o miles de nodos-. En virtud de esto solicité un trabajo escolar al respecto, cuyos resultados, a pesar de satisfacerme escolarmente, distaron de serme suficientes. A raíz de esa experiencia decidí por mi propia cuenta programar btreepic. Cuando tuve que dibujar árboles generales solicité el mismo tipo de programa; entonces apareció un estudiante excelsio, llamado José Brito, quien forjó una primera versión llamada xtreetopic y que está distribuida en *ALCEPH*. Lamentablemente, esta versión operaba sobre una versión de árboles que a la época de mi necesidad y revisión ya era caduca, por lo que preferí rehacer enteramente el programa bajo el nombre de ntreetopic. Para la elaboración del programa me fue muy útil el hermoso texto sobre dibujado de grafos de Kozo Sugiyama [165]. Finalmente, cuando me encontré en la misma necesidad respecto a los grafos, aproveché la ocasión para desarrollar casi enteramente el corpus algorítmico en geometría computacional. El programa resultante, graphpic, es aún muy simple y evade toda la algorítmica vinculada al dibujado automático de grafos; de hecho, las decisiones sobre dibujado son tomadas por el usuario, pero podría decir que graphpic tiene la virtud de operar enteramente sobre geometría computacional y que ello no sólo valida este campo, sino que abre futuros desarrollos.

En Marzo de 2008 inicié la escritura de la documentación de la biblioteca. Me enfrenté a un problema: ¿Cómo integrar la documentación en el fuente de este texto sin que ésta

aparezca en el texto? Requería escribirla en el mismo sitio donde escribí este libro porque de ese modo, bajo la misma doctrina de noweb, una modificación de la biblioteca se podría actualizar rápidamente en la documentación. Decidí entonces diseñar un filtro de texto que reconociese los bloques de documentación dentro del fuente noweb y los eliminase de manera que no apareciesen en la salida \LaTeX . Tal filtro se llama `deldoxygen` y fue escrito con el generador de analizadores lexicográficos `flex` y `C++`. Alguien dirá que pude haberlo hecho en treinta minutos con uno de los lenguajes de “scripting” modernos (`perl`, `python`, etc.). Probablemente sea cierto si yo fuese maestro en alguno de aquellos lenguajes, pero, en añadidura, déjeseme replicar con tres hechos. Primero, demoré un par de horas en escribir `deldoxygen` porque tenía quince años sin usar `flex` y no recordaba bien el lenguaje; no es pues mucha la diferencia. Segundo, mi filtro tiene muchas más posibilidades de ser correcto, pues fue especificado -no programado- bajo el formalismo de las expresiones regulares y los autómatas; en un lenguaje de scripting, esta correctitud tenía que programarse y no especificarse como fue el caso. Finalmente, el filtro debe ser considerablemente más veloz que una contraparte scripting; yo diría, cuando menos, en dos órdenes de magnitud.

Cuando me encontraba en la edición final de esta versión (Marzo 2008) hube de aprehender que el texto contenía (y aún contiene) bloques noweb con pedazos de código sin mucho valor didáctico, por ejemplo, repetir funciones cuyo sentido ya fue explicado en otro lugar para otra clase de objeto. Decidí entonces diseñar otro filtro de texto que reconociere delimitadores dentro del fuente noweb y que se invocase antes de noweb. El filtro se denomina `nobook`, fue escrito también en `flex` y elimina, para la salida \LaTeX , los bloques delimitados.

Cosas que faltan y sobran

Desde muchas perspectivas, siempre un texto adolece de falta de algo. Aquí deseo referirme a lo que, “creo”, hubiera debido perfectamente hacer y a lo que, “con certitud”, no debí hacer.

Creo sinceramente que las estructuras de datos y algoritmos fundamentales están presentes en este texto, aunque, con seguridad, algún par discrepará. Por otra parte, material que a mi juicio es de alto interés, que está presente en la biblioteca `ALCÉPH`, no está presente en este texto. Los ejemplos más notables de eso son las listas skip, coloreados de grafos, caminos eulerianos y hamiltonianos, estructuras de grafos concurrentes, de agentes y de simulación, estructuras de grafos y sus algoritmos basados en colonias de hormigas y geometría computacional. Me habría gustado incluirlas en este libro, pero ya me sobrepasó el momento y agoté el límite de espacio.

Aspectos no desarrollados en `ALCÉPH` y que serían dignos de incluirse son los heaps de Fibonacci, las familias de árboles dedicados a sistemas de archivo y búsqueda en memoria secundaria (B^+ y derivados) y los árboles quadtrees.

En este texto se apela al lenguaje de modelado UML para “facilitar la compresión de relaciones entre objetos”. Creo sinceramente que UML tiene mucho valor para comprender sistemas complejos, ya desarrollados, y un poco menos de valor, aunque apreciable, para diseñarlos. Pero en lo que atañe a este texto y sus cursos derivados, no es de trascendente utilidad.

Deudas

Sea cual sea la obra, ésta se circumscribe gracias a y para una cultura. La primera deuda, pues, que cualquier individuo adquiere con una obra es hacia su cultura, la cual le entrega el trasfondo circunstancial, en conocimientos y sentimientos, que posibilitan e inspiran la obra en cuestión. En este mismo espíritu, una vez entregada la obra, otra adquirida es hacia la cultura que recibe y reconoce la obra.

Parafraseando a Ortega y Gasset, uno es uno y su circunstancia, pero cualquiera que ésta sea, en ella siempre se encuentra la influencia abrumadora del otro. Me siento, pues, en franca deuda hacia quienes siento que debo lo que soy y, en lo particular de este texto, hacia aquellos que incidieron muy directamente en su elaboración.

Víctor Bravo, Carlos Nava, Juan Luís Chaves y Juan Carlos Vargas fueron mis primeros discípulos en esta y el área de sistemas distribuidos. Víctor instrumentó la primera versión de la clase `LinearHashTable<Key>` presentada en § 5.1.7 (Pág. 412). Carlos instrumentó la primera versión de un sistema comunicacional de envergadura sustentado en el uso de \mathcal{ALEPH} ; el sistema aún es operativo hoy en día. Juan Carlos instrumentó los árboles AVL hilados y con rangos, los cuales, si bien no están presentes en este texto, su instrumentación ayudó a mejorar y depurar la clase `Avl_Tree<Key>`.

Andrés Arcia fue un usuario intensivo de \mathcal{ALEPH} durante de su tesis de maestría, lo cual me permitió ver aspectos que luego incidieron en extensiones y mejoras de la biblioteca.

Leonardo Zúñiga, Bladimir Contreras y Carlos Acosta diseñaron y programaron `treepic`, precursor de `btreepic`, un programa para dibujar los árboles binarios de este libro. José Brito escribió `xtrreepic`, precursor de `ntreepic`, usado para dibujar árboles y arborescencias generales.

Jorge Redondo y Tomás López hicieron las primeras pruebas de desempeño sobre los diversos árboles binarios de búsqueda.

Jesús Sánchez hizo parte de la implantación parcial de la biblioteca estándar C⁺⁺ bajo \mathcal{ALEPH} . En pruebas de desempeño tradicionales, la biblioteca estándar bajo \mathcal{ALEPH} es de mejor desempeño que la de GNU.

Juan Fuentes instrumentó parte de y depuró la clase genérica de árbol `Tree_Node`. Orlando Vicuña y Alejandro Mujica encontraron errores importantes en los árboles y grafos, así como plantearon sugerencias muy apreciables sobre el estilo de implantación.

En la confección de este texto se han empleado enteramente programas libres: `LATEX` [106], `TEX` [169], `noweb`, `BIBTEX` [22], `gnu make` [30], `imake` [82], `gnuplot` [61], `R` [146], `Maxima` [121], `Xfig` [185], `dia` [36], `Umbrello` [170], `doxygen` [38], `bcpp` [17], `graphviz` [58, 42, 43], entre otros. Este texto y los programas fueron editados con `gnu Emacs` [44]. Los programas fueron manejados con todos los utilitarios GNU [60].

Juan Acevedo, profesor de la Facultad de Humanidades de la Universidad de Los Andes, me supervisó los comentarios etimológicos en latín y griego.

Luis Paniagua, corrector del Consejo de Publicaciones de la Universidad de Los Andes, se ha tomado muy gentilmente su deber de revisar concienzudamente este texto. A ese tenor, debo expresarle mi gratitud por sus numerosas y sorprendentes correcciones y enseñanzas.

Algunas veces, al observar algunos gestos y actitudes en discípulos me parece notarles algunas de mis enseñanzas, lo cual evoca la lejana posibilidad de mi impronta. Pero a ese

tenor debo aclarar que soy yo, más bien, quien porta sus lecciones y, por tanto, quien les expresa gratitud.

Mis amadísimos padres, Nelly y Adelis, han contribuido a lo que me atribuiría como una sensibilidad particular hacia la tecnología. Mi madre leyó y corrigió enteramente este trascrito, así como ellos, otrora mi adolescencia, me enseñaron a escribir un poco.

He observado que casi todo autor de libro texto técnico expresa agradecimientos a su familia (esposo(a) e hijo(a)(s)). En el transcurso de esta edición me percaté de que ello probablemente obedezca a que a ellos, en mi caso con descarada e irresponsable negligencia, uno les olvida. Por razones muy íntimas, para nada técnicas, no puedo medir ni expresar cómo y cuánto soy gracias a mi esposa, Magdiel, pero sí puedo clamar que no sería nada sin ella. Sea pues con y hacia ella, por su amor y su perdón, mi mayor y principal deuda ... y gratitud.

Índice

1 Abstracción de datos	1
1.1 Especificaciones de datos	3
1.1.1 Tipo abstracto de dato	4
1.1.2 Noción de clase de objeto	5
1.1.3 Lo subjetivo de un objeto	6
1.1.4 Un ejemplo de TAD	7
1.1.5 El lenguaje UML	10
1.2 Herencia	11
1.2.1 Tipos de herencia	12
1.2.2 Multiherencia	12
1.2.3 Polimorfismo	13
1.3 El problema fundamental de estructuras de datos	17
1.3.1 Comparación general entre claves	19
1.3.2 Operaciones para conjuntos ordenables	19
1.3.3 Circunstancias del problema fundamental	19
1.3.4 Presentaciones del problema fundamental	20
1.4 Diseño de datos y abstracciones	20
1.4.1 Tipos de abstracción	21
1.4.2 El principio fin-a-fin	22
1.4.3 Inducción y deducción	22
1.4.4 Ocultamiento de información	23
1.5 Notas bibliográficas	24
1.6 Ejercicios	25
2 Secuencias	27
2.1 Arreglos	28
2.1.1 Operaciones básicas con Arreglos	29
2.1.2 Manejo de memoria para arreglos	32
2.1.3 Arreglos dinámicos	34
2.1.4 El TAD DynArray<T>	34
2.1.5 Arreglos de bits	53
2.2 Arreglos multidimensionales	58
2.3 Iteradores	59
2.4 Listas enlazadas	61
2.4.1 Listas enlazadas y el principio fin a fin	64
2.4.2 El TAD Slink (enlace simple)	65

2.4.3	El TAD Snode<T> (nodo simple)	68
2.4.4	El TAD Slist<T> (lista simplemente enlazada)	69
2.4.5	Iterador de Slist<T>	70
2.4.6	El TAD DynSlist<T>	71
2.4.7	El TAD Dlink (enlace doble)	72
2.4.8	El TAD Dnode<T> (nodo doble)	83
2.4.9	El TAD DynDlist<T>	84
2.4.10	Aplicación: aritmética de polinomios	91
2.5	Pilas	98
2.5.1	Representaciones de una pila en memoria	99
2.5.2	El TAD ArrayStack<T> (pila vectorizada)	101
2.5.3	El TAD ListStack<T> (pila con listas enlazadas)	103
2.5.4	El TAD DynListStack<T>	105
2.5.5	Aplicación: un evaluador de expresiones aritméticas infijas	106
2.5.6	Pilas, llamadas a procedimientos y recursión	112
2.6	Colas	122
2.6.1	Variantes de las colas	123
2.6.2	Aplicaciones de las colas	123
2.6.3	Representaciones en memoria de las colas	123
2.6.4	El TAD ArrayQueue<T> (cola vectorizada)	124
2.6.5	El TAD ListQueue<T> (cola con listas enlazadas)	128
2.6.6	El TAD DynListQueue<T> (cola dinámica con listas enlazadas)	130
2.7	Estructuras de datos combinadas - Multilistas	131
2.8	Notas bibliográficas	133
2.9	Ejercicios	135
3	Crítica de algoritmos	145
3.1	Análisis de algoritmos	147
3.1.1	Unidad o paso de ejecución	148
3.1.2	Aclaratoria sobre los métodos de ordenamiento	150
3.1.3	Ordenamiento por selección	150
3.1.4	Búsqueda secuencial	154
3.1.5	El problema fundamental y los arreglos dinámicos	155
3.1.6	Búsqueda de extremos	156
3.1.7	Notación \mathcal{O}	157
3.1.8	Ordenamiento por inserción	162
3.1.9	Búsqueda binaria	165
3.1.10	Errores de la notación \mathcal{O}	166
3.1.11	Tipos de análisis	167
3.2	Algoritmos dividir/combinar	168
3.2.1	Ordenamiento por mezcla	169
3.2.2	Ordenamiento rápido (Quicksort)	173
3.3	Análisis amortizado	187
3.3.1	Análisis potencial	189
3.3.2	Análisis contable	191
3.3.3	Selección del potencial o créditos	192

3.4	Correctitud de algoritmos	193
3.4.1	Planteamiento de una demostración de correctitud	193
3.4.2	Tipos de errores	194
3.4.3	Prevención y detección de errores	194
3.5	Eficacia y eficiencia	210
3.5.1	La regla del 80-20	212
3.5.2	¿Cuándo atacar la eficiencia?	212
3.5.3	Maneras de mejorar la eficiencia	213
3.5.4	Perfilaje (profiling)	214
3.5.5	Localidad de referencia	214
3.5.6	Tiempo de desarrollo	215
3.6	Notas bibliográficas	216
3.7	Ejercicios	217
4	Árboles	223
4.1	Conceptos básicos	226
4.2	Representaciones de un árbol	228
4.2.1	Conjuntos anidados	228
4.2.2	Secuencias parentizadas	229
4.2.3	Indentación	230
4.2.4	Notación de Dewey	231
4.3	Representaciones de árboles en memoria	231
4.3.1	Listas enlazadas	231
4.3.2	Arreglos	232
4.4	Árboles binarios	233
4.4.1	Representación en memoria de un árbol binario	234
4.4.2	Recorridos sobre árboles binarios	234
4.4.3	Un TAD genérico para árboles binarios	238
4.4.4	Contenedor de funciones sobre árboles binarios	242
4.4.5	Recorridos recursivos	243
4.4.6	Recorridos no recursivos	246
4.4.7	Cálculo de la cardinalidad	249
4.4.8	Cálculo de la altura	249
4.4.9	Copia de árboles binarios	250
4.4.10	Destrucción de árboles binarios	250
4.4.11	Comparación de árboles binarios	250
4.4.12	Recorrido por niveles	251
4.4.13	Construcción de árboles binarios a partir de recorridos	253
4.4.14	Conjunto de nodos en un nivel	254
4.4.15	Hilado de árboles binarios	255
4.4.16	Recorridos pseudohilados	258
4.4.17	Correspondencia entre árboles binarios y m-rios	260
4.5	Un TAD genérico para árboles	264
4.5.1	Observadores de Tree_Node	266
4.5.2	Modificadores de Tree_Node	267
4.5.3	Observadores de árboles	269

4.5.4 Recorridos sobre Tree_Node	270
4.5.5 Destrucción de Tree_Node	270
4.5.6 Búsqueda por número de Deway	271
4.5.7 Cálculo del número de Deway	272
4.5.8 Correspondencia entre Tree_Node y árboles binarios	273
4.6 Algunos conceptos matemáticos de los árboles	275
4.6.1 Altura de un árbol	275
4.6.2 Longitud del camino interno/externo	277
4.6.3 Árboles completos	280
4.7 Heaps	282
4.7.1 Inserción en un heap	284
4.7.2 Eliminación en un heap	285
4.7.3 Colas de prioridad	288
4.7.4 Heapsort	289
4.7.5 Aplicaciones de los heaps	291
4.7.6 El TAD BinHeap<Key>	293
4.8 Enumeración y códigos de árboles	302
4.8.1 Números de Catalan	307
4.8.2 Almacenamiento de árboles binarios	311
4.9 Árboles binarios de búsqueda	313
4.9.1 Búsqueda en un ABB	315
4.9.2 El TAD BinTree<Key>	319
4.9.3 Inserción en un ABB	320
4.9.4 Partición de un ABB por clave (split)	321
4.9.5 Unión exclusiva de ABB (join exclusivo)	323
4.9.6 Eliminación en un ABB	324
4.9.7 Inserción en raíz de un ABB	326
4.9.8 Unión de ABB (join)	327
4.9.9 Análisis de los árboles binarios de búsqueda	329
4.10 El TAD DynMapTree	332
4.11 Extensiones a los árboles binarios	335
4.11.1 Selección por posición	337
4.11.2 Cálculo de la posición infija	338
4.11.3 Inserción por clave en árbol binario extendido	338
4.11.4 Partición por clave	339
4.11.5 Inserción en raíz	340
4.11.6 Partición por posición	340
4.11.7 Inserción por posición	341
4.11.8 Unión exclusiva de árboles extendidos	342
4.11.9 Eliminación por clave en árboles extendidos	342
4.11.10 Eliminación por posición en árboles extendidos	343
4.11.11 Desempeño de las extensiones	343
4.12 Rotación de árboles binarios	344
4.12.1 Rotaciones en árboles binarios extendidos	345
4.13 Códigos de Huffman	345
4.13.1 Un TAD para árboles de código	347

4.13.2 Decodificación	349
4.13.3 Algoritmo de Huffman	350
4.13.4 Definición de símbolos y frecuencias	352
4.13.5 Codificación de texto	353
4.13.6 Optimización de Huffman	355
4.14 Árboles estáticos óptimos	358
4.14.1 Objetivo	360
4.14.2 Implantación	360
4.15 Notas bibliográficas	363
4.16 Ejercicios	365
5 Tablas hash	379
5.1 Manejo de colisiones	381
5.1.1 El problema del cumpleaños	381
5.1.2 Estrategias de manejo de colisiones	382
5.1.3 Encadenamiento	383
5.1.4 Direccionamiento abierto	393
5.1.5 Reajuste de dimensión en una tabla hash	409
5.1.6 El TAD DynLhashTable (cubetas dinámicas)	410
5.1.7 Tablas hash lineales	412
5.2 Funciones hash	423
5.2.1 Interfaz a la función hash	423
5.2.2 Holgura de dispersión	423
5.2.3 Plegado o doblado de clave	424
5.2.4 Heurísticas de dispersión	424
5.2.5 Dispersión de cadenas de caracteres	429
5.2.6 Dispersión universal	430
5.2.7 Dispersión perfecta	431
5.3 Otros usos de las tablas hash y de la dispersión	431
5.3.1 Identificación de cadenas	431
5.3.2 Supertraza	434
5.3.3 Cache (el TAD Hash_Cache)	435
5.4 Notas bibliográficas	445
5.5 Ejercicios	446
6 Árboles de búsqueda equilibrados	451
6.1 Equilibrio de árboles	452
6.2 Árboles aleatorizados	455
6.2.1 El TAD Rand_Tree<Key>	455
6.2.2 Análisis de los árboles aleatorizados	460
6.3 Treaps	464
6.3.1 El TAD Treap<Key>	465
6.3.2 Inserción en un treap	467
6.3.3 Eliminación en treap	468
6.3.4 Análisis de los treaps	470
6.3.5 Prioridades implícitas	471
6.4 Árboles AVL	471

6.4.1	El TAD Avl_Tree<Key>	472
6.4.2	Análisis de los árboles AVL	483
6.5	Árboles rojo-negro	490
6.5.1	El TAD Rb_Tree<Key>	491
6.5.2	Análisis de los árboles rojo-negro	503
6.6	Árboles splay	507
6.6.1	El TAD Splay_Tree<Key>	509
6.6.2	Análisis de los árboles splay	513
6.7	Conclusión	519
6.8	Notas bibliográficas	523
6.9	Ejercicios	525
7	Grafos	535
7.1	Fundamentos	537
7.2	Estructuras de datos para representar grafos	542
7.2.1	Matrices de adyacencia	542
7.2.2	Listas de adyacencia	544
7.3	Un TAD para grafos (List_Graph)	545
7.3.1	Grafos	546
7.3.2	Digrafos (List_Digraph)	547
7.3.3	Nodos	547
7.3.4	Arcos	551
7.3.5	Atributos de control de nodos y arcos	555
7.3.6	Macros de acceso a nodos y arcos	561
7.3.7	Construcción y destrucción de List_Graph	562
7.3.8	Operaciones genéricas sobre nodos	563
7.3.9	Operaciones genéricas sobre arcos	563
7.3.10	Implantación de List_Graph	564
7.4	TAD camino sobre un grafo (Path<GT>)	578
7.5	Recorridos sobre grafos	581
7.5.1	Iteradores filtro	582
7.5.2	Recorrido en profundidad	585
7.5.3	Conectividad entre grafos	589
7.5.4	Recorrido en amplitud	589
7.5.5	Prueba de ciclos	592
7.5.6	Prueba de aciclicidad	593
7.5.7	Búsqueda de caminos por profundidad	596
7.5.8	Búsqueda de caminos por amplitud	600
7.5.9	Árboles abarcadores de profundidad	603
7.5.10	Árboles abarcadores de amplitud	605
7.5.11	Árboles abarcadores en arreglos	606
7.5.12	Conversión de un árbol abarcador a un Tree_Node	608
7.5.13	Componentes inconexos de un grafo	610
7.5.14	Puntos de articulación de un grafo	613
7.5.15	Componentes conexos de los puntos de corte	622
7.6	Matrices de adyacencia	628

7.6.1	El TAD Ady_Mat	629
7.6.2	El TAD Bit_Mat_Graph<GT>	633
7.6.3	Algoritmo de Warshall	635
7.7	Grafos dirigidos	637
7.7.1	Conejividad entre digrafos	638
7.7.2	Inversión de un digrafo	638
7.7.3	Componentes fuertemente conexos de un digrafo	639
7.7.4	Prueba de aciclicidad	648
7.7.5	Cálculo de ciclos en un digrafo	650
7.7.6	Prueba de conectividad	652
7.7.7	Digrajos acíclicos (DAG)	653
7.7.8	Planificación de tareas	654
7.7.9	Ordenamiento topológico	655
7.8	Árboles abarcadores mínimos	660
7.8.1	Manejo de los pesos del grafo	660
7.8.2	Algoritmo de Kruskal	661
7.8.3	Algoritmo de Prim	666
7.9	Caminos mínimos	674
7.9.1	Algoritmo de Dijkstra	675
7.9.2	Algoritmo de Floyd-Warshall	685
7.9.3	Algoritmo de Bellman-Ford	692
7.9.4	Discusión sobre los algoritmos de caminos mínimos	707
7.10	Redes de flujo	708
7.10.1	Definiciones y propiedades fundamentales	708
7.10.2	El TAD Net_Graph	710
7.10.3	Manejos de varios fuentes o sumideros	711
7.10.4	Operaciones topológicas sobre una red capacitada	713
7.10.5	Cortes de red	716
7.10.6	Flujo máximo/corte mínimo	718
7.10.7	Caminos de aumento	720
7.10.8	Cálculo de la red residual	723
7.10.9	Cálculo de caminos de aumento	725
7.10.10	Incremento del flujo por un camino de aumento	725
7.10.11	El algoritmo de Ford-Fulkerson	726
7.10.12	El algoritmo de Edmonds-Karp	731
7.10.13	Algoritmos de empuje y preflujo	736
7.10.14	Cálculo del corte mínimo	766
7.10.15	Aumento o disminución de flujo de una red	769
7.11	Reducciones al problema del flujo máximo	773
7.11.1	Flujo máximo en redes no dirigidas	773
7.11.2	Capacidades en nodos	774
7.11.3	Flujo factible	774
7.11.4	Máximo emparejamiento bipartido	777
7.11.5	Circulaciones	783
7.11.6	Conejividad de grafos	785
7.11.7	Cálculo de $K_v(e)$	792

7.12	Flujos de coste mínimo	792
7.12.1	El TAD Net_Max_Flow_Min_Cost	796
7.12.2	Algoritmos de máximo flujo con coste mínimo mediante eliminación de ciclos	798
7.12.3	Análisis de los algoritmos basados en eliminación de ciclos negativos	802
7.12.4	Problemas que se reducen a enunciados de flujo máximo a coste mínimo	802
7.13	Programación lineal	810
7.13.1	Forma estándar de un programa lineal	812
7.13.2	Un ejemplo de estandarización	813
7.13.3	Forma “holgada” de un programa lineal	814
7.13.4	El método simplex	815
7.13.5	Conclusión sobre la programación lineal	822
7.14	Redes de flujo y programación lineal	822
7.14.1	Conversión de una red capacitada de costes a un programa lineal .	822
7.14.2	Redes generalizadas	823
7.14.3	Capacidades acotadas	824
7.14.4	Redes con restricciones laterales	824
7.14.5	Redes multiflujo	825
7.14.6	Redes de procesamiento	825
7.14.7	Conclusión sobre el problema del flujo máximo a coste mínimo . .	828
7.15	Notas bibliográficas	829
7.16	Ejercicios	831
	Índice de identificadores	865

1

Abstracción de datos

Este texto concierne al diseño e implantación de estructuras de datos y algoritmos que instrumenten soluciones a problemas mediante programas de computador.

Un algoritmo es una secuencia finita de instrucciones que acomete la consecución de un fin. Usamos algoritmos en diversos contextos de la vida; por ejemplo, cuando preparamos un plato de comida según alguna receta. La cultura nos ha inculcado algunos “algoritmos”, culturales, no naturales, para desenvolvernos socialmente; por ejemplos, el algoritmo de conducir un automóvil o el algoritmo para cruzar una calle. En ambos ejemplos, el fin que se plantea es arribar a un sitio.

Según Knuth [97], el término “algoritmo” proviene del nombre del ancestral matemático *al-Khwārizmī*, de la Persia, parte del actual Irán, región del planeta muy amenazada de ser borrada del mapa. De *al-Khwārizmī* también proviene la palabra “álgebra”, dominio descubierto por vez primera en el actual invadido y devastado Irak.

Para la consecución de un fin se emplean “medios”. En el caso de un plato, los medios que se utilizan son los instrumentos de cocina e ingredientes; el cocinero, quien puede interpretarse también como un medio, conjuga, en un orden específico, los ingredientes mediante los instrumentos. La secuencia de ejecución, o sea, el algoritmo, es fundamental para conseguir el plato en cuestión. Una alteración del orden acarrearía posiblemente una alteración sobre el sabor del plato.

Durante la preparación de un plato, el cocinero requiere percibir y recordar la secuencia de ejecución. Él requiere, por ejemplo, sofreír algunos aliños antes de mezclarlos con la carne. Según la experiencia y la complejidad, es posible que el cocinero lleve notas que memoricen el estado de preparación; por ejemplo, anotar la hora en que comenzó a hornear. En todo momento, con notas o sin ellas, el cocinero requiere tener conciencia del estado en el cual se ubica la preparación respecto a su receta. Para eso se sirve de su memoria.

En el caso de un programa, el computador funge de cocinero, el cual ejecuta fielmente las recetas que se le proporcionan. El computador es entonces un ejecutor que organiza y usa algunos medios para alcanzar la solución de algún problema, según alguna receta llamada “programa” y que recuerda estados de cálculos mediante una “memoria”.

Aparte del CPU, quien funge de cocinero, el computador se vale de un medio fundamental: la memoria. De por sí, la memoria en bruto es una secuencia de ceros y unos cuyo sentido lo imparte el programador en forma de “datos”.

Cualquier programa que opere en un computador puede dividirse en dos partes: la secuencia de instrucciones de ejecución y los datos. Las instrucciones son resultado de aquello que escribimos como código fuente. En ocasiones, las instrucciones pueden generarse durante la ejecución del programa. Los datos representan el estado de cálculo en

algun momento del tiempo de ejecución.

La palabra “dato” proviene del latín “*datum*”, participio pasado de “dō” (dar). Dato connota, pues, algo que fue dado en el pasado y que nos interesa recordar; ese es el sentido de que sea memorizado. En el caso de la programación, un dato recuerda una parte del estado de ejecución del programa.

El mínimo nivel de organización de un dato es su “tipo” o “clase”. Un tipo de dato define un conjunto compuesto por todos los valores que puede adquirir una instancia del dato.

Un tipo de dato puede conformarse por varios tipos de datos. En este caso lo tipificamos de “dato estructurado” o “estructura de datos”. En algunos casos, los datos pueden ser recursivos (recurrentes¹); es decir, según el tipo de dato se recurren a sí mismos o entre ellos.

Algunos ejemplos en C++ podrán dar luz de este asunto.

Para representar números enteros se utiliza el tipo de dato int, el cual indica que su valor pertenece al conjunto \mathbb{Z} . En este caso no se dice nada acerca de cómo se representa el tipo int en la memoria de un computador; bien pudiera tratarse de 19, de 937 o de 2535301200456458802993406410049, entre infinitos valores posibles.

Para representar números reales que pertenecen a \mathbb{R} , usamos el tipo float, más interesante que el anterior porque trasluce parte de su implantación en la memoria de un computador cuando expresa, mediante el nombre float, que la representación del número es en punto flotante, es decir, está estructurada en tres campos de forma similar a la siguiente:

0	00001000	001011110010110001001010
Signo	Exponente	Parte fraccional

El sentido de esta estructura es hacer rápidamente sumas mediante ajuste del exponente y de la parte fraccional. Aunque no conocemos el tamaño de los campos anteriores, el conocimiento de la estructura y de la manera de manipularla nos alerta sobre el célebre e inevitable error de redondeo que ocurre cuando trabajamos con aritmética flotante.

Para ilustrar un dato recurrente nos valdremos del tipo *(Elemento de secuencia 2)*, cuya especificación en C++ puede plantearse como sigue:

2 *(Elemento de secuencia 2)*≡

```
struct Elemento
{
    int dato;
    Elemento * siguiente_elemento;
};
```

(Elemento de secuencia 2) modeliza un elemento entero perteneciente a una secuencia. Notemos que el atributo siguiente_elemento se refiere a un struct Elemento e indica la dirección en memoria del siguiente elemento en la secuencia.

Diversos intereses inciden en el diseño de una estructura de datos. Entre los más típicos podemos destacar: la comprensión y manipulación del programa por parte del programador, el desempeño y la adaptación a un algoritmo o concepto particular. En el ejemplo del tipo float hay dos características decisivas. La primera es que las operaciones

¹Según su raíz latina, el término “recurrir” *recurrō*, connota “volver”, “regresar”. En inglés, la raíz es la misma, pero basada en *recursūs*, que significa “vuelta”, “retorno”. En este texto se da preferencia a recursión en lugar de recurrencia.

aritméticas son muy rápidas. De hecho, siempre toman tiempo constante e independiente del valor particular del dato. La segunda característica concierne al espacio, es decir, cada dato en punto flotante siempre ocupa la misma cantidad de espacio, cuestión que no sucedería si usáramos aritmética arbitraria.

Para los tipos de datos que acabamos de exemplificar (`int` y `float`) disponemos de un fondo cultural, matemático y de programación que nos permite señalarlos y comprenderlos sin necesidad de detallar minuciosamente en qué consisten. Sabemos, sin tener que indicarlo explícitamente, que existen las operaciones aritméticas tradicionales de suma, resta, producto y división, así como qué hacen y cuáles son sus resultados.

1.1 Especificaciones de datos

Supongamos que un cocinero consumado desea escribir una de sus recetas para divulgarla entre otros. En esta situación, según el corpus cognitivo de su experiencia, el cocinero asume que sus lectores poseen un lenguaje común que les facilitará entender las instrucciones de su receta. Por ejemplo, se requiere que el cocinero y sus lectores tengan el mismo concepto de lo que es una olla.

En el marco de ese lenguaje común, la receta debe ser precisa; no debe contener ambigüedades que bloquee al ejecutante. Además, debe ser completa para que el ejecutante prepare plenamente el plato en cuestión.

En la percepción del aprendiz de programación existe una diferencia esencial entre elaborar un plato de cocina y ejecutar un programa. En cocina se opera sobre cosas concretas para la percepción humana. En programación, el computador opera sobre datos concretos en su memoria, pero abstractos para nuestra percepción. Preguntémosnos: ¿existe un número?, ¿existe un arreglo? En nuestra mente, un arreglo constituye una abstracción que no se capta con nuestra percepción sensorial. En el computador no tiene sentido la abstracción arreglo, pues éste no entiende lo que es un número o arreglo. En el caso de la programación, así como en otros dominios derivados de la matemática, un “número” o un “arreglo” son conceptos abstractos que conforman un lenguaje común para comunicarlos y permitir la construcción de más conceptos.

Entre programadores, así como en el resto de las prácticas, es muy importante disponer de un corpus común, sobre la manera de abstraer, que permita la comunicación de manera homogénea.

Consideremos una situación en la que deseemos disponer de un nuevo tipo de dato. Planteémosnos dos clases de preguntas en el siguiente orden:

1. P1: ¿Cuál es su fin? o, dicho de otra manera, ¿para qué puede servir?
2. P2: ¿Cómo se puede definir?, ¿qué representa el dato?

La primera pregunta nos indica la clase de problema para el cual el tipo de dato se circumscribe como parte de la solución. Si esto no está definido, entonces no tiene ningún sentido considerar el tipo de dato. La segunda pregunta nos expresa qué es el tipo de dato, pero ese qué-es depende del para qué éste se usa. Si tenemos claro el fin, entonces un dato se define según las operaciones permisibles y sus resultados.

Por ejemplo, si nos encontramos en una situación en la cual requerimos cálculo de variable compleja, entonces es esencial tener un tipo de dato “número complejo”. LLámese

$\langle \text{Complejo } 5 \rangle$ a este tipo y definímoslo como una suma $c_r + c_i i \mid c_r, c_i \in \mathbb{R}$, donde el coeficiente c_r es llamado “parte real” y, c_i “parte imaginaria”; este último representa una fracción del número “imaginado” $\sqrt{-1}$. Al igual que con los tipos anteriores, esta definición asume que el lector cuenta con una cultura matemática en la cual tienen sentido los números complejos.

Como operaciones establecemos la consulta de la parte real e imaginaria respectivamente.

1.1.1 Tipo abstracto de dato

Hemos dicho que un tipo de dato representa un conjunto. Bajo la presunción de una base cultural matemática, la cual comprende al concepto de conjunto, el tipo $\langle \text{Complejo } 5 \rangle$ se define en torno a la noción matemática de número complejo que le brinda su comprensión. Bajo ese lenguaje, el ejemplo anterior satisface las preguntas P1 y P2 respectivamente. Si no dispusiéramos del corpus matemático de número complejo, entonces nos sería muy difícil interpretar el sentido del tipo $\langle \text{Complejo } 5 \rangle$.

La matemática, siempre y cuando se haya pasado por su entrenamiento, define un corpus cognitivo, bastante abstracto por cierto, que nos permite definir el nuevo tipo de dato. Si nos remitimos a la definición de tipo de dato, entonces, según la matemática, definir un tipo de dato estriba en definir un conjunto de las dos maneras que tiene la matemática: por extensión o por comprensión. Definir un conjunto por extensión es muy objetivo, pero también muy arduo y, en muchos casos, imposible cuando el conjunto es infinito, por ejemplo. En el caso de la programación, un tipo de dato se define, paradójicamente, por comprensión mediante una forma metodológica denominada “tipo abstracto de dato” o “TAD”, la cual, en la versión de este texto, consta de las siguientes partes:

1. Una descripción del fin para el cual se destina el tipo de dato.
2. Un conjunto de axiomas y precondiciones que definen el dominio del tipo².
3. Una interfaz definida por todas las operaciones posibles sobre el TAD en la cual, por cada operación, se establezcan dos tipos de especificaciones:

Especificación sintáctica: nombre de la operación, tipo de resultado y nombres y tipos de los parámetros.

Especificación semántica: descripción de lo que hace la operación sobre el estado del TAD.

En esta parte puede ser útil indicar axiomas, precondiciones y postcondiciones.

Esencial destacar que, conocido, entendido y aceptado el fin, adquieren completo sentido las especificaciones sintáctica y semántica de un TAD.

En este texto nos valdremos del concepto de clase de objeto para llevar a cabo parte de la especificación.

²Dominio en el sentido de una función matemática.

1.1.2 Noción de clase de objeto

La palabra “objeto” proviene del latín *objectum* (*ob-jectum* u *ob-iectum*), composición de *ob*, que significa sobre el (la), frente a, y *jectum*, que es el participio pasado de *iacēre*, éntimo directo de yacer y que significa tender, echar. Así pues, *objectum* (*objecktum*) es lo que yace al frente, lo que es visible de una cosa, su cara externa. En similar contraste, la palabra “sujeto”, también proveniente del latín *subjectum*, cuyo prefijo latino *sub* señala debajo, significaba lo que está debajo de la cosa, oculto, que le es interno; dicho de otro modo, invisible en apariencia³.

En programación, así como en otras ingenierías, lo objetivo, o sea, la cara visible, se denomina “interfaz”; de inter-faz; es decir, lo que está entre la cara; lo que se ofrece al exterior.

En el contexto de la programación, una “clase de objeto”, o simplemente “clase”, es una representación objetiva de un tipo abstracto de dato que define su especificación sintáctica. Por objetiva pretendemos decir que sólo nos referimos a las partes externas, visibles, que tendría un objeto perteneciente a una clase dada. En el caso de un tipo de dato, las “partes visibles” las conforman el nombre del tipo, los nombres de las operaciones, los nombres de los parámetros, los tipos de dato de los parámetros y los resultados de las operaciones, es decir, su especificación sintáctica.

Para satisfacer la objetividad de la especificación sintáctica es necesario acordar un lenguaje común entre los programadores, pues de lo contrario sería muy difícil interpretar la interfaz. Un automóvil, por ejemplo, tiene una interfaz de uso consistente, entre otras cosas, de los pedales, el volante y el tablero. Para poder conducirlo se requiere que el conductor esté entrenado en la utilización de esa interfaz. Del mismo modo, para que un programador comprenda una especificación sintáctica, éste debe entender el lenguaje en que se especifica la clase o TAD. En el caso de este texto haremos especificaciones sintácticas de TAD en el lenguaje C++ o en diagramas de clases UML.

En C++, el ejemplo del TAD *(Complejo 5)* podría modelizarse del siguiente modo:

5 *(Complejo 5)≡*

```
struct Complejo
{
    Complejo(float r, float i);
    Complejo(const Complejo & c);
    float & obtenga_parte_real();
    float & obtenga_parte_imag();
};
```

Esta definición establece “objetivamente” la especificación sintáctica del TAD *(Complejo 5)*, la cual, aunada al lenguaje y al corpus matemático cultural de la noción de número complejo, completa la especificación sintáctica del TAD.

¿Qué sucedió con la especificación semántica?, ¿está completa la especificación? Si asumimos que el lector de la especificación del TAD *(Complejo 5)* conoce su matemática inherente, entonces los nombres de las operaciones permiten comprender directamente qué hace cada operación sin necesidad de explicitarlo. ¿Existe alguna duda sobre lo que hace la operación *obtenga_parte_real()*? La respuesta depende, entre otros factores, del grado de entendimiento matemático que tenga el cuestionante. Si el lector no conoce la noción

³Estas aclaratorias etimológicas fueron descubiertas en [56].

de número complejo, entonces se requerirá una especificación semántica que le imparta lo que es un complejo.

Lo objetivo posibilita un acuerdo común entre diferentes personas acerca de la interpretación de un tipo de dato. Para que este acuerdo ocurra, es necesario que las personas en cuestión “vean” o interpreten homogéneamente al objeto. Consecuentemente, la interpretación de un TAD depende del grado en que sus interesados comparten el lenguaje con que se exprese su especificación.

1.1.3 Lo subjetivo de un objeto

Si tratamos a un dato en términos objetivos, entonces, ¿en qué consiste tratarlo en términos subjetivos? *Grosso modo*, la respuesta es que la subjetividad de un TAD se trata durante su especificación semántica.

Existen, básicamente, tres fuentes de subjetividad.

La visión, interpretación y, en consecuencia, el sentido de un TAD, dependen de la experiencia y conocimiento que tenga la persona que utilice el TAD. Pero esto es muy subjetivo, pues cada quien tiene su propia experiencia, la cual no debe tratarse objetivamente. La primera fuente de subjetividad es entonces la interpretación del usuario del TAD acerca de su sentido. Quienes hayan estudiado cabalmente los números complejos tendrán una compresión del TAD (*Complejo 5*) más homogénea que quienes no lo hayan estudiado. Para estos últimos puede ser necesario complementar la especificación.

En el caso del TAD (*Complejo 5*) es muy conveniente tener una trayectoria de estudios matemáticos. ¿Puede un programador sin esta trayectoria manejar el TAD (*Complejo 5*)? Enfrentar esta pregunta revela perspectivas contradictorias.

En primer lugar, si el usuario del TAD (*Complejo 5*) acepta la interfaz, entonces, el desarrollo de programas que usen números complejos permite ganar comprensión acerca de la matemática compleja. Empero, este usuario, al no disponer del corpus matemático requerido, es más propenso a utilizar la interfaz para un fin diferente al que fue concebido el TAD (*Complejo 5*); por ejemplo, para representar puntos en el plano cartesiano. Si bien esto puede representar un ahorro de código, puede acarrear también grandes confusiones entre los programadores y mantenedores.

La segunda fuente de subjetividad proviene del mismo diseñador del TAD. El objeto resultante depende también de la experiencia del diseñador. Personas diferentes tienden a proponer interfaces diferentes.

La última fuente de subjetividad se refiere a la implantación del TAD. Distintos programadores harán implantaciones diferentes. Si bien esta es primariamente la subjetividad que pretende esconder un TAD, puede ser esencial considerarla por dos razones: el tiempo de implantación y el tiempo de ejecución.

El tiempo de desarrollo de un TAD puede ser tan extenso que comprometa un proyecto. Análogamente, el tiempo de ejecución del programa resultante puede ser tan lento que haga necesario repetir la implantación del TAD. En cualquiera de estas situaciones puede ser conveniente indicar los aspectos generales de la implantación.

En resumen, las fuentes de subjetividad se tipifican como sigue:

1. Subjetividad de interpretación del usuario
2. Subjetividad de interpretación del diseñador

3. Subjetividad de implantación

Por más énfasis que se le haga a la orientación a objetos, los programadores que se circunscriban en desarrollos cooperativos deben estar conscientes de estas subjetividades al momento de hacer la especificación semántica de un TAD. La idea en la especificación semántica es entonces adecuarse a la expectativa cognitiva del grupo de personas involucradas en el desarrollo y uso de un TAD. Si aquel grupo, por instancia, comprende la matemática compleja, entonces no sólo es innecesario ahondar en una especificación semántica que explique la noción de numero complejo, sino que también puede tornarse muy tedioso. En este caso, la fuente de subjetividad sólo es de implantación, la cual es precisamente la subjetividad que pretende ocultar un TAD.

Por la razón anterior, la especificación semántica no es objetiva. Ahora bien, ¿cómo llevar a cabo una especificación semántica efectiva, es decir, que logre el efecto de aceptarse entendida por un grupo de programadores? Respuesta resumida: mediante un lenguaje adecuado. Para disertar en torno a esta cuestión, es apropiado imaginar cómo dos interlocutores tratan la noción de lo objetivo y subjetivo acerca de una cosa de programación y un TAD.

En el sentido en que lo hemos tratado, lo objetivo de la cosa programada es perceptible a la visión común de los interlocutores, mientras que lo subjetivo les está oculto, al menos a la mirada de uno de ellos, o de ambos. Por "visión", el lector no debe asumir el mero sentido sensorial, sino la capacidad de percibir una cosa o fenómeno mediante las abstracciones y construcciones intelectuales que la experiencia de los interlocutores les permita. Como parte de la experiencia, es crítico que los interlocutores tengan destrezas de programación equiparables.

Supongamos que un interlocutor A le presenta un TAD a otro B. Dos situaciones iniciales son posibles: (1) B ve e interpreta el TAD de la misma manera que A y (2) B no lo interpreta igual. En cualquiera de los dos casos, es esencial que A conozca la opinión de B para poder determinarse cuál de las dos situaciones ocurre.

Cuando ocurre la primera situación, los interlocutores tienen entonces una mirada homogénea del TAD y la subjetividad que queda es de implantación, la cual, en el estadio de diseño, casi siempre es bueno ocultarla.

Si ocurre la segunda situación, entonces A y B deben homogeneizar la visión e interpretación del TAD de forma que sus subjetividades de interpretación lleguen a ser objetivas. La única manera hasta ahora conocida de hacerlo es mediante el diálogo. Por eso el lenguaje es fundamental en la especificación de un TAD. Pero el lenguaje no es meramente unidireccional. A no tiene ninguna forma de corroborar si B comparte su mirada si no escucha la interpretación que tenga B acerca del TAD. Por tanto, cuando se diseña un nuevo TAD, es esencial que el diseñador lo exponga ante los interesados e inicie un proceso de diálogo que dure hasta que no hayan subjetividades de interpretación y sólo queden las de implantación.

1.1.4 Un ejemplo de TAD

Experiencias adquiridas en el desarrollo de programas de dibujado permiten modelizar un TAD, llamado *«Figure 8a»*, cuyo fin es generalizar operaciones inherentes al dibujado de una figura sobre algún fondo de contraste. Tal TAD es útil para desarrollar programas

que hagan dibujos, y una propuesta de definición es como sigue:

8a *<Figure 8a>*≡
 struct Figure
 {
 <Constructores de Figure 8b>
 <Observadores de Figure 9b>
 <Modificadores de Figure 9c>
 };
<Figuras concretas 12>

Defines:

Figure, used in chunks 8b, 9a, 12, and 15a.

El TAD *<Figure 8a>* modeliza una figura “general” que se dibujaría en algún medio de contraste. Es “general” en el sentido de que sólo abstraemos operaciones generales sobre una figura geométrica cualquiera, es decir, las operaciones son “generales”⁴ porque operan sobre cualquier figura independientemente de su particularidad. Por “cualquier figura” pretendemos expresar que su forma, cuadrática, triangular, etcétera, no nos importa, sólo nos interesa una figura como abstracción general para dibujar y la manera general de operar sobre ella a través de operaciones generales comunes a todas las figuras existentes.

No se debe, y es preferible asumir que no se puede, definir una abstracción sin conocer el para qué de tal definición. Por esa razón, cuando diseñamos un TAD debemos asegurarnos de tener claro el fin que perseguimos. En este sentido, en lo que concierne al TAD *<Figure 8a>*, el fin es dibujar figuras en algún medio de contraste. Si este fin no está claro, no tiene sentido hablar de figuras y de sus operaciones.

La especificación sintáctica del TAD *<Figure 8a>* está dada por su definición en C++.

Una operación sobre una clase se denomina “método”, término proveniente del latín *methōdus*, el cual proviene del griego μέθοδος (meta - hōdos). En este caso “meta” connota “fuera”, “más allá” y *hōdos* significa camino. “Método” quiere decir, pues, un camino hacia el fin [que está fuera]; es decir, un camino con destino, con sentido.

Para definir la semántica de cada operación, debemos denotar el TAD Point, pues lo referencian algunas operaciones del TAD *<Figure 8a>*. Supeditado al fin del TAD *<Figure 8a>*, un punto destina la ubicación de la figura al momento de su dibujado y no nos conviene, por ahora, definirla más, pues no tenemos idea -y es por ahora también preferible no tenerla- del medio en el cual se dibujarían las figuras; por ejemplos, un medio planar: papel o pantalla; o un medio tridimensional: proyector tridimensional u holografía. Para el primer tipo de medio el punto requiere dos coordenadas, mientras que para el segundo tres.

Hay dos formas de construir una figura abstracta expresadas por los siguientes constructores:

8b *<Constructores de Figure 8b>*≡
 Figure(const Point & point);
 Figure(const Figure & figure);
 Uses Figure 8a.

(8a) 9a▷

El primer constructor requiere un punto; el segundo copia la figura a partir del punto donde se encuentre otra figura.

⁴La redundancia es adrede.

El destructor del TAD *(Figure 8a)* debe ser virtual:

9a *(Constructores de Figure 8b)* +≡
virtual ~Figure();

(8a) ▷8b

Uses *Figure 8a*.

pues de esa manera se garantiza la invocación de cualquier destructor asociado a una figura particular.

A un método que no altere o modifique el estado del objeto suele llamársele “observador”. En este sentido, una figura tiene un solo observador:

9b *(Observadores de Figure 9b)* ≡
const Point & get_point() const;

(8a) 16▷

el cual “observa” su punto de referencia en el plano. En C++, el calificador `const` sobre un método indica al compilador que el método no altera el estado del objeto.

A un método que altera o modifica el estado de un objeto se le califica de “modificador” o, a veces “actuador”. Los actuadores de una figura son los siguientes:

9c *(Modificadores de Figure 9c)* ≡
virtual void draw() = 0;
virtual void move(const Point & point) = 0;
virtual void erase() = 0;
virtual void scale(const Ratio & ratio) = 0;
virtual void rotate(const Angle & angle) = 0;

(8a)

Los nombres de métodos `draw()`, `move()` y `erase()` indican claramente su función⁵. El método `scale()` ajusta el tamaño (escala) de una figura según un radio dado. El tipo `Ratio` especifica una magnitud de escala que significa la proporción en que la escala se modifica; si éste es menor que uno, entonces la figura se achica, de lo contrario se agranda. Finalmente, el método `rotate()` gira o rota la figura en el medio según un ángulo de tipo `Angle`.

Los *(Modificadores de Figure 9c)* representan operaciones generales sobre una figura. Podemos dibujarla, moverla hacia otro punto, borrarla, escalarla según alguna magnitud, o rotarla según algún ángulo en radianes cuyo signo indica el sentido de rotación. En todos los casos tratamos con figuras abstractas, no concretas.

Al igual que con el tipo `Point`, en este estadio no es conveniente pensar en las implantaciones de los tipos `Ratio` y `Angle`. Sólo basta con conocer su utilización con objetos de tipo `Figure` circunscrita a fin de dibujarlas.

Mención particular merecen dos calificadores sintácticos del C++. El primero lo conforma el prefijo reservado “`virtual`”, el cual indica que la operación puede implantarse según la particularidad de la figura; por ejemplo, un cuadrado se dibuja diferente que un círculo. El segundo está dado por el hecho de inicializar la operación con el valor cero. Esta es la sintaxis de C++ para definir un “método virtual puro”, el cual, a su vez, define una clase abstracta, o sea, abstracción pura, sin ningún carácter concreto, pues si no, la clase no sería abstracta. Para aprehender esta observación, comencemos por preguntarnos ¿a cuál figura se refiere el TAD *(Figure 8a)*? La respuesta correcta es que no lo sabemos, pues se trata de una clase abstracta, no de una concreta.

Los métodos virtuales puros tienen que implantarse en “clases derivadas” de la clase `Figure` que concretan implantaciones particulares de una figura abstracta. Por ejem-

⁵A condición de que se conozcan los términos ingleses.

plo, el método `scale()` de un triángulo se implanta en una clase `Triangle` derivada de `Figure`.

El concepto de derivación será estudiado con más detalle en § 1.2 (Pág. 11).

1.1.5 El lenguaje UML

Hasta ahora nos hemos servido del lenguaje C++ para resolver la especificación sintáctica. En efecto, en este lenguaje, y en otros orientados a objetos, su propia sintaxis indica todos los aspectos sintácticos de interés y, si se escogen nombres adecuados, fundamenta los semánticos.

Hoy en día existen muchos lenguajes orientados a objetos, cuya presencia nos dificulta unificar miradas en especificaciones y diseños. Para paliar este problema, desde hace más de una década, un consorcio llamado OMG (Object Management Group) intenta "homogeneizar" la manera de especificar TAD [65]. Dicho lenguaje se llama acrónimamente UML: "*Unified Modeling Language*" y se sirve de un medio que no tienen todos los lenguajes y que es cómodo con la idea de lo objetivo: el gráfico. Un gráfico dice más que mil palabras reza un antiguo proverbio, y UML lo honra cuando se requiere observar diferentes TAD y sus relaciones.

El TAD `(Figure 8a)` puede modelizarse pictóricamente en UML como en la figura 1.1. Un rectángulo representa una clase con tres secciones. El nombre de la clase se encuentra en la sección superior. El título en letra cursiva indica que la clase es abstracta.

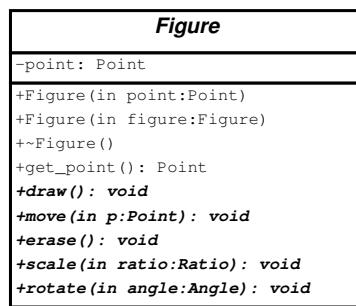


Figura 1.1: Diagrama UML de la clase `Figure`

La segunda sección indica los atributos y la tercera los métodos u operaciones. El prefijo “-” en cada nombre de atributo u operación indica que el miembro es inaccesible, mientras que el símbolo “+” indica que es completamente accesible.

Un nombre de operación en letra cursiva indica que la operación es virtual o polimorfa; concepto que estudiaremos pronto.

Los nombres de tipos de retorno y de parámetros se separan de los nombres de función y de parámetros con dos puntos (a la antigua, pero elegante, usanza del Pascal y Algol). Los parámetros tienen un prefijo calificador que pueden tener valores `in`, `out` o `inout` para especificar parámetros de entrada, salida o entrada/salida respectivamente.

Un programa complejo contiene muchos tipos abstractos. Cuando esta cantidad es grande, cualquier lenguaje resulta complicado para mirar en unidad al sistema y dentro de él observar sus tipos de datos e interrelaciones. La gran ventaja de UML es su carácter gráfico, el cual facilita observar, sólo visual y objetivamente, los tipos abstractos en una sola mirada.

En este texto usaremos UML sólo para especificar diagramas de clases e interrelaciones entre ellas. UML es un lenguaje de modelado mucho más rico y la experiencia ha demostrado que para ganar homogeneidad de interpretación en un proyecto, éste es más simple que un lenguaje de programación.

1.2 Herencia

El TAD *(Figure 8a)* modeliza una figura general que no indica nada acerca de su forma concreta. Sin embargo, al momento de dibujar una figura se tiene que concretar y conocer de cuál figura se trata. En la jerga a objetos, a “concretar” se le dice “especializar” y se representa mediante una relación llamada “herencia de clase”. En UML, concretar algunas “figuras” bajo un diagrama UML, resumido, ejemplariza el concepto de una forma que nos permite visualizar las clases y sus relaciones en una especie de “genealogía” o “taxonomía”, tal como se ilustra en la figura 1.2.

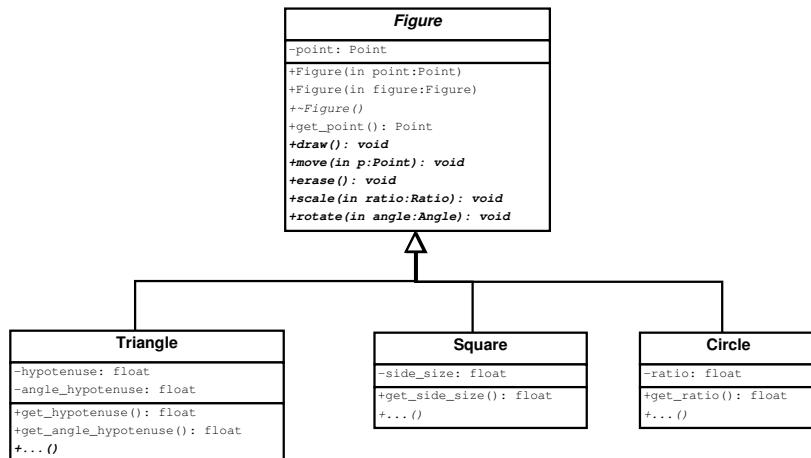


Figura 1.2: Jerarquía de clases especializadas de la clase Figure

La relación de herencia se expresa en UML mediante una flecha contigua que parte desde la clase especializada hacia la clase general. En el caso ejemplo, la clase general la conforma el TAD *(Figure 8a)*, mientras que las clases especializadas concretan las figuras específicas **Triangle**, **Square** y **Circle**, especializaciones de triángulo, cuadrado y círculo respectivamente.

La herencia se define entonces como la propiedad que tiene una clase de “heredar” el ser de otra clase .

A la clase general se le llama “clase base” o “fundamental”, mientras que la especializada recibe el nombre de “clase derivada”. En el ejemplo de las figuras, el TAD *(Figure 8a)* es clase base de las clases derivadas **Triangle**, **Square** y **Circle**.

Decimos también que la clase derivada “hereda” de la clase base en el sentido de que hereda toda su interfaz pública. Un triángulo, por instancia, hereda el punto de referencia atribuible a todas las figuras generales, es decir, un objeto de tipo **Triangle** puede invocar al método `get_point()`, pues éste fue heredado del TAD *(Figure 8a)*.

En el diagrama UML de la figura 1.2 se aprecia que las clases derivadas poseen atributos y métodos que no se encuentran en la clase base. Por ejemplo, la clase **Circle** posee un método llamado `get_ratio()` cuya función es observar el radio de la circunferencia que

representa una instancia de objeto de tipo Circle. El atributo “ratio” tiene sentido, según la geometría, para la clase Circle, pero no lo tiene para las clases Triangle y Square. Ahora bien, las tres clases derivadas de *Figure 8a* comparten el punto de referencia, pues son, por derivación, de tipo *Figure 8a*.

Lo anterior sugiere una interpretación de la herencia quizá más rica: la relación “ser”, es decir, la clase derivada, es también de clase base. De este modo, objetos de tipo Triangle, Square y Circle son, también, de tipo Figure.

La declaración en C++ de la relación de herencia anterior es la siguiente:

12 *Figuras concretas 12* ≡ (8a)

```
struct Triangle : virtual public Figure { ... };
struct Square : public Figure { ... };
struct Circle : public Figure { ... };
```

Uses Figure 8a.

La pseudoespecificación de la clase `Triangle` indica que es una clase abstracta. En efecto, notemos que, en el caso de un triángulo, según nuestra cultura matemática, podemos especializarlo aún más según las longitudes de sus lados.

Lo grandioso de la herencia es la posibilidad de expresar conceptos y abstracciones generales a través de clases bases para luego particularizarlas mediante derivaciones. Esto ofrece la posibilidad de diseñar inductivamente, yendo desde lo particular hacia lo general, o deductivamente, yendo desde lo general hacia lo particular. Dicho de otro modo, yendo desde lo concreto hacia lo abstracto o, paradójicamente, desde lo abstracto hacia lo concreto.

1.2.1 Tipos de herencia

La herencia del ejemplo anterior se denomina “herencia pública”, pues lo que es público de la clase base también llega a ser público en la clase derivada.

Hay otro modo de herencia denominado “de implantación” o “herencia privada”. Este modo expresa el hecho de que la clase base se usa para implantar la, o parte de la, clase derivada. En UML, la herencia privada se representa mediante una flecha punteada, mientras que en C++ se hace mediante el calificador `private` como prefijo al nombre de la clase base.

1.2.2 Multiherencia

En ocasiones, una clase de objeto es, a la vez, de dos o más clases. En el mundo a objetos, esto puede expresarse mediante una relación de herencia múltiple. Es decir, un objeto puede heredar de dos o más clases base. La figura 1.3 muestra una relación de clases que modeliza los actores de una universidad. Atención especial merece la clase Preparador. En la vida real, un preparador es un estudiante excelsior, cuya excelencia aprecia la universidad para la asistencia y mejora de sus cursos. Como retribución, la universidad le otorga al estudiante un estipendio. En los términos del diagrama UML, Preparador hereda de las clases Estudiante y Trabajador Universitario respectivamente. De este modo definimos que un preparador es, a la vez, estudiante y trabajador universitario.

Es posible tener relaciones de multiherencia de interfaz por una parte, y de implementación por alguna otra.

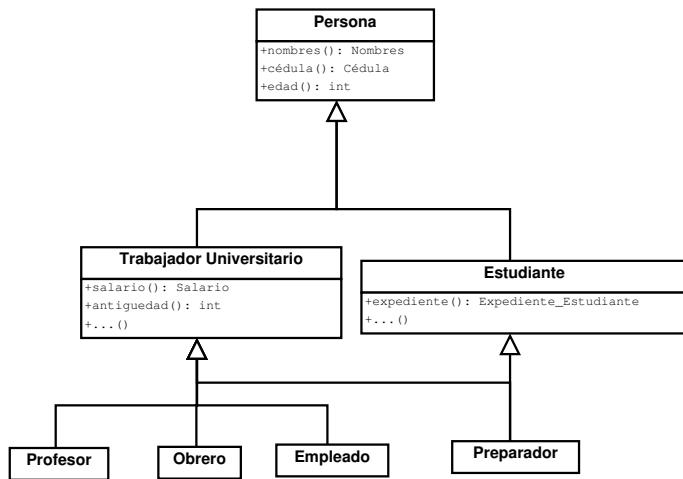


Figura 1.3: Relaciones de clases de personas en una universidad

1.2.3 Polimorfismo

La palabra “polimorfo” proviene del griego πόλυ (“poly”), que significa “mucho”, mientras que “morfo” proviene de μορφή (“morfé”), que significa “figura”, “forma”. Polimorfo connota, pues, “muchas figuras”, “muchas formas”. En la jerga a objetos, polimorfismo es la propiedad de expresar varias funciones o procedimientos diferentes o similares bajo el mismo nombre.

Hay tres clases de polimorfismo: de sobrecarga, de herencia y de plantilla.

1.2.3.1 Polimorfismo de sobrecarga

Sobrecarga es la capacidad de un lenguaje a objetos para definir nombres iguales de funciones, procedimientos y métodos. Consideremos, por ejemplo, la función siguiente:

```
const int sumar(const int & x, const int & y);
```

cuyo fin es sumar dos enteros. `sumar()` puede “sobrecargarse” para que sume más enteros:

```
int sumar(const int & x, const int & y, const int & z);
int sumar(const int & w, const int & x, const int & y, const int & z);
```

El compilador, a través de la cantidad de parámetros, hace la distinción y decide cuál de las tres funciones es la que se debe invocar. Por ejemplo, si el compilador encuentra:

```
sumar(1, 2, 3);
```

Entonces éste generará la llamada a `sumar()` con tres parámetros.

Es posible también definir `sumar()` para que “sume” otra clase de objetos. Por ejemplo:

```
const string sumar(const string & x, const string & y);
```

La suma de cadenas puede interpretarse como la concatenación. El compilador hace la distinción respecto a las versiones aritméticas mediante los tipos de los parámetros.

En C++ se pueden sobrecargar los operadores. Por ejemplo, podríamos especificar la concatenación de cadenas del siguiente modo:

```
const string operator + (const string & x, const string & y);
```

En este caso, puede escribirse algo así como:

```
string s1, s2;
```

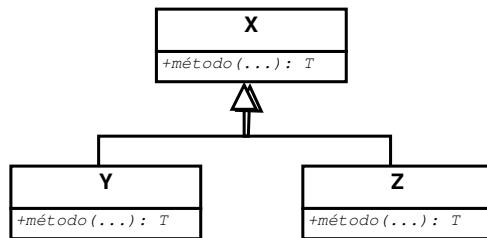
```
...
```

```
string s3 = s1 + s2;
```

La sobrecarga de operadores e, inclusive, la de funciones, es un asunto polémico porque oculta operaciones y puede contravenir el sentido cultural del operador. Por ejemplo, el código anterior tiene perfecto sentido cultural para la aritmética, pero quizás no para la concatenación. Cuando en una primera inspección un lector lea la suma de cadenas, posiblemente él pensará que los operandos son numéricos, lo cual, en el último ejemplo, no es el caso. Por esa razón es por lo general recomendable evitar la sobrecarga.

1.2.3.2 Polimorfismo de herencia

Dada una relación de herencia entre tres clases X, Y y Z, expresada gráficamente de la siguiente manera:



El polimorfismo de herencia se define como la posibilidad de definir (no de sobrecargar) métodos virtuales del mismo nombre en las tres clases, invocar al método desde la clase X y determinar, en tiempo de ejecución, cuál es el método concreto según sea la implantación real de la clase X.

A los métodos Y::método() y Z::método() se les connota como “especializaciones” del método en la clase base X::método().

Podría decirse que el polimorfismo de herencia es la sobrecarga de métodos virtuales entre las clases. Es importante resaltar que, en este caso, los prototipos de los métodos virtuales tienen que ser idénticos.

Recordemos que el TAD *<Figure 8a>* contiene métodos virtuales, puros, que no se pueden implantar. Si pretendiésemos dibujar un objeto de tipo *<Figure 8a>* se nos aparecerá la pregunta: ¿Cuál figura?, pues el TAD *<Figure 8a>* es una figura abstracta, no concreta. Es cuando conocemos cuál es la figura que tiene sentido indagar cómo dibujarla.

La implantación de un método virtual puro se le delega a una clase derivada. Para el caso del TAD *<Figure 8a>*, sus *<Figuras concretas 12>* deben implantar sus métodos virtuales puros. Por ejemplo, los métodos Square::draw() y Circle::draw() implantan de manera diferente el dibujado de su correspondiente figura. De este modo, cuando se opera sobre un objeto general de tipo *<Figure 8a>* y se invoca a un método virtual, se selecciona, en tiempo de ejecución, el método de especialización según la figura concreta sobre la cual se esté operando.

La virtud del polimorfismo de herencia es que permite escribir programas generales que manipulen “figuras” generales⁶. Estos programas no requieren conocer las figuras concretas, pues operan en función de métodos virtuales generales puros. Por ejemplo, partes de programas para dibujar figuras manipulan figuras en abstracto, las dibujan, las mueven, las rotan, etcétera. A efectos de economizar código resulta útil la posibilidad de escribir programas generales como el del ejemplo siguiente:

15a

Manejo general de Figure 15a≡

```
void release_left_button(const Action_Mode mode, Figure & fig)
{
    switch (mode)
    {
        case Draw:
        case Move:
            fig.draw(); break;
        case Delete: fig.erase(); break;
        case Scale: fig.scale(dif_mag_with_previous_click()); break;
        case Rotate: fig.rotate(dif_angle_with_previous_click()); break;
        ...
    }
}
```

Uses Figure 8a.

la cual sería invocada en caso de que se detectase que se suelta el botón izquierdo del ratón dentro de un “lienzo” abstracto de dibujado.

`release_left_button()` no requiere conocer la figura concreta. Su código es general y no se afecta por las modificaciones o añadiduras de las figuras. Por ejemplo, si `release_left_button()` opera sobre un cuadrado, entonces se invocarán a los métodos virtuales “concretos” de la clase `Square`; análogamente, ocurre con un círculo, caso en el cual se invocarán los métodos de `Circle`.

1.2.3.3 Polimorfismo de plantilla (tipos parametrizados)

Hay situaciones en las cuales un problema y su solución pueden especificarse de forma independiente del (o los) tipo(s) de dato(s). Por ejemplo, el problema de buscar un elemento en un conjunto, y su solución, son independientes del tipo de elementos. Si suponemos que el conjunto se representa mediante un arreglo, entonces, una posible manera, genérica, de buscar un elemento, es como sigue:

15b

Búsqueda dentro de un arreglo 15b≡

```
template <typename T, class Compare>
int sequential_search(T * a, const T& x, int l, int r)
{
    for (int i = l; i <= r; i++)
        if (are_equal<T, Compare> () (a[i], x))
            return i;
    return No_Index;
}
```

Uses `sequential_search` 154a.

⁶De nuevo, la redundancia es adrede.

Esta rutina busca el elemento *x* dentro del rango comprendido entre *l* y *r* del arreglo *a*. Se retorna un índice dentro del arreglo correspondiente a una entrada que contiene un elemento igual a *x*, o el valor *No_Index* (por lo general *-1*), si el arreglo no contiene *x*.

A parte de los parámetros pertinentes al conjunto, el algoritmo genérico *sequential_search*<*T*, *Compare*>() requiere dos tipos parametrizados: el tipo de dato del conjunto, llamado genéricamente *T*, y un tipo comparador de igualdad llamado *are_equals*<*T*, *Compare*>(), cuyo uso será abordado en § 1.3.1 (Pág. 19).

Funciones o métodos como *sequential_search*<*T*, *Compare*>() se llaman “plantillas”⁷. Decimos que *sequential_search*<*T*, *Compare*>() es “genérica” porque genera una familia de funciones para cada tipo existente en el cual exista una clase *are_equals*(). En otras palabras, una plantilla automatiza la sobrecarga de la función o clase para los tipos involucrados en la plantilla.

Observemos que aunque la plantilla es la misma, o sea, es genérica, el código genérico *sequential_search*<*int*, *are_equals*<*int*>>(...) es diferente al algoritmo *sequential_search*<*string*, *are_equals*<*string*>>(...). El compilador debe generar dos códigos distintos; uno para arreglos de enteros (*int*) y otro para arreglos de cadenas (*string*).

Cuando se exemplificó la clase *(Figure 8a)* se indicó que su fin es dibujarla en un medio de contraste. ¿De cuál medio se habla: papel, pantalla de video, televisor, holografía ...? El lector acucioso debe haberse percatado de que las implantaciones de la clase *(Figure 8a)* (*Square*, *Triangle* y *Circle*) tienen que asumir un medio en donde efectuar las operaciones. Una manera de independizarse del medio de contraste es hacer a la clase *(Figure 8a)* una plantilla cuyo parámetro sea, justamente, el medio de contraste. La idea se ilustra en el diagrama UML de la figura 1.4.

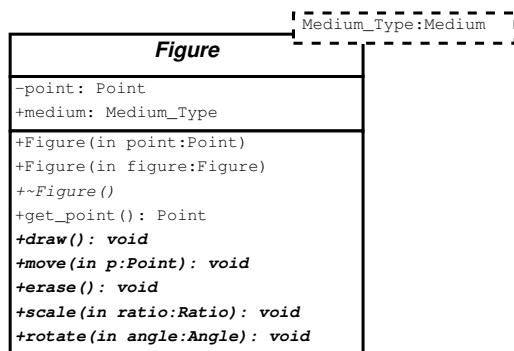


Figura 1.4: Diagrama UML de la clase *Figure* con el medio de contraste como parámetro

En UML, los parámetros plantilla de la clase se especifican en un rectángulo punteado situado en la esquina superior derecha.

Un objeto de tipo *(Figure 8a)* posee como tipo parametrizado el medio en donde se manipulan las figuras. Puede ser necesario que las especializaciones de *(Figure 8a)* conozcan el tipo concreto del medio de contraste. Por esta razón, la versión plantilla de *(Figure 8a)* exporta el tipo parametrizado bajo el nombre *Medium_Type*. En C++ esta acción se lleva a cabo mediante la siguiente declaración:

⁷En inglés, “template”.

```
typedef Medium Medium_Type;
```

De esta manera, una especialización puede instanciar un objeto de tipo `Medium_Type` como se ilustra en el siguiente ejemplo:

```
void Square::draw()
{
    /* .... */
    Medium_Type m; // Instancia un objeto "medio de contraste"

    /* .... */

    medium.put_line(...);
}
```

El atributo tipo `medium` de la clase `Figure<Medium>` le permite acceso a éste. La especialización `Square::draw()` dibujaría líneas en el medio de contraste correspondientes al respectivo cuadrado. La implantación de `Square::draw()` deviene genérica respecto al medio.

1.2.3.4 Lo general y lo genérico

Los términos “general” y “genérico” no sólo se parecen mucho léxicamente, sino que, en efecto, también son muy similares semántica y etimológicamente.

La raíz de ambos términos es el verbo latino *gēnērō*, que significa engendrar, crear. En esta época, tanto “general” como “genérico” connotan lo que es común a una especie. *Gēnērō* proviene a la vez del griego γένος (*genus*), que en el lenguaje moderno connota “raza” y que en griego se refería a lo común. De “genus” proviene una muy amplia variedad de términos: género, gen, genética, gentilicio, generoso, gente, genealogía, genio, genial, ingenio, ingeniería, etcétera.

En su celebrísima y trascendental *Metafísica*, Aristóteles distingue el “género” como lo que le es esencialmente común a una especie. Podemos decir, pues, que la jerga a objetos está, desde hace más de 2500 años, impregnada por esta idea.

En el caso de la programación a objetos, “general” identifica clases de objetos generales; es decir, bases de otras clases más particulares, individuales, o clases que operan sobre la generalidad. Por ejemplo, la clase `<Figure 8a>` es general a todas las figuras, mientras que la clase `are_equal()` representa la comparación general y genérica entre objetos.

“Genérico” connota lo que genera, o sea, en la orientación a objetos, a las plantillas, concepto que recién acabamos de presentar y exemplificar.

1.3 El problema fundamental de estructuras de datos

Existe una clase de problema cuya ocurrencia es tan ubicua en prácticamente todos los ámbitos de la programación, que ya es posible generizarla en una sola clase. Se trata del conjunto. Puesto que en la mayoría de los casos, los elementos son del mismo tipo, es posible objetizarlo en un TAD genérico tal como lo ilustra el diagrama UML de la figura 1.5. El diagrama en cuestión modeliza lo que se conoce como el “problema fundamental de estructuras de datos”. Las diferentes maneras de implantarlo y sus diversas circunstancias de aplicación, abarcan casi todo el corpus de este texto.

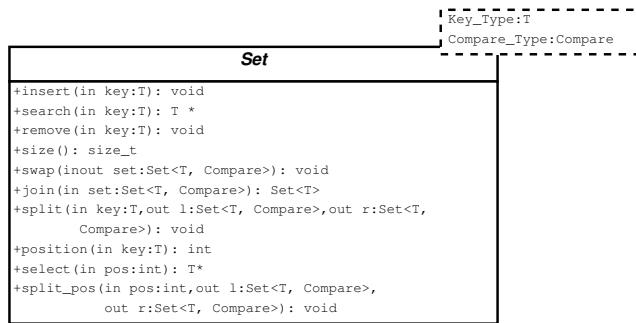


Figura 1.5: Diagrama UML de una clase genérica conjunto (Set)

La clase `Set<T, Compare>` modeliza un conjunto genérico de datos de tipo `T`, con criterio de comparación `Compare`, cuyo fin es generalizar operaciones sobre conjuntos sin que nos interese cómo éstos se implantan.

Hay muchas formas de implantar el tipo `Set<T, Compare>`. La decisión depende de sus circunstancias de uso.

Conjuntos de datos que se correspondan con el problema fundamental se denominan “contenedores”. Un contenedor es, pues, un conjunto de elementos del mismo tipo.

`Set<T, Compare>` puede modelizarse en C++ como sigue:

18

(Conjunto fundamental 18)≡

```

template <typename T, class Compare>
struct Set
{
    void insert(const T & key);
    T * search(const T & key);
    void remove(const T & key);
    size_t size() const;
    void swap(Set<T, Compare> & set);
    void join(Set<T, Compare> * set);
    void split(const T& key, Set<T, Compare> *& l, Set<T> *& r);
    const int position(const T& key) const;
    T * select(const int pos);
    void split_pos(const int & pos,
                  Set<T, Compare> *& l, Set<T, Compare> *& r);
};

```

En líneas generales, el problema fundamental se define como el mantenimiento de un conjunto `Set<T, Compare>` de elementos de tipo `T` con las operaciones básicas de inserción, búsqueda, supresión y cardinalidad y cuyas interfaces fundamentales son `insert()`, `search()`, `remove()` y `size()` respectivamente.

Frecuentemente, los elementos del conjunto se llaman “claves”. De allí el nombre de parámetro `key` en muchas de las interfaces de `Set<T, Compare>`.

`swap(set)` intercambia todos los elementos de `set` con los de `this`. Según la estructura de datos con que se implante `Set<T, Compare>`, algunas veces esta operación será muy rápida.

El método `join(set)` une `this` con el conjunto referenciado por el parámetro `set`. Después de la operación, `set` deviene vacío.

`join()` puede tener variaciones según el tipo de conjunto y la estructura de datos. Las más comunes son la concatenación y la intercepción.

1.3.1 Comparación general entre claves

Muchas veces, el tipo genérico `T` es ordenable, es decir, las claves pueden disponerse en una secuencia ordenada desde la menor hasta la mayor o viceversa. En esos casos aparecen el resto de las operaciones y la clase de comparación `Compare`.

`Compare` es una clase que implanta la comparación entre dos elementos de tipo `T`. Por lo general, `Compare` implanta el operador relacional `<`. Con una clase de este tipo es posible realizar el resto de los operadores relacionales. Por ejemplo, si tenemos dos claves `k1` y `k2` y una clase `Compare<T>` cuyo operador `()` implanta `k1 < k2`, entonces el siguiente pseudocódigo ejemplifica todas las comparaciones posibles:

```
if (Compare() (k1, k2)) // ¿k1 < k2?
    // acción a ejecutar si k1 < k2
else if (Compare() (k2, k1)) // ¿k2 < k1?
    // acción a ejecutar si k2 < k1
else // Tienen que ser iguales
    // acción a ejecutar si k1 == k2
```

1.3.2 Operaciones para conjuntos ordenables

Si el conjunto es ordenable, entonces éste puede interpretarse como una secuencia ordenada $S = \langle k_0, k_1, \dots, k_{n-1} \rangle$, en la cual n es la cardinalidad del conjunto. En ese caso, pueden hacerse varias operaciones sobre la secuencia S .

El método `split(key, l, r)` “particiona” el conjunto en dos subconjuntos $l = \langle k_0, k_1, \dots, k_i \rangle$ y $r = \langle k_{i+1}, k_{i+2}, \dots, k_{n-1} \rangle$ según la clave `key` tal que $l < key < r$. Es decir, `l` contiene las claves menores que `key` y `r` las mayores. Después de la operación, `this` deviene vacío.

El método `position(key)` retorna la posición de la clave dentro de lo que sería la secuencia ordenada. Si `key` no se encuentra en el conjunto, entonces se retorna un valor inválido.

El método `select(pos)` retorna el elemento situado en la posición `pos` según el orden. Si `pos` es mayor o igual a la cardinalidad, entonces se genera una excepción.

Finalmente, el método `split_pos(pos, l, r)` “particiona” el conjunto en $l = \langle k_0, k_1, \dots, k_{pos-1} \rangle$ y $r = \langle k_{pos}, \dots, k_{n-1} \rangle$. Una excepción ocurrirá si `pos` está fuera del rango.

1.3.3 Circunstancias del problema fundamental

Cualesquieran que sean las situaciones de utilización, los elementos de un conjunto tienen que guardarse en alguna clase de memoria. La manera de representar en memoria tal conjunto requiere de una estructura de datos cuya forma depende de sus circunstancias de uso.

Hay varios factores que inciden en el diseño o escogencia de la estructura de datos, entre los que cabe destacar los conocimientos que se tengan sobre la cardinalidad, la distribución

de referencia de los elementos, las operaciones que se usarían y la frecuencia con que éstas se invocarían.

La cardinalidad decide de entrada el tipo de memoria. Una cardinalidad muy grande requerirá memoria secundaria (disco) o terciaria (otros medios más lentos), y esta decisión afecta radicalmente el tipo de estructura de datos.

Hay ocasiones en que algunas claves son más propensas a ciertas operaciones que otras. Por ejemplo, si las claves fuesen apellidos, entonces el saber que las letras “x” o “y” son poco frecuentes, y que las vocales son más frecuentes, puede incidir en una estructura de datos que tienda a recuperar rápidamente claves que tengan, por ejemplo, “a” como segunda letra.

Según el problema, algunas operaciones son más probables que otras; inclusive, en muchos casos, no se utilizan todas las operaciones o la mayoría de las actividades sobre el conjunto se concentran en una o pocas operaciones. En estos casos, la estructura de datos puede diseñarse para optimizar la operación más frecuente.

1.3.4 Presentaciones del problema fundamental

En el ámbito funcional existen varias interpretaciones del tipo genérico *(Conjunto fundamental 18)*. Hay, en esencia, dos consideraciones dignas de resaltarse.

La primera consideración concierne a la repitencia o no de los elementos. Cuando se permite repetir los elementos de un conjunto, entonces a éste se le denomina “multiconjunto”.

En la biblioteca estándar C++, en adelante llamada stdc++, al conjunto se le conoce como `set<T, compare>`, mientras que al multiconjunto como `multiset<T, compare>`.

La segunda consideración es el almacenamiento de pares ordenados de tipo `(Key, Elem)`. La idea es una tabla asociativa que recupere una instancia `elem ∈ Elem` dada una clave `key ∈ Key`. A esta clase de conjunto se le conoce como “mapeo”⁸. Cuando las claves pueden repetirse, entonces al mapeo se le dice “multimapeo”.

En la biblioteca estándar C++ al mapeo se le conoce como `map<Key, Elem, compare>` mientras que al multimapeo como `multimap<Key, Elem, Compare>`. En los mapeos suele implantarse el operador `[]` según la clave.

1.4 Diseño de datos y abstracciones

El diseño de abstracciones y sus consecuentes TAD es un arte que se aprende con la experiencia. Adquirirla no tiene otra alternativa que enfrentarse responsablemente a problemas reales de programación. Por responsabilidad se entiende la actitud honorable a responder por los equívocos, lo cual no sólo está condicionado a la conciencia que el practicante tenga acerca de su conocimiento, sino a su honestidad y fuerza de carácter.

En lo que sigue de esta sección se plantean algunas reflexiones que debe considerar el aprendiz para enfrentar mejor el aprendizaje del diseño de datos y programación.

⁸Del inglés “mapping”, cuya connotación matemática significa función en el sentido de la teoría de conjuntos. Por otra parte, es importante destacar que el término fue recientemente aceptado por la RAE.

1.4.1 Tipos de abstracción

Según el interés que se tenga al momento de diseñar una abstracción o estructura de datos, ésta puede clasificarse en “orientada hacia los datos”, “orientada hacia el flujo” u “orientada hacia el concepto o abstracción”.

Una abstracción orientada hacia los datos es aquella cuyo fin está encauzado por la organización de los datos en el computador. A la vez, tal organización obedece a requerimientos de desempeño, ahorro de espacio o algún otro que atañe al computador, sistema operativo u otros programas sistema. Este es el caso de muchas de las estructuras de datos que estudiaremos en este texto. Ejemplos de estas clases de orientación son los arreglos, las listas enlazadas y las diversas estructuras de árbol que serán estudiadas en este texto.

Muchos problemas computacionales exhiben un patrón de procesamiento distintivo y uniforme. En tales situaciones puede ser muy conveniente disponer de una estructura de datos que represente el orden o esquema de procesamiento de los datos. En este caso decimos que la estructura está orientada hacia el flujo o al patrón. Por ejemplo, si los datos deben procesarse según el orden de aparición en el sistema, entonces una disposición de los datos en una secuencia puede representar el orden de llegada. Tal estructura se denomina “cola” y será estudiada en § 2.6 (Pág. 122). Notemos que en este caso no se piensa en la organización que los datos tengan en memoria, sino en el orden o patrón en que éstos se procesen.

Finalmente, el diseño de una estructura de datos puede facilitar la representación de un concepto o abstracción conocida con miras a comprender el problema y, consiguientemente, desenvolverse cómodamente en su solución. En este caso decimos que la estructura de datos está orientada hacia el concepto. La idea es simplificar al programador o a los usuarios el entendimiento del problema y de su solución.

Hay muchos caminos para resolver problemas. Cuentan que Michel Faraday, precursor de la teoría electromagnética, no tenía suficiente formación matemática para explicar los fenómenos electromagnéticos de sus experimentos. La genialidad de Faraday lo condujo a crear sus propias abstracciones gráficas, provenientes de sus observaciones experimentales, a partir de las cuales fundó y explicó el electromagnetismo. Al igual que Faraday, muchas veces creamos abstracciones que nos permiten comprender mejor un algoritmo. Estas abstracciones conforman estructuras de datos. Un ejemplo muy notable es el concepto de grafo. Etimológicamente, el término grafo proviene de “gráfico”, pues los grafos son expresados en términos gráficos. Sin embargo, en realidad, un grafo modeliza el concepto de relación sobre el cual existe todo un corpus matemático. A pesar del corpus y quizás porque éste es incompleto, los grafos ofrecen una visión gráfica de la relación matemática con la que es más cómoda trabajar. Esta es otra razón que justifica el diseño de una estructura de dato: una manera de representar el problema en términos más sencillos.

Estos tres tipos de orientación, de alguna forma clasifican el fin o el “para qué” se diseña o se selecciona una estructura de datos. La clasificación no es exacta ni excluyente. Una estructura de datos puede encajar a la vez en diferentes momentos, bajo todos o cualquiera de los tipos de orientación. Pero determinar en función de las circunstancias cuál es la orientación de una estructura de datos, puede guiar al programador en su diseño o selección.

1.4.2 El principio fin-a-fin

Consideremos el TAD *(Figure 8a)* y repitamos la pregunta fundamental: ¿para qué sirve?, ¿cuál es su finalidad? En la sección § 1.1.4 (Pág. 7) se pretendió “generalizar operaciones inherentes al dibujado de una figura sobre algún fondo de contraste”. Con este fin definido, las operaciones del TAD *(Figure 8a)*, dibujar, mover, etcétera, tienen sentido sin necesidad de conocer cuál es la figura en cuestión. Pensemos qué sucedería si no tuviésemos claro para qué se usaría el TAD *(Figure 8a)*. La respuesta, no tan obvia en estos tiempos, es que nos sería muy difícil comprenderlo. Si nuestro entendimiento no estuviese claro, entonces, quizá, cometeríamos el error de intentar implantar el TAD *(Figure 8a)*, el cual, como ya se mencionó, es abstracto.

Ahora pensemos en cuál sería la forma de un TAD *Figure* si éste estuviese destinado a cálculos geométricos en los cuales, en lugar de dibujar, se calculasen áreas e intersecciones entre figuras. Para este fin, las operaciones del TAD *(Figure 8a)* no tendrían mucho sentido.

Un principio de diseño de sistemas se conoce como “el principio fin-a-fin”. Este consiste en no especificar, menos, diseñar y mucho menos implantar, más allá del fin que se conozca y se acuerde para el programa. Violar este principio puede costar esfuerzo vano, pues sólo en los puntos finales del programa, o en sus usuarios finales, se tiene todo el conocimiento necesario para diseñar un programa con sentido [154].

En el ejemplo del TAD *(Complejo 5)*, tal como lo hemos tratado, ¿que conocemos acerca de su fin? Los números complejos tienen amplia aplicación en ciencias y en ingeniería, razón por la cual un programador pudiera verse tentado a enriquecer el TAD con métodos o clases derivadas que faciliten su futura manipulación. Se podría, por ejemplo, manejar coordenadas polares. Sin embargo, ampliar el TAD *(Complejo 5)* no tiene sentido si no se tiene la certitud de que las coordenadas polares serán usadas por los usuarios eventuales del TAD *(Complejo 5)*.

La observación anterior no se hace para economizar trabajo -una ganancia de consumo con el principio fin-a-fin-, sino porque el interesado en un TAD *(Complejo 5)* extendido con coordenadas polares podría manejar otra interpretación, en cuyo caso, la extensión podría ser un estorbo.

Un TAD debe ser “mínimo” y “suficiente”. Por mínimo pretendemos indicar que no tiene más de lo necesario. Por suficiente queremos decir que debe contener todo lo necesario para destinarlo al fin para el cual fue definido. Establecer estas barreras es relativo, pues depende del fin y de su interpretación. De allí, entonces, el carácter esencial que tiene, para el éxito de un proyecto, el que el fin esté claramente definido y que los participantes no sólo lo tengan claro, sino que estén comprometidos con él.

Quizá un aforismo de Saint-Exupéry exprese mejor el sentido de minimalidad y suficiencia del principio fin-a-fin: “*Parece que la perfección se alcanza no cuando no hay más nada que añadir, sino cuando no hay más nada que suprimir*”⁹.

1.4.3 Inducción y deducción

Inducción significa ir desde lo particular hacia lo general, mientras que deducción señala ir desde lo general hacia lo particular. Cuando se diseñan abstracciones, ¿por dónde

⁹ Traducción del autor de: “*Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher*”. Saint-Exupéry. “Terre des hommes”.

comenzar?.

Es un principio conocido en educación y diseño el ir desde lo concreto hacia lo abstracto. En otras palabras, el forjar abstracciones a partir de la experiencia concreta real. En ese sentido, cuando no se tenga conocimiento inicial acerca de un problema dado, el proceso de indagación debe comenzar a partir de fenómenos concretos del problema y, luego, a partir de esas particularidades, intentar definir abstracciones.

En la programación a objetos existen dos mecanismos de generalización: la herencia de clases y las plantillas. La herencia concierne a los datos, mientras que las plantillas se refieren al código.

La herencia se aplica para delinear generalidades y comportamientos comunes a una cierta clase de objeto. Las clases derivadas clasifican y aportan particularidades de comportamiento general y niveles de abstracción. Cuando se identifiquen clases o TAD, busque qué es lo común y eso llévelo a lo general a través de clases bases o abstractas.

Hay dos aspectos a generalizar mediante la herencia de clases. El primero lo componen los atributos, o sea, las características de un objeto dado. En el caso del TAD (*Figure 8a*), un atributo general lo conforma el punto de referencia común a todas las figuras particulares. El segundo aspecto de generalidad es funcional y ataña a las operaciones. En el caso del TAD (*Figure 8a*), los métodos virtuales `draw()`, `move()`, etcétera, generalizan operaciones comunes a todas las figuras.

Como ya indicamos, algunos algoritmos son susceptibles de ser genéricos bajo la forma de plantilla. Consideremos el problema de ordenar una secuencia de elementos que será tratado en el capítulo 3. Notemos que el enunciado no menciona el tipo de elementos a ordenar; sólo especifica que se trata de una secuencia. Podemos ordenar apellidos, enteros o elementos de cualquier otro tipo bajo el mismo esquema. Ordenar es independiente del dato. En esta clase de problemas se puede diseñar un ordenamiento genérico cuyo parámetro será el tipo de elementos.

Es importante destacar que un comportamiento genérico aparece después de conocer comportamientos concretos y no al contrario. Ni siquiera cuando se posea una amplia experiencia no se debe programar código genérico sin antes haberlo verificado exhaustivamente con al menos un tipo de dato conocido y concreto.

Para las dos técnicas de generalización, herencia y tipos parametrizados, el camino comienza en lo concreto y se dirige hacia lo abstracto, no al revés. Podemos decir que este es el estilo cuando se diseña y programa con sentido.

Conforme se gana experiencia concreta, un diseñador puede considerar algunas generalizaciones a priori (no todas) sin aún ver las particularidades. Esto es deducción y es posible después de aprehender o diseñar partes concretas de la solución.

La genialidad, cuando ocurre, es mirar en lo abstracto lo que puede devenir concreto. Ingenio significa tener genio desde adentro (in). Genio que genera ideas, buenas, por supuesto. Desde esta perspectiva ingeniería es entonces la práctica del in-genio; pero no se podría tener ingenio si siempre se exigiese permanecer en lo concreto y se supeditase a técnicas y métodos fijos. Esta es la razón por la cual la intuición nunca debe ser descartada.

1.4.4 Ocultamiento de información

Un TAD sólo especifica el fin de un dato y su interfaz. Cuando se acuerda un diseño en torno a un TAD, se acuerda una especificación objetiva que no dice nada acerca de su

implantación. Esto es conocido como el principio de ocultamiento de información, el cual consiste en ocultar deliberadamente la implantación de un TAD, pues, como ya dijimos, ésta conforma lo subjetivo, el cual, no sólo es mucho más complejo, sino que dificulta la comunicación.

A veces es bueno hablar de la implantación con la interfaz. Por ejemplo, decir que *(Conjunto fundamental 18)* está implantado con arreglos ordenados proporciona una idea acerca del desempeño; se sabrá, por ejemplo, que la búsqueda es rápida, pero que la inserción y supresión son lentas. Notemos que en este caso no se define exactamente cómo se implanta el TAD, sino que se indica, como parte de la especificación, un aspecto general de la implantación.

Por tanto, la recomendación general de diseño es que se oculte lo más que se pueda la implantación. Pero si por razones de desempeño o de requerimientos resulta conveniente establecer un tipo de implantación, trátese ésta entonces en los términos más genéricos posibles.

1.5 Notas bibliográficas

La programación orientada a objetos se remonta a finales de la década de 1960, cuando apareció el lenguaje Simula [34] con los conceptos de clase, herencia y polimorfismo. Hay dos observaciones históricas muy importantes. La primera es que Simula se circunscribió en el dominio de la simulación y no de la programación tradicional. La segunda es que todos los conceptos modernos de la orientación a objetos aparecieron primero que la noción matemática de tipo de dato abstracto.

Simula no debe haberse tenido muy en cuenta en su época porque transcurrieron algunas décadas antes de que se le desempolvase y considerase en el paradigma actual de los objetos. Por el contrario, los computistas, que se creen muy elitescos, conocen los tipos abstractos de datos desde los trabajos de Liskov y Zilles [109] y el ocultamiento de información desde los trabajos de Parnas [140].

El lenguaje C++, vehículo de enseñanza del presente texto, inspirado en los lenguajes C [94] y Smalltalk [83, 26, 168], fue creado por Bjarne Stroustrup. La mejor referencia para su aprendizaje es su propio texto *The C++ Programming Language* [164], el cual, aparte de que es posiblemente el mejor para comprender el lenguaje, es un excelente tratado de ingeniería de programación, que no tiene nada que ver con la gerencia de proyectos de software, actividad ahora injusta y vulgarmente conocida bajo el rótulo de ingeniería del software.

Este texto no versa sobre la interfaz de la biblioteca estándar C++ sino más bien acerca de su implantación. Es útil sin embargo estudiar la interfaz a efectos de no repetir trabajo y de homogeneizar criterios. Una excelente referencia sobre la biblioteca estándar C++ es el texto de Josuttis [88].

Un recuento histórico acerca del C++ y de la programación a objetos puede encontrarse en [163]. La lectura es muy interesante porque revela que las abstracciones y conceptos asociadas a los objetos son resultado del refinamiento a través de errores y fracasos.

El principio fin-a-fin ha sido observado desde épocas remotas, pero fue Guillermo de Occam, cuando enunció su célebre “navaja”, la cual reza “*no multiplique los entes sin necesidad*”, de quien primero se conoce su importancia epistemológica. En la programación, el principio ha sido observado en grandes sistemas, siendo al respecto emblemático el artículo

de Saltzer *et al* [154]. Sobre este artículo es menester comentar que Saltzer *et al* orientan su artículo a sistemas distribuidos y no directamente al diseño de datos. Por otra parte, el término “fin” se interpreta a menudo como “extremo” y no como un propósito.

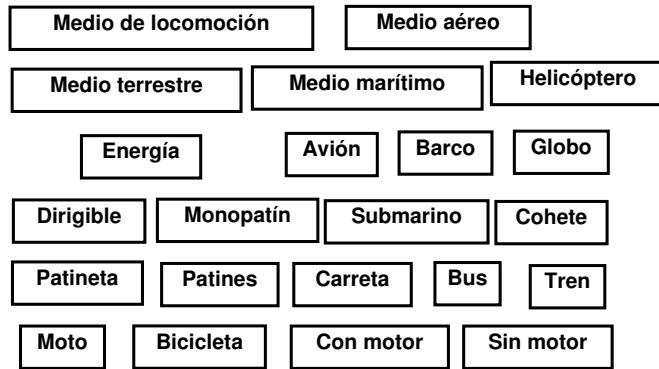
A través de la experiencia se descubren datos generales y géneros de código. A una categoría consolidada de clase o código suele denominársele “componente” o “patrón”. Un repertorio bastante rico de patrones genéricos básicos puede encontrarse en [57].

Si bien para dominar la programación se requieren algunos años de experiencia como autor de programas, los textos de Scott Meyers [126, 127] constituyen la mejor referencia para comprender, dominar y saber usar muchas de las idiosincrasias del C++.

La historia de UML [65] se remonta a OMT [152], un lenguaje gráfico, precursor del actual UML, propuesto por James Rumbaugh, un científico célebre de la programación a objetos. El consorcio OMG (Object Management Group), en el ámbito de los sistemas distribuidos a objetos, tomó OMT como base para desarrollar el actual UML.

1.6 Ejercicios

1. Diseñe e implante un TAD que represente números en punto flotante y en el cual se especifique la precisión, es decir, el tamaño de la mantisa y del exponente.
2. Critique el TAD *(Complejo 5)*. ¿Que tan completo es?, ¿es correcta su especificación semántica?, ¿Qué problemas de dominio y resultados pueden ocurrir?
3. Amplíe el TAD *(Complejo 5)* para manejar coordenadas polares. Discuta dónde poner la ampliación (en nuevos métodos, en una clase derivada, etcétera)
4. Diseñe e implante un TAD que represente números de precisión arbitraria.
5. Revise fuentes de programas libres para dibujar como Xfig y DIA e indague la jerarquía de clases con que ellos modelizan las figuras.
6. Identifique los objetos fundamentales para la conducción de un automóvil que se encuentran en la cabina. Para cada uno, establezca su fin y las operaciones junto con sus especificaciones sintáctica y semántica.
7. Diseñe inductivamente una jerarquía de clases que represente vehículos automotores. Dibuje los diagramas UML.
8. Diseñe deductivamente una jerarquía de clases que represente “viviendas”. Dibuje los diagramas UML.
9. Considere las siguientes clases de objeto cuyos nombres sugieren lo que representan:



Diseñe un esquema deductivo que relacione estas clases.

10. Mencione tres o más aplicaciones en las que aparezca el problema fundamental de las estructuras de datos. Para cada aplicación, explique la forma en que aparece y diserte brevemente acerca de cómo se implantaría.
11. Mencione tres o más aplicaciones en las que no aparezca el problema fundamental de las estructuras de datos.
12. Mencione algunas estructuras de datos conocidas y discuta su clasificación según los lineamientos explicados en la sección § 1.4.1 (Pág. 21).
13. Dado un conjunto S definido por el tipo $\text{Set}\langle T, \text{Compare} \rangle$, explique cómo conocer el elemento correspondiente a la mediana en el sentido estadístico.
14. Dado un conjunto S definido por el tipo $\text{Set}\langle T, \text{Compare} \rangle$, explique cómo se programaría, en función de las primitivas de *(Conjunto fundamental 18)*, la rutina:

```

template <class __Set>
__Set extraer(__Set & set, int i, int j);
  
```

la cual extrae de S , y retorna en un nuevo conjunto, todos los elementos que están entre las posiciones i y j respectivamente. Después de la operación, S contiene los elementos entre los rangos $[0..i - 1]$ y $[j + 1..n - 1]$, donde $n = |S|$.

15. Asuma que el fin de un sistema es la administración de la escolaridad de una carrera universitaria. Bajo este fin se desea disponer de bases de datos de estudiantes, profesores, carreras, cursos, secciones, salones, horarios y demás aspectos propios de la administración escolar de una universidad.

Plantee TAD generales, particulares y parametrizados, que modelicen las diversas abstracciones que manejaría el sistema. Dibuje los diagramas UML para todas las clases diseñadas.

16. Reconsidere el ejercicio anterior para ciclos escolares de secundaria y primaria.

2

Secuencias

En programación, así como en otros ámbitos de nuestra cultura, una secuencia se define como una sucesión de elementos de algún tipo. Por sucesión entendemos que los elementos de la secuencia puedan mirarse según cierto orden de aparición o procesamiento, uno tras otro, de izquierda a derecha, de arriba hacia abajo y otras combinaciones que mantengan el carácter sucesivo subyacente a la idea de secuencia.

En la vida cotidiana lidiamos con secuencias sin que casi nunca nos maravillemos de sus consecuencias, las cuales se nos desaparecen en la unidad de la vida. En castellano, y en otras lenguas, leemos y escribimos de izquierda a derecha y desde arriba hacia abajo, o sea, secuencialmente. La mayoría de las veces, la lectura ocurre sin que nuestro pensamiento intervenga y fragmente el sentido de lo que leemos. Decimos entonces que leemos fluidamente. Secuencial situación a veces sucede en matemática, dominio donde solemos operar, al menos asociativamente, de izquierda a derecha y de algunos otros modos más reservados para los matemáticos excelentes.

No sólo las secuencias nos son ubicuas, sino que muchas veces éstas tienen diferentes perspectivas de mira o distintos modos y tiempos para presentarse e interpretarse. Consideremos, por ejemplo, el caso de una película vista como una secuencia de escenas hilvanada según el sentido que el director nos pretenda transmitir de la historia. Para aproximarnos a lo que como película se quiere presentar, debemos mirar la película secuencialmente. Un corte en la secuencia o una permutación entre sus escenas, puede volver a la película ininteligible o, quizás, “en el mejor de los casos”, ofrecer una interpretación distinta. Por supuesto, lo anterior no nos impide, en retrospectiva, luego de haber presenciado enteramente la película, mirar algunas de sus partes para mejorar nuestra comprensión. Visto de otro modo, una película consiste en una secuencia de fotografías cuya sucesión reconstruye las escenas con impresión de realidad.

El computador no escapa a la ubicuidad de las secuencias. No muy otrora fue clasificado de “máquina secuencial”, pues se remite a leer y ejecutar programas, los cuales no son más que secuencias de instrucciones escritas en una “memoria”. He aquí, pues, un indicio serio de que cualquier abstracción de secuencia es ampliamente usada en la computación.

En este capítulo estudiaremos las siguientes estructuras de datos caracterizadas como secuencias:

- Arreglos
- Listas enlazadas

- Pilas
- Colas

Cualquiera de estas estructuras es ubicua por todas las ciencias computacionales y es muy probable que sean parte del camino crítico de ejecución. Por esta razón, es deseable conocer las implantaciones más eficientes posibles.

Los arreglos y listas enlazadas conforman abstracciones de datos, mientras que las pilas y colas son abstracciones de flujo. Una secuencia de elementos del mismo tipo conforma un conjunto, lo cual sugiere de entrada que una secuencia, vista como abstracción de dato, puede implantar el problema fundamental de estructura de datos estudiado en el capítulo 1. Este será el énfasis que le impartiremos a los arreglos y las listas enlazadas.

Otras situaciones computacionales requieren un patrón de procesamiento particular. Este es el sentido de las pilas y las colas, cuyos nombres abstraen en cierta forma el orden de procesamiento.

2.1 Arreglos

Un arreglo de dimensión dim es una secuencia de $n \leq \text{dim}$ elementos del mismo tipo en la cual el acceso a cada elemento es directo y consume un tiempo constante e independiente de su posición dentro de la secuencia.

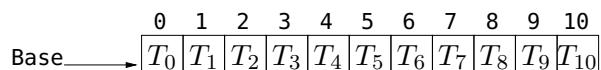
Por lo general, los elementos de un arreglo se organizan de forma directamente contigua en la memoria del computador, lo que proporciona dos bondades que hacen al arreglo muy interesante para la programación de sistemas:

1. Se puede acceder directamente a cualquier elemento según su posición dentro de la secuencia. Por lo general, esto se implanta a nivel de compilación y se especifica a nivel del lenguaje de programación mediante el operador `[]`. Por ejemplo, la expresión en C++: `a[15] = 9` asigna el entero 9 al décimo sexto elemento del arreglo.
2. El costo en espacio es mínimo, casi siempre la cantidad de elementos.

Ninguna estructura de dato ofrece mejor rendimiento en el acceso por posición. El hecho de que los elementos estén contiguos favorece extraordinariamente a las aplicaciones que exhiban localidad de referencia, pues es muy probable que los elementos cercanos en espacio y tiempo se encuentren en el cache¹.

La mayoría de los lenguajes de programación modernos ofrecen soporte para el manejo de arreglos.

Consideremos un arreglo `A[11]` de elementos de tipo `T` cuyo tamaño en bytes es `sizeof(T)` y que se pictoriza del siguiente modo:



El compilador asocia al arreglo la dirección del primer elemento llamada “base del arreglo”, cuyo valor se determina según el lugar donde ocurra la declaración o instanciación y el

¹En este contexto un cache es una estructura especial de dato, soportada por el hardware, la cual, si se usa correctamente, acelera considerablemente la velocidad de acceso a la memoria.

modo de declaración. Cuando el compilador encuentra un acceso a la i -ésima posición, éste genera el siguiente cálculo de acceso a la memoria:

$$\text{Dirección del elemento} = \text{base} + i \times \text{sizeof}(T) \quad (2.1)$$

La operación asume que el primer índice del arreglo es cero, que es el caso en los lenguajes C y C++. Si el lenguaje permite que los índices comiencen en valores arbitrarios, entonces el compilador debe generar una operación adicional antes de poder hacer el cálculo anterior, lo que hace un poco más lento el acceso.

En general, los compiladores no efectúan verificación de rango. Es decir, no se aseguran de que el índice de referencia al arreglo esté dentro de los rangos correctos. La razón es que esta verificación le añade un coste constante a cada acceso que puede ser importante. Consecuentemente, un acceso fuera de rango es un grave error de programación cuyos síntomas pueden pasar desapercibidos durante algún tiempo. Este tipo de error es muy difícil de detectar, razón por la cual es buena idea utilizar asertos² de verificación de rangos antes de referenciar un arreglo.

2.1.1 Operaciones básicas con Arreglos

Hay varios aspectos a considerar a la hora de operar sobre arreglos:

- Si los elementos están o no ordenados.
- Si se conoce el orden de inserción/supresión de los elementos.
- Si se conoce con antelación el número de elementos.
- ¿Qué tan iterativas serán las operaciones? Una alta interactividad significa que las inserciones, búsquedas y eliminaciones pueden suceder frecuentemente y con la misma probabilidad.
- El tamaño y patrón de acceso de los elementos que alberga el arreglo. Cuanto más grande sea el elemento, menos beneficios se obtendrán por el cache.

2.1.1.1 Aritmética de punteros

Una de las grandes virtudes del lenguaje C, trascendida al C++, es lo que se conoce como “aritmética de punteros”. Tal concepto consiste en incorporar al lenguaje operaciones sobre punteros cuyos resultados consideran el tipo de dato al cual se apunta. Para aclarar este concepto desarrollemos un ejemplo.

La instrucción `T * arreglo_ptr = new T[n];` declara un puntero a una celda de memoria de tipo genérico T. A la vez, se aparta memoria para albergar n celdas contiguas de tipo T, o sea, un arreglo de dimensión n.

Normalmente, si `arreglo_ptr` es la base de un arreglo, entonces, para acceder al i -ésimo elemento tendríamos que efectuar un cálculo similar a (2.1). Ahora bien, el cálculo implicado en (2.1) lo genera automáticamente el compilador cuando, por ejemplo, escribimos: `*(arreglo_ptr + i) = 5;`. El operando izquierdo de la asignación se interpreta

²El D.R.A.E. expresa para “aserto”: “Afirmación de la certeza de algo”. A un aserto se le conoce más como precondition o invariante.

En este texto se utilizan invocaciones a la primitiva I (predicado) de la biblioteca nana [144].

como: acceda al contenido de la dirección de memoria `arreglo_ptr` más `i` enteros. El compilador sabe que se trata del tipo `int` y no de otro porque el puntero fue declarado como un puntero a entero. Con este conocimiento, el compilador, implícita y transparentemente, incorpora el tamaño de un entero (`sizeof(int)`) en el cálculo del acceso.

La expresión `*(arreglo_ptr + i)` es exactamente equivalente a `arreglo_ptr[i]`, la cual se traduce: acceda al `i`-ésimo entero de la secuencia cuya dirección base es `arreglo_ptr`. Quizá un problema de este enfoque es que es más difícil distinguir el acceso a un arreglo explícitamente declarado de un acceso mediante un puntero. Por esta razón es conveniente que el nombre del puntero refleje su condición. En el caso ejemplo, el sufijo `ptr`³ denota que se trata de un apuntador.

2.1.1.2 Búsqueda por clave

Si los elementos están ordenados, entonces podemos usar un fabuloso algoritmo llamado “búsqueda binaria”, cuya especificación genérica es como sigue:

30

(Búsqueda binaria 30)≡

```
template <typename T, class Compare>
int binary_search(T a[], const T & x, int l, int r)
{
    int m; // índice del medio
    while (l <= r) // mientras los índices no se crucen
    {
        m = (l + r)/2;

        if (Compare() (x, a[m]))           // ¿es x < a[m]?
            r = m - 1;
        else if (Compare() (a[m]), x)     // ¿es x > a[m]?
            l = m + 1;
        else
            return m; // ==> a[m] == x ==> encontrado!
    }
    return m;
}
```

Defines:

`binary_search`, used in chunks 31a and 32.

`binary_search<T, Compare>()` busca una ocurrencia del elemento `x`, de tipo genérico `T`, dentro del rango comprendido entre `l` y `r` del arreglo ordenado `a`.

El principio de la búsqueda binaria es dividir el arreglo en dos partes iguales de tamaño `m = (l + r)/2`, y en caso de que `a[m]` no contenga `x`, buscar en alguna de las mitades según el orden de `x` respecto a `a[m]`.

Nótese que `binary_search<T, Compare>()` retorna una posición válida aun si `x` no se encuentra en el arreglo. Esto requiere que el cliente, luego de la búsqueda, compare de nuevo `x` con el valor de retorno para verificar si `x` fue hallado o no. Aunque esto conlleva un ligero coste, permite conocer la posición en dónde se insertaría `x` en el arreglo de manera que éste siga ordenado.

Como estudiaremos en la subsección § 3.1.9 (Pág. 165), el rendimiento de este algoritmo es proporcional $\lg n$. Si bien este coste es mayor que el constante del acceso por posición, en la práctica es bastante bueno.

³Abreviación en inglés de “pointer”.

Si los elementos no están ordenados, entonces puede efectuarse, a costa del desempeño, la búsqueda secuencial descrita en § 1.2.3.3 (Pág. 15) bajo el nombre de función `sequential_search<T, Compare>()`. Si bien la iteración requerida por esta clase de búsqueda puede requerir inspeccionar todas las celdas del arreglo, es aceptable como solución para escalas medianas.

2.1.1.3 Inserción por clave

Si el arreglo se mantiene desordenado, entonces la inserción es rapidísima mediante añadidura al final de la secuencia. De lo contrario, la inserción deviene más costosa, pues se requiere buscar en el arreglo la posición de inserción, luego, desplazar los elementos de la secuencia en una posición hacia la derecha y, finalmente, copiar el elemento. Tal algoritmo se denomina “inserción por apertura de brecha” y se instrumenta como sigue:

31a

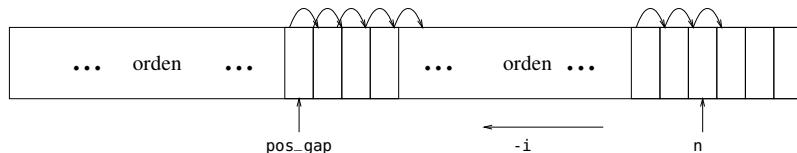
(Inserción por brecha 31a)≡

```
template <typename T, class Compare>
void insert_by_gap(T a[], const T & x, int & n)
{
    const int pos_gap = binary_search<T, Compare>(T, x, 0, n - 1);
    for (int i = n; i > pos_gap; -i)
        a[i] = a[i - 1];

    a[pos_gap] = x;
    ++n;
}
```

Uses `binary_search` 30.

Este algoritmo se pictoriza del siguiente modo:



La rutina asume que `n` es la cantidad de elementos que tiene la secuencia y que este valor no iguala o excede la dimensión del arreglo.

`insert_by_gap<T, Compare>()` toma el tiempo de búsqueda, que es rapidísimo, más el tiempo de abrir la brecha que requiere la iteración y la copia.

2.1.1.4 Eliminación por clave

Con arreglos, la eliminación primero requiere conocer la posición dentro del arreglo del elemento a eliminar. Esto puede hacerse mediante la rutinas `sequential_search<T, Compare>()` o `binary_search<T, Compare>()`, según que el arreglo esté o no ordenado. En ambas situaciones, el elemento eliminado deja un “hueco” o “brecha” que debe “taparse”.

Si el arreglo está desordenado, entonces la eliminación puede llevarse a cabo como sigue:

31b

(Eliminación en arreglo desordenado 31b)≡

```
template <typename T, class Equal>
void remove_in_unsorted_array(T a[], const T & x, int & n)
{
    const int pos_gap = sequential_search<T, Equal>(T, x, 0, n - 1);
    if (a[pos_gap] != x)
```

```
// excepción: x no se encuentra en a[]

a[pos_gap] = a[n - 1];
-n;
}

Uses sequential_search 154a.
```

`remove_in_unsorted_array<T, Equal>()` tapa la brecha con el último elemento de la secuencia.

Si el arreglo está ordenado, entonces se requiere tapar la brecha mediante desplazamiento hacia la izquierda de todos los elementos que están a la derecha de la brecha de la siguiente forma:

32 *(Eliminación en arreglo ordenado 32)≡*

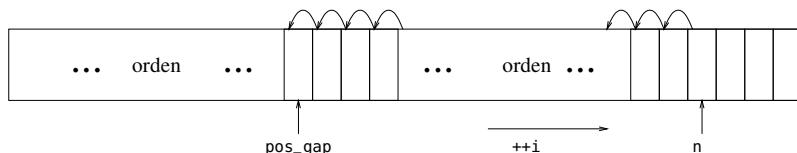
```
template <typename T, class Equal>
void remove_in_sorted_array(T a[], const T & x, int & n)
{
    const int pos_gap = binary_search<T, Equal>(T, x, 0, n - 1);
    if (a[pos_gap] != x)
        // excepción: x no se encuentra en a[] ;

    for (int i = pos_gap; i < n; ++i)
        a[i] = a[i + 1];

    -n;
}
```

Uses `binary_search` 30.

La técnica en cuestión se pictoriza de la siguiente forma:



2.1.2 Manejo de memoria para arreglos

Según el esquema en que se aparte la memoria para un arreglo, éste se clasifica en estático y dinámico.

2.1.2.1 Arreglos en memoria estática

Este tipo de arreglo se declara global o estáticamente dentro de una función ⁴ y exige que se conozca la dimensión en tiempo de compilación. El arreglo ocupa todo su espacio durante toda la vida del programa y su dimensión no puede cambiarse.

En general, este tipo de arreglo debe usarse para guardar constantes que serán utilizadas a lo largo de toda (o la mayor parte de) la vida del programa. También puede utilizarse como depósito de valores iterativos; por ejemplo, una aplicación numérica que

⁴En C y en C++, un arreglo estático se declara dentro de una función o un módulo precedido de la palabra reservada `static`. En el caso de un módulo, el arreglo sólo es visible dentro del módulo y a partir del punto de declaración. En el caso de una función, el arreglo sólo es visible dentro de la función.

siempre efectúe operaciones con matrices de dimensión fija y cuyo valores cambian en función de la entrada del problema.

2.1.2.2 Arreglos en pila

Este tipo de arreglo es el que se declara dentro de una función⁵. El espacio de memoria es apartado de la pila de ejecución del programa. Puesto que la pila de ejecución es vital para el programa, se debe tener cuidado con un desborde de pila causado por una excesiva longitud del arreglo. Esto puede ser crítico en programas concurrentes multi-thread o en ambientes que soporten corrotinas.

La ventaja de este tipo de arreglo es doble. Primero el compilador no requiere conocer su dimensión para generar cualquier código de reservación de memoria o de referencia al arreglo. En consecuencia, es posible esperar hasta ejecutar la declaración para conocer su dimensión. La siguiente sección de código es factible en compiladores [GNU]:

```
void foo(size_t n) { int array[n]; ... }
```

La segunda ventaja es que el espacio ocupado por el arreglo se aparta automáticamente en el momento de ejecución de la declaración y se libera a la salida del ámbito de la declaración. En el ejemplo anterior, el arreglo es liberado a la salida de la función `foo()`.

A causa del riesgo de desborde de pila, el uso de este tipo de arreglo debe restringirse a dimensiones pequeñas. Debe prestarse atención esmerada al nivel de las llamadas a las funciones, pues a medida que se aniden las llamadas, se consume más pila.

No use arreglos en pila dentro de funciones recursivas o dentro de funciones que serán llamadas recursivamente.

2.1.2.3 Arreglos en memoria dinámica

Un arreglo en memoria dinámica es aquel apartado a través del manejador de memoria. En C++, por ejemplo, la siguiente instrucción reserva un arreglo de 1000 enteros:

```
int * array = new int [1000];
```

El tamaño del arreglo sólo está limitado por la cantidad de memoria disponible. La dimensión puede especificarse como una variable.

Este tipo de arreglo exige que la memoria se libere cuando el arreglo ya no se requiera. En el ejemplo anterior, esto se efectúa mediante:

```
delete [] array;
```

Es importante resaltar que el operador `delete []` debe imperativamente usar los corchetes, pues esta es la manera de indicar al compilador que se libera un arreglo y no un bloque de memoria. Se requiere de esta sintaxis porque es necesario llamar a todos los destructores; uno por cada entrada del arreglo.

La memoria del arreglo debe liberarse apenas estemos seguros de que éste no será requerido.

El arreglo en memoria dinámica es la opción de facto para la mayoría de las aplicaciones que usen arreglos de dimensión mediana o grande. Delegué la verificación de rangos a utilitarios de depuración especializados tales como `dmalloc` [179] o a `valgrind` [129].

⁵Atención: esto no necesariamente es cierto en otros lenguajes. En FORTRAN, por ejemplo, la mayoría de los arreglos son estáticos.

2.1.3 Arreglos dinámicos

Un arreglo dinámico es aquél en el cual su dimensión puede modificarse dinámicamente. Esto es muy útil para aplicaciones que se deben beneficiar del rápido acceso por posición, pero que no conocen el número de elementos que se pueden manejar. Por ejemplo, un tipo especial de recuperación clave-dirección, llamado hashing dinámico, aumenta y contrae progresivamente la dimensión de un arreglo. Esta estructura se explica en la sección § 5.1.7 (Pág. 412).

C y C⁺⁺ no soportan arreglos dinámicos. Esto implica que los arreglos dinámicos deben surtirse por un TAD en biblioteca.

Ahora bien, ¿cómo implementar un arreglo dinámico? La candidez sugiere que se relocalice el arreglo. Inicialmente se fija una dimensión y se aparta memoria para el arreglo. Si la dimensión es alcanzada, entonces se determina una nueva dimensión mayor y se aparta un nuevo bloque de memoria. Después, los elementos se copian al nuevo bloque y, finalmente, se libera el antiguo bloque.

Este enfoque ha sido utilizado con éxito en algunos sistemas importantes. La biblioteca estándar C ofrece una primitiva, `realloc()`, cuya sintaxis es la siguiente:

```
void *realloc(void *ptr, size_t size);
```

La rutina cambia el tamaño del bloque de memoria apuntado por `ptr`, el cual debe haber sido obtenido de llamadas previas a `malloc()`, `calloc()` o `realloc()`. El contenido de `ptr` permanecerá inmodificado al mínimo posible entre el antiguo y el nuevo tamaño `size`.

“Teóricamente”, `realloc()` es “inteligente” y capaz de apartar un bloque más grande sin cambiar la dirección de memoria. Esto es interesante porque se evita la copia de los elementos del antiguo bloque hacia el nuevo. Empero, en la práctica, esto no es suficiente porque no es una garantía. De hecho, muy pocas implementaciones de `realloc()` hacen un esfuerzo por verificar si hay memoria contigua disponible a `ptr`.

2.1.4 El TAD DynArray<T>

En esta sección mostraremos toda la interfaz e implementación de un TAD, llamado `DynArray<T>`, el cual modeliza un arreglo dinámico de elementos de tipo `T`.

`DynArray<T>` abstrae una “dimensión actual” con un valor inicial especificado en tiempo de construcción. Tal dimensión actual se conoce mediante el método `size()`.

`DynArray<T>` garantiza un consumo de memoria proporcional a la cantidad de entradas escritas del arreglo, es decir, las entradas que no han sido escritas, no necesariamente consumen memoria. La memoria se reserva sólo cuando se escriben las entradas por primera vez.

La dimensión actual se expande perezosa y automáticamente cuando se escribe sobre un índice que está fuera de la dimensión actual. Si se referencia un índice como lectura mayor o igual a la dimensión actual, entonces se genera la excepción fuera de rango.

La dimensión actual puede contraerse explícitamente mediante una operación llamada `cut()`. La función libera entonces toda la memoria ocupada entre la nueva y anterior dimensión actual.

`DynArray<T>` se especifica en el archivo `(tpl_dynArray.H 34)`, el cual exporta la clase parametrizada `DynArray<T>`:

```
template <typename T> class DynArray
{
    ⟨miembros constantes de DynArray<T> 37a⟩
    ⟨miembros privados de DynArray<T> 38b⟩
    ⟨miembros públicos de DynArray<T> 42⟩
};

⟨Iniciación constantes DynArray<T> (never defined)⟩
```

Defines:

DynArray, used in chunks 42, 43c, 45a, 49, 50a, 54b, 155b, 253, 315a, 361–63, 607, 630a, 633c, 641, 693b, 704, and 756.

Es indispensable que exista el constructor por omisión `T:::T()`, así como el destructor `T:::~T()`. La ausencia de cualquiera de ellos se reporta como error en tiempo de compilación.

Un `DynArray<T>` asocia una dimensión máxima. Un arreglo no puede expandirse más allá de aquella dimensión. Por omisión, la dimensión máxima es 2 Giga entradas ($2 \times 1024 \times 1024 \times 1024$)⁶.

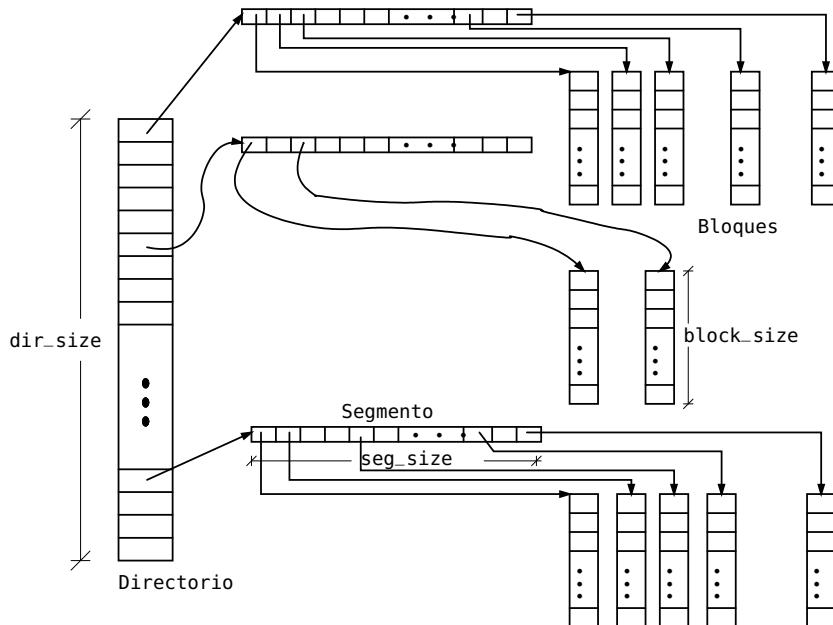


Figura 2.1: Estructura de datos de `DynArray<T>`

⁶Sobre una máquina de 32 bits, Pentium III, por ejemplo, un proceso dispone de un espacio de direccionamiento de 2^{32} . Como el espacio de direccionamiento debe contener el código del programa, sus datos estáticos y sus datos dinámicos, la cantidad de espacio disponible es menor que 2^{32} . Además, el sistema operativo reserva una parte del espacio de direccionamiento para sí mismo. Un arreglo de 2 Giga enteros cortos (`short`) no cabría en un proceso. Por otra parte, se requeriría al menos una memoria virtual de 4 Giga bytes para albergar este arreglo. Por estas razones creemos que 2 Giga entradas es suficiente para la mayoría de las situaciones.

Lo anterior deja de ser válido en sistemas persistentes en los cuales grandes arreglos son mapeados en disco y en memoria y dinámicamente cargados a la demanda. Las arquitecturas de 64 bits o más, hacen posible espacios de direccionamiento persistentes de envergadura muy grande. Empero, la consecución de un sistema operativo, paradigma de la persistencia, es aún un tema muy inmaduro de investigación.

2.1.4.1 Estructura de datos de DynArray<T>

La figura 2.1 (página 35) esquematiza la representación en memoria de un DynArray<T>, la cual ilustra tres clases de componentes:

Directorio: Arreglo de apuntadores a segmentos cuya dimensión es `dir_size`.

Segmento: Arreglo de apuntadores a bloques cuya dimensión es `seg_size`. Pueden existir hasta `dir_size` segmentos.

Bloque: Arreglo de elementos de tipo T cuya dimensión es `block_size`. En total, pueden existir hasta `dir_size × seg_size` bloques. Los bloques almacenan los elementos del arreglo. La dimensión máxima del arreglo es `dir_size × seg_size × block_size` elementos.

El carácter dinámico del arreglo reside en el hecho de que sólo se utilizan los bloques y segmentos que corresponden a elementos del arreglo que ya han sido escritos o mediante un método especial llamado `touch()`.

Dado un índice *i*, el acceso implica calcular la entrada en el directorio, luego la entrada en el segmento y, finalmente, la entrada en el bloque. Estos cálculos pueden efectuarse como sigue:

Índice del segmento en el directorio: Cada segmento representa `seg_size × block_size` elementos del arreglo. El índice en el directorio está dado por la expresión siguiente:

$$\text{pos_in_dir} = \frac{i}{\text{seg_size} \times \text{block_size}} \quad (2.2)$$

Utilizaremos el nombre de variable `pos_in_dir` cada vez que requiramos almacenar un índice en el directorio.

Índice del bloque en el segmento: El resto de la división (2.2), denotado *resto*, expresa el número de bloques que faltan para llegar al índice *i*.

Puesto que cada bloque posee `block_size` elementos, la posición dentro del segmento está dada por:

$$\text{pos_in_seg} = \frac{\text{resto}}{\text{block_size}} \quad (2.3)$$

Donde *resto* se define como:

$$\text{resto} = i \bmod (\text{seg_size} \times \text{block_size}) \quad (2.4)$$

El nombre de variable `pos_in_seg` será utilizado cada vez que se desee memorizar un índice de segmento.

Índice del elemento en el bloque: El índice del elemento dentro del bloque se obtiene a través del resto de la división (2.3). Así pues, el resto de esta división es el número de elementos que faltan para llegar al *i*-ésimo elemento dentro del bloque:

$$\text{pos_in_block} = \text{resto} \bmod \text{block_size} \quad (2.5)$$

Sustituyendo (2.4) en (2.5):

$$\text{pos_in_block} = (\text{i mod } (\text{seg_size} \times \text{block_size})) \text{ mod } \text{block_size} \quad (2.6)$$

Estas operaciones deben efectuarse siempre que se desee hacer un acceso. Puesto que cada operación toma un máximo de tiempo constante, se garantiza que el acceso a un arreglo dinámico tome un máximo de tiempo que también es constante.

Para mejorar el tiempo de cálculo de (2.2), (2.3) y (2.6), los tamaños de directorio, de segmento y de bloque son potencias exactas de dos. De esta forma, la multiplicación y la división pueden hacerse con simples desplazamientos.

A efectos de ganar tiempo de ejecución, algunos cálculos se realizan en tiempo de construcción y jamás vuelven a modificarse durante el tiempo de vida de una instancia de `DynArray<T>`.

37a *(miembros constantes de DynArray<T> 37a)≡* (34) 37b ▷
 mutable size_t pow_dir;
 mutable size_t pow_seg;
 mutable size_t pow_block;

Estas constantes almacenan las potencias de 2 de las longitudes del directorio, segmento y bloque, la cuales son $2^{\text{pow_dir}}$, $2^{\text{pow_seg}}$ y $2^{\text{pow_block}}$ respectivamente.

La dimensión máxima resultante es $2^{\text{pow_dir}} \times 2^{\text{pow_seg}} \times 2^{\text{pow_block}}$. Si este cálculo resulta mayor que la máxima dimensión permitida `Max_Dim_Allowed`, entonces se genera la excepción `std::length_error`. Si no hay capacidad numérica para efectuar las operaciones mediante desplazamientos, entonces se genera la excepción `std::overflow_error`.

Antes de continuar, ténganse en cuenta las siguientes igualdades:

$$\text{seg_size} = 2^{\text{pow_seg}} \quad (2.7)$$

$$\text{block_size} = 2^{\text{pow_block}} \quad (2.8)$$

Sustituyendo (2.7) y (2.8) en (2.2) tenemos:

$$\text{pos_in_dir} = \frac{\text{i}}{2^{\text{pow_seg}} \times 2^{\text{pow_block}}} = \frac{\text{i}}{2^{(\text{pow_seg}+\text{pow_block})}} \quad (2.9)$$

37b *(miembros constantes de DynArray<T> 37a)≡* (34) ▷ 37a 38a ▷
 mutable size_t seg_plus_block_pow;

El valor $2^{(\text{pow_seg}+\text{pow_block})}$ se guarda previamente en `seg_plus_block_pow` para futuros cálculos.

Sustituyendo (2.7) y (2.8) en (2.4), tenemos:

$$\text{resto} = \text{i mod } (2^{\text{pow_seg}} \times 2^{\text{pow_block}}) = \text{i mod } 2^{(\text{pow_seg}+\text{pow_block})} \quad (2.10)$$

Ahora debemos encontrar una manera de calcular rápidamente el cociente y el resto de una división entre una potencia exacta de 2. Para el caso del cálculo de la expresión (2.9), ya sabemos que el cociente es el resultado de desplazar i hacia la derecha `seg_plus_block_pow` veces. De esta manera, la expresión (2.9) resulta, mediante desplazamientos, en:

$$\text{pos_in_dir} = \text{i} \gg \text{seg_plus_block_pow} \quad (2.11)$$

El resto está dado por los `seg_plus_block_pow` bits menos significativos de `i`. Para conocer los `seg_plus_block_pow` bits menos significativos construimos un número con los `seg_plus_block_pow` bits menos significativos en uno y los restantes en cero.

38a $\langle \text{miembros constantes de } \text{DynArray} < T > \rangle \equiv$ (34) $\triangleleft 37\text{b}$ $39\text{b} \triangleright$
`mutable size_t mask_seg_plus_block;`

`mask_seg_plus_block` se calcula del siguiente modo:

$$\text{mask_seg_plus_block} = 2^{\text{seg_plus_block_pow}} - 1 \quad (2.12)$$

En base binaria, $2^{\text{seg_plus_block_pow}}$ contiene puros ceros excepto un uno en la posición `seg_plus_block_pow`. Como hay `seg_plus_block_pow` ceros antes de llegar al único uno, cada bit provoca un presto que es por fin pagado cuando se llega al bit en `seg_plus_block_pow`. Así pues, este número tiene los primeros `seg_plus_block_pow` en uno y los restantes en cero. El resto en la expresión (2.10) puede calcularse con un AND lógico cuyo operador en C⁺⁺ es `&`:

$$\text{resto} = i \text{ AND } \text{mask_seg_plus_block} = i \& \text{mask_seg_plus_block} \quad (2.13)$$

Es común denotar a la variable `mask_seg_plus_block` como una “máscara” porque enmascara los bits a la izquierda de `seg_plus_block_pow`.

38b $\langle \text{miembros privados de } \text{DynArray} < T > \rangle \equiv$ (34) $39\text{a} \triangleright$
`size_t mask_seg;`
`size_t mask_block;`

`mask_seg` se denota por la siguiente expresión:

$$\text{mask_seg} = 2^{\text{pow_seg}} - 1 = \text{seg_size} - 1 ; \quad (2.14)$$

Lo que permite un cálculo para la posición en el bloque:

$$\text{pos_in_seg} = \text{resto} \gg \text{pow_block} \quad (2.15)$$

Donde `resto` es (2.13).

`mask_block` se denota por:

$$\text{mask_block} = 2^{\text{pow_block}} - 1 = \text{block_size} - 1 \quad (2.16)$$

Para calcular el índice dentro de un bloque (2.5), hay que calcular el resto de dividir entre `block_size`. Para ello utilizamos la máscara `mask_block`. De esta forma, (2.6) se plantea como sigue:

$$\text{pos_in_block} = (i \& \text{mask_seg_plus_block}) \& \text{mask_block} \quad (2.17)$$

Ahora podemos implantar eficientemente los accesos al directorio, segmento y bloque expresados. Para ello aislaremos los cálculos en funciones separadas con nombres que

indiquen claramente los cálculos en cuestión:

39a *(miembros privados de DynArray<T> 38b) +≡* (34) ◁38b 39c▷

```

size_t index_in_dir(const size_t & i) const
{
    return i » seg_plus_block_pow;
}
size_t modulus_from_index_in_dir(const size_t & i) const
{
    return (i & mask_seg_plus_block);
}
size_t index_in_seg(const size_t & i) const
{
    return modulus_from_index_in_dir(i) » pow_block;
}
size_t index_in_block(const size_t & i) const
{
    return modulus_from_index_in_dir(i) & mask_block;
}

index_in_dir() calcula el índice dentro del directorio dado el índice del arreglo i mediante la expresión (2.11).
```

index_in_seg() calcula el índice dentro del segmento dado el índice del arreglo i según la (2.15). Esta función requiere el valor *resto* que corresponde a modulus_from_index_in_dir() según (2.13).

index_in_block() calcula el índice dentro del bloque dado el índice i del arreglo según (2.17).

Como las funciones son inline, la secuencia interna de operaciones se expone completamente al optimizador del compilador, el cual es capaz de efectuar muchas optimizaciones, por ejemplo, la de memorizar el resto de la división efectuada en el cálculo del índice del directorio.

39b *(miembros constantes de DynArray<T> 37a) +≡* (34) ◁38a

```

mutable size_t max_dim; // 2^(pow_dir + pow_seg + pow_block) - 1

max_dim almacena la máxima dimensión que puede alcanzar el arreglo según las longitudes del directorio, segmento y bloque.
```

39c *(miembros privados de DynArray<T> 38b) +≡* (34) ◁39a 40a▷

```

size_t current_dim;
size_t num_segs;
size_t num_blocks;

current_dim almacena la dimensión actual. num_segs y num_blocks contabilizan el total de segmentos y de bloques que han sido reservados.
```

2.1.4.2 Manejo de memoria

En el manejo de memoria se deben considerar los siguientes aspectos:

- El valor NULL indica que una entrada de directorio o segmento no tiene memoria apartada. Esto implica que cuando un bloque es liberado, la entrada en el directorio debe ser restaurada al valor NULL.

- Los segmentos y bloques reservados o liberados deben contabilizarse.

La entrada a al directorios, segmento y bloque comienza a partir del directorio, el cual se declara como sigue:

40a *(miembros privados de DynArray<T> 38b) +≡* (34) ◁39c 40b▷
T *** dir;

dir representa el apuntador al directorio.

El manejo de memoria se realiza mediante métodos privados especializados que separan los detalles en puntos únicos, sencillos de depurar y mantener:

40b *(miembros privados de DynArray<T> 38b) +≡* (34) ◁40a 40c▷
void fill_dir_to_null()
{
 for (size_t i = 0; i < dir_size; ++i)
 dir[i] = NULL;
}
void fill_seg_to_null(T ** seg)
{
 for (size_t i = 0; i < seg_size; ++i)
 seg[i] = NULL;
}

`fill_dir_to_null()` asegura que todas las entradas del directorio sean nulas. El valor NULL indica que la entrada del directorio no posee un segmento reservado. Análogamente, un NULL en una entrada de un segmento indicará que la entrada no posee un bloque reservado.

`fill_seg_to_null()` inicializa⁷ todas la entradas del segmento apuntado por `seg` al valor NULL. `fill_dir_to_null()` se llama cuando se crea el directorio y `fill_seg_to_null()` cuando se crea un nuevo segmento.

40c *(miembros privados de DynArray<T> 38b) +≡* (34) ◁40b 41a▷
void allocate_dir()
{
 dir = new T ** [dir_size];
 fill_dir_to_null();
}
void allocate_segment(T **& seg)
{
 seg = new T* [seg_size];
 fill_seg_to_null(seg);
 ++num_segs;
}

`allocate_dir()` reserva memoria para el directorio y fija todas sus entradas en NULL. `allocate_segment()` asigna al puntero `seg` la dirección de memoria de un nuevo segmento cuyas entradas están inicializadas en NULL.

Las entradas del directorio y de los segmentos siempre deben inicializarse en NULL. `dir[i] == NULL` indica que no se ha apartado un segmento, mientras que `dir[i][j] == NULL` indica que no se ha apartado el bloque.

⁷El verbo “inicializar” ya es parte “real” de la lengua española.

En el caso de inicializar un bloque debemos considerar la posibilidad de que el usuario de `DynArray<T>` especifique un valor inicial. Para ello guardaremos en los siguientes campos un valor inicial de `T` de todas las entradas de un bloque:

41a *(miembros privados de DynArray<T> 38b) +≡* (34) ◁40c 41b ▷
 `T default_initial_value;`
 `T * default_initial_value_ptr;`
 `default_initial_value almacena el valor inicial, mientras que`
 `default_initial_value_ptr un puntero a la celda anterior. Por omisión, este`
 `puntero es NULL, de modo tal que, también por omisión, no se realice un inicialización`
 `explícita, sino la implícita subyacente al constructor por omisión T::T(). Por esta`
 `razón, todos los constructores de DynArray<T> inicializan default_initial_value_ptr`
 `en NULL`

41b *(miembros privados de DynArray<T> 38b) +≡* (34) ◁41a 41c ▷
 `void allocate_block(T *& block)`
 `{`
 `block = new T [block_size];`
 `++num_blocks;`
 `if (default_initial_value_ptr == NULL)`
 `return;`
 `for (size_t i = 0; i < block_size; ++i)`
 `block[i] = default_initial_value;`
 `}`

`allocate_block()` asigna al puntero `block` la dirección de memoria de un nuevo bloque. Según se haya o no especificado un valor por omisión, se recorrerá `block` y se le asignará el valor `default_initial_value`, lo que exige la existencia del operador de asignación `T & operator = (const T &)`.

41c *(miembros privados de DynArray<T> 38b) +≡* (34) ◁41b 41d ▷
 `void release_segment(T **& seg)`
 `{`
 `delete [] seg;`
 `seg = NULL;`
 `-num_segs;`
 `}`
 `void release_block(T *& block)`
 `{`
 `delete [] block;`
 `block = NULL;`
 `-num_blocks;`
 `}`

Estos métodos se encargan de liberar consistentemente un segmento o un bloque. Otra manera de liberar involucra un segmento con todos sus bloques, o todos los segmentos y bloques del arreglo o el directorio:

41d *(miembros privados de DynArray<T> 38b) +≡* (34) ◁41c 43b ▷
 `void release_blocks_and_segment(T ** & seg)`
 `{`

```

    for(size_t i = 0; i < seg_size ; ++i)
        if (seg[i] != NULL)
            release_block(seg[i]);

        release_segment(seg);
    }
void release_all_segments_and_blocks()
{
    for(size_t i = 0; i < dir_size ; ++i)
        if (dir[i] != NULL)
            release_blocks_and_segment(dir[i]);

    current_dim = 0;
}
void release_dir()
{
    release_all_segments_and_blocks();
    delete [] dir;
    dir = NULL;
    current_dim = 0;
}

```

2.1.4.3 Especificación e implantación de métodos públicos

El constructor por omisión plantea una sutileza acerca de la selección del tamaño del bloque:

42 ⟨miembros públicos de DynArray<*T*> 42⟩≡ (34) 43c▷

```

DynArray(const size_t & dim = 0)
    : pow_dir          ( Default_Pow_Dir           ),
      pow_seg          ( Default_Pow_Seg          ),
      ⟨Determinación de la potencia de 2 del bloque 43a⟩
      seg_plus_block_pow ( pow_seg + pow_block      ),
      mask_seg_plus_block ( two_raised(seg_plus_block_pow) - 1 ),
      dir_size          ( two_raised(pow_dir)       ),
      seg_size           ( two_raised(pow_seg)       ),
      block_size         ( two_raised(pow_block)      ),
      max_dim            ( two_raised(seg_plus_block_pow + pow_dir) ),
      mask_seg           ( seg_size - 1             ),
      mask_block          ( block_size - 1           ),
      current_dim        ( dim                      ),
      num_segs            ( 0                        ),
      num_blocks          ( 0                        ),
      default_initial_value_ptr (NULL)
    {
        allocate_dir();
    }

```

Uses DynArray 34.

Como se ve, el constructor selecciona una longitud de bloque. Cuanto más grande sea esta longitud, menor será la cantidad de reservaciones de memoria causadas por la expansión del arreglo. Un tamaño de bloque muy pequeño implicaría que nuevos bloques tendrían

que apartarse muy a menudo. En añadidura, la máxima dimensión del arreglo sería muy limitada.

Debemos privilegiar una longitud de bloque más o menos grande. En principio se selecciona una octava parte de la dimensión inicial especificada en el constructor. El único problema ocurre cuando el usuario sugiere una dimensión inicial muy pequeña, lo cual acarrearía un tamaño de bloque muy pequeño. Para evitar esto, seleccionamos una potencia de 2 de base como longitud mínima del bloque. Como se especificó en *(miembros constantes de DynArray<T> 37a)*, tal potencia será el valor de 12, equivalente a 4096 entradas, lo que arroja una dimensión máxima por omisión de $2^4 + 2^6 + 2^{12} = 2^{22} = 4194304$.

En una máquina de 32 bits, `Max_Bits_Allowed = 32`. Entonces, quedarán $32 - 22 - 1$ potencias de dos de más para un bloque antes de llegar a `Max_Bits_Allowed`. La máxima potencia de dos del tamaño de un bloque será:

$$\text{Max_Pow_Block} = 32 - \text{Default_Pow_Dir} - \text{Default_Pow_Seg} - 1 = 21 \quad (2.18)$$

Lo que hace una máxima dimensión direccionable con el constructor por omisión de:

$$2^{\text{Default_Pow_Dir}} \times 2^{\text{Default_Pow_Seg}} \times 2^{21} = 2^4 \times 2^6 \times 2^{21} = 536870912 ,$$

dimensión más que suficiente para la mayoría de los arreglos que quepan en memoria.

La máxima potencia de dos del bloque es almacenada en una constante calculada según (2.18).

Así pues, se debe garantizar que `Default_Pow_Block ≤ pow_block ≤ Max_Pow_Block`. Esta expresión se traduce del siguiente modo:

43a *(Determinación de la potencia de 2 del bloque 43a) ≡* (42)
`pow_block = std::min(Default_Pow_Block, Max_Pow_Block),`

El cálculo de la próxima potencia de dos de un número es realizado por la función estática `next2Pow`:

43b *(miembros privados de DynArray<T> 38b) +≡* (34) ◁ 41d 44a ▷
`static size_t next2Pow(const size_t & number)
{
 return (size_t) ceil(log((float) number) / log(2.0));
}`

El constructor copia y el operador de asignación realizan una operación común: las reservaciones de memoria y copias del directorio, segmentos y bloques del arreglo destino, sea por el constructor o por la asignación. Por ello, encapsulamos esta operación en un método privado común llamado `copy_array()`:

43c *(miembros públicos de DynArray<T> 42) +≡* (34) ◁ 42 45a ▷
`void copy_array(const DynArray<T> & src_array)
{
 for (int i = 0; i < src_array.current_dim; ++i)
 if (src_array.exist(i))
 (*this)[i] = src_array.access(i);
}`

Uses `DynArray 34`.

`copy_array()` hace uso de dos métodos públicos que implantaremos más adelante. `src_array.exist(i)` retorna true si la i -ésima entrada existe en `src_array`; false, de lo contrario. `src_array.access(i)` retorna la i -ésima entrada de `src_array` sin verificar que dicha entrada exista; es decir, que tenga un segmento y un bloque apartados. No hay necesidad de verificar que haya memoria porque esto fue verificado con `src_array.exist(i)`. El operando izquierdo de la asignación, `(*this)[i] = ...` aparta el segmento o el bloque si se requiere.

Notemos que `copy_array()` recorre cada posición de los arreglos. Hay una manera mucho más eficiente, pero más difícil, de implantar que se vale de la copia directa de bloques. La dificultad estriba cuando los arreglos tienen tamaños de segmentos y bloques diferentes. Tal implantación se delega a ejercicio. Como ayuda, se provee el método `advance_block_index()`, el cual, dados los índices del segmento y del bloque, calcula los índices correspondientes a `len` bloques:

44a *(miembros privados de DynArray<T> 38b) +≡* (34) $\triangleleft 43b \ 44b \triangleright$

```

size_t divide_by_block_size(const size_t & number) const
{
    return number >> pow_block;
}
size_t modulus_by_block_size(const size_t & number) const
{
    return number & mask_block;
}
void advance_block_index(size_t & block_index, size_t & seg_index,
                        const size_t & len) const
{
    if (block_index + len < block_size)
    {
        block_index += len;
        return;
    }
    seg_index += divide_by_block_size(len);
    block_index = modulus_by_block_size(len);
}
```

Otra forma de copiar rápidamente requiere asegurar que el arreglo destino tenga exactamente la misma estructura que el del fuente, es decir, los tamaños de directorio, segmento y bloque deben ser idénticos. Bajo esa premisa podemos copiar sólo aquellas entradas que contengan memoria de la siguiente manera:

44b *(miembros privados de DynArray<T> 38b) +≡* (34) $\triangleleft 44a \ 49a \triangleright$

```

void allocate_dir(T *** src_dir)
{
    allocate_dir();
    for (int i = 0; i < dir_size; i++)
        if (src_dir[i] != NULL)
            allocate_segment(dir[i], src_dir[i]);
}
```

El método hace uso de la primitiva `allocate_seg()` en su versión de copia, la cual, a su vez, hace uso de `allocate_block(seg[i], src_seg[i])`. Estas primitivas no están mostradas en este texto, pero son fácilmente

deducibles.

Mediante `allocate_dir()` en su versión de copia podemos ofrecer un modelo de copia más eficiente. Esta forma de copia la llamaremos `copy_array_exactly()` y la definimos bajo el siguiente modo:

45a *(miembros públicos de DynArray<T> 42) +≡* (34) ◁ 43c 45b ▷

```
void copy_array_exactly(const DynArray<T> & array)
{
    release_dir(); // liberar toda la memoria de this
    pow_dir = array.pow_dir;
    // asignar el resto de los atributos ...
    allocate_dir(array.dir);
}
```

Uses DynArray 34.

Mediante `allocate_dir()` se puede implantar eficientemente el constructor copia. La asignación, por el contrario, hace la copia de elemento por elemento, lo que no es lo más eficiente.

El acceso a un elemento de un `DynArray<T>` puede hacerse directamente mediante:

45b *(miembros públicos de DynArray<T> 42) +≡* (34) ◁ 45a 45c ▷

```
T & access(const size_t & i)
{
    return dir[index_in_dir(i)][index_in_seg(i)][index_in_block(i)];
}
```

`access()` no verifica que la memoria del bloque y su segmento haya sido apartada. Este método debe usarse sólo cuando se esté absolutamente seguro de que se ha escrito en la posición de acceso `i`.

Para indagar si una entrada dada puede accederse sin error, es decir, que ésta tenga su bloque y su segmento, se provee:

45c *(miembros públicos de DynArray<T> 42) +≡* (34) ◁ 45b 45d ▷

```
bool exist(const size_t & i) const
{
    const size_t pos_in_dir = index_in_dir(i);
    if (dir[pos_in_dir] == NULL)
        return false;

    const size_t pos_in_seg = index_in_seg(i);
    if (dir[pos_in_dir][pos_in_seg] == NULL)
        return false;

    return true;
}
```

`exist()` retorna `true` si la posición `i` tiene su bloque y segmento; `false`, de lo contrario.

En muchas ocasiones se requiere verificar existencia de una entrada y, si ésta existe, entonces realizar el acceso. Para evitar ahorrar el doble cálculo de índices en directorio, segmento y bloque, podemos exportar la primitiva siguiente:

45d *(miembros públicos de DynArray<T> 42) +≡* (34) ◁ 45c 46a ▷

```
T * test(const size_t & i) const
{
```

```

const size_t pos_in_dir = index_in_dir(i);
if (dir[pos_in_dir] == NULL)
    return NULL;

const size_t pos_in_seg = index_in_seg(i);
if (dir[pos_in_dir][pos_in_seg] == NULL)
    return NULL;

return &dir[index_in_dir(i)][index_in_seg(i)][index_in_block(i)];
}

```

Esta es la manera más rápida posible de acceder a una entrada del arreglo dinámico. Notemos sin embargo que no se hace ninguna verificación. Si la entrada existe, entonces `test()` retorna un puntero a la entrada; `NULL` de lo contrario.

Para apartar explícitamente el bloque y segmento de una posición dentro del arreglo, se provee:

```

46a    <miembros públicos de DynArray<T> 42>+≡          (34) ◁45d 46b▷
        T & touch(const size_t & i)
        {
            const size_t pos_in_dir = index_in_dir(i);
            if (dir[pos_in_dir] == NULL)
                allocate_segment(dir[pos_in_dir]);
            const size_t pos_in_seg = index_in_seg(i);
            if (dir[pos_in_dir][pos_in_seg] == NULL)
                allocate_block(dir[pos_in_dir][pos_in_seg]);
            if (i >= current_dim)
                current_dim = i + 1;
            return dir[pos_in_dir][pos_in_seg][index_in_block(i)];
        }

```

También puede apartarse toda la memoria requerida para garantizar el acceso directo a un rango de posiciones. Esto se efectúa mediante el siguiente método:

```

46b   ⟨miembros públicos de DynArray<T> 42⟩+≡                                (34) ▷46a 47▷
      void reserve(const size_t & l, const size_t & r)
      {
          const size_t first_seg    = index_in_dir(l);
          const size_t last_seg     = index_in_dir(r);
          const size_t first_block  = index_in_seg(l);
          const size_t last_block   = index_in_seg(r);
          for (size_t seg_idx = first_seg; seg_idx <= last_seg; ++seg_idx)
          {
              if (dir[seg_idx] == NULL)
                  allocate_segment(dir[seg_idx]);
          }
          size_t block_idx = (seg_idx == first_seg) ? first_block : 0;
          const size_t final_block =
              (seg_idx == last_seg) ? last_block : seg_size - 1;
          while (block_idx <= final_block)
          {

```

```
    if (dir[seg_idx][block_idx] == NULL)
        allocate_block(dir[seg_idx][block_idx]);

    ++block_idx;
}
} // end for (...)

if (r + 1 > current_dim)
    current_dim = r + 1;
}

reserve() aparta los bloques y segmentos que abarcan el rango de posiciones entre l
```

De la misma manera en que se puede apartar memoria para asegurar acceso a determinadas posiciones, también es plausible indicar que se libere la memoria de un rango de posiciones que el usuario determine que no usará más. El método `cut()` ajusta la dimensión del arreglo a una dimensión inferior llamada `new_dim` y libera toda la memoria entre `new_dim` y la antigua dimensión:

```

47  <miembros públicos de DynArray<T> 42>+≡ (34) ◁46b 49b▷
    void cut(const size_t & new_dim = 0)
    {
        if (new_dim == 0)
        {
            release_all_segments_and_blocks();
            current_dim = 0;
            return;
        }

        const size_t old_dim = current_dim; // antigua dimensión
        // índices cotas bloques
        const int idx_first_seg = index_in_dir(old_dim - 1);
        const int idx_first_block = index_in_seg(old_dim - 1);
        // índices cotas segmentos
        const int idx_last_seg = index_in_dir(new_dim - 1);
        const int idx_last_block = index_in_seg(new_dim - 1);
        for (int idx_seg = index_in_dir(old_dim - 1);
             idx_seg >= idx_last_seg;
             -idx_seg) // recorre descendentemente los segmentos
        {
            if (dir[idx_seg] == NULL) // ¿hay un segmento?
                continue; // no ==> avance al siguiente

            <Liberar descendentemente los bloques del segmento 48a>
            <Liberar el segmento 48e>
        }

        current_dim = new_dim; // actualiza nueva dimensión
    }

```

La liberación de bloques es algo delicada porque hay que manejar casos particulares para el último y primer bloque⁸. `idx_block` lleva el índice del bloque en el segmento actual `idx` `seg`. Para el valor inicial dentro del segmento, hay dos casos posibles:

⁸ La inversión es adrede porque la liberación de bloques se efectúa descendentemente.

1. Si se trata del primer segmento, entonces `idx_block` comienza en `idx_first_block`. Este caso ocurre una sola vez cuando `idx_seg == idx_first_seg`.
 2. `idx_block` comienza en `block_size - 1`. Este es el caso común y se verifica cuando `idx_seg != idx_first_seg`.

El valor inicial se define entonces del siguiente modo:

También hay dos casos posibles para el valor final de `idx_block`:

- Si no nos encontramos en el último segmento (`idx_last_seg`), entonces el último bloque será el de índice 0. La condición de parada será, entonces, la siguiente:

48b $\langle \text{Se esté en bloque de un segmento general } 48b \rangle \equiv$ (48d)
 $(\text{idx_seg} > \text{idx_last_seg} \text{ and } \text{idx_block} \geq 0)$

- nos encontramos en el último segmento, entonces el último bloque sería el `last block`. Esta condición la detectamos mediante el predicado:

El bucle que libera descendente el bloque del segmento actual se define entonces como sigue:

El segmento sólo debe liberarse si todos sus bloques lo han sido. Esto se verifica si ha recorrido completamente el segmento actual:

48e $\langle \text{Liberar el segmento 48e} \rangle \equiv$ (47)
 if (idx_block < 0)
 release segment(dir[idx_seg]);

2.1.4.4 Acceso mediante operador []

Hasta el presente, las interfaces de acceso a los elementos de un arreglo dinámico están diseñadas para efectuar el acceso en tiempo constante de la manera más rápida posible. `access()` no verifica si la memoria ha sido apartada, razón por la cual el usuario debe explícitamente verificarla a través de `exist()` o apartarla mediante `touch()` o `reserve()`.

Una forma más simple y transparente de manipular un arreglo dinámico, en detrimento de un poco de tiempo, es a través del operador `[]`. Inicialmente, el arreglo se construye

con tan solo el directorio apartado. Paulatinamente, a medida que se escriben datos en las posiciones, se apartan, perezosamente, los segmentos y bloques correspondientes a las entradas que han sido escritas. De esta manera, la memoria ocupada por el `DynArray<T>` es proporcional a las entradas escritas y no a su dimensión.

Para implantar esta funcionalidad requerimos distinguir los accesos de lectura de los de escritura. Para eso usaremos la clase mediadora `Proxy`:

49a *(miembros privados de DynArray<T> 38b) +≡* (34) ▷44b
`class Proxy`
`{`
(Miembros dato del proxy 49c)
(Métodos del proxy 50a)
`};`

Esta clase es el valor de retorno del operador `[]` en la clase `DynArray<T>`:

49b *(miembros públicos de DynArray<T> 42) +≡* (34) ▷47 52▷
`Proxy operator [] (const size_t & i)`
`{`
 `return Proxy (const_cast<DynArray<T>&>(*this), i);`
`}`

Uses `DynArray 34`.

Si se efectúa un acceso de lectura fuera de la dimensión actual, entonces se genera `std::length_error`. Si es un acceso de escritura, aun fuera de la dimensión actual, entonces se verifica si existen el bloque y el segmento; si no es el caso, entonces éstos se apartan. Puede generarse `std::bad_alloc` si no hay memoria, o `std::length_error` si el índice `i` es mayor o igual que la máxima dimensión.

La dimensión se ajusta automáticamente según la mayor posición escrita. Si un acceso de lectura a la i -ésima entrada detecta que el bloque no ha sido apartado, entonces se genera la excepción `std::invalid_argument`.

El acceso de lectura a una entrada que no ha sido escrita no siempre causa una excepción, pues la entrada ya puede tener su bloque apartado; empero, en este caso el valor de la lectura es indeterminado.

La idea del operador `[]` es retornar una clase `Proxy` en espera de conocer si el acceso es de lectura o de escritura.

Los miembros datos de `Proxy` mantienen la información suficiente para validar y efectuar el acceso:

49c *(Miembros dato del proxy 49c) ≡* (49a)
`size_t index;`
`size_t pos_in_dir;`
`size_t pos_in_seg;`
`size_t pos_in_block;`
`T **& ref_seg;`
`T * block;`
`DynArray & array;`

Uses `DynArray 34`.

`index` es el índice del elemento que el proxy accede. `pos_in_dir`, `pos_in_seg` y `pos_in_block` son los índices en el directorio, segmento y bloque correspondientes a `index`. Estos valores se calculan durante la construcción del proxy.

`ref_seg` es una referencia a la entrada `pos_in_dir` del directorio. Debe ser una referencia porque ésta es susceptible de modificarse. Dada esta referencia, `ref_seg[pos_in_seg]` denota el apuntador al bloque y `ref_seg[pos_in_seg][pos_in_block]` denota el elemento del arreglo correspondiente al índice `index`. `block` es un apuntador igualado a `ref_seg[pos_in_seg][pos_in_block]`. `array` es una referencia al `DynArray<T>` utilizada para observar y modificar su estado.

(Miembros dato del proxy 49c) se inician en el constructor del proxy como sigue:

50a *(Métodos del proxy 50a)≡* (49a) 50b▷

```
Proxy(DynArray<T>& _array, const size_t & i) :
    index(i), pos_in_dir(_array.index_in_dir(index)),
    pos_in_seg(_array.index_in_seg(index)),
    pos_in_block(_array.index_in_block(index)),
    ref_seg(_array.dir[pos_in_dir]), block (NULL), array(_array)
{
    if (ref_seg != NULL)
        block = ref_seg[pos_in_seg]; // ya existe bloque para entrada i
}
```

Uses `DynArray 34`.

`i` es el índice del `DynArray<T>array` al cual el proxy hace referencia. Las posiciones se calculan según las funciones auxiliares de `DynArray<T>`. Si la referencia `ref_seg` contiene un apuntador válido, entonces `block` se inicializa para apuntar al bloque.

No se efectúa ninguna acción cuando se destruye un proxy.

Un Proxy construido está listo para acceder a una posición dada. Según el contexto de invocación del operador `[]`, el compilador determinará si el acceso es de lectura o de escritura. Cuando ocurra una lectura, el compilador intentará convertir el Proxy al tipo genérico `T`. Tal conversión se define del siguiente modo:

50b *(Métodos del proxy 50a)+≡* (49a) ▷50a 50c▷

```
operator T & () {
{
    return block[pos_in_block];
}
```

`block == NULL` indica que no existe un bloque para el índice `index`. Si no hay bloque, entonces la lectura es inválida y se genera la excepción `std::invalid_argument`. De lo contrario se retorna el elemento referido.

La escritura puede ocurrir cuando se efectúa una asignación, la cual se define para un Proxy de las dos siguientes formas:

50c *(Métodos del proxy 50a)+≡* (49a) ▷50b

```
Proxy & operator = (const T & data)
{
    (Obtener o apartar el segmento y el bloque 51b)
    (Verificar y actualizar la dimensión 51a)
    block[pos_in_block] = data;
    return *this;
}
Proxy & operator = (const Proxy & proxy)
{
    if (proxy.block == NULL) // ¿operando derecho puede leerse?
        throw std::domain_error("right entry has not been still written");
```

```

    ⟨Obtener o apartar el segmento y el bloque 51b⟩
    ⟨Verificar y actualizar la dimensión 51a⟩
block[pos_in_block] = proxy.block[proxy.pos_in_block];
return *this;
}

```

Es decir, asignación del tipo genérico T a un proxy, por ejemplo `array[i] = variable;` o asignación de proxy a proxy, por ejemplo `array[i] = array[j]`. En el último caso se debe verificar que la lectura del operando derecho sea válida, cual es el sentido del primer if.

De resto, las dos asignaciones son muy similares y se remiten a *(Obtener o apartar el segmento y el bloque 51b)* en donde se escribirá y luego a *(Verificar y actualizar la dimensión 51a)*.

La dimensión actual sólo se actualiza si se escribe en una posición mayor que la dimensión actual:

```
if (index >= array.current_dim)
    array.current_dim = index + 1;
```

⟨Obtener o apartar el segmento y el bloque 51b⟩ realiza dos pasos:

```
1. Obtener el segmento:  
  
{Obtener o apartar el segmento y e  
    bool seg_was_allocated_in_current  
    if (ref_seg == NULL) // hay segmento  
    {  
        // No ==> apartarlo!  
        array.allocate_segment(ref_seg);  
        seg_was_allocated_in_current = true;  
    }  
}
```

`seg_was_allocated_in_current_call` memoriza si se aparta memoria para el segmento. Esto permite determinar si hay que liberar o no el segmento en caso de que ocurra una excepción si se aparta el bloque.

2. Obtener bloque:

Si ocurre una falla de memoria durante la reservación del segmento, la excepción `std::bad_alloc`, generada por `new` y regenerada por `allocate_segment()` o `allocate_block()` puede dejarse sin capturar. Si la falla de memoria ocurre durante la reservación del bloque, entonces la excepción `std::bad_alloc`, generada por `new`, debe capturarse para eventualmente poder liberar la memoria del segmento que fue apartada por la llamada actual. Esta es la razón por la cual hay un manejador de excepción para la reservación del bloque.

2.1.4.5 Tratamiento del arreglo como pila o cola

A menudo, un arreglo dinámico se utiliza como “cola” o “pila”; dos ideas que aún no hemos presentado y que estudiaremos ampliamente en § 2.5 (Pág. 98) y § 2.6 (Pág. 122). Debido a su importancia, vale la pena, a efectos de la economía de código, encapsular algunas operaciones:

52 *(miembros públicos de DynArray<T> 42)* +≡ (34) ◁ 49b

```

void append(const T & data) { (*this)[size()] = data; }

void push(const T & data) { append(data); }

T pop()
{
    T ret_val = (*this)[size() - 1];
    cut(size() - 1);
    return ret_val;
}

T & top() { return (*this)[size() - 1]; }

T & get_first() { return top(); }

T & get_last() { return (*this)[0]; }

```

2.1.4.6 Uso del valor por omisión

Hemos recalculado el hecho de que la memoria de un *DynArray<T>* se aparta perezosamente en tiempo de escritura. También hemos destacado que esta característica permite ahorrar memoria en aplicaciones que usen arreglos esparcidos.

¿Qué sucede cuando se accede a una entrada que no ha sido escrita?. Para estudiar esta cuestión, supongamos un acceso de lectura a una *i*-ésima entrada; ante este evento podemos plantear los siguientes escenarios:

1. La *i*-ésima entrada ha sido previa y explícitamente escrita, en cuyo caso se retornará el último valor escrito.
2. La entrada no ha sido previamente escrita. En esta situación pueden ocurrir dos cosas:
 - (a) Que no se haya apartado un bloque para la *i*-ésima entrada, en cuyo caso se trata de un error de programación (lectura de un valor que no ha sido escrito), el cual, en el mejor de los escenarios, generará una excepción. Esta sería la situación si el acceso se efectúa mediante el operador `[]`. Si el acceso se hace mediante `access()`, entonces, con suerte, ocurrirá la excepción sistema `segmentation fault`. Por el contrario, con mala suerte no ocurrirá ninguna excepción, lo que tenderá a ocultar el error.
 - (b) Que sí se haya apartado el bloque durante la escritura de alguna entrada en el mismo bloque de la *i*-ésima entrada. En este caso se retornaría el valor que haya escrito el constructor por omisión `T::T()`. Notemos la posibilidad, no tan excepcional, de que este constructor por omisión no registre ninguna escritura, en cuyo caso, el valor de retorno puede ser cualquiera, según el estado del bloque.

El TAD DynArray<T> puede ser útil para modelizar arreglos esparcidos; por ejemplo, una matriz esparcida que contiene muchos ceros. En este caso podemos usar un DynArray<T> mat[n*n]. El acceso a una entrada de la matriz puede hacerse mediante el operador (i, j) o mediante el uso de una clase proxy. Una pseudoimplantación sugerida del acceso de lectura a un TAD Matriz<T> es como sigue:

53

```
acceso matriz 53>≡
    template <typename T>
    const T & Matriz<T>::operator () (long i, long j) const
    {
        const int & n = mat.size();
        const long index_dynarray = i + j*n; // posición dentro de mat
        if (not mat.exist(index_dynarray))
            return Zero_Value;
        return mat.access(index_dynarray);
    }
```

Lo guardado en la constante index_dynarray se corresponde con considerar a una matriz como un arreglo de arreglos⁹. Si el predicado not mat.exist(index) es cierto, entonces no hay memoria apartada para el bloque que contendría la entrada (i, j), razón por la cual se retorna un valor por omisión denominado en este caso Zero_Value; de lo contrario retornamos mat.access(index), pues el acceso está garantizado si el flujo llega a esa línea.

Puede presentarse la situación descrita en 2b, es decir, mat.access(index) no ha sido previamente escrito, pero not mat.exist(index) == true. El valor de retorno será entonces el especificado por T::T() o el asignado en una llamada a set_default_initial_value().

set_default_initial_value() permite especificar diversos valores iniciales para el mismo tipo, lo cual será muy útil para campos algebraicos diferentes que manipulen el mismo tipo de operando; las matrices de adyacencia en los grafos, por ejemplo.

2.1.5 Arreglos de bits

Recordemos que un dato de tipo lógico, bool en C++ sólo puede tomar dos valores: true o false. Para representar un bool sólo se requiere un bit, pero, por diversas razones, por moda, la alineación de memoria, en C++ un bool es traducido a un byte cuyo valor puede ser cero (0) o uno (1); desperdicio de siete (7) bits despreciable en la inmensa mayoría de las aplicaciones, pues la cantidad de datos lógicos es pequeña.

Ahora bien, ¿qué sucede si tenemos un arreglo de datos lógicos? Si la dimensión del arreglo es considerable, entonces podemos incurrir en un desperdicio de memoria importante. Pero, ¿existen aplicaciones que requieran arreglos lógicos? La respuesta es afirmativa y cabe decir que se presentan con cierta frecuencia.

Un ejemplo es el manejo de páginas de memoria virtual efectuado por un sistema operativo. Consideremos una memoria principal de 512 Mb y páginas de 4 Kb; un sistema operativo debe mantener el estado de $\frac{512 \times 1024 \text{ Kb}}{4 \text{ Kb}} = 131072$ páginas. Si cada estado se representa con un byte, entonces se requieren 128 Kb de memoria para almacenar el estado de las 131072 páginas. Por cada página, el sistema operativo maneja dos bits. El primero indica si la página se encuentra o no en memoria principal. El segundo indica si una página ha sido o no modificada. Si se utilizan dos bits por página, se requieren

$$\frac{2 \times 131072}{8} = 32768 = 32 \text{ Kb};$$

⁹Véase § 2.2 (Pág. 58).

que, como se ve, representa un ahorro de memoria significativo.

Para manejar arreglos de bits introduciremos el TAD BitArray, el cual modeliza vectores de bits. Este TAD está especificado e implantado en el archivo *<bitArray.H>* 54a):

54a *<bitArray.H>* 54a) \equiv
(clase Byte 54c)
 class BitArray
{
(Miembros privados de BitArray 54b)
(Miembros públicos de BitArray 55c)
};

Defines:

BitArray, used in chunks 55–57, 311–13, 349, 351, 353b, 354, and 633a.

La base del arreglo de bytes será almacenada en:

54b *(Miembros privados de BitArray 54b)* 54a) \equiv (54a) 56a) \triangleright
 size_t current_size;
 DynArray<Byte> array_of_bytes;

Uses DynArray 34.

Por razones de versatilidad, la muy apreciada simplicidad, entre ellas, BitArray se fundamenta sobre un arreglo dinámico.

El “truco” para realizar una implantación sencilla es disponer de un soporte que manipule los bits dentro de un byte. Dicho soporte está dado por la clase Byte, la cual se define así:

54c *(clase Byte 54c)* 54a) \equiv (54a)
 class Byte
{
 unsigned int b0 : 1;
// ...
(Lectura de bit 54d)
(Escritura de bit 55a)
(Constructor de Byte 55b)
};

Básicamente, la clase BitArray define una máscara de 8 bits mediante campos bits¹⁰. Esta máscara permite un acceso sencillo dado un índice de bit. La lectura es, entonces, implantada como sigue:

54d *(Lectura de bit 54d)* 54c) \equiv (54c)
 unsigned int read_bit(const unsigned int & i) const
{
 switch (i)
{
 case 0 : return b0;
 case 1 : return b1;
// ...
 case 7 : return b7;
}
}

¹⁰Los campos bits fueron originalmente definidos para el lenguaje C y son parte del C++

Donde i es el índice del bit que se desea leer.

La escritura es implantada como sigue:

55a $\langle\text{Escritura de bit 55a}\rangle \equiv$ (54c)

```
void write_bit(const unsigned int & i, const unsigned int & value)
{
    switch (i)
    {
        case 0 : b0 = value; break;
        case 1 : b1 = value; break;
        // ...
        case 7 : b7 = value; break;
    }
}
```

Donde i es el índice del bit que se desea acceder y $value$ es el bit que se desea escribir.

Es muy útil, a expensas de un poco de tiempo, que cuando se aparte la memoria para un objeto de tipo `Byte` todos sus bits estén iniciados en cero. Implantaremos esta semántica en el constructor por omisión:

55b $\langle\text{Constructor de Byte 55b}\rangle \equiv$ (54c)

```
Byte() : b0(0), b1(0), b2(0), b3(0), b4(0), b5(0), b6(0), b7(0) {}
```

Este constructor garantiza que todo arreglo de tipo `Byte` tenga todos sus bits en cero.

Para declarar un objeto de tipo `BitArray` basta con especificar la dimensión del arreglo. El constructor y el destructor tienen, pues, la siguiente forma:

55c $\langle\text{Miembros públicos de BitArray 55c}\rangle \equiv$ (54a) 55d>

```
BitArray(const size_t & dim = 0)
    : current_size(dim), array_of_bytes(get_num_bytes())
{
    array_of_bytes.set_default_initial_value(Byte());
}
```

Uses `BitArray 54a`.

Para conocer la dimensión del arreglo, se puede emplear el método:

55d $\langle\text{Miembros públicos de BitArray 55c}\rangle + \equiv$ (54a) <55c 55e>

```
const size_t & size() const { return current_size; }
```

Podemos cambiar la dimensión del arreglo mediante:

55e $\langle\text{Miembros públicos de BitArray 55c}\rangle + \equiv$ (54a) <55d 56b>

```
void set_size(const size_t & sz)
{
    array_of_bytes.cut();
    current_size = sz;
}
```

El acceso a los elementos del arreglo se realiza mediante el operador `[]`, el cual es complicado de especificar y de implantar porque éste debe distinguir los accesos de lectura de los de escritura.

Dado el índice i de un bit, la posición `byte_index` dentro de `array_of_bytes` se calcula como sigue:

$$\text{byte_index} = \left\lfloor \frac{i}{8} \right\rfloor .$$

Su posición bit_index dentro del byte estará dada por:

$$\text{bit_index} = i \bmod 8 .$$

Consideremos la instrucción `i = array[40]`; que leerá el valor del bit correspondiente a la entrada 40. En este caso, la implantación del operador `[]` debe buscar el bit 40 y entonces retornar una representación entera -de tipo int- del bit accedido. En este caso, el operador `[]` debe invocar a `read_bit()`.

Ahora consideremos la instrucción `array[40] = i`; que escribirá en la entrada 40 del arreglo el valor contenido en la variable `i`. En este caso, el acceso debe localizar el bit 40 en el arreglo y escribir, no leer, el valor de `i`. En este caso, el operador `[]` debe invocar a `write_bit()`.

La implantación del operador `[]` requiere, entonces, realizar operaciones diferentes según el tipo de acceso: lectura o escritura. Esto no se puede realizar en la implantación de alguna de las declaraciones tradicionales del operador `[]`, pero sí se puede postergar la distinción hasta algún momento en que sea posible conocer si el acceso es de lectura o de escritura. Tal distinción se realiza mediante una combinación del operador `[]` y de una clase especial denominada "proxy", la cual se define como sigue:

56a *(Miembros privados de BitArray 54b) +≡* (54a) ◁ 54b
class BitProxy
`{`
(miembros privados de BitProxy 56c)
(miembros públicos de BitProxy 57a)
`};`

El operador `[]` de `BitArray` retorna un `BitProxy` así:

56b *(Miembros públicos de BitArray 55c) +≡* (54a) ◁ 55e 58a ▷
`BitProxy operator [] (const size_t & i) const`
`{`
 `return BitProxy(const_cast<BitArray&>(*this), i);`
`}`
`BitProxy operator [] (const size_t & i) { return BitProxy(*this, i); }`
Uses BitArray 54a.

El primer operador se invoca para un `BitArray` constante (declarado `const`), mientras que el segundo para un `BitArray` que no es constante.

El operador de asignación y el constructor copia de `BitProxy` implantan la escritura, mientras que el operador de conversión de tipo implanta la lectura.

Cualquiera que sea el tipo de acceso, previamente es necesario calcular el índice del byte dentro de `array_of_bytes` y el índice del bit dentro del byte. Esta información la guardamos en los siguientes atributos:

56c *(miembros privados de BitProxy 56c) ≡* (56a)
`const size_t index;`
`const size_t bit_index;`
`const size_t byte_index;`
`BitArray * array;`
`Byte * byte_ptr;`
Uses BitArray 54a.

Tales atributos se inician en el constructor de BitProxy:

57a *(miembros públicos de BitProxy 57a)≡* (56a) 57b▷

```
BitProxy(BitArray & a, const size_t & i) :
    index(i), bit_index(i % 8), byte_index(i/8), array(&a)
{
    if (array->array_of_bytes.exist(byte_index))
        byte_ptr = &array->array_of_bytes.access(byte_index);
    else
        byte_ptr = NULL;
}
```

Uses BitArray 54a.

Cuando se instancia una clase BitProxy, es decir, cuando se invoca al operador [] de BitArray, el campo byte_ptr apunta al byte dentro de array_of_bytes que contiene el bit que se desea acceder. bit_index contiene el índice del bit dentro del byte *byte_ptr. Si el índice i está fuera de rango, entonces el constructor dispara la excepción std::out_of_range().

Nótese que al momento de construcción de un BitProxy aún no se ha efectuado el acceso, pero sí se ha calculado la dirección del byte dónde se encuentra el bit (byte_ptr) y se ha determinado cuál es el bit dentro de ese byte (bit_index).

Según el contexto de la expresión que invoque al operador [], el compilador distingue si el acceso es de lectura o de escritura. Si es de lectura, entonces el compilador tratará de convertir el BitProxy a un entero resultante de la lectura:

57b *(miembros públicos de BitProxy 57a)+≡* (56a) ◁57a 57c▷

```
operator int () const
{
    return byte_ptr != NULL ? byte_ptr->read_bit(bit_index) : 0;
}
```

Cuando el compilador encuentra una expresión como i = array[40], se busca un constructor de int con parámetro BitProxy. Si no, el compilador busca una asignación de BitProxy a int. Como no existe ninguno de estos operadores, el compilador convierte el proxy a un int.

En la expresión array[40] = i, basta con definir un operador de asignación de proxy a partir de un entero, tal como sigue:

57c *(miembros públicos de BitProxy 57a)+≡* (56a) ◁57b

```
BitProxy & operator = (const size_t & value)
{
    if (byte_ptr == NULL)
        byte_ptr = &array->array_of_bytes.touch(byte_index);

    if (index >= array->current_size)
        array->current_size = index;

    byte_ptr->write_bit(bit_index, value);

    return *this;
}
```

La clase BitProxy permite manejar un arreglo de bits casi exactamente de la misma manera que se usan los arreglos. Sin embargo, esta técnica no garantiza que siempre sea

posible usar el operador `[]`, ni es la de más alto desempeño. Por esa razón exportamos dos operaciones explícitas de lectura y de escritura:

58a *(Miembros públicos de BitArray 55c) +≡* (54a) ◁56b 58b ▷

```

int read_bit(const size_t & i) const
{
    const int bit_index = i % 8;
    const Byte byte = array_of_bytes[i/8];
    return byte.read_bit(bit_index);
}
void write_bit(const size_t & i, const unsigned int & value)
{
    array_of_bytes.touch(i/8).write_bit(i%8, value);
    if (i >= current_size)
        current_size = i + 1;
}
```

Puesto que un arreglo de bits es dinámico, podemos alargarlo o cortarlo, al estilo de una pila¹¹. Para ello proveemos las siguientes funciones:

58b *(Miembros públicos de BitArray 55c) +≡* (54a) ◁58a

```

void push(const unsigned int & value)
{
    write_bit(current_size, value);
}
void pop()
{
    current_size--;
    array_of_bytes.cut(get_num_bytes());
}
void empty()
{
    current_size = 0;
    array_of_bytes.cut();
}
```

Estas operaciones cambian la dimensión del arreglo.

2.2 Arreglos multidimensionales

En una muy cierta medida, los arreglos son reminiscentes a los vectores de la matemática. En este sentido es posible representar una matriz $n \times m$ mediante un arreglo de n de arreglos de dimensión m , es decir, una secuencia de secuencias.

En C++ una matriz de este tipo se declara del siguiente modo:

```
T matriz[n][m];
```

El lenguaje implanta el acceso a una entrada de la matriz mediante combinaciones del operador `*`. Por ejemplo, la expresión:

```
matriz[i][j] = dato;
```

Asigna dato al j -ésimo elemento de la i -ésima fila.

¹¹El concepto de pila será tratado ampliamente en § 2.5 (Pág. 98).

El compilador genera automáticamente el cálculo de la dirección de acceso, el cual se puede enunciar de la siguiente forma:

$$\text{Dirección del elemento} = \text{base} + i \times m \times \text{sizeof}(T) + j \times \text{sizeof}(T) \quad (2.19)$$

En la declaración anterior, el compilador debe buscar un espacio contiguo lo suficientemente grande para albergar los $n \times m$ elementos de la matriz. Conforme aumentan n y m , disminuye la posibilidad de encontrar tal espacio.

Por otra parte, en C++, la matriz anterior es estática en el sentido de que no es posible cambiar su dimensión. El cálculo (2.19) generado por el compilador asume que n y m permanecen constantes durante la vida de la matriz. Si bien esto no es tan problemático, pues se podría generar código para n y m variables, un cambio en alguna de las dimensiones obliga a desplazar la mayoría de los elementos.

Por las razones anteriores es preferible tratar explícitamente a un arreglo multidimensional como lo que es: una secuencia de secuencias. Bajo esta perspectiva, una matriz $n \times m$ puede declararse como sigue:

```
T ** matriz = new T * [n]; // Aparta arreglo de punteros a filas
for (int i = 0; i < n; ++i)
    matriz[i] = new T [m]; // aparta la fila matriz[i]
```

Este esquema es de entrada más dinámico que el anterior. Si se cambia el número de filas n , entonces sólo es necesario relocalizar el arreglo matriz. Si, por el contrario, se modifica el número de columnas, entonces hay que relocalizar cada fila matriz[i].

En C++, así como en la mayoría de los lenguajes, una matriz se considera como un arreglo de n filas de tamaño m . Sin embargo, también puede interpretarse al revés; es decir, como un arreglo de m columnas de tamaño n . Este es el caso del lenguaje de programación Fortran, intensivamente usado para cálculos numéricos.

Aunque discutible, la disposición especial de las matrices en Fortran no es un capricho. Sigue que muchas operaciones matemáticas sobre matrices exhiben un patrón de acceso que favorece la secuencialización por columnas en lugar de por filas.

Cuenta habida de la popularidad del Fortran, los programadores de C y C++ deben considerar variantes para los arreglos adecuadas al Fortran, pues es bastante factible utilizar bibliotecas escritas en Fortran y viceversa.

2.3 Iteradores

En programación, dicen que al fin de cuentas no quedan sino secuencias. Cualquiera que sea la estructura de datos, el “quid” de procesamiento siempre se remite a secuencias. El patrón es tan frecuente que merece un patrón genérico denominado `Iterator<Set, T>` y especificado en el diagrama UML de la figura 2.2.

La clase `Iterator<Set, T>` recibe dos tipos parametrizados `Set<T>` del tipo explicado en § 1.3 (Pág. 17) y `T`. Su fin es facilitar el procesamiento secuencial de elementos en un conjunto `Set<T>`.

Para usar un iterador se requiere un conjunto o contenedor `Set<T>`. Hay dos formas de instanciarlo: mediante el constructor que reciba el conjunto o mediante el constructor copia. En el primero, el iterador se posiciona en el primer elemento de la secuencia, mientras que en el segundo lo hace en el elemento actual del iterador fuente de la copia.

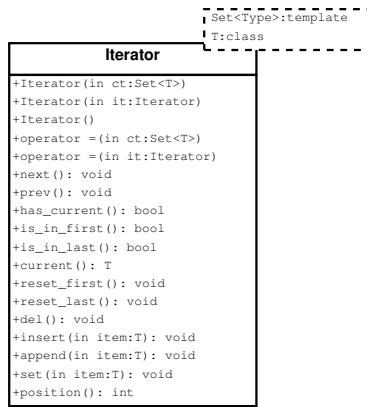


Figura 2.2: Diagrama UML de la clase genérica `Iterator<Set, T>`

Pictóricamente, un conjunto puede interpretarse como un secuencia genérica de n elementos de algún tipo genérico T según la figura 2.3

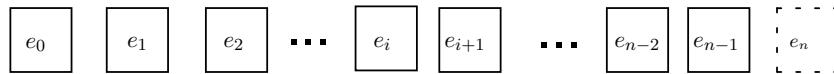


Figura 2.3: Representación pictórica de un iterador sobre una secuencia

Cuando se instancia un iterador con un contenedor `Set<T>`, éste queda posicionado en el primer elemento de la secuencia designado en el dibujo como e_0 .

El elemento designado como e_n no existe en la secuencia, pero éste, lógicamente, denota que el iterador está “desbordado” o “fuera de rango”. La primitiva `has_current()` retorna `false` si el iterador se encuentra posicionado en el elemento ficticio e_n ; `true` de lo contrario.

En algunas situaciones puede convenir saber si el iterador está posicionado en el primer o último elemento. Si el iterador se encontrase en el primer elemento, entonces `is_in_first()` retornaría `true`; la misma semántica se aplicaría con `is_in_last()` si el iterador se encontrase en el último elemento.

El elemento actual del iterador se consulta mediante el método `get_current()`. En el caso general, según la implantación de `Set<T>`, el iterador genera la excepción `std::overflow_error` si se intenta acceder a un iterador desbordado.

Para salvaguardar el estado de un iterador pueden construirse iteradores sin que se proporcione el conjunto. Obviamente, un iterador sin conjunto no es válido hasta que éste no se defina mediante alguna de las dos versiones del operador de asignación.

Para “avanzar” el iterador hacia la “derecha” o hacia “adelante” se ejecuta el método `next()`. Análogamente, para retrocederlo hacia la “izquierda” o hacia “atrás” se ejecuta `prev()`. Cualquiera de estos métodos puede generar las excepciones `std::overflow_error` o `std::underflow_error` si el iterador está desbordado.

En *ALÉPH*, así como en otros ámbitos, un conjunto contiene su clase `Iterator`. Es decir, `Iterator` es una subclase de la especialización de `Set<T, Compare>`.

El siguiente código ilustra una búsqueda genérica en un conjunto mediante un iterador:

```

template <template <class> class Set, typename T, class Compare>
T * search(Set<T> & set, const T & item)

```

```

{
    for (typename Set<T>::Iterator it(set); it.has_current(); it.next())
        if (are_equals<T, Compare>() (it.get_current(), item))
            return &it.get_current();

    return NULL;
}

```

A veces, un iterador puede “reusarse”. Para ello es posible reiniciarlo a fin de que éste apunte al primer o último elemento del conjunto. `reset_first()` reinicia el iterador al primer elemento, mientras que `reset_last()` lo reinicia al último (indicado con e_{n-1} en la figura 2.3).

Las operaciones que siguen son dependientes de la estructura de datos con que se implante el conjunto.

El método `del()` elimina del conjunto el elemento actual y avanza el iterador una posición hacia delante. Debe tenerse especial cuidado cuando se intercalen `next()` y `del()` en una misma iteración.

El método `insert()` inserta un nuevo elemento en la secuencia “a la derecha” del elemento actual del iterador sin avanzar. Análogamente, el método `append()` inserta hacia la izquierda.

La primitiva `set()` posiciona un iterador en el elemento `item` perteneciente al conjunto. Por lo general, no se valida si `item` en efecto es parte del conjunto.

`position()` retorna la posición del elemento actual dentro de la secuencia.

Según la implantación de `Set<T>`, el iterador puede enriquecerse con otras primitivas. Posiblemente la más importante de ellas es el operador de acceso `[]`, el cual retorna un elemento del conjunto según su posición dentro de la secuencia.

Los iteradores pueden usarse genéricamente, es decir, pueden haber programas que reciban iteradores sin tener conocimiento del tipo de conjunto sobre el cual iteran. Para estas situaciones puede ser deseable tener el tipo del conjunto y el de sus elementos. Para ello se exportan los sinónimos de tipo de `Set_Type` y `Item_Type` respectivamente. `Set_Type` proporciona el tipo de conjunto con que normalmente se construye el iterador, mientras que `Item_Type` el tipo que retorna `get_current()`.

2.4 Listas enlazadas

Una lista es una secuencia de longitud “variable” $\langle t_1, t_2, t_3, \dots, t_n \rangle$ de n elementos de algún tipo T con las siguientes propiedades:

Restricción de acceso:

1. Al primer elemento, t_1 , se accede directamente en tiempo constante.
2. Para acceder al elemento T_i es necesario acceder secuencialmente los $i - 1$ previos elementos $\langle t_1, t_2, t_3, \dots, t_{i-1} \rangle$.

Esta restricción se impone sobre todas las operaciones que requieran acceder cualquier posición de la secuencia. En consecuencia, el tiempo de ejecución en el acceso, inserción y eliminación es dependiente de la posición dentro de la lista.

Flexibilidad: Si se conoce la dirección de un elemento t_i dentro de la secuencia $\langle t_1, t_2, \dots, t_i, t_{i+1}, \dots, t_n \rangle$, entonces:

1. t_{i+1} puede eliminarse en tiempo constante. El estado después de esta eliminación es $\langle t_1, t_2, \dots, t_i, \dots, t_n \rangle$.
2. Un elemento cualquiera t_j puede insertarse después de t_i en tiempo constante. El estado final después de la inserción es:

$$\langle t_1, t_2, \dots, t_{i-1}, t_i, t_j, t_{i+1}, \dots, t_n \rangle$$

Espacio proporcional al tamaño: El espacio ocupado por los elementos de la lista siempre es proporcional al número de elementos, es decir, puede expresarse como una función $f(n) = k \times n$.

Las listas son idóneas para aplicaciones con, o algunas de, las siguientes características:

- Se requieren varios conjuntos (quizá muchos) y su cantidad es variable a lo largo de la aplicación.
- Las cardinalidades de los conjuntos son desconocidas y muy variables en el tiempo de vida de la aplicación.
- El procesamiento es secuencial, es decir, el procesamiento del i -ésimo elemento se efectúa después de procesar los $(i - 1)$ elementos previos.

Listas con las características señaladas forman parte de los lenguajes de programación funcionales, LISP, ML, CAML, por ejemplos; y de los modernos y potentes lenguajes de “scripting”; en las ocurrencias más populares, perl y python. Lenguajes de mediano nivel, tales como Pascal, C o C++, no poseen listas como parte del lenguaje, razón por la cual es conveniente implantarlas general y genéricamente en biblioteca.

A las listas se les tilda de “enlazadas” porque a diferencia del arreglo, cada elemento de la secuencia reside en una dirección de memoria que no es contigua. Para conocer cuál es el siguiente elemento de la secuencia, es necesario un “enlace” al bloque de memoria que lo alberga. La figura 2.4 muestra la clásica representación pictórica de una lista simplemente enlazada cuyos elementos son las letras desde la A hasta la E.

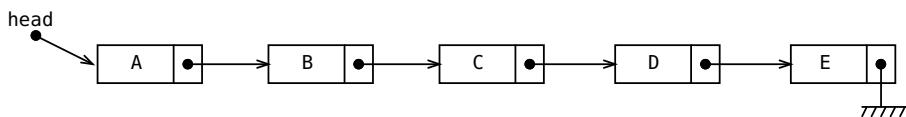


Figura 2.4: Una lista simplemente enlazada

Cada elemento de la secuencia se representa mediante una “caja” dividida en dos campos: el elemento de la secuencia y el apuntador a la próxima caja. Una caja se denomina “nodo”.

El punto de entrada a una lista es la dirección del primer nodo, denominado `head` en la figura 2.4.

En C o C⁺⁺, un nodo puede especificarse como sigue:

63 *(Definición de nodo simple 63)≡*

```
struct Node
{
    char data;
    Node * next;
};
```

Esta es la típica declaración de un nodo perteneciente a una lista enlazada. data almacena la letra y next es el puntero al próximo elemento. Puesto que un Node se refiere a sí mismo a través de next, las listas enlazadas han sido clasificadas de “estructuras recurrentes”.

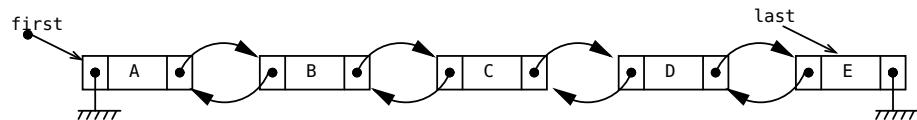


Figura 2.5: Una lista doblemente enlazada

La lista de la figura 2.4 se categoriza como “simplemente enlazada” porque sus nodos poseen un solo enlace hacia el siguiente nodo en la secuencia. En detrimento de un poquito más de memoria, es posible utilizar un enlace adicional al elemento predecesor. En este caso, la lista se categoriza como “dblemente enlazada” y se representa pictóricamente como en la figura 2.5. A efectos de manejar el otro extremo se requiere un puntero adicional al último elemento de la lista.

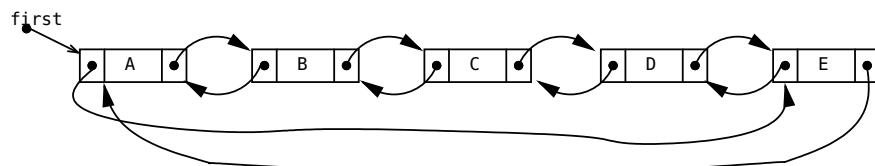


Figura 2.6: Una lista doblemente enlazada circular

El costo en espacio de una lista doblemente enlazada se recompensa con creces con su versatilidad. Dado un elemento perteneciente a la lista, las operaciones pueden referir al elemento predecesor o al sucesor. Consecuentemente, la lista puede recorrerse en los dos sentidos, de izquierda a derecha o viceversa.

Quizá la gran virtud de una lista doblemente enlazada es la capacidad de “autoeliminación”. Cualquier nodo posee todo el contexto necesario (predecesor y sucesor) para él mismo eliminarse de la lista.

En la figura 2.5 se usan dos puntos de entrada, uno por cada extremo. Es posible y más versátil si se cierran los enlaces y se hace a la lista “circular”, tal como ilustra la figura 2.6. En este caso, basta con mantener un puntero al primer nodo de la lista para tener acceso a su otro extremo.

En añadidura, tal como lo estudiaremos posteriormente, si se pone como cabecera un nodo adicional, el cual no forma parte lógica de la secuencia, entonces se gana en linealidad de flujo, pues no hay que considerar los casos particulares cuando la lista esté o devenga vacía.

Las técnicas de circularidad y de nodo cabecera también pueden aplicarse a listas simplemente enlazadas, aunque la ganancia en versatilidad no es tan notable como con las doblemente enlazadas.

2.4.1 Listas enlazadas y el principio fin a fin

Las operaciones sobre listas enlazadas pueden separarse en los niveles jerárquicos, según el tipo de operaciones, ilustrados en el diagrama UML de las clases que manipulan listas en *ALÉPH*.

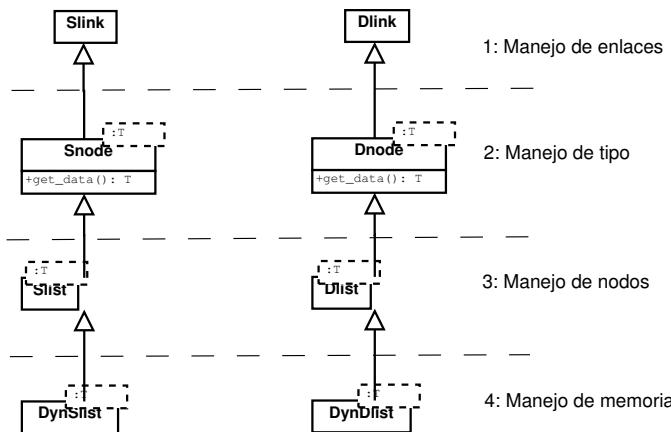


Figura 2.7: Diagrama general UML de las clases para listas enlazadas

En el primer nivel, el fin es el manejo de apuntadores. Sólo nos conciernen las operaciones de enlace de un nodo con otro sin considerar, el resto de los nodos ni el tipo de dato que éstos contienen. En este nivel se encuentran las clases Slink y Dlink, las cuales modelizan enlaces de nodos de listas simple y doblemente enlazadas respectivamente. Las operaciones en este nivel son completamente independientes del tipo de dato que alberguen los nodos, lo que las hace generales para todas las clases de listas.

En el segundo nivel el fin es albergar un dato genérico en el nodo que luego resulte en el tipo de elemento de la secuencia. En este nivel encontramos las clases Snode<T> y Dnode<T> respectivamente. El fin de estas clases es manejar un dato genérico T asequible mediante la operación `get_data()`. Nótese que por herencia pública, un Snode<T> es un Slink y un Dnode<T> es un Dlink.

En el tercer nivel, el fin es manipular listas de nodos, sean simples de tipo Slist<T> o dobles de tipo Dlist<T>. Nótese que en este nivel se habla de listas de nodos y no de listas de elementos. Cada nodo de una lista requiere apartar un espacio de memoria. En lenguajes como C++, el manejo de memoria es delicado porque hay que asegurarse de liberar cada bloque, en nuestro caso, cada nodo, que haya sido apartado una vez que se determina que éste ya no se usará más. En añadidura, hay situaciones en las cuales no es necesario apartar o liberar memoria. Otra ventaja de este enfoque es que podemos eliminar un nodo de un conjunto e insertarlo en otro sin necesidad de liberar y luego apartar memoria. En virtud de estas razones, el manejo por nodos permite especificar operaciones sobre listas sin considerar el manejo de la memoria.

Finalmente, el último nivel corresponde al concepto tradicional de lista, en el cual, todo está resuelto; incluido el manejo de memoria. En este sentido se exportan las clases

`DynSlist<T>` y `DynDlist<T>`, la cuales modelizan listas implantadas mediante enlaces simples y dobles respectivamente

2.4.2 El TAD Slink (enlace simple)

El TAD Slink, definido en el archivo `<slink.H 65a>`, representa un enlace simple de una lista simplemente enlazada circular con nodo cabecera.

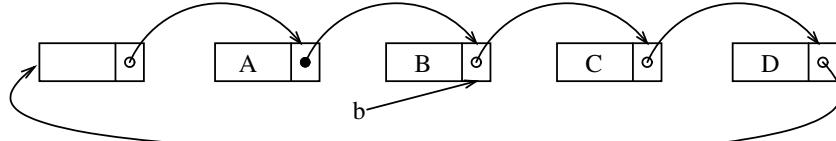


Figura 2.8: Lista enlazada simple con Slink

`<slink.H 65a>` exporta la clase Slink:

65a `<slink.H 65a>≡`
`class Slink`
`{`
`⟨miembros protegidos de Slink 65b⟩`
`⟨miembros públicos de Slink 65c⟩`
`};`
`⟨función de conversión de Slink a clase 68a⟩`

Defines:

Slink, used in chunks 65, 66, 68, 69, and 72.

Objetos que deban enlazarse en una lista simple deben tener un Slink como atributo o ser Slink por herencia pública. La figura 2.8 muestra una lista de cuatro elementos implantada mediante Slink.

Hay una diferencia inmediatamente apreciable con la figura 2.4: cada lazo apunta al lazo del siguiente nodo y no al nodo. Esto plantea una diferencia substancial con las listas enlazadas tradicionales, en las cuales el apuntador direcciona al registro completo.

Slink tiene un solo miembro dato y protegido:

65b `⟨miembros protegidos de Slink 65b⟩≡` (65a)
`protected:`
`Slink* next;`

Uses Slink 65a.

La protección permite que una clase derivada de Slink utilice directamente el miembro dato next.

El TAD Slink es tan simple que muchos de sus métodos conllevan una sola línea:

65c `⟨miembros públicos de Slink 65c⟩≡` (65a) 66a▷
`Slink() : next(this) { /* Empty */ }`

`Slink(const Slink &)`
`{`
`next = this;`
`}`
`Slink & operator = (const Slink & link)`

```

{
    next = this;
}
void reset()
{
    next = this;
}
bool is_empty() const
{
    return next == this;
}

```

Uses Slink 65a.

El constructor Slink() crea un lazo simple apuntando a sí mismo. La copia de un Slink sólo es permitida si el lazo no está incluido dentro de alguna lista; un Slink sólo puede copiarse si éste está vacío.

`is_empty()` retorna verdadero si el `next` apunta a sí mismo. La idea de este método es invocarlo desde un nodo cabecera.

`get_next()` retorna el siguiente lazo atado a `this`.

La inserción es respecto al sucesor y es como sigue:

66a *(miembros públicos de Slink 65c) +≡* (65a) ◁65c 66b ▷

```

void insert_next(Slink * p)
{
    p->next = next;
    next = p;
}

```

Uses Slink 65a.

`insert_next()` inserta el lazo `p` después de `this`.

Del mismo modo, la eliminación ataña al siguiente elemento y también es muy sencilla:

66b *(miembros públicos de Slink 65c) +≡* (65a) ◁66a

```

Slink * remove_next()
{
    Slink * ret_val = next;
    next = ret_val->next;
    ret_val->reset();
    return ret_val;
}

```

Uses Slink 65a.

`remove_next()` suprime el siguiente elemento respecto a `this` y retorna el lazo eliminado.

Supongamos que deseamos enlazar objetos de tipo `Window` en una lista simplemente enlazada. Hay dos formas de hacerlo. La primera es por derivación:

```

class Window : public slink
{
    ...
};

```

En este caso podemos insertar directamente instancias de `Window` después de un nodo particular `first`:

```
Window * window_ptr = new Window (...);
```

```
...
    first->insert_next(window_ptr);
```

Esto es posible porque Window es, por derivación, de tipo Slink. Puesto que las operaciones de la clase Slink son en función de Slink¹², puede ser necesario efectuar una conversión. Por ejemplo:

```
window_ptr = static_cast<Window*>(first->remove_next());
```

Como first->remove_next() retorna un Slink, la conversión lo modifica para que se trate como de tipo Window, el cual es el caso por derivación.

La segunda forma para enlazar clases es por colocación de un Slink como atributo de Window:

```
class Window
{
    ...
    Slink link;
};
```

Si first apunta al primer nodo de una lista, entonces una instancia window_ptr de tipo Window puede insertarse después de first del siguiente modo:

```
first->insert_next(&window_ptr->link);
```

El enfoque anterior tiene la ventaja de que hace posible enlazar objetos a varias listas, o sea, un objeto puede ser parte de varias listas. Para ello, simplemente debemos declarar como atributos tantos Slink como la cantidad de listas a las cuales pertenecería el objeto. El ejemplo anterior puede multiplicarse de la siguiente forma:

```
class Window
{
    ...
    Slink link_1;
    Slink link_2;
    ...
    Slink link_n;
};

...
first->insert_next(&window_ptr->link_1);
first->insert_next(&window_ptr->link_2);
...
first->insert_next(&window_ptr->link_n);
```

Con esta técnica no es posible hacer una conversión por compilador, pues cualquier atributo de tipo Slink no es la propia clase Window, sino parte de ella. Para obtener la dirección correcta a partir de un Slink, son necesarios algunos cálculos en aritmética de apuntadores. Puesto que tales cálculos son muy susceptibles de error, es bastante deseable encapsularlos en una primitiva de la siguiente forma:

```
static Window * link_to_window(Slink *& link)
{
    Window * ptr_zero      = 0;
```

¹²Redundancia adrede.

```

size_t offset_link    = (char*) &(ptr_zero->link);
char * address_result = ((char*) link) - offset_link;
return (Window *) address_result;
}

```

No tenemos forma de conocer los nombres de los atributos Slink ni su cantidad. No podemos, pues, ofrecer una rutina genérica que nos convierta un Slink a la clase que contenga el Slink, pero sí ofrecer un macro que prepare el terreno:

68a *(función de conversión de Slink a clase 68a)*≡ (65a)

```

#define SLINK_TO_TYPE(type_name, link_name) \
static type_name * slink_to_type(Slink * link) \
{ \
    type_name * ptr_zero = 0; \
    size_t offset_link = (size_t) &(ptr_zero->link_name); \
    char * address_type = ((char*) link) - offset_link; \
    return (type_name *) address_type; \
}

```

Uses Slink 65a.

El macro tiene dos parámetros. *type_name* es el nombre de la clase que alberga un Slink. *link_name* es el nombre de un atributo de tipo Slink contenido dentro de la clase *type_name*.

Mediante el macro SLINK_TO_TYPE, el usuario dispone de una implantación de conversión de Slink a su clase. El macro se debe incluir dentro de la clase que utilice el Slink. Por ejemplo, para la clase Window:

```

class Window
{
/* ... */
SLINK_TO_TYPE(Window, link);
};

```

Se genera un miembro estático que efectúa el mismo trabajo que *(función de conversión de Slink a clase 68a)*.

Si bien el macro es algo complicado, éste es independiente de la posición de Slink en Window. La estructura del miembro estático es la misma para toda clase que use un Slink, sólo hay que cambiar Window y link por los nombres que el cliente escoja.

2.4.3 El TAD Snode<T> (nodo simple)

Snode<T> es una clase parametrizada que abstrae un nodo perteneciente a una lista simplemente enlazada circular, con nodo cabecera, cuyos datos son de tipo T. Snode<T> se define e implanta en el archivo *(tpl_snode.H 68b)*, el cual posee la siguiente estructura:

68b *(tpl_snode.H 68b)*≡

```

template <typename T> class Snode : public Slink
{
    <miembros privados de Snode<T> 69a>
    <miembros públicos de Snode<T> 69b>
};

```

Defines:

Snode, used in chunks 69, 71c, 103h, 104b, 116, and 129a.

Uses Slink 65a.

Puesto que `Snode<T>` deriva de `Slink`, éste es también un `Slink` y hereda, a excepción de algunos métodos que deben sobrecargarse, la mayor parte de los métodos de `Slink`.

Un `Snode<T>` tiene un único atributo:

69a $\langle \text{miembros privados de } Snode<T> \rangle \equiv$ (68b)
`T data;`

`data` contiene el dato almacenado en el nodo. Este dato puede consultarse mediante el observador:

69b $\langle \text{miembros públicos de } Snode<T> \rangle \equiv$ (68b) 69c ▷
`T& get_data() { return data; }`

Hay dos maneras de construir un `Snode<T>`:

69c $\langle \text{miembros públicos de } Snode<T> \rangle + \equiv$ (68b) ▷ 69b 69d ▷
`Snode() { /* empty */ }`

`Snode(const T & _data) : data(_data) { /* empty */ }`

Uses `Snode` 68b.

El constructor vacío crea un nodo con valor de dato indeterminado. El segundo constructor crea un nodo con el valor de dato `data`.

El método `insert_next()` se hereda directamente de `Slink`, lo cual es permisible porque un `Snode<T>` es también un `Slink`.

Por el contrario, debemos sobrecargar `remove_next()`, pues ésta retorna un `Slink` y quereremos que se retorne un `Snode<T>`:

69d $\langle \text{miembros públicos de } Snode<T> \rangle + \equiv$ (68b) ▷ 69c 69e ▷
`Snode * remove_next() { return (Snode*) Slink::remove_next(); }`

Uses `Slink` 65a and `Snode` 68b.

La misma sobrecarga debe efectuarse para `get_next()`:

69e $\langle \text{miembros públicos de } Snode<T> \rangle + \equiv$ (68b) ▷ 69d
`Snode * get_next() const { return (Snode*) Slink::get_next(); }`

Uses `Slink` 65a and `Snode` 68b.

2.4.4 El TAD `Slist<T>` (lista simplemente enlazada)

El TAD `Slist<T>` abstrae una lista simplemente enlazada circular de nodos simples y se define en el archivo `<tpl_slist.H 69f>`, en el cual se define la clase en cuestión:

69f $\langle \text{tpl_slist.H 69f} \rangle \equiv$
`template <typename T> class Slist : public Snode<T>`
`{`
 `<\text{definiciones de } Slist<T> 69g>`
 `<\text{métodos públicos de } Slist<T> 70a>`
 `<\text{iterador de } Slist<T> 70d>`
`};`

Uses `Snode` 68b.

La clase `Slist<T>` modeliza una lista simplemente enlazada de nodos simples que almacenan datos de tipo `T`:

69g $\langle \text{definiciones de } Slist<T> 69g \rangle \equiv$ (69f)
`typedef Snode<T> Node;`

Uses `Snode` 68b.

Esta declaración exporta el tipo `Slist<T>::Node` sinónimo de `Snode<T>`.

La única inserción posible es al principio de la lista, tal como sigue:

70a *(métodos públicos de Slist<T> 70a)≡* (69f) 70b▷
 void insert_first(Node * node)
 {
 this->insert_next(node);
}

`insert_first()` inserta el nodo al principio de la lista; es decir, `node` deviene el primer elemento de la lista.

Del mismo modo, la única forma de suprimir es por el principio de la lista:

70b *(métodos públicos de Slist<T> 70a)+≡* (69f) <70a 70c>
 Node * remove_first()
{
 return this->remove_next();
}

`remove_first()` elimina el primer elemento de la lista y retorna su dirección.

El usuario de `Slist<T>` puede conocer la dirección del primer nodo mediante:

70c *(métodos públicos de Slist<T> 70a)+≡* (69f) <70b
 Node * get_first() const
{
 return this->get_next();
}

El resto de los elementos pueden accederse a través de la interfaz de `Slist<T>::Node`, la cual es la misma de `Snode<T>`. El usuario puede efectuar inserciones y supresiones de la lista a través de esta vía.

2.4.5 Iterador de `Slist<T>`

`Slist<T>` exporta un iterador bajo la sub-clase:

70d *(iterador de Slist<T> 70d)≡* (69f)
 class Iterator
{
(miembros privados de iterador de Slist<T> 70e)
(miembros públicos de iterador de Slist<T> 70f)
};

De alguna manera, el iterador debe poseer la información suficiente para poder recorrer la lista. Tal información está constituida por:

70e *(miembros privados de iterador de Slist<T> 70e)≡* (70d)
 Slist * list;
 Node * current;

`list` es un apuntador a la lista enlazada sobre la cual se itera y `current` es el elemento actual del iterador.

De resto, el iterador se define simplemente como sigue:

70f *(miembros públicos de iterador de Slist<T> 70f)≡* (70d)
 Iterator(Slist & _list) : list(&_list), current(list->get_first()) {}

 bool has_current() const { return current != list; }

```

Node * get_current()
{
    return current;
}
void next()
{
    current = current->get_next();
}
void reset_first() { current = list->get_next(); }

El siguiente ejemplo visita todos los elementos de una lista:
```

```

for (typename Slist<int>::Iterator itor(list); itor.has_current();
     itor.next())
    // procesar itor.get_current()->get_data();
```

2.4.6 El TAD DynSlist<T>

El TAD `DynSlist<T>` modeliza una lista de elementos de tipo `T` cuyo manejo de memoria lo hace internamente el propio TAD. Como tal, este TAD se acerca más al concepto ideal de lista enunciado al principio del capítulo. Su definición e implantación se encuentran en el archivo `<tpl_dynSlist.H 71a>`, cuya estructura es la siguiente:

71a `<tpl_dynSlist.H 71a>≡`

```

template <typename T>
class DynSlist : public Slist<T>
{
    (Miembros privados de DynSlist<T> 71b)
    (Miembros públicos de DynSlist<T> 72)
};
```

`DynSlist<T>` hereda parte de la interfaz e implantación de `Slist<T>`. `this` es entonces el nodo cabecera de la lista. Adicionalmente, `DynSlist<T>` contabiliza la cantidad de elementos de la lista:

71b `<Miembros privados de DynSlist<T> 71b>≡` (71a) 71c ▷

```

size_t num_items;
```

Antes de la aparición del patrón de iterador presentado en § 2.3 (Pág. 59), las interfaces a las listas enlazadas manejaban el concepto de “cursor”. Un cursor abstrae una posición de la lista dentro la secuencia respecto a la cual se efectúan sus operaciones principales. A un cursor se le llama también “posición actual”.

Para llevar el estado del cursor se utilizan los siguientes atributos:

71c `<Miembros privados de DynSlist<T> 71b>+≡` (71a) ▷71b 71d ▷

```

int         current_pos;
Snode<T> * current_node;
```

Uses `Snode 68b`.

`current_pos` es el ordinal del elemento actual dentro de la secuencia. `current_node` es el nodo predecesor al elemento actual.

Antes de implantar las primitivas principales de `DynSlist<T>`, requerimos una rutina de acceso por posición:

71d `<Miembros privados de DynSlist<T> 71b>+≡` (71a) ▷71c

```

typename Slist<T>::Node * get_previous_to_pos(const int & pos)
{
```

```

if (pos < current_pos) // hay que retroceder?
{ // Si, reinicie posición actual
    current_pos = 0;
    current_node = this;
}
while (current_pos < pos) // avanzar hasta nodo predecesor a pos
{
    current_node = current_node->get_next();
    ++current_pos;
}
return current_node;
}

```

`get_previous_to_pos()` retorna el nodo predecesor a la posición `pos` y deja `current_node` posicionado en dicho predecesor. La memorización del predecesor es indispensable para los algoritmos de inserción y eliminación.

Los miembros públicos restantes se especifican como sigue:

72 (Miembros públicos de `DynSlist<T>` 72)≡ (71a)

```

T & operator [] (const size_t & i)
{
    return get_previous_to_pos(i)->get_next()->get_data();
}

void insert(const int & pos, const T & data)
{   // apartar nodo para nuevo elemento
    typename Slist<T>::Node * node = new typename Slist<T>::Node (data);
    typename Slist<T>::Node * prev = get_previous_to_pos(pos);
    prev->insert_next(node);
    ++num_items;
}

void remove(const int & pos)
{   // obtener nodo predecesor al nuevo elemento
    typename Slist<T>::Node * prev = get_previous_to_pos(pos);
    typename Slist<T>::Node * node_to_delete = prev->remove_next();
    delete node_to_delete;
    -num_items;
}

~DynSlist()
{   // eliminar nodo por nodo hasta que la lista devenga vacía
    while (not this->is_empty())
        delete this->remove_first(); // remove_first de clase Slink
}

```

Uses `Slink 65a.`

`DynSlist<T>` exporta un iterador cuya especificación es muy simple al hacerla derivada de `Slist<T>::Iterator`.

2.4.7 El TAD Dlink (enlace doble)

En el mismo espíritu de diseño que el TAD `Slink`, el TAD `Dlink` define un doble enlace contenido en un nodo perteneciente a una lista doblemente enlazada circular con nodo cabecera.

Dlink se especifica e implanta en el archivo `<dlink.H 73a>` que se estructura del siguiente modo:

73a `<dlink.H 73a>≡`
`class Dlink`
`{`
 `<miembros protegidos de Dlink 73b>`
 `<miembros públicos de Dlink 73c>`
`};`
`(macros conversión de Dlink a clase (never defined))`

A partir de este momento es muy importante puntualizar y distinguir los siguientes fines:

1. El fin de un Dlink es fungir de nodo cabecera de una lista circular doblemente enlazada. En este sentido, las operaciones de la clase Dlink refieren a listas doblemente enlazadas, circulares, cuyo nodo cabecera es `this`.
2. El fin de un `Dlink*` es fungir de apuntador a un nodo perteneciente a una lista circular doblemente enlazada.

Dlink posee dos atributos protegidos:

73b `<miembros protegidos de Dlink 73b>≡` (73a)
`mutable Dlink * prev;`
`mutable Dlink * next;`

`next` y `prev` son apuntadores al sucesor y predecesor de `this`.

Un nuevo Dlink siempre se crea con sus lazos apuntando a sí mismo; es decir, como un nodo cabecera cuya lista está vacía:

73c `<miembros públicos de Dlink 73c>≡` (73a) 73d▷
`Dlink() : prev(this), next(this) {}`

Eventualmente, aunque delicado, es conveniente exportar interfaces para copias y asignaciones:

73d `<miembros públicos de Dlink 73c>+≡` (73a) ▷73c 73e▷
`Dlink(const Dlink &) { reset(); }`

`Dlink & operator = (const Dlink & l)`
`{`
 `reset();`
 `return *this;`
`}`

En realidad, las dos operaciones no efectúan copia; se exportan para satisfacer operaciones con variables temporales usadas por el compilador. No es posible hacer copia a este nivel porque no se maneja ningún mecanismo de asignación de memoria ni se conoce el tipo de dato que albergan los nodos.

Un doble enlace puede “reiniciarse” mediante:

73e `<miembros públicos de Dlink 73c>+≡` (73a) ▷73d 74▷
`void reset()`
`{`
 `next = prev = this;`
`}`

`reset()` reinicia una lista a que apunte a sí mismo. Esto no es equivalente a eliminar los nodos atados a `this`.

Aunque no es posible asignar sobre una lista que contenga elementos, sí lo es intercambiar los contenidos de dos listas doblemente enlazadas en tiempo constante. Por esta razón se ofrece la primitiva swap() cuya implantación es como sigue:

```
74 <miembros públicos de Dlink 73c>+≡ (73a) ◁73e 75a▷
    void swap(Dlink * link)
    {
        if (is_empty() and link->is_empty())
            return;

        if (is_empty())
        {
            link->next->prev = this;
            link->prev->next = this;
            next = link->next;
            prev = link->prev;
            link->reset();

            return;
        }
        if (link->is_empty())
        {
            next->prev = link;
            prev->next = link;
            link->next = next;
            link->prev = prev;
            reset();

            return;
        }
        std::swap(prev->next, link->prev->next);
        std::swap(next->prev, link->next->prev);
        std::swap(prev, link->prev);
        std::swap(next, link->next);
    }
```

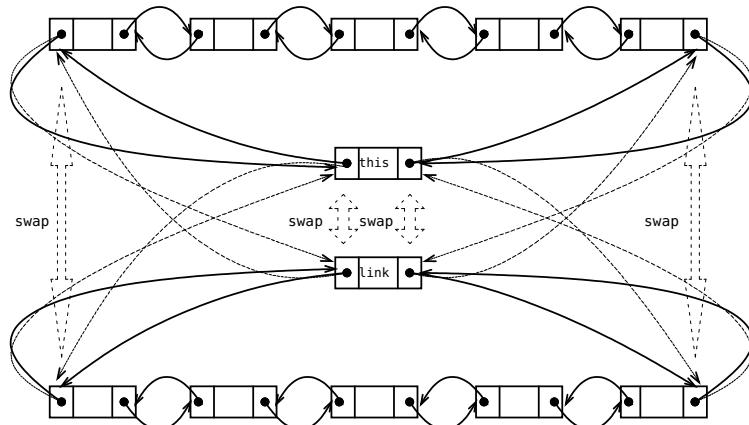


Figura 2.9: Operación swap(link)

`swap()` es una gran operación, pues aparte de que su tiempo de ejecución es espectacular, pues es constante, es general para todas las listas doblemente enlazadas con nodo cabecera, sin importar ni el tipo de dato que se maneje ni cómo se reserve la memoria. La figura 2.9 ilustra los nodos de las listas en los cuales se llevan a cabo los intercambios.

Asumiendo que `this` es el nodo cabecera de una lista podemos saber si la lista está vacía o no si contiene un solo elemento o si su cardinalidad es menor o igual a uno:

75a *(miembros públicos de Dlink 73c) +≡* (73a) ◁74 75b ▷
`bool is_empty() const { return this == next and this == prev; }`

`bool is_unitarian() const { return this != next and next == prev; }`
`bool is_unitarian_or_empty() const { return next == prev; }`

Un punto a destacar de estas operaciones es que no requieren contabilizar la cantidad de nodos de la lista.

La figura 2.10 enumera los diferentes pasos involucrados en la inserción, los cuales se efectúan en la rutina `insert()`, la cual inserta a `node` como el sucesor de `this`:

75b *(miembros públicos de Dlink 73c) +≡* (73a) ◁75a 75c ▷
`void insert(Dlink * node)`
`{`
 `node->prev = this;`
 `node->next = next;`
 `next->prev = node;`
 `next = node;`
`}`

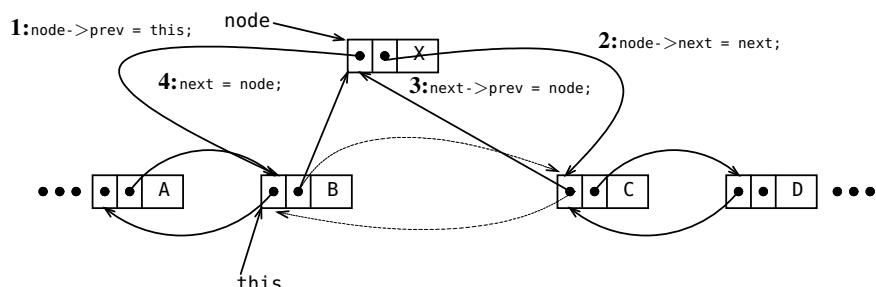


Figura 2.10: Inserción en una lista doblemente enlazada implantada con Dlink

La inserción como predecesor se denomina `append()` y se define como sigue:

75c *(miembros públicos de Dlink 73c) +≡* (73a) ◁75b 76a ▷
`void append(Dlink * node)`
`{`
 `node->next = this;`
 `node->prev = prev;`
 `prev->next = node;`
 `prev = node;`
`}`

Puesto que la lista es circular, `insert()` desde el nodo cabecera inserta un nodo al principio de la lista. Análogamente, `append()` los inserta al final.

Dada la dirección de un nodo podemos acceder a su sucesor y predecesor mediante los siguientes métodos:

76a *(miembros públicos de Dlink 73c) +≡*

```
Dlink *& get_next()
{
    return next;
}

Dlink *& get_prev()
{
    return prev;
}
```

(73a) ◁75c 76b▷

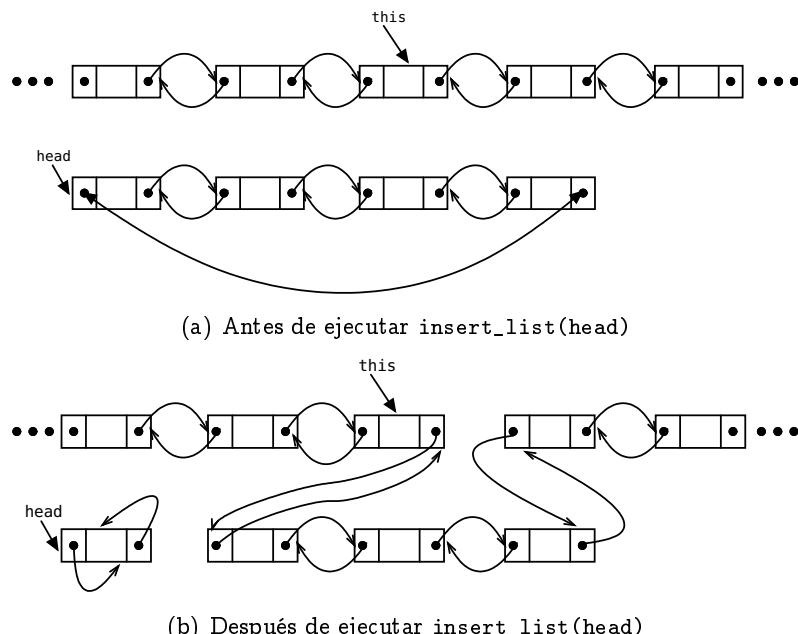


Figura 2.11: Inserción de lista dentro de una lista

Existen circunstancias en las cuales se requiere insertar una lista completa dentro de otra a partir de uno de sus nodos. Para ello utilizamos las primitivas `insert_list()` y `append_list()`. La figura 2.11 muestra el proceso de ejecución de `insert_list(head)`. Después su ejecución, la lista cuyo nodo cabecera estaba apuntado por `head` deviene vacía, pues sus nodos fueron incluidos en `this`. Es muy importante notar que en este caso `this` no necesariamente es un nodo cabecera, sino que puede ser cualquier otro nodo. Las implementaciones son como sigue:

76b *(miembros públicos de Dlink 73c) +≡*

```
void insert_list(Dlink * head)
{
    if (head->is_empty())
        return;

    head->prev->next = next;
    head->next->prev = this;
    next->prev      = head->prev;
```

(73a) ◁76a 77a▷

```

next           = head->next;
head->reset();
}
void append_list(Dlink * head)
{
    if (head->is_empty())
        return;

    head->next->prev = prev;
    head->prev->next = this;
    prev->next       = head->next;
    prev           = head->prev;
    head->reset();
}

```

En algunos contextos, las operaciones `insert_list()` y `append_list()` se conocen como “splice”¹³.

Un caso particular, quizá mucho más común que el splice, es la operación de concatenar listas `concat_list(head)`, la cual concatena la lista cuyo nodo cabecera es `head` con `this`. `head` deviene vacía después de la operación. En este caso, sí se asume que `this` es nodo cabecera. Al respecto, se plantea la siguiente implementación:

77a <*miembros públicos de Dlink 73c*>+≡ (73a) ◁76b 77b▷

```

void concat_list(Dlink * head)
{
    if (head->is_empty())
        return;

    if (this->is_empty())
    {
        swap(head);
        return;
    }
    prev->next       = head->next;
    head->next->prev = prev;
    prev           = head->prev;
    head->prev->next = this;
    head->reset();
}

```

Dada la dirección de un nodo hay varias maneras de invocar una eliminación. La más útil de todas, denominada “autoeliminación”, se efectúa mediante `del()`. La rutina es muy útil porque permite que otras estructuras de datos almacenen referencias eliminables a elementos de una lista enlazada. En otras palabras, un nodo cualquiera puede suprimirse a sí mismo de la lista. Su implementación es como sigue:

77b <*miembros públicos de Dlink 73c*>+≡ (73a) ◁77a 78a▷

```

void del()
{
    prev->next = next;
}

```

¹³En inglés, este término se utiliza cuando se desea expresar que dos cosas se pegan, se juntan, por sus extremos -una punta con la otra-. En la opinión de este redactor, el equivalente castellano más próximo es “enlazar”, cuyo uso plantea una ambigüedad en la jerga de listas enlazadas. Por esa razón, continuaremos utilizando el término en inglés.

```

    next->prev = prev;
    reset();
}

```

Los pasos de del() se muestran en la figura 2.12.

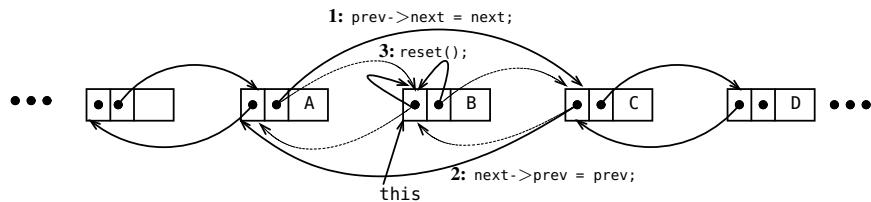


Figura 2.12: Autoeliminación en una lista doblemente enlazada implantada con Dlink

Dado un nodo hay otras dos maneras de eliminar: su predecesor o su sucesor, las cuales se implantan mediante los siguientes métodos:

78a *(miembros públicos de Dlink 73c) +≡*

(73a) ◁ 77b 78b ▷

```

Dlink * remove_prev()
{
    Dlink* retValue = prev;
    retValue->del();
    return retValue;
}

Dlink * remove_next()
{
    Dlink* retValue = next;
    retValue->del();
    return retValue;
}

```

`remove_prev()` suprime el predecesor respecto a `this`; `remove_next()` suprime el sucesor. Estas primitivas son particularmente útiles para el nodo cabecera de la lista. Puesto que la lista es circular, `remove_prev()`, invocada desde el nodo cabecera, suprime el último nodo de la lista; similarmente, `remove_next()` suprime el primero.

Una aplicación directa de algunos de los métodos explicados se ejemplifica mediante una rutina de inversión de nodos de la lista:

78b *(miembros públicos de Dlink 73c) +≡*

(73a) ◁ 78a 79a ▷

```

size_t reverse_list()
{
    if (is_empty())
        return 0;

    Dlink tmp; // cabecera temporal donde se guarda lista invertida

    // recorrer lista this, eliminar primero e insertar en tmp
    size_t counter = 0;
    for /* nada */; not is_empty(); counter++)
        tmp.insert(remove_next()); // eliminar e insertar en tmp

    swap(&tmp); // tmp == lista invertida; this vacía ==> swap
}

```

```

    return counter;
}

```

A parte de invertir la lista, `reverse_list()` aprovecha el recorrido para contar la cantidad de nodos; cantidad que retorna la función.

Dada una lista, ¿cómo partirla por el centro en dos listas del mismo tamaño? Un truco consiste en avanzar dos apuntadores. Por cada iteración, un puntero avanza un paso, mientras que el otro avanza dos¹⁴. Cuando el segundo puntero se encuentre al final de la lista, el primero se encontrará en el centro; este es el punto de partición. Este enfoque, es el más adecuado para particionar una lista simple, pero debe programarse cuidadosamente los casos extremos de cero, uno o dos elementos.

Para listas doblemente enlazadas existe un enfoque mucho más simple y, como todo lo simple, más confiable: recorrer la lista por cada extremo. Por el lado izquierdo se recorre hacia la derecha; por el derecho hacia la izquierda. En cada iteración se elimina de cada extremo y se inserta en cada una de las listas resultado. Para ello diseñamos el método `split_list(l, r)`, el cual recibe dos cabeceras de listas vacías `l` y `r` y partitiona `this` por el centro en dos partes, la izquierda en `l` y la derecha en `r`:

79a ⟨miembros públicos de `Dlink` 73c⟩+≡ (73a) ◁ 78b 79b ▷

```

size_t split_list(Dlink & l, Dlink & r)
{
    size_t count = 0;
    while (not is_empty())
    {
        l.append(remove_next()); ++count;

        if (is_empty())
            break;

        r.insert(remove_prev()); ++count;
    }
    return count;
}

```

Dada una lista y uno de sus nodos, puede ser conveniente particionarla en un nodo dado. Para ello, se provee la función `cut_list()` cuya implantación es como sigue:

79b ⟨miembros públicos de `Dlink` 73c⟩+≡ (73a) ◁ 79a 80a ▷

```

void cut_list(Dlink * link, Dlink * list)
{
    list->prev = prev;           // enlazar list a link (punto de corte)
    list->next = link;
    prev = link->prev;          // quitar de this todo a partir de link
    link->prev->next = this;
    link->prev = list;          // colocar el corte en list
    list->prev->next = list;
}

```

El esquema de este algoritmo se ilustra en la figura 2.13. `cut_list()` partitiona `this` en el nodo cuya dirección es `link`, el cual debe pertenecer a la lista `this`. Los nodos a la izquierda de `link` se preservan en `this`, mientras que los restantes, incluido `link`, se copian a `list`.

¹⁴Se puede decir que el segundo duplica en velocidad al primero.

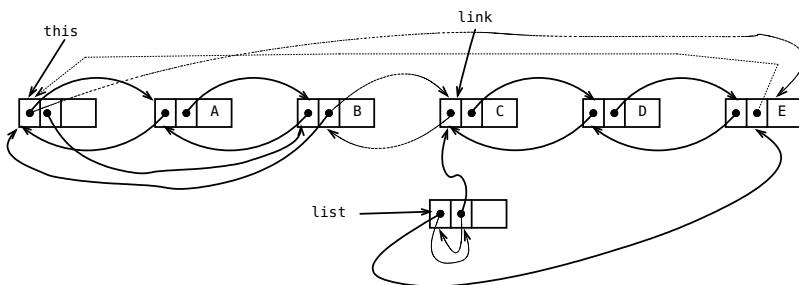


Figura 2.13: Corte de una lista en un nodo dado

Los usos de Dlink son casi los mismos que el de Slink desarrollado en la subsección § 2.4.2 (Pág. 65). Podemos hacer a una clase ser un Dlink por derivación pública, o hacerla parte de un nodo doble por declaración de un Dlink como atributo de la clase. La diferencia esencial respecto a Slink es su versatilidad, expresada por la riqueza de las operaciones que hemos estudiado, las cuales serían más difíciles de desarrollar que las operaciones de Slink.

Al igual que con Slink, para situaciones en que se usan registros que contengan Dlink se requieren macros que generan funciones de conversión de un Dlink hacia una clase que contenga un atributo Dlink. En este sentido hay dos posibilidades probables, aunque no generales: conversión hacia una clase simple o conversión hacia una clase parametrizada. Estas posibilidades se engloban en los macros DLINK_TO_TYPE(type_name, link_name) y LINKNAME_TO_TYPE(type_name, link_name), los cuales generan una función de conversión que convierte un puntero link de tipo Dlink en un puntero a una clase que contiene el Dlink.

El macro DLINK_TO_TYPE genera una función general con nombre dlink_to_type(). El macro LINKNAME_TO_TYPE recibe un parámetro llamado link_name, el cual debería corresponder exactamente con el nombre del campo dentro de la clase. La razón de ser de esta función es que permite al usuario distintos nombres de funciones para distintos nombres de campos; esto es indispensable en los casos en que la clase contenga dos o más enlaces dobles. Los macros DLINK_TO_TYPE y LINKNAME_TO_TYPE deben utilizarse dentro de la clase que use el tipo Dlink.

Iterador de Dlink

Dlink exporta un iterador (ver § 2.3 (Pág. 59)) cuyo esquema se presenta como sigue:

80a *(miembros públicos de Dlink 73c)*+≡ (73a) ▷79b 82c▷
 class Iterator
 {
 (atributos Dlink::Iterator 80b)
 (miembros públicos iterador Dlink::Iterator 80c)
 };

Para mantener el estado del iterador requerimos dos miembros:

80b *(atributos Dlink::Iterator 80b)*≡ (80a)
 mutable Dlink * head;
 mutable Dlink * curr;

head es el nodo cabecera de la lista. curr es el nodo actual del iterador.

Hay varias maneras de construir un iterador:

80c *(miembros públicos iterador Dlink::Iterator 80c)*≡ (80a) 81a▷

```
Iterator(Dlink * head_ptr) : head(head_ptr), curr(head->get_next()) {}
```

```
Iterator(const Iterator & itor) : head(itor.head), curr(itor.curr) {}
```

Un iterador puede asignarse:

81a *(miembros públicos iterador Dlink::Iterator 80c)+≡* (80a) ◁80c 81b▷

```
Iterator & operator = (const Iterator & itor)
{
    head = itor.head;
    curr = itor.curr;
    return *this;
}
```

Un iterador puede reutilizarse, lo que requiere funciones de reiniciación:

81b *(miembros públicos iterador Dlink::Iterator 80c)+≡* (80a) ◁81a 81c▷

```
void reset_first()
{
    curr = head->get_next();
}
void reset_last()
{
    curr = head->get_prev();
}
```

`reset_first()` posiciona el iterador sobre el primer elemento. `reset_last()` posiciona el iterador sobre el último elemento.

Existen situaciones especiales en las cuales se desea inicializar un iterador a partir de un elemento actual ya conocido:

81c *(miembros públicos iterador Dlink::Iterator 80c)+≡* (80a) ◁81b 81d▷

```
void set(Dlink * new_curr)
{
    curr = new_curr;
}
void reset(Dlink * new_head)
{
    head = new_head;
    curr = head->get_next();;
```

`set()` sitúa el iterador a un nuevo nodo actual, mientras `reset()` sitúa el iterador a una nueva lista.

El elemento actual del iterador se accede y se verifica mediante:

81d *(miembros públicos iterador Dlink::Iterator 80c)+≡* (80a) ◁81c 82a▷

```
bool has_current() const
{
    return curr != head;
}
Dlink * get_current()
{
    return curr;
}
bool is_in_first() const { return curr == head->next; }

bool is_in_last() const { return curr == head->prev; }
```

Para avanzar el iterador utilizamos las siguientes primitivas:

```
82a <miembros públicos iterador Dlink::Iterator 80c>+≡ (80a) ◁81d 82b▷
    void prev()
    {
        curr = curr->get_prev();
    }
    void next()
    {
        curr = curr->get_next();
    }
```

A menudo pueden combinarse iteradores en un `for` que semejan la iteración sobre un arreglo. Como una lista no tiene acceso directo, la condición de iteración no debe ser una comparación por posición del tipo `i < n`. En una lista `y`, en general, para secuencias que se manipulen con iteradores, la comparación debe ser entre iteradores. Puesto que no se puede conocer la posición dentro de la secuencia para todas las situaciones, no es posible emplear los comparadores relacionales `<`, `<=`, `>`, `>=`, pero sí se pueden sobrecargar los operadores `==` y `!=` tal como en efecto lo están en la biblioteca.

Un estilo tradicional de uso de la comparación entre iteradores se ilustra en este ejemplo:

```
for (Dlink::Iterator curr(list); curr != end; curr.next())
```

Donde `end` es un iterador sobre la lista `list` apuntando al final. Este es el estilo de la biblioteca estándar `stdc++`. `end` es equivalente a:

```
Dlink::Iterator end(list);
list.reset_last();
list.next();
```

Es posible insertar respecto al elemento actual del iterador. Para ello basta con invocar sobre el elemento actual cualquiera de las primitivas de inserción `insert()` o `append()`.

Hay situaciones en las que se requiere eliminar el elemento actual del iterador. En este caso no es posible efectuar `del()` sobre el nodo actual, pues podría perderse el estado del iterador. Para solventar esta situación, la clase `Dlink::Iterator` exporta una función de eliminación sobre el nodo actual que deja al iterador en el nodo sucesor del eliminado:

```
82b <miembros públicos iterador Dlink::Iterator 80c>+≡ (80a) ◁82a
    Dlink * del()
    {
        Dlink * current = get_current(); // obtener nodo actual
        next(); // avanzar al siguiente nodo
        current->del(); // eliminar de la lista antiguo nodo actual
        return current;
    }
```

`del()` retorna el enlace eliminado de manera tal que el cliente pueda disponer de él en la forma que prefiera.

Como ejemplo de uso de `Dlink::Iterator` consideremos el siguiente método:

```
82c <miembros públicos de Dlink 73c>+≡ (73a) ◁80a
    void remove_all_and_delete()
    {
        for (Iterator itor(this); itor.has_current(); delete itor.del()) ;
    }
```

Este método elimina todos los nodos y asume que la memoria de cada nodo fue asignada mediante new.

2.4.8 El TAD Dnode<T> (nodo doble)

En un nivel superior respecto a Dlink, el TAD Dnode<T> modeliza un nodo perteneciente a una lista doblemente enlazada circular. Dnode<T> se define e implanta en el archivo *<tpl_dnode.H 83a>* cuya estructura es la siguiente:

83a *<tpl_dnode.H 83a>*≡
 template <typename T> class Dnode : public Dlink
 {
 <métodos privados Dnode<T> 83b>
 <métodos públicos Dnode<T> 83c>
 };

Defines:

Dnode, used in chunks 83–91, 153–55, 165a, 185, 384b, 385c, and 440a.

Dnode<T> es un Dlink público por derivación. La única diferencia reside en que Dnode<T> almacena un elemento de tipo de T:

83b *<métodos privados Dnode<T> 83b>*≡ (83a)
 mutable T data;

Los métodos de Dlink que retornen punteros Dlink* deben sobrecargarse para que retornen punteros tipeados Dnode<T>*:

83c *<métodos públicos Dnode<T> 83c>*≡ (83a) 83d>
 Dnode<T> *& get_next() { return (Dnode<T>*&) next; }
 Dnode<T> *& get_prev() { return (Dnode<T>*&) prev; }
 Dnode<T>* remove_prev() { return (Dnode<T>*) Dlink::remove_prev(); }
 Dnode<T>* remove_next() { return (Dnode<T>*) Dlink::remove_next(); }

Uses Dnode 83a.

La única función de estos métodos es efectuar la conversión hacia Dnode<T>. El manejo de los enlaces se implanta por el método de la clase base Dlink.

El acceso al dato se realiza por:

83d *<métodos públicos Dnode<T> 83c>+≡* (83a) <83c 83e>
 T & get_data() { return data; }

Un punto sumamente importante es el tipo de dato de Dnode<T>, el cual se exporta mediante la siguiente declaración:

83e *<métodos públicos Dnode<T> 83c>+≡* (83a) <83d 84>
 typedef T dnode_type;

Esta declaración permite que programas genéricos conozcan el tipo de dato de Dnode<T>. Por ejemplo, si se tiene un Dnode<T> bajo el tipo genérico Node, entonces, un fragmento de programa podría ser:

typename Node::dnode_type data;

el cual declara una variable de nombre data del tipo involucrado en Node, el cual es el mismo tipo genérico T.

Hay un método cuyo uso es excepcional, pero que puede ser extraordinariamente útil en ciertas circunstancias:

```
84  <métodos públicos Dnode<T> 83c>+≡ (83a) <83e
    static Dnode * data_to_node(T & data)
    {
        Dnode * zero = 0;
        long offset = (long) &(zero->data);
        char * addr = (char*) (&data);
        return (Dnode*) (addr - offset);
    }
Uses Dnode 83a.
```

Iterador de Dnode<T>

Dnode<T> exporta un iterador cuya sintaxis y semántica son idénticas a las de Dlink::Iterator. Puesto que el estado y control del iterador ya está implantado, la especificación de Dnode<T>::Iterator sólo se remite a la sobrecarga de los constructores y a la obtención del elemento actual.

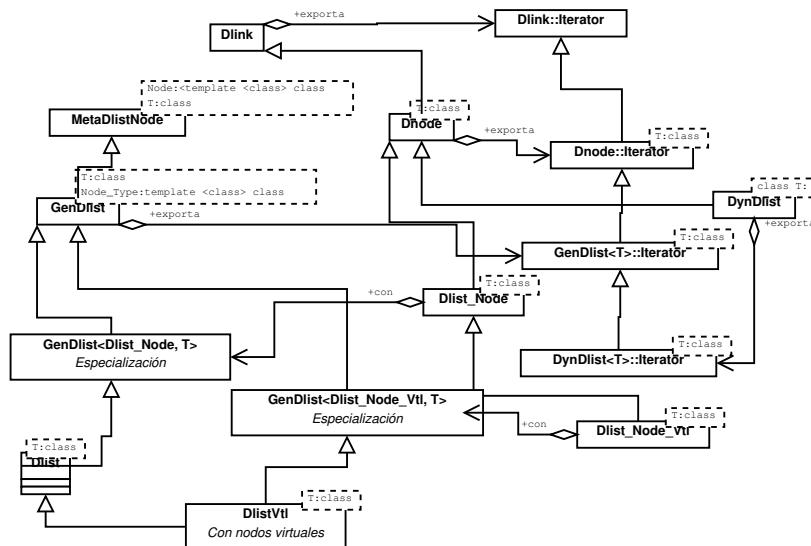


Figura 2.14: Diagrama general UML de las clases *ALÉPH* para listas enlazadas dobles

2.4.9 El TAD DynDlist<T>

Las clases Dnode<T> y Dlist<T> son suficientemente versátiles como para llevar a cabo aplicaciones complejas que requieran listas doblemente enlazadas. Sin embargo, estas clases aún requieren que el programador comprenda cabalmente sus sutilezas de uso; en particular, el manejo de memoria.

En un último nivel, diseñamos la clase DynDlist<T>, la cual modeliza una lista doblemente enlazada circular con manejo interno de memoria. DynDlist<T> se especifica e

implanta en el archivo *<tpl_dynDlist.H 85a>*:

85a *<tpl_dynDlist.H 85a>*≡
 template <typename T> class DynDlist : public Dnode<T>
 {
 (métodos públicos DynDlist<T> 85b)
 };

Defines:

DynDlist, used in chunks 85, 87–91, 94a, 95d, 98, 155a, 207, 254b, 255, 579–81, 610b, 611b, 615b, 617, 626a, 628, 644–48, 656, 657, 659, 706, 712b, 769, 780–82, 788a, 789, and 791.

Uses Dnode 83a.

La idea DynDlist<T> es que soporte toda la funcionalidad que arrastramos desde Dlink sin necesidad de pensar en el manejo de apuntadores, nodos y memoria.

DynDlist<T> hereda públicamente su implantación de la clase Dnode<T> y una buena parte de su interfaz. La responsabilidad esencial de DynDlist<T> es el manejo de memoria.

El único atributo de DynDlist<T> es num_elem, el cual contabiliza la cantidad de elementos que posee la lista y puede observarse mediante:

85b *<métodos públicos DynDlist<T> 85b>*≡
const size_t & size() const { return num_elem; }

(85a) 85c▷

La construcción es muy simple, pues, a excepción del manejo de memoria y de la contabilización del número de elementos, todo está implantado desde la clase Dnode<T>:

85c *<métodos públicos DynDlist<T> 85b>*+≡
DynDlist() : num_elem(0) { / Empty */ }*

(85a) ◁85b 85d▷

Uses DynDlist 85a.

Al igual que en Dnode<T>, podemos insertar elementos por el principio o final de la lista.

85d *<métodos públicos DynDlist<T> 85b>*+≡
T & insert(const T & _data)
 {
 *Dnode<T> * p = new Dnode<T> (_data);*
 Dnode<T>::insert(p);
 ++num_elem;
 return p->get_data();
 }
 T & append(const T & _data)
 {
 *Dnode<T> * p = new Dnode<T> (_data);*
 Dnode<T>::append(p);
 ++num_elem;
 return p->get_data();
 }

(85a) ◁85c 85e▷

Uses Dnode 83a.

Las mismas operaciones se definen para listas, es decir, se puede concatenar una lista entera tanto por el principio como por el final:

85e *<métodos públicos DynDlist<T> 85b>*+≡
size_t insert_list(DynDlist & list)
 {
 Dlink::insert_list(&list);

(85a) ◁85d 86a▷

```

    num_elem += list.num_elem;
    list.num_elem = 0;
    return num_elem;
}
size_t append_list(DynDlist & list)
{
    Dlink::append_list(&list);
    num_elem += list.num_elem;
    list.num_elem = 0;
    return num_elem;
}

```

Uses DynDlist 85a.

El acceso a una secuencia `DynDlist<T>` se lleva a cabo por uno de sus extremos: el primer o el último elemento:

86a *(métodos públicos DynDlist<T> 85b) +≡* (85a) ◁ 85e 86b ▷

```

T & get_first()
{
    return this->get_next()->get_data();
}
T & get_last()
{
    return this->get_prev()->get_data();
}
```

Ambos métodos retornan referencias al dato incluido dentro de la lista. Esto significa que el usuario puede modificarlos directamente.

Tradicionalmente, la eliminación se realiza por alguno de los extremos:

86b *(métodos públicos DynDlist<T> 85b) +≡* (85a) ◁ 86a 86d ▷

```

T remove_first()
{
    Dnode<T> * ptr = this->remove_next();
    {obtener dato y liberar ptr 86c}
}
T remove_last()
{
    Dnode<T> * ptr = this->remove_prev();
    {obtener dato y liberar ptr 86c}
}
```

Uses Dnode 83a.

86c *(obtener dato y liberar ptr 86c) ≡* (86b)

```

T retVal = ptr->get_data();
delete ptr;
-num_elem;
return retVal;
```

`remove_first()` suprime el primer elemento, `remove_last()` el último. A diferencia de la consulta, la eliminación retorna copias de los valores, no referencias, pues el nodo cesa de existir después de liberar la memoria ocupada por el nodo.

Es posible vaciar una lista, esto es, eliminar todos sus elementos:

86d *(métodos públicos DynDlist<T> 85b) +≡* (85a) ◁ 86b 87a ▷

```

void empty()
{
    while (not this->is_empty())
        delete this->remove_next();
    num_elem = 0;
}

```

Esta primitiva debe ser invocada por el destructor:

87a *(métodos públicos DynDlist<T> 85b) +≡* (85a) ◁86d 87b ▷
`DynDlist()
{
 empty();
}

Uses DynDlist 85a.

En algunas circunstancias es muy útil poder eliminar un elemento que, sabemos, pertenece a la secuencia:

87b *(métodos públicos DynDlist<T> 85b) +≡* (85a) ◁87a 87c ▷
void remove(T & data)
{
 Dnode<T> * p = data_to_node(data);
 p->del();
 delete p;
 -num_elem;
}

Uses Dnode 83a.

Una primitiva esencial para el desempeño y elegancia de algunos algoritmos es swap(). Como todo el trabajo ya fue implantado desde la clase Dlink, nuestra primitiva sólo se remite a tipar e invocar:

87c *(métodos públicos DynDlist<T> 85b) +≡* (85a) ◁87b 87d ▷
void swap(DynDlist & l)
{
 std::swap(num_elem, l.num_elem);
 this->Dlink::swap(&l);
}

Uses DynDlist 85a.

Otra forma de modificación de una lista dinámica es la partición equitativa. Esto puede hacerse desde el método Dlink::split_list():

87d *(métodos públicos DynDlist<T> 85b) +≡* (85a) ◁87c 88a ▷
void split_list(DynDlist & l, DynDlist & r)
{
 Dlink::split_list(l, r);
 l.num_elem = r.num_elem = num_elem/2;

 if (num_elem % 2 == 1) // ¿es num_elem impar?
 l.num_elem++;

 num_elem = 0;
}

Uses DynDlist 85a.

2.4.9.1 Iterador de DynDlist<T>

Hasta el presente, la inserción, eliminación y otras operaciones de modificación están restringidas a los extremos de la lista. Si una aplicación requiere modificación en posiciones diferentes, entonces ésta puede servirse de un iterador aunado a operaciones especiales sobre el elemento actual:

88a *(métodos públicos DynDlist<T> 85b) +≡* (85a) ◁87d 91a▷
 class Iterator : public Dnode<T>::Iterator
 {
 DynDlist * list_ptr; // puntero a la lista
 int pos; // posición del elemento actual en la secuencia
(métodos iterador de DynDlist<T> 88b)
 };

Uses Dnode 83a and DynDlist 85a.

La clase guarda un apuntador a la lista con el propósito de mantener el contador de elementos; de esta manera se puede actualizar el contador cuando se ejecuten operaciones sobre el iterador que modifiquen la cantidad de elementos.

El atributo pos refleja la posición ordinal dentro de la secuencia del elemento actual y puede observarse mediante:

88b *(métodos iterador de DynDlist<T> 88b) ≡* (88a) 88c▷
 const int & get_pos() const { return pos; }

Para mantener correctamente el estado de este valor debemos sobrecargar las funciones de avance del iterador de manera tal que actualicen su valor:

88c *(métodos iterador de DynDlist<T> 88b) +≡* (88a) ◁88b 88d▷
 const int & next()
 {
 Dnode<T>::Iterator::next();
 pos++;
 return pos;
}

Uses Dnode 83a.

El manejo de la posición del elemento actual del iterador también requiere sobrecargar los métodos `reset_first()` y `reset_last()`.

Para crear un iterador sólo se requiere una lista o por copia de otro iterador:

88d *(métodos iterador de DynDlist<T> 88b) +≡* (88a) ◁88c 89a▷
 Iterator(DynDlist<T> & _list)
 : Dnode<T>::Iterator(_list), list_ptr(&_list), pos(0) {}

 Iterator(const Iterator & it)
 : Dnode<T>::Iterator(it), list_ptr(it.list_ptr), pos(it.pos) {}

 Iterator() : list_ptr(NULL) {}

 Iterator & operator = (const Iterator & it)
 {
 Dnode<T>::Iterator::operator = (it);
 list_ptr = it.list_ptr;
 pos = it.pos;
 return *this;
}

```
}
```

Uses Dnode 83a and DynDlist 85a.

has_current() se hereda de la clase base Dlink.

Dnode<T> trabaja en función de nodos y nuestro iterador debe trabajar en función de elementos de tipo T. Por esa razón debemos sobrecargar get_current():

89a *(métodos iterador de DynDlist<T> 88b) +≡* (88a) ◁88d 89b ▷
 T & get_current()
 {
 return Dnode<T>::Iterator::get_current()->get_data();
}

Uses Dnode 83a.

La inserción y supresión mediante el iterador de Dnode<T> es directa porque Dnode<T> maneja elementos de tipo Dnode<T> que poseen sus propias primitivas de inserción y supresión. En nuestro caso no es posible hacerlo porque manejamos elementos genéricos de tipo T. Así pues, requerimos que el iterador exporte funciones que permitan insertar y eliminar.

La inserción la presentamos bajo las mismas formas que DynDlist<T>:

89b *(métodos iterador de DynDlist<T> 88b) +≡* (88a) ◁89a 89c ▷
 void insert(const T & _data)
{
 Dnode<T>::Iterator::get_current()->insert(new Dnode<T>(_data));
 ++list_ptr->num_elem;
}
void append(const T & _data)
{
 Dnode<T>::Iterator::get_current()->append(new Dnode<T>(_data));
 ++list_ptr->num_elem;
}

Uses Dnode 83a.

Las otras maneras de insertar son listas enteras a partir del elemento actual:

89c *(métodos iterador de DynDlist<T> 88b) +≡* (88a) ◁89b 89d ▷
 void insert_list(const DynDlist & list)
{
 Dnode<T>::Iterator::get_current()->insert_list(&list);
 list_ptr->num_elem += list.num_elem;
 list.num_elem = 0;
}
void append_list(const DynDlist & list)
{
 Dnode<T>::Iterator::get_current()->append_list(&list);
 list_ptr->num_elem += list.num_elem;
 list.num_elem = 0;
}

Uses Dnode 83a and DynDlist 85a.

La supresión la vemos como la supresión del elemento actual:

89d *(métodos iterador de DynDlist<T> 88b) +≡* (88a) ◁89c 90a ▷
 T del()
{
 Dnode<T> * ptr = Dnode<T>::Iterator::get_current();

```

T ret_val      = ptr->get_data();
Dnode<T>::Iterator::next();
ptr->del();
delete ptr;
-list_ptr->num_elem;
return ret_val;
}

```

Uses Dnode 83a.

La operación del() puede ser muy restrictiva en el sentido de que avanza el iterador; además, lo hace en un solo sentido. Nos conviene, pues, ofrecer la primitivas de eliminación “contextuales”, es decir, el predecesor o sucesor del elemento actual:

90a *(métodos iterador de DynDlist<T> 88b) +≡* (88a) ◁89d 90b ▷

```

T remove_prev()
{
    Dnode<T> * curr_ptr = Dnode<T>::Iterator::get_current();
    Dnode<T> * ptr      = curr_ptr->remove_prev();
    T ret_val = ptr->get_data();
    delete ptr;
    -list_ptr->num_elem;
    return ret_val;
}
T remove_next()
{
    Dnode<T> * curr_ptr = Dnode<T>::Iterator::get_current();
    Dnode<T> * ptr      = curr_ptr->remove_next();
    T ret_val = ptr->get_data();
    delete ptr;
    -list_ptr->num_elem;
    return ret_val;
}

```

Uses Dnode 83a.

La última operación de modificación de una lista dinámica mediante un iterador puede ser muy útil en algunos casos; se trata del corte de una lista a partir de la posición actual del iterador:

90b *(métodos iterador de DynDlist<T> 88b) +≡* (88a) ◁90a

```

size_t cut_list(DynDlist & list)
{
    list_ptr->Dnode<T>::cut_list(Dnode<T>::Iterator::get_current(), &list);
    list.num_elem = list_ptr->num_elem - pos;
    list_ptr->num_elem -= pos;
    return list.num_elem;
}

```

Uses Dnode 83a and DynDlist 85a.

El iterador permite una manera concisa y expresiva de copiar listas:

90c *(copiar lista 90c) ≡* (91)

```

for (typename DynDlist<T>::Iterator itor(const_cast<DynDlist&>(list));
     itor.has_current(); itor.next())

    this->append(itor.get_current());

```

Uses DynDlist 85a.

Lo que permite implantar la asignación y el constructor copia:

91a $\langle \text{métodos públicos } \text{DynDlist} < T > \text{ 85b} \rangle + \equiv$ (85a) $\triangleleft 88a \ 91b \triangleright$

```

DynDlist < T > & operator = (const DynDlist & list)
{
    while (not this->is_empty()) // vaciar this
        this->remove_first();

    < copiar lista 90c >

    return *this;
}

```

Uses DynDlist 85a.

Del mismo modo, el constructor copia:

91b $\langle \text{métodos públicos } \text{DynDlist} < T > \text{ 85b} \rangle + \equiv$ (85a) $\triangleleft 91a \ 91c \triangleright$

```

DynDlist (const DynDlist & list) : Dnode < T > (list)
{
    this->reset();
    num_elem = 0;
    < copiar lista 90c >
}

```

Uses Dnode 83a and DynDlist 85a.

La llamada `this->reset()` es indispensable para asegurar que la lista esté lógicamente vacía. Si no se hiciera así, entonces el valor de `this`, que funge de nodo cabecera, correspondería a la copia de `list`.

Si la condición de lista imposibilita el acceso directo a sus elementos, a efectos de interfaz puede ser altamente deseable disponer del operador de acceso por posición:

91c $\langle \text{métodos públicos } \text{DynDlist} < T > \text{ 85b} \rangle + \equiv$ (85a) $\triangleleft 91b \triangleright$

```

T & operator [] (const size_t & n)
{
    typename DynDlist < T > :: Iterator it(*this);
    for (int i = 0; i < n and it.has_current(); i++, it.next()) ;
    return it.get_current();
}

```

Uses DynDlist 85a.

2.4.10 Aplicación: aritmética de polinomios

Consideremos un TAD que modelice polinomios y sus operaciones básicas. Tal TAD está realizado en el archivo $\langle \text{polinom.C} \rangle 91d$, el cual se define a continuación:

91d $\langle \text{polinom.C} \rangle 91d \equiv$

```

class Polinomio
{
    < Miembros privados Polinomio 93c >
    < interfaz de Polinomio 91e >
};

< Implementación de métodos Polinomio 94f >

```

Hay varias maneras de crear un polinomio, la primera es por omisión:

91e $\langle \text{interfaz de Polinomio} \rangle 91e \equiv$ (91d) $\triangleleft 92a \triangleright$

```

Polinomio();

```

Este constructor crea el polinomio $0x^0$.

Para construir un polinomio cualquiera, el punto de partida será un polinomio de un solo término:

92a $\langle\text{interfaz de Polinomio 91e}\rangle + \equiv$ (91d) $\triangleleft 91e \ 92b \triangleright$

```
Polinomio(const int& coef, const size_t & pot);
```

Este constructor instancia el polinomio $\text{coef } x^{\text{pot}}$; por ejemplo:

```
Polinomio p(20, 7);
```

Instancia un polinomio con valor $20x^7$.

Si queremos construir un polinomio más complejo, entonces podemos sumar varios términos mediante el operador de suma:

92b $\langle\text{interfaz de Polinomio 91e}\rangle + \equiv$ (91d) $\triangleleft 92a \ 92c \triangleright$

```
Polinomio operator + (const Polinomio&) const;
```

Por ejemplo, para construir el polinomio $20x^7 + 3x^3 + x^2 + 20$ podemos efectuar:

```
Polinomio p(Polinomio(20, 7) + Polinomio(3, 3) +
           Polinomio(1, 2) + Polinomio(20, 0));
```

Por razones de eficiencia que discutiremos posteriormente, es conveniente ofrecer la siguiente versión de la suma:

92c $\langle\text{interfaz de Polinomio 91e}\rangle + \equiv$ (91d) $\triangleleft 92b \ 92d \triangleright$

```
Polinomio& operator += (const Polinomio&);
```

Mediante este operador, el polinomio anterior puede construirse como sigue:

```
Polinomio p(20, 7);
p += Polinomio(3, 3) + Polinomio(1, 2) + Polinomio(20, 0);
```

El producto de dos polinomios se especifica mediante el operador *:

92d $\langle\text{interfaz de Polinomio 91e}\rangle + \equiv$ (91d) $\triangleleft 92c \ 92e \triangleright$

```
Polinomio operator * (const Polinomio&) const;
```

Otras operaciones que se delegan a ejercicios son:

92e $\langle\text{interfaz de Polinomio 91e}\rangle + \equiv$ (91d) $\triangleleft 92d \ 92f \triangleright$

```
Polinomio operator - (const Polinomio&) const;
```

```
Polinomio operator / (const Polinomio&) const;
```

```
Polinomio operator % (const Polinomio&) const;
```

Estas operaciones corresponden a la substracción, división y residuo.

Las aplicaciones que utilicen polinomios requerirán una forma de acceder a sus términos. Para ello debemos proveer un conjunto mínimo de primitivas. Un atributo esencial de un polinomio es conocer la cantidad de términos:

92f $\langle\text{interfaz de Polinomio 91e}\rangle + \equiv$ (91d) $\triangleleft 92e \ 92g \triangleright$

```
const size_t & size() const;
```

Requerimos conocer cada uno de los términos; esto lo podemos hacer mediante:

92g $\langle\text{interfaz de Polinomio 91e}\rangle + \equiv$ (91d) $\triangleleft 92f \ 93a \triangleright$

```
size_t get_power(const size_t & i) const;
```

`get_power()` retorna la potencia correspondiente al i -ésimo término diferente de cero asumiendo que los términos del polinomio están ordenados desde la mayor hasta la menor potencia. El coeficiente se obtiene mediante:

93a $\langle\text{interfaz de Polinomio } 91e\rangle + \equiv$ (91d) $\triangleleft 92g \ 93b \triangleright$
`const int & get_coef(const size_t & i) const;`

El grado del polinomio puede consultarse mediante:

93b $\langle\text{interfaz de Polinomio } 91e\rangle + \equiv$ (91d) $\triangleleft 93a \triangleright$
`const size_t & get_degree() const;`

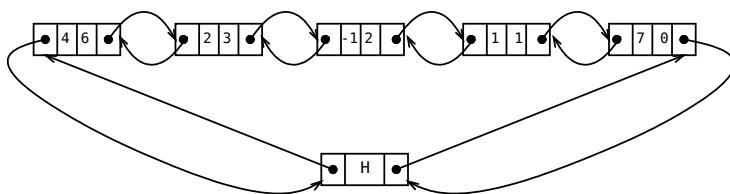
2.4.10.1 Implementación de polinomios

Una primera tentación como estructura de dato para representar un polinomio es un arreglo. Para un polinomio de grado n se reserva un arreglo de dimensión $n + 1$, el cual contiene los coeficientes. No es necesario almacenar la potencia, pues ésta se representa mediante el propio índice del arreglo.

En apariencia, el arreglo ofrece ventajas fenomenales; por ejemplo, la suma es directamente la suma de vectores. Lamentablemente, el arreglo puede desperdiciar una cantidad de espacio impresionante; por ejemplo, el polinomio $20x^{100}$ requiere un arreglo de 101 elementos para almacenar un solo coeficiente; un desperdicio del 99%.

Los polinomios son muy diversos. La multiplicación y división arrojan polinomios de grados diferentes a sus operandos. En un ambiente donde se efectúen muchas operaciones, es de esperar que dos polinomios tengan sus grados diferentes. En añadidura, muchas veces un polinomio es esparcido, es decir, varios de sus coeficientes son nulos. Por estas razones, una lista enlazada que sólo almacene términos diferentes de cero es la estructura idónea para representar un polinomio.

La figura 2.15 ilustra la representación con listas doblemente enlazadas del polinomio $4x^6 + 2x^3 - 1x^2 + x + 7$. La lista está ordenada desde la mayor hasta la menor potencia. El orden es esencial para facilitar la eficiencia de los algoritmos.



Defines:

Termino, used in chunks 94–98.

La lista enlazada contiene entonces términos:

94a $\langle Miembros\ privados\ Polinomio\ 93c \rangle + \equiv$ (91d) $\triangleleft 93c\ 94c \triangleright$
 $\text{DynDlist}\langle\text{Termino}\rangle\ \text{terminos};$
 Uses DynDlist 85a and Termino 93c.

Hay dos formas de construir un término:

94b $\langle Miembros\ Término\ 94b \rangle + \equiv$ (93c) 94d
 $\text{Termino}() : \text{coef}(0), \text{pot}(0) \{\}$
 $\text{Termino}(\text{const int} & c, \text{const size_t} & p) : \text{coef}(c), \text{pot}(p) \{\}$
 Uses Termino 93c.

A nivel privado es importante construir un Polinomio a partir de un término:

94c $\langle Miembros\ privados\ Polinomio\ 93c \rangle + \equiv$ (91d) $\triangleleft 94a\ 98b \triangleright$
 $\text{Polinomio}(\text{const Polinomio}::\text{Termino} & \text{termino})$
 $\{$
 $\text{terminos.append}(\text{termino});$
 $\}$
 Uses Termino 93c.

La suma de términos se define como sigue:

94d $\langle Miembros\ Término\ 94b \rangle + \equiv$ (93c) 94b 94e
 $\text{Termino} \& \text{operator } += (\text{const Termino} & \text{der})$
 $\{$
 $\text{coef} += \text{der.coef};$
 $\text{return } *this;$
 $\}$
 Uses Termino 93c.

La multiplicación forzosamente debe producir un nuevo término, por esa razón debemos implantar estrictamente el operador *:

94e $\langle Miembros\ Término\ 94b \rangle + \equiv$ (93c) 94d
 $\text{Termino} \operatorname{operator} * (\text{const Termino} & \text{der}) \text{ const}$
 $\{$
 $\text{return } \text{Termino}(\text{coef} * \text{der.coef}, \text{pot} + \text{der.pot});$
 $\}$
 Uses Termino 93c.

Definida la construcción de un término podemos definir la construcción de polinomios:

94f $\langle Implementación\ de\ métodos\ Polinomio\ 94f \rangle + \equiv$ (91d) 95a
 $\text{Polinomio}::\text{Polinomio}(\text{const int} & \text{coef}, \text{const size_t} & \text{pot})$
 $\{$
 $\text{terminos.append}(\text{Termino}(\text{coef}, \text{pot}));$
 $\}$
 $\text{Polinomio}::\text{Polinomio}() \{ /* \text{empty} */ \}$
 Uses Termino 93c.

Suma de polinomios

La suma de dos polinomios $P_1 = \sum_{i=0}^n c_i x^i$ y $P_2 = \sum_{j=0}^m d_j x^j$ se define como:

$$P_1 + P_2 = \sum_{k=0}^{\max(n,m)} (c_k + d_k) x^k.$$

Es fácil obtener un algoritmo eficiente, pues las listas están ordenadas por potencia. En este sentido es preferible implementar el operador `+=`, pues así nos ahorraremos el crear nuevos términos para el polinomio resultado. De este modo, el operador `+` podemos escribirlo en función de `+=` como sigue:

95a *(Implementación de métodos Polinomio 94f)* \equiv (91d) ◁ 94f 95b ▷

```
Polinomio Polinomio::operator + (const Polinomio & der) const
{
    Polinomio ret_val(*this); // valor de retorno operando derecho
    ret_val += der; // súmelo operando derecho
    return ret_val;
}
```

La suma es extremadamente simple porque subyace en la llamada al operador `+=`, el cual posee la siguiente estructura:

95b *(Implementación de métodos Polinomio 94f)* \equiv (91d) ◁ 95a 98a ▷

```
Polinomio & Polinomio::operator += (const Polinomio& der)
{
    (Implantación de Polinomio::operator += 95c)
    return *this;
}
```

Antes de efectuar la suma como tal, debemos verificar que ninguno de los dos polinomios corresponda al elemento neutro:

95c *(Implantación de Polinomio::operator += 95c)* \equiv (95b) 95d ▷

```
if (der.terminos.is_empty())
    return *this;

if (terminos.is_empty())
{
    *this = der;
    return *this;
}
```

Si ambos polinomios no son nulos, entonces debemos recorrerlos. Para ello usamos un iterador por cada polinomio operando:

95d *(Implantación de Polinomio::operator += 95c)* \equiv (95b) ◁ 95c 95e ▷

```
DynDlist<Termino>::Iterator it_izq(terminos);
DynDlist<Termino>::Iterator it_der(const_cast<DynDlist<Termino>&>(der.terminos));
Uses DynDlist 85a and Termino 93c.
```

La figura 2.16 ejemplifica el polinomio $6x^6 + 7x^5 - x^4 + 3x^2 + 4x$ como operando izquierdo, y $8x^8 + x^6 - 2x^2 + 2$ como operando derecho.

Ahora recorreremos las listas de términos hasta que uno de los iteradores alcance su fin de lista:

95e *(Implantación de Polinomio::operator += 95c)* \equiv (95b) ◁ 95d 97b ▷

```
while (it_izq.has_current() and it_der.has_current())
{
    (Procesar términos actuales 96a)
}
```

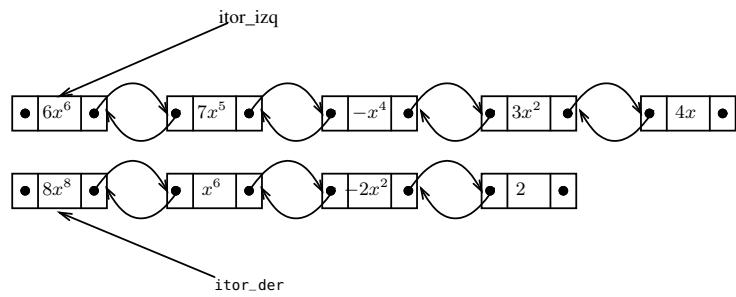


Figura 2.16: Suma de polinomios

Sean izq y der las potencias de los términos actuales de los iteradores izquierdo y derecho respectivamente:

96a *<Procesar términos actuales 96a>*≡ (95e) 96b▷
const size_t & izq = it_izq.get_current().pot;
const size_t & der = it_der.get_current().pot;
izq y der deben procesarse según los siguientes casos:

- Si la potencia del izq es menor que der, entonces der estará presente en el resultado. Como la lista está ordenada, es seguro que this no contiene un término con la misma potencia de der. Esta es la situación con los términos $6x^6$ y $8x^8$ de los polinomios izquierdo y derecho respectivamente.

Debemos crear un nuevo término copia de der e insertarlo en this (el polinomio izquierdo). La copia de der debe preceder a izq en this, pues el resultado debe estar ordenado.

Las acciones anteriores se traducen al siguiente código:

```

96b   ⟨Procesar términos actuales 96a⟩+≡ (95e) ◁ 96a 96c▷
      if (izq < der)
        {      // insertar a la izquierda del actual de it_izq
          it_izq.append(Termino(it_der.get_current().coef, der));
          it_der.next(); // ver próximo término de polinomio derecho
          continue;
        }

```

append() sobre `it_izq` garantiza que el nuevo término sea predecesor del actual izquierdo. Luego, avanzamos el iterador derecho, pues el término de potencia derecha ya ha sido procesado. Finalmente, repetimos el procedimiento.

2. Si las potencias son iguales, entonces debemos sumar los coeficientes, avanzar ambos iteradores y repetir el procedimiento. Esta es la situación con los términos $6x^6$ y x^6 de los polinomios izquierdo y derecho respectivamente.

```

96c   ⟨Procesar términos actuales 96a⟩+≡                               (95e) ▷ 96b 97a▷
      if (izq == der)
          {           // calcular coeficiente resultado
              it izq.get_current() += it der.get_current(); // += Termino

```

```
    it_der.next(); // avanzar a sig término polinomio derecho
    if (it_izq.get_current().coef == 0) // ¿suma anula término?
        { // sí, borrarlo de polinomio izquierdo (coeficiente 0)
            it_izq.del();
            continue;
        }
}
```

Uses Termino 93c.

3. En el último caso ($izq > der$), izq será parte del resultado. Puesto que izq ya pertenece a `this`, simplemente avanzamos el iterador `izquierdo` y repetimos el procedimiento. Esta es la situación con los términos $7x^5$ y $-2x^2$ de los polinomios `izquierdo` y `derecho` respectivamente.

97a *⟨Procesar términos actuales 96a⟩+≡
 it_izq.next();*

(95e) <96c

El proceso iterativo anterior culmina cuando alguno de los iteradores alcanza el final de una de las listas. En este caso queda por recorrer una lista. Si la lista por recorrer es la izquierda, no debemos hacer nada, pues sus elementos, que son parte del resultado, ya están incluidos en `this`. Si, por el contrario, la lista que resta por recorrer es la derecha, entonces debemos copiar todos sus términos restantes e insertarlos secuencialmente en la lista izquierda:

97b *⟨Implantación de Polinomio::operator += 95c⟩+≡*

(95b) <95e

```

while (it_der.has_current())
{
    // copia términos restantes derecho a izquierdo
    terminos.append(Termino(it_der.get_current().coef,
                           it_der.get_current().pot));
    it_der.next();
}

```

Uses Terminus 03c

Multiplicación de polinomios

La multiplicación de dos polinomios $P_1 = \sum_{i=0}^n c_i x^i$ y $P_2 = \sum_{j=0}^m d_j x^j$ se define como:

$$P_1 P_2 = \sum_{i=0}^n c_i x^i \left(\sum_{j=0}^m k_j x^j \right).$$

Esta expresión sugiere un algoritmo basado en suma de productos parciales, el cual puede resumirse como sigue:

Algoritmo 2.1 (Multiplicación de dos polinomios)

Algoritmo 2.1 (Multiplicación de dos polinomios)
 La entrada son dos polinomios $P_1 = \sum_{i=0}^n c_i x^i$ y $P_2 = \sum_{j=0}^m d_j x^j$ de grados n y m respectivamente.

La salida es un polinomio R correspondiente al producto $P_1 P_2$

1. Instancie un polinomio nulo R.
 2. Repita para $i = 0$ hasta n

- (a) Sea $R' = c_i x^i \times P_2$
 (b) $R = R + R'$

Ahora podemos codificar el operador * completamente reminiscente del algoritmo 2.1:

98a *Implementación de métodos Polinomio 94f* +≡ (91d) ↣ 95b

```
Polinomio Polinomio::operator * (const Polinomio & der) const
{
    Polinomio result;
    if (terminos.is_empty() or der.terminos.is_empty())
        return result;

    for (DynDlist<Termino>::Iterator
          it_izq(const_cast<DynDlist<Termino>&)(terminos));
        it_izq.has_current(); it_izq.next())
        result += der.multiplicado_por(it_izq.get_current());

    return result;
}
```

Uses DynDlist 85a and Termino 93c.

El if verifica que si alguno de los polinomios es nulo, entonces el producto es nulo. El for es exactamente la versión codificada del algoritmo 2.1: cada término del polinomio izquierdo se multiplica enteramente por el polinomio derecho y el resultado es acumulado en el polinomio result. El fragmento der.por_termino(it_izq.get_current()) corresponde a la multiplicación de un polinomio por un término, operación que definimos a continuación:

98b *Miembros privados Polinomio 93c* +≡ (91d) ↣ 94c

```
Polinomio multiplicado_por(const Termino & term) const
{
    Polinomio result;
    if (terminos.is_empty() or term.coef == 0)
        return result;

    for (DynDlist<Termino>::Iterator it((DynDlist<Termino>&) terminos);
         it.has_current(); it.next())
        result.terminos.append(Termino(it.get_current().coef * term.coef,
                                       it.get_current().pot + term.pot));
    return result;
}
```

Uses DynDlist 85a and Termino 93c.

El algoritmo es sencillo. Se instancia un polinomio result con valor inicial igual al polinomio nulo. Luego, por cada término del polinomio se multiplica por el término operando y el resultado se inserta ordenadamente en result.

2.5 Pilas

Una pila es una abstracción de flujo consistente de una secuencia de elementos en la cual las operaciones de inserción, consulta y eliminación se ejecutan por un solo extremo. La

propiedad esencial es que el último elemento en insertarse siempre será el primer elemento en eliminarse. Tal propiedad se conoce como UEPS (Último en Entrar, Primero en Salir)¹⁵.

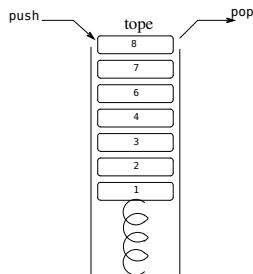


Figura 2.17: Abstracción de una pila

Gráficamente podemos ver una pila como un recipiente con una sola entrada tal como en la figura 2.17. Cuando se inserta un elemento, los elementos previamente insertados se “empujan” hacia abajo. Por esa razón, la operación de inserción se denomina comúnmente “push”. El extremo del recipiente, por donde entran y salen los elementos se denomina “tope”, el cual es el único punto de manipulación de la pila.

Alegóricamente hablando, cuando se activa una supresión, el resorte empuja los elementos hacia arriba y el elemento situado en el tope “salta” hacia afuera. Por esa razón, a la supresión de una pila se le denomina comúnmente “pop”.

Una pila se utiliza cuando se requiere un flujo de procesamiento de elementos inverso a la secuencia de observación o de entrada, es decir, desde el más recientemente observado hasta el más antiguamente observado. Cualquier patrón de procesamiento que visite elementos de un conjunto en este orden es susceptible de implantarse con una pila.

La vida real ofrece algunos ejemplos de la disciplina pila. Cualquiera de nosotros habrá experienciado el refrán popular que reza “los últimos serán los primeros”.

El lavado manual de platos en un fregadero doméstico sigue la disciplina pila. Probablemente, los platos se “apilan” según el orden de culminación de la comida, es decir, el tope de la pila contiene el plato de la última persona en culminar su comida. Cuando lavamos los platos, los enjabonamos y los apilamos en el segundo recipiente del fregadero. Posteriormente, enjuagamos los platos y los apilamos en el escurridor. Este ejemplo utiliza tres pilas y refleja un tipo de procesamiento en el cual lo importante es culminar eficazmente el trabajo y no el orden en que los platos son procesados.

2.5.1 Representaciones de una pila en memoria

Hay dos métodos para representar una pila: arreglos y listas enlazadas. En ambas representaciones, los elementos se disponen secuencialmente según el mismo orden de la pila.

En un arreglo se requiere mantener el índice al último elemento. La primera entrada almacena el elemento más antiguo, mientras que el índice referencia el tope más uno. La figura 2.18 ilustra una pila de 5 elementos contenida en un arreglo de 11 elementos.

En una lista enlazada requerimos mantener un apuntador al primer nodo. Puesto que sólo nos interesa el tope, la secuencia de la lista está invertida; es decir, el primer nodo

¹⁵En inglés LIFO (last in, first out).

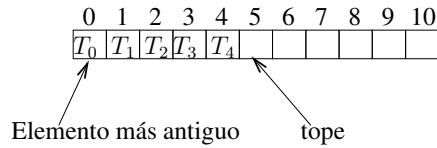


Figura 2.18: Representación de una pila mediante un arreglo

de la lista almacena el tope y el último almacena el más antiguo. La figura 2.19 ilustra la representación.

La selección de representación dependerá de las circunstancias. En lo que atañe al espacio, quizás un factor considerable es la máxima cantidad de elementos que pueda contener la pila, la cual depende de la aplicación.

Las listas acarrean un sobrecoste por elemento debido al apuntador adicional. Si se conoce la máxima cantidad de elementos y la aplicación la aprovecha, entonces está claro que el arreglo es más compacto que la lista.

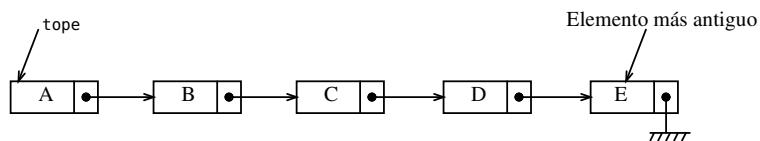


Figura 2.19: Representación de una pila con listas enlazadas

Otro factor a evaluar es la variación máxima del tamaño de la pila. Podemos conocer el máximo, pero si éste es muy poco probable, entonces, en promedio, puede ser más barato en espacio una representación con listas que enfoque el tamaño promedio. La selección se efectuaría considerando la desviación típica del tamaño de la pila. Sea \bar{l} el tamaño promedio de la pila, σ_l la desviación típica, l_{\max} el tamaño máximo que puede alcanzar la pila, T_s el espacio ocupado por un elemento de la pila y p_s el espacio ocupado por un apuntador, entonces, si

$$(\bar{l} + \sigma_l)(T_s + p_s) \leq l_{\max} T_s , \quad (2.20)$$

entonces, en promedio, la lista ocupará menos espacio que el arreglo.

En lo que concierne el tiempo de ejecución, el arreglo es la representación más rápida. Puesto que los elementos del arreglo son secuenciales en memoria, la localidad espacial y temporal es aprovechada y el cache del computador logra su cometido. Este no es el caso con las listas donde los nodos estarán en direcciones de memoria diferentes.

Por otra parte, cuando se inserta o se suprime un elemento en una lista enlazada, puede invocarse al manejador de memoria. El manejo de memoria impone un sobrecoste adicional en una lista respecto al arreglo. La eficiencia del manejador de memoria para apartar un bloque es muy variable y puede ser proporcional a la cantidad de bloques apartada. La misma consideración es válida cuando se libera un bloque.

Una pila puede instrumentarse directamente mediante el tipo `DynDlist<T>` definido en § 2.4.9 (Pág. 84). Si miramos el primer elemento de la lista como el tope, entonces `push(item)` equivale a `insert(item)`, `pop()` a `remove_first()` y `top()` a `get_first()`.

2.5.2 El TAD ArrayStack<T> (pila vectorizada)

El desempeño de una pila implantada mediante un arreglo es tan excelente que vale la pena hacer explícito el mecanismo de implantación.

`ArrayStack<T>` dispara excepciones cuando se excede la capacidad del arreglo o cuando se intenta acceder el tope de una pila vacía. Disparar estas excepciones tiene un ligero coste en desempeño que puede ser importante si la pila se utiliza muy a menudo.

Existen muchas aplicaciones en las cuales se puede conocer el tamaño máximo de la pila y en las que es innecesario considerar el desborde. Por otra parte, un desborde negativo es un error de programación a verificar en las primeras etapas de un proyecto. Por esa razón, el TAD `FixedStack<T>` modeliza las mismas primitivas que `ArrayStack<T>` con la diferencia de que no se efectúan verificaciones y no se disparan excepciones. De esta manera, aplicaciones que conozcan el máximo tamaño de pila se benefician de la ganancia en velocidad dada por la ausencia de verificaciones.

Pilas implantadas con arreglos se definen en el archivo `<tpl_arrayStack.H 101a>`, el cual exporta dos tipos principales: `ArrayStack<T>` y `FixedStack<T>`:

101a *<tpl_arrayStack.H 101a>*≡
 template <typename T, const size_t dim = 100> class ArrayStack
 {
<miembros privados de pila vectorizada 101b>
<miembros públicos de ArrayStack<T> 101c>
 };
 template <typename T, const size_t dim = 100> class FixedStack
 Defines:
 ArrayStack, used in chunks 101c, 110, 118e, 246–48, and 509b.
 FixedStack, used in chunks 180c, 473b, and 492b.

Los dos tipos de datos poseen los mismos atributos; éstos son:

101b *<miembros privados de pila vectorizada 101b>*≡ (101a)
 T array[dim];
 size_t head;

array es un arreglo estático de elementos de tipo T con dimensión dim. dim es un parámetro de la plantilla que representa la dimensión del arreglo y con un valor por omisión de 100.

head es el índice de la próxima entrada disponible. También indica la cantidad de elementos de la pila. head - 1 es el índice del elemento tope.

La construcción de una pila sólo requiere inicializar head:

101c *<miembros públicos de ArrayStack<T> 101c>*≡ (101a) 101d▷
 ArrayStack() : head(0) { /* empty */ }

Uses `ArrayStack` 101a.

Para insertar un elemento se usa el método `push()`:

101d *<miembros públicos de ArrayStack<T> 101c>*+≡ (101a) ▷101c 102a▷
 T & push(const T & data)
 {
<push data 101e>
 }

101e *<push data 101e>*≡ (101d)
 array[head++] = data;
 return array[head - 1];

En algunas aplicaciones se requiere apartar espacio en la pila sin que aún se conozcan los valores de inserción. Si bien esto puede lograrse mediante una serie consecutiva de pushes de datos “vacíos”, es más eficiente ofrecer una sola operación. Tal operación se denomina `pushn()` y se implanta como sigue:

102a *(miembros públicos de ArrayStack<T> 101c) +≡* (101a) ◁ 101d 102c ▷
`T & pushn(const size_t & n = 1)`
`{`
 `<apartar n elementos 102b>`
`}`

A partir del tope, el usuario de `ArrayStack<T>` puede referenciar a los `n` elementos insertados que no han sido inicializados y así asignarles su valor cuando sea necesario.

<apartar n elementos 102b> es muy simple con un arreglo; basta con incrementar `head`:

102b *(apartar n elementos 102b) ≡* (102a)
`head += n;`
`return array[head - 1];`

Para sacar un elemento de la pila utilizamos `pop()`:

102c *(miembros públicos de ArrayStack<T> 101c) +≡* (101a) ◁ 102a 102e ▷
`T pop()`
`{`
 `<pop data 102d>`
`}`

102d *(pop data 102d) ≡* (102c)
`return array[-head];`

Por razones de eficiencia, también es deseable ofrecer una primitiva que libere varios elementos en una sola operación:

102e *(miembros públicos de ArrayStack<T> 101c) +≡* (101a) ◁ 102c 102g ▷
`T popn(const int & n)`
`{`
 `<liberar n elementos 102f>`
`}`

En algunos contextos, a esta operación se le llama “multipop”.

102f *(liberar n elementos 102f) ≡* (102e)
`head -= n;`
`return array[head];`

Hay varias formas de consultar la pila; todas se efectúan respecto al tope y retornan una referencia a un elemento dentro de la pila. La primera forma es la del elemento en el tope:

102g *(miembros públicos de ArrayStack<T> 101c) +≡* (101a) ◁ 102e 103a ▷
`T & top()`
`{`
 `<retornar referencia al tope 102h>`
`}`

102h *(retornar referencia al tope 102h) ≡* (102g)
`return array[head - 1];`

Otro tipo de consulta, menos frecuente pero posible en algunas aplicaciones, es conocer el *i*-ésimo elemento respecto al tope:

103a $\langle \text{miembros públicos de } \text{ArrayStack}\langle T \rangle \text{ 101c} \rangle + \equiv$ (101a) $\triangleleft 102g \ 103d \triangleright$
 $T \& \text{top}(\text{const int} \& i)$
 $\{$
 $\quad \langle \text{referencia al } i\text{-ésimo respecto a tope 103b} \rangle$
 $\}$

Una vez validado el rango de acceso, el elemento es accedido mediante:

103b $\langle \text{referencia al } i\text{-ésimo respecto a tope 103b} \rangle \equiv$ (103a)
 $\text{return array}[\text{head} - i - 1];$

En algunas ocasiones una pila es reutilizable a condición de que ésta sea previamente vaciada. Vaciar la pila implantada con un arreglo es una operación extremadamente rápida:

103c $\langle \text{vaciar pila 103c} \rangle \equiv$ (103d)
 $\text{head} = 0;$

Esta facilidad merece ofrecerse en una primitiva:

103d $\langle \text{miembros públicos de } \text{ArrayStack}\langle T \rangle \text{ 101c} \rangle + \equiv$ (101a) $\triangleleft 103a \ 103e \triangleright$
 $\text{void empty()} \{ \langle \text{vaciar pila 103c} \rangle \}$

Podemos consultar un predicado que nos diga si la pila está o no vacía:

103e $\langle \text{miembros públicos de } \text{ArrayStack}\langle T \rangle \text{ 101c} \rangle + \equiv$ (101a) $\triangleleft 103d \ 103g \triangleright$
 $\text{bool is_empty()} \text{ const } \{ \langle \text{pila vacía? 103f} \rangle \}$

103f $\langle \text{pila vacía? 103f} \rangle \equiv$ (103e)
 $\text{return head} == 0;$

La cantidad de elementos de la pila es directamente el valor de head:

103g $\langle \text{miembros públicos de } \text{ArrayStack}\langle T \rangle \text{ 101c} \rangle + \equiv$ (101a) $\triangleleft 103e \triangleright$
 $\text{const size_t} \& \text{size()} \text{ const } \{ \text{return head;} \}$

A nivel de interfaz, $\text{FixedStack}\langle T \rangle$ es idéntica a $\text{ArrayStack}\langle T \rangle$. Por razones de compatibilidad es preferible definir las mismas excepciones de $\text{ArrayStack}\langle T \rangle$ en las primitivas de $\text{FixedStack}\langle T \rangle$. De este modo, $\text{FixedStack}\langle T \rangle$ es aplicable en cada sitio donde se utilice $\text{ArrayStack}\langle T \rangle$.

A nivel de implantación, $\text{FixedStack}\langle T \rangle$ es muy similar a $\text{ArrayStack}\langle T \rangle$. La diferencia principal reside en que $\text{FixedStack}\langle T \rangle$ no efectúa verificación de desborde ni dispara excepciones. De resto, cada primitiva de $\text{FixedStack}\langle T \rangle$ es idéntica a su par en $\text{ArrayStack}\langle T \rangle$.

2.5.3 El TAD $\text{ListStack}\langle T \rangle$ (pila con listas enlazadas)

El TAD $\text{ListStack}\langle T \rangle$ modeliza una pila implantada mediante listas enlazadas. Al conocer el tipo de implantación, el usuario conoce los costes en espacio y tiempo, así como las ganancias en versatilidad impartidas por la “naturaleza” dinámica de las listas enlazadas.

$\text{ListStack}\langle T \rangle$ se basa en el TAD $\text{Snode}\langle T \rangle$, el cual, en esencia, contiene todo lo necesario para manejar las listas. $\text{ListStack}\langle T \rangle$ se define en el archivo $\langle \text{tpl_listStack.H 103h} \rangle$:

103h $\langle \text{tpl_listStack.H 103h} \rangle \equiv$
 $\text{template } \langle \text{typename } T \rangle \text{ class ListStack : private Snode}\langle T \rangle$
 $\{$

```
    ⟨atributos de ListStack<T> 104a⟩  
    ⟨métodos públicos de ListStack<T> 104b⟩  
};
```

Mediante de

Mediante derivación de `StackNode<T>`, `this` funge de nodo cabecera de la lista cuyo primer elemento siempre será el nodo tope de la pila.

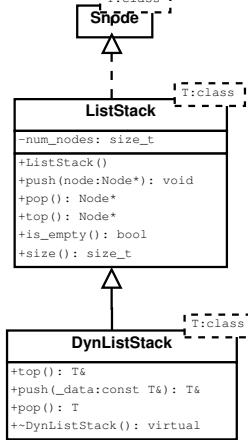


Figura 2.20: Diagrama UML de las clases vinculadas a ListStack<T>

ListStack<T> posee un atributo único:

104a *atributos de ListStack*<*T*> 104a}≡ (103h)
 size_t num_nodes;

el cual contabiliza la cantidad de nodos que posee la pila.

ListStack<T> exporta dos sinónimos de nodo:

104b *⟨métodos públicos de ListStack<T> 104b⟩* ≡ (103h) 104c▷
 typedef Snode<T> Node;

Uses Snode 68b.

A nivel de interfaz, las operaciones de ListStack<T> manejan “nodos” de la pila. Item es un sinónimo ofrecido por compatibilidad con versiones anteriores.

El constructor `ListStack<T>` sólo se remite a iniciar el contador de nodos:

La operación `push()` se define en función de un `ListStack<T>::Node`:

```
void push(Node * node)
{
    ++num_nodes;
    this->insert_next(node);
}
```

`push()` no genera ninguna excepción, pues su única responsabilidad es enlazar el nodo a la lista. Puesto que `this` es la cabecera, `insert_next()` siempre insertará `node` al frente de la lista, el cual siempre se corresponde con el tope.

Suprimir el tope equivale a suprimir el nodo del frente de la lista. Esto es directo según la interfaz de Snode<T>:

105a *(métodos públicos de ListStack<T> 104b) +≡* (103h) ◁ 104d 105b ▷
 Node * pop()
 {
 -num_nodes;
 return this->remove_next();
}

El tope puede consultarse mediante:

105b *(métodos públicos de ListStack<T> 104b) +≡* (103h) ◁ 105a
 Node * top() const
{
return static_cast<Node*>(this->get_next());
}

2.5.4 El TAD DynListStack<T>

El TAD ListStack<T> maneja pilas en función de nodos de una lista enlazada. Esto obliga al usuario a controlar y verificar detalles del manejo de memoria. La responsabilidad de ListStack<T> es el manejo de los nodos; la del cliente es apartar y liberar los nodos. Como hemos aprendido en ocasiones anteriores, manejar los enlaces, tipos y memoria puede hacerse bajo un solo TAD, derivado de TAD más sencillos.

El TAD DynListStack<T> cumple el cometido mencionado en el párrafo anterior. DynListStack<T> define una pila implantada con listas enlazadas simples y con manejo de memoria incluido. DynListStack<T> está definido en el archivo *<tpl_dynListStack.H 105c>*:

105c *<tpl_dynListStack.H 105c>*
 template <typename T> class DynListStack : public ListStack<T>
{
(métodos de DynListStack<T> 105d)
};

Defines:

DynListStack, used in chunks 106c, 188, 645b, 647b, 649, 650, 652b, and 764b.

DynListStack<T> almacena datos del tipo genérico T. Al derivar públicamente de ListStack<T>, DynListStack<T> hereda gran parte de su implantación y de su interfaz. El manejo de listas lo aporta entonces ListStack<T>. Las primitivas que no dependen del tipo T, *is_empty()* y *size()*, se heredan directamente. Las otras primitivas, que sí dependen de ListStack<T>, deben sobrecargarse para que manipulen datos de tipo T.

Comenzamos por las primitivas más sencillas:

105d *(métodos de DynListStack<T> 105d) ≡* (105c) 106a ▷
 T & top() const
{
return ListStack<T>::top()->get_data();
}

top() consulta el nodo tope de la clase base y retorna una referencia al dato.

`push()` debe insertar un dato de tipo `T`. Esto requiere apartar el nodo, copiarle el dato e invocar al `push()` de `ListStack<T>`:

106a *(métodos de DynListStack<T> 105d) +≡* (105c) ◁ 105d 106b ▷

```

T & push(const T & item)
{
    typename ListStack<T>::Node *ptr =
        new typename ListStack<T>::Node (item);
    ListStack<T>::push(ptr);
    return ptr->get_data();
}

```

`push()` retorna una referencia al dato contenido en el tope.

La labor de `pop()` es extraer el nodo de `ListStack<T>`, copiar el dato a retornar y liberar la memoria. Esta conducta se especifica como sigue:

106b *(métodos de DynListStack<T> 105d) +≡* (105c) ◁ 106a 106c ▷

```

T pop()
{
    typename ListStack<T>::Node* ptr = ListStack<T>::pop();
    T retVal = ptr->get_data();
    delete ptr;
    return retVal;
}

```

Puesto que `DynListStack<T>` es responsable del manejo de memoria, `DynListStack<T>::~DynListStack` debe liberar toda la memoria. Así pues, nos aseguramos de vaciar toda la pila en tiempo de destrucción:

106c *(métodos de DynListStack<T> 105d) +≡* (105c) ◁ 106b

```

virtual ~DynListStack()
{
    while (not this->is_empty())
        pop();
}

```

Uses `DynListStack 105c`.

Notemos que `is_empty()` pertenece a `ListStack<T>` mientras que `pop()` a `DynListStack<T>`.

2.5.5 Aplicación: un evaluador de expresiones aritméticas infijas

En el dominio de la matemática hay varias maneras de representar una operación binaria. Cada una asume una convención acerca de la posición del operador y los operandos. En notación prefija, el operador precede a los operandos. En notación sufija¹⁶, el operador sucede a los operandos. En notación infija, el operador se pone entre los dos operandos. Por ejemplo, podemos representar la suma de dos operandos `x` e `y` como sigue:

- Prefija: `+xy`
- Sufija: `xy+`
- Infija: `x + y`

¹⁶En inglés es traducido como *postfix*. De allí que algunos textos la denominen postfija.

La notación prefija se aproxima a las matemáticas tradicionales. El resultado de evaluar una función f con dos operandos x e y se denota comúnmente como $f(x, y)$. Del mismo modo, la composición funcional se denota como $h(f(x, y))$.

Las notaciones prefija y sufija permiten una cantidad arbitraria de operandos. Esta es una gran ventaja para denotar funciones multivariadas de índole cualquiera. Además, cuando hay varias operaciones, las notaciones prefija y sufija permiten expresar todas las operaciones sin necesidad de paréntesis. Por ejemplo, la forma sufija de $x * (y + z)$ es $yz + x*$.

Para evaluar una expresión prefija o sufija se utiliza una pila. El algoritmo para evaluar expresiones sufijas se describe como sigue:

Algoritmo 2.2 (Evaluación de expresión sufija) La entrada del algoritmo es una secuencia conteniendo una expresión sufija compuesta por operadores y operandos numéricos.

La salida es el valor correspondiente al resultado final.

El algoritmo utiliza una pila para guardar resultados parciales y una variable llamada "ficha actual" o simplemente la ficha. La ficha contiene el símbolo, operando u operador, que se está procesando.

Algoritmo

1. Repita mientras la ficha no esté al final de la secuencia.

- (a) Si la ficha es un operador \Rightarrow saque dos valores de la pila, efectúe la operación dada por el operador leído y almacene el resultado en la pila.
- (b) De lo contrario, si la ficha es operando \Rightarrow métalo en la pila.

2. Cuando se alcance el final de la secuencia, el resultado almacenado en la pila es el resultado de la expresión. Si la pila está vacía o ésta contiene más de dos valores, entonces la expresión sufija contenía algún error.

Un algoritmo similar puede deducirse para evaluar expresiones prefijas.

La notación sufija junto con el algoritmo 2.2 es común en algunas calculadoras de bolsillo, antiguos procesadores y algunos lenguajes de programación.

Por supuesto, la notación infija es la más familiar, pero ésta sólo permite a lo sumo dos operandos. En añadidura, la precedencia de las operaciones puede requerir el uso de paréntesis. A causa de esto, expresiones complicadas son normalmente más largas en forma infija que sus equivalentes prefijo o sufijo. Este hecho sugiere que manipular y evaluar expresiones infijas es más costoso en espacio y en tiempo que en las otras dos notaciones. Posiblemente por esta razón, algunos fabricantes de calculadoras de mano prefieren la notación sufija.

Existe un algoritmo muy eficiente para evaluar una expresión infija, el cual requiere dos pilas. Una pila almacena operandos y resultados parciales y la otra almacena operadores y paréntesis que representan la precedencia.

Antes de explicar el algoritmo requerimos alguna maquinaria de base que efectúe el procesamiento de la cadena de caracteres y que nos indique si estamos en presencia de

un operando u operador. Para ello definimos los posibles valores de “fichas” que pueden encontrarse en una cadena:

108a *(tipo de ficha 108a)≡*

```
enum Token_Type { Value, Operator, Lpar, Rpar, End, Error };
```

Value representa un operando, Operator representa uno de los operadores +, -, *, o /, Lpar y Rpar representan los paréntesis izquierdo y derecho respectivamente, End representa el fin de la cadena y, finalmente, Error indica que se encontró un símbolo que no puede ser considerado operando ni operador.

Ahora podemos definir una rutina que lea la cadena y nos indique qué tipo de ficha se está leyendo:

108b *(rutina lectora de fichas 108b)≡*

```
Token_Type lexer(char *& str, size_t & len)
{
    (cuerpo de lexer 108c)
}
```

`lexer()` inspecciona secuencialmente la cadena de caracteres contenida en `str` a partir del símbolo `str + len`. Al final de la llamada, `str` apunta al primer símbolo de la ficha. El valor de `str` puede modificarse, pues puede haber espacios en blanco que no deben procesarse.

`lexer()` retorna el tipo de ficha que se ha detectado. El parámetro `len` se modifica para indicar la longitud en caracteres de la ficha actual encontrada.

Los valores de los parámetros se inician del siguiente modo:

108c *(cuerpo de lexer 108c)≡*

(108b) 108e▷

```
str += len;
len = 1;
(ignorar espacios en blanco 108d)
```

Por definición, la inspección comienza en `str + len`. Al inicio, la longitud de la ficha que se encuentre será al menos de una unidad.

Iniciada la inspección, la rutina debe ignorar los espacios en blancos. Esto se define fácilmente como:

108d *(ignorar espacios en blanco 108d)≡*

(108c)

```
while (isblank(*str))
    str++;
```

`isblank()` es una extensión GNU de la biblioteca estándar del lenguaje C y retorna un valor diferente de cero si el símbolo de parámetro corresponde a un blanco o tabulador.

Por simplicidad, nuestro evaluador sólo acepta los operadores +, -, * y /, los cuales representan las operaciones aritméticas clásicas. De la misma manera, los operandos sólo son números enteros. Salvo un operando, la longitud de una ficha es uno. Verificamos, pues, si el símbolo encontrado corresponde a un operador:

108e *(cuerpo de lexer 108c)+≡*

(108b) ▷ 108c 109a▷

```
switch (*str)
{
    case '(': return Lpar;
    case ')': return Rpar;
    case '+':
    case '-':
    case '*':
```

```

    case '/': return Operator;
    case '\0': return End;
}

```

Si el flujo sale del switch, entonces la única posibilidad correcta es encontrar un operando. Nos cercioramos entonces de que la cadena corresponda a un operando:

109a *<cuerpo de lexer 108c>* +≡ (108b) ◁108e 109b ▷
if (not isdigit(*str))
 return Error;

`isdigit()` es una función de la biblioteca estándar del lenguaje C y retorna un valor diferente de cero si el símbolo de parámetro corresponde a un dígito.

Si el flujo no retorna error, entonces contabilizamos la cantidad de dígitos por la derecha y retornamos el código de ficha Value:

109b *<cuerpo de lexer 108c>* +≡ (108b) ◁109a
char* base = str + 1;
while (isdigit(*base++))
 len++;
return Value;

Cuando `lexer()` retorna un operador, debemos tomar acciones según la precedencia del operador. Para ello elaboramos una función que, dado un operador, determine su precedencia:

109c *<función de precedencia 109c>* ≡
unsigned precedence(const char & op) // \$ < (< +- < */
{
 switch (op)
 {
 case '\$': return 0;
 case '(': return 1;
 case '+':
 case '-': return 2;
 case '/':
 case '*': return 3;
 }
}

`precedence()` tiene como parámetro un símbolo. Puesto que los operadores son de un solo símbolo, podemos identificar la ocurrencia del operador mediante inspección directa de la cadena.

Hay un operador ficticio, \$, con la menor precedencia, cuyo rol se explicará más adelante. El paréntesis izquierdo se considera un operador en el siguiente nivel de precedencia. La suma y resta ocupan el próximo nivel de precedencia. Finalmente, los operadores con más precedencia son el producto y la división.

Si `lexer()` retorna Value, entonces necesitamos obtener una cadena de caracteres que contenga el número. Tal función la efectuamos del siguiente modo:

109d *<función convertir ficha a cadena 109d>* ≡
char * str_to_token(char * token_str, const size_t & len)
{
 static char buffer[256];
 strncpy(buffer, token_str, len);
 buffer[len] = '\0';

```

        return buffer;
    }

str_to_token() extrae una copia del número contenido en token_str.
```

El algoritmo de evaluación usará dos pilas como sigue:

110a *(pilas de evaluación 110a)≡* (111a)

```

ArrayStack<int> val_stack;
ArrayStack<char> op_stack;
```

Uses ArrayStack 101a.

El algoritmo efectúa operaciones en línea conforme analiza la cadena de entrada. La operación fundamental es sacar dos operandos de la pila val_stack y un operador de la pila op_stack; luego, efectuar la operación y, finalmente, almacenar el resultado de nuevo en la pila val_stack. Este patrón se modulariza en la siguiente rutina:

110b *(función de ejecución de operación en pilas 110b)≡*

```

void apply(ArrayStack<int>& val_stack, ArrayStack<char>& op_stack)
{
    const char the_operator = op_stack.pop();
    const int right_operand = val_stack.pop();
    const int left_operand = val_stack.pop();
    int result;
    switch (the_operator)
    {
        case '+': result = left_operand + right_operand; break;
        case '-': result = left_operand - right_operand; break;
        case '*': result = left_operand * right_operand; break;
        case '/': result = left_operand / right_operand; break;
    }
    val_stack.push(result);
}
```

Uses ArrayStack 101a.

apply() asume que el operando derecho fue insertado en la pila después del izquierdo. Esto es lo normal si la cadena de entrada se analiza de izquierda a derecha.

Ahora disponemos de las herramientas necesarias para desarrollar una rutina que evalúe una expresión infija. Nuestra rutina se llamará eval() y tendrá la siguiente estructura:

110c *(función de evaluación de expresión infija 110c)≡*

```

int eval(char* input)
{
    <variables de eval 111a>
    <inicialización de eval 111b>
    while (true)
    {
        <leer próxima ficha 111c>
        switch (current_token)
        {
            case Value:
                <procesar operando 111d>
            case Lpar:
                <procesar paréntesis izquierdo 112b>
            case Operator:
                <procesar operador 112a>
```

```

        case Rpar:
            ⟨procesar paréntesis derecho 112c⟩
        case End:
            ⟨procesar fin de entrada 112d⟩
        }
    }
}

```

`input` es la cadena de caracteres que contiene la expresión infija.

`eval()` requiere siguientes variables:

111a $\langle\text{variables de eval 111a}\rangle \equiv$ (110c)
 $\langle\text{pilas de evaluación 110a}\rangle$
`Token_Type current_token;`
`size_t token_len = 0;`

`current_token` es la ficha actual observada en la cadena de entrada y `token_len` es la longitud de la ficha en caracteres. Inicialmente no hay ninguna ficha, por lo que `token_len` debe ser cero.

La pila de operadores requiere un centinela especial que sea de la menor precedencia posible. Tal centinela es el valor '\$' que se inserta inicialmente en `op_stack`:

111b $\langle\text{inicialización de eval 111b}\rangle \equiv$ (110c)
`op_stack.push('$');`

Después de la inicialización, la rutina comienza a iterar. El cuerpo iterativo lee las fichas presentes en la entrada y luego procesa la ficha según su tipo. Leer la ficha consiste simplemente en una invocación a `lexer()`:

111c $\langle\text{leer próxima ficha 111c}\rangle \equiv$ (110c)
`current_token = lexer(input, token_len);`

Cuando la ficha es un operando, lo introducimos en la pila de operandos:

111d $\langle\text{procesar operando 111d}\rangle \equiv$ (110c)
`{`
 `const int operand = atoi(str_to_token(input, token_len));`
 `val_stack.push(operand);`
 `break;`
`}`

Previamente hay que efectuar la conversión de la cadena de caracteres a su representación numérica.

Podemos introducir el operador en la pila, pero antes podemos ejecutar todas las operaciones que están en la pila de operadores con mayor o igual precedencia que el operador actual. Esta alternativa es preferible porque disminuye el consumo en espacio de las pilas.

Para ejemplificar, consideremos la expresión $100 - 20 * 5 + 7$ y supongamos que la ficha actual es el operador de resta. En este momento, `val_stack` contiene $< 100 >$ y `op_stack` $< \$ >$ y no podemos efectuar ninguna operación porque el operador ficticio \$ tiene menor precedencia que el operador -. Cuando encontramos el operador *, tampoco podemos efectuar ninguna operación porque el producto tiene mayor precedencia que la resta. Finalmente, cuando nos encontramos el operador +, `val_stack` contiene $< 100, 20, 5 >$ y `op_stack` $< \$, -, * >$. Aquí efectuamos el producto, pues éste tiene mayor precedencia que la suma, lo que nos deja las pilas en los estados $< 100, 100 >$ y $< \$, - >$ respectivamente. Posteriormente, efectuamos la resta, pues ésta tiene la misma precedencia que la suma. Al

termino de esta operación introducimos el operador + en la pila lo que nos deja el estado de las pilas en < 0 > y < \$, + >.

La conducta anterior se modeliza entonces bajo la siguiente forma:

112a $\langle \text{procesar operador } 112a \rangle \equiv$ (110c)

```
{
    while (precedence(op_stack.top()) >= precedence(*input))
        apply(val_stack, op_stack);
    op_stack.push(*input); // introducir operador en op_stack
    break;
}
```

Los paréntesis en la expresiones infijas son utilizados para modificar la precedencia por omisión de los operadores. Cuando la ficha sea un paréntesis izquierdo, lo introducimos en la pila de operadores:

112b $\langle \text{procesar paréntesis izquierdo } 112b \rangle \equiv$ (110c)

```
{
    op_stack.push(*input); // introducir parentesis en op_stack
    break;
}
```

El paréntesis derecho indicará el final de una expresión que debe ser evaluada. Cuando encontramos el paréntesis derecho, ejecutaremos todas las operaciones entre los paréntesis. Para ello llamamos sucesivamente apply() hasta que el paréntesis izquierdo sea encontrado en el tope de la pila:

112c $\langle \text{procesar paréntesis derecho } 112c \rangle \equiv$ (110c)

```
{
    while (op_stack.top() != '(')
        apply(val_stack, op_stack);
    op_stack.pop(); /* saca el parentesis izquierdo */
    break;
}
```

El fin de la entrada lo indica lexer() cuando retorna el valor End. En este momento, si la expresión fue correcta, nos falta evaluar las operaciones que se encuentran en la pila de operandos, lo cual se define de la siguiente manera:

112d $\langle \text{procesar fin de entrada } 112d \rangle \equiv$ (110c)

```
{
    while (op_stack.top() != '$')
        apply(val_stack, op_stack);
    op_stack.pop(); // debe ser '$'
    const int ret_val = val_stack.pop();
    return ret_val;
}
```

2.5.6 Pilas, llamadas a procedimientos y recursión

Uno de los usos más trascendentales de la pila es la gestión de llamadas a procedimientos de un programa. Consideremos el siguiente fragmento de código:

```
int fct_1(int i) { /* ... */ }
```

```
int fct_2(int i)
{
```

```
int x;
x = fct_1(i);
...
}
int fct_3(int j)
{
    int x;
    x = fct_2(j);
    ...
}
void main()
{
    int y;
    y = fct_3(3);
    ...
}
```

Ahora examinemos los eventos involucrados a la llamada a `fct_3()` desde `main()`.

Cuando se alcanza la llamada a `fct_3()`, el flujo de control salta hacia la dirección de memoria de `fct_3()` y comienza a ejecutarla. Posteriormente, cuando se alcanza la llamada a `fct_2()`, y el flujo de control salta de nuevo hacia la dirección de `fct_2()`. Finalmente, se alcanza la función `fct_1()` y el flujo salta de nuevo hacia la dirección de `fct_1()`.

Cuando `fct_1()` finaliza, el flujo de programa recorre el camino inverso hasta `main()`. Planteemos, entonces, las siguientes preguntas:

- Las funciones `fct_2()` y `fct_3()` tienen parámetros de nombre `i` y variables locales de nombre `x`. ¿Cómo se diferencian entre si los parámetros y variables?
- ¿Cómo se gestiona la memoria para los parámetros y variables locales de un procedimiento?
- ¿Cómo hace el flujo de control para saltar a una función y regresar cuando culmina la llamada a la función?

Estos problemas son resueltos por el compilador, el sistema operativo y el hardware a través de una pila. La mayoría de los procesadores manejan dos registros especiales comúnmente denominados SB y SP. Estos registros manejan la denominada “pila del sistema”. SB es la dirección base de una pila y SP es la dirección actual del tope de la pila. Cuando se hace un push sobre la pila del sistema, SP es decrementado. Del mismo modo, cuando se hace un pop(), SP es incrementado.

Cuando el compilador procesa una llamada a una función, por ejemplo, `fct_1()` desde `main()`, se genera, antes de llamar a la función, la secuencia de pseudocódigo sobre la pila sistema mostrada en el cuadro 2.5.1.

El compilador también genera código adicional al principio y al final de la función `fct_1()` bajo el esquema mostrado en el cuadro 2.5.2.

Generalmente, un push() equivale a restar al registro SP el tamaño de lo que se le inserta. Simétricamente, un pop() equivale a una suma. Para las funciones `fct_2()` y `fct_3()`, el compilador genera códigos similares a 2.5.1 y 2.5.2.

1. Un push() para el espacio del resultado que retorna fct_1().
2. Varios pushes para los parámetros de fct_1().
3. Un push() para la dirección actual del flujo del programa.
4. Una instrucción de salto hacia la dirección de la función fct_1(), el cual lleva el flujo de control hacia el inicio de la función fct_1().
5. Una vez que fct_1() haya sido ejecutada, el flujo de programa regresará al punto posterior donde se efectuó el salto. Para ello, el compilador genera un pop() que recupera la dirección de retorno de la pila. Al regresar a main() se efectúan varios pops correspondientes a la cantidad de pushes que se hizo para pasar los parámetros.
6. Finalmente, se hace un último pop() que recupera el resultado de fct_1().

Pseudocódigo 2.5.1: Estructura de código generada para la llamada a funcion_1()

1. Al inicio de la función se efectúan varios pushes proporcionales al número de variables locales de fct_1().
2. El código de la función se genera con referencias a las variables locales apartadas en el punto anterior y los parámetros apartados y copiados en 2.5.1.
3. El resultado de la función también se guarda en el espacio apartado en el punto 1 de 2.5.1.
4. Al final de la función, el compilador añade una cantidad de pops igual a la cantidad de pushes usada para las variables locales. Estos pops liberan la memoria usada por las variables locales.
5. Se hace un pop() que recupera la dirección de retorno al invocante de fct_1() y se salta al punto 5 de 2.5.1.

Pseudocódigo 2.5.2: Estructura de código generada para la llamada a la funcion_1()

La información pertinente a una llamada a procedimiento que se guarda en la pila sistema es contigua. El bloque compuesto por el valor de retorno, parámetros, dirección de retorno y variables locales se denomina “registro de activación”.

El soporte ejecutivo para las llamadas a procedimientos se implanta con una pila porque el programa fluye a través de las funciones bajo una disciplina UEPS. Una vez que se alcanza fct_3(), el programa debe regresar por el camino inverso a main() pasando por fct_2() y fct_1().

Aunque la recursión es una técnica de programación bastante poderosa, es importante aprehender que conlleva costes importantes en espacio y tiempo. Como ejemplo, consideremos la función factorial implantada recursivamente:

114

```
(Factorial recursivo 114)≡
int fact(const int & n)
{
    if (n <= 1)
        return 1;
    return n*fact(n-1);
}
```

La versión recurrente del factorial es un mal ejemplo del uso ineficiente de la recursión, pues aparte de ser extremadamente ineficiente, la contraparte iterativa es muy eficiente

y más fácil de implantar. Sin embargo, a efectos didácticos, su codificación recurrente es muy fácil de comprender porque refleja idénticamente la definición matemática.

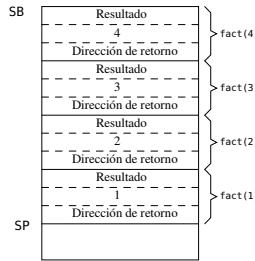


Figura 2.21: Capa de la pila sistema con la llamada a `fact(4)`

La figura 2.21 ilustra el estado de la pila sistema cuando se alcanza la llamada a `fact(1)`. Cada vez que se efectúa una llamada recursiva se gasta en espacio de pila el resultado de la llamada recursiva, el parámetro y la dirección de retorno. Del mismo modo, por cada llamada se gasta en tiempo los tres pushes y los tres pops.

2.5.6.1 Consejos para la recursión

Si estamos diseñando un algoritmo y nos es más fácil trabajar con la recursión, entonces deben tenerse en cuenta las siguientes consideraciones:

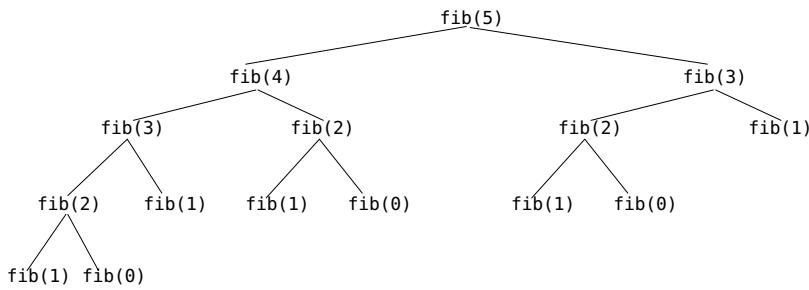
1. Todo algoritmo recursivo debe tener un caso base donde no ocurra ninguna llamada recursiva.
2. El algoritmo debe progresar y converger en tiempo finito a encontrar la solución. Por cada llamada recursiva, la entrada debe disminuir. Si este no es el caso, hay bastantes probabilidades de que el razonamiento subyacente al algoritmo esté errado.
3. Evite cálculos repetidos. Este es el principal peligro de la recursión. Un ejemplo notable son los números de Fibonacci cuya definición matemática es la siguiente:

$$\text{Fib}(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{si } n > 1 \end{cases}$$

La codificación recursiva que calcula el i -ésimo número de Fibonacci se desprende de su definición matemática:

```
115   ⟨Función de Fibonacci 115⟩≡
      int fib(const int & n)
      {
          if (n <= 1)
              return 1;
          return fib(n - 1) + fib(n - 2);
      }
```

Notemos que $\text{Fib}(n - 2)$ se calcula 2 veces. La primera cuando se llama explícitamente a $\text{Fib}(n - 2)$; la segunda, dentro de la llamada a $\text{Fib}(n - 1)$. Esta duplicidad de llamadas

Figura 2.22: Llamadas a fib() para $n = 5$

se expande exponencialmente a medida que aumenta n . La figura 2.22 ilustra la cantidad de llamadas a fib() para la pequeña escala $n = 5$.

Cuando se diseña un algoritmo recursivo debemos cerciorarnos de que no hayan cálculos redundantes. Por lo general, la redundancia requiere anotar los cálculos en alguna estructura de datos especial.

2.5.6.2 Eliminación de la recursión

Hay tres maneras conocidas para eliminar la recursión, las cuales mencionaremos en los subpárrafos subsiguientes.

Eliminación de la recursión cola

Si un procedimiento $P(x)$ tiene como última instrucción una llamada a $P(y)$, entonces es posible reemplazar $P(y)$ por una asignación $x = y$ y un salto al inicio del procedimiento. Por ejemplo, el recorrido infinito de los nodos de una lista enlazada podría definirse, recursiva y erróneamente, del siguiente modo:

116a

(Recorrido recursivo de lista enlazada 116a)≡

```

template <typename T>
void recorrer(Snode<T> * head, void (*visitar)(Snode<T>*))
{
    if (head->is_empty())
        return;
    (*visitar)(head->get_next());
    recorrer(head->get_next());
}
  
```

Uses Snode 68b.

Este procedimiento puede transformarse mediante eliminación de la recursión cola en:

116b

(Recorrido no-recursivo de lista enlazada 116b)≡

```

template <typename T>
void recorrer(Snode<T> * head, void (*visitar)(Snode<T>*))
{
    start:
    if (head->is_empty())
        return;
    (*visitar)(head->get_next());
    head = head->get_next();
    goto start;
}
  
```

```
}
```

Uses Snode 68b.

Muchas veces es posible eliminar la recursión cola aun si la última instrucción no es una llamada recursiva. Por ejemplo, la última instrucción de la versión recursiva del factorial (*Factorial recursivo 114*) no es la llamada a fact(n - 1), sino la llamada a return. En este caso podemos eliminar la recursión cola y obtener la siguiente versión:

117 *(Factorial no recursivo 117)≡*

```
int fact(const int & n)
{
    int result = 1;
    start:
    if (n <= 1)
        return result;
    result *= n;
    n--;
    goto start;
}
```

Emulación de los registros de activación

Podemos “suprimir” la recursión si emulamos las llamadas recursivas y los registros de activación. Para ello debemos definir una pila con los siguientes atributos:

1. Los valores de los parámetros. En caso de que se trate de una función, se incluye el valor de retorno.
2. Los valores de las variables locales.
3. Una indicación que señale la dirección de retorno. Cada valor de indicación debe corresponder a un punto de retorno en el procedimiento recursivo.

Una vez definida la pila, se modifica el procedimiento recursivo como sigue:

1. Se añade a la fase de inicio la declaración e inicialización de la pila.
2. Se ponen etiquetas de salto en los puntos de salida de las llamadas recursivas.
3. A la excepción de la fase de inicialización, el cuerpo de la función se envuelve con un lazo repetitivo cuya condición de parada es que la pila esté vacía.
4. Los puntos donde hay llamadas recursivas se sustituyen por un push() de lo que sería el registro de activación y un salto al inicio del lazo.
5. Los puntos donde el procedimiento recursivo retorne se sustituyen por un pop() seguido de un switch que determina, según la indicación de salto del registro de activación, la dirección de salto.

Por ejemplo, para el cálculo del i -ésimo número de Fibonacci modificamos la *(Función de Fibonacci 115)* del siguiente modo:

118a *(Versión recursiva mejorada de Fibonacci 118a)≡*

```
int fib(const int & n)
{
    if (n <= 1)
        return 1;
    const int f1 = fib(n - 1);
    const int f2 = fib(n - 2);
    return f1 + f2;
}
```

Esta forma facilita la eliminación de la recursión.

Definimos la siguiente estructura para el registro de activación:

118b *(Registro activación 118b)≡*

(118d)

```
# define P1 1
# define P2 2

struct Activation_Record
{
    int n;
    int f1;
    int result;
    char return_point;
};
```

Defines:

Activation_Record, used in chunk 118e.

Con miras a la claridad, el acceso a cualquiera de los campos se efectúa a través de alguno de los siguientes macros:

118c *(acceso registro activación 118c)≡*

(118d)

```
# define NUM(p)          ((p)->n)
# define F1(p)           ((p)->f1)
# define RESULT(p)       ((p)->result)
# define RETURN_POINT(p) ((p)->return_point)
```

La versión no recursiva, con pila, que calcula el i -ésimo número de Fibonacci, la definimos en el archivo *(fib_stack.C 118d)* cuya definición es la siguiente:

118d *(fib_stack.C 118d)≡*

```
<Registro activación 118b>
<acceso registro activación 118c>
int fib_st(const int & n)
{
    <cuerpo de fib_st 118e>
}
```

Ciñéndonos al método, lo primero es definir el preámbulo de la rutina:

118e *(cuerpo de fib_st 118e)≡*

(118d)

```
ArrayStack<Activation_Record> stack;
Activation_Record * caller_ar = &stack.pushn();
Activation_Record * current_ar = &stack.pushn();
NUM(current_ar) = n;
<emulación de recursión 119a>
```

Uses Activation_Record 118b and ArrayStack 101a.

A lo largo de la rutina manejaremos dos apuntadores. `caller_ar` representa el registro de activación del invocante. Requerimos este registro de activación porque debemos guardar el resultado de la llamada actual que emulamos, el cual se guarda en el registro de activación del invocante y no del invocado. `current_ar` es el registro de activación de la llamada actual.

Ahora emulamos el cuerpo del procedimiento recursivo. Para ello comenzamos por colocar el código mostrado en *⟨Versión recursiva mejorada de Fibonacci 118a⟩* y luego sustituimos por segmentos de código que emulan la recursión:

^{119a} *⟨emulación de recursión 119a⟩* ≡ (118e)

```

start:
  ⟨caso base de recursión 119c⟩
  ⟨llamada a fib(n - 1) 120a⟩
p1: /* punto de retorno de fib(n - 1) */
  ⟨retorno desde fib(n - 1) 120b⟩
  ⟨llamada a fib(n - 2) 120c⟩
p2: /* punto de retorno de fib(n - 2) */
  ⟨retorno desde fib(n - 2) 120d⟩
return_from_fib:
  ⟨retornar desde fib 120e⟩

```

Distinguimos cuatro etiquetas de salto del flujo de ejecución:

1. start es por donde se inicia el método. Cada vez que se emule una llamada recursiva se actualizan los punteros a los registros de activación y se salta hacia este punto.

Este segmento de código asume que un nuevo registro de activación ha sido previamente insertado en la pila. Por esa razón actualizamos los apuntadores a los registros de activación antes de saltar al inicio del programa.

La primera línea del programa recursivo procesa el caso base, el cual es muy similar en la versión no recursiva:

```

119c   ⟨caso base de recursión 119c⟩≡ (119a)
        if (NUM(current_ar) <= 1)
        {
            RESULT(caller_ar) = 1;
            goto return_from_fib;
        }

```

Al igual que en la versión recursiva, la condición se evalúa sobre el parámetro n del registro actual. Si caemos al caso base, entonces ponemos el resultado en el registro de activación del invocante y enviamos el flujo de programa hacia la parte que emula el retornar desde la función recursiva.

Si no se cae en el proceso base, entonces hay que emular la llamada afib($n - 1$), la cual se realiza como sigue:

120a *llamada a fib(n - 1) 120a* ≡ RETURN_POINT(current_ar) = P1;
 NUM(&stack.pushn()) = NUM(current_ar) - 1; // crea reg. act.
 llamada recursiva a fib 119b

La primera línea marca la dirección de retorno. La segunda aparta el registro de activación de la nueva llamada e inicia su parámetro n según la llamada `fib(n - 1)`.

2. Cuando se haya calculado `fib(n - 1)`, (*retornar desde fib 120e*) inspeccionará el valor `RETURN_VALUE` del registro actual de activación y se percatará de que hay que saltar el flujo hacia la etiqueta `p1`. En este momento debemos emular la asignación `int f1 = fib(n - 1);`

120b $\langle \text{returno desde fib}(n - 1) \rangle \equiv$ (119a)
 F1(current_ar) = RESULT(current_ar);

Luego, emulamos la llamada a `fib(n - 2)`:

120c $\langle \text{llamada a fib}(n - 2) \rangle_{120c} \equiv$ (119a)
 NUM(&stack.pushn()) = NUM(current_ar) - 2; // crea reg. act.
 RETURN_POINT(current_ar) = P2;
 $\langle \text{llamada recursiva a fib } 119b \rangle$

`llamada a fib(n - 2) 120c` es casi idéntica a `llamada a fib(n - 1) 120a`. Lo único que cambia es que el valor del parámetro n corresponde a $n - 2$.

3. Una vez calculado $\text{fib}(n - 2)$, (retornar desde fib 120e) saltará el flujo hacia p2. En este momento emulamos la asignación `int result = f1 + f2;`

$$120d \quad \langle \text{retorno desde fib}(n - 2) \rangle_{120d} \equiv \text{RESULT}(\text{caller_ar}) = F_1(\text{current_ar}) + \text{RESULT}(\text{current_ar}); \quad (119a)$$

4. Finalmente, `return_from_fib` es el punto donde se emula la instrucción `return` del procedimiento recursivo. Tal emulación consiste en:

```
120e  <retornar desde fib 120e>≡ (119a)
      stack.pop(); /* cae en el registro del invocante */
      if (stack.size() == 1)
          return RESULT(caller_ar);

      caller_ar = &stack.top(1);
      current_ar = &stack.top();

      switch (RETURN_POINT(current_ar))
      {
          case P1: goto p1;
          case P2: goto p2;
      }
```

Cada vez que se retorna de una función se elimina un registro de activación, ese es el cometido del `pop()`. Si la pila contiene un registro, entonces el flujo está terminando la llamada inicial y el resultado definitivo se encuentra en `RESULT(caller_ar)`. De lo contrario se actualizan los apuntadores a los registros de activación y se determina donde se debe retornar.

La rutina anterior exhibe un desempeño inferior a su versión recursiva. De entrada, el TAD `ArrayStack<T>` añade un sobrecoste de validación de desborde que no tiene la versión recursiva. Hay otras fuentes de sobrecoste, la indicación de salto es una que no tiene la versión recursiva.

Podemos efectuar mejoras sobre la versión no recursiva. En primer lugar podemos eliminar el campo `f1` del registro de activación y utilizar `result`. De este modo disminuimos la cantidad de pushes. Otra mejora es disminuir la cantidad de observaciones a la pila -instrucción `top()`-. La parte *(llamada recursiva a fib 119b)* podría definirse como:

```
caller_ar = current_ar;
++current_ar;
goto start;
```

Puesto que el registro del invocante pasará a ser el actual, no hay necesidad de observar la pila. Puesto que la pila está implantada con un arreglo, el siguiente registro de activación será el vecino de `current_ar`; por eso lo incrementamos.

Una optimización final consiste en colocar direcciones reales de salto en lugar de etiquetas. Este método eliminaría el `switch` y el `if` de *(retornar desde fib 120e)*, pues el flujo saltaría directamente a la dirección dada. Desafortunadamente, ni el lenguaje C ni el C⁺⁺ poseen mecanismos que permitan almacenar direcciones de salto¹⁷. Tendríamos, entonces que programar algunas partes de la rutina en ensamblador.

Eliminación total de la recursión

Como ya hemos visto, eliminar la recursión no es trivial. La experiencia indica que es muy difícil obtener una versión no recursiva que sea más eficiente que su equivalente recursivo. Por esa razón, la eliminación de la recursión de un algoritmo “naturalmente” recursivo sólo debe hacerse si la parte recursiva está dentro del camino crítico de ejecución y es vital aumentar el desempeño. En la mayoría de las ocasiones es preferible trabajar con la versión recursiva.

Existen situaciones en las que el tamaño de la pila del sistema es pequeño. En estos casos, la recursión es peligrosa, pues aumenta las posibilidades de desborde de pila. Situaciones de este tipo son programas empotrados, de tiempo real o aplicaciones paralelas con varios flujos (*threads*) de ejecución. En este caso, a fin de proteger la pila del sistema es justificable -algunas veces esencial- disponer de una versión no recursiva que maneje su propia pila.

En la práctica, el uso de la recursión es una cuestión de comodidad. Si los conceptos inherentes al algoritmo son recursivos, entonces es preferible trabajar en términos recursivos, pues hay más seguridad de entendimiento. Si por razones de desempeño o espacio se

¹⁷A la excepción del ensamblador, el autor no conoce ningún lenguaje que posea este mecanismo. Las primitivas `longjmp()` y `setjmp()` de la biblioteca estándar del lenguaje C permiten guardar direcciones de flujo y saltar a ellas. Lamentablemente, a nuestros efectos, sus costes en tiempo y en espacio son superiores que el guardar indicaciones.

hace necesario eliminar la recursión, entonces es preferible formular conceptos iterativos, en lugar de eliminar la recursión de un algoritmo recursivo. Un ejemplo notable es la misma función factorial cuya definición iterativa es:

$$n! = \prod_{i=1}^n i .$$

Esta definición nos conduce a una versión iterativa sin ningún razonamiento recursivo.

Algunas veces se puede encontrar una solución analítica. Por ejemplo, como demostraremos en § 6.4.2 (Pág. 483), el i -ésimo número de Fibonacci puede definirse como:

$$\text{Fib}(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right] .$$

Un algoritmo basado en esta expresión no requiere ninguna iteración ni recursión.

En definitiva, lo más simple es a menudo lo idóneo. Si pensar recursivamente es más simple, entonces transe por algoritmos recursivos. Si la versión recursiva es muy ineficiente, el factorial o los números de Fibonacci, por ejemplos, entonces busque soluciones que disminuyan los cálculos repetidos o utilice conceptos iterativos que le conduzcan hacia algoritmos iterativos.

2.6 Colas

Una cola es una secuencia con dos extremos de acceso. La inserción se efectúa por un extremo, la supresión por el otro. El extremo de inserción se denomina comúnmente “cola” o “trasero” o, en sus formas sajonas “rear” o “tail”. El extremo de supresión se llama “cabeza”, “frente” o, en sus formas sajonas, “front” o “head”.

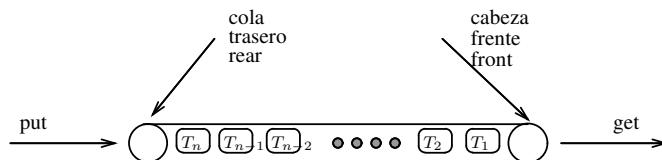


Figura 2.23: Visión abstracta de una cola

La figura 2.23 ilustra una visión abstracta de una cola. Los elementos entran por el extremo izquierdo mediante la operación `put()`. Del mismo modo, los elementos salen por el extremo derecho mediante la operación `get()`. Ambos extremos son observables.

El trasero de una cola denota el elemento más recientemente insertado y el frente el menos recientemente insertado. Esta noción de reciente es fundamental en algunos algoritmos y se denomina PEPS (Primero en Entrar, Primero en Salir) o, en inglés, FIFO (First In, First Out). Podemos decir que los elementos de la cola están ordenados desde el más joven hasta el más antiguo. Este patrón caracteriza a los contextos algorítmicos en los cuales una cola puede usarse.

Al igual que con las pilas, las colas son muy comunes en la vida real. Encontramos ejemplos típicos en las colas de espera de muchos servicios de nuestra sociedad. Cuando vamos al cine debemos hacer una cola para comprar la entrada; igualmente debemos hacer

otra cola para ingresar a la sala. Cuando vamos al banco debemos hacer cola. Cuando conducimos un carro y arribamos a un semáforo se forma una cola.

2.6.1 Variantes de las colas

Una variante más general del orden FIFO consiste en ver el orden desde el más recientemente accedido hasta el menos recientemente accedido. El orden de inserción y extracción es el mismo, pero hay una operación adicional de acceso o consulta. Cualquier elemento de la cola puede ser accedido, pero este acceso causa que el elemento se desplace hacia el trasero. De este modo, el frente es el elemento que tiene más tiempo sin referenciarse, mientras que el trasero es el que tiene menos tiempo. Esta política cobra importancia en problemas que manipulan conjuntos de datos de cardinalidad finita. Cuando se ingresa un nuevo elemento en un conjunto lleno, hay que seleccionar un elemento a suprimir. Si el acceso al conjunto exhibe localidad temporal, lo cual sucede en muchos contextos aplicativos, entonces el mejor elemento a suprimir es el menos recientemente utilizado. En una cola que maneje el orden de acceso, este elemento se encuentra en el frente.

Otra variante importante es permitir la inserción y supresión por los dos extremos. Esta variante es denominada “dicola” o “dipolo”¹⁸. En cierta medida, una dicola es una forma general de pila y de cola. Cualquiera de las dos estructuras puede modelizarse mediante una dicola. La familia en torno al TAD Dlink (§ 2.4.7 (Pág. 72)) es un dipolo.

2.6.2 Aplicaciones de las colas

En general, los sistemas programados utilizan colas para atender solicitudes bajo la justa disciplina FIFO. Por ejemplo, un servidor WEB puede encolar solicitudes de búsqueda de página según el orden de llegada. Muchos sistemas de control de transacciones encolan las solicitudes de servicios; por ejemplo, transacciones financieras.

El rol fundamental del software de red es transmitir paquetes de información entre los diversos nodos de una red. El programa encargado de esta transmisión se denomina “enrutador” o “encaminador”. Generalmente, un enrutador procesa sus paquetes según una disciplina FIFO.

Los sistemas operativos también utilizan colas para compartir recursos entre diversos usuarios. Tradicionalmente, procesos que arriben a un sistema se insertan en una cola. El proceso en el frente se extrae y se ejecuta durante un tiempo llamado quantum. Cuando el quantum expira, el proceso se inserta de nuevo en la cola y el sistema operativo extrae el siguiente proceso de la cola.

Existe toda una rama de la matemática, conocida como teoría de colas, la cual modeliza entidades abstractas que se encolan para solicitar algún servicio. Aunque la matemática arroja resultados analíticos muy útiles, algunos modelos son tan complejos que es necesario “simularlos”. Tales sistemas de simulación usan intensivamente colas.

2.6.3 Representaciones en memoria de las colas

Al igual que la pila, una cola es una secuencia de elementos de algún tipo. Como tal, hay dos formas básicas de representar una cola: con un arreglo o con una lista.

¹⁸En inglés, dequeue.

La implantación de la cola con un arreglo es similar a la pila. La diferencia estriba en que debemos manejar los dos extremos, ergo debemos mantener dos contadores: uno para la cola, otro para la cabeza. La figura 2.24 ilustra una cola implantada mediante un arreglo.

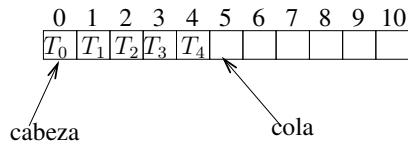


Figura 2.24: Cola implantada mediante un arreglo

Para insertar, copiamos el elemento e incrementamos la cola. Para eliminar, copiamos el valor de retorno e incrementamos la cabeza. La cola está llena cuando la cantidad de elementos es igual a la dimensión del vector.

Notemos que la eliminación de elementos puede dejar espacios disponibles a la izquierda del índice cabeza. Así pues, la inserción y supresión deben considerar esta circularidad. Esto puede manejarse mediante un *if* que verifique si el índice no se encuentra en el borde del arreglo, o mediante la función módulo.

Para las colas es preferible que la lista sea circular, pues podemos acceder al frente y el trasero mediante un solo apuntador. Si se trata de una lista simplemente enlazada, ordenamos los elementos desde el frente hasta el trasero y mantenemos un apuntador al último elemento (figura 2.25). De este modo podemos insertar y eliminar desde el puntero a la cola.

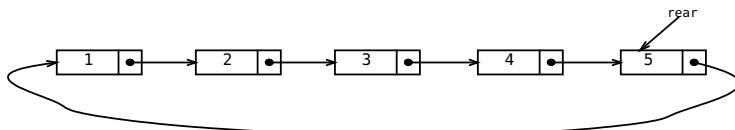


Figura 2.25: Cola implantada mediante una lista simple

Las listas doblemente enlazadas pueden ser una buena escogencia para las colas ordenadas según el tiempo de acceso y para las dicolas (§ 2.6.1 (Pág. 123)). La ventaja es que la supresión de un elemento es directa con un nodo doble, mientras que en una lista simple habría que mantener el puntero al predecesor.

El tipo *DynDlist<T>* estudiado en § 2.4.9 (Pág. 84) nos proporciona una manera directa de implementar una cola. Si vemos el último elemento como el trasero, entonces la operación *get()* se implanta mediante *return remove_first();* y *put(item)* mediante *append(item)*. El trasero y el frente se observan a través de *get_last()* y *get_first()*, respectivamente.

2.6.4 El TAD *ArrayQueue<T>* (cola vectorizada)

En esta sección desarrollaremos una implantación de colas mediante arreglos estáticos. Implementaremos dos TAD muy similares: *ArrayQueue<T>* y *FixedQueue<T>*. Ambos tipos son casi idénticos salvo que *FixedQueue<T>* no efectúa verificaciones de desborde.

Los tipos se definen en el archivo `<tpl_arrayQueue.H 125a>` de la siguiente manera:

```
125a <tpl_arrayQueue.H 125a>≡
      template <typename T> class ArrayQueue
      {
        <atributos de cola vectorizada 125b>
        <métodos privados de cola vectorizada 126d>
        <métodos públicos de ArrayQueue<T> 126c>
        <observadores de cola vectorizada 128g>
      };
      template <typename T> class FixedQueue
Defines:
  ArrayQueue, used in chunks 126c and 251c.
```

Ambas clases modelizan colas de elementos genéricos de tipo T implantadas mediante un arreglo estático. `ArrayQueue<T>` verifica desbordes cuando el arreglo está vacío o lleno. Un desborde provoca una excepción. `FixedQueue<T>` no efectúa ninguna verificación ni genera ninguna excepción. La ausencia de estas verificaciones conlleva una ligera ganancia en desempeño.

La dimensión del arreglo interno está restringida a una potencia exacta de 2.

```
125b <atributos de cola vectorizada 125b>≡ (125a) 125c▷
      const size_t two_pow;
      const size_t dim;
      T *           array;
```

El puntero `array` contiene los elementos de la cola. `dim` representa la dimensión del arreglo interno, el cual es una potencia exacta de 2 tal que `dim == 2two_pow`.

El frente de la cola está dado por:

```
125c <atributos de cola vectorizada 125b>+= (125a) ◁125b 125e▷
      size_t front_index; /* index of oldest inserted item */
```

`front_index` es el índice del elemento más antiguo en la cola; es decir, `array[front_index]` contiene el frente de la cola:

```
125d <frente de la cola 125d>≡ (127g)
      array[front_index]
```

El trasero de la cola está indicado por:

```
125e <atributos de cola vectorizada 125b>+= (125a) ◁125c 126a▷
      size_t rear_index;
```

`rear_index` indica la posición de la primera celda disponible para insertar; es decir, `array[rear_index - 1]` contiene el elemento más reciente en la cola.

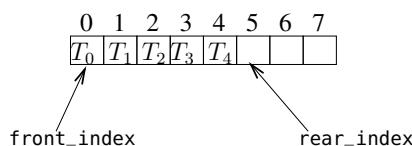


Figura 2.26: Organización del arreglo en `ArrayQueue<T>`

Los valores de `front_index` y `rear_index` se inician en cero. La figura 2.26 ilustra el estado del arreglo y de los índices después de insertar los primeros cinco elementos en un arreglo de 8 elementos.

Tanto para insertar, como para eliminar, basta con incrementar el índice correspondiente. Si queremos insertar, incrementamos `rear_index`; si queremos eliminar, incrementamos `front_index`.

Cuando alguno de los índices alcanza la dimensión del arreglo lo iniciamos de nuevo en cero; esto simula la circularidad. Para esto basta poner un `if` después del incremento que verifique si se alcanza la dimensión; si tal es el caso, entonces reiniciamos el índice en cero. Podemos evitar el `if` si cada vez que incrementamos un índice calculamos el módulo de la dimensión del arreglo. De este modo, el valor del índice siempre estará comprendido entre 0 y `dim - 1`. Este método linealiza el código, pero es más lento que con el `if`. Es para solventar este problema, que forzamos la dimensión a que sea potencia exacta de 2, pues, en este caso, el módulo puede calcularse muy rápidamente a través de un `and lógico`. Por tanto, uno de los atributos será una máscara de bits para calcular el módulo:

126a $\langle\text{atributos de cola vectorizada } 125b\rangle + \equiv$ (125a) $\triangleleft 125e \ 126b \triangleright$
`const size_t mask;`

Es tentador calcular el número de elementos mediante el valor absoluto de `rear_index - front_index`. Lamentablemente, tenemos una ambigüedad, pues la resta es cero cuando la cola está llena. No hay otra alternativa, entonces, que contabilizar directamente la cantidad de elementos en un atributo:

126b $\langle\text{atributos de cola vectorizada } 125b\rangle + \equiv$ (125a) $\triangleleft 126a$
`size_t num_items;`

Los atributos de la cola se inician como sigue:

126c $\langle\text{métodos públicos de } \text{ArrayQueue} < T > 126c\rangle + \equiv$ (125a) $\triangleleft 127b \triangleright$
`ArrayQueue(const size_t & tp = 8)`
`: two_pow(tp), dim(1<two_pow), array(NULL), front_index(0),`
`rear_index(0), mask(dim - 1), num_items(0)`
`{`
 `array = new T [dim];`
`}`
`Uses ArrayQueue 125a.`

La operación de incrementar un índice y luego calcular el módulo será efectuada por los modificadores de la cola. Es muy importante, pues, asegurarnos de hacerla correctamente. Para ello, la modularizamos y la aislamos en una sola operación:

126d $\langle\text{métodos privados de cola vectorizada } 126d\rangle + \equiv$ (125a) $\triangleleft 127a \triangleright$
`void increase_index(size_t & i, const size_t & inc = 1) const`
`{`
 `i += inc;`
 `i &= mask;`
`}`

`increase_index()` incrementa `i` en `inc` unidades y calcula el módulo mediante la máscara.

El método del módulo puede aplicarse también con restas en lugar de sumas. Cuando se decrementa un entero sin signo con valor cero, el desborde negativo causa que el entero adquiera el mayor valor. Puesto que la dimensión es potencia de dos, el módulo dará como resultado la dimensión menos uno. Debemos asegurarnos, empero, de que las restas se apliquen sobre enteros sin signo y, por supuesto, de que la aritmética sea binaria.

Si se desea consultar el trasero, entonces es necesario acceder a `array[rear_index - 1]`. Puesto que esta operación puede arrojar un desborde

negativo, la aislamos en una sola función que nos garantice una operación correcta:

127a *(métodos privados de cola vectorizada 126d)* +≡ (125a) ◁ 126d

```
T & rear_item(const size_t & i = 0)
{
    return array[static_cast<size_t> ((rear_index - i - 1) & mask)];
}
```

127b *(métodos públicos de ArrayQueue<T> 126c)* +≡ (125a) ◁ 126c 127d ▷

```
T & put(const T & item)
{
    (put en cola vectorizada 127c)
}
```

`put()` genera la excepción `std::overflow_error` si se intenta insertar en una cola llena. La inserción como tal es como sigue:

127c *(put en cola vectorizada 127c)* ≡ (127b)

```
array[rear_index] = item;
T & ret_val = array[rear_index];
increase_index(rear_index);
num_items++;
return ret_val;
```

En lugar de insertar un elemento es posible apartar espacio, bajo disciplina FIFO, para `n` elementos. Esto es tan rápido a través de un arreglo que vale la pena exportar la funcionalidad en el siguiente método:

127d *(métodos públicos de ArrayQueue<T> 126c)* +≡ (125a) ◁ 127b 127f ▷

```
T & putn(const size_t & n)
{
    (putn en cola vectorizada 127e)
}
```

La primera parte verifica el desborde y la segunda se remite a incrementar `rear_index`:

127e *(putn en cola vectorizada 127e)* ≡ (127d)

```
increase_index(rear_index, n);
num_items += n;
return rear_item();
```

La supresión es casi simétrica a `put()`:

127f *(métodos públicos de ArrayQueue<T> 126c)* +≡ (125a) ◁ 127d 128a ▷

```
T get()
{
    (get en cola vectorizada 127g)
}
```

donde la supresión consiste en:

127g *(get en cola vectorizada 127g)* ≡ (127f)

```
num_items--;
T ret_val = (frente de la cola 125d);
increase_index(front_index);
return ret_val;
```

La contraparte de `putn()` es `getn()`, es decir, liberar n elementos por el frente de la cola:

128a *(métodos públicos de `ArrayQueue<T>` 126c)* +≡ (125a) ◁127f 128c▷
`T & getn(const size_t & n) throw(std::exception, std::underflow_error)`
`{`
(getn en cola vectorizada 128b)
`}`

128b *(getn en cola vectorizada 128b)* ≡ (128a)
`num_items -= n;`
`increase_index(front_index, n);`
`return array[front_index];`

La consulta por el frente se hace mediante:

128c *(métodos públicos de `ArrayQueue<T>` 126c)* +≡ (125a) ◁128a 128e▷
`T & front(const size_t & i = 0)`
`{`
(retornar i-ésimo elemento del frente 128d)
`}`

128d *(retornar i-ésimo elemento del frente 128d)* ≡ (128c)
`return array[front_index + i];`

La consulta por el trasero también es simétrica:

128e *(métodos públicos de `ArrayQueue<T>` 126c)* +≡ (125a) ◁128c
`T & rear(const size_t & i = 0)`
`{`
(retornar i-ésimo elemento del trasero 128f)
`}`

128f *(retornar i-ésimo elemento del trasero 128f)* ≡ (128e)
`return rear_item(i);`

Finalmente tenemos los clásicos observadores:

128g *(observadores de cola vectorizada 128g)* ≡ (125a)
`const size_t & size() const { return num_items; }`

`bool is_empty() const { return num_items == 0; }`

`const size_t & capacity() const { return dim; }`

La implantación de `FixedQueue<T>` es muy similar a `ArrayQueue<T>`. La única diferencia será que `FixedQueue<T>` no verifica desbordes.

2.6.5 El TAD `ListQueue<T>` (cola con listas enlazadas)

El TAD `ListQueue<T>` modeliza una cola implantada mediante una lista simplemente enlazada circular. `ListQueue<T>` está especificado e implantado en el archivo *(tpl_listQueue.H 128h)*:

128h *(tpl_listQueue.H 128h)* ≡
`template <typename T> class ListQueue`
`{`
(Node de ListQueue<T> 129a)

```

⟨Atributos de ListQueue<T> 129b⟩
⟨Métodos públicos de ListQueue<T> 130a⟩
};

Defines:
ListQueue, used in chunks 129–31.

```

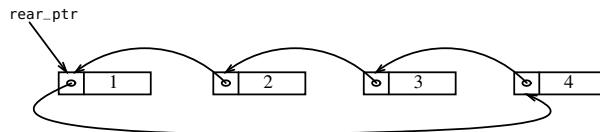


Figura 2.27: Cola implantada mediante Snode<T>

Las operaciones de ListQueue<T> se especifican en función de nodos simples de tipo ListQueue<T>::Node:

129a ⟨Node de ListQueue<T> 129a⟩≡ (128h)

```

typedef Snode<T> Node;
typedef ListQueue Set_Type;
typedef Node * Item_Type;

```

Uses ListQueue 128h and Snode 68b.

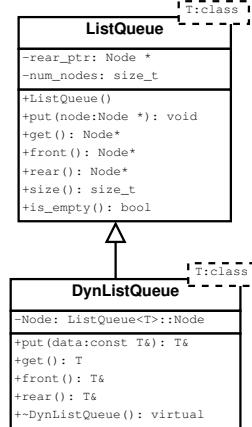


Figura 2.28: Diagrama UML de las clases vinculadas a ListQueue<T>

La lista está ordenada desde la cabeza hasta la cola. De este modo, manteniendo un puntero para la cola podemos eliminar en tiempo constante. La figura 2.27 muestra una disposición de cola implantada mediante lazos simples de tipo Snode<T>.

La configuración de la figura 2.27 es general sólo cuando la lista no está vacía. Es inevitable, pues, tener que manejar dos casos: uno general y uno cuando la cola esté vacía.

Ahora podemos comenzar a introducir los atributos de ListQueue<T>:

129b ⟨Atributos de ListQueue<T> 129b⟩≡ (128h) 129c▷

```

Node * rear_ptr;

```

rear_ptr será el puntero a la cola circular de elementos de tipo Snode<T>. El otro atributo es el número de elementos que posee la cola:

129c ⟨Atributos de ListQueue<T> 129b⟩+≡ (128h) ◁ 129b

```

size_t num_nodes;

```

Los valores anteriores requieren ser iniciados por el constructor:

130a *<Métodos públicos de ListQueue<T> 130a>+≡* (128h) 130b▷
 ListQueue() : rear_ptr(NULL), num_nodes(0) { /* empty */ }
 Uses ListQueue 128h.

Tenemos dos modos de saber si una cola está vacía: si `rear_ptr == NULL` o si `num_nodes == 0`.

La inserción posee la siguiente estructura:

130b *<Métodos públicos de ListQueue<T> 130a>+≡* (128h) ▷130a 130c▷
 void put(Node * node)
 {
 if (num_nodes > 0)
 rear_ptr->insert_next(node);
 num_nodes++;
 rear_ptr = node;
 }

`put()` no dispara ninguna excepción porque no hay manejo de memoria. Es imposible que `put()` fracase si el nodo fue correctamente apartado.

La eliminación es un poco más complicada porque debe verificar desborde negativo y que la cola devenga vacía:

130c *<Métodos públicos de ListQueue<T> 130a>+≡* (128h) ▷130b 130d▷
 Node * get()
 Node * ret_val = rear_ptr->remove_next();
 num_nodes-;
 if (num_nodes == 0)
 rear_ptr = NULL;
 return ret_val;
 }

La consulta se define de dos formas:

130d *<Métodos públicos de ListQueue<T> 130a>+≡* (128h) ▷130c
 Node * front() const
{
 return rear_ptr->get_next();
}
Node * rear() const
{
 return rear_ptr;
}

`front()` retorna el nodo por el frente y `rear()` el nodo por la cola.

2.6.6 El TAD DynListQueue<T> (cola dinámica con listas enlazadas)

El TAD `DynListQueue<T>` modeliza una cola implantada a través de una lista simplemente enlazada circular. `DynListQueue<T>` está especificado e implantado en el archivo `(tpl_dynListQueue.H 131)`.

`DynListQueue<T>` hereda parte de la interfaz e implantación de `ListQueue<T>`. El único rol de `DynListQueue<T>` es manejar la memoria y manipular datos de un tipo genérico `T` en lugar de nodos de una lista enlazada. Puesto que este proceso ya lo hemos

puesto en práctica para las listas y las pilas, pondremos en esta sección toda la implantación directa de `DynListQueue<T>`:

```
131 <tpl_dynListQueue.H 131>
      template <typename T> class DynListQueue : public ListQueue<T>
      {
          typedef typename ListQueue<T>::Node Node;
          T & put(const T & data)
          {
              Node * ptr = new Node (data);
              ListQueue<T>::put(ptr);
              return ptr->get_data();
          }
          T get()
          {
              Node * ptr = ListQueue<T>::get();
              T ret_val = ptr->get_data();
              delete ptr;
              return ret_val;
          }
          T & front() const
          {
              return ListQueue<T>::front()->get_data();
          }
          T & rear() const
          {
              return ListQueue<T>::rear()->get_data();
          }
          virtual ~DynListQueue()
          {
              while (not this->is_empty())
                  get();
          }
      };
      Uses ListQueue 128h.
```

2.7 Estructuras de datos combinadas - Multilistas

Consideremos una lista de “estudiantes” de un curso de estructura de datos. Los nombres, apellidos y cédula de identidad pueden representarse mediante cadenas de caracteres. Las notas pueden representarse mediante enteros. Toda esta información la estructuramos en un registro y usamos un arreglo de registros para representar la lista. Esta manera de organizar los datos, que posiblemente ya nos sea “natural”, es un ejemplo de combinaciones entre diversas estructuras de datos.

Según los requerimientos del problema, una secuencia de elementos puede representarse como un arreglo o como una lista. Comúnmente, una matriz se representa como un arreglo de arreglos. ¿Podemos hacer lo mismo con listas?, es decir, ¿podemos tener una lista de listas? o ¿arreglo de listas? o ¿una lista de arreglos? Las respuestas son afirmativas.

En lenguaje C, una matriz se representa mediante un arreglo de filas, en el cual cada fila es también un arreglo. Así pues, podríamos declarar lo siguiente:

```
DynSlist<int> mat[5];
```

Aquí tenemos un arreglo de listas donde cada `mat[i]` refiere a una lista enlazada.

También podríamos declarar:

```
typedef int Arreglo[5];
```

```
DynSlist<Arreglo> mat;
```

Lo que nos define una lista enlazada de arreglos de dimensión 5.

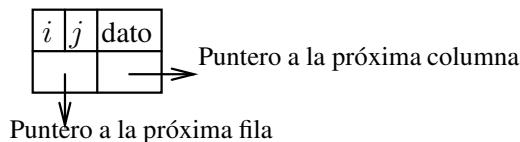
Igualmente podemos declarar:

```
DynSlist< DynSlist<int> > mat;
```

para definir una lista de listas.

Los ejemplos anteriores pueden modelizarse a partir de TAD conocidos. Es posible que tengamos que bajar de nivel y manejar directamente las listas. Como ejemplo, considere una aplicación que manipule matrices de muy alta escala, del orden de millones de filas y columnas. Tales aplicaciones son comunes en problemas de programación lineal y entera de la vida real. La escala de estas matrices agota la memoria de cualquier computador.

Está claro que no tenemos espacio para almacenar una matriz de billones de elementos bajo la representación clásica. Empero, aunque las matrices son enormes, la mayoría de las veces éstas poseen muchísimos elementos nulos. Podemos, pues, aplicar el mismo principio que en los polinomios: sólo guardamos los elementos diferentes de cero. Bajo este lineamiento, imaginamos el siguiente tipo especial de nodo:



que nos representa una entrada de una matriz `mat[i, j]` con los siguientes atributos:

- Índice de fila; es decir el valor de `i`
- Índice de columna, es decir, el valor de `j`
- Valor del dato diferente de cero contenido en `mat[i, j]`
- Puntero al siguiente `mat[i + x, j]` con valor diferente de cero
- Puntero al siguiente `mat[i, j + y]` con valor diferente de cero

De esta manera, la matriz:

$$\begin{pmatrix} 1 & -5 & 0 & 0 & 10 \\ 0 & 0 & 1 & -2 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

puede representarse como en la figura 2.29.

Como vemos en la figura 2.29, tenemos 5 listas de filas y 5 listas de columnas. Cada lista posee un nodo cabecera, el cual, para mejorar la rapidez de acceso, conviene ubicar en un arreglo. Así pues, tendríamos un arreglo de 5 cabeceras para las filas y otro de 5 cabeceras para las columnas.

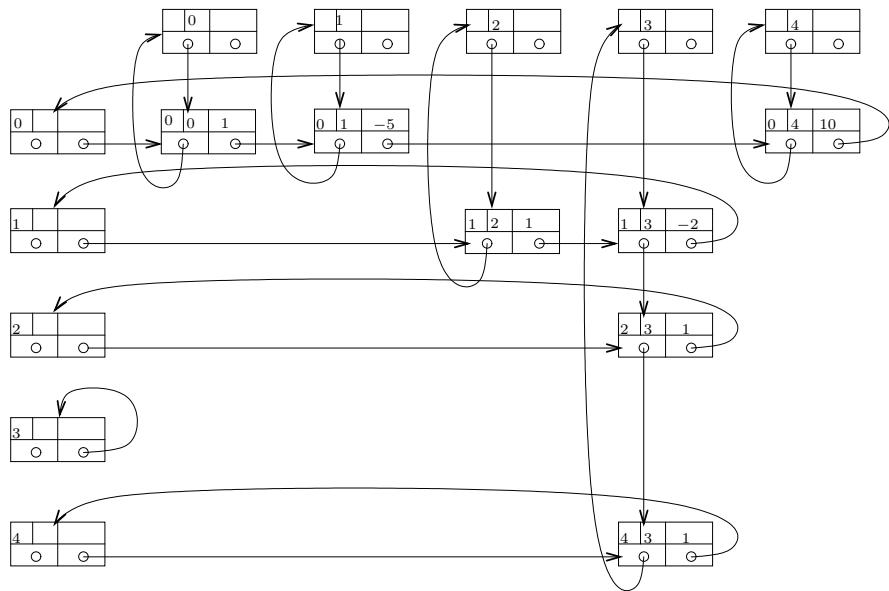


Figura 2.29: Una matriz esparcida representada mediante multilistas

Dado un nodo cualquiera, el elemento hacia la derecha indica la próxima entrada en la misma fila con un valor diferente de 0; la columna es averiguada por el índice de la columna. Del mismo modo, el elemento hacia abajo indica la próxima entrada en la misma columna con un valor diferente de cero; la fila es averiguada por el índice de la fila.

La representación de la figura 2.29 compensa con creces si el número de entradas nulas es muy grande. Por ejemplo, una matriz $10^4 \times 10^4$ ocuparía 400×10^6 de bytes en una arquitectura de 32 bits. Si un 4% de los elementos es diferente de cero, que representa 4 millones de elementos, la representación con multilistas ocupará $(4 + 4 + 4 + 4 + 4) \times 0,04 \times 100 \times 10^6 = 80 \times 10^6$ bytes.

La suma, resta y división de matrices son fáciles de programar con matrices basadas en la estructura de la figura 2.29. Para otras operaciones puede ser más conveniente tener lazos a la fila y a la columna predecesoras.

El algoritmo cándido para multiplicar matrices $n \times n$ es cúbico. Esto significa que su tiempo de ejecución es proporcional a n^3 . Con millones de columnas y filas no hay muchas esperanzas de obtener un buen desempeño si escogemos la representación secuencial. Sobre matrices esparcidas representadas con multilistas el tiempo es mucho menor, pues las entradas de las matrices son visitadas en el mismo orden de las listas y no hay necesidad de insertar ceros en la matriz resultado. En definitiva, la duración depende de la cantidad de elementos diferentes de cero.

2.8 Notas bibliográficas

Los orígenes de las estructuras básicas se remontan a mediados de los años 30 del siglo XX durante los desarrollos de programas para los primeros computadores. Esta época es tan arcana a este redactor que éste no se siente autorizado para ofrecer un discurso histórico sobre el origen de las estructuras de datos que representan secuencias ni algunas de las técnicas de búsqueda tratadas en este capítulo. Para observar un hermoso panorama de

esta apasionante historia, escrito por uno sus protagonistas, recomendamos nuevamente la lectura de la sección histórica contenida en el primer volumen de Knuth [97] (páginas 457-465).

Por su carácter secuencial “innato”, la noción de arreglo fue manejada desde los primeros programas de computador. En añadidura, el arreglo tiene su equivalente “vector” en matemática. Por lo anterior, sería muy aventurado atribuir alguna autoría exclusiva. Empero, cabe señalar el Fortran [12] como el primer lenguaje de programación y el primero en soportar arreglos a través del propio lenguaje.

El enfoque de diseño de la jerarquía de clases Dlink está primigeniamente inspirado en utilitarios diseñados para el micro-núcleo Chorus [151], sistema sobre el cual este redactor cursó sus estudios doctorales. Algunas bibliotecas de programación usan enfoques similares, entre los cuales cabe destacar `net.datastructures` reportada en [63].

En el mismo sentido de lo arreglos, las listas enlazadas fueron descubiertas independiente y separadamente en los primigenios y ya arcanos contextos computacionales. Sin embargo, cabe señalar al IPL (Information Processing Languaje) [132] como el primer lenguaje que incorporó las listas a un lenguaje de programación. La clásica pictorización consistente de rectángulos para representar bloques de memoria y de flechas para los punteros, aparece en un artículo de Newell y Shaw sobre programación con IPL [131].

Los arreglos, listas, pilas y colas ya son parte de los lenguajes de muy alto nivel modernos, entre los que cabe señalar Perl [177] y Python [172]. Éstos se circunscriben en los llamados de “scripting”, término madurado por John K. Ousterhout [136] en un célebre y visionario artículo mencionado en la sección bibliográfica de este capítulo.

La búsqueda binaria es un principio de búsqueda milenario, el cual, según Knuth [97], se remonta hasta la misma Mesopotamia, precursora de la civilización y hoy en día miserablemente aniquilada en nombre de la propia civilización. Consultese el trabajo histórico de Zohar Manna and Richard Waldinger sobre los orígenes de la búsqueda binaria [116].

Aparte sus anécdotas históricas [97] es una de las más sólidas referencias formales en el campo de algoritmos y estructuras de datos. Las estructuras lineales son tratadas de manera muy formal y con aplicaciones reales. La primera edición fue redactada en una época en que el poder computacional de una máquina aún era muy bajo y el lenguaje ensamblador era la única alternativa seria para programar aplicaciones de alto desempeño. Por esta razón, el texto de Knuth contiene una amplia gama de “trucos” a utilizar con estructuras lineales.

El clásico y antiguo texto de Wirth [182] fue escrito durante los años 70 del siglo XX. A pesar de su edad, hoy en día este texto continua bastante vigente y presenta una exposición de las estructuras lineales superior a la norma. Además, Wirth consagra un excelente y completo capítulo a la recursión.

El libro de Sedgewick [155] está repleto de algoritmos con estructuras lineales; por ejemplo, algoritmos de ordenamiento de listas. A nivel práctico, ésta es, en nuestra opinión, la mejor referencia. Cabe mencionar empero que los algoritmos que usan estructuras lineales están distribuidos a lo largo del texto en capítulos que no necesariamente tratan de estructuras lineales.

Lewis y Denenberg [108] presentan algunos trucos interesantes con listas enlazadas.

La técnica de clase intermedia (Proxy), utilizada por `BitArray` y `DynArray<T>` para distinguir los accesos de lectura de los de escritura, la analiza intensivamente Scott Meyer [125].

El problema de José es analíticamente tratado por Graham, Knuth y Patashnik [64].

La aritmética de polinomios explicada en § 2.4.10 (Pág. 91) es ampliamente desarrollada por Aho, Hopcroft y Ullman[9]. Una alternativa puede encontrarse en el texto enciclopédico de Cormen, Leiserson y Rivest [32].

El algoritmo de evaluación de expresiones infijas desarrollado en § 2.5.5 (Pág. 106) fue tomado de un texto sobre programación de sistemas de Peter Calingaert [27]. Este libro está tan magistral y concisamente escrito que lo recomendamos con creces para una introducción al campo de la programación de sistemas, compiladores, ensambladores y cargadores.

2.9 Ejercicios

1. ¿Por qué la búsqueda binaria no debe utilizarse para elementos ordenados en un archivo almacenado en memoria secundaria?
2. Implante enteramente el problema fundamental de estructura de datos estudiado en § 1.3 (Pág. 17) mediante arreglos desordenados.
3. Implante enteramente el problema fundamental de estructura de datos estudiado en § 1.3 (Pág. 17) mediante arreglos ordenados.
4. Explique cómo es el mecanismo de acceso directo a los elementos de una matriz.
5. En general, explique cómo es el mecanismo de acceso directo a los elementos de un arreglo n -dimensional.
6. Diseñe e implante un TAD que modelice arreglos n dimensionales.
7. Escriba una implantación lo más eficiente posible de la rutina `DynArray<T>copy()` según las indicaciones dadas en § 2.1.4.3 (Pág. 44).
8. Escriba un algoritmo eficiente que efectúe la inserción ordenada en un `DynArray<T>`.
9. Escriba un algoritmo eficiente que efectúe la eliminación ordenada en un `DynArray<T>`.
10. Suponga un proceso con una memoria total disponible de M bytes y una máximo permitido por una llamada a `new` de S bytes. Considere un arreglo dinámico de elementos de tamaño T . Seleccione los tamaños de directorio, segmento y bloque que maximicen la dimensión de un `DynArray<T>`.
11. Escriba la sobrecarga del método `DynArray<T>::cut(l, r)`, el cual libera toda la memoria entre $[0..l - 1]$ y $r+1..dim$ (dim) es la dimensión actual del arreglo.
12. La biblioteca *ALÉPH* implanta un prototipo del TAD de la biblioteca `stdc++` llamado `vector<T>`, el cual exporta un arreglo dinámico. El tipo en cuestión se encuentra en el archivo `Vector.H`.
 - (a) Revise minuciosamente la implantación en `Vector.H`. Pruebelo, busque errores, eventualmente depúrela y eventualmente mejore su eficiencia.
 - (b) Compare críticamente `Vector.H` con otras implantaciones de `stdc++`.

13. Escriba un algoritmo eficiente que invierta un arreglo. Por ejemplo, al invertir ABCDEFG se debe arrojar GFEDCBA.
14. Escriba un algoritmo que elimine los elementos duplicados de un arreglo ordenado.
15. Escriba un algoritmo que elimine los elementos duplicados de un arreglo desordenado.
16. Dado un arreglo desordenado de $n - 1$ elementos enteros comprendidos entre 1 y n , diseñe un algoritmo que no contenga ciclos anidados, o sea, de una sola pasada, que determine el elemento faltante (+).
17. Considere la operación de rotación de un arreglo. Por ejemplo, rotar 3 veces hacia la izquierda el arreglo ABCDEFGHI arroja como resultado DEFGHIABC. Diseñe un algoritmo general que no contenga ciclos anidados (sólo ciclos de una sola pasada), que rote m veces un arreglo de dimensión n (++).
18. Suponga una secuencia $S = \langle s_1, s_2, \dots, s_n \rangle$. La operación void `rotar_derecha(S, n)` "rota" a S n posiciones hacia la derecha. Por ejemplo,

$$\text{rotar_derecha}(\langle 1, 2, 3, 4, 5, 6, 7 \rangle, 3) = \langle 5, 6, 7, 1, 2, 3, 4 \rangle$$

Asumiendo secuencias implantadas con listas circulares doblemente enlazadas que utilizan nodo cabecera, escriba un algoritmo que implante `rotar_derecha(S, n)` en tiempo proporcional al tamaño del arreglo y sin usar espacio que crezca en función del arreglo. En otras palabras, no debe usar listas o arreglos.

19. Dada una cadena de caracteres que contiene palabras, suponga una operación de justificación de la cadena de caracteres a un máximo de n . Por justificar se entiende que se coloquen todas las palabras posibles en exactamente n caracteres intercalando proporcionalmente blancos. Por ejemplo, la cadena:

Esta es una prueba de justificación

tiene una longitud de 32 caracteres. Si se justifica a 30 caracteres, entonces el resultado es:

Esta es una prueba de

Diseñe un algoritmo que justifique una cadena de caracteres a un máximo n menor que la longitud de la cadena. El algoritmo debe tener dos salidas: la cadena justificada y la cadena extraída por la derecha.

20. Considere n elementos enteros, almacenados en un arreglo, comprendidos entre 0 y m , $m \gg n$.
 - (a) Use el tipo `BitArray` para diseñar un algoritmo que determine algún elemento faltante.
 - (b) Suponga que m es suficientemente pequeño para que todos los números quepan en memoria. Diseñe un algoritmo que construya un arreglo que contenga los elementos faltantes.

- (c) Suponga que n es muy grande, pero que el arreglo aún cabe en memoria. Suponga condiciones muy restrictivas de memoria: no es posible usar un arreglo de bits ni, en general, cualquier otra estructura de dato. Empero sí es posible utilizar archivos en modo append, es decir, sólo se puede insertar al final del archivo.
- i. Use el principio de búsqueda binaria para diseñar un algoritmo que determine algún elemento faltante (+++).
 - ii. Use el principio de búsqueda binaria para diseñar un algoritmo que construya un archivo con los números faltantes (+ -luego de resolver anterior-) .
21. Considere la selección aleatoria de un elemento contenido en un arreglo de n elementos enteros.
- (a) Diseñe un algoritmo que escoja aleatoriamente un elemento del arreglo.
 - (b) Suponga ahora que no se conoce n y que el final del arreglo corresponde a un elemento centinela con valor genérico FIN. Diseñe un algoritmo eficiente que seleccione aleatoriamente un elemento del arreglo.
 - (c) Ahora asuma que los enteros están almacenados en una lista simplemente enlazada. Diseñe un algoritmo que efectúe una sola pasada y seleccione aleatoriamente un elemento de la lista (++).
22. Considere el siguiente prototipo de función:
- ```
template <typename T> int select(T a[], size_t n, int i);
```
- El cual retorna el índice de  $i$ -ésimo menor elemento en el arreglo desordenado  $a[]$  que contiene  $n$  elementos. Escriba una implantación que no modifique la permutación del arreglo ni utilice arreglos auxiliares (+).
23. Extienda el TAD DynArray< $T$ > para manejar matrices. Suponiendo que el TAD es llamado DynMatrix< $T$ >:
- (a) ¿Puede implantarse este TAD a partir del TAD DynArray< $T$ >?
  - (b) ¿Puede usarse una implantación independiente más eficiente?
  - (c) ¿Cómo sería la forma y cuáles las funciones de la clase proxy sobre el TAD DynMatrix< $T$ >?
24. Extienda el TAD DynArray< $T$ > para manejar arreglos multidimensionales.
25. La mayoría de los TAD que manejan listas no utilizan memoria. Ninguno de estos TAD verifica si la dirección de un objeto que inserta o elimina es válida. Diseñe un método que verifique si una dirección de objeto es válida (+).
26. En el mismo espíritu de la pregunta anterior, diseñe un método que, además de verificar si el puntero es válido, verifique que la dirección corresponda a un objeto del tipo esperado por la función (++).
27. Especifique cuáles ayudas podría ofrecer un compilador para facilitar la implantación genéricamente de la conversión de un Slink al registro que lo incluya (+++).

28. Especifique cuáles ayudas podría ofrecer el lenguaje de programación para implantar genéricamente la conversión de un Slink al registro que lo incluya (+++).
29. Escriba un algoritmo que cuente el número de nodos en una lista simplemente enlazada.
30. Estudie y discuta el diseño de los TAD Slist<T> y DynSlist<T>. ¿De cuáles formas pueden extenderse o mejorarse?, ¿cuáles son sus problemas?, ¿están completos?
31. Implante, en detrimento del tiempo de ejecución, un TAD llamado Single\_Link que tenga exactamente los mismos métodos que la clase Dlink.
32. Escriba un algoritmo que cuente el número de ocurrencias de un elemento  $x$  en una lista simplemente enlazada.
33. Dado un apuntador  $\text{ptr}$  a un nodo de una lista enlazada, escriba un algoritmo que determine la posición ordinal de  $\text{ptr}$  dentro de la lista. Diseñe su algoritmo para que funcione con listas simples y dobles.
34. Discuta brevemente alternativas para implantar la substracción de polinomios.
35. Discuta brevemente cómo implantar los algoritmos de división y residuo de polinomios. Derive expresiones del tiempo de ejecución.
36. Implante el TAD Polinomio, explicado el § 2.4.10 (Pág. 91), mediante listas simplemente enlazadas. Use el TAD Snode<T> o Slink.
37. Considere dos listas de enteros doblemente enlazadas, circulares, con nodo cabecera, denominadas  $l_1$  y  $l_2$  respectivamente.  
Considere la operación:

```
const bool similar(Dnode<int> & l1, Dnode<int> & l2)
```

La cual retorna true si las listas contienen exactamente los mismos enteros, false en caso contrario.

Escriba algoritmos que implanten la operación para las siguientes situaciones:

- (a) Las dos listas están ordenadas del menor al mayor entero
- (b) Las listas no están ordenadas

38. Diseñe, instrumente e incorpore al TAD Slink las operaciones del TAD Dlink `insert_list()`, `append_list()` y `concat_list (+)`.
39. Considere el siguiente prototipo de función:

```
void permutar_pares(Dlink * lista);
```

La función toma cada par dentro de la secuencia e los intercambia. Por ejemplo, si  $l = <1, 2, 3, 4, 5, 6, 7, 8, 9>$ , entonces, luego de `permutar_pares(l)`,  $l = <2, 1, 4, 3, 6, 5, 8, 7, 9>$ .

Escriba la función en cuestión.

40. Considere el siguiente prototipo de función:

```
void conmuta_pares(Dlink * l1, Dlink * l2);
```

La función intercambia los elementos de posición par de  $l1$  con los de posición par de  $l2$ . Por ejemplo, si  $l1 = \langle 1, 2, 3, 4, 5, 6, 7 \rangle$  y  $l2 = \langle a, b, c, d, e \rangle$ , entonces, luego de `conmuta_pares(l1, l2)` las listas devienen en  $l1 = \langle a, 2, c, 4, e, 7 \rangle$  y  $l2 = \langle 1, b, 3, d, 5 \rangle$ .

Escriba la función en cuestión.

41. Escriba un algoritmo iterativo que calcule el  $i$ -ésimo número de Fibonacci.
42. Escriba un algoritmo recursivo que compute el  $i$ -ésimo número de Fibonacci y que no redunde cálculos (+).
43. Escriba una versión del algoritmo con pila desarrollado en 2.5.6.2 para calcular el  $i$ -ésimo número de Fibonacci que no redunde en cálculos (+).
44. Escriba un algoritmo recursivo que convierta una cadena de caracteres que contiene un entero a su representación entera. En otras palabras, escriba una rutina que convierta un entero almacenado en un `char*` a un entero de tipo `int`.
45. En un tablero de ajedrez de  $n \times n$  escaques<sup>19</sup>, se sitúa un caballo en las coordenadas  $x_0, y_0$ . Escriba un algoritmo que encuentre, si existe, un recubrimiento completo del tablero de  $n^2 - 1$  movimientos tal que cada escaque sea visitado exactamente una vez (++).
46. Dado un tablero de ajedrez de  $8 \times 8$  escaques escriba un algoritmo que distribuya ocho reinas en el tablero de manera tal que ninguna reina pueda amenazar a cualquiera de las otras (++).
47. Resuelva el problema anterior de las ocho reinas para encontrar todas las soluciones diferentes (+).
48. **El problema del morral**

Dado un objetivo  $O$  y una colección de pesos  $P = \{p_1, p_2, \dots, p_n, p_i\} \in \mathcal{N}$ , se pide determinar si existe un selección de pesos  $P' \subset P$  tal que  $\sum_{\forall p_i \in P'} =$ .

Este problema es conocido como el problema del morral porque alegoriza el seleccionar cuáles cargas llevar en un morral de manera tal que no se porte más  $O$  kilogramos.

Para el problema del morral:

- (a) Escriba un algoritmo recursivo
- (b) Elimine la recursión del algoritmo anterior
- (c) Escriba un algoritmo iterativo que no utilice pila (++)
49. Dada una colección de objetos  $S = \{s_1, s_2, \dots, s_n, s_i\}$  con pesos  $P = \{p_1, p_2, \dots, p_n, p_i\} \in \mathcal{R}$  y valores  $V = \{v_1, v_2, \dots, v_n, v_i\} \in \mathcal{R}$ , se pide encontrar un subconjunto  $S' \subset S$ , que maximice el valor total  $\sum_{\forall s \in S'} v_i$  restringido a una capacidad de pesos  $\sum_{\forall s \in S} p_i \leq C$ , es decir, que todos los objetos quepan en un morral de capacidad  $C$ .

---

<sup>19</sup>En la jerga ajedrecística, un escaque es uno de los cuadros del tablero.

- Diseñe un algoritmo que resuelva esta variante del problema del morral (++).
50. Escriba un algoritmo recursivo que genere las  $n!$  permutaciones de  $n$  elementos  $x_1, x_2, \dots, x_n$  con un consumo de espacio constante, que no crezca según  $n$  (++).
51. La función de Ackerman  $A(m, n)$  está definida de la forma siguiente:
- $$A(m, n) = \begin{cases} n + 1 & m = 0 \\ A(m - 1, 1) & m > 0, n = 0 \\ A(m - 1, A(m, n - 1)) & m, n > 0 \end{cases}$$
- (a) Escriba una rutina recursiva que calcule la función de Ackerman. ¿Cuál es la complejidad de tiempo de la función? (+)
- (b) Escriba una función iterativa que calcule la función de Ackerman. ¿Cuál es la complejidad de tiempo de la función? (+)
52. Sean  $A$  y  $B$  dos listas enlazadas ordenadas. Considere la “mezcla” ordenada de  $A$  y  $B$ , cuyo resultado es una lista fusionada y ordenada de los nodos de  $A$  y  $B$ . Escriba un algoritmo que mezcle dos listas:
- (a) Simplemente enlazadas
- (b) Dblemente enlazadas
53. Implemente una biblioteca que efectúe todas las verificaciones de tipo en tiempo de ejecución tal como fue planteado en ejercicio 26 (++).
54. Implante el constructor copia y el operador de asignación para la clase  $\text{DynArray}\langle T \rangle$  definida en § 2.1.4 (Pág. 34) (+).
55. Especifique, diseñe e implante totalmente el TAD  $\text{DynMatrix}\langle T \rangle$  mencionado en el problema 23.
56. Diseñe e implante un TAD que maneje arreglos de bits y que utilice el TAD  $\text{DynArray}\langle T \rangle$ .
57. Diseñe e implante un TAD que maneje matrices de bits.
58. Implante un TAD llamado  $\text{DynArrayStack}\langle T \rangle$ , el cual maneja pilas implantadas con arreglos dinámicos. El arreglo debe ser contraído dinámicamente.
59. Implante un TAD llamado  $\text{DynArrayQueue}\langle T \rangle$ , el cual maneja colas implantadas con arreglos dinámicos. El arreglo debe ser contraído dinámicamente.
60. Implante la substracción de polinomios.
61. Implante la división de polinomios (++).
62. Implante la operación residuo producto de la división de dos polinomios (++)
63. Modifique el TAD Polinomio para que haga los términos visibles al usuario y permita accederlos eficazmente mediante un iterador.

64. Implante una función que calcule la derivada de un polinomio.
65. Implante una función que calcule la integral de un polinomio.
66. Modifique el TAD polinomio para que utilice coeficientes en punto flotante.
67. Implante una función que evalúe un polinomio.
68. Escriba un algoritmo de una sola pasada que encuentre el menor elemento de una lista enlazada.
69. Dada una lista circular doblemente enlazada, escriba un algoritmo que coloque los nodos que están en posiciones impares después de los que están en posiciones pares. Se debe preservar el orden relativo entre los nodos.
70. Considere la eliminación de los elementos repetidos de una lista. Diseñe un algoritmo que filtre los elementos repetidos de una lista. Realice para los siguientes casos:
  - (a) La lista es simplemente enlazada.
  - (b) La lista es doblemente enlazada.

En ambos casos, el resultado debe componerse de dos listas. Una correspondiente al filtrado; la otra correspondiente a los elementos suprimidos.

71. Suponga dos listas simplemente enlazadas A y B y las operaciones de unión e intersección. Escriba algoritmos que calculen la unión e intersección si:

- (a) Las listas están ordenadas.
- (b) Las listas no están ordenadas.

Analice el desempeño de cada uno de los algoritmos.

72. Suponga dos listas enlazadas A y B. Considere la operación de fusión cuyo resultado será una sola lista cuyos primeros elementos corresponden a la lista A seguida por los elementos de la lista B. Escriba un algoritmo de fusión para cada uno de los siguientes casos:

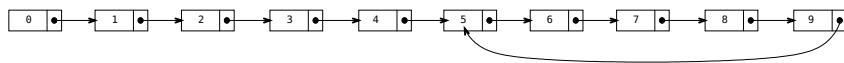
- (a) Listas simples no circulares.
- (b) Listas simples circulares.
- (c) Listas dobles no circulares.
- (d) Listas dobles circulares.

73. Considere el problema de invertir los elementos de una lista en una sola pasada, sin usar una estructura de dato adicional y sin copiar los contenidos de los nodos. Escriba algoritmos para:

- (a) Una lista simplemente enlazada.
- (b) Una lista simplemente enlazada circular.
- (c) Una lista doblemente enlazada. Proponga al menos dos algoritmos.

(d) Una lista doblemente enlazada circular. Proponga al menos dos algoritmos.

74. Suponga la situación siguiente:



(a) Explique un algoritmo que determine si la lista en cuestión tiene o no un ciclo y, si existe un ciclo, cuál es el nodo de comienzo y cuál es su longitud.

Discuta el desempeño de su método.

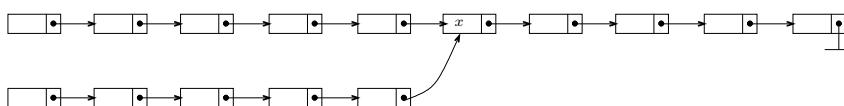
(b) Asuma que es imposible mantener alguna estructura de dato o escribir sobre los nodos. Sólo es posible algunas celdas de memoria.

i. Mencione algún escenario real donde estas restricciones se aplicarían.

ii. Con las restricciones dadas, explique un método que ocupe espacio constante y que determine si existe un ciclo, en cuál nodo y de cuánta longitud.

iii. ¿Cuál es el desempeño del método anterior?

75. Imagine la siguiente situación:



El problema consiste en determinar cuál es el nodo de intersección, cuántos nodos preceden a  $x$  en cada lista, y cuántos nodos suceden a  $x$ .

(a) Suponiendo que se puede marcar cualquier nodo, diseñe un algoritmo que no contenga ciclos anidados.

(b) A partir de ahora asuma que no se puede marcar ningún nodo. Diseñe un algoritmo cándido basado en dos ciclos anidados. El algoritmo puede ocupar espacio proporcional al número de nodos.

(c) Diseñe un algoritmo basado en dos ciclos anidados que ocupe espacio constante.

(d) Diseñe un algoritmo basado en ciclos simples, no anidados, que ocupe espacio constante.

76. Considere la operación de partición de una lista en dos listas de igual tamaño. La primera lista corresponde a los  $n/2$  primeros elementos y la segunda a los  $n/2$  elementos restantes. Escriba algoritmos basados en ciclos simples para partitionar:

(a) Una lista simplemente enlazada.

(b) Una lista simplemente enlazada circular.

(c) Una lista doblemente enlazada.

(d) Una lista doblemente enlazada circular.

77. Escriba un procedimiento general, basado en ciclos simples, que efectúe una  $m$ -partición. Es decir, después de la operación, la lista es partida en  $m$  listas de tamaño  $n/m$ .

78. Escriba un programa que mueva el menor elemento de una lista a la primera posición.
79. Escriba un programa que mueva el mayor elemento de una lista a la última posición.
80. Con base en los dos ejercicios previos, escriba un programa que ordene una lista doblemente enlazada.
81. Diseñe e implante un TAD que modelice una pila implantada con arreglos dinámicos. Discuta todas las opciones, ventajas y desventajas de su diseño.
82. Diseñe e implante un TAD que modelice una cola implantada con arreglos dinámicos. Discuta todas las opciones, ventajas y desventajas de su diseño.
83. Considere una lista circular de forma tal que la lista pueda recorrerse eficazmente en ambas direcciones.
- (a) Considere una función  $f(p_1, p_2) = p_3$  tal que  $f(p_1, p_3) = p_2$ ,  $f(p_2, p_3) = p_1$ ,  $\forall p_i, p_j, p_k \in \mathcal{N}$ ,  $p_i \neq p_j \neq p_k$ . Dado un nodo con un solo campo para una dirección de memoria, explique un método que utilice una función como la anterior para recorrer la lista en los dos sentidos aún cuando un nodo contiene un solo apuntador (++).
- (b) Encuentre operaciones que puedan usarse para implantar la función  $f$  de la pregunta anterior (+).
- (c) Con base en las respuestas anteriores, diseñe e implante un TAD que modelice una lista circular recorrible en ambos sentidos y que use un solo apuntador por nodo (++).
84. Implante los métodos `pushn()`, `popn()` y `empty()` definidos en `ArrayList<T>`, para la clase `ListStack<T>`. ¿Por qué no han sido implantados directamente dentro de la clase?
85. Explique como implantar una pila mediante dos colas.
86. Explique como implantar un cola mediante dos pilas.
87. Para convertir un número en base 10 se divide sucesivamente entre dos hasta que el cociente devenga en cero y en cada división se almacena el resto, el cual es un dígito del equivalente en base binaria.

Por ejemplo,  $37_{10}$  puede calcularse en binario como:

$$\begin{array}{rcl}
 37 \mod 2 & = & 1 \\
 18 \mod 2 & = & 0 \\
 9 \mod 2 & = & 1 \\
 4 \mod 2 & = & 0 \\
 2 \mod 2 & = & 0 \\
 1 \mod 2 & = & 1
 \end{array}$$

De este modo,  $37_{10} = 100101_2$ .

Implante, usando una pila, la rutina:

```
void imprime2(const int & n);
```

La cual imprime el equivalente binario del parámetro  $n$ .

88. Escriba un programa que evalúe una expresión prefija.
89. Escriba un programa que evalúe una expresión sufija.
90. Expanda el evaluador de expresiones infijas para que efectué operaciones de exponenciación, trigonometría y logaritmos.
91. Modifique el evaluador de expresiones infijas para que, en lugar de evaluar la expresión, genere la expresión sufija. Incluya las operaciones de exponenciación, trigonometría y logaritmos.
92. Modifique el evaluador de expresiones infijas para que, en lugar de evaluar la expresión, genere la expresión prefija. Incluya las operaciones de exponenciación, trigonometría y logaritmos.
93. Diseñe e implante un TAD que modelice dicolas implantadas con arreglos.
94. Diseñe e implante un TAD que modelice dicolas implantadas con listas.
95. Discuta las alternativas de diseño de un TAD que modelice matrices esparcidas representadas con multilistas. Explique cómo se realizarían las operaciones de suma, resta, multiplicación e inversión.
96. Calcule los costes en espacio de una matriz representada mediante multilistas. Proporcione una expresión analítica en función de la dimensión de la matriz y del porcentaje de elementos nulos.
97. Diseñe e implante un TAD que modelice matrices esparcidas representadas con multilistas. Implante las operaciones de suma, resta, multiplicación e inversión (++).

# 3

## Crítica de algoritmos

En este capítulo estudiaremos algunas técnicas de análisis y verificación de correctitud cuyo fin es criticar algoritmos. Criticar proviene del griego κριτικός (criticos), que significa “el que critica”, el cual, a su vez, deriva del verbo griego κρίνω (crino) y connota separar, discernir, distinguir, cribar<sup>1</sup>. Criticar consiste, pues, en “separar”, “seleccionar” la cosa de interés a efectos de estudiarla. Pero, ¿para qué la estudiamos? Tanto desde las perspectivas del obrante como desde las de los beneficiarios de la obra, la crítica se hace con pretensión de mejorar el bien.

Hay dos maneras de ver una crítica. Una se concentra en lo “interno”<sup>2</sup> y ataña al proceso de hechura de la obra. Este tipo de crítica es más propio entre los obrantes (practicantes) que entre los beneficiarios que reciben (usan) la obra.

A la otra faceta de la crítica se le denomina “externa” y ataña a la apreciación social de la obra, es decir, cómo el entorno o sociedad aprecia o desprecia el bien de la obra. La crítica externa es fundamental y primaria respecto a la interna, pues ésta ataña al destino de la obra.

La crítica externa se concentra en el qué es la obra, lo que ésta representa socialmente como bien, mientras que la crítica interna se concentra en el cómo se lleva a cabo la obra.

La ciencia moderna suele usar el término “análisis” para referir a un estilo técnico de estudio interno en la cual la cosa de estudio se divide en partes para estudiarlas por separado. “Análisis” proviene del griego ἀνάλυσις (análisis) y en última instancia del verbo ἀναλύω, el cual también significa “soltar” (ἀνά), distintivamente, por partes, y “separar” (λύω), lo cual corresponde con su primera acepción en el D.R.A.E.: “distinguir y separar las partes de un todo”. Debemos aclarar vehementemente que criticar es mucho más que analizar y, para aprehender ello, quizá lo mejor sea mirar el antónimo de análisis, cual es “síntesis”, proveniente del griego σύνθεσις (síntesis) que, D.R.A.E. dixit, significa “composición de un todo por la reunión de sus partes”. Como debemos apreciar en este contexto, componer programas, o sea, sintetizarlos, tiene más sentido para nosotros que descomponerlos, es decir, que analizarlos. Surge entonces la siguiente pregunta, que dejaremos abierta: cuando programamos, ¿analizamos o sintetizamos o ambos? Planteado de otro modo: ¿creamos programas a partir de un todo que luego descomponemos o a partir de partes que después componemos, ambas cosas o ninguna?

Las críticas suelen ser clasificadas en “constructiva” o “destructiva”. Hoy se aboga

---

<sup>1</sup>Una criba es una especie de colador con que antiguamente se limpiaba el trigo del polvo y otras impurezas. Quizá por esta acepción sea más aceptable decir que criticar no significa dividir.

<sup>2</sup>Según [113].

por que la crítica sea constructiva y no destructiva. ¿Por qué estos adjetivos?, ¿qué se construye o destruye? Más en particular, ¿por qué una crítica podría ser destructiva? Es correcto decir que una obra se “construye”, sentido por el cual una crítica, sobre todo, cuando ésta revela errores, cuyas asunciones y enmiendas mejoran la obra, puede considerarse constructiva. Aunque hemos dicho que, etimológicamente, el criticar conlleva “romper”, no debemos aceptar que una crítica destruya literalmente a una obra; todo lo contrario, cuanto más aguda y precisa sea una crítica, más “constructiva” es, pues ésta permite mejorar la obra. Lo anterior es correcto inclusive si una crítica revela que no tiene sentido construir la obra, pues, al fin y al cabo, éste es un conocimiento práctico. ¿Cómo es entonces que una crítica puede calificarse de destructiva?

Ocurre que la razón de la crítica de esta época se centra en el individuo, o sea, en la personalidad del obrante, quien es el criticado, y del criticante, en lugar de concentrarse sobre la obra como un todo. En este marco, cuando el obrante de esta época escucha una crítica, es muy posible que en lugar de considerarla hacia el sentido de la obra, qué es ella y cómo se lleva a cabo, se confunda con quién es el obrante y cómo es él como autor. Por las mismas razones, a menudo, el criticante incurre en el mismo error: concentra su crítica (si acaso ahora se le puede llamar así) en la personalidad del obrante, es decir, en el autor, en detrimento del sentido de la obra y su instrumentación. En el marco hedonista, la crítica es constructiva si se logra interpretar como elogio, mientras que es destructiva cuando se entiende como reproche. El problema es que al confundir el qué se hace con el quién hace, o, el cómo se hace con el cómo es quien lo hace, se pierde el valor esencial de la crítica, cual no es otro que mejorar, perseguir la excelencia y, en última instancia, el bien en el cual trasciende la obra.

Abocaremos el objeto de estudio de este capítulo desde dos perspectivas: la eficacia y la eficiencia.

La eficacia concierne a que se logre el efecto, es decir, a que el algoritmo alcance el fin de resolver el problema para el cual fue destinado. Esto es la “correctitud”.

La eficiencia atañe a qué tan bien un algoritmo o programa alcanza su solución, lo cual implica decir que para poder hablar de eficiencia, previamente debe haberse sido eficaz. Podemos decir entonces que la eficiencia atañe a cómo se puede ser mejor eficaz.

Si un programa no es correcto, entonces posiblemente no tenga sentido hablar de eficiencia. Por eso la correctitud prima a la eficiencia, pues de nada vale la última si no se alcanza el efecto, es decir, si no se resuelve el problema.

Para estudiar la eficiencia se hacen análisis. En este sentido caben preguntas como: ¿cuánto dura el programa?, ¿cuántos recursos computacionales consume? El sentido crítico sobre esta clase de preguntas determinará la factibilidad de un programa y las circunstancias en que éste pueda o deba usarse.

Durante un análisis, es muy posible que se revelen otras formas o variantes para elaborar el algoritmo o programa, algunas de ellas lo mejorarán, otras lo empeorarán y otras develarán algoritmos muy distintos. Por añadidura, el entrenamiento en el análisis de algoritmos requiere su compresión cabal. Consecuentemente, analizar algoritmos es una de las mejores maneras de ganar conocimiento sobre su diseño.

Cuando se tiene claro el problema, un análisis puede indicar aspectos de correctitud. Veamos un ejemplo extremo: una duración nula, que no tiene sentido concreto, podría indicar no sólo un error en el análisis, sino en el algoritmo mismo.

En lo que sigue, entonces, abordaremos dos temas principales: el análisis de algoritmos

y su correctitud. Ambos atanen a la crítica interna, la cual (no debemos olvidarlo), no tiene sentido sin una perspectiva de crítica externa.

### 3.1 Análisis de algoritmos

Intuitivamente hay dos métricas fundamentales y elementales, las cuales, “en teoría”, permiten cuantificar la calidad de un algoritmo:

1. La duración de la ejecución, comúnmente llamada “tiempo de ejecución”, la cual consiste en medir el tiempo desde que se inicia el programa hasta que se culmina. En este sentido, cuanto menos dure un programa, mejor es éste.
2. La cantidad de consumo de otros recursos<sup>3</sup>; en particular, la memoria. Medimos los bytes de memoria y de otros recursos que ocupa un programa durante su ejecución. Entre dos programas a comparar, el que consuma menos recursos es el mejor.

Tanto la duración como el consumo de recursos dependen de la configuración de la entrada. Uno de los factores más importantes de esta configuración, pero no el único, es el tamaño de la entrada, pues cuanto mayor es ésta, posiblemente mayor es la cantidad de datos a procesar y mayores duraciones y cantidades de recursos hay que gastar. En este orden de ideas, en lo que sigue nos referiremos a  $T(n)$  como una función que describe el comportamiento de un algoritmo, según su duración o su consumo de recursos, al tamaño de la entrada  $n$ .

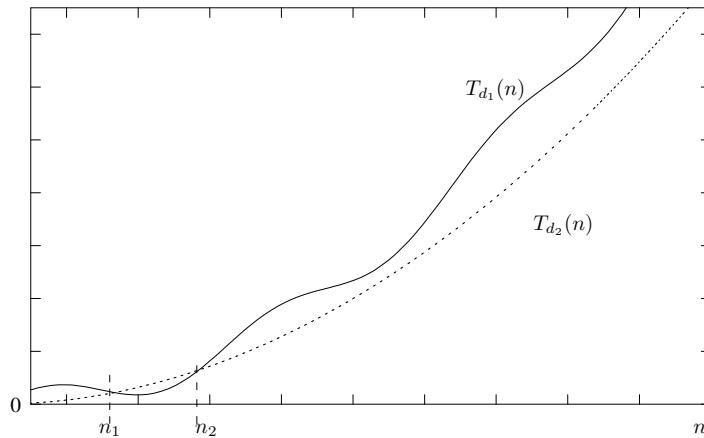


Figura 3.1: Un ejemplo de  $T_d$

El tamaño  $n$  es una parte de la configuración, la cual, en sí, consiste en el orden o permutación en que le aparece la entrada al programa. Puesto que conocer todas las permutaciones posibles de una entrada es muy laborioso (y subjetivo), en el análisis de un algoritmo se suelen emplear algunos estadísticos, de los cuales, los principales son el promedio o esperanza y el máximo.

Las observaciones anteriores suponen que el tamaño de la entrada puede expresarse en función de una sola variable, lo cual no siempre es el caso. En algunas circunstancias,

<sup>3</sup>“otros” porque el tiempo de ejecución puede también considerarse como un recurso.

el tamaño, u otra cuantificación acerca de la entrada, se definen mediante una función multivariable. Estos casos son, por supuesto, más complejos que el univariable.

Dados dos algoritmos  $A_1$  y  $A_2$ , que resuelven el mismo problema, ¿cuál de los dos es mejor? Objetiva e idóneamente hablando, podemos calcular a priori, o sea, sin ejecutar los algoritmos, las funciones de duración  $T_{d_1}(n)$  y  $T_{d_2}(n)$  y las funciones de consumo de espacio  $T_{e_1}(n)$  y  $T_{e_2}(n)$ . Si tuviésemos estas funciones, entonces tendríamos criterios bastante objetivos y precisos para decidir. Por ejemplo, si  $T_d$  exhibiese las formas máximas de la figura 3.1, entonces sabríamos que para entradas mayores a  $n_2$   $T_{d_2}(n)$  es mejor que  $T_{d_1}(n)$ , pues esta última es la que tiene mayor duración.

El análisis anterior sólo es posible si se pueden conocer a priori las formas de  $T_{d_1}(n)$  y  $T_{d_2}(n)$ . Lamentablemente, asumir tal posibilidad no es nada realista, pues para el caso general es extremadamente difícil calcular a priori  $T_d(n)$ . Para aprehender el porqué de esta dificultad es necesario comprender que calcular  $T_d(n)$  requiere contar la cantidad de "instrucciones" que ejecutaría la secuencia finita denominada algoritmo.

Afortunadamente, no es conveniente conocer con precisión a  $T_d(n)$ , pues, a la poste, la duración real dependerá de factores subjetivos que modificarán la forma exacta de  $T_d(n)$ . En efecto,  $T_d(n)$  tendrá formas diferentes según las circunstancias de ejecución del algoritmo. Entre tales circunstancias cabe destacar las siguientes características:

- **Características materiales del computador:** existen muchas arquitecturas diferentes de computadores. Algunas arquitecturas se prestan mejor para una clase de problemas que otras. Posiblemente, un algoritmo tendrá duraciones distintas en arquitecturas distintas.

Aun entre arquitecturas homogéneas existen diferencias dadas por la velocidad del reloj, cantidad de memoria, sistema operativo y número de procesos que actualmente alberga el computador, entre otros factores que inciden en la duración de un programa.

- **Características de la codificación:** un algoritmo se instrumenta en algún lenguaje de programación. Lenguajes diferentes que codifiquen el mismo algoritmo producirán programas con duraciones diferentes.

Por añadidura, los programadores también difieren en sus estilos de codificación. Un algoritmo implantado por dos programadores en un mismo lenguaje y ejecutado en un sólo computador, probablemente exhibirá diferencias en el tiempo de ejecución.

- **Características del código generado por el compilador:** los compiladores difieren en la traducción del código fuente al objeto. Es bastante probable que dos compiladores diferentes generen traducciones diferentes de un mismo programa y, por tanto, dichos programas exhibirían tiempos de ejecución diferentes.

Estos factores, entre otros más, heterogéneos entre sí, hacen más arduo -además de impráctico-, conocer no sólo la forma exacta de  $T(n)$ , sino expresar el tiempo de ejecución de un programa en alguna unidad específica y precisa. Debemos, pues, indagar un criterio de análisis que eluda los factores mencionados.

### 3.1.1 Unidad o paso de ejecución

Como ya señalamos, analizar un algoritmo consiste en contar la cantidad de instrucciones o pasos de ejecución que éste ejecuta. ¿Qué es un paso de ejecución? En principio podemos

decir que es la mínima instrucción que se puede ejecutar. Esta noción establece un grano mínimo que nos permite cuantificar la duración de un algoritmo sin considerar el tiempo absoluto de ejecución. ¿Qué se considera una mínima instrucción? Las consideraciones anteriores reflejan lo difícil que es sentar este umbral. Por tal razón, consideraremos un paso de ejecución como cualquier secuencia de ejecución cuya duración sea constante. Transamos por lo constante sin saber cual es la duración real.

Propiciemos el entendimiento mediante algunos ejemplos. La siguiente instrucción:

$$149a \quad \langle \text{ejemplo 1 paso de ejecución 149a} \rangle \equiv \quad (149e)$$

$$\quad \quad \quad x = y;$$

se considera un paso de ejecución. Cada vez que el flujo pase por esta instrucción, su ejecución tendrá una duración máxima “constantemente acotada”. En el mismo sentido

$$149b \quad \langle \text{ejemplo 2 paso de ejecución 149b} \rangle \equiv \quad (149e)$$

$$\quad \quad \quad x = y + z;$$

cuya duración es mayor que el  $\langle \text{ejemplo 1 paso de ejecución 149a} \rangle$ , también está constantemente acotada; por tanto, el  $\langle \text{ejemplo 2 paso de ejecución 149b} \rangle$  también se considera un paso de ejecución.

Una secuencia de instrucciones tal como la siguiente:

$$149c \quad \langle \text{ejemplo 3 paso de ejecución 149c} \rangle \equiv \quad (149e)$$

$$\begin{aligned} &\quad x = h; \\ &\quad \text{if } (x < y) \\ &\quad \quad \{ \\ &\quad \quad \quad x = y + z; \\ &\quad \quad \quad a = b * c; \\ &\quad \quad \} \\ &\quad \text{else} \\ &\quad \quad \{ \\ &\quad \quad \quad x = p + q + r; \\ &\quad \quad \quad a = b + c; \\ &\quad \quad \} \end{aligned}$$

puede considerarse más costosa que  $\langle \text{ejemplo 2 paso de ejecución 149b} \rangle$ , sin embargo, independientemente del valor del predicado y de lo larga que pueda ser la ejecución de cualquiera de los bloques,  $\langle \text{ejemplo 3 paso de ejecución 149c} \rangle$  es un paso de ejecución, pues éste tiene una duración que puede acotarse por una constante.

El bloque repetitivo

$$149d \quad \langle \text{ejemplo 4 paso de ejecución 149d} \rangle \equiv \quad (149e)$$

$$\begin{aligned} &\quad \text{for } (i = 0; i < 100000; i++) \\ &\quad \quad \quad a[i] = b[i] + c[i]; \end{aligned}$$

también tiene una duración constante y se considera, consecuentemente, un paso de ejecución.

Finalmente, la ejecución secuencial de todos los bloques anteriores

$$149e \quad \langle \text{Último ejemplo paso de ejecución 149e} \rangle \equiv$$

$$\begin{aligned} &\quad \langle \text{ejemplo 1 paso de ejecución 149a} \rangle \\ &\quad \langle \text{ejemplo 2 paso de ejecución 149b} \rangle \\ &\quad \langle \text{ejemplo 3 paso de ejecución 149c} \rangle \\ &\quad \langle \text{ejemplo 4 paso de ejecución 149d} \rangle \end{aligned}$$

también es un paso de ejecución, pues su duración, a pesar de que comprenda a la de todos los bloques anteriores, también es constante.

A partir de este momento, una **unidad de ejecución** se define como una secuencia de instrucciones cuya duración es constante. Esto conlleva errores, pero recordemos que el tiempo de ejecución varía según circunstancias ajenas al algoritmo. Cualquiera que sea el tamaño y la disposición de la entrada, un flujo de ejecución siempre tendrá duración constante cada vez que pase por alguna instrucción secuencial. En el mismo espíritu de razonamiento, cualquier secuencia de instrucciones con duración constante también se considera una unidad de ejecución.

Es importante destacar el sacrificio que impone nuestro concepto de unidad de ejecución al considerar como constante lo que de hecho es, en la mayoría de los casos, variable. El concepto equipara, por ejemplo, una secuencia de diez instrucciones con una de un millón pues en ambos casos se conocen las constantes diez y un millón, respectivamente. Sin embargo, está claro que, muy probablemente, la ejecución de la primera secuencia dure  $10^5$  veces menos que la segunda. Bajo la conciencia del sacrificio anterior, la ventaja del concepto es que el carácter constante no se pierde con lo variable de la entrada. En palabras más simples, secuencias de ejecución cuyas longitudes sean distintas pero constantes, siempre tendrán duraciones constantes no afectadas por el tamaño de la entrada. Esto último es lo apreciable del concepto.

### 3.1.2 Aclaratoria sobre los métodos de ordenamiento

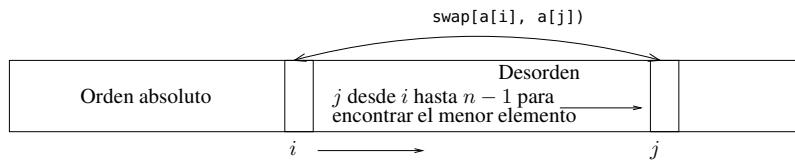
En este capítulo nos serviremos de algunos métodos de ordenamiento para ilustrar el análisis de algoritmos, pues éstos plantean “naturalmente” el tamaño de la entrada con base en la cantidad de elementos a ordenar. Por añadidura, la permutación de la secuencia de entrada incide en el desempeño de la mayoría de los métodos de ordenamiento. Por esta razón, cuando se analiza un método de ordenamiento hay que considerar, por lo general con criterios estadísticos, la forma de la permutación.

A parte de fungir como un buen vehículo de enseñanza para el análisis y diseño de algoritmos, los métodos de ordenamiento son parte de los conocimientos esenciales de todo buen programador. En efecto, el ordenamiento se usa en una amplia gama de problemas más complejos. Por ejemplo, en grafos, ámbito que estudiaremos en el capítulo 7, a veces se requieren ordenar relaciones como preprocesamiento a algunos algoritmos. Otro ejemplo notable se encuentra en el “cascarón convexo”, problema de la geometría computacional que consiste en encontrar un polígono que “envuelva” a un conjunto de puntos. Un algoritmo simple y eficiente para este problema requiere ordenar los puntos según sus coordenadas.

Los métodos de ordenamiento, así como otros utilitarios relacionados, residen en el archivo `tpl_sort_utils.H`.

### 3.1.3 Ordenamiento por selección

Hagamos nuestro primer análisis a través de la noción de paso de ejecución. A tal fin, consideremos el método de ordenamiento de selección, el cual puede plasmarse pictóricamente como sigue:



En esta clase de diagrama, el rectángulo principal representa el arreglo. Los subrectángulos internos representan pedazos del arreglo que conciernen a su estado de ordenamiento. Las flechas, junto con algún nombre de variable, indican un contador o iterador junto con su sentido de iteración.

Nuestra implantación del ordenamiento por selección se estructura del siguiente modo: cuya implantación es:

151a *(Métodos de ordenamiento 151a)*≡ 151c▷

```
template <typename T, class Compare = Aleph::less<T> > inline
void selection_sort(T * a, const size_t & n)
{
 for (int i = 0, min, j; i < n - 1; ++i)
 {
 <Sea min el índice menor entre i + 1 y n - 1 151b>
 if (Compare () (a[min], a[i]))
 std::swap(a[min], a[i]);
 }
}
```

La búsqueda del menor elemento se hace secuencialmente de la siguiente manera:

151b *(Sea min el índice menor entre i + 1 y n - 1 151b)*≡ (151a)

```
for (min = i, j = i + 1; j < n; ++j)
 if (Compare () (a[j], a[min]))
 min = j;
```

El mayor coste del if incluido en el bloque *(Sea min el índice menor entre i + 1 y n - 1 151b)* es que el predicado sea cierto. Si este fuese el caso, la duración del if aún permanece constantemente acotada. Concluimos, pues, que el bloque dentro del for toma una unidad de ejecución. Por lo tanto, el bloque *(Sea min el índice menor entre i + 1 y n - 1 151b)* ejecuta  $n - i - 1$  pasos de ejecución, el cual puede aproximarse, con un ligero error, a  $n - i$ . De este modo, el método toma:

$$T(n) = \sum_{i=0}^{n-2} n - i = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = (n-1)n - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} \text{ Pasos de ejecución } (3.1)$$

Esto nos revela que independientemente de que tengamos un computador muy rápido o muy lento, y demás idiosincrasias presentadas en 3.1.1, el tiempo de ejecución crece cuadráticamente según el número de elementos que estemos ordenando. ¿Puede encontrarse algún método objetivo que caracterice el tiempo de ejecución sin considerar los aspectos relativos? La respuesta es afirmativa y la abordaremos en la siguiente sub-sección.

Exactamente el mismo algoritmo puede instrumentarse para ordenar listas doblemente enlazadas cuyo nodo cabecera sea de tipo Dlink. Pero para eso es conveniente instrumentar una rutina que busque el menor elemento de una lista enlazada:

151c *(Métodos de ordenamiento 151a)+≡* ◁151a 152▷

```
template <class Compare> inline Dlink * search_min(Dlink & list)
{
 typename Dlink::Iterator it(list);
```

```

Dlink * min = it.get_current();
for (it.next(); it.has_current(); it.next())
{
 Dlink * curr = it.get_current();
 if (Compare () (curr, min))
 min = curr;
}
return min;
}

```

Defines:

`search_min`, used in chunks 152 and 156b.

Es extremadamente importante notar que el método de ordenamiento recibe el tipo `Dlink`, desde el cual no se puede conocer genéricamente el tipo dato que permite hacer las comparaciones que conducirán al ordenamiento. Justamente, este es el fin de la clase `Compare`, la cual debe implantar el operador `bool Compare::operator () (Dlink *, Dlink *)` y que se encarga de comparar dos nodos apuntados por variables `Dlink*`. Podríamos implantar el ordenamiento de listas a partir del tipo `Dnode<T>`, desde cual ya conoceríamos un tipo de base. Pero esta decisión excluiría el ordenamiento de listas basadas en otros usos de `Dlink`; por ejemplo, supongamos la siguiente estructura de nodo:

```

struct Nodo
{
 Dlink enlace_lista_1;
 Dlink enlace_lista_2;
 Dlink enlace_lista_3;
 D1 d1;
 D2 d2;
 D3 d3;
};

```

Esta clase de nodo no podría ordenarse por una rutina genérica de ordenamiento basada en un `Dnode<T>`, pues el tipo ejemplo `Nodo` no es un `Dnode<T>`. Por añadidura, `Nodo` tiene tres enlaces que se encadenan a listas diferentes. ¿Cómo ordenamos genéricamente según cualquiera de las tres listas? La respuesta consiste en delegar la comparación a `bool Compare::operator () (Dlink *, Dlink *)`. Por ejemplo, si deseamos ordenar la lista 1 según el atributo `d1`, podemos hacerlo como sigue:

```

struct Comparar_D1
{
 bool operator () (Dlink * l1, Dlink* l2)
 {
 Nodo * n1 = Dlink::dlink_to_Node(l1);
 Nodo * n2 = Dlink::dlink_to_Node(l2);
 return n1->d1 < n2->d1;
 }
}

```

Análogamente, podemos realizar comparaciones para los tipos `D2` y `D3` y así ordenar por los enlaces `enlace_lista_2` y `enlace_lista_3`.

Con lo anterior en mente, el ordenamiento por selección se explica por sí mismo:

```

{
 Dlink aux;
 while (not list.is_empty())
 {
 Dlink * min = search_min <Compare> (list); // menor de list
 min->del(); // saque menor de list
 aux.append(min); // insértelo ordenado en aux;
 }
 list.swap(&aux);
}

```

Uses `search_min` 151c.

Esta versión expresa mejor el principio del método: buscar el menor elemento e insertarlo en una lista. La lista resultante estará ordenada. El método es el mismo que aplicamos cuando ordenamos barajas.

El usar:

```
bool Compare::operator () (Dlink *, Dlink *)
```

como criterio general de comparación pretende servir para todas las circunstancias de ordenamiento de listas enlazadas. Sin embargo, en la mayoría de las circunstancias, se tratará de ordenar una lista de `Dnode<T>`, en la cual sí se conoce el tipo. Para endulzar el asunto diseñaremos la versión genérica:

```
bool Compare::operator () (Dnode<T> *, Dnode<T> *)
```

la cual será el criterio de comparación por omisión:

153a *(Métodos de ordenamiento 151a)*+≡ ◁152 153b ▷

```

template <typename T, class Compare> class Compare_Dnode
{
 bool operator () (Dlink * l1, Dlink * l2) const
 {
 Dnode<T> * n1 = static_cast<Dnode<T>*>(l1);
 Dnode<T> * n2 = static_cast<Dnode<T>*>(l2);
 return Compare () (n1->get_data(), n2->get_data());
 }
};

```

Defines:

`Compare_Dnode`, used in chunks 153b and 165a.

Uses `Dnode` 83a.

De este modo podemos exportar una especialización de `selection_sort()` que considera un criterio de comparación de los contenidos de los nodos:

153b *(Métodos de ordenamiento 151a)*+≡ ◁153a 154a ▷

```

template <typename T, class Compare = Aleph::less<T> >
inline void selection_sort(Dnode<T> & list)
{
 selection_sort < Compare_Dnode<T, Compare> > (list);
}

```

Uses `Compare_Dnode` 153a and `Dnode` 83a.

### 3.1.4 Búsqueda secuencial

Antes de pasar a estudiar un corpus objetivo para analizar algoritmos, presentemos uno de los aspectos que a menudo complican el análisis o selección de un algoritmo: el azar. Para eso nos valdremos de uno los algoritmos más simples, populares y útiles: la búsqueda secuencial. Comencemos por un arreglo, cuya búsqueda puede plasmarse, sin necesidad de explicación previa, de la siguiente manera:

154a *(Métodos de ordenamiento 151a) +≡* ◀153b 154b▶  
 template <typename T, class Equal = Aleph::equal\_to<T> > inline  
 int sequential\_search(T \* a, const T & x, const int & l, const int & r)  
 {  
   for (int i = l; i <= r; i++)  
     if (Equal () (a[i], x)) return i;  
   return Not\_Found;  
 }

Defines:

sequential\_search, used in chunks 15b, 31b, 154, and 155.

sequential\_search() busca secuencialmente en el arreglo a, entre los índices l y r, el elemento x. Si la búsqueda es exitosa, entonces se retorna el índice contentivo del elemento; de lo contrario, se retorna Not\_Found.

sequential\_search() emplea como criterio de igualdad el operador () de la clase parametrizada Equal.

¿Cuántos pasos de ejecución consume este algoritmo? La sutileza de su análisis es que la duración de ejecución depende de dos factores: (1) la permutación del arreglo a, y (2) del valor de búsqueda x.

Veámoslo desde la perspectiva del valor de x. Si x no está contenido en el arreglo, entonces, independientemente de la permutación, el for se ejecutará completamente, razón por la cual el algoritmo consume  $r - l = n$  pasos de ejecución. De lo contrario, entonces, la duración dependerá de la permutación; específicamente, de la posición en donde se encuentre x dentro del arreglo. Si se encuentra en la primera posición, entonces la duración es un paso de ejecución. Si se encuentra en la última entonces son  $n - 1$ . ¿Cuál escoger? No lo sabemos, pero sí podemos tener una expectativa de lo que va a suceder y ésta la expresa el concepto de esperanza estadística.

Si la permutación es aleatoria, entonces la esperanza será por el centro. La duración esperada será  $\frac{l+r}{2}$ .

Culminemos esta sección con las versiones genéricas de la búsqueda sobre listas:

154b *(Métodos de ordenamiento 151a) +≡* ◀154a 155a▶  
 template <typename T, class Equal = Aleph::equal\_to<T> > inline  
 Dnode<T> \* sequential\_search(Dnode<T> & list, const T & x)  
 {  
   for (typename Dnode<T>::Iterator it(list); it.has\_current(); it.next())  
   {  
     Dnode<T> \* curr = it.get\_current();  
     if (Equal () (curr->get\_data(), x))  
       return curr;  
   }  
   return NULL;  
 }

Uses Dnode 83a and sequential\_search 154a.

Es fácil adaptar estas búsquedas al TAD DynDlist<T>;

155a *(Métodos de ordenamiento 151a)* +≡ 154b 156a

```
template <typename T, class Equal = Aleph::equal_to<T> > inline
T * sequential_search(DynDlist<T> & list, const T & x)
{
 Dnode<T> * ret = sequential_search<T, Equal>((Dnode<T>&) list, x);
 return ret != NULL ? &ret->get_data() : NULL;
}
```

Uses Dnode 83a, DynDlist 85a, and sequential\_search 154a.

### 3.1.5 El problema fundamental y los arreglos dinámicos

Quizá la manera más simple y versátil de instrumentar el problema fundamental definido en § 1.3 (Pág. 17) sea hacerlo mediante el tipo DynArray<T>. En este caso, transamos por un arreglo desordenado que puede crecer y decrecer dinámicamente. La ventaja del desorden es que no requerimos abrir y cerrar brechas ante inserciones y eliminaciones. La desventaja es que la búsqueda debe ser secuencial:

155b *(tpl\_dynarray\_set.H 155b)* ≡

```
template <typename T, class Equal = Aleph::equal_to<T> >
class DynArray_Set : public DynArray<T>
{
 T & insert(const T & item)
 {
 (*this)[this->size()] = item;
 return this->access(this->size() - 1);
 }
 T * search(const T & item)
 {
 int i =
 Aleph::sequential_search<T, Equal>(*this, item, 0, this->size() - 1);
 return (i >= 0 and i < this->size()) ? &this->access(i) : NULL;
 }
 void remove(T & item)
 {
 item = access(this->size() - 1);
 this->cut(this->size() - 1);
 }
};
```

Uses DynArray 34 and sequential\_search 154a.

La rutina sequential\_search() lleva a cabo, como homónimamente se infiere, una búsqueda secuencial sobre el arreglo dinámico<sup>4</sup>.

Claramente, la inserción es sumamente rápida, pues equivale a un paso de ejecución. La búsqueda depende la suerte, pero en el peor de los casos, cuando se trate de una fallida, tomará  $n$  pasos de ejecución ( $n$  es la cantidad de elementos). Finalmente, la eliminación toma un paso de ejecución, pero se requiere conocer una referencia al elemento que deseamos eliminar, lo cual a menudo exige una búsqueda exitosa que está acotada por  $n$  pasos de ejecución.

---

<sup>4</sup>Véase § 3.1.4 (Pág. 154) para más detalles.

### 3.1.6 Búsqueda de extremos

Un caso especial de la búsqueda es la determinación de un valor extremo de una secuencia; es decir, el valor mínimo o máximo. A través de la clase de comparación genérica, esto puede resolverse, para arreglos, del siguiente modo:

156a *(Métodos de ordenamiento 151a) +≡* △155a 156b▷  
 template <typename T, class Compare = Aleph::less<T> > inline  
 int search\_extreme(T \* a, const int & l, const int & r)  
 {  
 T extreme\_index = l;  
 for (int i = l + 1; i <= r; i++)  
 if (Compare () (a[i], a[extreme\_index])) // ¿se ve un nuevo menor?  
 extreme\_index = i; // si  
 return extreme\_index;  
 }

search\_extreme() busca el elemento extremo según el criterio de comparación Compare. Sin embargo, con esta sintaxis no queda semánticamente claro que se trata de buscar el mínimo o el máximo. Para aclarar el asunto usaremos dos distintas funciones:

156b *(Métodos de ordenamiento 151a) +≡* △156a 156c▷  
 template <typename T, class Compare = Aleph::less<T> > inline  
 int search\_min(T \* a, const int & l, const int & r)  
 {  
 return search\_extreme<T, Compare> (a, l, r);  
 }  
 template <typename T, class Compare = Aleph::greater<T> > inline  
 int search\_max(T \* a, const int & l, const int & r)  
 {  
 return search\_extreme<T, Compare> (a, l, r);  
 }

Uses search\_min 151c.

¿Cuál es la diferencia con la búsqueda particular que estudiamos en la subsección precedente? La estructura es prácticamente la misma: un lazo de  $r - l = n$  pasos; la diferencia radica en la detención. Para una permutación aleatoria, la búsqueda exitosa es proporcional a  $n$  en promedio, mientras que la búsqueda de un extremo requiere siempre examinar la totalidad de la secuencia. Consecuentemente, la duración de search\_extreme() es proporcional a  $n$  para todos los casos.

Para completar esta subsección debemos programar search\_extreme() para las listas basadas en Dlink, lo que se plantea del siguiente modo:

156c *(Métodos de ordenamiento 151a) +≡* △156b 163a▷  
 template <class Compare> inline Dlink \* search\_extreme(Dlink \* list)  
 {  
 typename Dlink::Iterator it(\*list);  
 Dlink \* curr\_min = it.get\_current();  
 for (it.next(); it.has\_current(); it.next())  
 {  
 Dlink \* curr = it.get\_current();  
 if (Compare () (curr, curr\_min))  
 curr\_min = curr;  
 }

```

 return curr_min;
}

```

Esta función se especializa para el resto de las clases de la jerarquía derivada desde Dlink.

### 3.1.7 Notación $\mathcal{O}$

La noción de paso de ejecución nos permitió encontrar una aproximación de  $T_d(n)$  para el ordenamiento por selección. Acullá, en la oquedad de las inmensidades de  $T_d(n)$ , podemos encontrar un patrón de constancia acerca de lo que acá, en la ejecución concreta de un algoritmo, se mide como variable. En el horizonte de aquella lejanía tiene sentido la siguiente definición.

**Definición 3.1 (Notación  $\mathcal{O}$ )** Sean dos funciones  $T(n)$  y  $F(n)$ . Entonces se dice que  $T(n)$  es  $\mathcal{O}(f(n))$ , denotado como  $T(n) = \mathcal{O}(f(n))$ , si y sólo si  $\exists c, n_1 \mid T(n) \leq cn^2, n \geq n_1$ .

Cuando decimos que  $T(n) = \mathcal{O}(f(n))$  estamos aproximando el comportamiento de  $T(n)$  a una función diferente y, para que tenga sentido práctico, más sencilla. Afirmemos, por ejemplo, que  $T(n) = n^2 + n + 1 = \mathcal{O}(n^2)$ . Si esto es correcto, entonces  $\exists c, n_1 \mid T(n) \leq cn^2$ . Seleccionemos  $c = 1$  e intentemos demostrar la afirmación.

Para probar que  $n^2 + n + 1 = \mathcal{O}(n^2)$  planteamos la inecuación  $n^2 + n + 1 \leq cn^2, n \geq n_1 \implies (1 - c)n^2 + n + 1 \leq 0$ . Lo que arroja como raíces

$$n = \frac{-1 \pm \sqrt{1 - 4(1 - c)}}{2(1 - c)}.$$

Si seleccionamos  $c = 7/4$  tenemos como raíces a  $n_0 = -2/3, n_1 = 2$ . Así pues,  $c = 7/4, n_1 = 2$  confirman que  $n^2 + n + 1 = \mathcal{O}(n^2)$ .

Demostrar que  $T(n) = \mathcal{O}(f(n))$  para alguna función  $f(n)$  no implica que no existan otras funciones que satisfagan la definición. De hecho, es posible decir que hay mejores funciones que otras. Por ejemplo, son demostrables las siguientes afirmaciones:

$$\sum_{i=1}^n i^2 = \mathcal{O}(n^4) \quad (3.2)$$

$$\sum_{i=1}^n i^2 = \mathcal{O}(n^3) \quad (3.3)$$

$$\sum_{i=1}^n i^2 = \mathcal{O}(n^2) \quad (3.4)$$

pero la última afirmación es la más precisa de las tres.

Puesto que hay dos incógnitas ( $c, n_1$ ) y una sola desigualdad, demostrar  $T(n) = \mathcal{O}(f(n))$  puede ser difícil en algunos casos matemáticos, pero, por lo general en el análisis de algoritmos, la demostración es sencilla. En algunos casos puede ser útil prefijar un valor de la constante  $c$  y resolver:

$$\lim_{n \rightarrow \infty} T(n) - c f(n) \leq 0 \quad (3.5)$$

La afirmación  $T(n) = \mathcal{O}(f(n))$  sólo tiene una dirección respecto a la igualdad. No es correcto decir  $\mathcal{O}(f(n)) = T(n)$ , pues no refleja la desigualdad de la definición.

Lo idóneo cuando se afirme que  $T(n) = \mathcal{O}(f(n))$ , es que  $f(n)$  acote asintóticamente a la función  $T(n)$ . A nivel asintótico, la mejor aproximación es que la asíntota de  $f(n)$  sea paralela a la de  $T(n)$ . Hay casos en que se puede demostrar que  $f(n)$  no es asintóticamente paralela a  $T(n)$  y esto se expresa mediante la siguiente definición.

**Definición 3.2 (Notación o)** Sean dos funciones  $T(n)$  y  $F(n)$ . Entonces se dice que  $T(n)$  es  $o(f(n))$ , denotado como  $T(n) = o(f(n))$ , si y sólo si  $\exists c, n_1 \mid T(n) < cf(n), n \geq n_1$ .

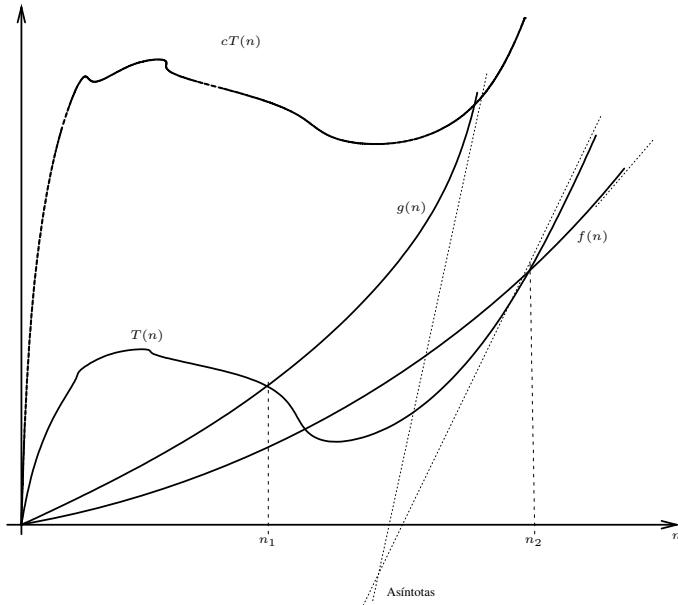


Figura 3.2: Ejemplo general de funciones que se acotan asintóticamente

La idea de estudiar un algoritmo mediante operaciones  $\mathcal{O}$  es obtener la aproximación asintótica más cercana al tiempo de ejecución. Podemos trabajar con una aproximación y no con un resultado exacto porque lo que nos concierne es la tendencia de  $T(n)$  en el algoritmo en sí, sin considerar los detalles subjetivos circunstanciales pertinentes al tipo de hardware, compilador y otros factores previamente mencionados. Podemos, pues, asumir que la constante  $c$  abstrae las diferencias circunstanciales de ejecución. Las diferencias de desempeño dadas por otros factores representan diferentes constantes cuyo valor no nos concierne. En el mismo sentido, la forma exacta de  $T(n)$  no es relevante, sino la tendencia asintótica a partir de la constante  $n_1$ .

En algunos casos pueden convenirnos las definiciones para los equivalentes inferiores de  $\mathcal{O}(f(n))$  y  $o(f(n))$ ; es decir, asíntotas que estén por debajo de  $T(n)$ .

**Definición 3.3 (Notación  $\Omega$ )** Sean dos funciones  $T(n)$  y  $F(n)$ . Entonces se dice que  $T(n)$  es  $\Omega(f(n))$ , denotado como  $T(n) = \Omega(f(n))$ , si y sólo si  $\exists c, n_1 \mid T(n) \geq cf(n), n \geq n_1$ .

**Definición 3.4 (Notación  $\omega$ )** Sean dos funciones  $T(n)$  y  $F(n)$ . Entonces se dice que  $T(n)$  es  $\omega(f(n))$ , denotado como  $T(n) = \omega(f(n))$  si y sólo si  $\exists c, n_1 \mid T(n) > cf(n), n > n_1$ .

Para indicar que la aproximación asintótica  $\mathcal{O}(f(n))$  es “paralela” a  $T(n)$  enunciamos la siguiente definición:

**Definición 3.5 (Notación  $\Theta$ )** Sean dos funciones  $T(n)$  y  $F(n)$ . Entonces se dice que  $T(n)$  es  $\Theta(f(n))$ , denotado como  $T(n) = \Theta(f(n))$ , si y sólo si  $T(n) = \mathcal{O}(f(n))$  y  $T(n) = \Omega(f(n))$ . Podemos decir también,  $T(n) = \Theta(f(n)) \iff T(n) = \mathcal{O}(f(n))$  y  $f(n) = \mathcal{O}(T(n))$ .

La figura 3.2 pictORIZA  $T(n)$  y algunas funciones que la acotan asintóticamente, por abajo y por arriba.

Identifiquemos en primer lugar la función  $T(n)$ , la cual describe hipotéticamente el tiempo de ejecución de un algoritmo. En la figura, la función  $g(n)$  está por encima de  $T(n)$  para todo  $n \geq n_1$ , es decir,  $T(n) = o(g(n))$ . La definición de  $o$  3.2 nos permite multiplicar  $T(n)$  por alguna constante  $c$  de modo que  $T(n)$  sea mayor que  $g(n)$ . Pero eso es irrelevante en la oquedad de lo infinito, pues la pendiente de  $g(n)$  (véanse sus puntos de corte) es mayor que la de  $T(n)$ . Consecuentemente, no importa cuánta sea la enormidad de la constante, en alguna lejanía,  $g(n)$  siempre sobrepasará a  $T(n)$ .

El mismo estilo de razonamiento puede usarse para constatar  $T(n) = \omega(f(n))$ . En el horizonte de los grandes números,  $T(n)$  sobrepasará a  $f(n)$ .

Ahora estamos preparados para comprender el sentido del concepto de  $\Theta$ , el cual puede apreciarse en la figura 3.3.

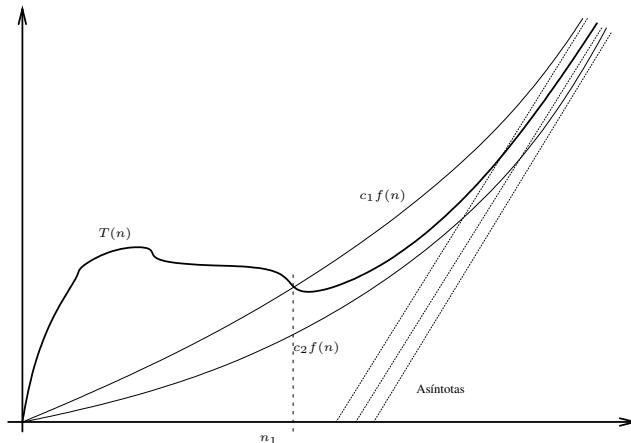


Figura 3.3: Ejemplo general de curva asintóticamente paralela

Observemos primero la curva  $c_2 f(n)$ , cuya asíntota es paralela a la de  $T(n)$ . Un cierto valor de constante  $c_2$  hace que  $T(n) = \Omega(f(n))$ . Ahora bien, bastaría con multiplicar  $f(n)$  por una constante  $c_1 > c_2$  para que  $T(n) = \mathcal{O}(f(n))$ . Por tanto,  $T(n) = \Theta(f(n))$

### 3.1.7.1 Álgebra de $\mathcal{O}$

En el mundo de lo infinito, lo que hace la diferencia es la tendencia que lleve la curva. Sumas y productos de constantes son descartables en el dominio de la notación  $\mathcal{O}$ . Esto posibilita un álgebra sobre la notación  $\mathcal{O}$  que simplifica considerablemente el análisis de un algoritmo. Para ello, permítasenos enunciar las siguientes reglas básicas, cuyas

demostraciones se delegan a ejercicios:

$$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max(f(n), g(n))) \quad (3.6)$$

$$c \mathcal{O}(f(n)) = \mathcal{O}(f(n)) \quad (3.7)$$

$$\mathcal{O}(\mathcal{O}(f(n))) = \mathcal{O}(f(n)) \quad (3.8)$$

$$\mathcal{O}(f(n)) \mathcal{O}(g(n)) = \mathcal{O}(f(n) g(n)) \quad (3.9)$$

$$\mathcal{O}(f(n) g(n)) = f(n) \mathcal{O}(g(n)) \quad (3.10)$$

$$T(n) = n^k + n^{k-1} + \dots + n + c = \mathcal{O}(n^k) \quad (3.11)$$

$$\log^k n = \mathcal{O}(n) \quad (3.12)$$

En ocasiones tenemos que comparar funciones con el objeto de determinar cuál es la mayor. A efectos del rigor, o si la comparación es difícil de establecer, planteamos el radio o razón entre las funciones como criterio de comparación y consecuente decisión. De este modo:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \begin{cases} 0 \Rightarrow f(n) = o(g(n)) \\ \infty \Rightarrow g(n) = o(f(n)) \\ c \Rightarrow f(n) = \Theta(g(n)) \end{cases} \quad (3.13)$$

### 3.1.7.2 Análisis de algoritmos mediante $\mathcal{O}$

Como ya harto hemos recalcado, en programación, a fin de cuentas todo es secuencia. Analizar un algoritmo requiere entonces, o “desenrollar” la máxima secuencia de ejecución posible, o “desenrollar” la secuencia de ejecución probable.

Para entender el sentido de “desenrollar”, consideremos el siguiente bucle:

```
for (int i = 0; i < 3; ++i)
 printf("Prueba %d\n", i);
```

“Desenrollar” este bucle equivale a escribir la ejecución como una secuencia:

```
int i = 0;
if (i < 3)
 printf("Prueba %d\n", i);
++i;
if (i < 3)
 printf("Prueba %d\n", i);
++i;
if (i < 3)
 printf("Prueba %d\n", i);
++i;
if (i < 3) ;
```

Notemos que sólo en el último if el predicado  $i < 3$  es falso, por lo que la impresión siempre se ejecuta y, en este caso particular, el if completo se considera un paso de ejecución<sup>5</sup>. Puesto que fue posible “desenrollar” el bucle en 11 instrucciones de duración constante, el bucle en sí es de duración constante. Por lo tanto, el bucle conforma un paso de ejecución y su tiempo de duración es  $\mathcal{O}(1)$ .

---

<sup>5</sup>Esa es la razón por la cual la instrucción dentro del if no se coloca indentada.

En el dominio  $\mathcal{O}$ , el arte de analizar un algoritmo consiste en identificar secuencias de ejecución que comporten funciones de duración diferentes y aplicar reglas de conteo; sobre todo las de la suma y del producto.

En el sentido de lo recién explicado son aplicables las siguientes reglas para los clásicos bloques de programación procedural:

1. **InSTRUCCIONES SECUENCIALES:** la ejecución de dos instrucciones secuenciales con tiempo  $T_1(n) = \mathcal{O}(f_1(n))$ ,  $T_2(n) = \mathcal{O}(f_2(n))$  toman la suma  $T_1(n) + T_2(n)$ , la cual, por la regla (3.6) (de la suma), toma  $\mathcal{O}(\max(f_1(n), f_2(n)))$ .

2. **InSTRUCCIONES DE DECISIÓN:** asumamos como caso general la siguiente estructura:

```
if ($\mathcal{O}(f_1(n))$)
 $\mathcal{O}(f_2(n))$
else
 $\mathcal{O}(f_3(n))$
```

En este caso, el tiempo de ejecución estaría dado por  $\mathcal{O}(f_1(n)) + \max(\mathcal{O}(f_2(n)), \mathcal{O}(f_3(n)))$ . Aquí estamos considerando lo peor; es decir, la ejecución del bloque más costoso. Eventualmente, según la distribución de certitud del predicado del if, puede considerarse la esperanza.

3. **InSTRUCCIONES DE REPETICIÓN:** la ejecución de una repetición con la siguiente estructura:

```
for ($\mathcal{O}(f_1(n))$)
 $\mathcal{O}(f_2(n))$
```

se rige por la regla del producto. El tiempo de ejecución está, pues, determinado por  $\mathcal{O}(f_1(n))$  veces la ejecución del bloque interno de coste  $\mathcal{O}(f_2(n))$ , es decir,  $\mathcal{O}(f_1(n)) \times \mathcal{O}(f_2(n)) = \mathcal{O}(f_1(n) \times f_2(n))$ .

Para analizar una repetición anidada deben estudiarse sus dependencias con los ciclos externos. Existen diversas formas, las cuales no necesariamente corresponden con la estructura anterior. En líneas generales, el “truco” para analizar una repetición consiste en averiguar la cantidad de veces que ésta se efectúa.

Por lo general, el análisis de una repetición se hace desde el bucle más interno hasta el más externo.

La duración de un algoritmo se remite a identificar sus secuencias de ejecución y “desenrollarlas” en una sola. Secuencias de longitud fija, es decir, pasos de ejecución, son  $\mathcal{O}(1)$ . Secuencias de longitud proporcionalmente variable son  $\mathcal{O}(n)$ . Secuencias de secuencias de longitud proporcionalmente variable son  $\mathcal{O}(n^2)$ . Secuencias de longitud cuadráticamente variable son  $\mathcal{O}(n^3)$ . Siguiendo esta secuencia de razonamiento no es difícil intuir que el conjunto de posibles funciones descriptoras del tiempo de ejecución de un algoritmo está circunscrito a una pequeña familia, cuyas curvas se ilustran en la figura 3.4.

En las subsecciones subsiguientes estudiaremos una clase de algoritmo denominada “dividir/combinar”, la cual divide la entrada en dos partes aproximadamente iguales. Dentro de esta clase de algoritmo, aquellos que trabajan sobre sólo una de las partes divididas

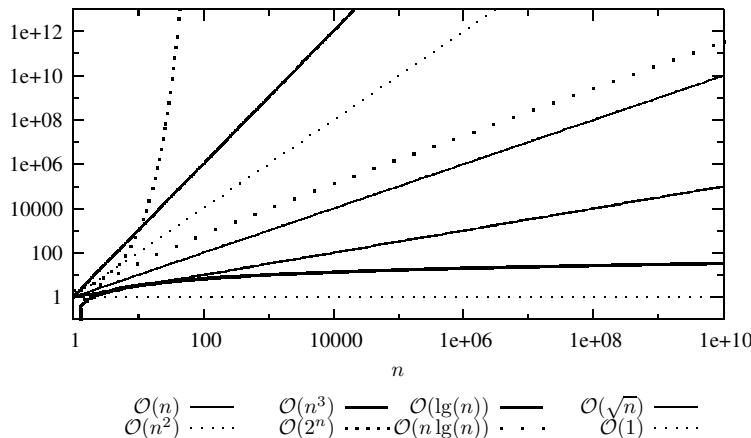


Figura 3.4: Curvas de  $\lg(n)$ ,  $\sqrt{n}$ ,  $n$ ,  $n \lg(n)$ ,  $n^2$ ,  $n^3$ ,  $2^n$  a escalas logarítmicas

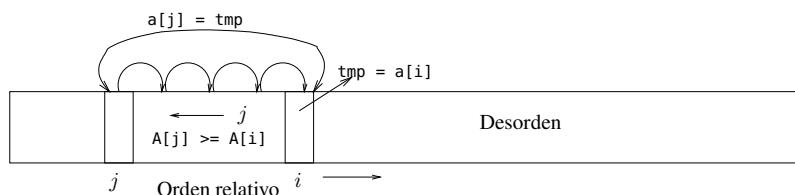
dejando la otra sin procesar son  $\mathcal{O}(\lg(n))$ , mientras que aquellos que requieren procesar lineal y enteramente las dos partes son  $\mathcal{O}(n \lg(n))$ .

Existe otra clase de problema cuya solución, por lo general muy simple, consiste en calcular y analizar todas las permutaciones de secuencias solución posibles. Cuando los elementos no están repetidos, el número de permutaciones de una secuencia es  $n!$ . Cuando, por el contrario, lo están, es  $m^n$ , donde  $m$  es la cantidad de elementos y  $n$  la longitud de la secuencia. Ambas situaciones están acotadas asintóticamente por la función  $f(n) = 2^n$ , la cual no es polinómica. Por esa razón, algoritmos cuyo tiempo de ejecución es  $\mathcal{O}(2^n)$  son denominados no-polinomiales o, acrónimamente, NP.

No se sabe si para los problemas NP existen algoritmos correctos con soluciones polinómicas. En todo caso, hasta que no se demuestre lo contrario, los problemas NP son “intratables” en el sentido de que, aun para una pequeña escala, no existen recursos computacionales que permitan ejecutar el algoritmo. Por esa razón, los problemas NP son tildados de intratables o hercúleos.

### 3.1.8 Ordenamiento por inserción

Pongamos en práctica la teoría de la notación  $\mathcal{O}$  mediante el diseño y análisis de desempeño de otro método de ordenamiento, denominado “de inserción”, cuyo esquema general se bosqueja del siguiente modo:



Primero, el elemento en la entrada  $a[i]$  se guarda en  $tmp$ . Hacia la izquierda del índice  $i$  tenemos orden relativo, es decir, los elementos están ordenados mas no necesariamente en su posición definitiva. Luego, el índice  $j$  retrocede a partir de  $i - 1$  hasta encontrar un elemento menor o igual que  $tmp$ ; en el interín de la iteración, cada elemento se va copiando hacia la derecha de manera tal de ir dejando una brecha. Cuando  $j$  cesa de iterar,  $a[j]$  tiene una brecha donde colocamos el valor de  $a[i]$  memorizado en  $tmp$ .

Una implantación es como sigue:

163a  $\langle$ Métodos de ordenamiento 151a $\rangle + \equiv$   $\triangleleft$ 156c 164a $\triangleright$   
 template <typename T, class Compare = Aleph::less<T> > inline  
 void insertion\_sort(T \* a, const size\_t & l, const size\_t & r)  
 {  
 for (int i = l, j; i <= r; ++i)  
 {  
 T tmp = a[i]; // memorice a[i], pues será sobre escrito  
*(Abra una brecha a[j] donde insertar tmp 163b)*  
 a[j] = tmp; // inserte tmp en la brecha  
 }  
 }

A diferencia del método anterior, la interfaz de `insertion_sort()` toma los índices izquierdo y derecho entre los cuales se desea ordenar el arreglo. Esta interfaz será útil para combinar este método con otros más sofisticados. En lo que sigue, asumiremos  $l = 0$  y  $r = n - 1$ .

163b  $\langle$ Abra una brecha a[j] donde insertar tmp 163b $\rangle \equiv$  (163a)  
 for (j = i; j > l and Compare() (tmp, a[j - 1]); -j)  
 a[j] = a[j - 1]; // desplazar hacia la derecha

El ciclo externo efectúa  $r - l + 1 = n$  iteraciones. Por lo tanto, éste es  $\mathcal{O}(n)$ .

Para estudiar el ciclo interno de  $\langle$ Abra una brecha a[j] donde insertar tmp 163b $\rangle$  requerimos conocer el número de veces que éste itera. Esta cantidad depende del valor del predicado  $j > l$  and `Compare() (tmp, a[j - 1])`, cuyo resultado depende de la permutación de los elementos en el intervalo  $[l, i - 1]$ . En este sentido debemos considerar varias posibilidades:

- Si el intervalo  $[l, i - 1]$  está ordenado, entonces no se efectúa ninguna repetición y el ciclo es  $\mathcal{O}(1)$ . En este caso, el método es  $\mathcal{O}(n) \times \mathcal{O}(1) = \mathcal{O}(n)$ .
- Si el intervalo  $[l, i - 1]$  está ordenadamente invertido, es decir, sus elementos están ordenados desde el mayor hasta el menor, entonces el ciclo itera  $i - l = i$  veces, lo que nos da  $\mathcal{O}(i)$ . Puesto que  $i$  depende de  $n$ , podemos decir directamente que el ciclo es  $\mathcal{O}(n)$ . Consecuentemente, el método es  $\mathcal{O}(n) \times \mathcal{O}(n) = \mathcal{O}(n^2)$ .

Las situaciones planteadas consideran dos escenarios de ejecución: el mejor caso, cuando el arreglo está ordenado, y el peor, cuando el arreglo está completamente invertido. Si la disposición del arreglo es aleatoria, entonces, cada uno de los escenarios tiene una probabilidad de ocurrencia de  $\frac{1}{n!}$ ; una muy baja probabilidad cuando  $n$  es grande. Así pues, en un caso o en el otro, el análisis peca de exageradamente optimista o pesimista, con el consecuente peligro de ser irrealista al momento de expresar el desempeño del método. Para predecir lo que sucederá debemos considerar el caso esperado.

Si consideramos a la secuencia como una permutación aleatoria, entonces el valor del índice de brecha  $j$  tiene una probabilidad uniforme entre  $l$  e  $i - 1$ . El número esperado de iteraciones del ciclo  $\langle$ Abra una brecha a[j] donde insertar tmp 163b $\rangle$  será la esperanza de una distribución uniforme discreta entre  $l$  e  $i - 1$  y el ciclo repetirá  $\frac{r-l+1}{2}$  veces. Esto es  $\mathcal{O}(n)$ , lo que nos da como conclusión de desempeño del método  $\mathcal{O}(n) \times \mathcal{O}(n) = \mathcal{O}(n^2)$ ; el mismo resultado del peor caso.

Comparemos el método de inserción con el de selección. ¿Cuál es el mejor? Si examinamos ligeramente el resultado, entonces podemos concluir, erróneamente, que son equivalentes -que lo son, asintóticamente-. Empero, el lazo interno del método de selección (*Sea min el índice menor entre  $i + 1$  y  $n - 1$*  151b)) siempre itera  $n - i + 1$  veces, independientemente de la permutación  $a[i+1 \dots n-1]$ ; mientras que la cantidad de iteraciones del lazo interno del de inserción (*Abra una brecha  $a[j]$  donde insertar tmp* 163b)) depende de la permutación  $a[1 \dots i-1]$ <sup>6</sup>. Para aprehender este hecho, es conveniente profundizar el análisis del método de inserción para el caso promedio. Si aproximamos el número de repeticiones del lazo interno a  $\frac{i}{2}$ , entonces tenemos como desempeño:

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4};$$

que es equivalente a la mitad de tiempo del método de selección. Si bien las curvas de ambos métodos son cuadráticas, la del de inserción es inferior a la del de selección.

Por su simplicidad de implantación, su bajo coste constante y su buen desempeño para permutaciones quasi-ordenadas ( $\mathcal{O}(n)$ ), el método de inserción es una excelente alternativa para ordenar secuencias pequeñas.

La versión para listas es más sencilla, pues no se requieren manejar índices; tan solo recorrer la lista e ir insertando ordenadamente en otra lista cada nodo observado. Esta actividad justifica la presencia de la primitiva siguiente:

164a *(Métodos de ordenamiento 151a) +≡* △163a 164b ▷  
 template <class Compare> inline void insert\_sorted(Dlink & list, Dlink \* p)

```
{
 typename Dlink::Iterator it(list);
 while (it.has_current() and Compare () (it.get_current(), p))
 it.next();
 if (it.has_current())
 it.get_current()->append(p); // insertar antes de current
 else
 list.append(p);
}
```

Defines:

`insert_sorted`, used in chunk 164b.

`insert_sorted()` inserta el nodo `p` en la lista ordenada `list` según el criterio de comparación `Compare`. Con esta primitiva, el ordenamiento por inserción queda concisamente definido:

164b *(Métodos de ordenamiento 151a) +≡* △164a 165a ▷

```
template <class Compare> inline void insertion_sort(Dlink & list)
{
 Dlink aux;
 while (not list.is_empty())
 insert_sorted<Compare>(aux, list.remove_next());
 list.swap(&aux);
}
```

Uses `insert_sorted` 164a.

---

<sup>6</sup>Note que tiene sentido comparar  $a[i+1 \dots r]$  con  $a[1 \dots i-1]$  en cada uno de los métodos. En efecto, por cada permutación  $a[i+1 \dots r]$ , existe otra complementaria  $a[1 \dots i-1]$  que tiene la misma probabilidad.

`insert_sorted()` ejecuta una sola iteración de secuencias constantes. La duración de la iteración dependerá de la permutación de la secuencia `list` y del valor almacenado en el nodo `p`. Esta es la misma situación que con la versión para arreglos. El tiempo de `insert_sorted()` es entonces  $\mathcal{O}(n)$  para el peor caso y para el esperado. La inserción es llamada desde otro bloque iterativo cuyo tiempo es, con certitud,  $\mathcal{O}(n)$ , pues hay que recorrer toda la lista. Así pues, como es de esperarse, el método es  $\mathcal{O}(n^2)$ .

Nos resta exportar especializaciones por omisión para el tipo `Dnode<T>`:

165a *(Métodos de ordenamiento 151a)≡* ◁164b 169▷  
 template <typename T, class Compare = Aleph::less<T> >  
 inline void insertion\_sort(Dnode<T> & list)  
 {  
 insertion\_sort < Compare\_Dnode<T, Compare> > (list);  
 }

Uses `Compare_Dnode` 153a and `Dnode` 83a.

### 3.1.9 Búsqueda binaria

El análisis del método de inserción o de selección a través de operaciones  $\mathcal{O}$  es simple y directo. En muchas ocasiones, sobre todo en algoritmos recursivos, se debe plantear una ecuación de recurrencia. Ilustremos esta idea mediante la versión recursiva de la búsqueda binaria sobre un arreglo ordenado:

165b *(Métodos de búsqueda 165b)≡*  
 template <typename T, class Compare = Aleph::less<T> > inline  
 int binary\_search\_rec(T \* a, const T & x, const int & l, const int & r)  
 {  
 const int m = (l + r) / 2;  
 if (l > r)  
 return m;  
  
 if (Compare() (x, a[m]))  
 return binary\_search\_rec<T, Compare>(a, l, m - 1);  
 else if (Compare() (a[m], x))  
 return binary\_search\_rec<T, Compare>(a, m + 1, r);  
  
 return m; // encontrado  
 }

El algoritmo busca recursivamente una ocurrencia del elemento `x` en un arreglo que está ordenado entre `l` y `r`. Si `l > r`, entonces el rango de búsqueda está vacío y tenemos la certitud de que el elemento `x` no se encuentra dentro del rango. De lo contrario inspeccionamos el elemento situado en el centro; si éste contiene el valor `x`, entonces `m` es el resultado. Si no, decidimos, según el orden de `x`, en cuál de los dos subrangos debemos buscar: el izquierdo o el derecho.

Asumamos que el rango  $[l, r]$  es de cardinalidad  $n$ .

Cuando  $n \leq 0$  el algoritmo termina en tiempo constante  $\mathcal{O}(1)$ . Si, por el contrario, existen elementos dentro del rango, entonces pueden ocurrir dos cosas: o el elemento se encuentra en el centro, o hay que buscar recursivamente en alguna de las particiones. Puesto que la partición ocurre exactamente por el centro, el resto del tiempo se consume

sobre aproximadamente la mitad de la entrada. Planteamos, pues, la siguiente ecuación recurrente:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{si } n \leq 0 \\ T(n/2) + \mathcal{O}(1) & \text{si } n \geq 1 \end{cases}. \quad (3.14)$$

Notemos que existe un ligero error en la recurrencia: la llamada recursiva ocurre sobre  $T((n - 1)/2)$  y no sobre  $T(n/2)$ , como se expresa en (3.14). Este error, que facilita manipular  $T(n)$ , es despreciable, pues un paso de ejecución de menos no alterará el comportamiento asintótico de  $T(n)$ .

Efectuemos la transformación  $n = 2^k \implies k = \lg(n)$ ; esto asume que  $n$  es una potencia exacta de 2, lo cual, si muchas veces no será el caso, sí es válido en el dominio de la notación  $\mathcal{O}$ . La recurrencia (3.14) se expresa entonces:

$$T(2^k) = \begin{cases} \mathcal{O}(1) & \text{si } k = 0 \\ T(2^{k-1}) + 1 & \text{si } k > 0 \end{cases}. \quad (3.15)$$

Lo interesante de la transformación es que expresa directamente la recurrencia como una sumatoria. En efecto, expandiéndola tenemos:

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + 1 = T(2^{k-2}) + 1 + 1 \\ &= T(2^0) + 1 + \dots + 1 + 1 = k + 1 \end{aligned} \quad (3.16)$$

Ahora hacemos la transformación inversa:

$$T(n) = \lg(n) + 1, \quad (3.17)$$

la cual, claramente, es  $\mathcal{O}(\lg(n))$ ; una eficiencia espectacular. La búsqueda binaria es, en el peor caso,  $\mathcal{O}(\lg(n))$ . Se requiere duplicar la entrada para que el algoritmo demore una unidad de tiempo de más.

### 3.1.10 Errores de la notación $\mathcal{O}$

Hay dos bondades bastante notables de la notación  $\mathcal{O}$ . En primer lugar, ella objetiza la imprecisión. Podemos interpretar que las constantes de la definición  $c$  y  $n_1$  absorben las idiosincrasias subjetivas que ya hemos señalado (tipo computador, ...). En segunda instancia, la notación permite estudiar funciones  $T(n)$  del tiempo de un algoritmo bajo un método considerablemente más sencillo que el conteo y el álgebra tradicional.

Pero toda objetivización acarrea un precio, por cierto bastante objetivo y general a toda objetivización, cual es el ocultamiento de aquello subjetivo que no sólo en muchos casos puede ser importantísimo, sino serle esencialísimo, pues se arriesga ese bien supremo llamado verdad.

Después de analizar el método de inserción y compararlo con el de selección, podemos resaltar que la notación  $\mathcal{O}$  sólo arroja la forma de  $T(n)$  para entradas muy grandes. De hecho, la imprecisión de la notación  $\mathcal{O}$  desecha, entre otros aspectos, los que se resumen en las siguientes observaciones:

1. No se consideran los costes constantes del algoritmo: la propia definición de  $\mathcal{O}$  descarta las operaciones que toman tiempo constante en  $\mathcal{O}(1)$ .

2. Los costes variables, pero inferiores a la curva dominante, tampoco se consideran. Por ejemplo, supongamos un algoritmo cuadrático con una fase inicial de preprocesamiento  $\mathcal{O}(n)$ , luego, una fase  $\mathcal{O}(n^2)$  donde se ejecuta el algoritmo en sí y, finalmente, una fase final de postprocesamiento  $\mathcal{O}(n)$ . En este caso, el tiempo de ejecución estará dado por  $\mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$ . Ahora bien, el resultado final  $\mathcal{O}(n^2)$  oculta completamente dos costes considerables asociados al algoritmo: el pre y el postprocesamiento, los cuales, aun cuando son lineales, pueden ser costosísimos si llevan una constante muy alta.
3. El carácter real de  $T(n)$  se esconde totalmente por la acotación  $\mathcal{O}(f(n))$  pues, en la mayoría de las veces,  $T(n) \neq f(n)$ : en este sentido cabe señalar que  $\mathcal{O}(f(n))$  sólo tiene sentido para entradas mayores que la constante  $n_1$  planteada en la definición 3.1. En otras palabras, afirmar que  $T(n) = \mathcal{O}(f(n))$  desconoce el comportamiento de  $T(n)$  antes de  $n_1$ .

La gráfica 3.5 señala, cartesianamente hablando, el dominio donde la notación  $\mathcal{O}$  es con, certitud, verdad: La notación  $\mathcal{O}$  es certamente precisa en el dominio de lo infinito

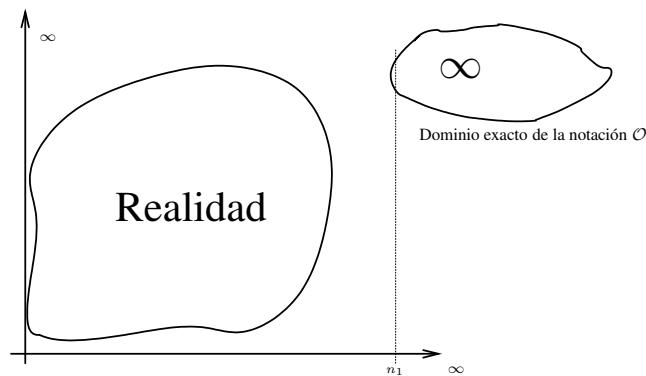


Figura 3.5: Ilustración gráfica del dominio de la notación  $\mathcal{O}$

(o de la eternidad), pero este dominio puede ser muy abstracto y lejano de alguna realidad concreta. Por eso, como en todo lo objetivo, hay que andar con sumo cuidado.

Costes muy grandes, inabordables, pero constantes, pueden quedar completamente ocultos por un análisis  $\mathcal{O}$ . Lo mismo puede suceder con otros costes asintóticamente inferiores pero constantemente superiores. Por añadidura, el ocultamiento total de la forma de  $T(n)$  puede acarrear sorpresas si la ejecución sólo tiene sentido para escalas inferiores a la cota asintótica  $n_1$  de la definición 3.1. Quizá lo peor, políticamente hablando, es que una vez publicado un análisis  $\mathcal{O}$ , el conocimiento de aquellos costes ya señalados tienda a sepultarse para siempre.

### 3.1.11 Tipos de análisis

Como ya reiterativamente hemos dicho, hay algoritmos que no sólo dependen del tamaño de la entrada, sino también de la forma en que se presente la entrada. El método de inserción ofrece un buen ejemplo.

Por lo general, cuando se analiza un algoritmo se consideran dos perspectivas. La primera consiste en analizarlo para la entrada que cause la peor ejecución; la segunda para la entrada esperada.

Casi siempre se requiere el análisis para la peor entrada, pues éste plantea una cota de ejecución. El peor caso nos ofrece una especie de garantía acerca del tiempo de ejecución, pues cualquier otra entrada diferente a la peor será mejor. Esta garantía es apreciable cuando se requieren algoritmos con requerimientos de ejecución duros que no se pueden flexibilizar, y en los cuales no se puede tener el lujo de tener una mala ejecución.

El análisis para la entrada esperada plantea una predicción, una expectativa acerca del tiempo de ejecución más probable. Se trata de una esperanza, es decir, de lo que se espera que suceda muchas veces, pero no siempre. La proporción de veces que el algoritmo se comportará como lo esperado dependerá de la varianza de la entrada. Básicamente, este tipo de análisis se puede plantear en dos situaciones. La primera cuando la entrada esperada sea bastante probable, es decir, cuando la varianza de la entrada sea pequeña. La segunda situación cuando se plantee como requerimiento satisfacer al promedio de casos sin importar algunas malas ejecuciones causadas por una mala entrada. En ambas situaciones es importante conocer la varianza.

### 3.2 Algoritmos dividir/combinar

Una técnica elegante para resolver un problema consiste en dividirlo en dos problemas equitativos, cuyas complejidades, a menudo expresadas en función de la escala, son menores que las del problema inicial. El proceso de división se repite recurrentemente con cada una de las partes hasta alcanzar complejidades particulares cuya solución sea conocida. A partir de este momento se combina un par de soluciones para encontrar la solución al problema general primigeniamente dividido. A la combinación de dos soluciones para obtener la solución completa se le denomina “combinación” o “conquista”.

Para aplicar esta técnica se requiere (1) saber dividir el problema y (2) saber combinar. Esta es la parte subjetivísima en el diseño de un algoritmo dividir/combinar. Por contraste, la estructura y análisis de esta clase de algoritmos tiene un esquema muy objetivo.

Dado un problema  $P$  con entrada de tamaño  $n$ , un algoritmo dividir/combinar, cuya duración se describe por la función  $T(n)$  y que resuelve a  $P$  se estructura, de manera general y objetiva, en las partes siguientes:

1. **Identificación de un problema particular y su solución:** esta parte identifica una solución particular del problema para un tamaño particular de entrada  $n < n_b$ . En esta fase se requiere conocer cómo resolver el problema para la ocurrencia de entrada particular.

A esta parte se le llama “base”.

Asumamos que la solución base cuesta  $T(n_b)$ . Puesto que  $n_b$  es constante, lo más probable es que  $T(n_b)$  sea constante, por lo que, la mayoría de las veces, el coste de  $T(n_b) = \mathcal{O}(1)$ ,  $n \leq n_b$ .

2. **División:** en esta parte se divide el problema en dos particiones equitativas de tamaños  $n_1 \approx n_2 \approx \frac{n}{2}$ .

Asumamos que la división cuesta  $\mathcal{O}(f_d(n))$ .

3. Recursión: esta parte consiste en resolver recursivamente cada partición y obtener soluciones  $s_1(n_1)$  y  $s_2(n_2)$ .

Puesto que ocurren llamadas recursivas, el coste es desconocido hasta tanto no se conozca el coste del caso base y de la combinación. Empero, a menudo el coste puede expresarse recursivamente como  $T(n_1) + T(n_2)$ , pues las llamadas recursivas se corresponden con la misma función de duración, pero para entradas menores.

4. Combinación: en esta fase se combinan las soluciones  $s_1(n_1)$  y  $s_2(n_2)$  resultantes de las llamadas recursivas en una solución definitiva al problema.

Asumamos que la combinación cuesta  $\mathcal{O}(f_c(n))$ .

Según lo expresado, la duración de un algoritmo dividir/combinar se formula mediante la siguiente recurrencia:

$$T(n) = \begin{cases} \mathcal{O}(f_b(n)) & , n \leq n_b \\ \underbrace{\mathcal{O}(f_d(n))}_{\text{división}} + \underbrace{T(n_1)}_{\text{primera partición}} + \underbrace{T(n_2)}_{\text{segunda partición}} + \underbrace{\mathcal{O}(f_c(n))}_{\text{combinación}} & , n > n_b \end{cases} \quad (3.18)$$

Muy a menudo, la estructura de un algoritmo dividir/combinar se presta fácilmente a su manejo concurrente, es decir, las particiones pueden tratarse por hilos o procesos de ejecución distintos. Si estos hilos son ejecutados por procesadores materiales distintos, entonces los sub-problemas pueden resolverse simultáneamente y, de este modo, más rápidamente que un solo procesador. La velocidad, por supuesto, depende de la cantidad de procesadores que se disponga, de la índole del algoritmo a paralelizar y de las virtudes del diseñador o programador que haga la paralelización.

### 3.2.1 Ordenamiento por mezcla

Ejemplifiquemos el diseño y análisis de un algoritmo dividir/combinar mediante un algoritmo de ordenamiento denominado de “mezcla”, el cual se describe a continuación:

169

*(Métodos de ordenamiento 151a) +≡  
(mezcla de arreglos 171)*

◀165a 172▶

```
template <typename T, class Compare = Aleph::less<T> > inline
void mergesort(T * a, const int & l, const int & r)
{
 if (l >= r)
 return;

 const int m = (l + r)/2;

 mergesort<T, Compare>(a, l, m);
 mergesort<T, Compare>(a, m + 1, r);

 merge<T, Compare>(a, l, m, r);
}
```

Uses merge 171 and mergesort 173a.

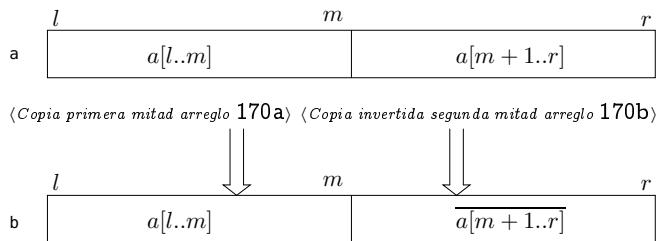
El algoritmo se corresponde exactamente con el patrón dividir/combinar. La fase de combinación la efectúa el procedimiento `merge()`, el cual se encarga de “mezclar” dos particiones ordenadas. Antes de indagar el tiempo de ejecución requerimos conocer cómo se efectúa la mezcla y analizar su desempeño.

### 3.2.1.1 Mezcla (merge)

La mezcla asume dos secuencias ordenadas, en los intervalos  $[l, m]$  y  $[m + 1, r]$ . Si tuviésemos dos arreglos ordenados, entonces podríamos mezclarlos en un tercero por barrido. A cada iteración, comparamos el elemento actual de cada arreglo, copiamos el menor de los dos al arreglo resultado y avanzamos el elemento actual en el arreglo que contenía el menor elemento.

En nuestra situación tenemos un arreglo partitionado en dos partes equitativas que ya están ordenadas. El proceso de mezcla copiará el arreglo  $a[]$  hacia uno auxiliar que llamaremos  $b[]$ . No es posible evadir el uso de un arreglo adicional.

Con miras a simplificar la mezcla haremos la copia del arreglo  $a$  hacia el segundo arreglo  $b$  según el siguiente esquema:

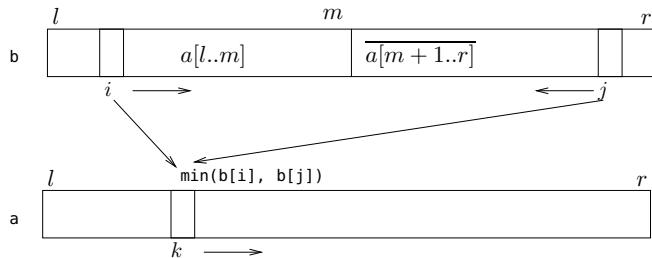


Es decir, entre  $0$  y  $m$  copiamos el arreglo normalmente, mientras que entre  $m + 1$  y  $r$  lo invertimos. De esta forma, cuando se efectúe la mezcla, los mayores elementos fungirán de centinelas naturales que nos ahorrarán una segunda repetición junto con un *if* que decida sobre cuál de las dos mitades hay que realizar el resto de la copia.

170a  $\langle \text{Copia primera mitad arreglo 170a} \rangle \equiv$  (171)  
 $\text{for } (i = 1; i \leq m; i++)$   
 $\quad b[i] = a[i];$

170b  $\langle \text{Copia invertida segunda mitad arreglo 170b} \rangle \equiv$  (171)  
 $\text{for } (j = r, i = m + 1; i \leq r; i++, j-)$   
 $\quad b[i] = a[j];$

Una vez copiadas las particiones en el arreglo  $b$ , procedemos a “mezclarlas” en el arreglo  $a$  como sigue:



Lo cual se implanta del siguiente modo:

170c  $\langle \text{mezclar mitades 170c} \rangle \equiv$  (171)  
 $\text{for } (k = 1, i = l, j = r; k \leq r; k++)$   
 $\quad \text{if } (\text{Compare}() \ (b[i] < b[j]))$   
 $\quad \quad a[k] = b[i++];$   
 $\quad \text{else}$   
 $\quad \quad a[k] = b[j-];$

y que completa todo lo necesario para programar la rutina `merge()`:

171    *mezcla de arreglos 171*≡ (169)

```
template <typename T, class Compare = Aleph::less<T> > inline
void merge(T * a, const int & l, const int & m, const int & r)
{
 int i, j, k;
 T b[r - l + 1];
 <Copia primera mitad arreglo 170a>
 <Copia invertida segunda mitad arreglo 170b>
 <mezclar mitades 170c>
}
```

Defines:

`merge`, used in chunk 169.

El parámetro `m` es el centro de la partición.

### 3.2.1.2 Análisis del mergesort

Para analizar el `mergesort()` usamos la ecuación (3.18) y planteamos:

$$T(n) = \begin{cases} \mathcal{O}(1) & , n \leq 1 \\ 2T(n/2) + \mathcal{O}(f_c(n)) & , n > 1 \end{cases} . \quad (3.19)$$

Donde  $\mathcal{O}(f_c(n))$  es la complejidad de tiempo de la fase de combinación, o sea, de la función `merge()`. Claramente, los bloques *<Copia primera mitad arreglo 170a>* y *<Copia invertida segunda mitad arreglo 170b>* consumen  $n/2$  pasos de ejecución, lo que los hace  $\mathcal{O}(n)$ . El bloque *<mezclar mitades 170c>* consume exactamente  $n$  unidades de ejecución, lo que lo hace también  $\mathcal{O}(n)$ . Por tanto, `merge()` consume  $\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$ . De este modo, (3.19) se plantea como:

$$T(n) = \begin{cases} \mathcal{O}(1) & , n \leq 1 \\ 2T(n/2) + \mathcal{O}(n) & , n > 1 \end{cases} . \quad (3.20)$$

Esta recurrencia también puede resolverse asumiendo  $n = 2^k \implies k = \lg(n)$ , o sea, asumimos que  $n$  es una potencia exacta de 2. Para  $n > 1$ , (3.20) se plantea como:

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + \mathcal{O}(2^k) \implies (\text{dividimos toda la ecuación entre } 2^k) \\ \frac{T(2^k)}{2^k} &= \frac{T(2^{k-1})}{2^{k-1}} + \mathcal{O}(1) \\ &= \frac{T(2^{k-2})}{2^{k-2}} + \mathcal{O}(1) + \mathcal{O}(1) = \underbrace{\mathcal{O}(1) + \cdots + \mathcal{O}(1)}_{k \text{ veces}} + \mathcal{O}(1) \\ &= \mathcal{O}(k) + 1 \end{aligned} \quad (3.21)$$

Al regresar al plano  $n$  tenemos que (3.21) se plantea como:

$$\begin{aligned} \frac{T(n)}{n} &= \mathcal{O}(\lg(n)) + \mathcal{O}(1) \implies \\ T(n) &= \mathcal{O}(n \lg(n)) + \mathcal{O}(n) = \mathcal{O}(n \lg(n)) ; \end{aligned} \quad (3.22)$$

para  $n = 2^k$ . El cual está asintóticamente acotado por  $\mathcal{O}(n \lg(n))$ . En el infinito, la multiplicidad de  $n$  no afecta el comportamiento asintótico, pues ésta no afecta la recurrencia.

En otros términos, el que  $n$  sea o no potencia exacta de dos no cambiará el hecho de que  $T(n)$  sea  $\mathcal{O}(n \lg(n))$ .

El método por mezcla es  $\mathcal{O}(n \lg(n))$  para todos los casos: mejor, peor y promedio. Notemos que el algoritmo en sí no distingue la forma de la entrada; siempre cuesta  $\mathcal{O}(n \lg(n))$ , independientemente de la disposición de los elementos en el arreglo.

### 3.2.1.3 Estabilidad del mergesort

Dada una tabla de registros, es decir, un arreglo con columnas de tipos iguales o diferentes, una “clave primaria” es una que no se repite en una fila. Por ejemplo, el número único de identidad nacional no está diseñado para repetirse. Una clave secundaria es una que puede repetirse en dos o más filas; ejemplos, los nombres y apellidos.

Bajo la definición anterior, un método de ordenamiento se califica de “estable” si, al ordenar por clave secundaria una secuencia de registros previamente ordenada por clave primaria, las claves secundarias repetidas se conservan entonces ordenadas según la clave primaria.

En el sentido de lo anterior, el método mergesort es estable.

### 3.2.1.4 Coste en espacio

El consumo de memoria es otro coste que siempre debe considerarse al momento de analizar un algoritmo. En este sentido, el ordenamiento de arreglos por mezcla es un buen ejemplo, pues la rutina `merge()` requiere un arreglo proporcional a  $n$ , lo cual implica que el método en cuestión requiere espacio proporcional a  $\mathcal{O}(n)$ . Así pues, si se desea utilizar este método, entonces debe asegurarse el disponer del espacio adicional requerido para su ejecución.

### 3.2.1.5 Ordenamiento por mezcla de listas enlazadas

El ordenamiento por mezcla fue concebido muy otrora, cuando había muy poca memoria y los medios de almacenamiento secundario eran cintas magnéticas lentísimas. En aquellos tiempos, ordenar requería la “mezcla” de varias cintas.

Aunque hoy en día disponemos de muy extensas memorias y de medios de almacenamiento masivo no secuenciales, el ordenamiento por mezcla no es para nada obsoleto. Todo lo contrario, es un método con garantía  $\mathcal{O}(n \lg(n))$  que permite ordenar efectivamente listas enlazadas y que, por añadidura, no requiere espacio adicional. Tal mezcla se especifica como sigue:

172

*(Métodos de ordenamiento 151a) +≡* ◀169 173a▶

```
template <class Compare> inline
void merge_lists(Dlink & l1, Dlink & l2, Dlink & result)
{
 while (not l1.is_empty() and not l2.is_empty())
 if (Compare () (l1.get_next(), l2.get_next()))
 result.append(l1.remove_next());
 else
 result.append(l2.remove_next());

 if (l1.is_empty())
 result.concat_list(&l2);
```

```

 else
 result.concat_list(&l1);
}

```

En consonancia con su carácter de secuencia, el ordenamiento por mezcla de una lista enlazada tiene exactamente la misma estructura que su versión para arreglos.

173a *(Métodos de ordenamiento 151a) +≡* △172 173b ▽

```

template <class Compare> inline void mergesort(Dlink & list)
{
 if (list.is_unitarian_or_empty())
 return;

 Dlink l, r;
 list.split_list(l, r); // dividir en dos listas

 mergesort <Compare> (l); // ordenar la primera
 mergesort <Compare> (r); // ordenar la segunda

 merge_lists <Compare> (l, r, list); // mezclarlas
}

```

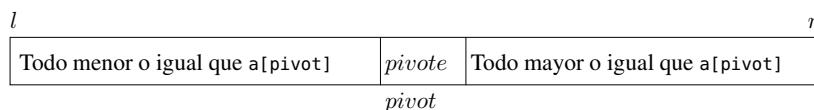
Defines:

`mergesort`, used in chunks 169 and 577b.

La biblioteca define especializaciones para las otras clases de listas mediante la utilización de la clase `Compare_Dnode`.

### 3.2.2 Ordenamiento rápido (Quicksort)

Consideremos un arreglo desordenado `a[]` y la invocación `pivot = partition(a, l, r)`, la cual particiona el arreglo en dos partes generales como se pectoriza en la siguiente figura:



La ejecución `pivot = partition(a, l, r)` efectúa algunas operaciones sobre el arreglo y retorna un índice `pivot` tal que el arreglo satisface la partición según el esquema pictorizado. Es decir, después de invocar `partition()`, el elemento `a[pivot]` se encuentra en su posición definitiva dentro de lo que sería la secuencia ordenada; o sea, la entrada `a[pivot]` corresponde al pivot-ésimo menor elemento del arreglo. El arreglo ha sido dividido, según `a[pivot]`, en dos partes desordenadas, que pueden ordenarse recursiva e independientemente como sigue:

173b *(Métodos de ordenamiento 151a) +≡* △173a 177 ▽

```

template <typename T, class Compare = Aleph::less<T> > inline
void quicksort_rec(T * a, const int & l, const int & r)
{
 if (l >= r)
 return;

 const int pivot = partition <T, Compare> (a, l, r);

```

```
 quicksort_rec <T, Compare> (a, l, pivot - 1);
 quicksort_rec <T, Compare> (a, pivot + 1, r);
}
```

Este procedimiento es el mejor método de ordenamiento hasta hoy conocido. “Su rendimiento es tan espectacular que su descubridor<sup>7</sup>, C.A.R. Hoare [71], lo bautizó “quicksort”” [182], o sea, ordenamiento rápido, o, abreviadamente, en castellano, “el rápido”. La esencia del método subyace en la manera de efectuar la partición, pero antes de estudiarla es interesante fundamentar el análisis del algoritmo.

Conviene notar que la estructura dividir/combinar del quicksort difiere ligeramente de su tradicional orden objetivo. Las llamadas recursivas ocurren al final de la rutina sin que se note explícitamente una fase de combinación. De hecho, es en la propia partición del arreglo cuando ocurre la combinación, pues una vez efectuada la partición, cada parte puede ordenarse, independientemente, por separado.

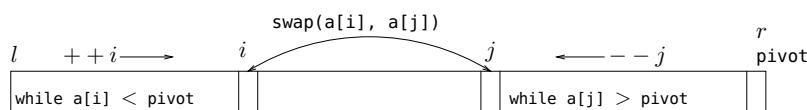
El tiempo de ejecución se caracteriza por la siguiente ecuación recurrente:

$$T(n) = \begin{cases} \mathcal{O}(1) & , n \leq 1 \\ \mathcal{O}(f_p(n)) + T(\text{pivot} - l) + T(r - \text{pivot}) & , n > 1 \end{cases}; \quad (3.23)$$

donde  $\mathcal{O}(f_p(n))$  caracteriza la duración de `partition()`. `pivot` es el índice del elemento en torno al cual se efectúa la partición.

### 3.2.2.1 Partición

La partición es la parte más delicada y complicada del quicksort. Asumamos un elemento “pivot” que representará al elemento de partición y que de alguna forma, que explicaremos más adelante, se ubica en  $a[r]$ . Iniciemos nuestro estudio mediante observación de la siguiente figura:



(avance i hasta primer elemento mayor que pivot 174) (avance i hasta primer elemento menor que pivot 175a)

La figura representa el arreglo entre l y r. i es un índice por el lado izquierdo del arreglo y j es por el lado derecho. i se mueve hacia la derecha, lo que implica que debe incrementarse; mientras que j se mueve hacia la izquierda, por lo que se debe decrementar. Los valores iniciales de estos índices se colocan en una posición previa a su primer acceso; es decir, i en l - 1 y j en r.

Luego de iniciados los índices, la rutina entra en un lazo cuyo cuerpo consiste en buscar una inversión relativa al pivote, es decir, un par de elementos que estén invertidos en orden respecto al pivote. El primero de estos elementos se busca por la izquierda:

*(avance i hasta primer elemento mayor que pivot 174) ≡* (175b)  
*// avance mientras a[i] < a[pivot]*  
*while (Compare() (a[++i], pivot)) { /\* no hay cuerpo \*/ }*

<sup>7</sup>En la cita original Wirth emplea el término “inventor”.

En la instrucción  $a[++i]$ , la carga de  $a[i]$  se efectúa después del incremento. Por tanto, el primer acceso ocurre sobre  $a[1]$ . El desplazamiento de  $i$  jamás sobrepasará a  $r$ , pues el pivote funge de centinela y la comparación es por desigualdad estricta. Por tanto,  $i$  nunca accederá un índice fuera del rango  $[l, r]$ .

La búsqueda del elemento de inversión del lado derecho es similar, pero aquí no tenemos un centinela, por lo que debemos verificar que  $j$  no sobrepase a  $1$ :

175a  $\langle\text{avance } j \text{ hasta primer elemento menor que pivot } 175a\rangle \equiv$  (175b)  
 $\text{while } (\text{Compare}() \text{ (pivot, } a[-j])) // \text{ avance mientras } a[\text{pivot}] < a[j]$   
 $\text{if } (j == 1) // \text{ ¿se alcanzó el borde izquierdo?}$   
 $\text{break;} // \text{ sí } ==> \text{ hay que terminar la iteración}$

Si  $i$  y  $j$  no se cruzan ( $i < j$ ), entonces existe una inversión que se corrige intercambiando  $a[i]$  con  $a[j]$ .

Si, por el contrario, los índices  $i$  y  $j$  se cruzan ( $i \geq j$ ), el proceso de partición culmina y el índice  $i$  constituye el punto de partición. En este momento se intercambia con el pivote  $e$   $i$  divide al arreglo en dos partes; la izquierda contiene claves menores o iguales al pivote, mientras que la derecha contiene claves mayores o iguales.

Es posible que el rango  $[l, i]$  contenga claves iguales al pivote; esto sucede si  $j$  cruza a  $i$ , es decir, si no se encuentra una clave del lado derecho que sea menor que el pivote  $e$   $i$  queda posicionado en una clave igual al pivote.

Con las explicaciones anteriores apropiadas, estamos listos para mostrar la partición:

175b  $\langle\text{Definición de partición de arreglo } 175b\rangle \equiv$   
 $\text{template } <\text{typename T, class Compare}> \text{ inline}$   
 $\text{int partition}(T * a, \text{const int} & l, \text{const int} & r)$   
 $\{$   
 $\quad \langle\text{Seleccionar pivote } 180b\rangle$   
 $\quad T \text{ pivot} = a[r]; // \text{elemento pivot}$   
 $\quad \text{int } i = l - 1, // \text{índice primer elemento a la izquierda } > \text{ que pivot}$   
 $\quad \quad j = r; // \text{índice primer elemento a la derecha } > \text{ que pivot}$   
 $\quad \text{while } (\text{true})$   
 $\quad \{$   
 $\quad \quad \langle\text{avance } i \text{ hasta primer elemento mayor que pivot } 174\rangle$   
 $\quad \quad \langle\text{avance } j \text{ hasta primer elemento menor que pivot } 175a\rangle$   
 $\quad \quad \text{if } (i \geq j)$   
 $\quad \quad \text{break};$   
 $\quad \quad // \text{En este punto hay una inversión } a[i] > a[\text{pivot}] > a[j]$   
 $\quad \quad \text{std::swap}(a[i], a[j]); // \text{Eliminar la inversión}$   
 $\quad \}$   
 $\quad \text{std::swap}(a[i], a[r]);$   
 $\quad \text{return } i;$   
 $\}$

Defines:

`partition`, used in chunks 173b, 177, 180c, 182b, 184, and 186a.

La rutina toma como parámetros el arreglo y los límites izquierdo y derecho, denominados  $l$  y  $r$  respectivamente.

La primera línea del método es un paso fundamental en el desempeño del quicksort y consiste en seleccionar un elemento que funja de particionador del arreglo. Tal ele-

mento se denomina “pivot”. Por ahora, asumamos que seleccionamos el último elemento, es decir,  $a[r]$ .

Existen otras formas de escoger el elemento pivot que son más costosas en tiempo constante, pero que mejoran la esperanza de duración del ordenamiento. Cualquiera que sea el método, nos remitiremos a colocar el pivot en el extremo derecho. De este modo, una modificación del enfoque de selección de pivot no afecta el resto del algoritmo de partición.

### 3.2.2.2 Análisis del quicksort

El análisis del quicksort requiere resolver la ecuación (3.23), la cual, a su vez, requiere del análisis de la partición.

Cualquiera sea la disposición del arreglo, el while más externo rompe cuando se crucen los índices  $i$  y  $j$ . Para que esto suceda tienen que ocurrir exactamente  $r-l+1$  incrementos y decrementos. Por tanto, si  $n = r - l + 1$ , entonces el tiempo de ejecución es  $\mathcal{O}(n)$ . Sustituyendo este tiempo en la ecuación (3.23) puede aproximarse a:

$$T(n) = \begin{cases} \mathcal{O}(1) & , n \leq 1 \\ \mathcal{O}(n) + T(pivot-l) + T(r-pivot) & , n > 1 \end{cases} . \quad (3.24)$$

Si suponemos una permutación aleatoria, entonces la esperanza sobre el punto de partición se sitúa en  $\frac{l+r}{2}$ , o sea, en el centro del arreglo, por lo que la ecuación (3.24) puede aproximarse como:

$$T(n) = \begin{cases} \mathcal{O}(1) & , n \leq 1 \\ \mathcal{O}(n) + 2T(n/2) & , n > 1 \end{cases} ; \quad (3.25)$$

la cual es  $\mathcal{O}(n \lg(n))$ .

El desempeño del quicksort es  $\mathcal{O}(n \lg(n))$  para el caso esperado y para el mejor caso.

El peor caso puede ocurrir cuando el pivot sea justamente el mayor de todos los elementos entre  $i$  y  $j$ . En esta desafortunada situación, la partición izquierda es de longitud  $r - l - 1$ , mientras que la de la derecha es 0. En otras palabras, no se logra dividir efectivamente. La ecuación (3.24) se expresa como:

$$T(n) = \begin{cases} \mathcal{O}(1) & , n \leq 1 \\ \mathcal{O}(n) + T(n-1) & , n > 1 \end{cases} ; \quad (3.26)$$

la cual es  $\mathcal{O}(n^2)$ . El quicksort deviene, en el peor caso, “el lento” [182]. ¿Cuán probable es que acaezca lo peor? Al igual que en la vida, muy poco. Para intuir este hecho, planteemos algunos “muy malos casos”.

Un muy mal caso está dado, paradójicamente, cuando el arreglo está ordenado. En esta situación, el pivot siempre es el mayor elemento, el cual ya se encuentra en su posición definitiva; la partición izquierda es de longitud  $n - 1$  mientras que la derecha es nula. La probabilidad de este infortunio es  $\frac{1}{n!}$ , pues sólo existe una posibilidad entre las  $n!$  posibles.

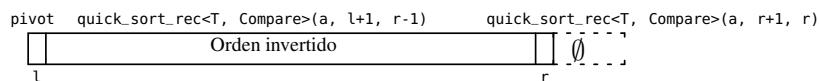
Un caso ligeramente peor que el anterior se da cuando el arreglo está ordenadamente invertido. Esta disposición causa una partición izquierda nula y una derecha de longitud  $n - 1$ . La probabilidad es idéntica al caso anterior:  $\frac{1}{n!}$ .

¿Qué tan probable es tener un mal caso? En estadística, esta respuesta la da la varianza. Si la permutación es aleatoria, entonces la partición variaría según una distribución uniforme, es decir,  $\frac{(r-l+1)^2-1}{2}$ , por lo que el punto de partición se desviaría dentro del rango definido por  $\sqrt{\frac{(n)^2-1}{12}}$ , lo cual arroja un error esperado de 29% aproximadamente. En otros términos, una especie de mal caso esperado de partición es que ésta sea de proporción 1/3; proporción que aún divide el arreglo de manera efectiva. Además, esta “desventura” tendría que ocurrir con la mayoría de las particiones; cuestión altamente improbable si las permutaciones se distribuyen según una densidad uniforme.

### 3.2.2.3 Análisis de consumo de espacio del quicksort

Los algoritmos recursivos llevan un coste en espacio debido a la pila. En este sentido, un algoritmo dividir/combinar consume espacio de pila cuando “memoriza” una de las divisiones a efectos de procesar recursivamente la otra. De este modo, conforme aumentan las divisiones, mayor memoria se requiere.

El quicksort puede ser muy costoso en espacio si ocurren malas particiones sucesivas. Aprehendamos esto mirando el peor de todos los malos casos, el cual ocurre cuando el arreglo está inversamente ordenado. En esta situación, según *(Definición de partición de arreglo 175b)*, el arreglo siempre se partitiona en:



Lo cual, según la implantación de `quicksort_rec()`, causa las llamadas recursivas (colocadas con parámetros reales) a `quicksort_rec(a, l, r-1)` y `quicksort_rec(a, r+1, r)`, justo en ese orden respectivo.

Notemos que este orden de llamadas empila la partición más pequeña mientras se efectúa la llamada recursiva sobre la partición más grande cuyo tamaño es  $r - l - 1$ . A su vez, puesto que el resto del arreglo está inversamente ordenado, la siguiente invocación a `quicksort_rec(a, l, r - 1)` causa de nuevo la peor partición, por lo que se vuelve a empilar la partición vacía mientras se llama recursivamente a la partición de tamaño  $r - l - 2$ . Este proceso de malas particiones continúa hasta que se ordene enteramente el arreglo.

El peor caso del quicksort sólo empila particiones vacías. Para precisar el coste en espacio de estas llamas vanas hay que contar la cantidad de veces que se llama recursivamente al quicksort, la cual es, exactamente, el número de elementos  $n$ , o sea,  $\mathcal{O}(n)$ . Un muy mal caso del quicksort puede fácilmente causar un desborde de pila.

Para evitar esto debemos asegurarnos que la primera llamada recursiva se efectúe sobre la partición más pequeña, de modo tal que se empile menos, pues a menor tamaño de partición ocurrirá una menor cantidad de llamadas recursivas. De esta manera, una muy ligera variación de `quicksort_rec()`, que minimiza el consumo de pila, se enuncia como sigue:

*(Métodos de ordenamiento 151a) +≡*

*<173b 180a>*

```
template <typename T, class Compare = Aleph::less<T> > inline
void quicksort_rec_min(T * a, const int & l, const int & r)
{
 if (r <= l)
```

```

 return;

 const int pivot = partition<T, Compare>(a, l, r);

 if (pivot - l < r - pivot) // ¿cuál es la partición más pequeña?
 {
 // partición izquierda más pequeña
 quicksort_rec_min<T, Compare>(a, l, pivot - 1);
 quicksort_rec_min<T, Compare>(a, pivot + 1, r);
 }
 else
 {
 // partición derecha más pequeña
 quicksort_rec_min<T, Compare>(a, pivot + 1, r);
 quicksort_rec_min<T, Compare>(a, l, pivot - 1);
 }
}

```

Uses partition 175b.

Esta versión invoca la recursión desde la menor hasta la mayor partición, lo cual garantiza un consumo de pila mínimo. ¿De cuánto se trata este consumo? Para responder la cuestión, debemos primero identificar cuál es el máximo tamaño que puede tomar una menor partición y asumir que ello ocurre recursivamente en cada llamada recursiva. El máximo de que hablamos se presenta cuando la partición ocurre exactamente por el centro, el cual, no sorprendentemente, es el mejor caso de partición en velocidad de ordenamiento y ocurre, a lo sumo,  $\mathcal{O}(\lg(n))$  veces, que es la mayor proporción de veces que podemos dividir al arreglo. Por tanto, `quicksort_rec_min()` tiene un consumo de espacio  $\mathcal{O}(\lg(n))$  en el peor caso, coste perfectamente manejable aun para pilas pequeñas.

El análisis precedente también nos da cuenta acerca del número de invocaciones recursivas que ocurrirían en el mejor caso del quicksort, es decir, cuando todas las particiones ocurran por el centro. En este caso, cada llamada al quicksort ocasiona dos llamadas recursivas sobre particiones del mismo tamaño. Por tanto, el total de llamadas recursivas que tomaría el mejor caso del quicksort estaría dado por:

$$\sum_i^{lg(n)} 2^i \leq 2^{lg(n)+1} - 1; \quad (3.27)$$

donde  $n$  es la cantidad de elementos. La sumatoria exacta ocurriría si  $n = 2^k - 1$ , o sea, si  $n + 1$  es una potencia exacta de 2. En este caso, todas las particiones ocasionadas siempre son exactamente del mismo tamaño. La figura 3.6 pictoriza todas las llamadas recursivas que ocurren para  $n = 15$  en una estructura denominada "árbol", la cual será estudiada en el capítulo § 5 (Pág. 379).

Prestemos atención especial al diagrama 3.6 e identifiquemos la primera llamada `qs(0, 14)`<sup>8</sup>. Esta llamada genera las llamadas `qs(0, 6)` y `qs(8, 14)`. El algoritmo selecciona `qs(0, 6)` para continuar la recursión y empila `qs(7, 14)`. A su vez, `qs(0, 6)` empila `qs[4, 6]` y prosigue recursivamente sobre `qs(0, 2)`. Cuando se llega por primera vez al caso base, la pila contiene las llamadas `qs(0, 14)`, `qs(7, 14)`, `qs(4, 6)`, `qs(2, 2)` pendientes por hacer.

---

<sup>8</sup>El nombre del método ha sido abreviado por economía de espacio.

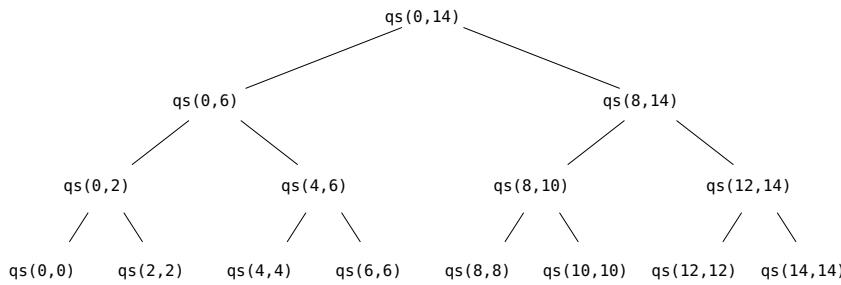


Figura 3.6: Estructura de las particiones y llamadas recursivas para el mejor caso del quicksort y 15 elementos.

La máxima cantidad de llamadas a empilar está dada por la máxima profundidad del árbol desde  $qs(0, 14)$  hasta cualquier caso base  $qs(i, i)$ . Cualquiera que sea el valor y tamaño de la entrada, si se procesa primero la partición más pequeña, entonces el consumo en pila no sobrepasa  $\mathcal{O}(\lg(n))$ .

### 3.2.2.4 Selección del pivote

Si la permutación del arreglo es aleatoria, entonces, la selección del pivote por la derecha del arreglo, tal como hasta el presente la hemos tratado, también es aleatoria. Por tanto, para permutaciones aleatorias, nuestra versión del quicksort debe comportarse  $\mathcal{O}(n \lg(n))$  en el caso esperado. Por añadidura, la varianza (29%) indica que en caso de mala suerte, el quicksort aún es rápido.

El razonamiento anterior presume que la permutación se distribuye uniformemente; cuestión plausible e idónea en el mundo objetivo de la estadística, pero que puede pecar de cándida en el mundo real. Todo depende de “qué” es lo que se ordena y “desde” dónde proviene la secuencia. En la vida ocurre que los “qué” y los “desde”, casi siempre son sesgados.

Por otra parte, el peor caso del quicksort es tan severamente calamitoso que ninguna aplicación sería pueble subyacer sobre él si el desempeño es lo crítico.

Lo anterior justifica consumir un poco de tiempo constante para forzar, en la medida posible, a que ocurran buenas particiones. La mejor manera de eliminar un sesgo es aleatorizar explícitamente. La idea es forzar a que cualquiera sea la permutación, la selección del pivote obedezca a un criterio aleatorio, el cual, tal como lo argüimos, probabilísticamente cause buenos casos.

En este orden de ideas, una primera estrategia consistiría en “aleatorizar” la partición, es decir, en sortear aleatoriamente la selección del índice del pivote. El índice pivote sería entonces un número aleatorio entre 1 y  $r$ . De este modo, la aleatoriedad ya no depende de modo alguno del valor de la permutación.

La técnica anterior es buena, pero no evita la posibilidad de que el pivote aleatorio cause una mala partición. Una técnica para disminuir esta posibilidad es sortear varios índices y seleccionar como pivote el valor de la mediana. Esto conlleva la dificultad de que hay que verificar que los sorteos no se repitan, lo que es algorítmicamente engorroso y, en duración constante, costoso. Por esa razón es preferible no verificar repitencia y simplemente seleccionar la mediana, así ocurran repeticiones. Queda por decidir cuántos sorteos se deben hacer. Cuanto mayor sea esta cantidad, mayor posibilidad se tendrá de

obtener un buen pivote, pero también se tendrá mayor consumo de tiempo constante y mayor impacto en el desempeño.

La aleatorización es un principio algorítmico tendiente a disminuir la “mala suerte” o a favorecer la “buena”. El principio ahora conocido bajo el rótulo de “algoritmos aleatorizados” y constituye toda una rama de la algorítmica. En este texto se estudiarán diversas clases de aleatorización.

Un esquema que no requiere generar índices aleatorios es tomar los índices extremos  $l$ ,  $r$  más el índice del centro y escoger la mediana de las tres entradas. Si bien, como siempre, la aleatoriedad depende de la permutación, tendría que haber permutaciones sesgadas por los tres lados para que ocurra un muy mal caso, algo poco probable de concebir pero susceptible de ocurrir o de ser adredeadamente provocado<sup>9</sup>. En este sentido, la siguiente rutina selecciona el pivote según lo recién explicado:

180a *(Métodos de ordenamiento 151a) +≡* ◀177 180c▶

```
template <typename T, class Compare> inline
int select_pivot(T * a, const int & l, const int & r)
{
 if (l - r <= 2)
 return r;

 const int m = (r + 1) / 2; // índice del centro
 const int p = Compare () (a[l], a[m]) ? m : l; // p=max(a[l],a[m])

 return Compare () (a[r], a[m]) ? r : p; // retornar min(a[r], a[m])
}
```

La rutina anterior permite completar el método definitivo de partición de la siguiente manera:

180b *(Seleccionar pivote 180b) ≡* (175b)

```
const int p = select_pivot <T, Compare> (a, l, r);
std::swap(a[p], a[r]);
```

### 3.2.2.5 Quicksort sin recursión

Conocido el patrón de demanda de espacio del quicksort, resulta atractivo a efectos de aligerar la carga por cálculo constante, intentar obtener una versión no recursiva del método. Asimismo, el patrón del algoritmo iterativo ya se conoce desde el diseño de la versión iterativa de la búsqueda binaria (§ 2.1.1.2 (Pág. 30)).

El quicksort debe usar una pila para recordar una partición mientras ordena la otra. Esta es la diferencia con la búsqueda binaria iterativa, la cual no requiere una pila porque no se requiere recordar la partición que se descarta después de la división.

Así pues, usaremos una pila de pares que contengan los índices izquierdo y derecho de la partición, cuya capacidad máxima (de la pila) será de 32 particiones, tamaño suficiente si siempre ordenamos de primero la partición más pequeña. De resto, condicionada al estadio de una apropiada redacción y de un responsable estudio, la siguiente rutina no debe plantear problemas de comprensión:

180c *(Métodos de ordenamiento 151a) +≡* ◀180a 181▶

```
template <typename T, class Compare = Aleph::less<T>> inline
```

---

<sup>9</sup>En seguridad de sistemas, una técnica de denegación de servicio, llamada “ataque por complejidad algorítmica”, consiste en explotar malos casos del algoritmo. En este sentido podría ocurrir una situación en la cual, deliberadamente, se le dé a un sistema malas particiones a efectos de causar su colapso.

```

void quicksort(T * a, const int & l, const int & r)
{
 typedef typename Aleph::pair<int, int> Partition;
 FixedStack<Partition, 32> stack;
 stack.push(Partition(l, r)); // todo el arreglo como partición inicial

 while (stack.size() > 0)
 {
 const Partition p = stack.pop();
 const int pivot = partition <T, Compare>(a, p.first, p.second);

 if (pivot - p.first < p.second - pivot) // ¿cuál más pequeña?
 { // partición izquierda más pequeña
 stack.push(Partition(pivot + 1, p.second));
 stack.push(Partition(p.first, pivot - 1));
 }
 else
 { // partición derecha más pequeña
 stack.push(Partition(p.first, pivot - 1));
 stack.push(Partition(pivot + 1, p.second));
 }
 }
}

```

Uses FixedStack 101a and partition 175b.

### 3.2.2.6 Quicksort sobre listas enlazadas

De todos los algoritmos de ordenamiento, quizá el más simple de expresarse sea el quicksort sobre listas enlazadas. La siguiente rutina lo indica:

181 ⟨Métodos de ordenamiento 151a⟩+≡ ◁180c 182b▷

```

template <class Compare> void quicksort(Dlink & list)
{
 if (list.is_unitarian_or_empty())
 return;

 Dlink * pivot = list.remove_next();
 Dlink smaller, bigger; // listas de menores y mayores que pivot

 ⟨particionar lista 182a⟩

 quicksort <Compare> (bigger);
 quicksort <Compare> (smaller);

 list.concat_list(&smaller); // restaurar listas ordenadas en list
 list.append(pivot);
 list.concat_list(&bigger);
}

```

La estructura del algoritmo no es nada misteriosa, pues se corresponde con el esquema del algoritmo: particionar la secuencia en el patrón smaller→pivot→bigger. La list

smaller sólo contiene los elementos menores que pivot, mientras que la lista bigger incluye los mayores.

Ahora nos falta especificar la partición, la cual es considerablemente más sencilla que con los arreglos:

182a  $\langle \text{particionar lista} \rangle \equiv$  (181 185)

```

while (not list.is_empty())
{
 Dlink * p = list.remove_next();
 if (Compare () (p, pivot))
 smaller.append(p);
 else
 bigger.append(p);
}

```

### 3.2.2.7 Mejoras al quicksort

Revisemos el procedimiento de partición (*Definición de partición de arreglo 175b*) y los ordenamientos de selección e inserción presentados en § 3.1.3 (Pág. 150) y § 3.1.8 (Pág. 162) respectivamente. Una primera mirada indica que estos métodos de ordenamiento tienen menos instrucciones que el procedimiento de partición del quicksort. Una revisión más acuciosa revela que los métodos de ordenamiento, que son  $\mathcal{O}(n^2)$ , contienen menos if, menos llamadas a Compare() y menos comparaciones de índices que el procedimiento de partición.

Los comentarios anteriores justifican la observación empírica de que los métodos de selección e inserción son más rápidos que el quicksort para arreglos pequeños. ¿Cómo es posible esto? Recordemos que la notación  $\mathcal{O}$  oculta los costes constantes y que ésta sólo rige después de un valor de entrada específico. He aquí, en el quicksort, un buen caso de aquellos costes que oculta la notación  $\mathcal{O}$ .

Una mejora substancial al quicksort consiste en comutar a otro método de ordenamiento para tamaños de partición pequeños. Lo lo más grande que pueda ser lo pequeño depende básicamente del hardware y del compilador. Puesto que el método de inserción exhibe mejores tiempos constantes que el de selección, lo integraremos a la siguiente versión mejorada del quicksort:

182b  $\langle \text{Métodos de ordenamiento} \rangle + \equiv$  ◁181 184▷

```

template <typename T, class Compare = Aleph::less<T> > inline
void quicksort_insertion(T * a, const int & l, const int & r)
{
 if (r <= l)
 return;

 const int pivot = partition<T, Compare>(a, l, r);

 const int l_size = pivot - l; // tamaño partición izquierda
 const int r_size = r - pivot; // tamaño partición derecha
 bool left_done = false; // true si partición izq está ordenada
 bool right_done = false; // true si partición der está ordenada

 if (l_size <= Aleph::Insertion_Threshold)
 { // ordene partición izq por inserción
 insertion_sort<T, Compare>(a, l, pivot - 1);
 }
}

```

```

 left_done = true;
 }
 if (r_size <= Aleph::Insertion_Threshold)
 {
 // ordene partición der por inserción
 insertion_sort<T, Compare>(a, pivot + 1, r);
 right_done = true;
 }
 if (left_done and right_done)
 return; // ambas particiones ordenadas por inserción

 if (left_done) // ¿partición izq ordenada por inserción?
 {
 // sí; sólo resta ordenar recursivamente partición der
 quicksort_insertion<T, Compare>(a, pivot + 1, r);
 return;
 }
 if (right_done) // ¿partición der ordenada por inserción?
 {
 // sí; sólo resta ordenar recursivamente partición izq
 quicksort_insertion<T, Compare>(a, l, pivot - 1);
 return;
 }
 // aquí, ambas particiones no fueron ordenadas por inserción
 if (l_size < r_size) // ordenar primero partición más pequeña
 {
 // partición izquierda más pequeña
 quicksort_insertion <T, Compare> (a, l, pivot - 1);
 quicksort_insertion <T, Compare> (a, pivot + 1, r);
 }
 else
 {
 // partición derecha más pequeña
 quicksort_insertion <T, Compare> (a, pivot + 1, r);
 quicksort_insertion <T, Compare> (a, l, pivot - 1);
 }
}

```

Uses partition 175b.

El valor exacto de `Insertion_Threshold` es relativo y depende de las condiciones materiales y operativas. Un valor entre 20 y 30 es suficiente en el momento de redacción de este texto<sup>10</sup>.

### 3.2.2.8 Claves repetidas

Aunque el algoritmo de partición (*Definición de partición de arreglo 175b*) funciona adecuadamente para claves repetidas, podría aprovecharse el proceso de partición para considerar repetiticia y realizar una partición con la forma siguiente:

| <i>l</i>                    |                         | <i>r</i>                    |
|-----------------------------|-------------------------|-----------------------------|
| Elementos menores que pivot | Elementos iguales pivot | Elementos mayores que pivot |

A este tipo de partición se le denomina “de la triple forma” o “bandera holandesa” en honor a la nacionalidad de Edsger W. Dijkstra, el primer hombre que se conoce haber reportado

<sup>10</sup>En enero 2006, procesadores i686 a 3 Ghz, con 2Mb de cache L2.

el descubrimiento del método. La ventaja de este esquema es que todas las claves repetidas quedan al centro y las invocaciones recursivas se efectúan sobre conjuntos más pequeños que no contienen repeticiones, lo cual acelera el ordenamiento.

La partición debe modificarse de manera tal que las repeticiones se vayan colocando por los extremos en un esquema similar al siguiente:



Luego, cuando los índices se crucen, se hacen los intercambios necesarios para dejar el arreglo triplemente particionado.

Los detalles del algoritmo se dejan como ejercicio.

### 3.2.2.9 Quicksort concurrente o paralelo

En el quicksort con listas enlazadas es muy fácilmente aprensible su facilidad de paralelización. En efecto, como luego de la partición el pivote ya se encuentra en su posición definitiva dentro del orden absoluto, las particiones pueden ordenarse independientemente, por separado, por distintos algoritmos, variantes y, en lo que concierne al propósito de esta observación, por procesadores diferentes. Como manera más familiar de aprehender el asunto, consideremos un mazo de barajas a ordenar y tres personas, una para particionar y las dos restantes para ordenar. Mientras la primera efectúa la partición, las dos restantes deben esperar, pero una vez que la partición está culminada, entonces las personas encargadas de ordenar pueden llevar a cabo el ordenamiento sin que interfieran entre ellas y con el orden final del mazo. Cuando las particiones estén ordenadas, el particionador junta los mazos, con lo que tiene un mazo completamente ordenado. La observación es idéntica para el ordenamiento con arreglos, aunque un poco más difícil de entender, dado el carácter de copia que se requiere.

### 3.2.2.10 Búsqueda aleatoria de clave

Cuando intentamos resolver el problema fundamental mediante arreglos nos encontramos con una disyuntiva. Si ordenamos, entonces podemos emplear la eficiente búsqueda binaria, pero tenemos una inserción y supresión  $\mathcal{O}(n)$ . Por el contrario, si mantenemos la secuencia desordenada, entonces tenemos una inserción muy rápida  $\mathcal{O}(1)$ , pero una búsqueda y supresión lentas  $\mathcal{O}(n)$ . En la mayoría de las ocasiones, la razón de ser de resolver el problema fundamental es la búsqueda, razón por la cual podríamos ordenar de vez en cuando o cuando estemos seguros de que no habrán más inserciones. Pero eso aún es costoso y limita la solución a un espectro pequeño de situaciones.

Una alternativa de búsqueda, híbrida entre el orden y el desorden, que permite implantar el conjunto fundamental mediante secuencias, se sirve de la partición del quicksort explicada en § 3.2.2.1 (Pág. 174). En efecto, podemos servirnos del método desarrollado en *(Definición de partición de arreglo 175b)* (§ 3.2.2.1 (Pág. 174)) para implantar la búsqueda sobre un arreglo desordenado:

184 *(Métodos de ordenamiento 151a) +≡* <182b 185>  
 template <typename T, class Compare= Aleph::less<T> > inline  
 int random\_search(T \* a, const T & x, const int & l, const int & r)

```

{
 if (l > r)
 return Not_Found;

 const int pivot = partition<T, Compare>(a, l, r);

 if (Compare() (x, a[pivot]))
 return random_search<T, Compare>(a, x, l, pivot - 1);
 else if (Compare() (a[pivot], x))
 return random_search<T, Compare>(a, x, pivot + 1, r);

 return pivot; // elemento encontrado en el índice x
}

```

Uses partition 175b.

La idea del algoritmo es particionar aleatoriamente el arreglo y obtener el punto de partición pivot. Recordemos que el pivote siempre se encuentra en su posición de orden definitiva. Por tanto, el orden de  $x$  respecto a pivot determina en cuál de las dos particiones se continúa recursivamente la búsqueda.

Por supuesto, según el análisis de § 3.2.2.1 (Pág. 174), el algoritmo recorre enteramente el arreglo, razón por la cual es fácil ver que la búsqueda general no podría apoyarse enteramente en random\_search(). Pero sí podría, empero, mantener una lista ordenada de pivotes a partir de la cual hacer búsquedas secuenciales que serían considerablemente más cortas que recorrer enteramente al arreglo.

El análisis formal de este algoritmo se delega como ejercicio. La búsqueda aleatorizada es perfectamente plausible con listas enlazadas. El principio (y el algoritmo resultante) es el mismo: particionar la lista según el pivote y luego decidir en cuál de las dos particiones continuar la búsqueda:

185

*(Métodos de ordenamiento 151a)* +≡

△184 186a▷

```

template <typename T, class Compare> inline
Dnode<T> * dlink_random_search(Dlink & list, const T & x)
{
 if (list.is_empty())
 return NULL;

 Dnode<T> item(x);
 Dnode<T> * item_ptr = &item; // puntero a celda que contiene a x
 Dlink smaller; // lista de los menores que pivot
 Dlink bigger; // lista de los mayores que pivot

 Dnode<T> * pivot = static_cast<Dnode<T>*>(list.remove_next());
 (particionar lista 182a)

 Dnode<T> * ret_val = NULL;
 if (Compare () (item_ptr, pivot))
 ret_val = dlink_random_search <T, Compare> (smaller, x);
 else if (Compare () (pivot, item_ptr))
 ret_val = dlink_random_search <T, Compare> (bigger, x);
 else
 ret_val = pivot;

 list.swap(&smaller);

```

```

 list.append(pivot);
 list.concat_list(&bigger);
 return ret_val;
}

```

Uses Dnode 83a.

Por economía de código, `dlink_random_search()` asume que se busca sobre una lista de Dlink. Esto requiere que el usuario surta una clase de comparación que considere objetos de tipo Dlink y no `Dnode<T>`, lo cual hace al asunto algo engoroso, pues el usuario debe convertir los punteros del tipo Dlink a `Dnode<T>`. Como ya explicamos en § 3.1.3 (Pág. 153), la clase genérica `Compare_Dnode` nos posibilita la implantación de `random_search()` para objetos de tipo `Dnode<T>` y derivados.

### 3.2.2.11 Selección aleatoria sobre secuencia desordenadas

Bajo el mismo fundamento del algoritmo anterior, es posible implantar la selección aleatoria, es decir, buscar el  $i$ -ésimo menor elemento dentro de la secuencia. Esto se lleva a cabo de la siguiente manera:

186a *(Métodos de ordenamiento 151a) +≡* ◀185 186b▶

```

template <typename T, class Compare> static inline
const T & __random_select(T * a, const int & i, const int & l, const int & r)
{
 const int pivot = partition<T, Compare>(a, l, r);
 if (i == pivot)
 return a[i];

 if (i < pivot) // ¿está en partición izquierda?
 return __random_select<T, Compare>(a, i, l, pivot - 1);
 else
 return __random_select<T, Compare>(a, i - (pivot - l + 1), pivot + 1, r);
}

```

Uses `partition` 175b.

Importante señalar que aquí el problema sí se resuelve más rápido con `random_select()` que mediante la mera búsqueda secuencial. En efecto, en último caso, la selección sería  $\mathcal{O}(n^2)$ <sup>11</sup>.

Es conveniente verificar que el valor de  $i$  se circunscriba al rango  $[l, r]$ . A efectos de ahorrar duración de cálculo ponemos la verificación una sola vez, dentro de la interfaz pública, en lugar de hacerlo a cada llamada recursiva. De este modo, la rutina pública se define así:

186b *(Métodos de ordenamiento 151a) +≡* ◀186a 187b▶

```

template <typename T, class Compare = Aleph::less<T> >
const T & random_select(T * a, const int & i, const int & n)
{
 return __random_select(a, i, 0, n - 1);
}

```

Para la selección aleatoria con listas enlazadas debemos añadir al proceso *(particionar lista 182a)* la contabilización de las cardinalidades de las dos particiones resultantes. Esto

<sup>11</sup>Véase ejercicio 22 de capítulo § 2 (Pág. 27).

se puede hacer de la siguiente manera:

187a *(particionar lista y contar 187a)*≡  
 while (not list.is\_empty())  
 {  
 Dlink \* p = list.remove\_next();  
 if (Compare () (p, pivot)) // ¿p < pivot?  
 { smaller.append(p); ++smaller\_count; }  
 else  
 { bigger.append(p); ++bigger\_count; }  
}

187b *(Métodos de ordenamiento 151a)*+≡  
 template <class Compare>  
 Dlink \* dlink\_random\_select(Dlink & list, const size\_t & i)  
{  
 if (list.is\_empty())  
 return NULL;

Dlink smaller; // lista de los menores que pivot  
 Dlink bigger; // lista de los mayores que pivot  
 size\_t smaller\_count = 0, // cantidad de elementos de smaller  
 bigger\_count = 0; // cantidad de elementos de bigger

Dlink \* pivot = list.remove\_next();  
*(particionar lista y contar 187a)*

Dlink \* ret\_val = NULL;  
 if (i == smaller\_count)  
 ret\_val = pivot;  
 else if (i < smaller\_count)  
 ret\_val = dlink\_random\_select<Compare>(smaller, i);  
 else  
 ret\_val = dlink\_random\_select<Compare>(bigger, i - (smaller\_count+1));  
  
 list.concat\_list(&smaller);  
 list.append(pivot);  
 list.concat\_list(&bigger);  
 return ret\_val;  
}

△186b

### 3.3 Análisis amortizado

Consideremos una situación en la cual queremos estudiar el desempeño de un TAD o una estructura de datos. La notación  $\mathcal{O}$  nos proporciona un mecanismo sencillo y objetivo para analizar la tendencia de la curva de duración de un algoritmo. Sin embargo, cuando analizamos el desempeño de un TAD debemos considerar otros aspectos adicionales:

- Un TAD tiene varias operaciones cuyos desempeños pueden ser diferentes y que pueden basarse en algoritmos distintos. Por tanto, analizar un TAD puede requerir analizar distintos algoritmos.

- Por lo general, un TAD subyace en una estructura de datos. En este sentido, su análisis requiere considerar el estado actual de la estructura de datos, pues éste, muy probablemente, tenga relación directa con el desempeño de la operación.
- El estado de la estructura de datos depende de la permutación de operaciones que se suceda sobre el TAD. Secuencias de operación distintas producirán tiempos distintos. Al respecto, ¿cuánto demoraría efectuar  $n$  operaciones sobre un TAD?, ¿podemos establecer una cota independiente del valor de la permutación?

Así las cosas, veamos los problemas del mero uso de la notación  $\mathcal{O}$  al analizar un TAD. Para eso, examinemos el TAD `DynListStack<T>` (§ 2.5.4 (Pág. 105)) y una secuencia cualquiera de operaciones sobre este TAD; en particular, consideremos las secuencias posibles que comporten pushes y pops. Si rememoramos la implantación de `DynListStack<T>`, entonces podemos aprehender que tanto `push()` como `pop()` toman  $\mathcal{O}(1)$ . Por tanto, cualquier secuencia de  $n$  operaciones sobre `DynListStack<T>` toma  $n\mathcal{O}(1) = \mathcal{O}(n)$ . Acabamos de hacer nuestro primer análisis sobre el comportamiento de un TAD como un todo.

Ahora añadimos al TAD `DynListStack<T>` la operación `popn(n)`, la cual extrae  $n$  elementos de la pila<sup>12</sup>. Tal operación puede hacerse del siguiente modo:

188 *(pushn para DynListStack<T> 188)≡*

```
template <typename T>
void popn(DynListStack<T> & stack, const size_t & n)
{
 for (int i = 0; i < n and stack.size() > 0; ++i)
 stack.pop();
}
```

Uses `DynListStack 105c`.

Si la pila contiene  $n$  elementos, entonces el coste de `popn()` es  $\mathcal{O}(n)$  para el peor caso. Supongamos que recibimos la versión extendida del TAD `DynListStack<T>` con las siguientes especificaciones de rendimiento basadas en el análisis tradicional<sup>13</sup>:

| Nombre de operación     | Eficiencia peor caso |
|-------------------------|----------------------|
| <code>push(item)</code> | $\mathcal{O}(1)$     |
| <code>pop()</code>      | $\mathcal{O}(1)$     |
| <code>popn(n)</code>    | $\mathcal{O}(n)$     |

¿Cuánto cuesta una secuencia de  $n$  operaciones consecutivas sobre un objeto `DynListStack<T>`? Para responder por el peor caso tomamos la operación `popn()`, que es la más costosa, y calculamos el coste de ejecutarla  $n$  veces consecutivas. A través de este razonamiento llegamos a la conclusión de que el coste total es  $n \times \mathcal{O}(n) = \mathcal{O}(n^2)$ . Pero este razonamiento no es justo, pues el desempeño de `popn()` depende de la cantidad de elementos en la pila. Una llamada a `popn(n)` toma  $\mathcal{O}(n)$ , pero las siguientes tomarán  $\mathcal{O}(1)$ , ¡pues se ejecutan sobre una pila vacía!

El análisis anterior, más subjetivo, que considera las circunstancias de invocación a `popn()`, nos revela que cualquier secuencia de  $n$  operaciones es, en promedio,  $\mathcal{O}(n)$ .

<sup>12</sup>Operación que fue definida para el TAD `ArrayList<T>` desarrollado en § 2.5.2 (Pág. 101) y que toma  $\mathcal{O}(1)$ .

<sup>13</sup>El resto de las operaciones `size()`, `is_empty()` y `top()` se omiten porque tienen tiempo constante y no producen modificaciones sobre la estructura de datos.

Una técnica de vanguardia que intenta con cierto éxito objetizar el análisis de tipos abstractos de datos se denomina “análisis amortizado”. La idea es ponderar los costes de operaciones de un TAD a través de un potencial abstracto o de créditos.

### 3.3.1 Análisis potencial

En esta clase de análisis debemos encontrar una “función potencial”  $\Phi : \mathcal{D} \rightarrow \mathcal{R}$ , donde  $\mathcal{D}$  representa la estructura de datos subyacente al TAD y  $\mathcal{R}$  es el conjunto de los números reales.  $\mathcal{D}$  se define como  $\mathcal{D} = \{D_0\} \cup \{D_1, D_2, \dots, D_n\}$ , donde  $D_0$  es el estado inicial de la estructura de datos y  $\{D_1, D_2, \dots, D_n\}$  son los estados subsecuentes resultantes de una secuencia de operaciones sobre el TAD.

¿De qué manera puede usarse la función potencial para analizar un TAD? Para abordar la pregunta, planteemos la siguiente idea de coste amortizado para la  $i$ -ésima operación sobre un TAD:

$$\hat{c}_i = t_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{Diferencia de potencial: } \Delta_{\Phi_i}} \quad (3.28)$$

$t_i$  es la duración de ejecución de la  $i$ -ésima operación, mientras que  $\Delta_{\Phi_i}$  es la “diferencia de potencial” entre la operación actual y la anterior, la cual representa el consumo o la compensación de energía según que el diferencial sea positivo o negativo. En este caso, el potencial denota a una especie de energía que se gasta o se gana entre una operación y otra.

El coste amortizado modeliza el hecho de que algunas operaciones cuesten más que otras, pero también el que otras operaciones aumenten el potencial de la estructura de datos de manera tal de que dispongan de energía cuando actúen.

La ecuación (3.28) nos conduce a definir el coste total de  $n$  operaciones de la siguiente forma:

$$\begin{aligned} T(n) &= \sum_{i=1}^n t_i = \sum_{i=1}^n \left( \hat{c}_i - \underbrace{(\Phi(D_i) - \Phi(D_{i-1}))}_{\Delta_{\Phi_i}} \right) \\ &= \sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n \Phi(D_i) + \sum_{i=1}^n \Phi(D_{i-1}) \Rightarrow \end{aligned} \quad (3.29)$$

$$\begin{aligned} \hat{C} &= \sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n t_i + \sum_{i=1}^n \Phi(D_i) - \sum_{i=1}^n \Phi(D_{i-1}) \\ &= \sum_{i=1}^n t_i + (\Phi(D_1) + \dots + \Phi(D_n)) - (\Phi(D_0) + \dots + \Phi(D_{n-1})) \\ &= \sum_{i=1}^n t_i + \Phi(D_n) - \Phi(D_0) \end{aligned} \quad (3.30)$$

Para que el potencial tenga sentido, es menester definir  $\Phi$  de manera tal que, para toda permutación posible de operaciones,  $\Phi(D_n) \geq \Phi(D_0)$ , es decir, que la diferencia potencial total entre el estado inicial de la estructura de datos y la última operación siempre sea positivo, pues si no, en algún momento, la estructura de datos carecería de potencial, lo cual parece no tener sentido en nuestro contexto terrenal. Podemos asumir que  $\Phi(D_0) \geq 0$ , pues la idea de una estructura de datos es dar potencia a las futuras operaciones. Bajo

esta asunción,  $\Phi(D_n) - \Phi(D_0) \geq 0$ , lo cual, en función de (3.30), nos conduce a afirmar la desigualdad siguiente:

$$T(n) = \sum_{i=1}^n t_i \leq \sum_{i=1}^n \hat{c}_i \quad (3.31)$$

Bajo las premisas planteadas, la desigualdad (3.31) nos demuestra que  $\hat{C}(n)$  acota superiormente a  $T(n)$ , razón por la cual podemos trabajar con seguridad mediante  $\hat{C}(n)$  y los peores casos de  $t_i$  según cada una de las operaciones del TAD y las posibles permutaciones de secuencias de operación.

Consideremos la versión extendida del TAD DynListStack<T> y definamos  $\Phi : \mathcal{D} \rightarrow \mathcal{R}$  como la cantidad de elementos que tiene la pila al término de una operación, por lo que  $\Phi(D_0) = 0$ . Estudiemos lo que ocurre con los potenciales mirando alguna secuencia de operación particular, por ejemplo, la mostrada por la tabla siguiente:

| i | Operación | $t_i$ | $\Phi(D_i)$ | $\Delta_{\Phi_i} = \Phi(D_i) - \Phi(D_{i-1})$ | $\hat{c}_i = t_i + \Delta_{\Phi_i}$ |
|---|-----------|-------|-------------|-----------------------------------------------|-------------------------------------|
| 0 | -         | -     | 0           | -                                             | -                                   |
| 1 | push()    | 1     | 1           | 1                                             | 2                                   |
| 2 | push()    | 1     | 2           | 1                                             | 2                                   |
| 3 | push()    | 1     | 3           | 1                                             | 2                                   |
| 4 | pop()     | 1     | 2           | -1                                            | 0                                   |
| 5 | push()    | 1     | 3           | 1                                             | 2                                   |
| 6 | popn(3)   | 3     | 0           | -3                                            | 0                                   |

Al final de un push(), el tamaño de la pila se incrementa en uno, por lo que siempre se cumple:

$$\hat{c}_i = \underbrace{t_i}_{1} + \underbrace{\Delta_{\Phi_i}}_{1} ;$$

si la  $i$ -ésima operación es push().

Al final de un popn( $k$ ),  $\Phi(D_i) = N_{i-1} - k$  donde  $N_{i-1}$  es la cantidad de elementos antes de efectuar popn(). De este modo, luego de ejecutado un popn( $k$ ), el coste amortizado será:

$$\hat{c}_i = \underbrace{k}_{k} + \underbrace{N_{i-1} - k - N_i}_{\Delta_{\Phi_i}} = 0 ;$$

cada vez que la  $i$ -ésima operación sea popn( $k$ ). Notemos que el coste de pop() es equivalente a popn(1).

En síntesis, independientemente de la permutación de la secuencia de operación, los costes amortizados del TAD DynListStack<T> son los siguientes:

| Nombre de operación | $\hat{c}$ |
|---------------------|-----------|
| push(item)          | 2         |
| pop()               | 0         |
| popn(n)             | 0         |

Ahora estudiemos el coste total de  $n$  operaciones consecutivas sobre el TAD DynListStack<T>. Podemos dividir las operaciones en  $n_1$  push() y  $n_2$  popn() (recordemos que pop()  $\iff$  popn(1)). Por tanto:

$$T(n) \leq \hat{C}(n) = \mathcal{O}(n_1) \times 2 + \mathcal{O}(n_2) \times 0 = \mathcal{O}(n_1) = \mathcal{O}(n)$$

Así pues, el coste promedio por operación entre cualquier permutación de  $n$  operaciones, está dado por:

$$\bar{t}_i \leq \frac{\hat{C}(n)}{n} = \frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$$

### 3.3.2 Análisis contable

En un análisis amortizado contable, a cada operación se le asigna una cantidad constante llamada crédito, escogida cuidadosamente según la naturaleza de la operación, la cual no necesariamente se corresponde con el coste real en duración. Si el coste en duración de la  $i$ -ésima operación es  $t_i$ , y  $\hat{c}_i$  su coste crédito, entonces requerimos satisfacer:

$$\hat{C} = \sum_{i=1}^n \hat{c}_i \geq T(n) = \sum_{i=1}^n t_i \quad (3.32)$$

para cualquier secuencia posible de  $n$  operaciones sobre el TAD o estructura de datos. El coste amortizado del TAD o estructura de datos para una secuencia de  $n$  operaciones es:

$$\hat{C} = \frac{\sum_{i=1}^n \hat{c}_i}{n} \quad (3.33)$$

La desigualdad es la misma que la expresada en (3.31), pero asumiendo que el coste amortizado  $\hat{c}_i$  es constante para cada operación (lo cual resultó el caso para el TAD DynListStack< $T$ >). Sin embargo, el razonamiento para la selección del crédito de cada operación, aunque inspirado en la idea de potencial, está prestado del argot mercantil. La idea es que siempre deba haber crédito para costear cualquier secuencia de operación. En este sentido, los créditos de cada operación deben seleccionarse de modo tal que “financien” con créditos a otras.

De cierta manera, la asignación de créditos puede interpretarse como una función potencial discreta y por partes según la operación.

Por ejemplo, asignemos al TAD extendido DynListStack< $T$ > los siguientes créditos amortizados:

| Nombre de operación | Créditos |
|---------------------|----------|
| push(item)          | 2        |
| pop()               | 0        |
| popn(n)             | 0        |

Para brindar sentido al razonamiento que nos conduce a esta selección de créditos debemos observar que no puede haber un pop() si antes no hubo un push(). De este modo podemos decir que cuando se efectúa un push() se dejan dos (2) créditos, uno correspondiente al coste en duración y otro dejado en préstamo o garantía para costear la duración de extracción del elemento cuando se efectúe su pop(). Puesto que las operaciones han sido costeadas por adelantado por el push(), pop() y popn() no tienen créditos, pues sus costes han sido amortizados por el push().

Con los costes amortizados anteriores podemos calcular una cota para el coste amortizado del TAD DynListStack< $T$ >, el cual no excederá de:

$$\hat{C} \leq \frac{2n}{n} = 2 = \mathcal{O}(1)$$

El término “crédito”, proveniente del verbo latino “*crēdēre*”, significa creer, confiar, dar fe en una persona. Hoy en día, el mundo económico emplea el término crédito para referir a una cantidad de dinero que se otorga en préstamo; tal parece que la presunción de buena es cuantificable o, por añadidura, está vinculada al poder económico<sup>14</sup>.

El término “amortizar” se usa en el mundo bancario y es desde allí que proviene la metáfora con este tipo de análisis. Curiosamente, “amortizar” proviene del francés, el cual, a su vez, proviene del latín medieval *admortizare*. El prefijo *ad* indica “proximidad”, mientras que *mortis* significa “muerte”. Amortizar significaría entonces la proximidad de la muerte, pero el término se usa desde la Edad Media para connotar la proximidad de la muerte de una deuda.

### 3.3.3 Selección del potencial o créditos

Para que un análisis amortizado tenga sentido, es esencial encontrar una buena función potencial o lograr interpretar las operaciones en función de su trabajo y la manera en que ellas amortizan o consumen créditos entre sí.

De una cierta forma, el análisis amortizado es subjetivo en el sentido de que delega en la función potencial la subjetividades del TAD y de su estructura de datos. Es entonces durante el estudio y selección de la función potencial (o de los créditos por operación) cuando se tratan tales subjetividades.

Por lo general, la función potencial debe ponderar el estado de la estructura de datos. En el caso de una secuencia, el tamaño es a menudo una buena escogencia.

Una manera de facilitar el análisis amortizado de una operación consiste en dividirla en etapas secuenciales, analizar cada una por separado y luego sumarlas. Supongamos que una secuencia de operaciones y una operación efectuada en la  $i$ -ésima vez con coste amortizado  $\hat{c}_i$  entre el estado  $D_{i-1}$  y  $D_i$ . Ahora supongamos que la fase  $D_{i-1} \rightarrow D_i$  puede dividirse en las fases  $D'_1, D'_2, \dots, D'_k$ , de manera tal que  $D_{i-1} = D'_0$  y  $D_i = D'_k$ . Sea  $t'_j$  el coste de pasar del estado  $D'_{j-1}$  a  $D'_j$ , entonces:

$$t_i = \sum_{j=1}^k t'_j \quad (3.34)$$

Según (3.28), el coste amortizado de  $t'_i$  puede definirse como:

$$\begin{aligned} \sum_{j=1}^k \hat{c}'_j &= \sum_{j=1}^k (t'_j + \Phi(D'_j) - \Phi(D'_{j-1})) \\ &= \sum_{j=1}^k t'_j + \Phi(D'_k) - \Phi(D'_0) \\ &= t_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \hat{c}_i \end{aligned} \quad (3.35)$$

<sup>14</sup>El premio Nobel de la Paz de 2006 fue conferido al profesor bengalí Muhammad Yunus, fundador del Banco Grameen (banco rural). El banco en cuestión concede “microcréditos” a prestatarios muy pobres materialmente, sin ninguna garantía o “crédito” económicos, permitiéndoles confrontar su pobreza material. No sorprendentemente, el 90% de los prestatarios reintegran completamente el préstamo, aunque también el 95% de los prestatarios son mujeres.

Por tanto, el coste amortizado de la operación es la suma de los costes amortizados de sus etapas.

Además de los conocimientos sobre la estructura de datos y de la técnica de análisis, en la búsqueda y selección de una función potencial intervienen la experiencia, el arte y la suerte.

## 3.4 Correctitud de algoritmos

Cuando hablamos acerca de la correctitud de un algoritmo o programa nos referimos a que éste logre el fin para el cual fue destinado. Puesto de otro modo, la correctitud concierne a lo que al principio de este capítulo denominamos “eficacia”, lo cual, en el caso de un programa, equivale a decir que éste no arroje malas respuestas. A menudo, esto no es sencillo de determinar porque hasta que el programa esté operativo no se puede verificar si es o no correcto.

La eficacia que, recordemos, no es lo mismo que la eficiencia, se juzga por completitud y minimalidad. Por completitud decimos que el efecto, o sea, el fin, se alcance cabalmente. Por minimalidad decimos que no se efectúe algo adicional al fin, lo que en el lenguaje tecnocrático se denomina “efecto colateral”, expresión que comúnmente indica que la consecución del fin acarrea efectos adicionales que frecuentemente son indeseables.

El 4 de julio de 1996 se lanzó por vez primera el cohete europeo “Ariane 5”. Inmediato al lanzamiento, el cohete empezó a desviarse de su trayectoria vertical y debió destruirse 40 segundos después. Las investigaciones ulteriores revelaron que una rutina trataba números en punto flotante como si estuviesen representados en aritmética entera. La versión anterior del cohete “Ariane 4” manejaba enteros y la rutina en cuestión no se verificó exhaustivamente para la nueva versión del cohete. La rutina era incorrecta [72, 85].

### 3.4.1 Planteamiento de una demostración de correctitud

Una interpretación de correctitud estriba en determinar si el programa arroja respuestas correctas para todas las entradas posibles. En función de esto, puesto que en la mayoría de las ocasiones, los valores posibles de entrada son o muy grandes o infinitos, una prueba de correctitud requiere clasificar la entrada en todos los subconjuntos posibles que conforman la entrada.

Lo anterior puede plantearse formalmente como sigue. Sea  $\mathcal{I}$  el conjunto dominio de entrada de un programa y  $\mathcal{R}$  el conjunto resultado. Entonces, una prueba de correctitud debe examinar el algoritmo y determinar en función del mismo (bifurcaciones, lazos, etcétera), los subconjuntos de entrada  $\mathcal{I} = \mathcal{I}_1 \cup \mathcal{I}_2 \cup \dots \cup \mathcal{I}_n$ , junto con sus subconjuntos de salida  $\mathcal{R} = \mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n$ . Notemos que los subconjuntos resultado pueden solaparse entre sí, o sea, que los resultados no tienen por qué ser excluyentes.

La prueba de correctitud consiste entonces en verificar si para cada subconjunto  $\mathcal{I}_i$  el algoritmo o programa arroja la salida  $\mathcal{R}_i$ . Puesto que a menudo  $\mathcal{I}_i$  y  $\mathcal{R}_i$  son muy grandes o infinitos, la prueba se efectúa con los valores frontera de los subconjuntos.

Por ejemplo, una inspección más profunda del ordenamiento por selección estudiado en § 3.1.3 (Pág. 150) revela que el bloque  *$\langle$ Sea min el índice menor entre  $i + 1$  y  $n - 1$  151b $\rangle$*  siempre encuentra el mínimo entre  $i+1$  y  $n-1$ , pues todo el rango ( $i..n$ ) es

inspeccionado. Por tanto, *(Sea min el índice menor entre  $i + 1$  y  $n - 1$  151b)* es correcto.

Por otra parte, el for más externo recorre todo el arreglo entre  $[0..n - 1]$ ; sólo falta el último elemento del arreglo, el cual sería inspeccionado por *(Sea min el índice menor entre  $i + 1$  y  $n - 1$  151b)*.

Por tanto, todos los elementos del arreglo son inspeccionados para determinar el mínimo. El algoritmo parece ser, pues, correcto. El conjunto de entrada fue  $\mathcal{I} = \{\text{Todas las permutaciones posibles}\}$  y el de salida  $\mathcal{R} = \{\text{Todas las permutaciones ordenadas}\}$ .

### 3.4.2 Tipos de errores

Un error en un programa puede cometerse durante las siguientes fases:

**Diseño:** premisas incorrectas, asumidas durante el diseño de un algoritmo, muy posiblemente conduzcan a resultados incorrectos.

**Prueba de correctitud del algoritmo:** a veces, la correctitud de un algoritmo no es sencilla de demostrar. En este sentido hay dos clases generales de errores en la prueba:

1. Que las instancias posibles de entrada no estén completamente cubiertas, caso en el cual no se sabría si el algoritmo es correcto o no para la clase de entrada ausente.
2. Que alguna o más de las pruebas esté errada.

Posiblemente, los errores en la prueba de correctitud son los más serios, pues inducen lo que se denomina como “falsa percepción”; el algoritmo podría o no estar correcto, pero afirmamos, sin certitud, que lo es.

**Codificación:** los algoritmos deben traducirse a un lenguaje de programación. Esta fase se llama codificación y durante ella se escogen las representaciones de los datos bajo la forma de estructura de datos, así como las representaciones de las secuencias de ejecución del algoritmo según los modelos del lenguaje (while, for, if, etcétera).

**Ejecución y falla de programas de verificación:** Como mencionaremos posteriormente, para evaluar correctitud se emplean diversos tipos de programas. En este sentido, una cuarta clase de error puede inducirse por el hecho de que uno de los programas usados para verificar correctitud sea incorrecto.

Ante estas perspectivas de la correctitud, aunado al hecho de que ésta en sí se aprehende cuando se ejecute el programa y se resuelva el problema, un programador debe asumir permanentemente una actitud de duda respecto a la correctitud.

### 3.4.3 Prevención y detección de errores

Hay dos enfoques complementarios entre sí para lidiar con la correctitud de un programa:

1. Prevención: ¿cómo evitar cometer un error?

## 2. Detección: ¿como detectarlo lo más pronto posible?:

La prevención requiere una “actitud” crítica por parte del programador en el sentido de que aprenda de sus errores y busque maneras de no volver a cometerlos.

La incorrectitud siempre es costosa en el sentido de que de alguna manera acarrea pérdidas, pero hay situaciones en las cuales es demasiado costosa. Por ejemplo, una falla en un programa controlador ejecutándose en un computador industrial puede causar una parada de planta que detiene la cadena productiva; más costoso es una falla en un programa de misión crítica, un satélite, por ejemplo.

La detección requiere forzosamente un análisis inspectivo del algoritmo o programa. Cuando el análisis se lleva a cabo sin ejecutar el programa en cuestión, se le denomina “estático”. Análogamente, cuando el análisis se vale de la ejecución del programa se le llama “dinámico”. A la pregunta de cuál es mejor, la respuesta es que cuanto antes se detecte un error menos costoso es éste. Consecuentemente, parece obligatorio efectuar análisis estático antes de instrumentar un programa. Para aproximar el entendimiento de este asunto, consideremos la gráfica<sup>15</sup> de la figura 3.7.

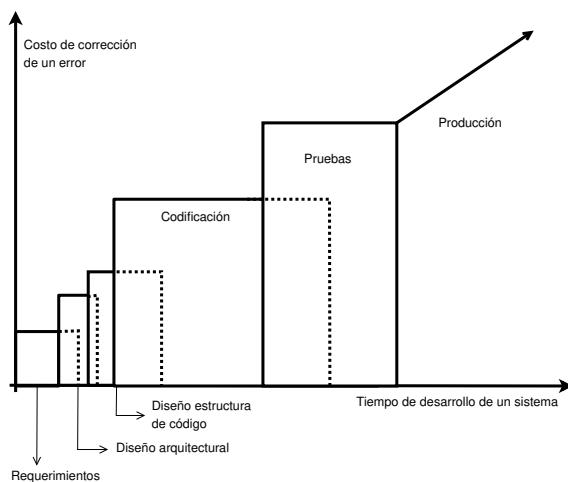


Figura 3.7: Costes de corrección de errores en función de la etapa de desarrollo de un proyecto de software

La figura 3.7 pictoriza las diferentes etapas del llamado “ciclo de desarrollo de software”. Durante las tres primeras fases se toman decisiones de diseño de algoritmos y mecanismos de sistema, mientras que durante las tres restantes es cuando típicamente se detectan los errores, siendo los de diseño los más costosos. Podemos decir que el coste de un error y su corrección es directamente proporcional al momento en que se detecta. Consiguientemente, cuanto más tarde se detecte un error, más impacto en costes éste tiene.

<sup>15</sup> Esta gráfica está inspirada del curso sobre Promela/Spin, impartido el 11 de abril de 2002, por Theo C. Ruys, en Grenoble, Francia, durante el *Spin 2002 Workshop*.

Para más detalles, véase:

- <http://wwwhome.cs.utwente.nl/~ruys/> y
- <http://www.spinroot.com/>

De la reflexión anterior se debe desprender un fuerte sentido de precaución que se debe asumir durante las tres primeras etapas del ciclo de desarrollo. El problema básico, y esta es la trampa a la que sucumben muchos ingenieros y programadores, es que cuando se lleva a cabo un nuevo sistema y se toman decisiones de diseño, no se conocen cuáles serán sus bondades en eficacia y eficiencia. Ahora bien, muchas veces se cae en la trampa de tomar decisiones de diseño sin conocer a priori sus bondades. Consecuentemente, muchas veces se encuentran errores de diseño, que son de los más costosos, durante la fase de instrumentación. ¿Cómo se puede disminuir este riesgo? La respuesta es haciendo modelos de diseño y verificándolos antes de su implementación, tal como se acostumbra hacer en otras ramas de la ingeniería.

#### 3.4.3.1 Disciplina de programación

Buenas son las costumbres, buenas deben ser las obras. Una costumbre es un hábito adquirido de conocimientos prácticos, comprobados, descubiertos durante hechuras de buenas obras. En programación, como en cualquier otra práctica, los practicantes forjan costumbres, buenas costumbres, porque si no se les moriría la práctica. Cuando una costumbre deviene común entre todos (o la mayor parte de) los practicantes, entonces ésta deviene en “norma”.

La observancia de una costumbre, más aún de una norma, requiere, según se sea aprendiz o maestro, de un esfuerzo consciente de voluntad por recordar la aplicación de la norma y de una fuerza de carácter por no transgredirla. A este esfuerzo y fuerza se le denomina “disciplina”.

Hoy en día, a las buenas costumbres se les llama “buenas prácticas”, pero la redundancia de este calificativo plantea una ambigüedad con la noción de práctica, razón por la cual seguiremos usando “costumbre” o, cuando ésta sea objetiva entre los practicantes, “norma”.

¿Cuáles son las costumbres de programación que subyacen detrás de programas correctos? Numerosos, voluminosos e interesantes textos han sido escritos sobre este asunto. No hay, pues, espacio en esta subsección para presentar y discutir todo lo inherente a la disciplina de programación. Pero sí podemos remitirnos al siguiente decálogo de Gerard Holtzmann [73] para la escritura de programas correctos críticos.

#### Decálogo de Holtzmann [73]

##### 1 No usar goto, longjmp o recursión directa o indirecta

Salvo algunas excepciones particulares, la instrucción goto ha sido abolida desde hace unas cuantas décadas porque va contra la legibilidad de código. Por añadidura, los analizadores estáticos tienen serias dificultades para interpretar qué sucede con un goto. Lo mismo ocurre con el conjunto de instrucciones longjmp de la biblioteca estándar C.

Rutinas recursivas, directa o indirectamente, son también muy difíciles de analizar por analizadores estáticos. Además, la recursión usa implícitamente la pila del sistema, cuya gestión es difícil de acotar.

Hay, por supuesto, excepciones. Muy en particular, en este texto, con la recursión directa. En ese sentido podemos decir que sólo debe utilizar recursión para funciones cuya correctitud haya sido demostrada y cuyo consumo de pila se garantice a estar acotado. En esos casos, la función recursiva puede substituirse por un tronco o indicarse a un analizador que la función en cuestión es correcta.

## 2 Siempre asegurar que todo lazo termine

¿Cuántas veces no hemos ejecutado un programa que no termina? En el caso general, los programas no terminan -cuando están diseñados para terminar- porque caen en lazos eternos<sup>16</sup>.

La regla consiste entonces en evidenciar garantía de que cada lazo del programa (while, do-while, o for) termine. Cada vez que se programe un lazo, pregúntese y respóndase cómo se asegura que éste siempre termine.

La única excepción a esta regla es cuando se escribe un lazo que jamás debe terminar. Normalmente, esto ocurre en programas iterativos, por ejemplo, un servidor. En este caso, la regla es asegurar que, en efecto, el lazo nunca termine.

## 3 No utilizar memoria dinámica después de haber inicializado el programa

En el espíritu de este texto, la regla es: independizar toda abstracción del uso de memoria dinámica. En otras palabras, cualquier algoritmo o estructura de datos debe concebirse sin dependencia en memoria dinámica.

Por memoria dinámica entendemos llamadas a malloc()/free(), new/delete, sbrk(), alloca() y, en general, a cualquier otra primitiva de manejo de memoria. Por inicialización entendemos la primera fase de un programa en la cual se declaran las estructuras de datos y demás clases de variables y se les asignan sus valores primigenios.

La razón de ser de esta regla es doble. En primer lugar, la inmensa mayoría de mecanismos de administración de memoria no ofrecen cotas constantes sobre el tiempo de ejecución. A medida que se gestiona más memoria, más fragmentación ocurre en el espacio de direccionamiento, lo cual degrada el desempeño y la probabilidad de éxito de reservación. Consecuentemente, cuanto más dure un programa, más probable es que la reservación de un bloque de memoria falle. Esta fuente potencial de falla hace imposible ofrecer una garantía de correctitud (desde la perspectiva de manejo de memoria).

En segundo lugar, el manejo de memoria es origen de una gran proporción de errores, razón por la cual, una forma de evadir este potencial de error consiste simplemente en no usar memoria dinámica. Observemos que el uso de un “recolector de basura” (garbage collector) no evade este principio, sino que más lo bien justifica, pues la gestión de memoria basada sobre un recolector exhibe mucha más incertidumbre y tiene menos desempeño que un manejador tradicional.

Los errores de manejo de memoria serán brevemente tratados en § 3.4.3.3 (Pág. 207).

Programas diseñados para apartar sus recursos computacionales durante su inicialización, no sólo son más fáciles de asegurar correctitud, sino que tienden a exhibir más desempeño.

En función de esta regla, ¿nos está vedado el uso de memoria dinámica? La respuesta es relativa. Notemos en primer lugar que los diseños de algoritmos y estructuras de datos que tratamos en este texto se acogen en su mayoría a esta regla. De hecho, hasta el presente hemos aplicado el principio fin-a-fin presentado en 1.4.2, cuya observancia conduce a, primero, abstraer sin considerar el manejo de memoria; como en efecto lo hemos hecho y haremos a través de este texto. Los TAD dependientes de memoria (con prefijo en nombre

<sup>16</sup>A menudo, la expresión es “lazo infinito”. Sin embargo, en matemática, así como en otros dominios, el adjetivo infinito se usa en el sentido de proporción, sentido para el cual, en la opinión del autor, no encaja el tiempo. Para el tiempo en sí existe el adjetivo “eterno”.

de clase “Dyn”), derivan de una versión independiente del manejo de memoria.

Hay problemas cuyas soluciones son “largas” en duración y para los cuales es ineludible el manejo de memoria. En esta clase de problemas encajan los grafos y otras clases de aplicaciones.

#### 4 MÁXIMO, 60 LÍNEAS POR RUTINA

La visión y comprensión humana de una rutina como una unidad de programa se remite a lo visible en una hoja de papel, o sea, aproximadamente 60 líneas. Rutinas que excedan esta longitud se fragmentan en varias hojas (o pantallas) y, por supuesto, tienen más probabilidad de fragmentar la comprensión.

#### 5 MÍNIMO DOS INVARIANTES (O ASERTOS) POR FUNCIÓN

Una invariante es una condición que siempre debería de cumplirse. Su incumplimiento denotaría, entonces, un error.

Las estadísticas industriales de prueba de programas indician ocurrencia de error entre 10 y 100 líneas de código. Por tanto, el uso de invariantes como precondiciones, postcondiciones e invariantes que se deben mantener en lazos, aunado a la regla inmediatamente anterior, debe favorecer la calidad del código por disminución de las probabilidades de ocurrencia de error y de aumento de su detección.

Las invariantes no afectan el efecto de una rutina, aunque sí su desempeño. Por esta razón, mecanismos de invariantes pueden deshabilitarse o hacerse de alto rendimiento. Más adelante, en § 3.4.3.3 (Pág. 204), hablaremos sobre el uso de invariantes.

#### 6 DECLARE E INICIALICE OBJETOS LO MÁS CERCA POSIBLE DE SU PRIMER USO

La corrupción de datos es otra de las ocurrencias principales de error. ¿Cuántas veces no nos hemos encontrado en la situación “misteriosa” en la que una variable no tiene el valor esperado?

Una de las virtudes del ocultamiento de información es que limita la corrupción al ámbito de uso del dato. Si se trata de un TAD, bien diseñado, por supuesto, la corrupción de uno de sus atributos sólo puede deberse a uno de sus métodos. Algo parecido en el ámbito de los objetos de uso interno de una rutina.

Por tanto, declarar un objeto lo más cerca posible de su primer uso, favorece que el objeto sólo exista en su ámbito de uso. Por otra parte, la inicialización inmediata a la declaración elimina la posibilidad de corrupción entre la declaración y su primera utilización.

#### 7 SIEMPRE VERIFICAR VALOR DE RETORNO DE UNA FUNCIÓN, ASÍ COMO SUS PARÁMETROS DE ENTRADA Y SALIDA

En palabras más precisas, cada punto invocante de una función debe verificar pertenencia al dominio del valor de retorno así como de sus parámetros de salida. Del mismo modo, cada función debe verificar que los parámetros recibidos también estén dentro de su dominio.

#### 8 MINIMIZE EL USO DEL PREPROCESADOR A INCLUSIÓN DE ARCHIVOS Y ESPECIFICACIÓN DE MACROS SIMPLES

Aunque los preprocesadores son muy poderosos como medios *ad hoc* para extender el

lenguaje, no son compiladores. Un preprocesador no verifica el sistema de tipos y, por añadidura, puede generar instrucciones que compilan, que corrientemente funcionen, pero que están totalmente ocultas a la mirada del programador.

Consecuentemente, se debe evitar la concatenación de nombres, los parámetros variables y los macros recursivos. La excepción de la regla es para aspectos que no pueda implantar el lenguaje, en cuyo caso se debe usar paréntesis por cada unidad de expansión del macro.

Recordemos que C y C<sup>++</sup> tienen funciones inline que son equivalentes en funcionalidad y desempeño a la mayoría de los macros, con el valor añadido de que usan el sistema de tipos del compilador.

#### 9 Restrinja el uso de apuntadores a no más de un nivel de dereferencia

Una dereferencia es un acceso a lo que contiene el puntero. En este sentido, la regla establece que no se pueden usar dos o más dereferencias, por ejemplo, algo como `**ptr = ...`

Para comenzar, más de un nivel es complicado de comprender aún para los más virtuosos programadores. En segundo lugar, también es difícil de estudiar por los analizadores estáticos.

Por supuesto, hay casos en que la declaración de punteros a punteros es necesaria, pero esto no acarrea necesariamente violación de la regla. Para un ejemplo de seguimiento de esta regla, véase la implantación de `DynArray<T>` presentada en § 2.1.4 (Pág. 34).

#### 10 Siempre compile con todos los alertas del compilador habilitados y use analizadores de código

La mayor parte de los compiladores modernos efectúa verificaciones de errores típicos. ¿Por qué razón habrían de descartarse esas verificaciones? La cruda respuesta es que muchos programadores son malcriados y prefieren saltar un error o alerta del compilador con tal de proseguir, con más probabilidad de error en el futuro, su proyecto. El mismo razonamiento aplica para los analizadores estáticos o dinámicos.

A veces ocurre lo que se denomina “falso positivo”, es decir, un reporte o alerta de un error que no existe. Por lo general, estos reportes ocurren en combinaciones de código intrincadas las cuales, a efectos de la claridad, es preferible modificarlas para que el compilador no arroje mensajes de alerta o de error.

#### Las preguntas actitudinarias de Van Cleck [173]

A través del devenir de la programación se ha observado que tras la corrección de un error se tiende a encontrar otro relacionado o uno nuevo introducido por la misma corrección. También se ha notado una resistencia “natural”, viciosa, por parte del programador promedio, en mirar minuciosamente el origen del error. Tal actitud es viciosa porque no basta con corregir el error, sino, en caso de que detrás del mismo subyazca una mala actitud, encontrar en qué consiste tal actitud de modo que ésta pueda enmendarse y, consiguientemente, no repetir la misma clase de error.

Tom Van Cleck, uno de los principales desarrolladores de Multics [31, 134], quizá el hito más importante en sistemas operativos, se plantea tres preguntas [173] que un desarrollador siempre debe hacerse cada vez que detecte un error y que lo encaminan a descubrir la actitud originaria de los errores.

1. ¿Se encontrará el error o la misma clase de error en otra parte del programa?

Plantearse esta pregunta conlleva establecer un patrón para el error y, en función éste, buscar en el resto del programa otras apariciones del patrón. Los hallazgos del patrón son lugares probables de repetición del error.

2. ¿Cuáles otros errores estarán relacionados con el encontrado o con su corrección?

Partiendo de la posibilidad de que la corrección introduzca un nuevo error, abordar esta pregunta confronta plantear una corrección e interrogar qué sucederá una vez que ésta se haga. A su vez, esto pasa por demostrar rigurosamente que la corrección sea correcta; o sea, que no contenga algún error.

3. ¿Qué se debería hacer para prevenir cometer de nuevo el error?

Una vez respondidas las dos preguntas anteriores se debe establecer, en función del patrón encontrado, un protocolo que evite repetir el cometimiento del error.

Quizá sea esta la pregunta más enriquecedora para el programador, pues su abordaje plantea una enseñanza.

### 3.4.3.2 Análisis estático

#### Compilador y sistema de tipos

El sistema de tipos del lenguaje de programación, en conjunción con el compilador, efectúan un enorme trabajo de verificación de correctitud al validar la correspondencias del programa respecto al sistema de tipo del lenguaje. Como ejemplo trivial, pero notable, consideremos:

```
int x;
string s;

s = x; // Violación de tipo. ERROR DE COMPILACIÓN
```

Los errores de compilación han sido causa de ofuscación y maldición entre los programadores noveles y algunos avanzados. ¿Cuántos de nosotros hemos permanecido alguna vez bloqueados porque un programa no nos compila?

Una mirada más crítica del problema anterior nos permite aprehender que, salvo un error del compilador o del lenguaje de programación, cuestión improbable pero factible, un error de compilación tiene la gran bondad de que nos fuerza a enfrentar el cometimiento de un error como programadores. En el ejemplo anterior, no tiene sentido asignar un entero a una cadena de caracteres.

Por lo general, los compiladores parametrizan sus modos de detección y tratamiento de errores. Consultese el manual del compilador **GNU gcc** para más detalles. En todo caso, como ya dicta la décima regla de Holtzmann, compílese con toda la detección de errores y alertas habilitados. No se prosiga hasta que el compilador deje de arrojar alertas y errores.

#### Analizadores externos

Resulta que la búsqueda de errores en el código fuente de un programa se puede automatizar para validar que éste siga los estándares de codificación del lenguaje u otros

establecidos por el grupo trabajo. Quizá el ejemplo más reputado de esto sea lint [87], un analizador estático de fuentes para el lenguaje C que ha devenido esencial para el desarrollo de programas en C, habida cuenta de la debilidad del sistema de tipos de este lenguaje. lint aún es ampliamente usado.

Hay unas cuantas herramientas, en su desafortunada mayoría de uso comercial, que hacen análisis estático de correctitud de un programa. Lamentablemente, aún existen pocas herramientas libres que hagan un análisis estático al mismo nivel que otras comerciales. Podemos citar, empero, a LC-LINT, complementario del célebre lint [87], y Flawfinder [2] y its4 [175] orientados a la búsqueda de errores de seguridad.

### Metacompilación

Metacompilación consiste en ofrecer un compilador extensible en el sentido de que el usuario pueda añadir nuevas reglas de compilación según el conocimiento aplicativo del usuario. Esta idea tiene mucho sentido porque el compilador posee la maquinaria necesaria para la verificación automática, pero no posee el conocimiento sobre el tipo de aplicación. Por el contrario, el codificador sí posee aquel conocimiento, pero adolece de la falta de aquella maquinaria. La idea se basa, por tanto, en el principio fin-a-fin.

Por ejemplo, un usuario podría especificar que la asignación entre punteros está prohibida y que para toda llamada al operador new sobre un puntero debe encontrarse por alguna parte otra llamada al operador delete. Del mismo modo, tal extensión al compilador podría permitir verificar estáticamente que aquel new ocurra antes que el delete.

Se pueden aprovechar también las propiedades transformacionales del compilador para ejecutar acciones semánticas. Por ejemplo, que el compilador implante la primera regla de Holtzmann y desplace la gestión de memoria a la fase de inicialización.

Reportes de Engler *et al* [45, 46] indican que la meta-compilación será una de los principales medios de verificación de correctitud en los años venideros. Mediante esta técnica, Engler *et al* [45, 46] ya han encontrado miles de errores en Linux, OpenBSD y muchos y complejos sistemas comerciales.

Lamentablemente, los programas resultantes de las investigaciones de Engler et at [45, 46] no son de libre acceso.

### Exploración del espacio global de estado

Una técnica grandiosa de verificación de correctitud requiere traducir el programa a una máquina de estado finito que represente los diferentes estados de los programas. Puesto que no disponemos de espacio para explicar formalmente qué es un autómata, lo intuiremos pictóricamente mediante un simplísimo autómata, mostrado en la figura 3.8, que calcula el menor entre dos números  $x, y$ . Cada círculo corresponde a un “estado”, mientras que cada flecha es una “transición” entre dos estados. El cambio de estado ocurre cuando se ejecuta la instrucción asociada a la transición. El estado inicial se denota con una flecha aislada que le llega, mientras que uno final se expresa con otra flecha que le sale.

Una vez obtenido el (o los) autómata(s), un simulador se encarga de explorar todas las combinaciones de estados posibles en la búsqueda de errores sintácticos y semánticos. Un error sintáctico se considera como un error general a cualquier clase de programa; por ejemplos, que un programa jamás ejecute alguna porción de su código, o que el programa

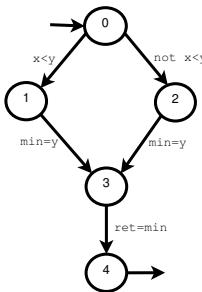


Figura 3.8: Un autómata que calcula el menor entre dos números

entre en un estado de bloqueo eterno en el cual se espera por un evento que jamás ocurrirá (deadlock), o que se entre en la ejecución de un lazo infinito. Un error semántico ataña al fin del programa y para especificarlo se utiliza por lo general lógica temporal<sup>17</sup> o autómatas de predicados (o de Büchi)<sup>18</sup>. Al proceso de verificación semántica mediante lógica temporal (o autómatas de Büchi) se le denomina “Verificación de modelo” (“Model Checking”).

Así las cosas, la verificación por este método parece bastante simple. Pero en la realidad, los autómatas resultantes de un programa real son extremadamente grandes y no siempre existe un método determinista para traducir programas hacia autómatas.

Este método de verificación es mucho más interesante cuando se verifican programas concurrentes; es decir, programas compuestos por varios programas que se ejecutan “simultáneamente”. Pero aquí se requiere construir un “autómata global” compuesto por la combinación de los estados de todos los autómatas.

Teóricamente, la exploración global del espacio de estado es capaz de detectar cualquier error sintáctico y todos los semánticos en función del fin definido. En la práctica, empero, existe el problema de que el espacio de estado del autómata global es explosivo  $\mathcal{O}(2^n)$ . Hay técnicas, basadas en desarrollos teóricos, que permiten lidiar con este problema, siendo las más importantes las siguientes:

1. Algoritmos simbólicos: la idea consiste en substituir el autómata, o parte de él, por una fórmula lógica. Por ejemplo, para el autómata anterior sería posible substituir las transiciones desde el estado 1 hacia el 4 por una transición cuyo predicado sería  $x < y \text{ or } \text{not } x < y, \text{ret}=\text{min}$ , pues el flujo de ejecución nunca se detendrá a causa de las transiciones suprimidas.
2. Reducción por orden parcial: esta técnica, que se deriva de algunos resultados matemáticos, consiste, muy *grosso modo*, en sustituir largas cadenas de transiciones secuenciales, que no tienen bifurcaciones, por una sola transición. El orden parcial se aplica dinámicamente y recursivamente cuando se trata del autómata global.

Por ejemplo, el autómata de la figura 3.9-a puede substituirse por el de la figura 3.9-b reducido a dos estados y una transición compuesta por las tres instrucciones del autómata extendido. Un simulador puede dinámicamente detectar y efectuar esta clase de reducción.

<sup>17</sup>La lógica temporal es una lógica de predicados cuyos cuantificadores consideran al tiempo [145].

<sup>18</sup>Un autómata de predicados es uno cuyas transiciones sólo contienen predicados lógicos relacionados con un autómata resultante del modelo [25, 115].

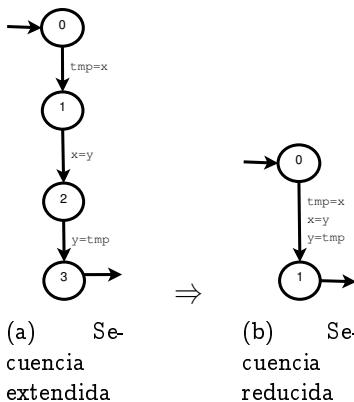


Figura 3.9: Ejemplo de reducción por orden parcial

3. Abstracción: es simple, esta técnica se basa en observar conjuntos de estados y “abstraer” su interfaz de manera general. El conjunto considerado se substituye por un autómata reducido cuyo rol es simular el resultado general.

Las técnicas anteriores permiten reducir, muy considerablemente, el espacio de estado. En añadidura, otra técnica, llamada “supertraza” [77], que examinaremos en § 5.3.2 (Pág. 434), basada en el uso de tablas hash sin detección de colisión, permite reducir todavía más el espacio global de estado. Lo anterior, aunado al conocimiento de las ciencias estadísticas, indica que es posible explorar “casi todo” o, “todo” [183], el espacio de estado si se ejecutan suficientes simulaciones aleatorias.

Quizá el exponente más popular de esta clase de verificación es spin/promela [75, 76, 74, 153]. promela es un lenguaje de especificación de programas (no un lenguaje de programación) y spin es su simulador.

El enfoque requiere que el programa se especifique en promela, lo cual añade un paso adicional y una fuente potencial de error al trasladar la especificación en promela hacia el lenguaje de programación. La necesidad de este enfoque se debe a que aún no se conoce generalmente cómo se traduce un programa en un lenguaje de programación hacia un autómata de estado finito.

spin es libre desde 1991.

Recientemente se han descubierto técnicas efectivas para traducir un programa en un lenguaje de programación a un autómata según la naturaleza de la aplicación. Esto permite automatizar el proceso de verificación sin necesidad de pasar por la fase de modelización en otro lenguaje.

### 3.4.3.3 Análisis dinámico

#### Sistema operativo

El hardware y el propio sistema operativo tienen medios para detectar errores durante la ejecución de programas. En el caso del hardware, por ejemplo, puede detectarse una división por cero. En el caso combinado de hardware y sistema operativo, puede detectarse una violación de memoria (segmentation fault), error también causante de desesperaciones y maldiciones en los programadores novatos, pero en realidad muy afortunado, pues indica el cometimiento de un error y la asunción de su acción correctiva.

### Invariantes

Una invariante o aserto es un predicado lógico que se introduce en algún punto de un programa y cuya certitud verifica si se cumplen las premisas de ejecución supuestas por el diseñador del algoritmo o el programador.

La idea es bastante simple: el predicado se evalúa sólo cuando el flujo de ejecución pasa por la invariante. En ese momento se evalúa el predicado y, si éste es falso, entonces se notifica al programador mediante algún evento, por lo general, el aborto del programa. El evento le permite aprehender que, (1) o las condiciones requeridas para ejecutar el flujo en cierto punto no se están cumpliendo, o (2) él cometió un error de razonamiento. El cualquiera de los dos casos se está en presencia de un error.

La manera más simple de interpretar una invariante es en forma de precondición o postcondición de una rutina. Notemos que esta idea puede aplicarse a cualquier flujo de ejecución, es decir, al principio de un flujo establecemos precondiciones que deben cumplirse para que sea ejecutado correctamente. Del mismo modo, luego de la ejecución del flujo, el estado habrá cambiado de tal forma que deben cumplirse algunas postcondiciones. Cada precondición o postcondición no es otra cosa que una invariante.

Algunos ejemplos pueden aclarar la idea.

Si diseñáramos una primitiva para calcular la raíz cuadrada  $\sqrt{x}$  de un número real, entonces es deseable poner como precondición el predicado  $x \geq 0$ . La violación de esta precondición evidenciaría que en alguna parte, antes de invocar a  $\sqrt{x}$ , se cometió un error. Del mismo modo podríamos colocar como postcondición  $\sqrt{x^2} = x$ , cuya violación, a condición de que la precondición no haya sido violada, probablemente indicaría un error en la instrumentación de  $\sqrt{x}$ .

Un eventual cuestionamiento a las invariantes es que éstas consumen tiempo de ejecución. Consecuentemente, su uso excesivo puede impactar el tiempo de ejecución. Notemos que este coste, aunque constante, fácilmente puede tornarse oneroso si, por ejemplo, está contenido dentro de un lazo. Por esa razón, muchos programadores supeditan el uso de invariantes al código de desarrollo y no al código de producción. Por lo general, en C y el C++, esto se automatiza mediante macros condicionales. La manera típica es compilar invariantes si el macro DEBUG está definido.

En muchas ocasiones es importante dejar las invariantes en código de producción, pues arrojan indicios de ocurrencia de error. En estos casos no se debe abortar el programa.

La biblioteca estándar C contiene una función llamada assert(predicado), cuya función es verificar una invariante. No obstante, esta primitiva es costosa en tiempo de ejecución, razón por la cual su uso es cuestionable para código productivo.

Existen bibliotecas especializadas en el uso de invariantes. Quizá una de las más populares sea GNU nana [144], cuyas bondades, respecto al assert() tradicional, pueden resumirse en:

1. Diseñada para la eficiencia en duración y espacio. De hecho, la verificación es tan eficiente que muchas veces no vale la pena eliminar invariantes en el código de producción.
2. La acción a tomar ante una violación de invariante es configurable. Se puede, entre otras cosas:
  - (a) Modificar la acción a tomar ante una violación de invariante; por ejemplos, abortarse, reiniciar, enviar a un depurador o, simplemente, reportar y continuar.
  - (b) Habilitar o deshabilitar selectivamente la evaluación de invariantes, tanto durante la compilación como durante la ejecución.

3. Enriquecida con otras clases formales primitivas que permiten verificaciones de invariantes más complejas.
4. Medición precisa y portátil del tiempo de ejecución.

Veamos algunas de las primitivas de GNU nana más importantes para manejar invariantes.

### Invariantes basadas en asertos

De esta clase de invariantes, la principal es:

```
void I(pred)
```

La cual verifica que pred sea cierto, en cuyo caso negativo imprime un mensaje de error y aborta.

Una forma condicional de evaluar una invariante es mediante:

```
void IG (bool pred, bool cond)
```

La cual evalúa el predicado pred como invariante sólo si el predicado cond es cierto. Esta forma permite definir casos especiales.

A veces es necesario mantener estado previo, esto ocurre por lo general con las post-condiciones. En este caso son de interés las siguientes primitivas:

- void ID (instrucción): permite ejecutar una instrucción, normalmente, una declaración de variable.
- void IS (Asignación): permite realizar un asignación a una variable declarada bajo ID().

Aunque ID() e IS() son sintácticamente similares, el primero debe usarse para declarar y el segundo para asignar, pues el código de generación e identificación asume esta funcionalidad.

- void ISG (instrucción, bool pred): efectúa una asignación condicional a que el predicado pred sea cierto.

Por ejemplo, un código que calcula la raíz cuadrada de un número real podría plantearse del siguiente modo:

205 *⟨raíz cuadrada 205⟩≡*

```
float raíz_cuadrada(const float & x)
{
 I(x >= 0); // precondition exigiendo que x sea positivo

 // cálculo de la raíz de x, el cual se guarda en ret_val

 I(ret_val*ret_val == x); // resultado igual a x
 ID(float ret_plus = ret_val + 1); // variable ret_plus
 I(ret_val < ret_plus*ret_plus); // menor que (ret_val+1)^2
}
```

Desde el lenguaje C es posible poner una secuencia de instrucciones como una sola separadas por el operador coma. Esto es importante de resaltar en este contexto porque es lo que permite englobar una secuencia completa de instrucciones en alguna primitiva de invariantes; por ejemplo:

```
ID(int i = 0, j = x + y);
```

### Verificación por cuantificadores de lógica de predicados

Derivado de la célebre “programación por contratos” [124], GNU nana ofrece verificaciones más refinadas bajo la forma de cuantificadores lógicos. Entre los más importantes podemos indicar:

- `bool A(inicio,condición,iteración,pred)`: correspondiente al cuantificador “para todo” ( $\forall$ ), el cual se corresponde con:

```
for (inicio; condición; iteración)
 if (condición)
 return false;
 return true;
```

- `bool E (inicio,condición,next,pred)`: correspondiente al cuantificador de existencia y que se define así:

```
for (inicio; condición; iteración)
 if (pred)
 return true;
 return false;
```

Es decir, es cierto si al menos en una de las iteraciones se cumple el predicado.

- `long C (inicio,condición,next,pred)`: contador de ocurrencias de un predicado consistente en:

```
int count = 0;
for (inicio; condición; iteración)
 if (pred)
 ++count;
return count;
```

- `bool E1 (inicio,condición,next,pred)` : verifica que el predicado se cumpla exactamente una vez. Se puede interpretar de la siguiente manera:

```
bool seen = false;
for (inicio; condición; iteración)
 if (pred)
 if (seen)
 return false;
 else
 seen = true;
return seen;
```

Como ejemplo, consideremos verificar la pertenencia exclusiva de un elemento específico a una lista enlazada:

207  $\langle \text{pertenencia a lista enlazada } 207 \rangle \equiv$   
 $E1(\text{typename DynDlist::iterator<int>} \text{ it}(l), \text{ it.has\_current}(),$   
 $\text{it.next}(), \text{ it.get\_current}() == x);$   
 Uses DynDlist 85a.

En este punto es menester señalar que GNU nana posee los mismos tipos de cuantificadores señalados para contenedores de la biblioteca estándar C++.

GNU nana es una biblioteca mucho más rica, tanto en especificación de invariantes, como en otras funcionalidades. Léase detenidamente el manual [144] para más información.

### Pruebas

Una prueba es un experimento consistente en ejecutar el programa para verificar si funciona correctamente. *Grosso modo*, existen dos tipos de prueba: (1) de caja negra, en la cual se establecen entradas y salidas correctas a verificarse durante el experimento; y (2) de caja blanca, en la cual se determinan entradas que verifiquen que el flujo pase por todas las partes del programa.

Por tradición, las pruebas, en particular las caja -negra, las hacen personas distintas al codificador llamadas “probadores” (testers).

La disciplina impartida por Maguire [114] (ver § 3.4.3.1 (Pág. 196)) hace costumbre que un programador siempre haga pruebas caja blanca para todo su código.

Existe toda una ciencia para probar programas cuyo ámbito y extensión están fuera de alcance en este texto. Además de los dos autores anteriores, la mejor referencia que se este redactor conoce para el arte de las pruebas se titula “*The Art of Software Testing*” por G. J. Myers [128].

### Generación automática de prueba

Engler *et al* han desarrollado una técnica, primigeniamente basada en análisis estático del código fuente, consistente en generar entradas al programa que ejecutan todo los caminos de ejecución posibles del programa [186]. Aparte de encontrar las entradas que cubren todos lo casos posibles, se generan entradas que someten al programa a máximos esfuerzos según desempeño, condiciones críticas de concurrencia, seguridad, etcétera, que han permitido encontrar errores en manejadores de sistema de archivo en producción desde hace años.

### Manejo de memoria

El manejo de memoria dinámica es una fuente de error tan común que merece consagrarse esta sub-sección a clasificar y describir las clases de error.

### Fugas de memoria (memory leaks)

Una “fuga de memoria” o, en inglés, “memory leak”, es un bloque que se reserva y que jamás en la vida del programa se libera. Por ejemplo, si tenemos la siguiente ineficiente, inelegante e incorrecta versión del swap() entre arreglos:

```
void array_swap(int * a, int * b, const size_t n)
{
 int * tmp = new int [n];
```

```

 for (int i = 0; i < n; ++i)
 tmp[i] = a[i];
 for (int i = 0; i < n; ++i)
 a[i] = b[i];
 for (int i = 0; i < n; ++i)
 b[i] = tmp[i];
}

```

entonces, cada vez que se invoque a `array_swap()` se dejará apartado, “para siempre”, un arreglo `tmp` de `n` elementos. Si `array_swap()` se invoca a menudo, entonces el programa colapsará en poco tiempo por falta de memoria.

La regla para evitar este error es que todo bloque de memoria debe liberarse inmediatamente después de que éste ya no se requiera.

### Acceso fuera de bloque

Menos frecuente que una fuga de memoria, también común y, a menudo, más grave, un acceso fuera de bloque consiste en la lectura o escritura de una zona de memoria que no se ha apartado. El caso más típico es escribir justo después del fin de un bloque de memoria reservado con `malloc()` o `new`. Por ejemplo:

```

int * tmp = new int [n];
for (int i = 0; i <= n; ++i)
 tmp[i] = a[i];

```

Este código con certitud escribirá un entero de más en el arreglo `tmp`. En el mejor de los casos, ocurrirá un “segmentation fault”; en el peor, el error pasará temporalmente desapercibido hasta que tiempo después (quizá mucho después) presente síntomas también erráticos en el sentido de que son variables, pudiendo ser, inclusive, de índole “heisenbergiano” (*heisenbug*)<sup>19</sup>.

La regla para evitar este error es algo vaga: siempre revise rigurosamente el acceso mediante un apuntador. Por esa razón son buenas costumbres:

1. Usar nombres de punteros que indiquen claramente que se trata de un puntero. El mecanismo de nombramiento favorito es sufijar el nombre con `_ptr`; de este modo, una herramienta estilo `grep` puede encontrar las ocurrencias de uso de un apuntador y permitir al programador, o a una herramienta automatizada, analizar la correctitud.
2. Aísle el uso de punteros como arreglos en operaciones genéricas, ergo reusables, que concentren la verificación de correctitud en una sola porción de código. Un ejemplo representativo lo constituyen las funciones `fill_dir_to_null()`, `release_segment()`, entre otras especificadas en § 2.1.4.2 (Pág. 40).
3. Evitar el uso de más de un indirecccionamiento por puntero. Por ejemplo:

```
**int_ptr = 5;
```

no estaría permitido. La excepción a esta regla es cuando se traten arreglos como punteros y tengamos arreglos de arreglos, tales como lo hicimos, por ejemplos para

---

<sup>19</sup>Por el célebre físico alemán Werner Heisenberg quien descubrió el principio homónimo acerca de la imposibilidad de observar el estado real de una partícula atómica, pues la misma observación cambia el estado.

el método `access()` de `DynArray<T>` (§ 2.1.4.3 (Pág. 45)) o con los arreglos multidimensionales (§ 2.2 (Pág. 58)).

#### Dirección de devolución inválida

Una invocación exitosa a `malloc()` o a `new` retorna un puntero al bloque reservado. El mecanismo de devolución, mediante `free()` o `delete`, debe recibir exactamente el mismo valor de puntero que entregó `malloc()` o `delete`. Liberar una dirección de memoria que no haya sido retornada por el manejador de memoria es, pues, un serio error de programación que compromete el estado de la aplicación y cuyas consecuencias son inciertas y, por lo general, permanecen indetectables durante algún tiempo.

Entregar una dirección inválida al manejador de memoria es un error más común de lo que podría pensarse. Quizá el caso más frecuente ocurre con punteros sobre clases derivadas y algunas transformaciones resultantes del “casting”. Otra posibilidad sucede cuando se manejan multiestructuras, por ejemplo, una multilista.

#### Verificadores dinámicos de memoria

Los errores de memoria fueron un perenne dolor de cabeza en el pasado hasta que aparecieron programas, en su mayoría en forma de bibliotecas, para detectarlos, siendo los más populares entre ellos `electric fence` [1] y `dmalloc` [179]. Con este esquema se requiere ayuda del preprocesador, la cual no está completamente disponible en C++ y el encadenamiento de una biblioteca, la cual intercepta las llamadas a `malloc()` y `free()`.

Un nuevo y versátil programa de verificación ha aparecido recientemente: `valgrind` [130, 129], el cual no requiere biblioteca, sino que interactúa sobre el ejecutable resultante.

Los mecanismos de detección son los mismos. En primer lugar, cada dirección de bloque de memoria, junto con su posición exacta de reservación, línea y nombre del archivo, se anota en una tabla de manera tal que al final del programa las entradas en la tabla corresponden a fugas de memoria. Del mismo modo, una entrega de dirección inválida puede detectarse porque la dirección no se encuentra en la tabla.

Cuando se solicita un bloque, se aparta un poco más de manera tal de poner “rejas”, las cuales son zonas con valores especiales, a cada lado del bloque, de poca posibilidad de escritura, y cuya pictorización es como sigue:



Al liberar un bloque de memoria se revisa el valor de la reja, y si éste es diferente al de inicio, entonces puede ser síntoma de que se ha escrito por fuera de un bloque.

Las verificaciones anteriores, en particular, las rejas, no garantizan detección para todos los casos.

El enfoque de `valgrind` es algo similar a los casos anteriores pero no intercepta la biblioteca en sí. En su lugar, `valgrind` intercepta, sobre el ejecutable, las llamadas críticas a reservación de memoria (`malloc()`, `free()`, `sbrk()`, `new`, `delete`, ...) y, por añadidura, intercepta los accesos a memoria y otras operaciones delicadas. Luego, ejecuta el programa y, durante la ejecución va verificando y reportando los eventuales errores. `valgrind` ha disminuido grandiosamente el tiempo de verificación en lo que concierne al manejo de memoria y otras clases de errores.

Bajo los dos enfoque presentados, el programa se ejecuta considerablemente más lento, lo que en circunstancias especiales puede hacer su uso complicado.

### Depuradores

A veces, un error no se detecta por los analizadores estáticos o dinámicos sino que se conoce su presencia porque el programa arroja resultados incorrectos. Su búsqueda es un arte que depende de la experiencia, instinto e intuición del programador para encontrar partes que ofrezcan pistas acerca del error. En este orden de ideas, una de las clases de programas más útiles que existen son los depuradores, programas especializados en observar la ejecución del programa y relacionarla al código fuente.

Uno de los depuradores más célebres es el GNU gdb [161], junto con algunos de sus frontales (“front-ends”), tales como el ddd [189].

### 3.5 Eficacia y eficiencia

A estas alturas del discurso, debemos tener claro que la eficacia prima a la eficiencia. Al final de cuentas, ¿de qué vale algo muy eficiente si no es correcto?. Pues bien, en algunos problemas, la eficacia puede comprometer a la eficiencia y viceversa: si se hace correctamente, entonces puede ser muy ineficiente y, posiblemente, inaplicable. Para aprehender la cuestión consideremos un ejemplo magistral mostrativo de las vicisitudes involucradas en la resolución de un problema y los compromisos que a menudo se presentan entre correctitud (eficacia) y eficiencia. El ejemplo está basado en Steven Skiena de su excelente libro *The Algorithm Design Manual* [159].

Consideremos un conjunto  $\mathcal{P} = \{p_1, p_2, p_3, \dots, p_n\}$  de  $n$  puntos en el plano. Cada punto se define por sus coordenadas cartesianas  $(x, y)$ . El problema consiste entonces en encontrar un orden de visita entre todos los puntos de manera tal que la longitud del recorrido sea mínima.

Hay varias maneras de abordar la solución de este problema. Quizá la más candida sea mediante la heurística del “vecino más cercano”: dado un punto  $p_i$ , el próximo punto a conectar es el más cercano a  $p_i$ .

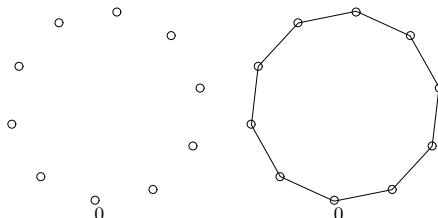
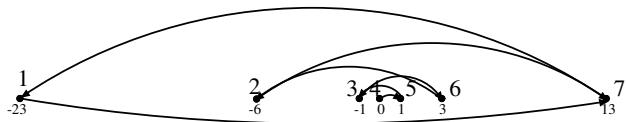


Figura 3.10: Una disposición de puntos y la solución del ciclo más corto mediante la heurística del vecino más cercano [159]

Parafraseando a Skiena [159], un algoritmo basado en el “punto más cercano” es bastante simple, fácil de comprender y de implementar. Podemos decir también que el algoritmo es muy eficiente, pues su complejidad es  $\mathcal{O}(n)$ . Lamentablemente, el algoritmo no es correcto, y para percatarnos de ello consideremos este contraejemplo, comenzando desde el punto 0 planteado por el propio Skiena [159]:



Con esta disposición de puntos la heurística del punto más cercano, paradójicamente, puede encontrar secuencias muy largas y no las más cortas.

Otra heurística más compleja, llamada “el par de puntos más cercano”, también señalada por Skiena [159], trabaja con los puntos extremos de caminos parciales. El algoritmo maneja un conjunto de caminos parciales (inicialmente de un solo punto). El algoritmo consiste en determinar y conectar el par de puntos extremos más cercanos de modo tal que la conexión sea un camino y no se forme un ciclo. Esta heurística conduce a un algoritmo menos eficiente que con la heurística del camino más corto, pero con mejores resultados para casos más diversos. Lamentablemente, un algoritmo basado en esta heurística no es absolutamente correcto para todas las entradas posibles.

La única solución correcta conocida está dada por el siguiente algoritmo:

**Algoritmo 3.1 (Camino más corto entre un conjunto de puntos)** La entrada del algoritmo es el conjunto  $\mathcal{P} = \{p_1, p_2, p_3, \dots, p_n\}$ . La salida es una secuencia  $S = < p_1, p_2, \dots, p_n >$ .

El algoritmo requiere construir un conjunto  $\Omega = \{\omega_1, \omega_2, \dots, \omega_m\}$  conformado por todas las permutaciones posibles de los puntos de  $\mathcal{P}$ .

1. distancia =  $\infty$
2. resultado =  $\emptyset$
3. Sea  $\Omega$  el conjunto de todas las permutaciones de  $(\mathcal{P})$
4.  $\forall \omega \in \Omega$ 
  - (a) Sea  $c$  la distancia total obtenida de recorrer los puntos en el orden de la permutación  $\omega$
  - (b) Si  $c < \text{distancia} \Rightarrow$ 
    - i. distancia =  $c$
    - ii. resultado =  $\omega$

Todas las posibles caminos se consideran y sólo se escoge el de menor coste. El algoritmo en sí es simple, aunque no considera cómo construir el conjunto de permutaciones, lo cual no es una tarea muy sencilla. El algoritmo es, pues, correcto. Infelizmente, tal como se plantea, es tan ineficiente que sólo podría ejecutarse en computadores divinos. Para aprehender esto analicemos la eficiencia.

El número de exploraciones que efectúa el algoritmo 3.1 es igual a la cantidad de permutaciones que existen entre los  $n$  puntos, o sea,  $\mathcal{O}(2^n)$ , una eficiencia intratable aun para pocos puntos. A veces, la correctitud y la eficiencia representan contradicciones insalvables.

Aunque Skiena presenta este problema como un compromiso entre correctitud, eficiencia y facilidad de programación [159], lo cual, en cierta medida es correcto, discreparemos un poco y plantearemos nuestros argumentos bajo las siguientes observaciones:

1. La instancia del problema del vendedor viajero que estamos tratando es general en el sentido de que se trata de resolverlo para todos los casos posibles. Sin embargo, en la vida real, casi siempre se dispone de más información acerca de las clases de entrada. Por ejemplo, si fuera necesario resolver el problema para planificar soldaduras sobre una placa electrónica, entonces se sabe que las configuraciones de puntos conforman polígonos monótonos y que para éstos la heurística del vecino más cercano arroja soluciones correctas.
2. En otras ocasiones, la diferencia entre la solución óptima y una buena, heurísticamente encontrada, no es muy notable. En estos casos, el tiempo de desarrollo y ejecución de un algoritmo bueno puede ser menor que el de uno óptimo.

En algoritmos y sistemas, así como en otros dominios de la ingeniería, se pronuncia un proverbio que reza que “lo mejor es enemigo de lo bueno”.

De lo anterior se desprende que la lección sigue siendo que la eficacia prima a la eficiencia.

### 3.5.1 La regla del 80-20

En 1906 el ingeniero franco-italiano Vilfredo Pareto señaló que en Italia el 80% de la riqueza se distribuía “muy injustamente” entre el 20% de la población. Su observación fue corroborada en algunas otras latitudes y contextos bajo “el Principio de Pareto”.

En algunos contextos en los cuales se habla de eficiencia, el Principio de Pareto ha sido aplicado para caracterizar una relación entre recursos de consumo o producción y los agentes consumidores o productores. Hay indicaciones empíricas de que el principio aplica a los sistemas computacionales. En efecto, se ha observado, entre otras cosas, que:

1. El 80% de la memoria consumida por un programa es usada por el 20% del programa.
2. El 80% del tiempo de ejecución es consumido por el 20% del programa.
3. El 80% de los accesos a disco es realizado por el 20% del programa.
4. El 80% del esfuerzo de desarrollo de un sistema es realizado sobre el 20% del mismo.

En programación, a este patrón se le denomina la “*regla del 80-20*”. La regla ha sido sólidamente comprobada a lo largo de la historia de la programación y a través de diversos sistemas: operativos, de bases de datos, y variadas aplicaciones.

La regla 80-20 no es una simple razón numérica o proporción. Se trata del valiosísimo conocimiento que representa el saber que la eficiencia de un programa está determinada por una parte correspondiente al 20%. Por tanto, cuando se desea mejorar un programa, o sea, de hacerlo más eficiente, debemos hacerlo sobre aquel 20% y no perdernos en mejorar, fútilmente, el restante 80%.

### 3.5.2 ¿Cuándo atacar la eficiencia?

Lo anterior reafirma que la eficacia prima a la eficiencia, pues hasta que no logremos el efecto, no podemos identificar ese 20% del programa sobre el cual deberíamos de concentrarnos si deseásemos mejorar la eficiencia. Más aún, si elaboramos programas eficaces

que satisfacen las expectativas, ¿vale la pena mejorar su eficiencia? Para responder esto debemos mejor preguntarnos: si tenemos un programa eficaz (correcto), ¿cuándo es que hay que atacar la eficiencia? Hay varias condiciones, siendo las más comunes las siguientes:

- El programa resultante no acomete la solución en una duración aceptable para su aplicación: diseñamos un programa, lo codificamos, comprobamos que es correcto, pero cuando lo ejecutamos nos percatamos de que su duración de ejecución no cumple las expectativas.
- Cambio en las condiciones ambientales de uso del programa: tenemos un programa correcto que cumple las expectativas. Un día acontece un cambio, por ejemplo, se aumenta la escala de entrada y el programa deja de ser eficiente.
- Uso del programa como un componente de otro programa o sistema: tenemos un programa correcto y eficiente y se nos plantea reusarlo como parte de otro programa. Sin embargo, el componente en cuestión no tiene la suficiente eficiencia para asegurar la eficiencia global del programa.

Si un programa dado encaja en alguna de estas condiciones, entonces puede adverarse la necesidad de hacerlo mas eficiente. Si, al contrario, el programa satisface todas las expectativas, entonces es preferible evitar el “efecto del segundo sistema” [24], el cual, citando a Brooks: “... is the most dangerous system a man ever designs” [24]<sup>20</sup>.

### 3.5.3 Maneras de mejorar la eficiencia

Una vez adverado que requerimos hacer más eficiente un programa, tenemos varias alternativas, combinables entre sí, para mejorar la eficiencia. Entre las más usuales tenemos:

1. Identificación y mejoramiento del 20%: cuenta habida del Principio de Pareto, eso tiene mucho sentido, pues eso beneficiaría al 80% del programa.
2. Cambio de algoritmo: a veces, sobre todo debido a la escala de la entrada, es el propio algoritmo la fuente de ineficiencia. En este caso puede adverarse necesario diseñar un algoritmo más eficiente computacionalmente. Por ejemplo, tal vez un proceso crítico de un sistema use algún método sencillo de ordenamiento cuya complejidad es  $\mathcal{O}(n^2)$ ; en este caso, el método podría substituirse por alguno  $\mathcal{O}(n \lg n)$ , el cual es considerablemente más eficiente.
3. Cambio en la actitud de programación: otra posibilidad de ineficiencia puede deberse a una actitud por parte de los programadores que introduzca ineficiencia. A menudo, esto ocurre entre programadores noveles.

Un ejemplo notable y real constituye el pase de parámetros por valor en C++. En efecto, en este lenguaje, si no se tiene el cuidado adecuado, el coste del pase de parámetros por valor puede ser altísimo, pues cada invocación a función requiere una copia de cada objeto parámetro, la cual, según la índole del objeto, puede consumir bastante tiempo.

4. Cambio ambiental adrede: hace unos 20 años, cuando un programador quería lograr algunos efectos visuales especiales, aumentaba explícitamente la velocidad de reloj y

---

<sup>20</sup>“Es el más peligroso sistema que alguien puede diseñar”.

usaba una bolsa de hielo sobre el procesador para atenuar el calor producido. Aquella limitada y efímera técnica permitía predecir que procesadores más veloces -o adaptaciones físicas- darían solución al problema.

Un problema de eficiencia puede atacarse a “fuerza bruta” aumentando el poder computacional. Aparte de cambios físicos del estilo descrito en el párrafo anterior, hay varios más loables. En primer lugar, puede substituirse el compilador por uno que efectúe optimizaciones agresivas especiales. En segundo lugar, puede adquirirse hardware especializado o, simplemente, de mayor potencia; velocidad de procesador, cantidad de memoria, cantidad de disco, tecnología de almacenamiento, número de procesadores, etcétera.

Esta es, grosso modo, la clasificación de las maneras objetivas en que se puede mejorar la eficiencia. Lo demás, y lo que vendría, es de índole subjetiva y parte del corpus de estudio de este texto.

### 3.5.4 Perfilaje (profiling)

Dado un programa del cual se desea estudiar su eficiencia, ¿cuáles son las partes del programa que representan aquel 20% crítico? Aparte del conocimiento del programador, en la determinación de esta respuesta puede usarse una clase especial de programa llamado “perfilador” (“profiler”).

De manera elemental, un profiler es un programa que, coasistido por el compilador, muestrea la ejecución de un programa y recaba estadísticas sobre la posición de ejecución del programa. Por lo general, un profiler ofrece dos gráficos fundamentales:

1. El perfil plano, el cual muestra la duración consumida por cada rutina y la cantidad de veces que fue invocada.
2. El grafo de llamadas, el cual muestra, para cada función, cuáles fueron las funciones que la llamaron y cuántas veces esto ocurrió.

Siendo este enfoque de índole estadística, la precisión del profiler depende de la duración total del programa: a mayor duración, mayor precisión. A pesar de este tipo de imprecisión, este enfoque tiene la enorme ventaja de que no impacta demasiado sobre la duración permitiéndole ejecutarse a velocidades cercanas al tiempo real.

El principal exponente del profiler estadístico es GNU gprof. Hay varios front-ends gráficos que versatilizan la visualización de los perfiles arrojados por GNU gprof, siendo el más notable de ellos kprof. Valgrind también posee un excelente profiler estadístico.

Otro enfoque de perfilaje más determinista consiste en incluir contadores directos en cada rutina y medidores de tiempo. El enfoque es mucho más preciso que el estadístico, pero tiene el problema de que aumenta considerablemente el tiempo de ejecución haciéndolo inaplicable para programas con requerimientos de tiempo real. Un prototipo libre de esta clase de profiler es FunctionCheck [142].

### 3.5.5 Localidad de referencia

Por lo general, el acceso a los datos por parte de un programa exhibe un patrón repetitivo denominado “localidad de referencia”. Cuando se accede a un dato, existe una alta probabilidad de que éste sea accedido de nuevo en un tiempo próximo. A este tipo de localidad

de referencia se le califica de “temporal”. Un muy simple ejemplo es la instanciación de una variable, la cual, sobre todo si se siguen buenas costumbres de programación, debería accederse para lectura o escritura en un tiempo muy próximo.

Otro patrón consiste en acceder a un dato cercano en memoria de otro recientemente accedido. A este clase de localidad de referencia se le califica de “espacial”, y un buen ejemplo lo constituye el acceso a vectores y matrices.

La localidad de referencia, aunado al conocimiento de la regla del 80-20, intuye un mecanismo muy utilizado para mejorar el rendimiento denominado “cache”; verbo francés que significa “esconder” y que alegoriza la transparencia del mecanismo. Un cache es un conjunto abstracto, finito, sustentado en una estructura de datos, que se antepone a un conjunto de datos mucho mayor y cuyo acceso es notablemente más rápido que el conjunto de datos. Cuando se accede por vez primera a un elemento del conjunto, éste se guarda en el cache. Los siguientes accesos se ejecutan sobre el cache, que es mucho más rápido. Puesto que el cache es finito, posiblemente éste devenga lleno. En este caso, se debe seleccionar una entrada del cache para eliminar de manera que pueda substituirse por la nueva. Por lo general, se elimina la entrada que tenga más tiempo sin usarse.

Caches son típicamente usados en las plataformas de hardware para paliar la diferencia de desempeño entre el CPU y la memoria. Por la misma razón, los sistemas operativos los usan para paliar el desfase entre la memoria y el disco. En ambas situaciones suele guardarse en el cache no sólo el dato recién accedido, el cual cubre la localidad de referencia temporal, sino también los datos adyacentes, de manera que también se cubra la localidad espacial.

En § 5.3.3 (Pág. 435) estudiaremos un enfoque general de cache para usarse sobre conjuntos en memoria.

### 3.5.6 Tiempo de desarrollo

Los amantes de los grandiosos y modernos lenguajes de “scripting” tales como Perl, Python y Ruby, los cuales facilitan muchísimo el desarrollo rápido de prototipos y aplicaciones, tienen el siguiente proverbio: *“el tiempo humano es más importante que el tiempo de CPU”*<sup>21</sup>. Con esto expresan un elemento de eficacia y eficiencia que a menudo es descuidado e, inclusive, completamente despreciado: el tiempo de desarrollo de un programa.

Modelos y prototipos son fundamentales en ingeniería, pero por una extraña razón, pocos ingenieros de software suelen usarlos cuando se encuentran ante diseños de software novedoso o que no tienen experiencia en hacerlo. En todo caso, parece absurdo elaborar programas cuyo tiempo de desarrollo sea mucho mayor que la vida del programa en ejecución o, inclusive, lo cual es patéticamente más común, que no sean eficientes o, mucho peor, ineficaces.

Según las consideraciones anteriores, concluyamos con las siguientes recomendaciones tendientes a acelerar el tiempo de desarrollo:

1. Prime la simplicidad de diseño por el desempeño, pues, como reiteradamente lo hemos señalado, la eficacia prima a la eficiencia. Por tanto, piense en optimización sólo si se ha asegurado el fin y se requiere más eficiencia.

<sup>21</sup>Escuchado por primera vez por este redactor por boca de Francisco Palm.

2. Invierta todo el tiempo necesario en garantizar correctitud de los diseños antes de proceder a la codificación.

En virtud de esta recomendación, use modelos y verifíquelos exhaustivamente. Si existen especificaciones de eficiencia, entonces verifíquelas en los modelos.

“Prototipee” rápidamente prototipos de los modelos ya verificados y convalídelos con las condiciones esperadas de ejecución. En este sentido, es perfectamente plausible usar lenguajes que permitan desarrollar efectivamente prototipos e, inclusive, si la eficiencia es satisfactoria, basar el desarrollo definitivo en el lenguaje de prototipeado.

### 3.6 Notas bibliográficas

El ordenamiento, como obrar del hombre, se encuentra en toda cultura y época humana. Pudiera decirse que el método de selección es de “autoría humana”, pues no se encuentra el nombre de algún individuo que se atribuya para sí mismo su autoría.

Los historiadores de la computación atribuyen el ordenamiento por mezcla a John von Neumann; una de las mentes más geniales del siglo XX. El método de inserción se le atribuye a Konrad Zuse, a quien también se atribuye el descubrimiento del computador digital.

El quicksort fue descubierto por Charles Anthony Richard Hoare en 1961 [71]. Desde entonces hasta el presente, el método ha sido intensivamente estudiado. Knuth [99] ofrece un muy detallado estudio así como un excelente recuento histórico del quicksort y demás métodos de ordenamiento.

Robert Sedgewick, bajo la tutoría de Knuth, consagró su tesis doctoral al estudio del quicksort. Como tal, Sedgewick [155] es una excelente referencia para el análisis del método y sus variantes aplicativas.

La notación  $\mathcal{O}$ , fundamento esencial del análisis de algoritmos, fue originalmente concebida por el matemático alemán Paul Bachmann en 1894 [11].

Según Tarjan [167], el análisis amortizado fue desarrollado por M. R. Brown, Robert E. Tarjan, S. Huddleston y K. Mehlhorn [167]. En su artículo, Tarjan atribuye el descubrimiento del método potencial a Daniel D. Sleator, mientras que el calificativo “amortizado” se atribuye a Tarjan y Sleator [167].

La idea de verificación de correctitud basada en invariantes ha sido popular desde el mismo inicio de la programación. Sin embargo, al respecto vale la pena destacar el trabajo de David Rosenblum [150, 149].

La especificación y verificación automática de programas fue iniciada a finales de los años 60, pero no fue sino hasta finales de la década del 70 cuando se tuvo claro, con los trabajos de Zafiroplulo *et al* [188], que el futuro de la verificación de sistemas se hará formal y automáticamente.

Si usted llega a ser un programador, entonces quizá algún día emprenderá la dirección de una obra compleja en un sistema computacional que requiera la participación de muchos programadores. En ese entonces requerirá dominar las buenas costumbres y normalizarlas entre los programadores. En ese momento deberá conocer muy bien acerca de la Ingeniería del Software. En el ínterin, primero domine el ser programador y comience por leer el primer libro, en tiempo y en excelencia, de Ingeniería del Software: *The Mythical Man-Month: Essays on Software Engineering* de Fredrick P. Brooks [24].

El campo sobre disciplina y buenas costumbres de programación tendientes a producir programas correctos es muy vasto. No hay, pues, suficiente espacio en este texto para discernir al respecto. Sin embargo, ante la posibilidad de fragmentación del lector ante la extensa variedad de fuentes, es esencial responsabilidad en esta subsección mencionar explícitamente los principales textos de referencia:

1. *Writing Solid Code* [114] de Steve Maguire versa sobre cómo escribir programas que no contengan errores. Aunque parezca muy osado, Maguire enseña una disciplina que fuerza codificar programas correctos. Para resumir, Maguire labra un camino en torno a las tres preguntas de Van Cleck.
2. *The Practice of Programming* [93] de Kernighan y Pike es un libro sobre las buenas costumbres de programación, escrito con mucha autoridad, pues sus autores son practicantes involucrados en excelsas obras de software.
3. *Beautiful Code: Leading Programmers Explain How They Think* [135] editado por Andy Oram y Greg Wilson, presenta una recopilación de varios diseños e instrumentaciones de programas célebres y elegantes.

Finalmente, “*Code Complete*” [122] de Steve McConnell es un excelente tratado enciclopédico sobre buenas costumbres en programación.

### 3.7 Ejercicios

1. Demuestre las afirmaciones § 3.2 (Pág. 157), § 3.3 (Pág. 157) y § 3.4 (Pág. 157).
2. Demuestre todas las reglas algebraicas enunciadas en § 3.1.7.1 (Pág. 159).
3. Resuelva exactamente las siguientes recurrencias:

(a)

$$T(n) = \begin{cases} 1, & n = 1 \\ 3T(n-1) + 2, & n > 2 \end{cases}$$

(b)

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n/2) + 1, & n \geq 2, n \text{ potencia exacta de } 2 \end{cases}$$

(c)

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + 6n - 1, & n \geq 2, n \text{ potencia exacta de } 2 \end{cases}$$

(d)

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T(n/2) + \lg n & \text{si } n > 1 \end{cases}$$

4. Considere la siguiente función:

```

unsigned int misterio(unsigned int & n)
{
 if (n == 0)
 return 0;
 unsigned int suma = 0;
 for (unsigned int i = 1; i <= n; i++)
 suma += 2*i - 1;
 return suma;
}

```

¿Qué retorna la función?

5. Diseñe un algoritmo de partición triple tal como el explicado en § 3.2.2.8 (Pág. 183).
6. Diseñe el procedimiento descrito en *(mezcla de arreglos 171)* de modo tal que el arreglo secundario sólo consuma la mitad del espacio. Es decir, si el arreglo está entre  $[l \dots r]$ , entonces la dimensión del arreglo  $b$  debe ser  $\left\lceil \frac{r-l-1}{2} \right\rceil$ .
7. Diseñe un algoritmo genérico que se sirva del patrón genérico `Iterator` y que ordene por selección una secuencia. Diseñe el algoritmo de forma tal que sea lo más eficiente posible en arreglos y listas doblemente enlazadas.
8. La mezcla de dos secuencias ordenadas usada en el método de ordenamiento por mezcla requiere copiar todos los elementos de los arreglos a mezclar en un arreglo auxiliar. Diseñe un algoritmo que disminuya el espacio adicional a la mitad.
9. Analice el tiempo de ejecución del siguiente pseudoprograma:

```

void programa(int l, int r)
{
 if (l >= r)
 return;

 0(1)

 int m = (l + r) / 2;
 programa(l, m - 1);
 programa(m + 1, r);
}

```

10. Dado el siguiente programa:

```

int programa(int l, int r, int A[])
{
 if (r < l)
 return 0;
 for (int i = l; i < r; i++)
 x = x + A[i];
 int y = 0;
 y += x + programa(l, r - 1, A);
 return y;
}

```

- (a) Analice el tiempo de ejecución
- (b) Para el arreglo  $A = [1, 2, 3, 4, 5, 6]$ . ¿Qué valor devuelve la llamada  $\text{programa}(3, 5, A)$ ?
11. Para cada uno de los siguientes programas, analice el tiempo de ejecución:

```

(a) void f()
{
 int i = 0;
 for(int j = i; j > -1; j++);
}

(b) void g(int n)
{
 for (int j = 2^n; j < lg n; j /= 2)
 i++;
}

(c) void h(int n)
{
 for (int x = 1; x < n; x += 2)
 printf("impar: %d", x);
}

(d) void f(int N)
{
 int n = 1;
 do
 {
 n *= 2;
 for (z = 0; z < n; z++)
 printf("%d\n", z);
 }
 while (n < N);
}

```

12. Considere un arreglo con  $n$  diferentes enteros desordenados comprendidos entre 0 y  $m$ ,  $m \gg n$ . Se requiere determinar un número faltante entre los enteros con un coste en espacio  $\mathcal{O}(1)$ . Lo último significa que no se pueden utilizar estructuras de datos auxiliares cuyo espacio crezca en función del tamaño del arreglo.

Por ejemplo, para  $m = 1000$  y un arreglo  $A = [0 1 2 3 37 28 36 273 2721 99 327 7 354 6 84]$ ,  $500 \notin A$  es una solución.

- (a) Proponga un algoritmo a fuerza bruta que resuelva el problema. Es decir,  $\forall i \in [0..m]$ ,  $0 \leq i \leq m$  busque en el arreglo hasta no encontrar el valor de  $i$ .
- (b) Analice rigurosamente el algoritmo anterior para el mejor, peor y caso promedio.
- (c) Proponga, explique y diseñe un algoritmo que resuelva el problema en  $\mathcal{O}(n \lg m)$  para el peor caso.
- (d) Bajo la existencia del algoritmo de la pregunta anterior, discuta cuál y en cuáles circunstancias un algoritmo es mejor, el  $\mathcal{O}(n \lg m)$  o el diseñado en 12a.

13. Calcule el tiempo de ejecución del algoritmo de multiplicación de polinomios desarrollado en la subsección § 2.4.10 (Pág. 91).
14. Elimine la recursión cola del método `quicksort_rec()`.
15. haga un estudio estadístico acerca de los tamaños de particiones a partir de las cuales el ordenamiento por inserción es más rápido que el quicksort.
16. Implante el ordenamiento por selección con listas simplemente enlazadas del tipo `Snode<T>`.
17. Implante el ordenamiento por inserción con listas simplemente enlazadas del tipo `Snode<T>`.
18. Implante el ordenamiento por mezcla con listas simplemente enlazadas del tipo `Snode<T>`.
19. Implante un quicksort iterativo que ordene listas simplemente enlazadas del tipo `Snode<T>`.
20. Implante el quicksort con listas simplemente enlazadas del tipo `Snode<T>`.
21. Escriba un algoritmo que particione una lista simplemente enlazada circular con nodo cabecera del tipo `Snode<T>`. El prototipo es el siguiente:

```
Snode<T> * partir(Snode<T> & list, Snode<T> & l1, Snode<T> & l2);
```

Inicialmente, `l1` y `l2` están vacías. Al final de la llamada se retorna el elemento pivote de la partición; la lista `list` queda vacía y `l1` contiene los elementos menores que el pivote mientras que `l2` los mayores.

22. Diseñe e implante un quicksort no recursivo sobre listas dobles de tipo `Dnode<T>`.
23. Diseñe e implante un quicksort no recursivo sobre listas simples de tipo `Snode<T>`.
24. Diseñe e implante un quicksort sobre listas dobles con `Dnode<T>` que seleccione la mediana entre algunos de los elementos constantemente asequibles de la lista.
25. Escriba el quicksort para listas doblemente enlazadas de manera tal que aleatorice la selección del pivote.
26. Diseñe e implante la siguiente primitiva

```
int select_pivot(T a[], const int l, const int r, const int n)
```

La cual sortea al azar `n` índices entre `l` y `r` y selecciona el índice correspondiente a la mediana.

No verifique repitencia entre los índices.

27. Modifique el quicksort para listas enlazadas presentado en § 3.2.2.6 (Pág. 181) para que haga una mejor selección del pivote.
28. Estudie el consumo de espacio en pila que ocasiona el mergesort debido a las llamadas recursivas.

29. Modifique el quicksort para listas enlazadas presentado en § 3.2.2.6 (Pág. 181) para que minimice el consumo de espacio en sus llamadas recursivas.
30. De los métodos de ordenamiento presentados en este capítulo, ¿cuál sería el mejor para ordenar listas enlazadas?
31. El mismo ejercicio que el anterior, pero esta vez no puede haber repitencia entre los índices sorteados.
32. Diseñe e implante un algoritmo que encuentre el  $i$ -ésimo menor elemento de una lista simplemente enlazada desordenada ( $\text{Snode}\langle T \rangle$ ).
33. Diseñe e implante un algoritmo que encuentre el  $i$ -ésimo menor elemento de una lista doblemente enlazada desordenada ( $\text{Dnode}\langle T \rangle$ ).
34. Diseñe e implante el esquema de triple particionamiento explicado en § 3.2.2.8 (Pág. 183) para manejar claves repetidas.
35. Analice rigurosamente, para los casos esperado y pero, el desempeño de la primitiva `random_search()`.
36. Escriba el `random_search()` para listas simplemente enlazadas de tipo `Slink`.
37. Analice rigurosamente, para los casos esperado y pero, el desempeño de la primitiva `random_select()`.
38. Implante el `random_search()` con listas simples de tipo  $\text{Snode}\langle T \rangle$ .
39. Implante el `random_select()` con listas simples de tipo  $\text{Snode}\langle T \rangle$ .
40. Implante el conjunto fundamental con arreglos desordenados basado en las primitivas `random_search()` y `random_select()`.
41. Implante el conjunto fundamental con listas simples de tipo  $\text{Snode}\langle T \rangle$  basado en las primitivas `random_search()` y `random_select()`.
42. Implante el conjunto fundamental con listas dobles de tipo  $\text{Dnode}\langle T \rangle$  basado en las primitivas `random_search()` y `random_select()`.
43. Extienda GNU nana para que maneje los cuantificadores que verifiquen contenedores de la biblioteca  $\mathcal{ALCOPH}$ .
44. Para todos los TAD diseñados hasta el presente, verifique la aplicación de las reglas de Holtzmann [73] presentadas en § 3.4.3.1 (Pág. 196).



## 4

# Árboles

Aunque actualmente con tendencia al olvido, desde hace mucho tiempo, a eso que le brinda identidad y sentido a una cosa se le llama “esencia” o “quid”. Desde y durante edades inmemoriales, ancestrales y lejanas, se cree que el quid de algo se estudia a partir de su contexto de origen. Lo que ahora vulgarmente llamamos futuro, ancestralmente se le llamaba destino, o sea, hacia dónde se va. Cuando se quería saber qué era (desde dónde viene) y qué devendría (hacia dónde va) una cosa, se estudiaba y se criticaba su “genealogía”, es decir, su origen y devenir a partir de su génesis.

El término “genealogía”, que contiene a “genus”, proviene del latín *genealogia*, el cual proviene a su vez del griego γενεαλογία y significa el estudio del camino desde el origen (génesis) hasta el presente. Hoy en día, la genealogía tiende a estudiarse como una secuencia de eventos respecto a un movimiento externo y ajeno [a los eventos] llamado tiempo. A este tipo de estudio se le llama por lo general “Historia”.

Hubo épocas, sin embargo, en que algunos genealogistas sentían que el quid de una cosa no se remitía a una mera y única secuencia de eventos, sino a diversas y variables secuencias que provenían desde un momento denominado origen, convergían hacia otro momento llamado presente y proseguían hacia algo por venir que se llama destino. Aquellos genealogistas se percataron de que convivían con testigos que habían acompañado a sus ascendientes y que les sobrevirían a ellos y sus descendientes. Asimismo se fascinaron por el hecho de que las propias formas de aquellos testigos indiciaban su devenir y edad. Al género de aquel testigo hoy se le mienta bajo el nombre de “árbol”, y a la excepción de la cultura tecnológica de esta época, en las restantes el árbol ha causado tanto asombro a través de las edades y latitudes, que ha sido al extremo sujeto de veneración.

¿Por qué asombraba? Aparte de su belleza, monumentalidad y longevidad, había varias razones. Una de las principales, de gran interés para nuestro contexto, era su estructura genérica, así como su evolución. Un árbol comienza en el mundo como una semillita que lentamente enraíza y crece mirando y buscando la luz en una trama jerárquica que traza su edad, su genealogía y que transige imaginar su destino. Aunque por lo general, a un árbol lo miramos como a un árbol, es decir, como un todo, en ocasiones miramos sus partes: raíz, tronco, ramas, hojas, flores, .... Cuanto más cerca se está de la raíz, más antigua es la región del árbol que miramos; análogamente, cuanto más alta o amplia ésta es, más joven es la parte respecto a otras inferiores o más internas. Este patrón estructural, que no se corresponde con una mera secuencia, aparece en una numerísima variedad de contextos de nuestra vida de los cuales no escapa la computación.

A los actos y hechos humanos les llamamos “gestos”, pues éstos “gestan”. Ser humano

sólo tiene sentido con los otros seres humanos. Más humanos somos en la medida en que más nos hayamos bien relacionado con los otros. De alguna manera podríamos decir que como humanos nos distinguimos entre nosotros porque los gestos que recibimos del otro y gestamos para él, son de alguna manera diferentes. Un antiguo y hermoso proverbio hindú reza: “*un gesto genera una actitud, una actitud genera un carácter y un carácter genera un destino*”. Si generalizamos nuestra identidad, o la de algún otro, podemos pensar en el carácter y hacer memoria en las actitudes y gestos que condujeron aquellas actitudes. Indagando en un carácter tenemos idea acerca de cómo se proyecta el destino. No en balde, Heráclito decía que “*el carácter de un hombre es su destino*”. Como un hipotético y muy reducido ejemplo consideremos la risa, la cual, en cierta forma, gesta alegría y a su vez gesta también un carácter amistoso. La amistad, cuando se circumscribe dentro de una auténtica noción de bien, gesta un destino enmarcado en la buena vida subyacente a la idea de bien que en amistad se comparta. La vida humana está plena de muchos más gestos, la sonrisa, el llanto, el grito, el asombro ... así como de sus consecuentes actitudes, la picardía, la tristeza, el enojo, la curiosidad .... Caracteres emergen de aquellas actitudes; el amistoso, el astuto, el deprimido, el desconfiado, el perseverante .... Dependiendo del devenir y de la suerte, esos caracteres conjugados pueden destinarnos a ser un padre, un político, un poeta, un guerrero, un filósofo .... Todo este discurso puede sintetizarse, de una manera un tanto reducida pero más aprensible, bajo el diagrama de la figura 4.1 En los gestos, los nuestros y los del otro, se gestan nuestros destinos.

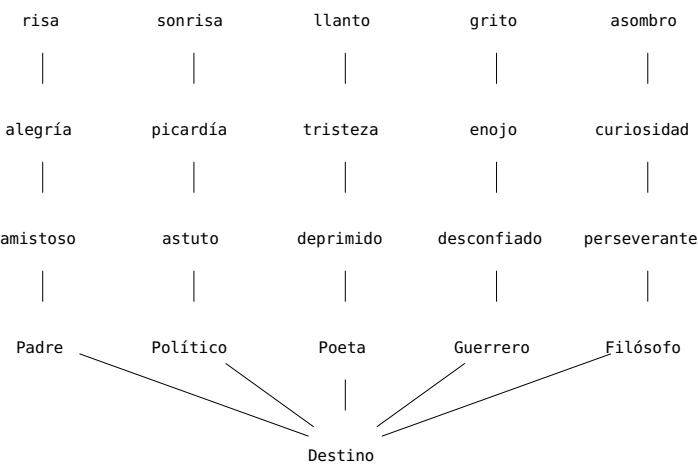


Figura 4.1: Un árbol “destinal”

Antes de abordar formalmente el concepto computacional de árbol, empapémosnos un poco más de la idea abstracta de árbol en un contexto más cotidiano.

En el entorno genealógico de esta época, quizá sea el “árbol genealógico”, el cual traza ascendencia, y cuya representación general se ilustra en la figura 4.2, un ejemplar representante cultural de la estructura árbol y de su similaridad pictórica y jerárquica con su parangón natural.

A diferencia de su estereotipo natural, en el árbol genealógico las semillas (los bisabuelos) se encuentran en las hojas, mientras que el último descendiente está en la raíz. En el mismo sentido genealógico, aunque más individualista, un patrón seminal más fiel con el árbol natural es el “árbol dinástico”, el cual traza las descendencias y cuya estructura básica, pero no general, se ilustra en la figura 4.3.

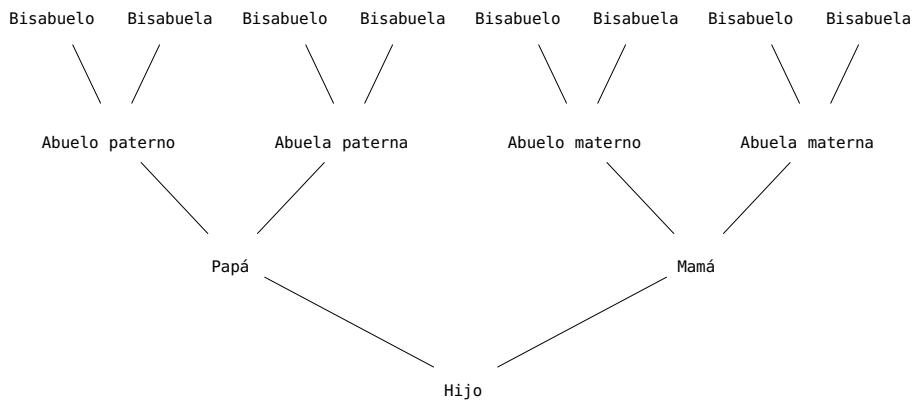


Figura 4.2: Forma general de un árbol genealógico

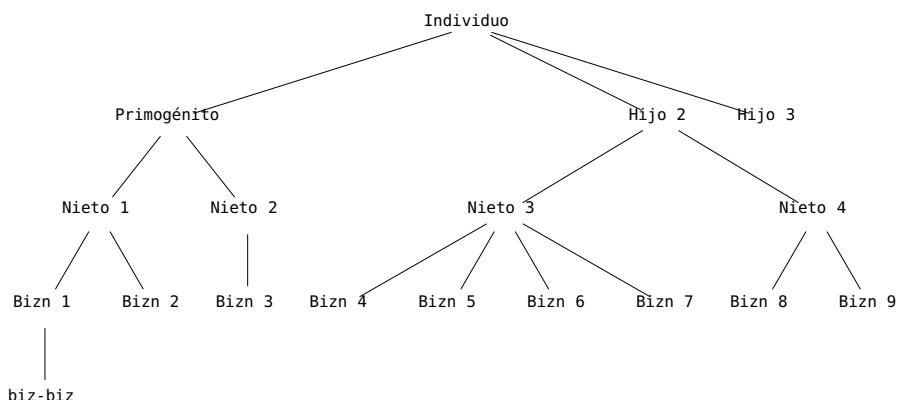


Figura 4.3: Forma básica de un árbol dinástico

El árbol genealógico general de la figura 4.2 es “binario” en el sentido de que cada miembro tiene exactamente dos ancestros directos: el padre y la madre. En cambio, la numericidad del árbol dinástico dependerá del carácter prolífico de la familia.

En programación se maneja la abstracción de árbol para representar jerarquías de cómputo, órdenes especiales de datos o situaciones que atañan a los árboles culturales e, inclusive, naturales. De hecho, en § 1.2 (Pág. 11) introdujimos el concepto de “herencia de clases” y su carácter “genealógico”.

En computación, las aplicaciones del árbol son muy diversas y van desde la recuperación eficiente de claves hasta la resolución analítica de expresiones del cálculo formal. El árbol fundamenta muchos esquemas de recuperación y reconstrucción de información, tanto en memoria primaria como en secundaria. Todo proceso de traducción, por ejemplo, la compilación de algún lenguaje de programación, utiliza el árbol como estructura fundamental. El cálculo formal automático, es decir, el cálculo de expresiones analíticas, por ejemplo, límites, derivadas, integrales, etcétera, usa al árbol como estructura de representación de las expresiones.

Una labor de este capítulo será formarnos con los lineamientos básicos a efectos de cumplir los siguientes objetivos:

1. Conocer los conceptos básicos y homogeneizar terminologías.

2. Comprender los algoritmos fundamentales y preparar al lector en el desarrollo de algoritmos más complejos.
3. Prepararnos para el estudio de estructuras árbol, especializadas, que serán desarrolladas en otros capítulos de este texto.

Presentada la idea intuitiva y genealógica del árbol, así como su interés en la computación, estamos prestos para iniciar formalmente nuestro estudio.

## 4.1 Conceptos básicos

Comencemos por una definición matemática.

**Definición 4.1 (Árbol)** Un árbol  $T$  se define por un conjunto  $\mathcal{N} = \{n_1, n_2, \dots, n_n\}$  de uno o más nodos que satisfacen las siguientes condiciones:

1. Existe un nodo especial llamado  $\text{raiz}(T) \in \mathcal{N}$ .
2. El conjunto  $\mathcal{N} - \{\text{raiz}(T)\}$  puede particionarse en  $m$  árboles  $T^1, T^2, \dots, T^m$  tal que:

$$\begin{aligned}\{\text{raiz}(T)\} \cup T^1 \cup T^2 \cup \dots \cup T^m &= T \\ T^1 \cap T^2 \cap \dots \cap T^m &= \emptyset\end{aligned}$$

Cada uno de los conjuntos  $T^1, T^2, \dots, T^m$  son “subárboles” de  $T$ .

El contenido de un nodo  $n_i \in T$  se representa como  $\text{KEY}(n_i)$ , al cual a menudo se le llama “clave”.

La figura 4.4 ilustra un ejemplo donde la raíz es el nodo etiquetado con la letra A y las raíces de los subárboles de A están etiquetados con B, C, D y E respectivamente. Notemos que el árbol dibujado en la figura 4.4 está invertido: la raíz está en la parte superior y las hojas en la parte inferior.

Evidentemente, existe todo un argot genealógico para describir un árbol. A un nodo directamente conectado a la raíz se le llama “hijo”. A excepción de la raíz, todo nodo tiene un “padre”. Dado un nodo  $n_i$ , llamaremos  $\text{padre}(n_i)$  al padre de  $n_i$  e  $\text{hijos}(n_i)$  al conjunto formado por los hijos inmediatos de  $n_i$ . Del mismo modo, dado un nodo  $n$ ,  $\text{hijo}(n, i)$  denota a la  $i$ -ésima rama de  $n$ . Si  $\text{hijo}(n, i) = \emptyset$ , entonces  $n$  no tiene una  $i$ -ésima rama.

A un nodo que no tenga hijos se le llama “hoja”, pues alegoriza la hoja de un árbol en el sentido de que es una terminación.

A parte de la raíz, al resto de los nodos se les llama “descendientes” del nodo  $\text{raiz}(T)$ .

Un árbol contiene nodos y arcos. Un nodo asocia un elemento de algún tipo genérico  $T$ . Un arco representa una relación entre los dos nodos involucrados.

Denominaremos  $\mathcal{T}$  al conjunto infinito de todos los árboles posibles.

Una arborescencia se define como un conjunto de cero o más árboles disjuntos.

Un camino  $< n_0, n_1, \dots, n_{n-1}, n_n >$  es una secuencia correspondiente a nodos interconectados por arcos pertenecientes al árbol tal que  $\text{hijo}(n_0) = n_1 | \text{hijo}(n_1) = n_2 | \dots |$

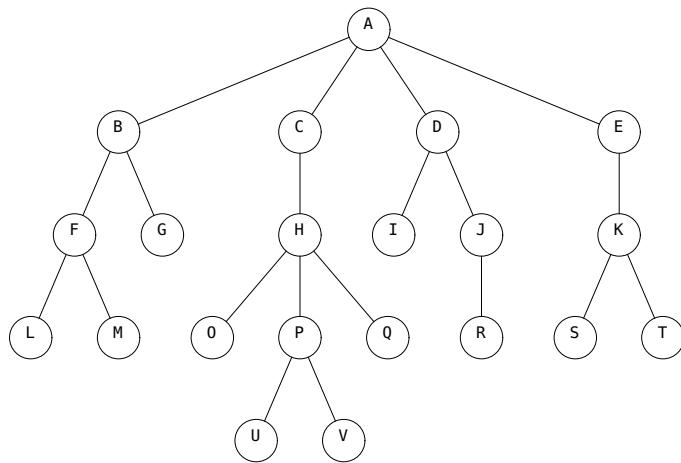


Figura 4.4: Un árbol de ejemplo

hijo( $n_{n-1}$ ) =  $n_n$ . Un camino se denota por  $C_{n_s, n_t}$  donde  $n_s$  es el nodo inicio del camino y  $n_t$  es el nodo final del camino.

La “longitud del camino”, denotada como  $|C_{n_s, n_t}|$ , es la cantidad de nodos que contiene el camino.

Los “ancestros” de  $n_i$ , denotado como  $\text{ancestros}(n_i)$ , es el conjunto conformado por los nodos pertenecientes al camino  $C_{\text{raiz}(\mathcal{T}), \text{padre}(n_i)}$ ; es decir, la secuencia de nodos predecesores desde la raíz hasta  $n_i$ .

Dados dos nodos  $n_i, n_j \in \mathcal{T}$ , entonces,  $n_j$  es ancestro de  $n_i$  si y solo si,  $n_j \in \text{ancestros}(n_i)$ .

Algunos discursos utilizan una noción de “edad de nodo”, la cual está dada por su posición dentro del camino de sus ancestros. De este modo, dado un camino  $C_{n_0, n_n} = < n_0, n_1, \dots, n_{n-1}, n_n >$ , decimos que  $n_i$  es más viejo (o mayor) que cualquier  $n_{i+c} \in C_{n_{i+1}, n_n}$ . Del mismo modo decimos que  $n_i$  es más joven (o menor) que cualquier  $n_{i-c} \in C_{n_0, n_{i-1}}$ .

Dado un nodo  $n_i \in C_{n_0, n_n} = < n_0, n_1, \dots, n_{n-1}, n_n >$ , el conjunto  $\text{hijos}(n_i)$  es llamado la  $i$ -ésima generación.

El “grado” de  $n_i$  denotado como  $\text{grado}(n_i)$ , es la cantidad de subárboles que contiene el nodo  $n_i$ .

El “orden” o “grado” de un árbol se define como el número máximo de ramas que puede tener cualquier nodo del árbol.

Si el grado de un nodo es igual al orden del árbol, entonces decimos que el nodo está completo. Cuando un árbol posee muchos nodos incompletos y éstos poseen muy pocas ramas, entonces decimos que el árbol es esparcido. Del mismo modo, cuando hay muchos nodos completos decimos que el árbol es denso.

El “nivel de un nodo”  $n_i$ , denotado como  $\text{nivel}(n_i)$ , se define recursivamente como sigue:

$$\text{nivel}(n_i) = \begin{cases} 0 & \text{si } n_i = \text{raiz}(\mathcal{T}) \\ 1 + \text{nivel}(\text{padre}(n_i)) & \text{si } n_i > 0 \end{cases}$$

La “altura de un nodo”  $n_i$ , denotada como  $h(n_i)$ , se define como el número de nodos menos uno que hay desde  $n_i$  hasta una hoja situada en el más alto nivel.

También puede definirse recursivamente como sigue:

$$h(n_i) = \begin{cases} 0 & \text{si } n_i \text{ es una hoja} \\ 1 + \max(h(\text{hijo}_1(n_i)), \dots, h(\text{hijo}_m(n_i))) & \text{de lo contrario} \end{cases}$$

La “altura de un árbol”  $T$  se define como la altura de  $\text{raiz}(T)$ ; es decir,  $h(\text{raiz}(T))$ .

La “cardinalidad de un árbol”  $T$ , denotada como  $|T|$ , se define por el número total de nodos del árbol.

Ilustraremos algunos de estos conceptos mediante la figura 4.4. La raíz es el nodo etiquetado con  $A$  y está situada en el nivel 0. El grado de  $\text{raiz}(T)$  es 4, que también corresponde al orden del árbol. Los hijos de  $A$  son  $B$ ,  $C$ ,  $D$  y  $E$ .  $\text{padre}(B) = \text{padre}(C) = \text{padre}(D) = \text{padre}(E) = A$ . La altura de  $A$  es 5, que corresponde a la longitud del camino más largo desde  $A$ . Tal camino está dado por  $\mathcal{C}_{A,u} = A, C, H, P, U$  o  $\mathcal{C}_{A,v} = A, C, H, P, V$  |  $|\mathcal{C}_{A,v}| = 5$ .

Ahora consideremos un nodo particular, por ejemplo,  $H$ . Tenemos:

- $\text{grado}(H) = 3$
- $\text{padre}(H) = C$
- $h(H) = 3 = |\mathcal{C}_{H,u}| = |\mathcal{C}_{H,v}|$

Ejercítense con esta terminología considerando cada nodo de la figura 4.4.

## 4.2 Representaciones de un árbol

Tradicionalmente, estos árboles abstractos se dibujan al revés de los naturales, como en la figura 4.4. Es decir, la raíz en la parte superior y sus descendientes en las partes inferiores. Esta representación es idónea en la mayoría de las situaciones, pero algunos problemas se prestan para otras representaciones más convenientes.

### 4.2.1 Conjuntos anidados

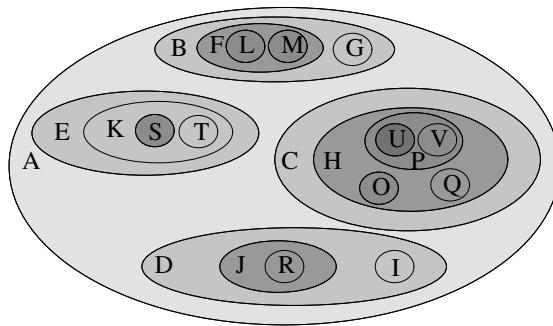


Figura 4.5: Conjuntos anidados

En esta representación, la raíz es un conjunto de nombre  $\text{raiz}(T)$  con subárboles como subconjuntos. Por ejemplo, la figura 4.5 ilustra el equivalente en conjuntos anidados del árbol de la figura 4.4.

Esta representación puede fundamentarse sobre estructuras de datos especializadas en conjuntos.

### 4.2.2 Secuencias parentizadas

Los conjuntos anidados ignoran el orden de aparición de los subárboles, el cual en algunas situaciones puede ser importante. Si se desea mantener este orden, un refinamiento consiste en representar los conjuntos entre paréntesis. En este caso, la asociatividad representa el orden de aparición de los subárboles. Para el árbol de las figuras 4.4 y 4.5 se tiene la siguiente representación parentizada:

$$(A(B(F(L)(M))(G))(C(H(O)(P(U)(V))(Q)))(D(I)(J(R)))(E(K(S)(T))))$$

Esta representación tiene la ventaja de que es lineal y refleja con exactitud la topología del árbol.

Consideremos la evaluación automática de expresiones algebraicas. Como ejemplo, asumamos esta expresión algebraica

$$\frac{3x^4 + 3x^3y^2 + x^2 - 1}{4x^2z} .$$

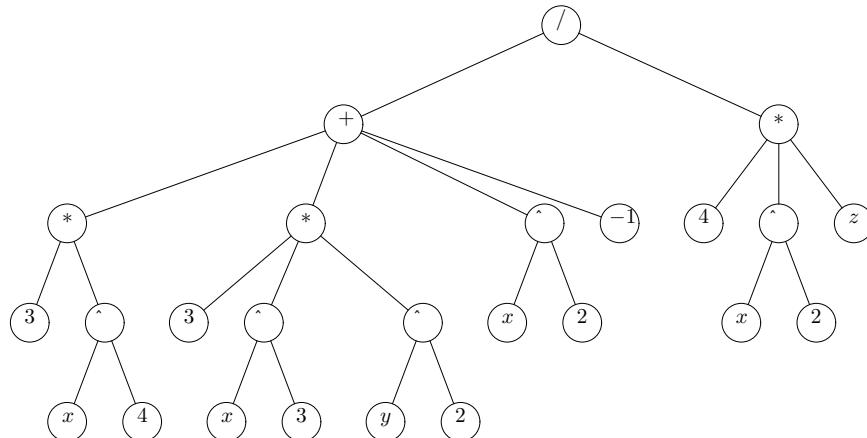


Figura 4.6: Árbol de la expresión infija  $\frac{3x^4 + 3x^3y^2 + x^2 - 1}{4x^2z}$

Un compilador podría traducir esta expresión al árbol de la figura 4.6, el cual debe evaluarse de manera “ascendente”, o sea, desde las hojas hasta la raíz. Este orden refleja fielmente las posibles y válidas maneras en que la expresión puede evaluarse sin que se pierda su sentido algebraico.

En matemática, la composición de funciones suele escribirse de manera prefija, es decir, el nombre de la función u operación “prefija”, “antecede”, a los operandos. Por ejemplo,  $f(x, y)$  representa una función  $f$  con dos operandos. Ocurre que la secuencia parentizada de un árbol de expresiones es prefija. En la ocurrencia, el árbol de la figura 4.6 se representa como:

$$(/(+(*(3) (^ (x)(4))) (*(3) (^ (x)(3)) (^ (y)(2))) (^ (x)(2)) (-1)) (* (4) (^ (x)(2)) (z)))$$

La linealidad de la representación parentizada puede utilizarse para la transmisión de árboles en una red. El sitio origen de la comunicación obtiene una representación parentizada que se embala en un mensaje. Al recibir el árbol parentizado, el sitio destino ejecuta la transformación inversa -desembalaje- y obtiene una copia exacta del árbol.

En la mayoría de los lenguajes de programación, en artilugios electrónicos de cálculo y en general en la ingeniería, las expresiones se colocan en forma infija y los paréntesis se ubican en donde queramos modificar la precedencia de los operadores. De este modo, la expresión anterior podría escribirse en forma infija como:

$$(3*x^4 + 3*x^3*y^2 + x^2 - 1) / (4*x^2*z)$$

La cual es la versión resumida de la expresión infija correspondiente al árbol de la figura 4.6. La expresión infija es una representación lineal del árbol y requiere el uso de paréntesis para especificar el orden de aparición de los nodos; exactamente de la misma manera en que debe hacerse con la precedencia en las expresiones algebraicas.

#### 4.2.3 Indentación

Una manera “conveniente y barata” para dibujar árboles pequeños, en la cual no se pierde la estructura, consiste en indentar por nivel. La idea se explica mediante el siguiente algoritmo:

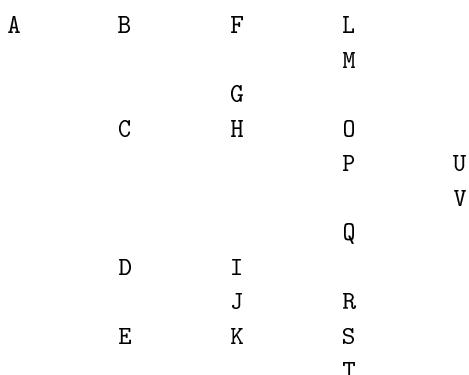
**Algoritmo 4.1 (Dibujo de un árbol en modo texto)** La entrada del algoritmo es un árbol en su representación gráfica clásica. La salida es la representación indentada del árbol.

El algoritmo tiene un parámetro llamado  $s$ , definido como el número de espacios en blanco de separación horizontal.

El algoritmo es recursivo con prototipo `void dibujar(Node * x)`. La primera llamada debe ejecutarse con la raíz  $p$ .

1. Imprimir  $p$ .
2.  $\forall y \in \text{hijos}(x)$ 
  - (a) Indentar  $s$  espacios hacia la derecha.
  - (b) `dibujar(y);`
  - (c) Indentar  $s$  espacios hacia la izquierda.
3. Si  $x$  es una hoja  $\implies$ 
  - (a) Salte de línea.

El árbol de las figuras 4.4 y 4.5 tiene la siguiente representación indentada:



#### 4.2.4 Notación de Deway

La estructuración y contenido de este texto en capítulos, secciones, etcétera, se corresponde con una arborescencia en la cual los capítulos son árboles disjuntos. Las secciones de un capítulo son los subárboles de la raíz y, a la vez, las subsecciones son los subárboles de una sección. Esta estructuración obedece a una jerarquía según el área de estudio. Pues bien, el enfoque utilizado para enumerar cada sección constituye una manera de representar un árbol. El método en cuestión se denomina “notación decimal de Deway”, por analogía a una notación similar usada para clasificar libros en bibliotecas. Básicamente, la notación de Deway es una manera de enumerar e identificar únicamente cada nodo del árbol. El árbol de las figuras 4.4 y 4.5 puede representarse mediante los siguientes números de Deway:

```
(1 : A), (1.1 : B), (1.2 : C), (1.3 : D), (1.4 : E), (1.1.1 : F), (1.1.2 : G),
(1.2.1 : H), (1.3.1 : I), (1.3.2 : J), (1.4.1 : K), (1.1.1.1 : L), (1.1.1.2 : M),
(1.2.1.1 : O), (1.2.1.2 : P), (1.2.1.3 : Q), (1.3.2.1 : R), (1.4.1.1 : S),
(1.4.1.2 : T), (1.2.1.2.1 : U), (1.2.1.2.2 : V)
```

El valor añadido de la notación de Deway es que se preserva íntegramente la forma del árbol sin necesidad de ajustarse a un orden de secuencia. El árbol puede restaurarse independientemente del orden en que se presenten los nodos.

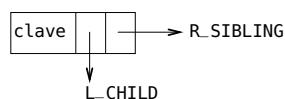
Una representación basada en la notación de Deway puede ser muy útil en algunas situaciones en que la construcción del árbol sea altamente dinámica. También puede adecuarse para la transmisión remota de árboles muy grandes que no tengan cabida en un solo mensaje.

### 4.3 Representaciones de árboles en memoria

Básicamente, existen dos maneras de representar un árbol en memoria: por arreglos y por listas enlazadas. Ambas representan un intermedio entre velocidad y espacio.

#### 4.3.1 Listas enlazadas

En esta representación, cada nodo se estructura según la forma siguiente:



Cuyos campos de interés son:

- **R\_SIBLING**: apuntador al nodo correspondiente al hermano derecho.
- **L\_CHILD**: apuntador a su hijo más a la izquierda.

Los campos **R\_SIBLING** conforman una lista simplemente enlazada de hermanos. La cabecera de esta lista es el nodo más a la izquierda. La entrada a esta lista es el campo **L\_CHILD** del nodo padre.

Los campos **L\_CHILD** conforman una lista simplemente enlazada de generaciones. Dado un nodo  $x$ ,  $L\_CHILD(x)$  es su hijo más a la izquierda;  $L\_CHILD(L\_CHILD(x))$  es el nieto

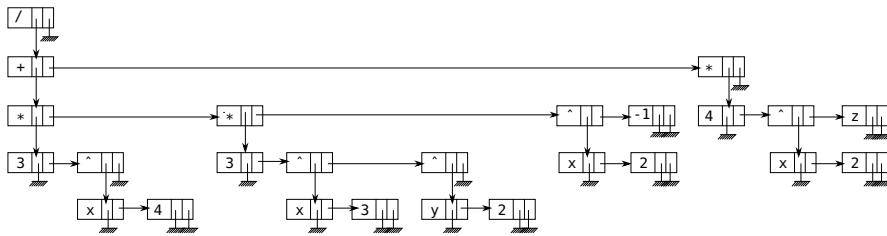


Figura 4.7: Representación con listas del árbol de la figura 4.6

más a la izquierda de  $x$ ;  $L\_CHILD(L\_CHILD(L\_CHILD(x)))$  es el biznieto más a la izquierda de  $x$  y así sucesivamente para el resto de las generaciones.

A menudo, al hijo más a la izquierda de un nodo se le tilda de “primogénito”.

La figura 4.7 ilustra la representación con listas del árbol mostrado en la figura 4.6.

Si no hay más hermanos o generaciones, entonces el apuntador correspondiente se marca con el valor especial NULL.

Esta representación es compacta y flexible, pues el orden y cardinalidad del árbol puede aumentarse o disminuirse dinámicamente. El espacio ocupado por cada nodo es constante e independiente del orden del árbol y del grado del nodo. El desperdicio en nodos incompletos (por las ramas no utilizadas) es pequeño:  $R\_SIBLING$  en el nodo más a la derecha de cada lista de hermanos, y  $L\_CHILD$  en el último descendiente de cada lista de generaciones.

Para acceder a un nodo de la  $i$ -ésima generación es necesario acceder las  $i - 1$  generaciones anteriores. Para acceder al  $j$ -ésimo hermano es necesario acceder los  $j - 1$  hermanos precedentes. Si el orden del árbol es conocido, entonces el tiempo de acceso está acotado y por lo tanto puede considerarse constante.

La versatilidad de la estructura puede mejorarse si se utilizan listas doblemente enlazadas. Muchos algoritmos que requieren regresar a sus ancestros pueden beneficiarse de esta extensión. Dado un nodo, una lista doblemente enlazada permite recuperar inmediatamente el parent. El enlace doble está dado por el puntero al hijo más a la izquierda y el puntero al parent. Del mismo modo, la lista enlazada puede ser de los hermanos, en cuyo caso el enlace doble está dado por los punteros al hermano izquierdo y derecho respectivamente.

### 4.3.2 Arreglos

Dado un árbol de orden  $m$ , cada nodo contiene el espacio para el dato y un arreglo  $m$ -dimensional de apuntadores a sus  $m$  subárboles. El valor NULL indica la ausencia de la rama correspondiente.

Esta representación tiende a utilizar más espacio que las listas enlazadas. El desperdicio de memoria en nodos incompletos puede ser importante si el árbol es muy esparcido. En la ocurrencia, en el árbol de orden 4 dibujado en la figura 4.8, sólo un nodo está completo; consecuentemente, el resto de los nodos incompletos desperdicia al menos una celda en punteros nulos.

La representación es estática en el sentido de que el orden del árbol no es fácil de modificar. Este enfoque no es conveniente para construir un árbol de orden desconocido.

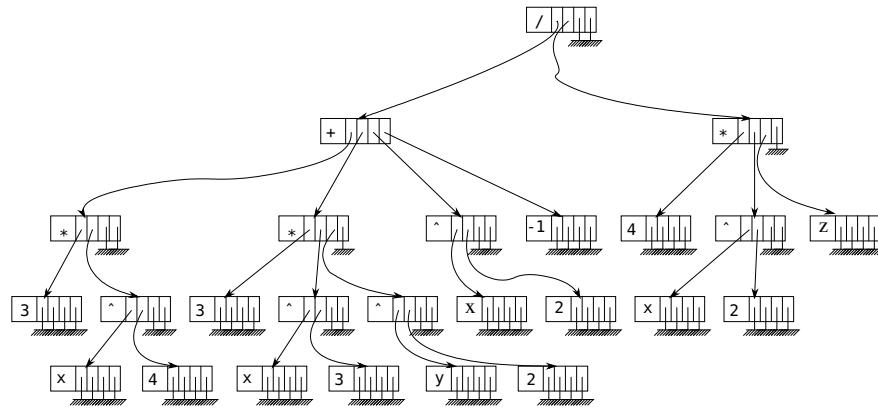


Figura 4.8: Representación con arreglos del árbol de la figura 4.6

En muchos casos es conveniente efectuar la construcción con la representación de listas enlazadas y convertirlo luego a la representación vectorizada.

El acceso a una generación es equivalente al de las listas enlazadas, pero el acceso a cualquiera de las ramas es directo; esta característica es la que hace a esta representación más rápida que su contraparte con listas enlazadas.

Una alternativa para este enfoque es utilizar un arreglo de pares <puntero a rama, ordinal de rama>. Este esquema utiliza un espacio por nodo proporcional al número de ramas y puede ganar espacio si el árbol es bastante esparcido. Si el número de ramas es susceptible de ser muy grande, el arreglo puede estar ordenado por ordinal de rama y la rama puede ser logarítmicamente localizable mediante la búsqueda binaria. Esto penaliza a la inserción y la eliminación, pues hay que abrir y cerrar brechas en el arreglo.

## 4.4 Árboles binarios

Ahora procederemos a estudiar una clase especial de árbol denominada “árbol binario”. El árbol binario es la base conceptual de una amplia gama de algoritmos y estructuras de datos.

**Definición 4.2 (Árbol binario)** Un árbol binario  $T$  es un conjunto finito de cero o más nodos definido recursivamente como sigue:

$$T = \begin{cases} \emptyset & \text{denota el árbol binario vacío} \\ < L(T), \text{raiz}(T), R(T) > & \text{de lo contrario} \mid \begin{cases} n & \text{es el nodo raíz} \\ L(T) & \text{subárbol izquierdo} \\ R(T) & \text{subárbol derecho} \end{cases} \end{cases} \quad (4.1)$$

El contenido de un nodo  $n_i$  se denota como  $\text{KEY}(n_i)$ .

Denominaremos  $B$  al conjunto infinito de todos los árboles binarios posibles.

La definición de árbol binario es diferente de la definición de árbol dada en § 4.1 (Pág. 226). La distinción la hace la noción de sentido de la rama. En la ocurrencia, los árboles binarios ilustrados en la figura 4.9 son diferentes bajo la definición 4.2, pero bajo

la definición 4.1 son iguales. Esta diferencia de sentido es precisamente la característica que nos garantizará una correspondencia unívoca entre un árbol binario y uno m-rio cualquiera.

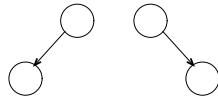


Figura 4.9: Dos árboles binarios diferentes

#### 4.4.1 Representación en memoria de un árbol binario

Los árboles binarios se representan en memoria con arreglos. Sin embargo, puesto que la dimensión del arreglo es dos, la representación es prácticamente la misma que con listas enlazadas. La figura 4.10 ilustra un ejemplo.

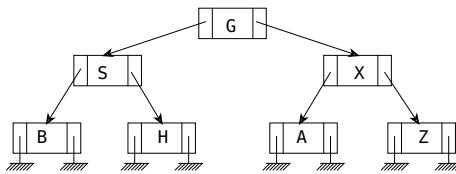


Figura 4.10: Representación en memoria de un árbol binario

Eventualmente puede convenir un tercer puntero por nodo correspondiente al nodo padre.

#### 4.4.2 Recorridos sobre árboles binarios

Los árboles y los árboles binarios representan órdenes jerárquicos. Pero recordemos que el computador sólo procesa secuencias, es decir, no puede distinguir, en el mismo sentido de nuestra percepción, que un árbol es un árbol.

Consideremos el problema de contar el número de nodos que posee un árbol binario. Para escalas pequeñas como seres humanos no estamos restringidos por la topología del árbol. De hecho, podemos borrar todas las conexiones entre los arcos y aún somos capaces de contar los nodos. Consideremos ahora la evaluación de la expresión  $(x^2 + x + 1)2y$ , la cual puede representarse mediante un árbol binario similar al de la figura 4.11. Podemos comenzar la evaluación sobre cualquiera de los tres subárboles que poseen hojas. Por ejemplo, podríamos evaluar  $x^2$ , luego  $2y$ , luego  $x + 1$  y así sucesivamente. Notemos que este orden de evaluación no considera distancias entre ramas, por ejemplo, la que hay entre la rama  $x^2$  y  $2y$ .

Nuestra “visión” nos permite improvisar diversas secuencias de recorrido o procesamiento sobre el árbol. En un computador, empero, con la representación de nodo binario dada, la visión de éste está restringida por las siguientes condiciones:

1. El primer nodo observable es la raíz. Consecuentemente, si deseamos mirar algún subárbol en particular, entonces debemos recorrer cada generación descendiente

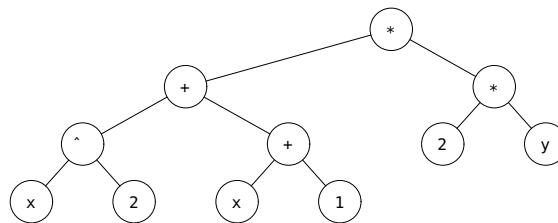


Figura 4.11: Expresión algebraica representada con un árbol binario

desde la raíz.

2. Desde un nodo cualquiera sólo se pueden ver sus dos hijos mediante los campos  $L(T)$  y  $R(T)$ .

Existen cuatro patrones de secuencias arquetípicas o “recorridos” para procesar un árbol binario.

#### Recorrido prefijo

1. Visite la raíz.
2. Visite la rama izquierda en prefijo.
3. Visite la rama derecha en prefijo.

#### Recorrido infijo

1. Visite la rama izquierda en infijo.
2. Visite la raíz.
3. Visite la rama derecha en infijo.

#### Recorrido sufijo

1. Visite la rama izquierda en sufijo.
2. Visite la rama derecha en sufijo.
3. Visite la raíz.

#### Recorrido por niveles

1. Repita desde  $i = 0$  hasta  $h(T) - 1$  del árbol
  - Imprima todos los nodos del nivel  $i$  ordenados de izquierda a derecha.

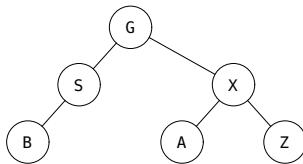


Figura 4.12: Árbol binario para ejemplificar recorridos

Muchos textos denominan los recorridos prefijo, infijo y sufijo como preorden, enorden y postorden respectivamente; anglicismos de preorder, inorder y postorder.

Aplicando las definiciones sobre el árbol de la figura 4.12 tenemos el siguiente recorrido prefijo:

$$G \ S \ B \ X \ A \ Z \quad (4.2)$$

Primero visitamos la raíz (G), luego, el subárbol izquierdo (S B), y luego, el subárbol derecho (X A Z).

Para el recorrido infijo tenemos:

$$B \ S \ G \ A \ X \ Z \quad (4.3)$$

Primero visitamos la rama izquierda en infijo (B S), luego, la raíz (G), y culminamos con el subárbol derecho (A X Z).

El recorrido sufijo es:

$$B \ S \ A \ Z \ X \ G \quad (4.4)$$

Primero visitamos la rama izquierda en sufijo (B S), luego, la rama derecha (A Z X) y, por último, la raíz (G).

Finalmente, el recorrido por niveles es:

$$G \ S \ X \ B \ A \ Z \quad (4.5)$$

A menudo, la secuencia del recorrido caracteriza la topología del árbol. En efecto, los recorridos están íntimamente relacionados con muchos de los algoritmos sobre árboles y caracterizan algo acerca de su forma. Consideremos el árbol de expresiones algebraicas mostrado en la figura 4.11. Su recorrido sufijo es:

$$x \ 2 \wedge \ x \ 1 \ + \ + \ 2 \ y \ * \ * \quad (4.6)$$

que es la expresión sufija de la expresión algebraica (vea § 2.5.5 (Pág. 106)).

El recorrido prefijo es:

$$* \ + \ \wedge \ x \ 2 \ + \ x \ 1 \ * \ 2 \ y \quad (4.7)$$

el cual puede evaluarse por la derecha de forma análoga a la evaluación de una expresión infija.

Ahora consideremos un algoritmo de impresión del árbol infijo.

**Algoritmo 4.2 (Impresión infija, parentizada, de un árbol binario)** La entrada del algoritmo es la raíz  $p$  de un árbol binario. La salida es el recorrido infijo, parentizado (véase § 4.2.2 (Pág. 229)).

El algoritmo es recursivo con prototipo `void imprimir(Node *nodo)`, donde `nodo` es la raíz del árbol a imprimir. La primera llamada debe ejecutarse con la raíz.

1. Si nodo == NULL termine.
2. Imprima " (".
3. imprimir(LLINK(nodo));
4. Imprima el símbolo contenido en nodo.
5. imprimir(RLINK(nodo));
6. Imprima ")".

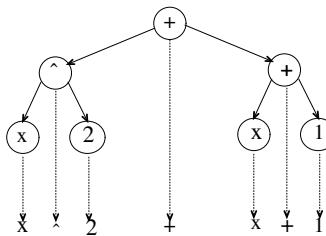


Figura 4.13: Proyección infija de un árbol binario

Para el árbol de la figura 4.11, el algoritmo arroja la secuencia:

$$(((x) \wedge (2)) + ((x) + (1))) + ((2) * (y))) ; \quad (4.8)$$

la cual es una secuencia infija, válida, de la expresión algebraica. En otras palabras, el algoritmo 4.2 obtiene una representación parentizada del árbol diferente a la representación dada en § 4.2.2 (Pág. 229). Esta representación también conserva la forma del árbol y permite reconstruir su forma original.

Una forma visual de interpretar el recorrido infijo, la cual requiere que los nodos estén suficientemente separados, consiste en imaginar la proyección de una luz sobre la raíz. El recorrido infijo se proyectará en un plano situado debajo del árbol. La figura 4.13 esquematiza la idea.

Consideremos ahora un problema inverso: dado algún recorrido cualquiera, ¿cómo obtener el árbol? La respuesta oficial es que no es posible. La respuesta oficial es que todo depende del tipo de árbol binario. De alguna manera, un recorrido, aunado al conocimiento acerca del tipo del árbol, conlleva suficiente información sobre su topología. En la ocurrencia, el recorrido prefijo permite identificar inmediatamente la raíz y la raíz del subárbol izquierdo. Pero esta información no es por sí sola suficiente para reconstruir el árbol original. Se requiere algo más.

Existen varios métodos para obtener la forma original. Todos requieren información adicional. Un primer método es añadir delimitadores al recorrido; la representación parentizada explicada en § 4.2.2 (Pág. 229) es un ejemplo; el algoritmo 4.2 de impresión infija parentizada es otro ejemplo.

¿Cuál es la diferencia entre las representaciones parentizadas de § 4.2.2 (Pág. 229) y la dada por el algoritmo 4.2? Para responder, observemos la representación parentizada del árbol de la figura 4.11:

$$(*(+(\wedge(x)(2))(+(x)(1))))(*(2)(y))) \quad (4.9)$$

Al eliminar los paréntesis, la expresión resultante es idéntica a la expresión (4.7), es decir, es el recorrido prefijo.

La representación parentizada de § 4.2.2 (Pág. 229) es una versión extendida del recorrido prefijo. Del mismo modo, la representación construida por el algoritmo 4.2 es una versión extendida del recorrido infijo. Ambas representaciones contienen información suficiente para reconstruir el árbol original.

Otra alternativa para reconstruir el árbol original consiste en combinar la información de dos recorridos diferentes sobre el mismo árbol, técnica ésta que explicaremos en § 4.4.13 (Pág. 253).

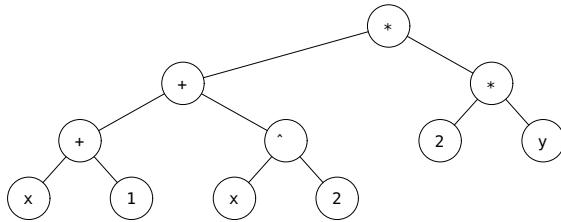


Figura 4.14: Árbol de operaciones equivalente al de la figura 4.11

Con el recorrido infijo normal, sin paréntesis, no es posible construir algún árbol porque es imposible conocer la precedencia.

Muchos tipos de árboles pueden construirse con tan solo el recorrido prefijo o sufijo. Existen casos en los cuales no es necesario obtener un árbol idéntico al original, pero sí equivalente a efectos de la abstracción. Un ejemplo notable está dado por los árboles de expresiones algebraicas; por ejemplo, el árbol de la figura 4.14 es diferente en forma pero equivalente en operación al árbol de la figura 4.11. A pesar de las ambigüedades es muy importante aprehender que, dada una expresión prefija o sufija, es posible construir automáticamente un árbol que represente correctamente la evaluación de la expresión. Esto es muy importante porque de alguna manera, la expresión sufija posee suficiente información para construir un árbol, ergo, para resolver el problema. En otras palabras, en el caso de las expresiones algebraicas podríamos optar por construir el árbol que efectúe la evaluación de izquierda a derecha. A efectos del problema, la solución está dada y la ambigüedad no es importante. Así pues, siempre podríamos trabajar con expresiones sufijas, una forma eficaz y muy compacta de almacenar expresiones algebraicas.

#### 4.4.3 Un TAD genérico para árboles binarios

Antes de abordar algoritmos que nos entrenen en el recorrido de árboles estableceremos un TAD sobre el cual fundamentaremos la mayoría de las estructuras de árbol binario de este texto.

Modelizaremos e implantaremos un mecanismo genérico para construir “nodos binarios”. A lo largo de este texto desarrollaremos diferentes estructuras de árbol binario. La mayoría de ellas siempre manejarán tres atributos básicos: una clave, un puntero a la rama izquierda y un puntero a la rama derecha. Según el tipo de árbol, el nodo puede contener alguna información adicional de control. Debemos entonces encontrar una forma de definir genéricamente nodos binarios de cualquier índole bajo los siguientes requerimientos:

1. **Soporte para atributos generales:** dado un nodo binario, debe poder accederse a la clave y a sus ramas descendientes. Estos atributos son comunes a todos las clases de nodos binarios posibles.
2. **Especificación de atributos opcionales:** eventualmente debe posibilitarse el declarar atributos propios de una clase particular de nodo binario.
3. **Soporte opcional para destructores virtuales:** el usuario puede trabajar opcionalmente con nodos que posean destructores virtuales.
4. **Soporte para nodos centinelas especiales<sup>1</sup>:** normalmente, el árbol vacío se señala mediante el valor especial NULL. Para ciertas estructuras de árbol es conveniente que este árbol vacío se represente mediante una instancia particular de nodo.

Desarrollaremos una solución basada en macros que se expanden a clases parametrizadas. Los macros se definen en el archivo `<tpl_binNode.H (never defined)>`.

`<tpl_binNode.H (never defined)>` exporta dos macros. El primero es el `DECLARE_BINNODE(Name, height, Control_Data)`, el cual genera dos clases parametrizadas que representan nodos binarios pertenecientes a alguna clase de árbol binario. El parámetro `Name` es el nombre prefijo de las clases que se desean generar. El macro se expande a dos clases casi idénticas: `Name` y `NameVtl`. La única diferencia es que `NameVtl` posee un destructor virtual.

El parámetro `height` representa un estimado de la altura máxima que puede alcanzar el árbol binario. Muchos algoritmos sobre árboles binarios son recursivos, por lo que el consumo de espacio en pila es un factor a considerar. En este sentido, el atributo `height` ofrece un aproximado de la altura máxima del árbol binario, cuyo valor sirve a los algoritmos a tomar previsiones acerca del consumo en pila.

Finalmente, el parámetro `Control_Data` es una clase que representa información de control del nodo binario pertinente a una especialización, la cual varía según el tipo de árbol binario que se maneje. Por ejemplo, los árboles rojo-negro, que estudiaremos en § 6.5 (Pág. 490), almacenan un color. Destaquemos que `Control_data` no está destinada a usarse por el usuario final.

Muchos tipos de árboles binarios se utilizan para construir mapeos entre claves de algún dominio y elementos de algún rango. En esta situación, lo que se requiere es que cada nodo almacene un elemento del rango dado. Puesto que nuestra pretensión es genérica, no podemos usar `Control_Data`, pues si no invalidaríamos, por ejemplo, un TAD árbol rojo genérico.

Una manera de hacer un mapeo es mediante herencia de interfaz. Supongamos que queremos un mapeo mediante árboles rojo negro. Por ejemplo, un mapeo entre números de cédula y apellidos puede especificarse del modo ilustra en la figura 4.15. En este caso implantamos un mapeo “fijo” en el sentido de que este contendría pares de tipo `[int, string]`. El `int` es la clave del nodo binario y el `string` se encuentra en la clase derivada.

Si deseamos un mapeo genérico, el cual es más completo, entonces la clase derivada debe ser una plantilla, lo cual, en el caso de nuestro ejemplo, puede especificarse como se ilustra en la figura 4.16.

---

<sup>1</sup>En inglés, centinela se escribe “sentinel”; (con “s”)

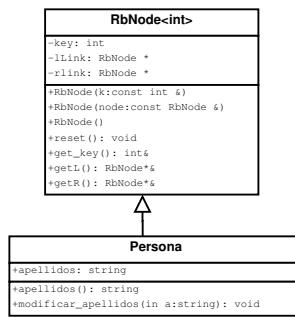


Figura 4.15: Atributos de nodo binario especificados por herencia de un objeto derivado de BinNode<Key>

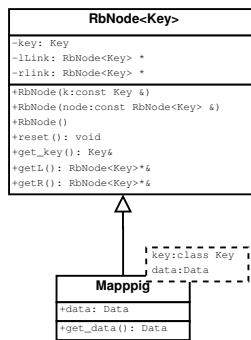


Figura 4.16: Atributos de un nodo binario genérico especificados por herencia de un objeto derivado de BinNode<Key>

Los árboles requieren un valor especial para representar el árbol vacío. Tal valor se almacena en el atributo estático `Name<T>::NullPtr`, el cual debe inicializarse explícitamente al valor de árbol vacío. Por omisión, `DECLARE_BINNODE()` asume que el árbol vacío es el valor `NULL`.

Algunas veces es conveniente que `NullPtr` apunte a un nodo centinela particular. En estas situaciones debemos usar el segundo macro `DECLARE_BINNODE_SENTINEL()`, cuya sintaxis es similar `DECLARE_BINNODE()`, pero que representa la instancia particular deseada para `NullPtr`.

El nodo binario más simple, llamado `BinNode<Key>`, se define en una sola línea:

```

240 <clase BinNode<Key> 240>≡
 DECLARE_BINNODE(BinNode, 2048, Empty_Node);
Defines:
 BinNode, used in chunks 319b, 347–52, and 509a.
 BinNodeVt1, never used.
Uses DECLARE_BINNODE.

```

La clase `BinNode<Key>` especifica el más simple nodo perteneciente a un árbol binario. Sin conocer el tipo de árbol binario que se trate, su altura es bastante variable y depende del orden en que las claves sean insertadas. Por esta razón, el tamaño de una pila que opere sobre un árbol binario debe ser lo suficientemente grande para que permita alturas elevadas. Como es muy difícil determinar la máxima altura que podría alcanzar un árbol

binario general, la fijamos en un valor grande, pero advertimos que esto no elimina la posibilidad de un desborde de pila.

`BinNode<Key>` no tiene atributos de control especiales; por eso, colocamos la clase vacía `Aleph::Empty_Node`, la cual está definida en la biblioteca.

Para facilitar la legibilidad, se exportan los siguientes macros:

241 *(macros externos de BinNode<Key> 241)≡*

```
define LLINK(p) ((p)->getL())
define RLINK(p) ((p)->getR())
define KEY(p) ((p)->get_key())
```

Dado un nodo `p`, el acceso a la clave se realiza con `KEY(p)`, el acceso a la rama izquierda con `LLINK(p)` y el acceso a la rama derecha con `RLINK(p)`.

Uno de los atributos de un nodo binario más importantes lo conforma el tipo de clave, el cual es genéricamente asequible mediante el nombre `key_type`. Una función genérica puede conocer el tipo de clave de un nodo mediante la instrucción:

```
typename Node::key_type
```

La cual refiere al tipo de clave del nodo. Mediante esta interfaz se puede diseñar código genérico sin necesidad de especificar ni conocer explícitamente el tipo de clave que alberga el nodo.

Supongamos que deseamos un nodo que guarde como atributo de control la altura del nodo. Definimos entonces una clase que contiene tal atributo y nos valemos de los macros de la siguiente forma:

```
class Altura
{
private:
 size_t altura;
public:
 Altura() { /* Empty */ }
 Altura(size_t a) : altura(a) { /* empty */ }
 size_t dar_altura() const { return altura; }
};

DECLARE_BINNODE(Nodo, 255, Altura);
SET_BINNODE_NULL_POINTER(NULL, Nodo);
```

El macro se expande a dos clases plantilla denominadas `Nodo<Key>` y `NodoVtl<Key>`, respectivamente. Las clases son casi idénticas, salvo que `NodoVtl<Key>` define un destructor virtual.

La estructura de `Nodo<Key>` es la siguiente:

```
template <typename Key, size_t _MaxHeight = 255>
class Nodo : public Altura
{
public:
 static Nodo * NullPtr;
 static const size_t MaxHeight = _MaxHeight;
 typedef Key key_type;

private:
 Key key;
 Nodo * lLink;
```

```

Nodo * rLink;

public:
 Key & get_key() { return key; }
 Nodo *&& getL() { return lLink; }
 Nodo *&& getR() { return rLink; }
 Nodo(const Key & k) : key(k), lLink(nullptr), rLink(nullptr) { }
 Nodo(const Altura & control_data, const Key& k) :
 Altura(control_data), key(k), lLink(nullptr), rLink(nullptr) { }
 Nodo(const Altura & control_data) :
 Altura(control_data), lLink(nullptr), rLink(nullptr) { }
 Nodo() : lLink(nullptr), rLink(nullptr) { }
 Nodo(EmptyCtor) { }
 void reset() { rLink = lLink = nullptr; }
};

template <typename Key, size_t _MaxHeight>
BinNode<Key, _MaxHeight> * BinNode<Key, _MaxHeight>::nullptr = NULL;

```

Las clases `Nodo<Key>` y `NodoVt1<Key>` representan familias de nodos binarios con atributos de control definidos en la clase `Altura` y con clave genérica `Key`. El acceso a la clave está dado por `get_key()` y los accesos a las ramas izquierda y derecha por `getL()` y `getR()`, respectivamente.

`Nodo<Key>` hereda públicamente de `Altura`. Por tanto, un `Nodo<Key>` es también de tipo altura. De este modo, `Nodo<Key>` tiene acceso a toda la interfaz pública de `Altura`.

Hay cinco maneras de construir un `Nodo<Key>` representadas por los cinco constructores generados los cuales, a excepción del quinto, se explican por sí solos. Si se requiere un nodo centinela, éste se instancia mediante el quinto constructor, el cual requiere que `Node::Node(SentinelCtor)` se defina en la clase `Altura`.

La plantilla `Nodo<Key>` tiene dos atributos estáticos. `Node::nullptr` es el apuntador al árbol vacío. En la mayoría de las situaciones, este puntero tendrá el valor nulo; en otras, `Node::nullptr` dirigirá al nodo centinela. El valor de `Node::nullptr` debe definirse explícitamente por el invocante de `DECLARE_BINNODE` mediante el macro `SET_BINNODE_NULL_POINTER`. Este macro toma como parámetros el nombre de la clase nodo y el puntero al nodo centinela.

El segundo atributo estático corresponde a un valor estimado de altura máxima, que puede ser requerido por algunos algoritmos para determinar el tamaño de sus pilas.

Un `Nodo<Key>` puede reutilizarse. Por ejemplo, podríamos extraerlo de un árbol e insertarlo en otro. En este caso, el estado del nodo debe ser reiniciado. Para ello se provee el método `reset()`.

#### 4.4.4 Contenedor de funciones sobre árboles binarios

Muchas operaciones sobre árboles binarios se encapsulan en una biblioteca especial que contendrá algoritmos típicos sobre árboles binarios y que se define en el archivo `tpl_binNodeUtils.H`.

La biblioteca contiene algoritmos típicos sobre árboles binarios; por ejemplo, ejecución de recorridos, duplicación de árboles, etcétera.

La mayor parte de las (*Funciones de BinNode\_Utils 243a*) son plantillas con la forma general siguiente:

```
template <class Node> función (...) { ... }
```

Es decir, el tipo parametrizado es un nodo binario genérico `Node`. El contrato mínimo para que la biblioteca opere es que `Node` exporte las funciones de `BinNode<Key>` definidas en `tpl_binNode.h`.

#### 4.4.5 Recorridos recursivos

La rutina de recorrido se remite a su función básica: recorrer. Así pues, una rutina de recorrido infijo tendría el siguiente prototipo:

```
template <class Node> inline
int inOrderRec(Node * root, void (*visitFct)(Node*, int, int))
```

La función `inOrderRec()` recorre en forma infija el árbol binario con raíz `root` y retorna el número de nodos visitados. Cada vez que se visita a un nodo, se efectúa una llamada a la función apuntada por `visitFct`, la cual debe corresponderse con el siguiente prototipo:

```
template <class Node>
void visitFct(Node* node, int level, int position)
```

`node` es un puntero al nodo visitado, `level` es el nivel o la profundidad del nodo respecto a la raíz y `position` es su posición dentro del recorrido.

`inOrderRec()` es un “wrapper” a una función recursiva, estática, encargada de efectuar el recorrido, la cual se deriva directamente de la definición:

243a *(Funciones de BinNode\_Utils 243a)≡* 243b ▷

```
template <class Node> inline static
void __inorder_rec(Node * node, const int& level, int & position,
 void (*visitFct)(Node *, int, int))
{
 if (node == Node::NullPtr)
 return;

 __inorder_rec((Node*) LLINK(node), level + 1, position, visitFct);

 (*visitFct)(node, level, position);
 ++position;

 __inorder_rec((Node*) RLINK(node), level + 1, position, visitFct);
}
```

El nivel `level` se incrementa en cada llamada, mientras que la `position` en cada visita. Notemos que `position` es un parámetro por referencia, pues éste requiere actualizarse en cada visita.

Para que la rutina sea consistente, la llamada inicial debe realizarse con valores de `level` y `position` iguales a cero. Este es el trabajo de `inOrderRec()`:

243b *(Funciones de BinNode\_Utils 243a)+≡* ◁243a 244▷

```
template <class Node> inline
int inOrderRec(Node * root, void (*visitFct)(Node*, int, int))
{
 int position = 0;
 __inorder_rec(root, 0, position, visitFct);
 return position;
}
```

`inOrderRec()` retorna el número de nodos que contiene el árbol.

Los recorridos prefijo y sufijo se definen de manera análoga:

```
244 <Funciones de BinNode_Utils 243a>+≡ ▷243b 245▷
 template <class Node> inline static
 void __preorder_rec (Node * p, const int & level, int & position,
 void (*visitFct)(Node*, int, int))
 {
 if (p == Node::NullPtr)
 return;

 (*visitFct)(p, level, position);
 ++position;

 __preorder_rec((Node*) LLINK(p), level + 1, position, visitFct);
 __preorder_rec((Node*) RLINK(p), level + 1, position, visitFct);
 }
 template <class Node> inline
 int preOrderRec(Node * root, void (*visitFct)(Node*, int, int))
 {
 int position = 0;
 __preorder_rec(root, 0, position, visitFct);
 return position;
 }
 template <class Node> inline static
 void __postorder_rec(Node * node, const int & level, int & position,
 void (*visitFct)(Node*, int, int))
 {
 if (node == Node::NullPtr)
 return;

 __postorder_rec((Node*) LLINK(node), level + 1, position, visitFct);
 __postorder_rec((Node*) RLINK(node), level + 1, position, visitFct);

 (*visitFct)(node, level, position);
 ++position;
 }
 template <class Node> inline
 int postOrderRec(Node * root, void (*visitFct)(Node*, int, int))
 {
 int position = 0;
 __postorder_rec(root, 0, position, visitFct);
 return position;
 }
```

Todas las primitivas de recorrido de `<Funciones de BinNode_Utils 243a>` tienen como primer parámetro la raíz del árbol binario. El segundo parámetro es una función de visita del nodo que provee el cliente de la biblioteca. La función será invocada durante la visita acorde a su recorrido.

Notemos que la condición de parada de la recursión es el visitar un nodo nulo.

Una interfaz alterna, que según el compilador puede ser un poco más costosa en desempeño pero mucho más versátil, consiste en exportar la interfaz bajo una clase. En este

caso, el recorrido infijo se instrumentaría así:

```
245 <Funciones de BinNode_Utils 243a>+≡ ◁244 246▷
 template <class Node, class Op> class For_Each_In_Order
 {
 static void for_each_inorder(Node * root, Op & op)
 {
 if (root == Node::NullPtr)
 return;

 for_each_inorder((Node*) LLINK(root), op);
 op(root);
 for_each_inorder((Node*) RLINK(root), op);
 }
 public:
 void operator () (Node * root, Op & op)
 {
 for_each_inorder(root, op);
 }
 };
};

template <class Node, class Op> class For_Each_Preorder
{
 static void preorder(Node * root, Op & op)
 {
 if (root == Node::NullPtr)
 return;

 op(root);
 preorder((Node*) LLINK(root), op);
 preorder((Node*) RLINK(root), op);
 }

 public:

 void operator () (Node * root, Op & op)
 {
 preorder(root, op);
 }

 void operator () (Node * root, const Op & op) const
 {
 preorder(const_cast<Node*>(root), const_cast<Op&>(op));
 }
 };

 template <class Node, class Op> class For_Each_Postorder
 {
 static void postorder(Node * root, Op & op)
 {
 if (root == Node::NullPtr)
 return;
```

```

 postorder((Node*) LLINK(root), op);
 postorder((Node*) RLINK(root), op);
 op(root);
}

public:
 void operator () (Node * root, Op & op)
 {
 postorder(root, op);
 }

 void operator () (Node * root, const Op & op) const
 {
 postorder(root, const_cast<Op&>(op));
 }
};

```

Este patrón ofrece la ventaja de que la operación es toda una “clase” de objeto, con la posibilidad de especificar el constructor y el destructor de la clase Op, lo cual permite pre y postprocesamiento.

#### 4.4.6 Recorridos no recursivos

El estudio de los recorridos es uno de los mejores entrenamientos en algorítmica de árboles, pues otros algoritmos obedecen a patrones de un recorrido o a combinaciones de ellos. Comencemos por la versión más simple del recorrido prefijo:

```

246 <Funciones de BinNode_Utils 243a>+≡ <245 247>
 template <class Node> inline
 size_t simple_preOrderStack(Node * node, void (*visitFct)(Node *, int, int))
 {
 if (node == Node::NullPtr)
 return 0;

 ArrayStack<Node *, Node::MaxHeight> stack;
 stack.push(node);

 Node * p;
 size_t count = 0;
 while (not stack.is_empty())
 {
 p = stack.pop();

 (*visitFct) (p, stack.size(), count++);

 if (RLINK(p) != Node::NullPtr)
 stack.push(RLINK(p));

 if (LLINK(p) != Node::NullPtr)
 stack.push(LLINK(p));
 }
 }

```

```

 return count;
}

```

Uses ArrayStack 101a.

El patrón iterativo del algoritmo es simple: visite, introduzca el árbol derecho, luego el izquierdo y saque de la pila. Puesto que el árbol izquierdo fue introducido después del derecho, el izquierdo será el primero a extraerse y visitarse.

Observemos que el nivel del nodo se corresponde con el tamaño actual de la pila

Ahora bien, según la definición del recorrido prefijo, el árbol izquierdo se visita inmediatamente después de visitar el nodo. Podemos entonces mejorar el algoritmo si, en lugar de guardar en pila el árbol derecho guardamos su padre y asumimos que todo nodo extraido de la pila ya fue visitado. Este patrón conduce al algoritmo siguiente:

247

```

⟨Funciones de BinNode_Utils 243a⟩+≡ ◁246 248a▷
 template <class Node> inline
 size_t preOrderStack(Node * node, void (*visitFct)(Node *, int, int))
 {
 if (node == Node::NullPtr)
 return 0;

 ArrayStack<Node *, Node::MaxHeight> stack;
 Node *p = node;
 size_t count = 0;

 while (true)
 {
 (*visitFct)(p, stack.size(), count++);

 if (LLINK(p) != Node::NullPtr)
 {
 stack.push(p); // p y RLINK(p) faltan por visitarse
 p = LLINK(p); // avanzar a la izquierda
 continue; // ir a visitar raíz rama izquierda
 }
 while (true)
 {
 if (RLINK(p) != Node::NullPtr)
 {
 p = RLINK(p); // avanzar a la derecha
 break; // ir a visitar raíz rama derecha
 }
 if (stack.is_empty())
 return count; // fin

 p = stack.pop(); // sacar para ir a rama derecha
 }
 }
 }

```

Uses ArrayStack 101a.

De una cierta manera, todo recorrido tiene algo de prefijo, pues todo árbol se accede a través de las raíces de sus subárboles. Por tanto, es intuitivamente esperable que el

recorrido infijo tenga una estructura similar al prefijo, pues la diferencia esencial es el momento en que se visita la raíz.

El recorrido prefijo visita la raíz antes de continuar por la rama izquierda, mientras que el infijo lo hace después de venir desde la rama izquierda. Dicho esto podemos presentar la versión no recursiva del recorrido infijo:

248a *<Funciones de BinNode\_Utils 243a>* +≡ ◀247 249a▶

```

template <class Node> inline
size_t inOrderStack(Node * node, void (*visitFct)(Node *, int, int))
{
 if (node == Node::NullPtr)
 return 0;

 ArrayStack<Node *, Node::MaxHeight> stack;
 Node *p = node;
 size_t count = 0;

 while (true)
 {
 if (LLINK(p) != Node::NullPtr)
 {
 stack.push(p); // p y RLINK(p) faltan por visitarse
 p = LLINK(p); // avanzar a la izquierda
 continue; // continuar bajando por la rama izquierda
 }
 while (true)
 {
 (*visitFct)(p, stack.size(), count);

 if (RLINK(p) != Node::NullPtr)
 {
 p = RLINK(p); // avanzar a la derecha
 break; // ir a visitar raíz rama derecha
 }
 if (stack.is_empty())
 return count;

 p = stack.pop(); // sacar para ir a rama derecha
 }
 }
}
```

Uses ArrayStack 101a.

El recorrido sufijo es más complejo y costoso porque al desempilar hay que verificar si se proviene desde la izquierda o la derecha. Una manera de distinguir esta clase de proveniencia consiste en marcar el nodo empilado con el sentido del recorrido. De este modo podemos definir la siguiente pila:

248b *<Declaración de pila sufija 248b>* ≡

```

typedef Aleph::pair<Node*, char> Postorder_Pair;
ArrayStack<Postorder_Pair, Node::MaxHeight> stack;
```

Uses ArrayStack 101a.

El char de la pila guarda los valores siguientes:

- 'i': Indica que se empiló antes de descender por la rama izquierda.
- 'l': Indica que se empiló de regreso de la rama izquierda.
- 'r': Indica que se empiló de regreso de la rama derecha. Es en este caso que se debe visitar el nodo.

Un recorrido sufijo basado en estos principios está implantado en la biblioteca bajo el nombre de `postOrderStack()`. Véase el fuente para mayores detalles sobre la instrumentación de este algoritmo.

#### 4.4.7 Cálculo de la cardinalidad

Probablemente, la aplicación más sencilla de los recorridos es calcular la cardinalidad de un árbol binario, la cual puede definirse recursivamente como:

$$|T| = \begin{cases} 0 & \text{si } T = \emptyset \\ |L(T)| + 1 + |R(T)| & \text{si } T \neq \emptyset \end{cases} \quad (4.10)$$

De esta definición se deriva directamente el algoritmo:

249a *(Funciones de BinNode\_Utils 243a) +≡* ◀248a 249b▶  
 template <class Node> inline size\_t compute\_cardinality\_rec(Node \* node)  
 {  
 if (node == Node::NullPtr)  
 return 0;  
 return (compute\_cardinality\_rec(LLINK(node)) + 1 +  
 compute\_cardinality\_rec(RLINK(node)));  
}

#### 4.4.8 Cálculo de la altura

Recursivamente, la altura de un árbol binario  $T$  se define como:

$$h(T) = \begin{cases} 0 & \text{si } T = \emptyset \\ 1 + \max(h(L(T)), h(R(T))) & \text{si } T \neq \emptyset \end{cases} \quad (4.11)$$

De este modo diseñamos un algoritmo completamente reminiscente de la definición:

249b *(Funciones de BinNode\_Utils 243a) +≡* ◀249a 250a▶  
 template <class Node> inline size\_t computeHeightRec(Node \* node)  
 {  
 if (node == Node::NullPtr)  
 return 0;  
 const size\_t left\_height = computeHeightRec(LLINK(node));  
 const size\_t right\_height = computeHeightRec(RLINK(node));  
  
 return 1 + std::max(left\_height, right\_height);  
}

#### 4.4.9 Copia de árboles binarios

Dado un árbol binario  $T$ , “copiarlo” consiste en obtener un árbol binario  $T'$  cuya forma y contenido se corresponda exactamente con  $T$ . Este algoritmo puede definirse recursivamente como sigue:

250a *(Funciones de BinNode\_Utils 243a) +≡* ◀249b 250b▶

```
template <class Node> inline Node * copyRec(Node * src_root)
{
 if (src_root == Node::NullPtr)
 return (Node*) Node::NullPtr;
 Node * tgt_root = new Node(*src_root);
 LLINK(tgt_root) = copyRec<Node>((Node*) LLINK(src_root));
 RLINK(tgt_root) = copyRec<Node>((Node*) RLINK(src_root));
 return tgt_root;
}
```

En este código es importante comentar el manejo recursivo de tipos. Notemos que `copyRec()` copia árboles genéricos con nodos de tipo `Node`, el cual podría derivar de una clase de nodo de árbol binario. Por esa razón, la llamada recursiva `copyRec<Node>(LLINK(src_root))` debe especificar explícitamente el tipo `Node`, pues si no, en el caso de que se trate de una clase derivada, el compilador invocaría `copyRec()` con la clase base a la cual correspondan `LLINK(tgt_root)` y a `RLINK(tgt_root)`. Si la clase `Node` es distinta a la clase que retornan `LLINK(tgt_root)` y `RLINK(tgt_root)`, entonces ocurriría un conflicto de tipos.

La copia es un algoritmo fundamental para implantar el constructor copia y el operador de asignación en tipos abstractos que manipulen árboles binarios.

#### 4.4.10 Destrucción de árboles binarios

Cualquier TAD que manipule árboles binarios debe estar en capacidad de destruir todo el árbol. Es decir, de invocar el destructor y liberar la memoria ocupada por cada nodo del árbol. Esto se hace de la siguiente forma:

250b *(Funciones de BinNode\_Utils 243a) +≡* ◀250a 251a▶

```
template <class Node> inline void destroyRec(Node *& root)
{
 if (root == Node::NullPtr)
 return;
 destroyRec((Node*&) LLINK(root));
 destroyRec((Node*&) RLINK(root));
 delete root;
 root = (Node*) Node::NullPtr;
}
```

¿Qué tipo de recorrido exhibe este algoritmo? Claramente, la forma es sufija: la raíz se libera luego de liberar las dos ramas. Otras variantes recursivas de este algoritmo, que se ajusten a los demás recorridos, son posibles y delegadas en ejercicios.

#### 4.4.11 Comparación de árboles binarios

A veces es necesario comparar dos árboles binarios. Básicamente, hay dos criterios, los cuales presentaremos a continuación.

#### 4.4.11.1 Similaridad

Dados dos árboles binarios  $T_1$  y  $T_2$ , se dice que  $T_1$  es similar a  $T_2$ , y se denota como:

$$T_1 \parallel T_2 \iff \begin{cases} T_1 = T_2 = \emptyset \\ T_1 \neq T_2 \neq \emptyset \quad \wedge \quad L(T_1) \parallel L(T_2) \quad \wedge \quad R(T_1) \parallel R(T_2) \end{cases} \vee \quad (4.12)$$

Esta definición sugiere el siguiente código recursivo:

251a *(Funciones de BinNode\_Utils 243a) +≡* △250b 251b ▷  

```
template <class Node> inline bool areSimilar(Node * t1, Node * t2)
{
 if (t1 == Node::NullPtr and t2 == Node::NullPtr)
 return true;
 if (t1 == Node::NullPtr or t2 == Node::NullPtr)
 return false;
 return (areSimilar(LLINK(t1), LLINK(t2)) and
 areSimilar(RLINK(t1), RLINK(t2)));
}
```

#### 4.4.11.2 Equivalencia

Dados dos árboles binarios  $T_1$  y  $T_2$ , se dice que  $T_1$  es equivalente a  $T_2$ , denotado como  $T_1 \equiv T_2$ , si y sólo si:

1. Si  $T_1 = T_2 = \emptyset$  o
2. Si  $T_1 \neq T_2 \implies T_1 \equiv T_2 \iff$ :

$$\text{KEY(raiz}(T_2)\text{)} \wedge L(T_1) \equiv L(T_2) \wedge R(T_1) \equiv R(T_2)$$

Es decir,  $T_1$  y  $T_2$  son similares y los contenidos de cada nodo son exactamente los mismos. La definición de equivalencia nos arroja la siguiente implantación recursiva:

251b *(Funciones de BinNode\_Utils 243a) +≡* △251a 251c ▷  

```
template <class Node, class Equal> inline
bool areEquivalents(Node * t1, Node * t2)
{
 if (t1 == Node::NullPtr and t2 == Node::NullPtr)
 return true;
 if (t1 == Node::NullPtr or t2 == Node::NullPtr)
 return false;
 if (not Equal () (KEY(t1), KEY(t2)))
 return false;
 return (areEquivalents<Node, Equal>(LLINK(t1), LLINK(t2)) and
 areEquivalents<Node, Equal>(RLINK(t1), RLINK(t2)));
}
```

#### 4.4.12 Recorrido por niveles

En esta clase de recorrido, una cola describe directamente el orden en que se deben procesar los nodos:

251c *(Funciones de BinNode\_Utils 243a) +≡* △251b 253 ▷

```

template <class Node> inline
void levelOrder(Node * root,
 void (*visitFct)(Node*, int, bool), const size_t & queue_size)
{
 const size_t two_power =
 (size_t) (std::log((float)queue_size)/std::log(2.0) + 1);
 ArrayQueue<Aleph::pair<Node*, bool> > queue(two_power);
 queue.put(root);

 for (int pos = 0; not queue.is_empty(); pos++)
 {
 Aleph::pair<Node*, bool> pr = queue.get();
 Node *& p = pr.first;

 (*visitFct) (p, pos, pr.second);

 if (LLINK(p) != Node::NullPtr)
 queue.put(Aleph::pair <Node*, bool> (LLINK(p), true));

 if (RLINK(p) != Node::NullPtr)
 queue.put(Aleph::pair <Node*, bool> (RLINK(p), false));
 }
}

```

Uses ArrayQueue 125a.

Es posible, en detrimento de mucho más tiempo mas no del espacio, diseñar un algoritmo recursivo para recorrer por niveles. Esta versión se delega en ejercicio.

La función de visita tiene el siguiente prototipo:

```
(*visitFct)(Node* p, int pos, bool is_left)
```

p es el nodo visitado, pos es la posición del nodo visitado dentro del recorrido y is\_left es un valor lógico cuyo valor es true si el nodo es izquierdo. Una de las aplicaciones típicas del recorrido por niveles es para el dibujado de árboles, en cuyo caso es útil saber si p es o no un hijo izquierdo.

La lógica del algoritmo es aprovechar la propiedad FIFO de la cola para garantizar el orden de visita requerido. Está claro que el primer nodo a visitar es la raíz, por lo que la metemos en la cola y comenzamos. Cada vez que se visite un nodo, todos sus compañeros de nivel o generación deben ser los siguientes a extraer de la cola. Esto se garantiza porque cada vez que visitamos un nodo introducimos en la cola su hijo izquierdo y luego su hijo derecho. Estos hijos van a estar en la cola después de los ancestros de su nivel anterior y después de sus compañeros de nivel que estén del lado izquierdo. En la figura 4.17, los próximos nodos en la cola son los hermanos derechos del actual, y fueron introducidos cuando se estaba en el nivel uno. Después vienen los nodos del nivel tres que fueron introducidos cuando se procesaron los hermanos izquierdos del nodo actual.

Este recorrido es el más costoso de los cuatro, pues puede hacer falta almacenar muchísimos nodos en la cola. Si las hojas del árbol tienden a estar en el mismo nivel, entonces, por cada nivel que se descienda se requerirá almacenar el doble de la cantidad de nodos hasta llegar al nivel donde se encuentren las hojas.

Supongamos que la cantidad de nodos de cada nivel es potencia exacta de dos; o sea, cada nivel del árbol está lleno. Entonces, asumiendo altura conocida h podemos expresar

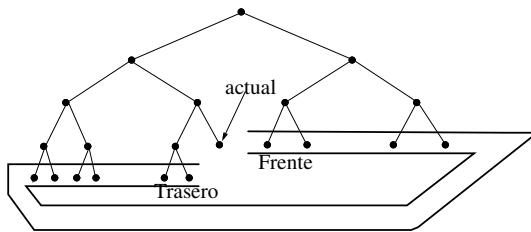


Figura 4.17: Estado de la cola cuando se procesa el nodo indicado con flecha

la cantidad de nodos  $n$  como  $\sum_{i=0}^{h-1} 2^i$ . Es decir, tendremos  $2^{h-1}$  nodos almacenados en la cola cuando nos toque procesar el último nivel.

Puesto que la forma de un árbol puede ser bastante arbitraria, la primitiva `levelOrder()` recibe como parámetro el tamaño máximo de la cola y delega al cliente el estimar un tamaño adecuado según su aplicación.

#### 4.4.13 Construcción de árboles binarios a partir de recorridos

Consideremos el problema de almacenar en una secuencia un árbol representado en memoria. Este problema es de alto interés en situaciones donde requiramos guardar el árbol en un archivo o transmitirlo en un mensaje<sup>2</sup>.

Cualquier técnica que secuencialice un árbol (lo almacene en una secuencia) está basada en sus recorridos. En § 4.4.2 (Pág. 234) estudiamos algunas técnicas que permiten reconstruir un árbol binario a partir de uno de sus recorridos. Nos bastaba con un sólo recorrido porque conocíamos el fin del árbol. Si este conocimiento no es asequible, o si queremos secuenciar de manera general, entonces hay dos maneras de salvar el árbol. Una consiste el calcular su “secuencia código” y luego almacenar las claves en prefijo; técnica que se revelará y estudiará en § 4.8.0.7 (Pág. 302). La segunda consiste en guardar dos recorridos cualesquiera y a partir de ellos reconstruir el árbol original.

Puesto que ya hemos estudiado los recorridos, nos debe ser sencillo vislumbrar rutinas que tomen un árbol y lo secuencialicen. Lo que quizás no sean tan fácil es la inversión; es decir, reconstruir el árbol a partir de sus secuencias.

La siguiente rutina reconstruye un árbol binario a partir de sus recorridos prefijo e infijo:

```
253 <Funciones de BinNode_Utils 243a>+≡ ◁251c 254b▷
 template <template <class> class Node, typename Key> inline
 Node<Key> * build_tree(DynArray<Key> & preorder,
 const int & l_p, const int & r_p,
 DynArray<Key> & inorder,
 const int & l_i, const int & r_i)
 {
 if (l_p > r_p) // ¿está vacío el recorrido?
 return Node<Key> ::NullPtr;

 Node<Key> * root = new Node<Key> (preorder[l_p]); // crear la raíz
 if (r_p == l_p)
```

<sup>2</sup>Nótese que según la equivalencia entre árboles y árboles binarios es suficiente con estudiar los binarios.

```

 return root; // recorrido de longitud 1

⟨Calcule en i la posición de preorder[l_p] en inorder[] 254a⟩

LLINK(root) = build_tree <Node, Key> (preorder, l_p + 1, l_p + i,
 inorder, l_i, l_i + (i - 1));
RLINK(root) = build_tree <Node, Key> (preorder, l_p + i + 1, r_p,
 inorder, l_i + i + 1, r_i);
return root;
}

```

Uses DynArray 34.

preorder[] es un arreglo que contiene el recorrido prefijo entre los índices l\_p y r\_p. Del mismo modo, inorder[] es un arreglo que contiene el recorrido infijo entre los índices l\_i y r\_i. La rutina retorna la raíz del nuevo árbol binario.

La lógica del algoritmo es simple. El primer elemento de un recorrido prefijo es la raíz del árbol, la cual buscamos en el recorrido infijo y obtenemos el desplazamiento i. El índice i divide el recorrido infijo en dos partes: los elementos a la izquierda que son l\_i .. i - 1 y que pertenecen a la rama izquierda, mientras que los elementos a la derecha son i + 1 .. r\_i y que pertenecen a la rama derecha. Ambos tipos de recorrido no avanzan a visitar la rama derecha hasta que no se haya visitado la totalidad del subárbol izquierdo y la raíz. Por tanto, justo antes de comenzar a visitar la rama derecha, ambos recorridos han visitado la misma cantidad de nodos (la rama izquierda y la raíz). Esto evidencia que los recorridos prefijo e infijo de la rama derecha están contenidos en ambos arreglos entre l\_p + i + 1 y r\_p.

La detección de árbol vacío se hace cuando los recorridos son vacíos, es decir, cuando los índices de los recorridos se cruzan.

La parte más delicada es buscar la posición relativa de la raíz dentro del recorrido infijo. Esto lo efectuamos cuidadosamente mediante inspección secuencial:

254a ⟨Calcule en i la posición de preorder[l\_p] en inorder[] 254a⟩≡ (253)

```

int i = 0;
for (int j = l_i; j <= r_i; ++j)
 if (inorder[j] == preorder[l_p])
 {
 i = j - l_i;
 break;
 }

```

#### 4.4.14 Conjunto de nodos en un nivel

En algunas situaciones es necesario conocer cuáles son los nodos de un árbol que se encuentran en un determinado nivel i; por ejemplo, cuando se dibujan árboles, a efectos de ajustar las posiciones de los nodos en un nivel determinado.

La siguiente rutina recorre recursivamente el árbol con raíz root y guarda en la lista level\_list los nodos que se encuentren en el nivel level. El parámetro current\_level denota el nivel del nodo que se está visitando <sup>3</sup>:

254b ⟨Funciones de BinNode\_Utils 243a⟩+≡ ◁253 255▷

```

template <class Node> inline static void

```

<sup>3</sup>Equivale a la profundidad recursiva.

```

__compute_nodes_in_level(Node * root, const int & level,
 const int & current_level,
 DynDlist<Node*> & level_list)
{
 if (root == Node::NullPtr)
 return;
 if (current_level == level)
 {
 level_list.append(root);
 return; // no vale la pena descender más
 }
 __compute_nodes_in_level(LLINK(root), level, current_level + 1, level_list);
 __compute_nodes_in_level(RLINK(root), level, current_level + 1, level_list);
}

```

Uses DynDlist 85a.

El árbol se recorre en prefijo y los nodos se guardan en la lista por nivel de izquierda a derecha. A diferencia del recorrido por niveles, no es necesario usar una cola.

La rutina anterior debe llamarse por la que fungirá de interfaz, la cual se especifica de la siguiente forma:

255   *Funciones de BinNode\_Utils 243a* +≡                                  ◁254b 259a ▷

```

template <class Node> inline
void compute_nodes_in_level(Node * root, const int & level,
 DynDlist<Node*>& list)
{
 __compute_nodes_in_level(root, level, 0, list);
}

```

Uses DynDlist 85a.

#### 4.4.15 Hilado de árboles binarios

Es fácil deducir que un árbol binario de  $n$  nodos consume  $2n$  apuntadores. Pero quizás no sea tan fácil aprehender que para cualquier árbol binario la cantidad total de punteros con el valor NULL siempre es mayor que la cantidad de punteros asignados. Más adelante (proposición § 4.2 (Pág. 277)) demostraremos la veracidad de esta afirmación.

Asumiendo veraz la afirmación anterior, algunos han cuestionado el desperdicio de espacio causado por los punteros nulos. A tenor del ahorre pueden considerarse las siguientes perspectivas:

1. Podemos tener tres tipos de nodos: completo, incompleto y hoja. Cada nodo ocuparía el espacio exacto según el tipo de nodo. Por supuesto, cada nodo tendría que tener un campo adicional que indique su tipo.

Esta solución es una alternativa importante y apelable si hay requerimientos críticos de espacio. Adolece sin embargo de estatismo en el sentido de que si el árbol cambia dinámicamente, entonces el tipo de nodo puede cambiar. También las operaciones sobre estos árboles son más complejas porque éstas deben distinguir los tipos de nodos.

2. Utilizar los apuntadores nulos para almacenar información adicional. Aquí surge las preguntas ¿cuál información?, ¿para qué?

Perlis y Thornton [141] -reportado por Knuth [97]- descubrieron un uso ingenioso de los punteros nulos. El método consiste en reemplazar punteros nulos por “hilos” que vayan hacia otras partes del árbol. La idea es facilitar el recorrido y, consecuentemente, los algoritmos.

Dado un nodo  $p$ , incompleto u hoja, el principio está dado por las dos reglas siguientes:

- $\text{LLINK}(p)$  apunta al nodo predecesor infijo.
- $\text{RLINK}(p)$  apunta al nodo sucesor infijo.

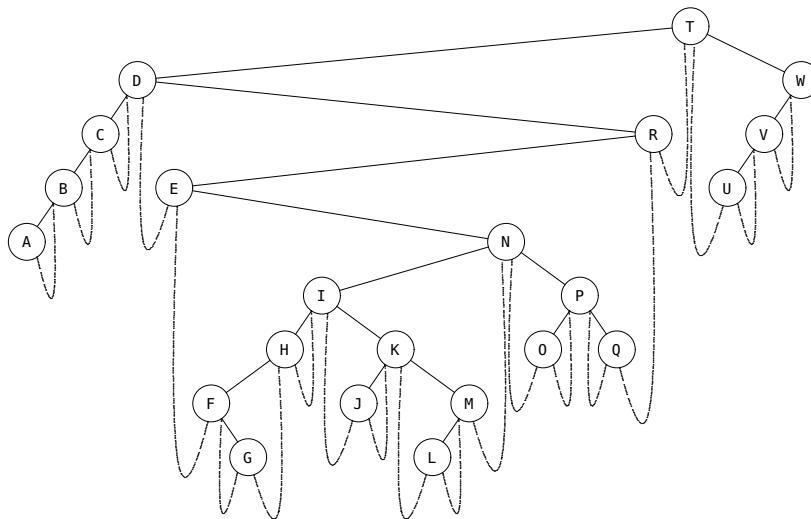


Figura 4.18: Ejemplo de árbol hilado

La figura 4.18 ilustra el “hilado” de un árbol. Los hilos son representados mediante líneas cortadas.

Ahora estudiemos cómo debe realizarse el recorrido infijo. Para ello veamos un algoritmo que busque el sucesor infijo.

**Algoritmo 4.3 (Búsqueda del sucesor en un árbol hilado)** La entrada del algoritmo es un apuntador  $p$  a un nodo. La salida es el sucesor infijo de  $p$ .

1. Si  $\text{RLINK}(p)$  es un hilo  $\Rightarrow$  retorne  $\text{RLINK}(p)$ .
2. Si  $\text{RLINK}(p)$  es el valor centinela fin de recorrido  $\Rightarrow$  retorne NULL.
3.  $p = \text{RLINK}(p)$
4. Repita mientras que  $\text{LLINK}(p)$  no sea un hilo
  - $p = \text{LLINK}(p)$
5. Retorne  $p$

La ventaja del hilado es la facilidad para recorrer el árbol. El recorrido infijo requiere determinar el primer elemento del recorrido, luego en llamar sucesivamente el algoritmo 4.3 hasta llegar al último elemento.

El hilado requiere distinguir si un apuntador es un hijo o un hilo. Aunque un bit por puntero es suficiente para especificar tal distinción, los sistemas de memoria de los computadores modernos trabajan por bytes. Se requiere al menos un byte para utilizar los dos bits necesarios. Las máquinas CISC puras no restringen el tamaño de un bloque de memoria; es posible, pues, apartar el byte extra. En máquinas RISC, los registros y sus campos deben estar alineados al tamaño de la palabra de memoria. En consecuencia, añadir un byte extra puede significar un incremento al tamaño del registro mayor a un byte, pues habría que alinear el byte extra hacia una dirección que sea adecuada al tamaño de la palabra.

Una técnica para reconocer hijos de hilos consiste en efectuar una transformación, inversible, que lleve el valor de un apuntador a un valor comprendido dentro de las zonas externas al manejador de memoria. Cuando se examina un apuntador, se verifica si éste se encuentra dentro del rango válido; si tal es el caso, entonces se trata de un apuntador; de lo contrario se trata de un hilo y, por supuesto, debe hacerse la transformación inversa antes de referenciar el apuntador. Un obstáculo a esta técnica es la portabilidad entre diferentes plataformas de hardware, sistema operativo, compilador y lenguaje.

La transformación tradicional consiste en restar a una dirección un valor que asegure que la resta estará dentro de la zona de datos o código o, por desbordamiento, dentro de la zona de la pila. La resta es posible sólo si el tamaño de la zona de memoria es menor o igual a la suma de las otras zonas. A veces, esto no es posible.

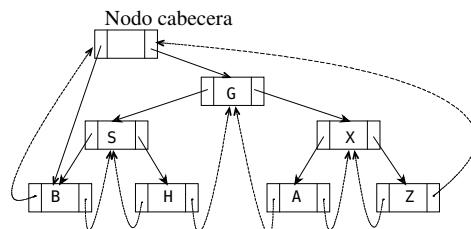


Figura 4.19: Representación en memoria de un árbol binario hilado

Algunas antiguas máquinas CISC tenían el bit más significativo reservado para el signo. Como los apuntadores utilizaban innecesariamente este signo, pues las direcciones negativas no tienen sentido, este bit podía utilizarse para denotar si el apuntador era o no un hilo. Las máquinas modernas son RISC, y las pocas CISC que existen tienen muchas funcionalidades RISC. Hoy en día, la alineación de las direcciones de memoria es una de las funcionalidades comunes entre los dos tipos de máquina. La alineación implica que las direcciones de memoria -los valores de los apuntadores- siempre están alineados al tamaño de la palabra de memoria. En la mayoría de arquitecturas modernas, la longitud de la palabra en bits siempre es potencia exacta de dos, es decir, puede expresarse como  $2^n$ , que es igual a  $2^n - \frac{2^n}{8}$  bytes. Esto implica que todo apuntador válido siempre tendrá los  $n - 3$  bits menos significativos colocados en cero. Estos bits pueden utilizarse para almacenar información adicional; en la ocurrencia, el bit menos significativo de un apuntador puede fungir como indicador que señale si el apuntador es o no un hilo.

Hay varias maneras de manipular el bit menos significativo de un apuntador. Quizá la más sencilla es mediante operaciones lógicas sobre el apuntador como sigue.

*(Determinar si un apuntador es un hilo 257)≡*

```

template <class Node> inline bool isThread(Node * p)
{
 return (Node*) (((long) p) & 1);
}

258a (Convertir un apuntador en un hilo 258a)≡
template <class Node> inline Node * makeThread(Node * p)
{
 return (Node*) (((long)p) | 1);
}

258b (Convertir un hilo en apuntador 258b)≡
template <class Node> inline Node * makePointer(Node * p)
{
 return (Node*) (((long) p) & -2);
}

```

De alguna manera, el hilado plantea una analogía equivalente a la diferencia entre las listas enlazadas circulares y las no circulares. En este sentido, un árbol hilado se parece a una lista doblemente enlazada. Al igual que en las listas circulares, es bastante deseable utilizar un nodo cabecera. El lazo izquierdo del nodo cabecera apuntaría al primer nodo infijo. El lazo derecho apuntaría a la raíz. El hilo izquierdo del primer nodo infijo y el lazo derecho del último nodo infijo apuntarían al nodo cabecera.

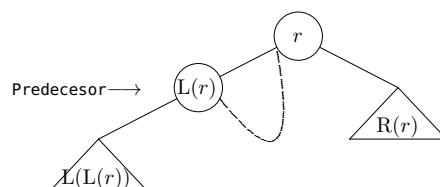
La figura 4.19 ilustra un ejemplo de la representación en memoria de un árbol hilado utilizando un nodo cabecera. Para notar la reminiscencia con una lista enlazada, tome el árbol por los nodos B y Z y “estire”.

#### 4.4.16 Recorridos pseudohilados

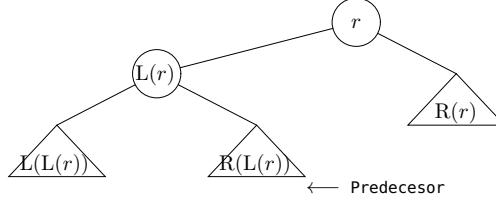
Si bien la representación hilada de árboles tiene sus bondades sobre el rendimiento de los recorridos, el hilado es más difícil de implantar que la representación tradicional. Si optásemos por la representación tradicional, ¿existiría aún alguna forma de aprovechar los punteros nulos “desperdiciados”? La respuesta es afirmativa: los punteros nulos pueden usarse como hilos temporales.

Dado un árbol binario con raíz  $r$ , ubiquemos cuál es su predecesor infijo. Esto es equivalente a determinar cuál es el último nodo que se visita en infijo de su rama izquierda. Podemos determinar tres casos:

- Si  $L(r) = \text{NULL} \Rightarrow r$  no tiene predecesor.
- Si  $L(r) \neq \text{NULL} \Rightarrow$  tenemos dos casos:
  1. Si  $R(L(r)) = \text{NULL} \Rightarrow L(r)$  es predecesor. Si  $L(r)$  no tiene rama derecha, entonces  $L(r)$  es el predecesor de  $r$ .



2. Si  $R(L(r)) \neq \text{NULL} \Rightarrow$  el predecesor es el descendiente más a la derecha de  $L(r)$ .



El predecesor de la raíz de un árbol binario es el nodo más a la derecha de su subárbol izquierdo.

Por simetría, el sucesor de la raíz de un árbol binario es el nodo más a la izquierda de su subárbol derecho.

Dado un árbol binario de raíz  $r$ , el recorrido infijo puede diseñarse con los siguientes lineamientos:

1. Antes de descender hacia la rama izquierda, determine el predecesor  $r_p$ .
2. Haga  $R(r_p) = r$ , es decir, ponga un hilo derecho a  $r_p$  para que apunte a la raíz  $r$  (que es el sucesor de  $r_p$ ).
3.  $r_p$  es el último nodo en infijo de la rama izquierda de  $r$ . Cuando lo visite, recupere  $r$  mediante el valor de  $R(r_p)$ . Antes de visitar  $r$ , asegúrese de restaurar  $R(r_p)$  al valor nulo.

Estamos en capacidad de abordar un algoritmo infijo que use hilos parciales y cuya forma general será la siguiente:

259a *(Funciones de BinNode\_Utils 243a) +≡* ◀255 278a▶  
 template <class Node> inline  
 void inOrderThreaded(Node \* root, void (\*visitFct)(Node\*))  
 {  
 if (root == Node::NullPtr)  
 return;  
 Node \*p = root, \*r = Node::NullPtr, \*q;  
*(recorrido infijo hilado 259b)*  
 }

$p$  será el nodo que se visita o un nodo de regreso para ir hacia la derecha.

$q$  es la dirección de un nodo sobre el cual existe un hilo temporal.  $r$  es un puntero al padre de  $p$ .

259b *(recorrido infijo hilado 259b) ≡* (259a)  
 while ( $p \neq \text{Node::NullPtr}$ )  
 {  
 q = LLINK(p);  
 if (q == Node::NullPtr)  
 { // No hay rama izq => visitar p  
 (\*visitFct)(p);  
 r = p;  
 p = RLINK(p);  
 continue;  
 }
 }

```

<sea q el predecesor infijo de p 260>
if (q != r) // tiene p un predecesor?
 { // si ==> dejar un hilo para luego subir a visitar p
 RLINK(q) = p; // Aquí se coloca el hilo
 p = LLINK(p); // Seguir bajando por la izquierda
 continue;
 }
(*visitFct)(p);

RLINK (q) = Node::NullPtr; // Borrar hilo
r = p;
p = RLINK(p); // avanzar a la rama derecha

```

El mecanismo del algoritmo no es trivial. Para apreciar completamente su funcionamiento, es necesaria una ejecución manual, la cual se delega al lector.

260     $\langle$ sea q el predecesor infijo de p 260 $\rangle \equiv$  (259b)  
           // avanzar hacia el nodo más a la derecha de la rama izquierda  
         while (q != r and RLINK(q) != Node::NullPtr)  
             q = RLINK(q);

El hilado parcial es muy importante porque puede usarse en algoritmos sobre árboles en los cuales sea delicado utilizar pila o recursión, pues nos ahorra espacio en pila, lo cual nos garantiza un recorrido seguro, sin preocupación por desborde de pila. Consiguentemente, el hilado debe ser la opción a escoger si la altura del árbol es desconocida.

Hay sin embargo tres problemas con el hilado. El primero es que la algorítmica es más complicada. El segundo lo constituye el eventual consumo de tiempo requerido para encontrar el nodo predecesor antes de bajar un nivel a la izquierda. Finalmente el último problema es que los algoritmos que utilicen hilado parcial no son reentrantes; es decir, no pueden ejecutarse concurrentemente sobre el mismo árbol.

#### 4.4.17 Correspondencia entre árboles binarios y m-rios

Existe un método general para representar una arborescencia como un árbol binario y viceversa. El procedimiento se explica en el siguiente algoritmo:

Algoritmo 4.4 (Conversión de un árbol m-rio a uno binario equivalente)

La entrada es un árbol m-rio  $T_m \in \mathcal{T}$ .

La salida es un árbol binario  $T \in \mathcal{B}$  equivalente a  $T_m$

1.  $T = \emptyset$ .
  2.  $\forall n_i \in T_m$  aplique las siguientes reglas:

- (a) El hijo más a la izquierda de  $n_i$  en  $T_m$  es el hijo izquierdo de  $n_i \in T$ .
  - (b) El hermano inmediatamente a la derecha de  $n_i$  en  $T_m$  es el hijo derecho de  $n_i$  en  $T$ .

Este algoritmo produce un árbol binario equivalente al m-rio dado. Puesto que la raíz de un árbol  $T_m$  no tiene hermanos, la raíz del árbol binario resultante nunca tiene hijo derecho.

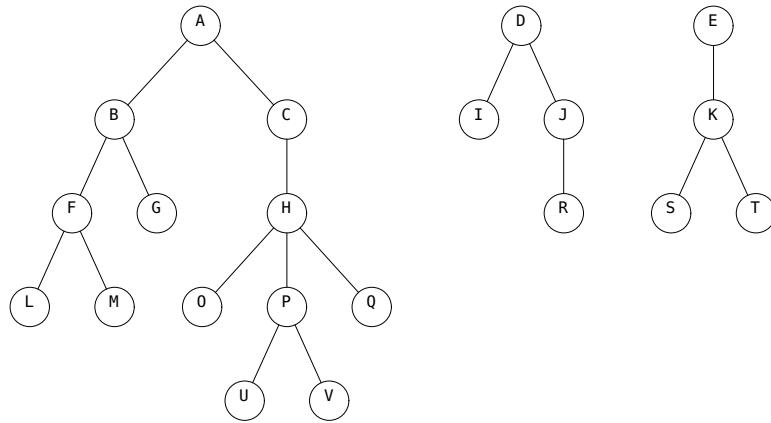


Figura 4.20: Una arborescencia

Esta observación nos conduce a extender el algoritmo 4.4 para que maneje arborescencias, lo cual consiste simplemente en considerar las raíces de los árboles de la arborescencia como hermanos de una primera raíz ficticia. En este caso se debe adoptar un criterio para determinar el orden en que los árboles de la arborescencia se procesan y que consiste en mirarlos de izquierda a derecha.

Hay una manera “visual” de interpretar el algoritmo 4.4, la cual es como sigue:

1. Encadene en una lista enlazada los hijos de cada familia.
2. Elimine los arcos excepto el que va hacia el hijo más a la izquierda.

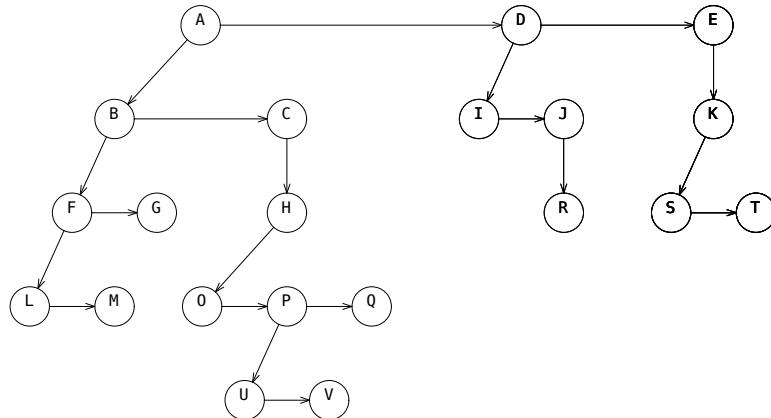


Figura 4.21: Representación con listas de la arborescencia de la figura 4.20

El árbol binario equivalente contiene la misma cantidad de arcos que el m-ario. Por cada lazo vertical eliminado existe un lazo horizontal añadido.

La figura 4.21 ilustra el árbol binario equivalente a la arborescencia de la figura 4.20 luego de ejecutar la interpretación visual del algoritmo 4.4.

El árbol binario equivalente de la figura 4.21 es exactamente la representación mediante listas enlazadas de la arborescencia esquematizada en la figura 4.20. Dado un nodo

cualquiera en la representación con listas enlazadas, el apuntador hacia la lista de hijos funge de apuntador al subárbol izquierdo. Análogamente, el apuntador hacia los hermanos funge de apuntador hacia el subárbol derecho.

Si se siente incrédulo, gire la figura 4.21 unos  $45^\circ$  en el sentido horario y estire un poquitín los lazos. Debe ver un árbol con la forma de la figura 4.22.

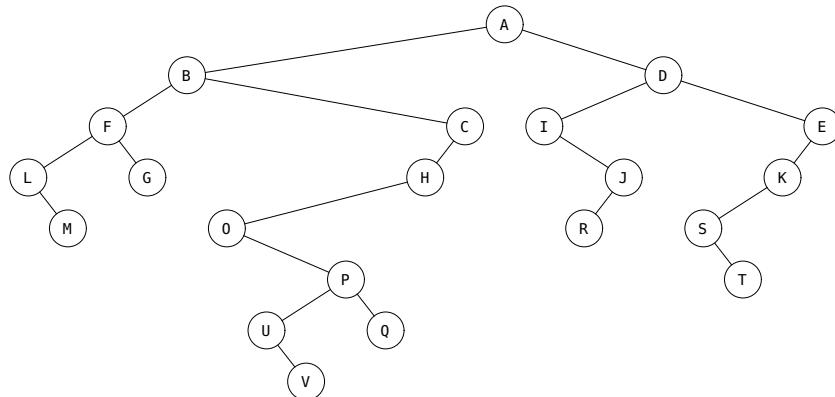


Figura 4.22: Árbol binario equivalente al árbol de la figura 4.20

La correspondencia es inyectiva; la conversión de un árbol  $m$ -rio arroja un único árbol binario. También es inversible y su algoritmo fácilmente deducible.

Puesto que  $T$  y  $T_b(T)$  son equivalentes para todo  $T$ , las manipulaciones que se hacen sobre un árbol  $m$ -rio pueden perfectamente realizarse sobre su equivalente binario.

Dada una arborescencia, existen dos maneras elementales de recorrerlas:

### Recorrido prefijo

1. Visite la raíz del primer árbol (el más a la izquierda).
2. Recorra en prefijo los subárboles del primer árbol.
3. Recorra en prefijo los árboles restantes de izquierda a derecha.

### Recorrido sufijo

1. Recorra en sufijo los subárboles del primer árbol (el más a la izquierda).
2. Visite la raíz del primer árbol.
3. Recorra en sufijo los árboles restantes de izquierda a derecha.

Ahora calculemos el recorrido prefijo de la arborescencia de la figura 4.20:

$$A \ B \ F \ L \ M \ G \ C \ H \ O \ P \ U \ V \ Q \ D \ I \ J \ R \ E \ K \ S \ T \quad (4.13)$$

Este recorrido corresponde exactamente al recorrido prefijo del árbol binario equivalente de la figura 4.22. Después de todo, el recorrido prefijo es la manera topológicamente natural de listar los nodos en un árbol.

El recorrido sufijo de la arborescencia de la figura 4.20 es:

$$L M F G B O U V P Q H C A I R J D S T K E , \quad (4.14)$$

el cual **no** corresponde al recorrido sufijo del árbol binario equivalente (figura 4.22). En su lugar, la secuencia (4.14) corresponde al recorrido infijo del árbol binario equivalente de la figura 4.22. Esta equivalencia de recorridos (sufijo en una arborescencia  $\iff$  infijo en su equivalente binario) es una ganancia algorítmica, pues el recorrido sufijo es más costoso de implantar que los recorridos prefijo e infijo.

En resumen, dada una arborescencia y el árbol binario equivalente calculado según el algoritmo 4.4, tenemos las siguientes correspondencias entre los recorridos:

1. El recorrido prefijo de la arborescencia corresponde al recorrido prefijo del árbol binario equivalente.
2. El recorrido sufijo de la arborescencia corresponde al recorrido infijo del árbol binario equivalente.

En una arborescencia, los recorridos infijo y por niveles no pueden definirse con precisión. Claramente, hay varias maneras de definir el recorrido infijo sobre una arborescencia, pues existen varias maneras de colocar la raíz entre sus subárboles. Del mismo modo, hay varias formas de definir el recorrido por niveles en una arborescencia.

La equivalencia entre arborescencias y árboles binarios es de suma importancia práctica. Cualquier problema representado con arborescencias puede representarse y resolverse en el plano de los árboles binarios; el enfoque inverso también es posible. Como diseñadores y programadores podemos escoger la representación más adecuada en función de bondades tales como la comprensión del problema, la eficiencia de ejecución, la simplicidad de la solución y la reutilización de algoritmos y de códigos existentes.

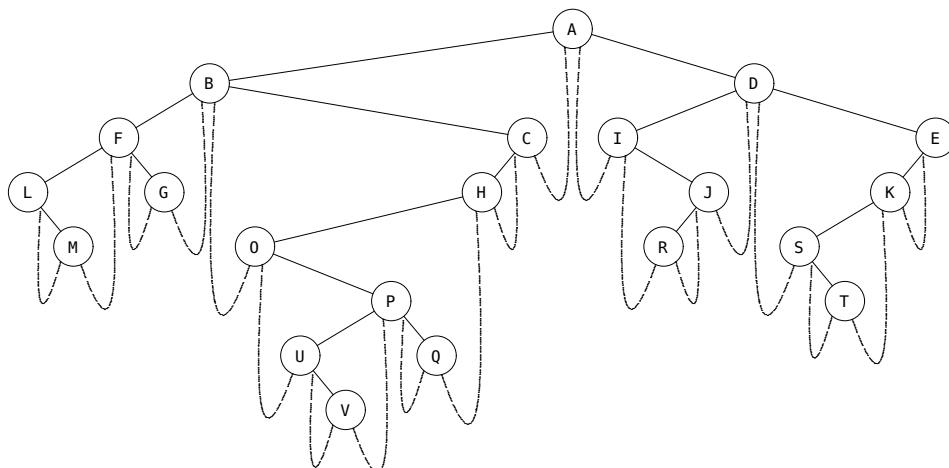


Figura 4.23: Árbol binario, hilado, equivalente al árbol de la figura 4.20

El hilado del árbol binario equivalente puede ser útil en ciertas clases de problemas. La figura 4.23 muestra el árbol binario hilado equivalente a la arborescencia de la figura 4.20. El dibujar esta representación con listas nos permitirá aprehender y corroborar el significado de los hilos y su utilización para regresar sobre los ancestros. Para eso recordemos

que el recorrido infijo en el árbol binario equivalente se corresponde con el recorrido sufijo en la arborescencia. Por lo tanto, a la excepción de los nodos más a la izquierda y más a la derecha del árbol binario equivalente, podemos concluir con las siguientes observaciones generales sobre los hilos y su sentido respecto al árbol m-rio:

1. Un hilo derecho apunta siempre a su padre, pues éste es el sucesor sufijo en la arborescencia.
2. Un hilo izquierdo denota que el nodo es hoja en el árbol m-rio; esto se deduce directamente de la equivalencia.

El nodo destino del hilo corresponde con el predecesor sufijo en el árbol m-rio. Aquí tenemos dos casos:

- (a) Si la hoja es el nodo más a la izquierda en el árbol m-rio, que puede identificarse en el equivalente binario porque es un hijo izquierdo, entonces el hilo apunta un tío primogénito en alguna generación. Notemos que puede tratarse de un tío directo o de un tío en generaciones anteriores, según la profundidad de la hoja.
- (b) En el caso contrario, si la hoja no es el más a la izquierda en árbol m-rio, entonces, el hilo apunta siempre a su hermano izquierdo.

## 4.5 Un TAD genérico para árboles

Los árboles binarios nos permiten resolver una amplia variedad de problemas. No obstante, en algunas ocasiones puede preferirse un árbol de algún orden dado. En este sentido, esta sección tratará sobre el diseño e implantación de un TAD que modelice un árbol.

El TAD en cuestión se encuentra en el archivo `<tpl_tree_node.H 264a>`<sup>4</sup>, cuya estructura general es como sigue:

264a    `<tpl_tree_node.H 264a>≡`  
`template <class T> class Tree_Node`  
`{`  
`⟨Atributos de Tree_Node 264b⟩`  
`⟨Métodos privados de Tree_Node 265c⟩`  
`⟨Métodos públicos de Tree_Node 264c⟩`  
`};`  
`⟨Métodos utilitarios de árboles 270a⟩`

El TAD `Tree_Node` modeliza un nodo de un árbol general, el cual guarda un atributo genérico de tipo `T` especificado de la siguiente forma:

264b    `⟨Atributos de Tree_Node 264b⟩≡` (264a) 265a▷  
`T data;`

El cual puede accederse mediante:

264c    `⟨Métodos públicos de Tree_Node 264c⟩≡` (264a) 265b▷  
`T & get_key() { return get_data(); }`  
`T & get_data() { return data; }`  
`typedef T key_type;`

<sup>4</sup>En la elaboración de este TAD se contó con las valiosas ayudas de los para la época estudiantes José Brito (maestría) y Juan Fuentes (pregrado).

Nuestra primera decisión de diseño consiste en escoger la representación en memoria. Puesto que pretendemos versatilidad y generalidad, transaremos por la representación con listas, pues ésta ofrece más dinamismo que su contraparte con arreglos. Consiguientemente, requerimos dos enlaces dobles:

265a *(Atributos de Tree\_Node 264b) +≡* (264a) ◁264b 265e▷  
 Dlink child;  
 Dlink sibling;

child enlaza a los hijos más a la izquierda y sibling enlaza a los hermanos; su acceso está dado por los siguientes métodos:

265b *(Métodos públicos de Tree\_Node 264c) +≡* (264a) ◁264c 266a▷  
 Dlink \* get\_child\_list() { return &child; }  
 Dlink \* get\_sibling\_list() { return &sibling; }

Muchas veces, luego de obtener un puntero Dlink a o desde child o sibling, necesitaremos convertirlo a un Tree\_Node. Para ello generamos funciones de conversión mediante los macros ofrecidos por la clase Dlink (§ 2.4.7 (Pág. 72)):

265c *(Métodos privados de Tree\_Node 265c) ≡* (264a) 265d▷  
 LINKNAME\_TO\_TYPE(Tree\_Node, child);  
 LINKNAME\_TO\_TYPE(Tree\_Node, sibling);

A su vez, este par de métodos nos permite definir el acceso a los nodos del entorno de un Tree\_Node :

265d *(Métodos privados de Tree\_Node 265c) +≡* (264a) ◁265c  
 Tree\_Node \* upper\_link()  
 {  
 return child\_to\_Tree\_Node(child.get\_prev());  
}  
 Tree\_Node \* lower\_link()  
{  
return child\_to\_Tree\_Node(child.get\_next());  
}  
 Tree\_Node \* left\_link()  
{  
return sibling\_to\_Tree\_Node(sibling.get\_prev());  
}  
 Tree\_Node \* right\_link()  
{  
return sibling\_to\_Tree\_Node(sibling.get\_next());  
}

Es decir, el padre y el hijo, a través de upper\_link() y lower\_link(), en caso de que se trate de un nodo que sea el más a la izquierda, y los hermanos izquierdo y derecho, left\_link() y right\_link(), si se trata de cualquier otro nodo diferente al primogénito.

Si bien las listas son circulares, ellas no tienen nodo cabecera. Por esta razón debemos estar muy pendientes de cuál es el extremo de cada lista. En este sentido “marcaremos” cada nodo con banderas que nos indicarán el tipo de nodo según que éste sea extremo de alguna lista. Para ello definimos los siguientes bits:

265e *(Atributos de Tree\_Node 264b) +≡* (264a) ◁265a  
 struct Flags  
{  
 unsigned int is\_root : 1;  
 unsigned int is\_leaf : 1;  
 unsigned int is\_leftmost : 1;

```

 unsigned int is_rightmost : 1;
 Flags() : is_root(1), is_leaf(1), is_leftmost(1), is_rightmost(1) {}
};

Flags flags;

```

La observación y modificación de estas banderas definen los siguientes métodos sobre un nodo:

266a *(Métodos públicos de Tree\_Node 264c)+≡* (264a) ◁ 265b 266b ▷

```

bool is_root() const { return flags.is_root; }

bool is_leaf() const { return flags.is_leaf; }

bool is_leftmost() const { return flags.is_leftmost; }

bool is_rightmost() const { return flags.is_rightmost; }

void set_is_root(bool value) { flags.is_root = value; }

void set_is_leaf(bool value) { flags.is_leaf = value; }

void set_is_leftmost(bool value) { flags.is_leftmost = value; }

void set_is_rightmost(bool value) { flags.is_rightmost = value; }

```

#### 4.5.1 Observadores de Tree\_Node

A parte de las banderas y el dato genérico, desde un Tree\_Node pueden observarse sus nodos adyacentes, es decir, su padre, su hijo más a la izquierda y sus hermanos. Comencemos por los hermanos, los cuales son los observadores más simples:

266b *(Métodos públicos de Tree\_Node 264c)+≡* (264a) ◁ 266a 266c ▷

```

Tree_Node * get_left_sibling()
{
 if (is_leftmost())
 return NULL;
 return left_link();
}

```

`get_right_sibling()` es simétricamente similar a `get_left_sibling()`.

Desde un nodo se puede acceder a sus hijos extremos: el más a la izquierda y el más a la derecha:

266c *(Métodos públicos de Tree\_Node 264c)+≡* (264a) ◁ 266b 267a ▷

```

Tree_Node * get_left_child()
{
 if (is_leaf())
 return NULL;
 return lower_link();
}
Tree_Node * get_right_child()

```

```
{
 if (is_leaf())
 return NULL;
 Tree_Node * left_child = lower_link();
 return left_child->left_link();
}
```

Ambos métodos pueden referir al mismo nodo en caso de que `this` tenga un solo hijo o esté vacío.

En caso de que el grado del nodo sea mayor que dos, el resto de los nodos puede accederse mediante `get_left_sibling()` y `get_right_sibling()`.

A efectos de la versatilidad, puede convenirnos el acceso según el ordinal:

267a *(Métodos públicos de Tree\_Node 264c) +≡* (264a) ◁266c 267b ▷

```
Tree_Node * get_child(const int & i)
{
 Tree_Node * c = get_left_child();
 for (int j = 1; c != NULL and j < i; ++j)
 c = c->get_right_sibling();
 return c;
}
```

El método retorna `NULL` si `index` es mayor o igual que el grado del nodo.

Finalmente, el último observador concierne al nodo padre:

267b *(Métodos públicos de Tree\_Node 264c) +≡* (264a) ◁267a 267c ▷

```
Tree_Node * get_parent()
{
 if (is_root())
 return NULL;
 Tree_Node * p = this;
 while (not ISLEFTMOST(p)) // baje hasta el nodo más a la izquierda
 p = p->left_link();
 return p->upper_link();
}
```

Los árboles pueden asociarse en una arborescencia

#### 4.5.2 Modificadores de Tree\_Node

Cuando se crea un nuevo nodo se asume raíz de un árbol. Hay dos tipos de inserción: como hermano y como hijo. Según el tipo de inserción, el nodo puede dejar de ser raíz.

Hay dos formas de insertar como hermano: a la izquierda y a la derecha. Si el nodo hermano es raíz, entonces el nodo insertado sigue siendo raíz. En este caso, tratamos con una arborescencia asequible mediante los hermanos. En caso contrario, el nodo insertado comparte el mismo parente.

La inserción como hermano derecho se especifica de la siguiente forma:

267c *(Métodos públicos de Tree\_Node 264c) +≡* (264a) ◁267b 268a ▷

```
void insert_right_sibling(Tree_Node * p)
{
 if (p == NULL)
 return;

 if (not is_root())
 p->right_link = new Tree_Node();
 else
 p = new Tree_Node();
 p->right_link = p;
}
```

```

p->set_is_root(false);

p->set_is_leftmost(false);
Tree_Node * old_next_node = get_right_sibling();

if (old_next_node != NULL)
 p->set_is_rightmost(false);

this->set_is_rightmost(false);
this->sibling.insert(SIBLING_LIST(p));
}

```

La inserción como hermano izquierdo es un poco más complicada porque p puede devenir el hijo más a la izquierda, en cuyo caso, el antiguo más izquierdo debe sacarse de la lista de hijos child y sustituirse por p.

268a ⟨Métodos públicos de Tree\_Node 264c⟩+≡ (264a) ◁ 267c 268b ▷

```

void insert_left_sibling(Tree_Node * p)
{
 if (p == NULL)
 return;

 if (not this->is_root())
 p->set_is_root(false);

 p->set_is_rightmost(false);
 Tree_Node * old_next_node = this->get_left_sibling();
 if (old_next_node != NULL)
 p->set_is_leftmost(false);
 else if (not this->child.is_empty()) // p será más a la izq
 { // this es más a la izq ==> p debe ser primogénito
 Tree_Node * parent = this->get_parent();
 Tree_Node * left_child = this->get_left_child();
 CHILD_LIST(this)->del(); // sacar this de lista de hijos
 if (parent != NULL) // ahora meter a p en la lista de hijos
 parent->insert(p);
 else
 left_child->append(p);
 }
 this->set_is_leftmost(false);
 this->sibling.append(SIBLING_LIST(p));
}

```

Ambas primitivas, insert\_right\_sibling() e insert\_left\_sibling(), insertan por la izquierda y derecha y requieren que el nodo a insertar esté vacío.

El segundo tipo de inserción, es decir, como hijo, puede efectuarse como el hijo más a la izquierda o el más a la derecha. He aquí la de la izquierda:

268b ⟨Métodos públicos de Tree\_Node 264c⟩+≡ (264a) ◁ 268a 269a ▷

```

void insert_leftmost_child(Tree_Node * p)
{
 if (p == NULL)
 return;

```

```

p->set_is_root(false);
if (this->is_leaf())
{
 this->set_is_leaf(false);
 CHILD_LIST(this)->insert(CHILD_LIST(p));
}
else
{
 Tree_Node * old_left_child_node = this->lower_link();
 old_left_child_node->set_is_leftmost(false);
 p->set_is_rightmost(false);
 CHILD_LIST(old_left_child_node)->del();
 CHILD_LIST(this)->insert(CHILD_LIST(p));
 SIBLING_LIST(old_left_child_node)->append(SIBLING_LIST(p));
}
}

```

También existe su equivalente por la derecha, llamado `insert_rightmost_child()`.

Aunque eventualmente es posible construir un arborescencia mediante algunas de estas clases de inserción, es preferible construir árboles y luego encadenarlos en un arborescencia mediante el siguiente método:

269a *(Métodos públicos de Tree\_Node 264c) +≡* (264a) ◁ 268b 269b ▷

```

void insert_tree_to_right(Tree_Node * tree)
{
 tree->set_is_leftmost(false);
 Tree_Node * old_next_tree = this->get_right_tree();
 if (old_next_tree != NULL)
 tree->set_is_rightmost(false);

 this->set_is_rightmost(false);
 SIBLING_LIST(this)->insert(SIBLING_LIST(tree));
}

```

El parámetro `tree` debe imperativamente ser un árbol, es decir, `tree` tiene que ser una raíz; de lo contrario se considera un error.

#### 4.5.3 Observadores de árboles

Arborescencias construidas mediante `insert_tree_to_right()` pueden consultarse a través de los siguientes métodos:

269b *(Métodos públicos de Tree\_Node 264c) +≡* (264a) ◁ 269a 269c ▷

```

Tree_Node * get_left_tree()
Tree_Node * get_right_tree()

```

Estos métodos retornan el árbol situado a la izquierda o derecha de `this` respectivamente.

Eventualmente, sólo desde el primer árbol de la arborescencia, o sea, desde el más a la izquierda puede consultarse el árbol más a la derecha de la arborescencia mediante la siguiente operación:

269c *(Métodos públicos de Tree\_Node 264c) +≡* (264a) ◁ 269b ▷

```

Tree_Node * get_last_tree()

```

#### 4.5.4 Recorridos sobre Tree\_Node

El primer ejercicio del TAD recién explicado es el desarrollo de los recorridos. Consideremos en primer lugar el recorrido prefijo recursivo de un árbol:

270a *(Métodos utilitarios de árboles 270a)≡* (264a) 270b▷

```
template <class Node> static inline
void __tree_preorder_traversal(Node * root, const int & level,
 const int & child_index,
 void (*visitFct)(Node *, int, int))
{
 (*visitFct)(root, level, child_index);
 Node * child = root->get_left_child();
 for (int i = 0; child != NULL;
 ++i, child = child->get_right_sibling())
 __tree_preorder_traversal(child, level + 1, i, visitFct);
}
```

La rutina recorre recursivamente, en prefijo, el árbol con raíz `root`. En cada visita se invoca a la función apuntada por `(*visitFct)`, cuyos parámetros son el nodo visitado, su nivel dentro del árbol y su ordinal como hijo respecto a su padre.

`__tree_preorder_traversal()` no está destinada como interfaz pública. En su lugar, diseñamos dos primitivas públicas de recorrido prefijo que llaman a la versión estática, una para un árbol y otra para una arborescencia:

270b *(Métodos utilitarios de árboles 270a)+≡* (264a) ▷270a 270c▷

```
template <class Node> inline
void tree_preorder_traversal(Node * root, void (*visitFct)(Node *, int, int))
{
 __tree_preorder_traversal(root, 0, 0, visitFct);
}

template <class Node> inline
void forest_preorder_traversal(Node * root, void (*visitFct)(Node *, int, int))
{
 for /* nada */; root != NULL; root = root->get_right_tree()
 __tree_preorder_traversal(root, 0, 0, visitFct);
}
```

El recorrido sufijo es muy similar y comprensible luego de estudiado el recorrido prefijo.

#### 4.5.5 Destrucción de Tree\_Node

Para hacer al TAD `Tree_Node` mínimamente operativo nos es esencial la destrucción, cuya especificación es como sigue:

270c *(Métodos utilitarios de árboles 270a)+≡* (264a) ▷270b 271a▷

```
template <class Node> inline void destroy_tree(Node * root)
{
 if (not IS_UNIQUE_SIBLING(root))
 SIBLING_LIST(root)->del(); // no ==> sacarlo de lista hermanos
 // recorrer los subárboles de derecha a izquierda
 for (Node * p = root->get_right_child(); p != NULL; /* nada */)
 {
 Node * to_delete = p; // respaldar subárbol a borrar p
```

```

 p = p->get_left_sibling(); // Avanzar p a hermano izquierdo
 destroy_tree(to_delete); // eliminar recursivamente árbol
}
if (root->is_leftmost()) // ¿sacar lista hijos?
 CHILD_LIST(root)->del();
delete root;
}
template <class Node> inline void destroy_forest(Node * root)
{
 while (root != NULL) // recorre los árboles de izquierda a derecha
 {
 Node * to_delete = root; // respalda raíz
 root = root->get_right_sibling(); // avanza a siguiente árbol
 SIBLING_LIST(to_delete)->del(); // elimine de lista árboles
 destroy_tree(to_delete); // Borre el árbol
 }
}

```

#### 4.5.6 Búsqueda por número de Deway

Planteémosnos la búsqueda de un nodo en un árbol dado su número de Deway (§ 4.2.4 (Pág. 231)). En ese sentido definimos un camino de Deway como una secuencia de enteros guardada en un arreglo de tipo int path[] y delimitada con un valor negativo que indica el fin de la secuencia. La primitiva fundamental de búsqueda de Deway es la siguiente:

271a *(Métodos utilitarios de árboles 270a)+≡* (264a) ◁270c 271b ▷

```

template <class Node> static inline
Node * __deway_search(Node * node, int path [],
 const int & idx, const size_t & size)
{
 if (node == NULL)
 return NULL;
 if (path[idx] < 0) // verifique si se ha alcanzado el nodo
 return node;
 // avance hasta el próximo hijo path[0]
 Node * child = node->get_left_child();
 for (int i = 0; i < path[idx] and child != NULL; ++i)
 child = child->get_right_sibling();
 return __deway_search(child, path, idx + 1, size); // próximo nivel
}

```

La rutina retorna NULL si el nodo con el número de Deway especificado en path[] no se encuentra en el árbol; la dirección de tal nodo, en caso contrario.

La primitiva anterior es privada porque una versión más completa que incluya búsqueda en una arborescencia, puede implantarse tal como sigue:

271b *(Métodos utilitarios de árboles 270a)+≡* (264a) ◁271a 272a ▷

```

template <class Node> inline
Node * deway_search(Node * root, int path [], const size_t & size)
{
 for (int i = 0; root != NULL; i++, root = root->get_right_sibling())
 if (path[0] == i)
 return __deway_search(root, path, 1, size);
}

```

```

 return NULL;
}

```

#### 4.5.7 Cálculo del número de Deway

En muchos contextos<sup>5</sup>, un problema importante consiste en buscar un nodo en el árbol y entonces determinar su número de Deway. Para ello planteamos un recorrido prefijo del árbol con un arreglo que contenga el número de Deway actual del nodo visitado según el siguiente prototipo:

272a *(Métodos utilitarios de árboles 270a) +≡* (264a) ◁271b 272b ▷

```

template <class Node, class Equal> inline static
Node * __search_deway(Node * root, const typename Node::key_type & key,
 const size_t & current_level, int deway [],
 const size_t & size, size_t & n);

```

`__search_deway()` busca recursivamente en el árbol cuya raíz es `root` la clave `key`. A cada llamada recursiva se incrementa el nivel `current_level`, el cual se usa para actualizar el número de deway almacenado en el arreglo `deway` y cuya capacidad es `size`. El parámetro `n` tiene sentido si se encuentra la clave y almacena la longitud del número de Deway.

La rutina anterior es iniciada por la siguiente interfaz pública:

272b *(Métodos utilitarios de árboles 270a) +≡* (264a) ◁272a 272c ▷

```

template <class Node,
 class Equal = Aleph::equal_to<typename Node::key_type>> inline
Node * search_deway(Node * root, const typename Node::key_type & key,
 int deway [], const size_t & size, size_t & n)
{
 n = 1; // valor inicial de longitud de número de Deway
 for (int i = 0; root != NULL; i++, root = root->get_right_sibling())
 {
 deway[0] = i;
 Node * result =
 __search_deway <Node, Equal> (root, key, 0, deway, size, n);
 if (result != NULL)
 return result;
 }
 return NULL;
}

```

Nos falta por definir la rutina que recorrerá en prefijo el árbol a la búsqueda de la clave:

272c *(Métodos utilitarios de árboles 270a) +≡* (264a) ◁272b 273 ▷

```

template <class Node, class Equal> inline static
Node * __search_deway(Node * root,
 const typename Node::key_type & key,
 const size_t & current_level, int deway [],
 const size_t & size, size_t & n)
{
 if (root == NULL)
 return NULL;

```

---

<sup>5</sup>En particular para programas que requieran como entrada el número de Deway de un nodo.

```

 if (Equal () (KEY(root), key))
 {
 n = current_level + 1; // longitud del arreglo deway
 return root;
 }
 Node * child = root->get_left_child();
 for (int i = 0; child != NULL;
 i++, child = child->get_right_sibling())
 {
 deway[current_level + 1] = i;
 Node * result = __search_deway <Node, Equal>
 (child, key, current_level + 1, deway, size, n);
 if (result!= NULL)
 return result;
 }
 return NULL;
}

```

#### 4.5.8 Correspondencia entre Tree\_Node y árboles binarios

En esta subsección implantaremos la correspondencia entre árboles m-rios y binarios explicada mediante el algoritmo 4.4 presentado en § 4.4.17 (Pág. 260)<sup>6</sup>.

El primer algoritmo que desarrollaremos será el de la conversión de un árbol m-rio, implantado mediante del TAD Tree\_Node hacia un árbol binario implantado a través del TAD BinNode<Key>. Cuando decimos “*implantado a través del TAD ...*” nos referimos a que el árbol m-rio y binario se fundamentan en las clases Tree\_Node y BinNode<Key> respectivamente, pero no por fuerza son con exactitud de estas dos clases, sino que podrían ser, por ejemplo, clases derivadas.

La conversión de un árbol m-rio hacia uno binario se expresa directamente según las pautas del algoritmo 4.4, es decir, en el árbol binario, toda rama izquierda corresponde a un hijo más a la izquierda del árbol m-rio, mientras que toda rama derecha corresponde al siguiente hermano. Según estos lineamientos, la implantación resultante es como sigue:

273

(Métodos utilitarios de árboles 270a) +≡ (264a) ▷ 272c 274a>

```

template <class TNode, class BNode>
BNode * forest_to_bin(TNode * root)
{
 if (root == NULL)
 return BNode::NullPtr;
 BNode * result = new BNode (root->get_key());
 LLINK(result) = forest_to_bin<TNode,BNode>(root->get_left_child());
 RLINK(result) = forest_to_bin<TNode,BNode>(root->get_right_sibling());
 return result;
}

```

La rutina maneja dos tipos genéricos. El primero, TNode, es un nodo m-rio basado en el TAD Tree\_Node . El segundo tipo es BNode, el cual representa el nodo binario y debe estar basado en BinNode<Key>. La verificación de estos tipos se lleva a cabo implícitamente en

<sup>6</sup>En el diseño e implantación de las funciones explicadas en esta subsección es menester señalar la participación fundamental de Juan Fuentes.

tiempo de instanciación de la plantilla cuando se compilan las tres penúltimas líneas y se verifica la existencia de los métodos que se invocan.

La conversión de un árbol binario hacia uno m-rio es un poco más laboriosa, razón por la cual nos conviene descomponerla en rutinas específicas y separadas que efectúen acciones concretas y modularizadas. En este sentido hay dos acciones para un Tree\_Node

1. *Inserción de primogénito*: como ya sabemos, en un árbol binario una rama izquierda representa un primogénito en el árbol m-rio equivalente. Cuando en un nodo binario miramos su hijo izquierdo lo insertamos en el árbol m-rio como primogénito. De este modo, planteamos la subrutina siguiente:

274a <Métodos utilitarios de árboles 270a>+≡ (264a) ◁273 274b▷  
template <class TNode, class BNode> inline static  
void insert\_child(BNode \* lnode, TNode \* tree\_node)  
{  
 if (lnode == BNode::NullPtr)  
 return;  
 TNode \* child = new TNode(KEY(lnode));  
 tree\_node->insert\_leftmost\_child(child);  
}

El parámetro `lnode` es un nodo binario que es hijo izquierdo. El parámetro `tree_node` es un nodo m-rio equivalente al nodo binario padre de `lnode`.

2. *Inserción de hermano*: análogamente, cuando en el nodo binario miramos su hijo derecho lo insertamos en el árbol m-rio como su hermano derecho. Esto conduce a la siguiente subrutina:

El parámetro `rnode` es un nodo binario que es hijo derecho. El parámetro `tree_node` es un nodo m-rio equivalente al nodo binario padre de `rnode`.

Ahora estamos listos para implantar el algoritmo que calcule el árbol m-rio equivalente a uno binario:

```

insert_child(LLINK(broot), troot);
TNode * left_child = troot->get_left_child();

bin_to_tree(LLINK(broot), left_child);

insert_sibling(RLINK(broot), troot);
TNode * right_sibling = troot->get_right_sibling();

bin_to_tree(RLINK(broot), right_sibling);
}

```

El parámetro broot es la raíz de un árbol binario fundamentado en el TAD BinNode<Key>. El parámetro troot es la raíz de un árbol m-rio, fundamentado en el TAD Tree\_Node , equivalente a broot.

Finalmente, la interfaz pública y definitiva que crea la raíz inicial y dispara la construcción recursiva, se define de la siguiente forma:

*<Métodos utilitarios de árboles 270a>+≡*

(264a) ↳ 274c

```

template <class TNode, class BNode> inline
TNode * bin_to_forest(BNode * broot)
{
 if (broot == BNode::NullPtr)
 return NULL;
 TNode * troot = new TNode (KEY(broot));
 bin_to_tree(broot, troot);
 return troot;
}

```

## 4.6 Algunos conceptos matemáticos de los árboles

Los árboles se encuentran entre las estructuras más interesantes y complejas de la matemática combinatoria. En esta sección presentaremos un corpus matemático mínimo que nos permita abordar el análisis de las diferentes estructuras árbol que luego estudiaremos.

Nos concierne conocer qué tanto un árbol se parece a un árbol, pues para algunas situaciones computacionales, cuanto más un árbol sea un árbol, entonces, más eficiente es su utilización. Notemos que una lista de elementos encaja en el concepto de árbol, empero, obviamente, si los árboles tuviesen topologías similares a las listas, entonces sería preferible utilizar listas.

### 4.6.1 Altura de un árbol

Un primer indicador acerca de qué tan bueno puede ser un árbol es conocer las alturas posibles en función del número de nodos, para ello es útil estudiar la siguiente proposición.

**Proposición 4.1** Sea T un árbol m-rio con n nodos. Entonces:

$$\lceil \log_m (n(m-1) + 1) \rceil \leq h(T) \leq n \quad (4.15)$$

### Demostración

- $h(T) \geq \log_m (n(m - 1) + 1)$

La altura de un árbol es mínima cuando el número de nodos completos es máximo. Para comprender esto consideremos construir progresivamente un árbol  $m$ -rio evitando en lo posible aumentar su altura. A partir del árbol vacío sólo podemos insertar el nodo raíz. Luego, tenemos  $m$  espacios disponibles en el nivel 1. Posteriormente, cada uno de los  $m$  nodos del nivel 1 también tiene  $m$  celdas disponibles que estarían en el nivel 2; es decir  $m \times m$  celdas que pueden colocarse en el nivel 2. Continuando esta línea de razonamiento es fácil observar que cada nivel  $i$  puede contener a lo sumo  $m^i$  nodos. Por tanto, el número de nodos de un árbol construido de esta forma puede expresarse como:

$$n \leq \sum_{i=0}^{h(T)-1} m^i = m^0 + m^1 + m^2 + \cdots + m^{h(T)-1} \quad (4.16)$$

Multiplicando (4.16) por  $m$  tenemos:

$$m n \leq \sum_{i=0}^{h(T)-1} m^{i+1} = m^1 + m^2 + \cdots + m^{h(T)-1} + m^{h(T)} \quad (4.17)$$

Restando (4.17) menos (4.16), tenemos:

$$\begin{aligned} m n - n &\leq m^{h(T)} - 1 \Rightarrow \\ n(m - 1) + 1 &\leq m^{h(T)} \Rightarrow \\ \lceil \log_m (n(m - 1) + 1) \rceil &\leq h(T) \quad \square \end{aligned}$$

- $h(T) \leq n$

Para este caso seguimos la línea de razonamiento inversa al punto anterior, es decir, construimos progresivamente un árbol  $m$ -rio tratando siempre de aumentar su altura. Tal árbol es aquél que tiene exactamente  $n$  niveles y un nodo por nivel. Pues este árbol tiene altura  $n$ . ■

Esta proposición revela cotas para el más corto y largo camino desde la raíz hasta una hoja en función de la cardinalidad del árbol. Suponiendo  $n$  nodos, el árbol más “corto” posible tiene altura  $h(T) = \lceil \log_m (n(m - 1) + 1) \rceil$ , mientras que el más alto  $h(T) = n$ .

Los árboles con altura mínima son de gran importancia computacional, pues garantizan la mayor eficiencia de muchos de los algoritmos sobre árboles. A la fecha actual hay pocos esquemas en los cuales se pueda restringir la altura dinámica y eficazmente para que ésta sea mínima. Así pues, muchas veces es necesario mantener árboles cuya altura no sea mínima. La calidad de esta última clase de árboles se mide en qué tanto estos árboles se acercan a los de altura mínima y no realmente por el valor de su altura. Requerimos entonces una medida adicional que nos indique qué tan bueno es el árbol. El marco conceptual de esta medida se denomina “longitud del camino interno/externo”, y se enunciará en la subsección siguiente.

### 4.6.2 Longitud del camino interno/externo

**Definición 4.3 (Nodo externo)** Sea  $T$  un árbol  $m$ -rio con  $n$  nodos y sea  $n_i$  algún nodo incompleto de  $T$  tal que  $\text{grado}(n_i) = m' < m$ . Entonces, los nodos externos de  $n_i$  se definen como los  $m - m'$  subárboles faltantes.

En otras palabras, un nodo externo es todo puntero nulo. Consecuentemente, un nodo interno es todo nodo perteneciente al árbol.

A menudo es conveniente y más fácil analizar un árbol con una versión especial en que se muestren todos sus nodos externos. Tradicionalmente, los nodos internos se representan con círculos y los externos con líneas horizontales. La figura 4.24 ilustra la versión extendida de un árbol binario.

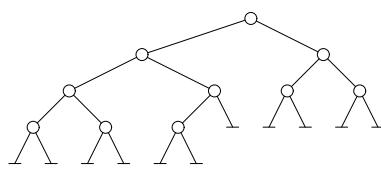


Figura 4.24: Árbol binario extendido

Por tradición, en expresiones algebraicas, un nodo interno se denota como  $n_i$  y uno externo como  $n_x$ .

**Proposición 4.2** Sea  $T$  un árbol  $m$ -rio con  $n$  nodos internos. Entonces, el número de nodos externos es:

$$(m - 1)n + 1 \quad (4.18)$$

**Demostración** Es claro que existen  $m \times n$  apuntadores. De los  $n$  nodos,  $n - 1$  tienen padre (la raíz no). Existen entonces  $n - 1$  ramas o apuntadores diferentes de nulo, por lo que el número de nodos externos estará dado por el total de apuntadores menos la cantidad de apuntadores no nulos. Esto es:

$$mn - (n - 1) = (m - 1)n + 1 \quad \blacksquare$$

Por la proposición 4.2, para un árbol binario de  $n$  nodos existen  $2n$  apuntadores,  $n - 1$  apuntadores a nodos internos y  $n + 1$  nodos externos.

**Definición 4.4 (Cardinalidad del nivel)** Sea  $T$  un árbol  $m$ -rio, entonces, la cardinalidad del nivel  $i$ , denotada  $|\text{nivel}(i)|$ , se define como el número de nodos internos en el nivel  $i$ .

**Definición 4.5 (Longitud del camino interno/externo)** Sea  $T$  un árbol  $m$ -rio con  $n$  nodos. La longitud del camino interno del árbol  $T$ , denotada como  $\text{IPL}(T)$ , se define como:

$$\text{IPL}(T) = \sum_{\substack{\forall n_i \in T \\ n_i \text{nodo interno}}} \text{nivel}(n_i) = \sum_{\forall i} (i \times |\text{nivel}(i)|) \quad (4.19)$$

Análogamente, la longitud del camino externo, denotada como  $\text{EPL}(T)$ , se define como:

$$\text{EPL}(T) = \sum_{\substack{\forall n_x \in T \\ n_x \text{nodo externo}}} \text{nivel}(n_x) \quad (4.20)$$

Para el árbol de la figura 4.24, la longitud del camino interno es

$$\text{IPL}(T) = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + 3 = 19$$

y la longitud del camino externo es

$$\text{EPL}(T) = 3 + 3 + 3 + 3 + 3 + 4 + 4 + 4 + 4 + 4 = 39$$

El cálculo de la longitud del camino es fundamental para el análisis empírico de desempeño de algoritmos basados en árboles binarios de búsqueda; una clase de árbol binario especial que estudiaremos más adelante. Esto justifica diseñar una primitiva que calcule el IPL:

278a *(Funciones de BinNode\_Utils 243a)*+≡ ◀259a 278b▶

```
template <class Node> inline static
size_t __internal_path_length(Node * p, const size_t & level)
{
 if (p == Node::NullPtr)
 return 0;
 return level + __internal_path_length(LLINK(p), level + 1) +
 __internal_path_length(RLINK(p), level + 1);
}
```

El parámetro `level` indica el nivel actual del nodo visitado y también debe inicializarse en cero. La interfaz pública `internal_path_length_rec()` funge pues de “wrapper” que ejecute la llamada inicial:

278b *(Funciones de BinNode\_Utils 243a)*+≡ ◀278a 311▶

```
template <class Node> inline size_t internal_path_length(Node * p)
{
 return __internal_path_length(p, 0);
}
```

**Proposición 4.3** Sea  $T$  un árbol  $m$ -rio, entonces:

$$|T| = \sum_{i=1}^m |T^i| + 1 \quad (4.21)$$

$$\text{IPL}(T) = \sum_{i=1}^m \text{IPL}(T^i) + |T| - 1 \quad (4.22)$$

$$\text{EPL}(T) = \sum_{i=1}^m \text{EPL}(T^i) + (m-1)|T| + 1 \quad (4.23)$$

**Demostración** Para (4.21) es evidente que  $|T|$  es la suma de las cardinalidades de sus subárboles más su raíz.

En cuanto a (4.22), notemos que el nodo raíz de  $T$  no tiene incidencia en la longitud del camino, pues su nivel es 0. Cuando calculamos las longitudes de los caminos de los subárboles de  $T$ , asumimos que los nodos están a un nivel inferior, es decir, cada nodo es sumado en un nivel menos. Por ejemplo, la raíz de  $T^i$ , que en  $T$  está en el nivel 1 se pondera con 0 en lugar de 1; lo mismo sucede con el resto de los nodos. Así pues, para obtener el valor de  $\text{IPL}(T)$  debemos sumar 1 a cada nodo considerado, es decir, la cantidad de nodos  $|T| - 1$ , pues la raíz no cuenta.

Finalmente, para (4.23), el razonamiento es exactamente el mismo que para (4.22), sólo que el número de nodos externos es, según (4.18) (proposición 4.2),  $(m-1)|T| + 1$  ■

Estas identidades son fundamentos de muchas de las demostraciones involucradas en los árboles.

**Proposición 4.4** Sea  $T$  un árbol  $m$ -rio. Entonces:

$$EPL(T) = (m - 1) IPL(T) + m |T| \quad (4.24)$$

**Demostración (por inducción sobre la cardinalidad de  $T$ )** Asumiremos que  $k$  representa la cardinalidad de un árbol  $T_k$ .

- $|T_k| = 0$  En este caso  $EPL(T_0) = (m - 1) IPL(T_0) + m |T_0| = 0$ . La proposición es cierta para el caso base.

- $|T_{k+1}| = k + 1$  En este caso asumimos que la proposición es cierta para todo  $k$  y procedemos a verificar si es cierta para  $k + 1$ .

Comenzamos por plantear la resta de la ecuación (4.23) menos (4.22) en función de los resultados de la proposición 4.3 (ecuaciones (4.23) y (4.22)):

$$\begin{aligned} EPL(T_{k+1}) - IPL(T_{k+1}) &= \sum_{i=1}^m EPL(T_{k+1}^i) + (m - 1)|T_{k+1}| + 1 - \\ &\quad \left\{ \sum_{i=1}^m IPL(T_{k+1}^i) + |T_{k+1}| - 1 \right\} \\ &= \sum_{i=1}^m EPL(T_{k+1}^i) + mk + m - 2k - \\ &\quad \sum_{i=1}^m IPL(T_{k+1}^i) \end{aligned}$$

Ahora aplicamos la hipótesis inductiva sobre el término  $\sum_{i=1}^m EPL(T_{k+1}^i)$ :

$$\begin{aligned} EPL(T_{k+1}) - IPL(T_{k+1}) &= \sum_{i=1}^m \left\{ (m - 1) IPL(T_{k+1}^i) + m |T_{k+1}^i| \right\} + \\ &\quad mk + m - 2k - \sum_{i=1}^m IPL(T_{k+1}^i) \\ &= (m - 2) \sum_{i=1}^m IPL(T_{k+1}^i) + 2mk + m - 2k \end{aligned}$$

Notemos que, por (4.21),  $\sum_{i=1}^m |T_{k+1}^i| = (k + 1) - 1 = k$ . La sumatoria  $\sum_{i=1}^m IPL(T_{k+1}^i)$  puede reemplazarse por  $IPL(T_{k+1}) - k$ , que es el resultado de despejar la sumatoria de (4.22) evaluada en  $T_{k+1}$ :

$$\begin{aligned} EPL(T_{k+1}) - IPL(T_{k+1}) &= (m - 2)(IPL(T_{k+1}) - k) + 2mk + m - 2k \\ &= (m - 2) IPL(T_{k+1}) - \\ &\quad mk + 2k + 2mk + m - 2k \Rightarrow \\ EPL(T_{k+1}) &= (m - 1) IPL(T_{k+1}) + m(k + 1) \end{aligned} \quad (4.25)$$

que es la misma expresión de la proposición evaluada para  $k + 1$  nodos ■

Esta proposición nos ayudará a estudiar los límites relacionados con la longitud del camino, los cuales se establecen en la siguiente proposición.

**Proposición 4.5** Sea  $T$  un árbol  $m$ -rio de  $n$  nodos. Entonces:

$$\frac{\lceil \log_m (n(m-1)) \rceil}{m-1} m^{\lceil \log_m (n(m-1)+1) \rceil} + mn \leq \text{IPL}(T) \leq \frac{n(n-1)}{2} \quad (4.26)$$

**Demostración** Para demostrar  $\text{IPL}(T) \leq \frac{n(n+1)}{2}$  planteamos la longitud del camino del árbol de mayor altura posible. Tal árbol es aquél cuyos nodos, con excepción de la raíz y la hoja, sólo tienen una rama. La longitud del camino de este árbol está dada por:

$$\text{IPL}(T) = \sum_{i=0}^n i = \frac{n(n-1)}{2} \quad \square$$

Para demostrar la cota inferior planteamos la longitud del camino externo del árbol de altura mínima  $h$  en función de la proposición 4.4 (pág. 279):

$$\begin{aligned} \text{EPL}(T) &\leq hm^h = (m-1)\text{IPL}(T) + mn \implies \\ \text{IPL}(T) &\geq \frac{hm^h + mn}{m-1} \end{aligned} \quad (4.27)$$

Por la proposición 4.1 (pág. 275), conocemos el valor mínimo de  $h$ . Sustituimos, pues, la parte izquierda de (4.15) en (4.27):

$$\text{IPL}(T) \geq \frac{\lceil \log_m (n(m-1)) \rceil}{m-1} m^{\lceil \log_m (n(m-1)+1) \rceil} + mn \quad \blacksquare$$

Varias clases de árboles no acotan la altura, pero son, en forma, buenos. El IPL es una métrica que permite comparar árboles entre sí y determinar qué tan bueno es uno respecto al otro.

Decimos que un nodo está “ lleno ” cuando éste contiene todos sus hijos. Por el contrario, decimos que un nodo está “ incompleto ” cuando le falta al menos un hijo.

#### 4.6.3 Árboles completos

Consideremos la manera de construir un árbol  $m$ -rio de  $n$  nodos cuya longitud del camino sea mínima. Para construir tal árbol, debemos poner la mayor cantidad de nodos en los niveles inferiores; no debemos ponerlos en un nivel superior hasta que no se hayan completado todos los niveles predecesores.

Siguiendo esta línea de razonamiento podemos ver que sólo un nodo, la raíz, puede estar a distancia 0 de la raíz. A lo sumo,  $m$  nodos pueden estar a distancia 1 de la raíz. A lo sumo  $m \times m = m^2$  nodos pueden estar a distancia 2. En general, a lo sumo,  $m \times m \times m \dots \times m = m^h$  nodos pueden estar en el  $h$ -ésimo nivel. Así pues, la longitud del camino interno de este árbol será la suma de los primeros  $n$  términos de las series:

$$0, \underbrace{1, 1, \dots, 1}_{m \text{ veces}}, \underbrace{2, 2, \dots, 2}_{m^2 \text{ veces}}, \underbrace{3, 3, \dots, 3}_{m^3 \text{ veces}}, \underbrace{4, 4, \dots, 4}_{m^4 \text{ veces}}, \dots, \underbrace{m^{h-1}, m^{h-1}, \dots, m^{h-1}}_{\text{cantidad de nodos en el último nivel}}.$$

Sea  $T$  un árbol de  $n$  nodos, de altura  $h$ , cuyos nodos están a las mínimas distancias de la raíz. Podemos entonces denotar una expresión de base para la longitud del camino:

$$\text{IPL}(T) = \sum_{i=0}^{h-2} i m^i + (h-1)n' \quad (n' \text{ es la cantidad de nodos en el último nivel}) \quad (4.28)$$

Para contar el número de nodos en el último nivel razonamos como sigue.  $\sum_{i=0}^{h-1} m^i$  sería el total de nodos si el último nivel estuviese completamente lleno. Ahora bien, si el último nivel no estuviese completo, entonces faltarían  $\sum_{i=0}^{h-1} m^i - n$  nodos para completar el nivel. Además, si el último nivel estuviese completo, entonces habría  $m^{h-1}$  nodos en el último nivel. Sea  $n'$  el número de nodos en el último nivel, entonces:

$$n' = m^{h-1} - \left( \sum_{i=0}^{h-1} m^i - n \right) = m^{h-1} - \sum_{i=0}^{h-1} m^i + n \quad (4.29)$$

es decir, el máximo número de nodos en el último nivel menos los nodos que faltan para completar el nivel.

Al resolver las sumatorias involucradas, tenemos:

$$IPL(T) = \frac{(h-1)m^{h-1} - (h-2)m^h - m}{(m-1)^2} + (h-1) \left( m^{h-1} - \frac{m^h - 1}{m-1} + n \right) \quad (4.30)$$

La solución final de (4.30) se delega como ejercicio.

Otra forma de resolver (4.28) y (4.30) es planteando una sumatoria distinta que involucre logaritmos:

$$IPL(T) = \sum_{i=1}^n \lfloor \log_m i \rfloor ; \quad (4.31)$$

cuya solución también se delega en ejercicio y que puede resolverse por descomposición de la sumatoria en dos partes.

Los árboles completos, que son de longitud de camino mínima, son muy importantes, pues representan la mayor eficiencia posible sobre muchos algoritmos.

Una propiedad muy interesante de los árboles completos está dada por el hecho de que ellos pueden representarse secuencialmente en memoria mediante un arreglo. La figura 4.25 ilustra un árbol trinario completo donde cada nodo está etiquetado con su respectivo índice dentro de un arreglo secuencial. La disposición de los índices corresponde a un recorrido por niveles.

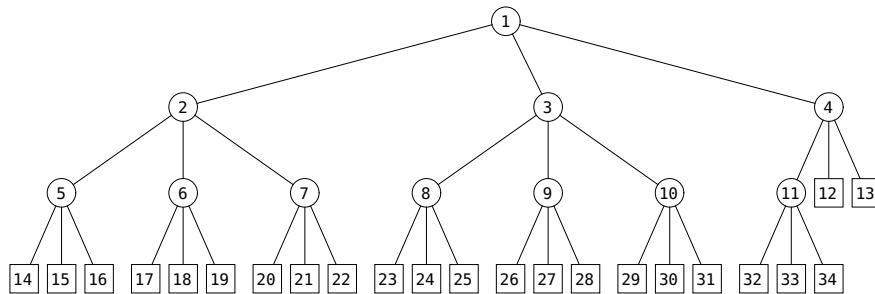


Figura 4.25: Árbol trinario completo; nodos enumerados según posición en arreglo secuencial

Para distinguir esta representación de la explicada en § 4.3.2 (Pág. 232) la denominaremos como “representación mediante arreglo secuencial”.

Sea  $i$  el índice de un nodo  $n_i$  dentro de un árbol de  $n$  nodos representado con un arreglo. Sea  $T(i, j)$  el  $j$ -ésimo subárbol de la raíz de un subárbol con índice  $i$  (los índices

comienzan en 1). Entonces:

$$\text{padre}(n_i) = \left\lfloor \frac{m+i-2}{m} \right\rfloor = \left\lceil \frac{i-1}{m} \right\rceil \quad (4.32)$$

$$T(i, j) = m(i-1) + 1 + j \quad (4.33)$$

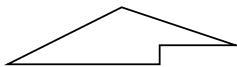
Estas fórmulas funcionan para cualquier orden de árbol. Por sus ventajas, la inmensa mayoría de autores y codificadores prefiere esta representación para algoritmos que manejen árboles completos, notablemente los heaps binarios, una estructura muy versátil que permite, entre otras cosas, ordenar eficientemente e implantar colas de prioridad.

## 4.7 Heaps

Comencemos esta sección desde lo abstracto introduciendo la definición de la estructura de datos de esta sección:

**Definición 4.6 (Heap)** Un “heap”<sup>7</sup>  $T$  es un árbol con las dos siguientes propiedades:

1. **Forma:** el árbol es completo, lo cual implica que, pictóricamente, exhibe la siguiente forma:



2. **Orden:**  $\forall n_i \in T \implies \text{KEY}(\text{hijo}_1(n_i)) < \text{KEY}(n_i) \wedge \text{KEY}(\text{hijo}_2(n_i)) < \text{KEY}(n_i) \wedge \dots \wedge \text{KEY}(\text{hijo}_m(n_i)) < \text{KEY}(n_i)$

Según el grado del árbol, un heap puede ser binario, trinario o de cualquier otro orden. Habida cuenta de la equivalencia entre los árboles binarios y las arborescencias (ver § 4.4.17 (Pág. 260)), en este texto sólo consideraremos los heaps binarios. Por tanto, si se trata de un árbol binario, entonces la propiedad de orden puede expresarse como:  $\forall n_i \in T \implies \text{KEY}(L(n_1)) < \text{KEY}(n_i) \wedge \text{KEY}(R(n_1)) < \text{KEY}(n_i)$ .

La bondad de un heap se resume en dos características:

1. El menor elemento de un conjunto de claves se encuentra directamente, por consecuencia de la propiedad de forma, en la raíz del árbol; dicho de otro modo, la consulta del menor elemento es  $\mathcal{O}(1)$ .
2. Cualquier modificación del heap, entendamos una inserción o eliminación, cuesta  $\mathcal{O}(\lg(n))$ . Intuitivamente, esto es una consecuencia de la propiedad de forma, la cual garantiza que la altura esté acotada a  $\mathcal{O}(\lg(n))$ .

<sup>7</sup>En pro de la compresión, preservaremos el término en su voz anglosajona, sin traducción. Cabe mencionar además que el término “heap” plantea algunas ambigüedades. Al respecto, es menester hacer dos aclaratorias:

1. En inglés, el término “heap” tiene varias acepciones, pero en nuestro caso connota un “montón”, “bulto” o “pila” de cosas. Por ejemplo, “*a heap of dirty clothes*” refiere a una pila de ropa sucia.
2. En las ciencias computacionales, “heap” se ha utilizado para connotar dos cosas. La primera concierne a una especie de árbol binario, la cual es el sujeto de estudio de la presente sección. La segunda refiere a la administración de memoria en un proceso.

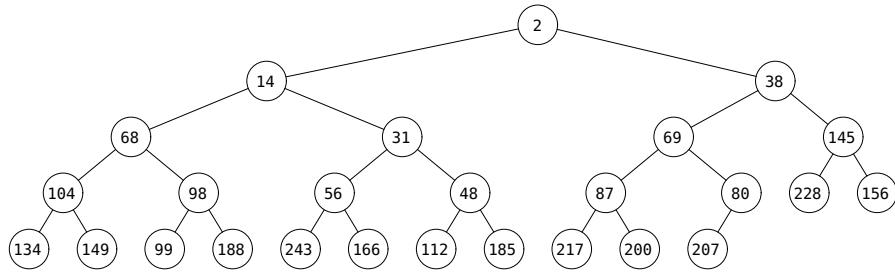


Figura 4.26: Un heap binario

La estructura de datos preferida para implantar un heap es un arreglo que almacene el recorrido por niveles del árbol completo. Por ejemplo, el heap de la figura 4.26 puede representarse con la siguiente secuencia: 2 14 38 68 31 69 145 104 98 56 48 87 80 228 156 134 149 99 188 243 166 112 185 217 200 207 la cual se almacena en un arreglo array[] especificado del siguiente modo:

283a *(miembros privados de ArrayHeap<Key> 283a)≡* (284b) 283c▷  
T \* array;

La raíz, o sea, el menor elemento del conjunto, se encuentra en:

283b *(Raíz del heap 283b)≡*  
array[1];

Notemos que es array[1] y no array[0], por eso el arreglo se aparta para dim + 1.

Para manejar esta estructura de datos, debemos especificar la dimensión del arreglo y contar la cantidad de elementos del heap mediante:

283c *(miembros privados de ArrayHeap<Key> 283a)+≡* (284b) ◁283a 284a▷  
const size\_t dim;  
size\_t num\_items;

Dado un índice i, los accesos al padre y los hijos se especifican de las siguientes formas:

1. **Padre:** en este caso nos basamos en (4.32):

283d *(Rutinas heap 283d)≡* (283e)▷  
static size\_t u\_index(const size\_t & i)  
{  
 return i >> 1; // divide i entre 2  
}

2. **Hijo izquierdo:** según (4.33):

283e *(Rutinas heap 283d)+≡* (283d 284c)▷  
static size\_t l\_index(const size\_t & i)  
{  
 return i << 1; // multiplica i por 2  
}

3. Hijo derecho: también según (4.33):

```
284a <miembros privados de ArrayHeap<Key> 283a>+≡ (284b) ↳283c 288a▷
 static size_t r_index(const size_t & i)
 {
 return (i << 1) + 1; // multiplica i por 2 y suma 1
 }
```

Estos cálculos revelan una ventaja del heap binario respecto a cualquiera de otro orden: los productos y divisiones pueden hacerse rápidamente mediante desplazamientos de bits, los cuales son substancialmente más rápidos que los productos y divisiones que ofrece el procesador o una biblioteca.

Los heaps binarios implantados mediante arreglos se especifican en el archivo (*tpl arrayHeap.H* 284b), cuya especificación general es como sigue:

```

284b <tpl_arrayHeap.H 284b>≡
 template <typename T, class Compare = Aleph::less<T> > class ArrayHeap
 {
 <miembros privados de ArrayHeap<Key> 283a>
 <miembros públicos de ArrayHeap<Key> 288b>
 };

```

Esta clase parametrizada implanta un heap binario de elementos de tipo genérico T, mediante un arreglo estático y con criterio de comparación establecido en la clase Compare.

#### 4.7.1 Inserción en un heap

Consideremos un heap binario de tamaño  $n$  y la inserción de un nuevo elemento  $x$ . Hay un solo lugar, dentro del árbol, en el cual puede añadirse un nuevo elemento sin alterar la propiedad de forma; tal lugar se pictoriza en la figura 4.27. Garantizada la propiedad de

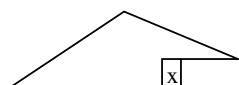


Figura 4.27: Lugar de inserción en un heap

forma, falta por verificar confirmar y, eventualmente, restaurar, la de orden. Para eso nos valdremos de la rutina:

```

284c <Rutinas heap 283d>+≡ ◁283e 286▷
 template <typename T, class Compare> inline
 void sift_up(T * ptr, const size_t & l, const size_t & r)
 {
 for (size_t p, i = r; i > l; i = p)
 {
 p = u_index(i); // indice del padre (c = i/2)
 if (Compare () (ptr[p], ptr[i])) // ¿cumple propiedad orden?
 return; // si, todo el arreglo es un heap

 std::swap(ptr[p], ptr[i]); // intercambie y restaure nivel p
 }
 }

```

La entrada de `sift_up()` es un arreglo contenido de un heap binario entre  $l$  y  $r - 1$ , con una eventual violación de la propiedad de orden en `array[n]`; la salida es un heap binario, en forma y orden, entre  $l$  y  $r$ , con la eventual violación corregida. El proceso se ejemplifica en la figura 4.28. Básicamente, `sift_up()` se verifica si existe una violación de orden entre `ptr[i]` y `ptr[p]`. Si tal es el caso, entonces intercambia sus elementos y asciende a procesar el nivel inferior; si no, entonces todo el árbol es un heap. El proceso continua hasta que el padre sea menor o se alcance la raíz.

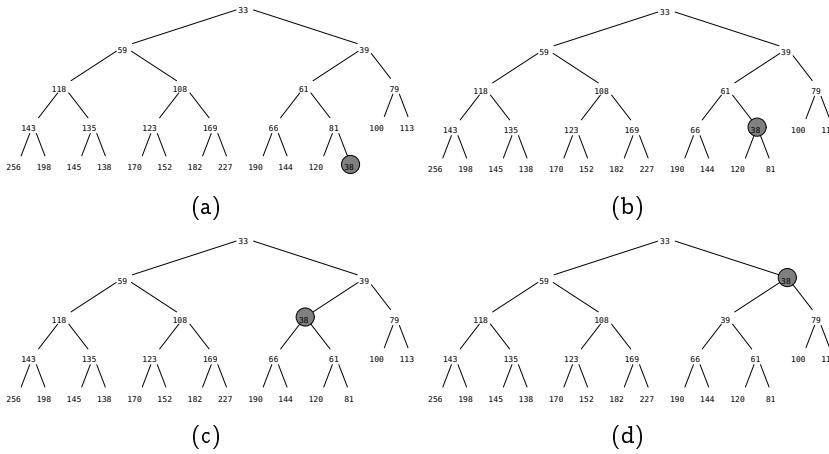


Figura 4.28: Operación `sift_up()` sobre el valor de clave 38

Debemos prestar especial cuidado al hecho de que intervalo del heap es  $[1 \dots n]$  y no  $[0 \dots n - 1]$ , como tradicionalmente se manejan los arreglos en lenguaje C y C<sup>++</sup>. Consecuentemente, la dirección `ptr` debe estar desplazada en una unidad.

El coste de `sift_up()` se expresa por el siguiente lema:

**Lema 4.1** Sea  $T$  un heap binario de  $n$  elementos con una violación eventual de orden en `array[n]`. Entonces, la duración de `sift_up(array, 1, n)` es  $\mathcal{O}(\lg(n))$ .

**Demostración** El tiempo de ejecución de `sift_up()` estará dominado por la cantidad de intercambios que ocasione el elemento violatorio. En este sentido, lo peor que puede ocurrir es que el elemento violatorio devenga la raíz de  $T$ . Por tanto, la cantidad máxima de intercambios es proporcional a la altura del árbol. Puesto que  $T$  es completo, y según se desprende de la proposición 4.1, su altura está acotada por  $\mathcal{O}(\lg(n))$   $\square$

Mediante `sift_up()` es simple añadir un nuevo elemento `key` al heap:

$\langle Añadir\ elemento\ al\ heap\ 285 \rangle \equiv$  (288d)

```
array[+num_items] = key; // colocar nuevo elemento
sift_up<T, Compare>(array, 1, num_items); // restaurar propiedad orden
lo cual, por el lema 4.1, es $\mathcal{O}(\lg(n))$.
```

### 4.7.2 Eliminación en un heap

La eliminación puede plantearse de dos maneras: (1) eliminación de la raíz y (2) eliminación de cualquier elemento. En realidad, el caso (2) es general y comprende al (1), pero, en favor de la simplicidad, trataremos el primero por separado.

Consideremos pues la eliminación del menor elemento de un heap, es decir, su raíz. Desde la perspectiva de forma, eliminar la raíz equivale pictóricamente a la figura 4.29-a, lo cual morfológicamente no es un heap. Hay, de nuevo, una manera de restaurar la forma, la cual equivale pictóricamente a la figura 4.29-b. Es decir, el último elemento sobreescribe

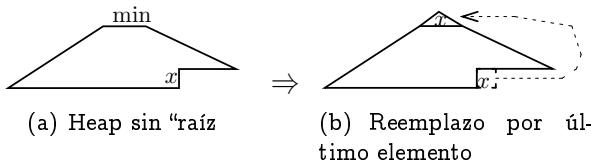


Figura 4.29: Esquema pictórico de la eliminación

a la raíz. Alegóricamente, el heap se corta por el último nivel del árbol y, con ese “pedazo”, “se rellena el hueco” dejado por la raíz.

El nuevo elemento raíz, muy probablemente violará la propiedad de orden, la cual puede restaurarse mediante el procedimiento `sift_down()`, encargado de intercambiar elementos del heap hasta que la propiedad de orden sea restaurada.

`sift_down()` es ligeramente más complejo que `sift_up()`, pues hay que mirar los dos hijos del eventual elemento violador y escoger para intercambiar el menor de ellos. De este modo se garantiza la propiedad de orden. El proceso continúa en el siguiente nivel hasta que los dos hijos sean mayores o se alcance el último nivel.

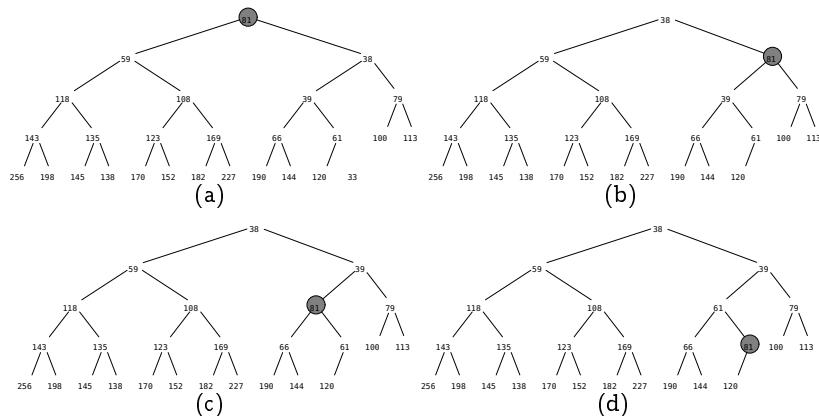


Figura 4.30: Operación `sift_down()` sobre la raíz 33 en el heap de la figura 4.28

286

*⟨Rutinas heap 283d⟩*  $\equiv$

$\triangleleft$  284c 287b  $\triangleright$

```
template <typename T, class Compare> inline
void sift_down(T * ptr, const size_t & l, const size_t & r)
{
 size_t i = l, c;
 while (true)
 {
 c = l_index(i); // índice del hijo izquierdo (c = i/2)
 if (c > r) // ¿hay hijo izquierdo?
 return; // no ==> termine

 if (c + 1 <= r) // ¿hay hijo derecho?
 if (Compare(ptr[c], ptr[c + 1]))
 swap(ptr[c], ptr[c + 1]);
 else
 break;
 else
 if (Compare(ptr[c], ptr[c + 1]))
 swap(ptr[c], ptr[c + 1]);
 else
 break;
 }
}
```

```

 if (Compare () (ptr[c + 1], ptr[c])) // sí ==> escoja menor
 c++;

 if (Compare () (ptr[i], ptr[c])) // ¿cumple propiedad orden?
 return; // sí ==> termine

 std::swap(ptr[c], ptr[i]);
 i = c;
 }
}

```

La rutina toma un heap entre  $[l + 1 \dots r]$  y restaura la propiedad de orden. Después de efectuada la llamada, la secuencia deviene un heap completo entre  $[l \dots r]$ .

La duración de `sift_down()` se revela en el siguiente lema:

**Lema 4.2** Sea  $T$  un heap binario de  $n$  elementos con una violación eventual de orden en `array[1]`. Entonces, la duración de `sift_down(array, 1, n)` es  $\mathcal{O}(\lg(n))$ .

**Demostración** El tiempo de ejecución de `sift_down()` está dominado por la cantidad de intercambios que ocasione el elemento violatorio. En este sentido, lo peor que puede ocurrir es que el elemento violatorio alcance el máximo nivel de  $T$ , o sea, su altura. Por tanto, la cantidad máxima de intercambios es proporcional a la altura del árbol. Como  $T$  es completo y, según la proposición 4.1, su altura está acotada por  $\mathcal{O}(\lg(n))$   $\square$

Con lo anterior en mente podemos plantear la eliminación del siguiente modo:

287a  $\langle$  Eliminar raíz en heap 287a  $\rangle \equiv$  (288d)  
`array[1] = array[num_items-];`  
`sift_down<T, Compare>(array, 1, num_items); // propiedad orden`  
la cual, según el lema 4.2, es  $\mathcal{O}(\lg(n))$ .

La segunda manera de eliminar, más general, es planteándonos la supresión de un elemento cualquiera. Para ello, simplemente aplicamos el mismo principio del primer tipo de eliminación: sobreescribir el elemento que deseamos eliminar con el último de la secuencia. La diferencia con el primer método es que aquí pueden presentarse violaciones del orden por el padre o por los hijos.

Llamaremos a la rutina general de supresión `sift_down_up()`, pues se remite sólo a esas acciones:

287b  $\langle$  Rutinas heap 283d  $\rangle + \equiv$  <286 290c>  
`template <typename T, class Compare> inline`  
`void sift_down_up(T * ptr, const size_t & l, const size_t & i, const size_t & r)`  
`{`  
 `sift_down <T, Compare> (ptr, i, r);`  
 `sift_up <T, Compare> (ptr, 1, i);`  
`}`

Con la rutina anterior podemos plantear la eliminación del  $i$ -ésimo elemento del heap:

287c  $\langle$  Eliminar elemento  $i$  287c  $\rangle \equiv$   
`array[i] = array[num_items-];`  
`sift_down_up <T, Compare> (array, 1, i, n);`

### 4.7.3 Colas de prioridad

Existe una amplísima gama de situaciones en las cuales dinámicamente se requiere manejar un conjunto de claves y conocer el menor elemento en cualquier momento. Para tal tipo de circunstancia se emplea un TAD conocido como “cola de prioridad”, cuyas operaciones y desempeño se resumen en la tabla 4.1.

| Nombre de operación | Eficiencia peor caso  |
|---------------------|-----------------------|
| insert(item)        | $\mathcal{O}(\lg(n))$ |
| top()               | $\mathcal{O}(1)$      |
| remove(n)           | $\mathcal{O}(\lg(n))$ |

Table 4.1: Desempeño de las operaciones de una cola de prioridad

Las colas de prioridad se modelizan directamente mediante el TAD `ArrayHeap<Key>`, ya introducido, y el TAD `BinHeap<Key>`, que será tratado en § 4.7.6 (Pág. 293).

La bandera `array_allocated` indica si se apartó o no el arreglo y se define así:

288a  $\langle \text{miembros privados de } \text{ArrayHeap}<\text{Key}\rangle 283\text{a} \rangle + \equiv \quad (284\text{b}) \triangleleft 284\text{a}$   
 $\quad \text{mutable bool array_allocated;}$

con esta bandera el destructor sabe si se debe o no liberar la memoria:

288b  $\langle \text{miembros públicos de } \text{ArrayHeap}<\text{Key}\rangle 288\text{b} \rangle \equiv \quad (284\text{b}) \triangleright 288\text{c} \triangleright$   
 $\quad \text{virtual } \sim \text{ArrayHeap}()$   
 $\quad \{$   
 $\quad \quad \text{if } (\text{array_allocated and array} \neq \text{NULL})$   
 $\quad \quad \text{delete } [] \text{ array;}$   
 $\quad \}$

La consulta del menor es la operación más simple y rápida de un heap:

288c  $\langle \text{miembros públicos de } \text{ArrayHeap}<\text{Key}\rangle 288\text{b} \rangle + \equiv \quad (284\text{b}) \triangleleft 288\text{b} \triangleright 288\text{d} \triangleright$   
 $\quad \text{T} \& \text{ top}()$   
 $\quad \{$   
 $\quad \quad \text{return array}[1];$   
 $\quad \}$

La inserción y eliminación también son muy sencillas, dado que todo el trabajo ya ha sido efectuado:

288d  $\langle \text{miembros públicos de } \text{ArrayHeap}<\text{Key}\rangle 288\text{b} \rangle + \equiv \quad (284\text{b}) \triangleleft 288\text{c} \triangleright 289\text{a} \triangleright$   
 $\quad \text{T} \& \text{ insert}(\text{const T} \& \text{ key})$   
 $\quad \{$   
 $\quad \quad \langle \text{Añadir elemento al heap 285} \rangle$   
 $\quad \quad \text{return array}[\text{num_items}];$   
 $\quad \}$   
 $\quad \text{T} \text{ getMin}()$   
 $\quad \{$   
 $\quad \quad \text{T ret_val} = \text{array}[1];$   
 $\quad \quad \langle \text{Eliminar raíz en heap 287a} \rangle$   
 $\quad \quad \text{return ret_val;}$   
 $\quad \}$

Recordemos que, según se desprende de los lemas 4.1 y 4.2, los bloques  $\langle \text{Añadir elemento al heap 285} \rangle$  y  $\langle \text{Eliminar raíz en heap 287a} \rangle$  son  $\mathcal{O}(\lg(n))$ . Puesto que el resto del entorno de `insert()` y `getMin()` es  $\mathcal{O}(1)$ , entonces estas operaciones son  $\mathcal{O}(\lg(n))$ .

#### 4.7.3.1 Modificación de prioridad

Es plausible modificar el valor de prioridad de un elemento particular. Como esta operación no altera topológicamente al árbol, la propiedad de forma está garantizada. Lo único que debemos cerciorar es que el orden se preserve, lo cual se hace mediante la siguiente operación:

289a *(miembros públicos de ArrayHeap<Key> 288b) +≡* (284b) ▷ 288d

```
void update(T & data)
{
 const size_t i = &data - array;
 sift_down_up <T, Compare> (array, 1, i, num_items);
}
```

La rutina asume que el valor de prioridad de la  $i$ -ésima entrada del arreglo ha sido modificada y restaura la propiedad de orden.

#### 4.7.4 Heapsort

Un método de ordenamiento, estable, con rendimiento  $\mathcal{O}(n \lg(n))$  para el peor caso, puede explicarse en dos fases:

1. Guarde en una cola de prioridad todos los elementos del arreglo.
2. Extraiga iterativamente de la cola y guarde el elemento extraído en la posición de iteración  $i$ . Puesto que la cola siempre retorna el menor elemento, al final de esta iteración, el arreglo estará ordenado.

Este principio puede llevarse a cabo del siguiente modo:

289b *(Heapsort con cola de prioridad 289b) ≡*

```
ArrayHeap<T, n> heap;
for (int i = 0; i < n; ++i) // inserta elementos en heap
 heap.insert(array[i]);

for (int i = 0; i < n; ++i) // saca en orden de heap y pone en array
 array[i] = heap.getMin();
```

Cada pasada demora  $\mathcal{O}(n) \times \mathcal{O}(\lg(n)) = \mathcal{O}(n \lg(n))$ . Por tanto, el procedimiento es  $\mathcal{O}(n \lg(n)) + \mathcal{O}(n \lg(n)) = \mathcal{O}(n \lg(n))$ , independientemente de la permutación del arreglo. Sin embargo, el uso del heap requiere almacenar todos los elementos del arreglo, por lo que el consumo de espacio es  $\mathcal{O}(n)$ , característica indeseable en muchas circunstancias.

No es difícil aprehender que podemos utilizar el propio arreglo que pretendemos ordenar para guardar el heap. Para eso es necesario invertir la condición de orden de modo tal que la raíz almacene el mayor entre todos los elementos. Esto se instrumenta fácilmente mediante una clase envoltoria que haga la inversión:

289c *(Comparación invertida 289c) ≡*

```
template <class T, class Compare> struct Inversed_Compare
{
 bool operator () (const T & op1, const T & op2) const
 {
 return Compare () (op2, op1);
 }
};
```

De este modo, la primera fase puede plantearse pictóricamente así:



la cual recorre el arreglo desde 2 hasta  $n$ , ejecutando, a cada iteración  $i$ ,  $\text{sift\_up } \langle T, \text{--Inversed\_Compare} < T, \text{ Compare} \rangle \text{(arreglo, } i\text{)}$ , es decir, corrigiendo la eventual violación que acarrearía insertar un nuevo elemento en la posición  $i$ . La primera fase puede codificarse, entonces, como sigue:

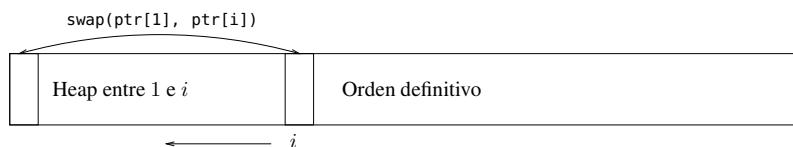
290a

$\langle \text{Convertir arreglo en un heap con sift\_up()} 290a \rangle \equiv$  (290c)

```
for (int i = 2; i <= n; ++i)
 sift_up <T, Aleph::Inversed_Compare<T, Compare> > (array, 1, i);
```

La idea es comenzar desde el heap comprendido entre  $[1 \dots 1]$  y luego, a cada iteración  $i$ , se asume una inserción en la posición  $i$ . Al arreglo entre  $[1 \dots i]$  se le restaura la propiedad de heap mediante  $\text{sift\_up}(array, 1, i)$ .

Al final de la primera fase, todo el arreglo es un heap cuyo máximo elemento se encuentra en la raíz. A partir del hecho de que la posición definitiva del mayor elemento es  $n$ , planteamos la segunda fase del siguiente modo:



la cual se implanta así:

290b

$\langle \text{Ordenar a partir del heap 290b} \rangle \equiv$  (290c 291c)

```
for (int i = n; i > 1; -i)
{
 std::swap(array[1], array[i]); // poner en raíz i -ésimo item
 sift_down<T, Aleph::Inversed_Compare<T, Compare> > (array, 1, i - 1);
}
```

Grosso modo, se intercambia  $\text{array}[1]$  con  $\text{array}[i]$ , lo cual fija el orden definitivo en el intervalo  $[i \dots n]$ . Este intercambio es equivalente a que se hubiese eliminado del heap el menor elemento, es decir, el de la raíz. Así pues,  $\text{sift\_down}(array, 1, i - 1)$  restaura el heap entre  $[1 \dots i - 1]$ .

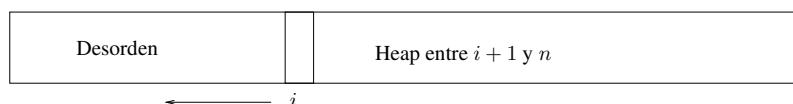
Con lo anterior, podemos finiquitar el método de ordenamiento:

290c

$\langle \text{Rutinas heap 283d} \rangle + \equiv$  ▷287b 291c▷

```
template <typename T, class Compare = Aleph::less<T> >
void heapsort(T * array, const size_t & n)
{
 -array; //desplazar posición hacia atrás ==> array[1] == primero
 ⟨ Convertir arreglo en un heap con sift_up() 290a ⟩
 ⟨ Ordenar a partir del heap 290b ⟩
}
```

Existe una manera alterna al bloque  $\langle \text{Convertir arreglo en un heap con sift\_up()} 290a \rangle$ , que logra, más eficientemente, en  $\mathcal{O}(n)$ , "heapifizar" el arreglo. Para aproximarnos, consideremos la primera pasada del heapsort de la siguiente forma:



o sea, un barrido desde la última posición  $\text{array}[n]$  hasta la primera  $\text{array}[1]$ . Podemos hacer esto mediante `sift_down()`:

291a  $\langle \text{Convertir arreglo en un heap con } \text{sift\_down}() \text{ 291a} \rangle \equiv$

```
for (int i = n - 1; i > 1; -i)
 sift_down <T, Aleph::Inversed_Compare<T, Compare> > (array, i, n);
```

El análisis de este bloque es un poco más sutil. Cada llamada a `sift_down()` cuesta  $\mathcal{O}(\lg(r) - \lg(l))$ , lo que nos da:

$$\sum_{i=1}^{n-1} \mathcal{O}(\lg(r) - \lg(l)) = \sum_{i=1}^{n-1} \mathcal{O}\left(\lg \frac{r}{l}\right) = \mathcal{O}(n) ; \quad (4.34)$$

coste mejor que el de  $\mathcal{O}(n \lg(n))$  asociado al bloque  $\langle \text{Convertir arreglo en un heap con } \text{sift\_up}() \text{ 290a} \rangle$ .

Hay aún una mejora demás si aprehendemos el hecho de que para  $l > \frac{n}{2}$ ,  $\text{array}[1..n]$  es, con certitud, un heap en propiedad de orden, pues para  $\forall l > \frac{n}{2} \implies 2l > n$ , es decir, no hay descendientes. Por tanto, la velocidad de  $\langle \text{Convertir arreglo en un heap con } \text{sift\_down}() \text{ 291a} \rangle$  puede duplicarse de la siguiente manera:

291b  $\langle \text{Convertir arreglo en un heap con } \text{sift\_down}() \text{ mejorado 291b} \rangle \equiv \quad (291c)$

```
for (int i = n/2; i >= 1; -i)
 sift_down <T, Aleph::Inversed_Compare<T, Compare> > (array, i, n);
```

lo que nos arroja la siguiente nueva versión mejorada del heapsort:

291c  $\langle \text{Rutinas heap 283d} \rangle + \equiv \quad \triangleleft 290c$

```
template <typename T, class Compare = Aleph::less<T> >
void faster_heapsort(T * array, const size_t & n)
{
 -array; // desplazar posición hacia atrás ==> array[1] == primero
 <Convertir arreglo en un heap con }sift_down() mejorado 291b>
 <Ordenar a partir del heap 290b>
}
```

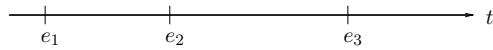
#### 4.7.5 Aplicaciones de los heaps

Los heaps son muy importantes en situaciones en las cuales se requiera mantener y conocer rápidamente los valores límites de un conjunto según algún criterio de orden, es decir, su(s) menor(es) elemento(s) o su(s) mayor(es). Hay abundantes ocasiones en que esto es necesario. He aquí una lista incompleta de las que creemos son las más populares:

1. **Aritmética flotante:** para la precisión de punto flotante es crítico el orden en que se hagan las operaciones, el cual, casi nunca necesariamente, corresponde con el expresado por el programador. Por ejemplo, la suma pierde precisión si un número muy pequeño es sumado (o multiplicando) a uno grande. Un esquema de alta precisión mantiene en un heap los operandos, de manera tal que la suma privilegia los operandos pequeños.
2. **Simulación a eventos discretos:** muchas situaciones de la vida real se modelizan y estudian mediante “eventos”. *Grosso modo*, un evento es una representación objetiva de algún suceso de interés dentro de la situación que se desea estudiar. Como objeto, un evento tiene dos atributos fundamentales:

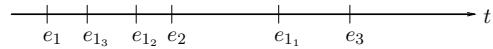
- (a) Tiempo de ocurrencia  $t_e$ .
- (b) Acción modificante sobre el estado del sistema, la cual se divide en la modificación de las variables del modelo y, sobre todo, en la generación de nuevos eventos en el futuro.

Los sistemas de simulación mantienen un tiempo simulado correspondiente al tiempo en que se ejecuta la simulación del evento actual. Inicialmente se programan algunos eventos iniciales a ejecutarse en algunos tiempos fijos. Por ejemplo, consideremos tres eventos iniciales a ejecutarse en tres tiempos:



El tiempo de simulación comienza en el tiempo  $t_{e_1}$ , que es cuando ocurriría  $e_1$ . Los eventos se ponen en una lista ordenada por tiempo. El simulador, sistemáticamente, extrae de la lista el próximo evento en el tiempo. Esto es lo maravilloso de un simulador a eventos discretos, pues el tiempo se simula mediante desplazamientos discretos entre los eventos y no continuamente como sucede en la vida real.

Cada evento genera nuevos eventos, cuyos tiempos de ocurrencia se conocen durante la generación. Supongamos que  $e_1$  genera los eventos  $e_{1_1}, e_{1_2}, e_{1_3}$ , cuyos tiempos de ocurrencia se representan así:



Observemos que los nuevos eventos  $e_{1_3}, e_{1_2}$  ocurrirán antes que  $e_2$ ; esto quiere decir que el próximo evento a simular es  $e_{1_3}$  y no  $e_2$ , cual era el que estaba programado antes de simular  $e_1$ . El simulador se percata de esto porque la lista está abstractamente ordenada por tiempo. Mantener una secuencia ordenada por tiempo, no sólo es costoso en duración, sino inútil, pues lo que en realidad se requiere es conocer el próximo evento más inmediato en tiempo, es decir, el menor.

La lista de eventos de un simulador se implementa mediante un heap. El simulador extrae del heap el menor evento en el tiempo. El heap permite también simular eficientemente la cancelación de un evento. Para ello se usa la operación `sift_down_up()`.

3. Códigos de Huffman: serán estudiados en § 4.13 (Pág. 345).
4. Búsqueda: Encontrar los  $m$  menores elementos de un conjunto de cardinalidad  $n \gg m$ , por ejemplo, conocer los 1000 menores elementos entre un conjunto de  $10^9$  claves. En este caso se debe usar un “heap finito”, es decir, uno cuya capacidad esté limitada a  $m$ . A medida que el heap se va actualizando se debe conocer el mayor elemento entre los  $m$  almacenados, de manera tal que éste sea substituido por el nuevo menor encontrado.
5. Procesamiento ordenado: en general, toda situación en la cual se requiera procesar de primero el menor elemento, es muy susceptible de considerar un heap.

#### 4.7.6 El TAD BinHeap<Key>

Hay situaciones algorítmicas, sobre todo aquellas que aspiran a la generalidad, en las cuales no se tiene idea exacta de la cantidad de elementos que se procesarán. Este desconocimiento, aunado a la escala, puede complicar y comprometer considerablemente el uso de un arreglo, pues, al menos, nuestra implantación `ArrayHeap<Key>` requiere que conozcamos su dimensión. Si la dimensión es muy grande y se usan varios heaps (lo cual puede ser el caso en algoritmos sobre grafos), entonces se puede incurrir en un desperdicio de memoria importante. Si la dimensión es pequeña, entonces se puede sacrificar procesamiento de alta escala. En estos tipos de situaciones se debe considerar un heap altamente dinámico que consuma memoria exactamente proporcional a la cantidad de elementos que maneja.

Una alternativa, delegada en ejercicio, tanto en estudio como en diseño e implantación, es utilizar arreglos dinámicos del tipo `DynArray<T>` desarrollado en § 2.1.4 (Pág. 34). Esta variante, aunque más flexible que el mero arreglo estático, también puede incurrir en los mismos problemas, pues se tiene un desperdicio debido a las entradas no usadas del directorio, segmento y bloque.

Una desventaja de implementar un heap mediante un arreglo es que, puesto que los elementos se mueven, no se pueden mantener apuntadores sobre ellos. Este inconveniente puede paliarse si en el `ArrayHeap<Key>` guardamos punteros tipeados a los datos y programamos la clase de comparación para que opere sobre punteros tipeados. Pero eso consume espacio y duración adicionales debido al apuntador extra y a su dereferencia.

En esa sección desarrollaremos una solución basada en nodos binarios del tipo `BinNode<Key>` y definida en el archivo:

293

```
<tpl_binHeap.H 293>≡
<Definición de nodo de BinHeap<Key> 295c>
 template <template <class> class NodeType, typename Key,
 class Compare = Aleph::less<Key> >
class GenBinHeap
{
 typedef NodeType<Key> Node;
<Atributos de BinHeap<Key> 295a>
<Miembros privados de BinHeap<Key> 296a>
<Miembros públicos de BinHeap<Key> 295b>
};

 template <class Key, typename Compare = Aleph::less<Key> >
class BinHeap : public GenBinHeap<BinHeapNode, Key, Compare>
{
 typedef BinHeapNode<Key> Node;
};
```

`GenBinHeap<NodeType, Key, Compare>` implanta un heap binario mediante nodos binarios de tipo `NodeType`.

Puesto que un heap binario es un árbol binario, podría pensarse que el tipo `BinNode<Key>` basta para implementar un heap altamente dinámico, pero hay dos dificultades:

1. El TAD `BinNode<Key>` no tiene medio directo de conocer el padre, el cual se requiere en las operaciones `sift_up()` y `sift_down()`.

Hay dos maneras de sortear este obstáculo. La primera es mediante una pila que guarde

el camino de acceso desde la raíz hasta el nodo sobre el cual se ejecuta `sift_up()` o `sift_down()`. La segunda consiste en mantener, por cada nodo, un puntero adicional al padre.

El primer enfoque tiene la ventaja de que consume menos espacio que el segundo, pero el manejo de la pila desde la raíz hasta el nodo de operación consume un tiempo  $\mathcal{O}(\lg(n))$ .

2. Cuando ocurre una inserción o eliminación, es necesario acceder al padre del nodo más a la derecha del último nivel, pues puede requerirse actualizar sus enlaces. Sea `last` el nodo más a la derecha del último nivel. ¿Cómo conocer este nodo y su padre? Percatémosnos de que `last` y su padre pueden cambiar cada vez que ocurre una modificación sobre heap.

Un enfoque para determinar `last` consiste en dividir sucesivamente el valor de cardinalidad del heap. Si la cardinalidad es  $n$ , entonces  $n \bmod 2$  nos indica si `last` es hijo izquierdo o derecho. Sucesivamente,  $n \bmod 4$  nos dice si el padre de `last` es descendiente izquierdo o derecho. El procedimiento se repite hasta alcanzar la raíz, es decir, hasta que  $n/i = 1$ . En este momento tenemos el número de Deway, invertido, de `last` y, a partir de él, toda su ascendencia.

El cálculo anterior es  $\mathcal{O}(\lg(n))$ , tanto en espacio como en duración. Podría pensarse que el mantener permanentemente la secuencia de Deway hacia `last` ahorra tiempo, pero en el peor caso puede ser necesario actualizarla hasta la raíz y esto cuesta  $\mathcal{O}(\lg(n))$ .

En nuestro diseño transaremos por la rapidez. Para sortear el primer obstáculo usaremos un puntero al padre, el cual posibilita todo el contexto para las operaciones de intercambio entre niveles requeridas por `sift_up()` y `sift_down()`.

Para evitar el segundo obstáculo aprovecharemos los  $n + 1$  punteros nulos del heap para mantener una lista, doblemente enlazada, circular, de los nodos borde de los últimos niveles, es decir, una lista conformada por los nodos hojas y, eventualmente, el nodo de padre de `last`, el cual es incompleto cuando `last` es hijo izquierdo. Una instancia de esta estructura de datos puede pictorizarse como en la figura 4.31. `last` siempre apunta al

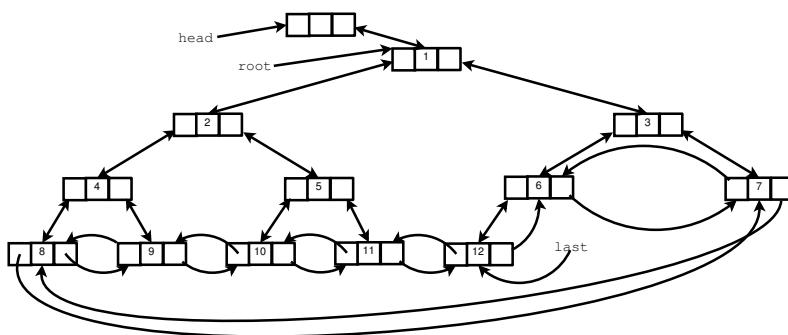


Figura 4.31: Representación del TAD BinHeap<Key>

nodo más a la derecha del último nivel. `head` es un nodo cabecera, centinela, que permite tratar de manera general el intercambio entre la raíz y alguno de sus dos hijos. `root` es un puntero referencial, permanente, a la raíz, que es hija derecha de `head`. Requerimos entonces los siguientes atributos para el TAD BinHeap<Key>:

295a *(Atributos de BinHeap<Key> 295a)≡* (293)  
 Node head\_node;  
 Node \* head;  
 Node \*& root;  
 Node \* last;  
 size\_t num\_nodes;

El último atributo, num\_nodes, contabiliza la cardinalidad del heap.

Un BinHeap<Key> se inicializa así:

295b *(Miembros públicos de BinHeap<Key> 295b)≡* (293) 297▷  
 GenBinHeap()  
 : head(&head\_node), root(RLINK(head)), last(&head\_node), num\_nodes(0) {}

Las consideraciones de diseño anteriores nos llevan a incluir la siguiente información adicional por nodo

295c *(Definición de nodo de BinHeap<Key> 295c)≡* (293)

```
class BinHeapNode_Data
{
 struct Control_Fields // Definición de banderas de control
 {
 int is_leaf : 4; // true si el nodo es hoja
 int is_left : 4; // true si el nodo es hijo izquierdo
 };
 BinHeapNode_Data * pLink; // puntero al padre
 Control_Fields control_fields;

 BinHeapNode_Data() : pLink(NULL)
 {
 control_fields.is_leaf = true;
 control_fields.is_left = true;
 }

 BinHeapNode_Data *& getU() { return pLink; }

 Control_Fields & get_control_fields() { return control_fields; }
};

DECLARE_BINNODE(BinHeapNode, 64, BinHeapNode_Data);
```

Uses DECLARE\_BINNODE.

Esta clase nodo justifica definir los macros siguientes a favor de la legibilidad de código:

295d *(Definición de macros de BinHeap<Key> 295d)≡*  
 # define PREV(p) (p->getL())
 # define NEXT(p) (p->getR())
 # define ULINK(p) reinterpret\_cast<Node\*&>((p)->getU())
 # define IS\_LEAF(p) ((p)->get\_control\_fields().is\_leaf)
 # define IS\_LEFT(p) ((p)->get\_control\_fields().is\_left)
 # define CTRL\_BITS(p) ((p)->get\_control\_fields())

Si IS\_LEAF(p) == true, entonces NEXT(p) y PREV(p) contienen el sucesor y predecesor de la lista de hojas. Si, por el contrario, IS\_LEAF(p) == false, entonces los mismos campos se usan como LLINK(p) y RLINK(p).

Mediante el campo IS\_LEAF(p) podemos determinar si p pertenece o no a la lista enlazada:

296a *(Miembros privados de BinHeap<Key> 296a)≡* (293) 296b▷

```
static bool is_in_list(Node * p)
{
 if (IS_LEAF(p))
 return true;
 return ULINK(LLINK(p)) == RLINK(LLINK(p));
}
```

El segundo `return` verifica si `p` es el padre de `last` y éste es su hijo izquierdo <sup>8</sup>.

Otro predicado que requeriremos es conocer si un nodo tiene o no un hermano:

296b *(Miembros privados de BinHeap<Key> 296a)+≡* (293) ▷296a 296c▷

```
static bool has_sibling(Node * p)
{
 return ULINK(p) != RLINK(p);
}
```

Usamos este método porque opera aún si `p` es parte de la lista enlazada.

#### 4.7.6.1 Intercambio entre nodos

Quizá la rutina esencial de nuestra implantación es la que intercambia dos nodos de nivel. Llamaremos a esta rutina `swap_with_parent(p)`, cuya misión es intercambiar el nodo `p` con su parent. Esta es la rutina más complicada de la implantación porque el intercambio debe distinguir los casos particulares en los cuales `p` no tiene abuelo (cuando sólo hay dos o tres nodos) y el caso en que `p` esté en último nivel, situación en la cual `p` es parte de la lista.

La rutina `swap_with_parent()` es bastante complicada y no se muestra en este texto (ella puede examinarse en la biblioteca). `swap_with_parent()` opera independientemente de la situación de `p` dentro del heap. Podemos, pues, emplearla para efectuar los intercambios requeridos por `sift_up()` y `sift_down()`, las cuales, una vez efectuada la operación `swap_with_parent()`, son mucho más simples que su contraparte con arreglos:

296c *(Miembros privados de BinHeap<Key> 296a)+≡* (293) ▷296b 298a▷

```
virtual void sift_up(Node * p)
{
 while (p != root and Compare() (KEY(p), KEY(ULINK(p))))
 swap_with_parent(p);
}
virtual void sift_down(Node * p)
{
 while (not IS_LEAF(p))
 {
 Node * cp = LLINK(p); // guarda el menor hijo de p
 if (has_sibling(cp))
 if (Compare() (KEY(RLINK(p)), KEY(LLINK(p))))
 cp = RLINK(p);

 if (Compare() (KEY(p), KEY(cp)))
 return;
 }
}
```

---

<sup>8</sup>Para aprehender este caso, ejecútese la rutina con `p=6` en la figura 4.31.

```

 swap_with_parent(cp);
 }
}

```

#### 4.7.6.2 Inserción en BinHeap<Key>

La rutina `sift_up()` nos permite especificar la primera rutina pública de `BinHeap<Key>`: la inserción, la cual se define del siguiente modo:

```

297 <Miembros públicos de BinHeap<Key> 295b>+≡ (293) ▷295b 298b▷
Node * insert(Node * p)
{
 if (root == NULL) // ¿heap está vacío?
 { // Sí, inicialice
 root = p;
 LLINK(p) = RLINK(p) = p;
 ULINK(p) = head;
 IS_LEAF(p) = true;
 IS_LEFT(p) = false; /* root is right child of header node */
 last = root;
 num_nodes = 1;
 return p;
 }
 // inserción general
 Node * pp = RLINK(last); // padre de actual last
 LLINK(p) = last;
 ULINK(p) = pp;

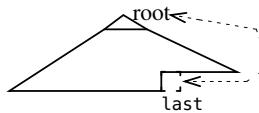
 if (IS_LEFT(last))
 { // p será hijo derecho
 IS_LEFT(p) = false;
 RLINK(p) = RLINK(pp);
 LLINK(RLINK(pp)) = p;
 RLINK(pp) = p;
 }
 else
 { // p será hijo izquierdo
 IS_LEFT(p) = true;
 RLINK(p) = pp;
 IS_LEAF(pp) = false; // si p es izquierdo ==> pp era hoja
 LLINK(pp) = p;
 }
 RLINK(last) = p;
 last = p;
 num_nodes++;
 sift_up(last);
 return p;
}

```

#### 4.7.6.3 Eliminación del elemento mínimo

Para la eliminación por la raíz aplicamos la misma estrategia explicada en § 4.7.2 (Pág. 285): intercambiar la raíz por last y luego “cortar” por last. Para ello nos valemos en primer lugar de una rutina `swap_root_with_last()`, cuya misión es intercambiar la raíz con last y que no se muestra en este texto (ella también puede ser examinada en la biblioteca).

Pictóricamente, `swap_root_with_last()`, a nivel de los nodos, no de sus contenidos, efectúa la siguiente operación:



Falta “cortar” el nodo más a la derecha del último nivel, lo cual se hace así:

298a *(Miembros privados de BinHeap<Key> 296a) +≡ (293) ↳ 296c 301a ▷*

```
Node * remove_last()
{
 Node * ret_val = last;
 Node * pp = ULINK(last);
 Node * new_last = LLINK(last);

 if (IS_LEFT(last))
 {
 IS_LEAF(pp) = true;
 LLINK(pp) = new_last;
 }
 else
 {
 RLINK(pp) = RLINK(last);
 LLINK(RLINK(last)) = pp;
 }
 RLINK(LLINK(last)) = pp;
 last = new_last;
 num_nodes--;
 ret_val->reset();

 return ret_val;
}
```

Esta maquinaria nos permite definir `getMin()` exactamente como explicado en § 4.7.2 (Pág. 285), es decir, intercambiar `root` con `last`, luego cortar por `last`:

298b *(Miembros públicos de BinHeap<Key> 295b) +≡ (293) ↳ 297 299a ▷*

```
Node * getMin()
{
 Node *ret_val = root;

 if (num_nodes == 1)
 {
 root = NULL;
 ret_val->reset();
 num_nodes = 0;
```

```

 return ret_val;
 }
 swap_root_with_last();
 remove_last();
 sift_down(root);
 ret_val->reset();

 return ret_val;
}

```

#### 4.7.6.4 Actualización en un heap

Al igual que con `ArrayHeap<Key>`, es plausible modificar el valor de prioridad de un nodo particular. En este caso efectuamos la misma corrección:

299a *(Miembros públicos de BinHeap<Key> 295b) +≡* (293) ◁ 298b 299b ▷

```

void update(Node * p)
{
 sift_down(p);
 sift_up(p);
}

```

#### 4.7.6.5 Eliminación de cualquier elemento

Quizá una de las operaciones que más cobre sentido con este tipo de implantación de heap es la de eliminar, arbitrariamente, cualquier elemento, dada la dirección del nodo. Un ejemplo de esta situación es la cancelación de un evento, tanto en un sistema de simulación a eventos discretos como en un manejador de eventos en tiempo real<sup>9</sup>. En este caso, teniendo un apuntador al evento, cual puede, por ejemplo, derivar de `BinHeap<Time>::Node`, puede eliminarse del heap cualquier elementos.

Es conveniente percatarse de que en la implantación del heap con arreglos, esta operación requiere un cierre de brecha que es  $\mathcal{O}(n)$ .

Para eliminar `node` se opera parecido a su contraparte `ArrayHeap<Key>`: (1) se desenlaza `last` del árbol y se respalda en una variable `p`; esto se realiza con la operación, ya definida, `remove_last()`. Luego, (2) se substituye el nodo a eliminar `node` por `p` y, finalmente, se ejecuta `sift_down()` y `sift_up()` para ajustar la propiedad de orden.

El paso (2) de la operación anterior se vale la operación `replace_node(Node * node, Node * new_node)`, cuyo rol es reemplazar dentro del árbol binario el nodo `node` por `new_node`. Todo el contexto necesario está dado, pues cada nodo contiene un puntero a su padre. Esta operación puede consultarse en la biblioteca.

Ahora, sólo nos falta remitirnos a la técnica explicada para implantar la eliminación de un nodo `node`:

299b *(Miembros públicos de BinHeap<Key> 295b) +≡* (293) ◁ 299a 301b ▷

```

Node * remove(Node * node)
{
 if (node == root)

```

<sup>9</sup>En *ALÉPH*, existe un TAD, llamado `TimeoutQueue`, cuyo fin es modelizar un manejador de eventos programados por el reloj del sistema. La implantación de `TimeoutQueue` hace uso del TAD `BinHeap<Key>` y ofrece, entre sus funciones, la cancelación de un evento previamente programado.

```

 return getMin();

 if (node == last)
 return remove_last();

 Node * p = remove_last();

 if (node == last)
 {
 remove_last();
 insert(p);
 return node;
 }
 replace_node(node, p);
 update(p);
 node->reset();

 return node;
}

```

#### 4.7.6.6 Destrucción de BinHeap<Key>

Al igual que con otros enfoques de solución del problema fundamental que hemos estudiado hasta el presente, en lugar de manejar directamente las claves del conjunto, el TAD BinHeap<Key> maneja los nodos que las contienen. Del mismo modo, en función del principio fin-a-fin, otro TAD, en nuestro caso, DynBinHeap<Key> se encarga de manejar las claves sin que el usuario tenga por qué pensar en los nodos como objetos de manipulación del heap.

Cuando implantamos una versión del problema fundamental basada en claves y no en los nodos que la contengan, una operación fundamental del TAD de base es la posibilidad de liberar todos los elementos -y toda la memoria ocupada- en una sola operación. Este es el caso, por ejemplo, de las operaciones `remove_all_and_delete()` (§ 2.4.7 (Pág. 72)) y `destroyRec()` (§ 4.4.10 (Pág. 250)), diseñadas para las listas enlazadas y los árboles binarios respectivamente.

En el caso de un heap dinámico como BinHeap<Key>, o del destructor del TAD DynBinHeap<Key>, no es posible invocar a la operación `destroyRec()`, pues a causa de la complejidad del nodo de un BinHeap<Key> no se dispone del valor `NullPtr`, del cual se sirve `destroyRec()` para reconocer las hojas.

Una tentación para tratar con este problema, es decir, para eliminar todos los elementos de BinHeap<Key>, es un ciclo del siguiente estilo:

300 *(Eliminar todos los elementos del BinHeap<Key> 300)≡*  
`while (not this->is_empty())
 delete getMin();`

Este bloque cumple la función, pero a expensas de un coste en tiempo que puede devenir importante, pues es  $\mathcal{O}(n \lg(n))$ , que es mayor que  $\mathcal{O}(n)$ , el tiempo que nos lleva el barrido de los elementos y su correspondiente eliminación. Desde esta perspectiva se nos revela necesario proveer desde el TAD BinHeap<Key> una primitiva que elimine todos los elementos en  $\mathcal{O}(n)$ . En tal sentido, haremos un recorrido sufijo, recursivo, que libere los

nodos del heap:

301a *(Miembros privados de BinHeap<Key> 296a) +≡* (293) ▷298a

```
static void __postorder_delete(Node * p, Node * incomplete_node)
{
 if (IS_LEAF(p))
 {
 delete p;
 return;
 }
 __postorder_delete(LLINK(p), incomplete_node);

 if (p != incomplete_node)
 __postorder_delete(RLINK(p), incomplete_node);

 delete p;
}
```

Puesto que en la estructura de datos se combina la estructura árbol binario con lista doblemente enlazada, se debe tener especial cuidado cuando el último nodo (*last*) sea izquierdo; sólo en este caso, *p* sería un nodo de un solo hijo, por lo que la llamada hacia la derecha sería incorrecta y causante de una doble eliminación.

Esta rutina se invoca por la siguiente interfaz pública:

301b *(Miembros públicos de BinHeap<Key> 295b) +≡* (293) ▷299b

```
void remove_all_and_delete()
{
 if (root == NULL)
 return;

 if (num_nodes <= 3)
 {
 while (not this->is_empty())
 delete getMin();
 return;
 }
 if (IS_LEFT(last))
 __postorder_delete(root, ULINK(last));
 else
 __postorder_delete(root, NULL);

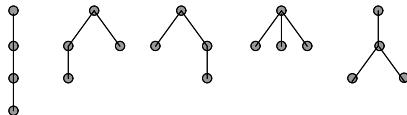
 root = NULL; // reiniciar como si fuese constructor
 last = &head_node;
 num_nodes = 0;
}
```

Dado que el heap es un árbol completo, el riesgo de desborde de pila debido a la recursión es mínimo.

La rutina *remove\_all\_and\_delete()* es invocada por el destructor del TAD *DynBinHeap<Key>*, el cual, como es de suponer, es una extensión de *BinHeap<Key>* orientada a manejar claves. De este modo, la destrucción es  $\mathcal{O}(n)$ .

## 4.8 Enumeración y códigos de árboles

Abordemos un problema combinatorio que, si bien ataÑe a los árboles, relaciona a muchos otros problemas combinatorios: ¿cuántos árboles diferentes pueden construirse con  $n$  nodos? Por ejemplo, estos son todos los posibles árboles de 4 nodos:



Recordemos que existe una correspondencia unívoca entre los árboles y los árboles binarios (§ 4.4.17 (Pág. 260)). Dado un árbol cualquiera de  $n$  nodos, entonces, según § 4.4.17 (Pág. 260), la raíz de su equivalente binario no tiene rama derecha. Faltan, entonces,  $n-1$  nodos para combinar en la rama izquierda. Por lo tanto, el número de posibles árboles que se pueden construir con  $n$  nodos es equivalente al número de árboles binarios que se pueden construir con  $n-1$  nodos. Podemos resolver este problema de conteo en el contexto más familiar y, quizá más simple, de los árboles binarios.

### 4.8.0.7 Códigos de un árbol binario

Comencemos por enunciar el concepto de código de un árbol binario. Para ello dibujemos un árbol extendido -con sus nodos externos- tal como el de la figura 4.32.

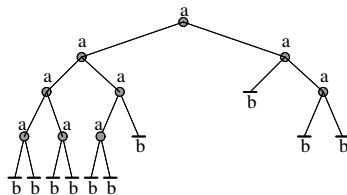


Figura 4.32: Un árbol binario extendido: nodos internos etiquetados con  $a$ ; nodos externos con  $b$

**Definición 4.7** Sea un árbol binario  $T$  de  $n$  nodos. Se define  $\text{codigo}(T) : \mathcal{B} \longrightarrow \{a, b\}^*$  como el recorrido prefijo del árbol extendido donde sus  $n$  nodos internos están etiquetados con  $a$  y sus externos con  $b$ .

Por ejemplo, para el árbol de la figura 4.32,  $\text{codigo}(T) = aaaaabbabbaabbbaabb$ .

Estudiemos cómo se caracteriza una palabra  $\text{codigo}(T) \in \{a, b\}^*$ . De entrada, asumamos que  $L \subset \{a, b\}^*$  es el lenguaje que caracteriza a un código de un árbol.  $L$  es llamado lenguaje de Lukasiewicz, en honor al matemático polaco Jan Lukasiewicz.

**Proposición 4.6** El lenguaje de Lukasiewicz, de todos los códigos posibles está definido por:

$$L = aLL \cup b \quad (4.35)$$

En otras palabras,  $L = \text{codigo}(\mathcal{B})$ .

La expresión (4.35) es una definición recursiva del conjunto  $L$  y significa que una palabra  $L$  es el resultado de concatenar  $a$  con dos palabras seguidas que pertenezcan al conjunto  $L$  o con la letra  $b$ . Lo más interesante de este tipo de definición es que hemos definido un conjunto infinito de palabras en una sola expresión recursiva.

**Demostración** Para demostrar  $L = \text{codigo}(\mathcal{B})$  debemos demostrar  $L \subset \text{codigo}(\mathcal{B})$  y  $L \supset \text{codigo}(\mathcal{B})$

- $aLL \cup b \subset \text{codigo}(\mathcal{B})$ : Si  $T = \emptyset$ , tenemos que  $\text{codigo}(\emptyset) = b \subset aLL \cup b$ . Si  $T \neq \emptyset$ , entonces  $T$  puede expresarse como  $T = \langle L(T), \text{raiz}(T), R(T) \rangle$ . Por la definición de código,  $\text{codigo}(\langle L(T), \text{raiz}(T), R(T) \rangle) = a \text{codigo}(L(T)) \text{codigo}(R(T)) \subset \text{codigo}(\mathcal{B})$   $\square$

- $\text{codigo}(\mathcal{B}) \subset aLL \cup b$ : Sea  $w \in L$ . Si  $|w| = 1 \Rightarrow w = \text{codigo}(\emptyset) = b$ .

De lo contrario,  $|w| > 1 \Rightarrow w = uv$ ,  $u, v \in L$ ; donde  $a$  corresponde a la raíz de un árbol  $T$  mientras que  $u = \text{codigo}(L(T))$  y  $v = \text{codigo}(R(T))$   $\blacksquare$

Cuando tenemos una palabra  $w = uv$ , es decir, que puede componerse de la concatenación de otras dos palabras  $u$  y  $v$ , entonces  $a u$  se le dice que es prefijo de  $w$ . Por ejemplo, si  $w = abba$ , entonces  $\epsilon, a, ab, abb, abba$  son prefijos de  $w$ <sup>10</sup>.

En lo que respecta a los prefijos, denotaremos:

$$\begin{cases} u \leq v & \text{si } u \text{ es prefijo de } v \\ u < v & \text{si } u \text{ es prefijo de } v \text{ y } u \neq v. \text{ (En este caso decimos que } u \text{ es prefijo propio de } v\text{)} \end{cases}$$

Para el ejemplo anterior,  $a < abba$ , pero  $abba \leq abba$ .

**Definición 4.8 (Conjunto prefijo)** Un conjunto de palabras es prefijo si ninguna de sus palabras es prefijo de alguna de las otras. Esto es, dado un lenguaje  $X$ , se dice que  $X$  es un lenguaje prefijo  $\iff \forall u, v \in X, u$  y  $v$  no son prefijos entre sí. Es decir,  $u \leq v \Rightarrow u = v$ .

Por ejemplo,  $X = \{ab, baa, bab\}$ .

**Lema 4.3** Sea  $X$  un lenguaje prefijo y  $u, v, u', v' \in X$ . Entonces,  $uv = u'v' \Rightarrow u = u'$  y  $v = v'$ . En otros términos, si  $u, v, u', v' \in X \mid u \neq u', v \neq v' \Rightarrow uv \neq u'v'$ .

**Demostración (por contradicción)** Supongamos que  $X$  es un lenguaje prefijo, es decir,  $uv = u'v' \Rightarrow u \neq u'$  o  $v \neq v'$ . Para facilitar la comprensión, superpongamos las palabras  $uv$  y  $u'v'$  como sigue:

|    |  |    |
|----|--|----|
| u  |  | v  |
| u' |  | v' |

En el diagrama se aprecia que  $u' \leq u$ . Podemos imaginar otras superposiciones y suponer otros prefijos. En todo caso, es claro que  $o u \leq u'$  o  $v \leq v'$ . Sin embargo, puesto que hemos dicho que  $X$  es un conjunto prefijo, es imposible que  $u'$  sea prefijo de  $u$ . Puesto que la negación del lema es falsa, el lema es cierto  $\square$

Este lema nos proporciona el argumento principal para demostrar la proposición siguiente.

**Proposición 4.7 (Huffman 1952 [81])** El lenguaje de Lukasiewicz es prefijo.

<sup>10</sup>El símbolo  $\epsilon$  denota la palabra vacía.

**Demostración (por contradicción inductiva sobre la longitud de una palabra)**  
 Supongamos que el lenguaje de Lukasiewicz no es prefijo, entonces  $\exists u, v, x \in L \mid ux = v$ .

- $|u| = 1$ : En este caso,  $u = b \implies x = \epsilon \implies v = b$ , pero  $L$  no contiene a  $\epsilon$  lo que implica que  $L$  es prefijo para todas las palabras de longitud 1.
- Ahora asumimos que  $L$  es prefijo para todas las palabras  $u, v \in L$  de longitud nominal y examinaremos si  $\exists u, x \in L \mid ux = v$ . Es decir, si existe una palabra de mayor longitud que pueda ser compuesta con una palabra perteneciente a  $L$ .

Sean  $u_1, u_2, v_1, v_x, x \in L \mid |u_1| < |u|, |u_2| < |u|, |u_2| < |u|, |v_1| < |v|, |v_2| < |v|$ . Hagamos las siguientes descomposiciones:

$$\begin{aligned} u &= au_1u_2 \\ v &= av_1v_2 \end{aligned}$$

Y plateamos:

$$au_1u_2x = av_1v_2 \implies u_1u_2x = v_1v_2$$

Esto nos arroja dos casos a examinar:

1.  $|u_1| \leq |v_1|$ : En este caso,  $u$  y  $v$  se superponen de la forma siguiente:

|       |       |       |
|-------|-------|-------|
| $u_1$ | $u_2$ | $x$   |
| $v_1$ |       | $v_2$ |

La única posibilidad es que  $u_1 = v_1$ , pues de lo contrario  $u_1$  sería prefijo de  $v_1$ , pero ello no es así por la hipótesis inductiva. Por lo tanto,  $x = \epsilon \notin L$ . El planteamiento es, entonces, contradictorio.

2.  $|v_1| < |u_1| < |u|$ : En este caso,  $u$  y  $v$  se superponen de la forma siguiente:

|       |       |       |
|-------|-------|-------|
| $u_1$ | $u_2$ | $x$   |
| $v_1$ |       | $v_2$ |

De nuevo,  $v_1$  no puede ser prefijo de  $u_1$ , pues si no se contradice la hipótesis inductiva. Por lo tanto,  $v_1 = u_1$  y  $x = \epsilon \notin L$ .

Ambos casos, con  $|u_1| \neq |v_1| \mid |u_1| < |u|, |v_1| < |v|$ , contradicen la hipótesis inductiva. Por lo tanto, la hipótesis es cierta ■

**Proposición 4.8**  $\forall T_1, T_2 \in \mathcal{B}, T_1 \neq T_2 \iff \text{codigo}(T_1) \neq \text{codigo}(T_2)$ .

**Demostración** La prueba se estructura en dos partes:

- **Inyectividad ( $\implies$ )**: Aplicamos inducción sobre las cardinalidades de  $T_1$  y  $T_2$ :  $\forall T_1, T_2 \in \mathcal{B}, T_1 \neq T_2 \implies \text{codigo}(T_1) \neq \text{codigo}(T_2)$ .
  - **Caso base**: Si  $T_1 = \emptyset, T_2 \neq \emptyset \implies \text{codigo}(T_1) = b, \text{codigo}(T_2) = auv, u, v \in L \implies \text{codigo}(T_1) \neq \text{codigo}(T_2)$ .
  - **Hipótesis inductiva**: Si  $T_1 \neq \emptyset, T_2 \neq \emptyset$  y  $|T_1| \neq |T_2| \implies |\text{codigo}(T_1)| \neq |\text{codigo}(T_2)| \implies \text{codigo}(T_1) \neq \text{codigo}(T_2)$ .

Si  $|T_1| = |T_2|$  entonces:

$$\begin{aligned} w_1 &= \text{codigo}(T_1) = au_1v_1 & \begin{cases} u_1 = \text{codigo}(L(T_1)) \\ v_1 = \text{codigo}(R(T_1)) \end{cases} \\ w_2 &= \text{codigo}(T_2) = au_2v_2 & \begin{cases} u_2 = \text{codigo}(L(T_2)) \\ v_2 = \text{codigo}(R(T_2)) \end{cases} \end{aligned}$$

□

Ahora que la proposición es cierta para dos árboles de una misma cardinalidad nominal y probamos si la proposición es cierta para árboles de cardinalidad mayor, por la hipótesis inductiva tenemos que o  $u_1 \neq u_2$  o  $v_1 \neq v_2$ . Debemos responder si:

$$au_1v_1 \neq au_2v_2 \quad (4.36)$$

Para ello, planteemos el siguiente lema:

**Lema 4.4** Sean  $u_1, u_2, v_1, v_2 \in L$ . Si  $u_1 \neq u_2$  o  $v_1 \neq v_2$ , entonces,  $u_1v_1 \neq u_2v_2$ .

**Demostración** La demostración se deriva del lema 4.3. Si negamos el lema 4.3 tenemos que  $u_1 \neq u_2$  o  $v_1 \neq v_2 \implies u_1v_1 = u_2v_2$  con  $u_1, u_2, v_1, v_2 \in \{\text{Lenguaje prefijo}\}$ . Empero por la proposición 4.7 sabemos que  $L$  es un lenguaje prefijo. Así pues, el lema es cierto.

El lema 4.4 nos garantiza que (4.36) es cierto para árboles binarios de cardinalidad mayor, lo cual implica que  $\text{codigo}(T) : \mathcal{B} \rightarrow L$  es inyectiva.

La inyectividad prueba  $\forall T_1, T_2 \in \mathcal{B}, T_1 \neq T_2 \implies \text{codigo}(T_1) \neq \text{codigo}(T_2)$  □

- **Suprayectividad:** En este caso debemos demostrar que:

$$\forall w \in L \exists T \in \mathcal{B} \mid \text{codigo}(T) = w \quad (4.37)$$

La demostración es por inducción sobre la longitud de  $w$ . La hipótesis inductiva es (4.37).

- $|w| = 1 \implies \text{codigo}(\emptyset) = b \implies (4.37)$  es cierto para  $|w| = 1$ .
- Ahora asumimos que (4.37) es cierto para todo  $|w| = n$  y verificamos si la hipótesis es satisfecha para  $|w| = n + 1$ .

Si  $|w| = n + 1$ , la palabra  $w = auv$  satisface  $|auv| = 1 + |u| + |v|$ . Por la hipótesis inductiva,  $\exists T_u \in \mathcal{B} \mid \text{codigo}(T_u) = u$  y  $\exists T_v \in \mathcal{B} \mid \text{codigo}(T_v) = v$  tal que  $|\text{codigo}(T_u)| + |\text{codigo}(T_v)| = n \implies |w| = 1 + |u| + |v| = n + 1$  □

Puesto que  $\text{codigo}(T) : \mathcal{B} \rightarrow L$  es inyectiva y suprayectiva,  $\text{codigo}(T) : \mathcal{B} \rightarrow L$  es biyectiva. Uno de los teoremas fundamentales de la teoría de conjuntos establece que si una función  $f : A \rightarrow B$  es biyectiva, entonces,  $|A| = |B|$ . Por este teorema podemos concluir que  $|\mathcal{B}| = |L|$ . Lo que demuestra que  $\text{codigo}(T_1) \neq \text{codigo}(T_2) \implies \forall T_1, T_2 \in \mathcal{B}, T_1 \neq T_2$  ■

Ahora veamos cómo se caracteriza una palabra en  $L$ . Para ello, definamos:

$$\begin{aligned} |w|_a &= \text{el número de ocurrencias de } a \text{ en } w \\ |w|_b &= \text{el número de ocurrencias de } b \text{ en } w \end{aligned}$$

### Proposición 4.9

$$w \in L \iff \begin{cases} (a) & |w|_b = 1 + |w|_a \quad y \\ (b) & \forall u, v, |u| < |w| \quad |w - uv|_a \geq |w|_b \end{cases} \quad (4.38)$$

## Demostración

- (a) Sea  $T$  un árbol binario cualquiera con  $w = \text{codigo}(T)$ . Por la definición de código sabemos que  $|w|_a = |T|$ ; es decir, el número de nodos internos. Por la proposición 4.2 sabemos que  $T$  extendido tiene  $2|T| + 1$  nodos, es decir,  $n$  nodos internos y  $n + 1$  nodos externos. Así pues,  $|w| = 2|T| + 1 = 2|w|_a + 1 = |w|_a + |w|_b \implies |w|_b = |w|_a + 1 \square$

(b) (inducción sobre  $|w|$ )

- $|w| = 1 \implies w = b \implies u = \epsilon \implies |u|_a = 0 \geq |u|_b = 0$ . (b) es cierto para el caso base.
  - Sea  $|w| = n + 1$  y asumamos (b) cierto para todo  $|w| < n + 1$ . Por definición,  $w$  tiene que estar compuesto como  $w = auv$ . El símbolo  $a$  por la hipotética raíz de un árbol binario y  $u$  y  $v$  los códigos de las rama izquierda y derecha.

Consideremos un prefijo  $w'$  cualquiera de  $w$ , el cual puede tener alguna de las siguientes configuraciones:

|              |     |      |      |
|--------------|-----|------|------|
| $w$          | $a$ | $u$  | $v$  |
| Caso 1: $w'$ | $a$ | $u'$ |      |
| Caso 2: $w'$ | $a$ | $u$  | $v'$ |

En el caso 1,  $|u'| < |u|$ , por lo que la hipótesis inductiva es cierta entre  $u'$  y  $u$ . Por lo tanto,  $|u'|_a \geq |u'|_b$  y, obviamente,  $|au'| \geq |u'|$ .

En el caso 2,  $|v'| < |v|$ , por lo que la hipótesis inductiva es cierta entre  $v'$  y  $v$ , lo cual permite determinar que  $|uv'|_a \geq |uv'|_b$  y, en consecuencia,  $|auv'|_a \geq |auv'|_b$  ■

Esta proposición nos da una manera eficientísima de verificar si una secuencia de bits es una palabra Lukasiewicz, o sea, si una palabra se corresponde con el código de un árbol binario.

#### 4.8.0.8 Códigos de Dick

Una alternativa para codificar árboles, completamente equivalente a los códigos de la sección anterior, es etiquetar el árbol extendido de manera diferente: las ramas izquierdas con a y las derechas con b tal como el árbol de la figura 4.33.

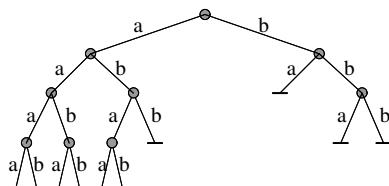


Figura 4.33: Árbol binario extendido: ramas izquierdas etiquetadas con a; ramas derechas con b

Una palabra de Dick, denotada como  $\text{dick}(T) : \mathcal{B} \longrightarrow \{a, b\}^*$  es el recorrido prefijo de un árbol etiquetado con sus ramas izquierdas en  $a$  y sus ramas derechas en  $b$ . Por ejemplo, para el árbol de la figura 4.33:

$$\text{dick}(T) = aaaabbabbaabbbbabab$$

Es fácil obtener una definición recursiva de una palabra de Dick. En primer lugar definimos  $\text{dick}(\emptyset) = \epsilon$ . Para una árbol no vacío, en su versión extendida,  $T = < L(T), \text{raiz}(T), R(T) >$ ,  $\text{dick}(< L(T), \text{raiz}(T), R(T) >) = a \text{dick}(L(T))b \text{dick}(R(T))$ . Así pues, si  $D$  es el conjunto de todas las palabras de Dick, entonces

$$D = aDbD \cup \epsilon$$

**Proposición 4.10** Sea  $D$  el lenguaje de todas las palabras de Dick, entonces

$$Db = L$$

En otras palabras,  $\text{codigo}(T) = \text{dick}(T)b$ .

#### Demostración

$$\begin{aligned} D &= aDbD \cup \epsilon \text{ concatenamos la ecuación con } b \implies \\ Db &= (aDbD \cup \epsilon)b \implies \\ \underbrace{Db}_{L} &= a \underbrace{Db}_{L} \underbrace{Db}_{L} \cup b \quad \text{según (4.35)} \blacksquare \end{aligned}$$

Estamos en capacidad de formular un nuevo problema combinatorio: ¿cuántas expresiones que utilicen  $n$  paréntesis izquierdos y  $n$  paréntesis derechos están balanceadas? Por ejemplo,  $((())())()$ , es una expresión balanceada, mientras que  $()((((())())())$ ) no lo es (falta un paréntesis izquierdo).

La respuesta está dada por la cantidad de palabras de Dick de longitud  $2n$ . En efecto, si consideramos el carácter  $a$  como el paréntesis izquierdo y el  $b$  como el derecho, una palabra de Dick representa una expresión con paréntesis bien formada.

#### 4.8.1 Números de Catalan

Ahora retomemos el problema de contar la cantidad de árboles binarios que contienen  $n-1$  nodos. Notemos que por la proposición 4.8, contar el número de árboles es equivalente a contar el número de palabras de Lukasiewicz de longitud  $2n+1$  o el número de palabras de Dick de longitud  $2n$ .

**Proposición 4.11** El número total de árboles binarios diferentes con  $n$  nodos es:

$$C_n = \frac{1}{n+1} \binom{2n}{n} . \quad (4.39)$$

Donde  $C_n$  es conocido como el  $n$ -ésimo número de Catalan en honor al matemático belga Eugene Catalan.

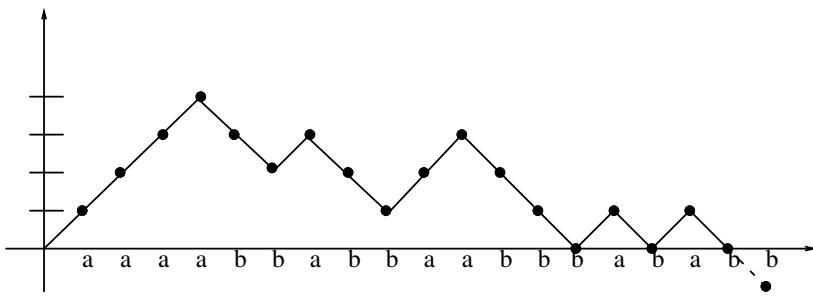


Figura 4.34: Camino de Catalan del árbol de la figura 4.33

**Demostración** Existe una demostración clásica basada en plantear la siguiente ecuación recurrente:

$$C_n = \begin{cases} 1 & n = 0 \\ \sum_{i=0}^{n-1} C_i C_{n-i-1} & n > 0 \end{cases} \quad (4.40)$$

Es decir, ponemos el nodo raíz y contamos el número de subárboles izquierdos y el número de subárboles derechos. El árbol izquierdo puede comenzar desde el vacío hasta el que tiene  $n-1$  nodos. Si un subárbol izquierdo tiene  $i$  nodos, entonces el derecho tiene  $n-i-1$  nodos. Por la regla del producto, tenemos  $C_i C_{n-i-1}$  árboles posibles cuyo subárbol izquierdo tiene  $i$  nodos. En lugar de resolver esta ecuación recurrente, nuestra demostración estará basada en una interpretación geométrica planteada por Kreher y Stinson [102].

Puesto que la correspondencia entre árboles  $m$ -rios y binarios es inyectiva (ver § 4.4.17 (Pág. 260)), contar el número de árboles de  $n$  nodos es equivalente a contar el número de árboles binarios de  $n$  nodos. Esto equivale a contar el número de palabras de Lukasiewicz de longitud  $2n+1$ , cual, por la proposición 4.10, es equivalente a contar el número de palabras de Dick de longitud  $2n$ .

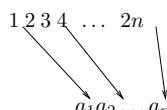
Existe una interpretación gráfica de una palabra de Dick (o de Lukasiewicz) denominada “camino de Catalan”. Dada una palabra  $w = w_1 w_2 \dots w_{2n+1} \in L$  Un camino de Catalan es formado por una secuencia de puntos en el plano:

$$P(w) = \left\{ p = (x, y) \mid \forall w_i \in w_1 w_2 \dots w_{2n+1} \in L, y = i \text{ y } \begin{cases} x = x_{i-1} + 1 & \text{si } w_i = a \\ x = x_{i-1} - 1 & \text{si } w_i = b \end{cases} \right\} \quad (4.41)$$

El camino de Catalan correspondiente al código del árbol de la figura 4.33 se muestra en la figura 4.34.

Según la proposición 4.9,  $|u|_a \geq |u|_b$  para todo  $u$  prefijo propio de  $w \in L$  y esto implica que todo camino de Catalan jamás traspasará el eje  $x$  antes del final de la palabra. Contar el número de palabras de Dick de longitud  $2n$  es equivalente a contar el número de caminos de Catalan generales que pueden formarse con  $n$  aes y  $n$  bes tal que nunca crucen el nivel  $x$ .

El número total de caminos posibles de Catalan, que crucen o no el eje  $x$ , está dado por el número de palabras que se pueden formar con  $n$  aes y  $n$  bes. Este conteo puede visualizarse mediante la siguiente figura:



es decir, el número de combinaciones de  $2n$  celdas enumeradas en  $n$  a's; la cantidad de b's son las celdas restantes. De este modo concluimos que  $\binom{2n}{n}$  es el total de caminos de Catalan posibles que puede conformar una palabra compuesta por el alfabeto  $\{a, b\}^*$ . De este modo, podemos plantear:

$$C_n = \binom{2n}{n} - \text{cantidad de caminos que cruzan el eje } x \quad (4.42)$$

Requerimos, pues, contar la cantidad de caminos que cruzan el eje  $x$  y que terminan en el eje  $x$ , correspondiente a una palabra  $w \notin D \mid |w|_a = |w|_b$ . Sea

$$P = \{(0, 0), (1, y_1), \dots, (2n-1, y_{2n-1}), (2n, 0)\},$$

un conjunto de puntos que representa un camino de Catalan que cruza el eje  $x$  correspondiente a una palabra  $w \notin D$ . Como ejemplo, veamos el camino de Catalan de la figura 4.35.

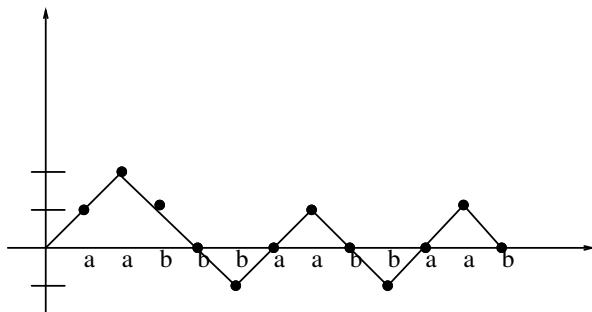


Figura 4.35: Camino de Catalan de la palabra aabbbaabbaab  $\notin D$ ,

Sea  $p' = (x', -1) \in P$  el primer punto de  $P$  por debajo del eje  $x$ , entonces podemos descomponer:

$$P(w) = P(uv) = P(u) \cup P(v) = \{(0, 0), (1, y_1), \dots, (x', -1)\} \cup (P(w) - P(u))$$

Para todo  $P(w) = uv$ ,  $w \notin D$ ,  $u$  corresponde al prefijo de  $w$  entre  $\{(0, 0), \dots, (x', -1)\}$ , es decir, el prefijo hasta el primer punto que esté por debajo del eje  $x$ . Definamos el camino siguiente:

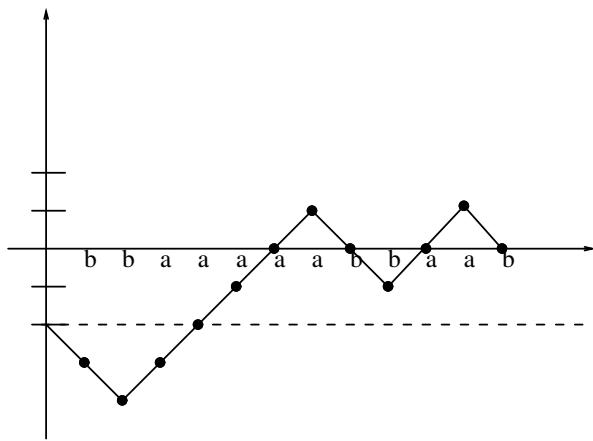
$$P'(w) = P'(\bar{u}) \cup (P(w) - P'(\bar{u})) ;$$

$\bar{u}$  es el complemento de  $u$ , es decir, la palabra correspondiente a cambiar todas las letras a por b y todas las letras b por a.  $P'(\bar{u})$  se define como:

$$P'(\bar{w}) = \{(-2 - x, y) \in P(\bar{u}) \mid 1 \leq x \leq x' - 1, (x, y) \in P(u)\}$$

Para el ejemplo anterior, aabbbaabbaab  $\notin D$  se partitiona entonces como bbaaa.aabbaab y tiene el camino de Catalan mostrado en la figura 4.36.

Geométricamente,  $P'(w) = P'(\bar{u}v) \mid u$ , que es el prefijo correspondiente al camino de Catalan  $P(uv)$  hasta el primer punto debajo del eje  $x$ , puede ser interpretado como  $\bar{u}$  desplazado 2 unidades por debajo del eje  $x$ .

Figura 4.36: Camino de Catalan de  $bbaaa.aabbaab \notin D$ 

La relación  $P(w) \longrightarrow P'(w) \mid \forall w \notin D$  es una función biyectiva. Geométricamente,  $\forall P(w_1), P(w_2)$ ,  $P(w_1) \neq P(w_2) \implies P'(w_1) \neq P'(w_2)$  tendrá una imagen única. Lo mismo sucede  $\forall P'(w_1), P'(w_2)$ ,  $P'(w_1) \neq P'(w_2) \implies P(w_1) \neq P(w_2)$ .

Retomando (4.42), contar el número de caminos que cruzan el eje  $x$  es equivalente a contar el número de caminos que parten desde  $(0, -2)$  y arriban a  $(0, 2n)$ . Cualquier camino de esta forma debe corresponder a  $n+1$  letras  $a$  y  $n-1$  letras  $b$ . Partiendo desde  $-2$  se requieren 2 letras  $a$  más que la letra  $b$ . Así pues, existen  $\binom{2n}{n+1}$  caminos que cruzan el eje  $x$ . Sustituimos en (4.42):

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1} \binom{2n}{n} \quad \blacksquare$$

Los números de Catalan tienen una importancia trascendental en la combinatoria, pues hay muchas situaciones de conteo identificadas mediante esta clase de número. Entre algunas de las aplicaciones tenemos:

- La cantidad de maneras en que un polígono regular convexo de  $n+2$  lados puede ser cortado en  $n$  triángulos. Por ejemplo, hay  $\frac{1}{4}\binom{6}{3} = 5$  diferentes maneras de cortar un pentágono en triángulos.
- El número de maneras en que  $n$  números pueden ser multiplicados asociativamente con  $2n$  paréntesis. Por ejemplo, hay  $\frac{1}{4}\binom{6}{3} = 5$ .

$$(n_1(n_2(n_3n_4))) (n_1((n_2n_3)n_4)) ((n_1n_2)(n_3n_4)) ((n_1(n_2n_3))n_4) (((n_1n_2)n_3)n_4)$$

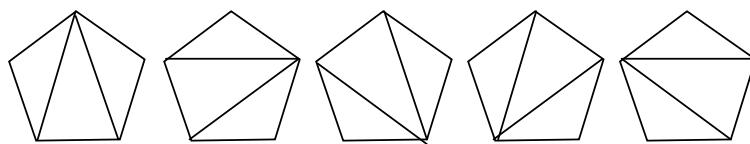


Figura 4.37: Diferentes maneras de cortar un pentágono en 3 triángulos

- El número de expresiones parentizadas con  $n$  paréntesis izquierdos y  $n$  paréntesis derechos que están balanceadas.
- El número de árboles binarios de  $n$  nodos.
- El número de árboles de  $n + 1$  nodos.
- El número de caminos de longitud  $2n$  en una malla  $n \times n$  que están por debajo de la diagonal. Por ejemplo, para una malla  $4 \times 4$ , hay  $\frac{1}{5} \binom{8}{4} = 14$  caminos diferentes que pasan debajo de la diagonal.

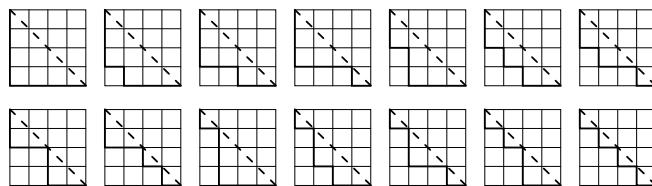


Figura 4.38: Diferentes caminos debajo de la diagonal en una malla  $4 \times 4$

- El número de palabras de Dick de longitud  $2n$  o el número de palabras Lukasiewicz de longitud  $2n + 1$ .
- El número de caminos en el plano, conformados por  $2n$  puntos, que parten desde  $(0, 0)$  y terminan en  $(2n, 0)$  que no cruzan el eje  $x$ .

#### 4.8.2 Almacenamiento de árboles binarios

Quizá una de las funcionalidades más apreciada de una cadena de Lukasiewicz es que nos ofrece una manera más compacta de secuenciar la topología de un árbol binario que las estudiadas en § 4.4.13 (Pág. 253). Así, podemos usar la secuencia para guardar el árbol en un medio de almacenamiento secundario o para transmitirlo por una red. Dado un árbol abarcador, asistidos del tipo `BitArray` § 2.1.5 (Pág. 53), podemos obtener su palabra de Lukasiewicz de la siguiente manera:

```
311 <Funciones de BinNode_Utils 243a>+≡ ◁278b 312a▷
 template <class Node> inline void tree_to_bits(Node * root, BitArray & array)
 {
 if (root == Node::NullPtr)
 {
 array.push(1);
 return;
 }
 array.push(0);
 tree_to_bits((Node*) LLINK(root), array);
 tree_to_bits((Node*) RLINK(root), array);
 }
 Uses BitArray 54a.
```

La palabra de Lukasiewicz resultante es guardada en array. No requerimos retornar su longitud porque es derivable de la cardinalidad del árbol ( $2n + 1$ ).

El arreglo de bits resultante ya está listo para los usos mencionados y otros adicionales; entre ellos, a partir de la correspondencia entre árboles y árboles binarios (§ 4.4.17 (Pág. 260)), el guardar árboles en general.

Para obtener el árbol a partir de la secuencia programamos la siguiente rutina inversa:

312a *<Funciones de BinNode\_Utils 243a>*+≡ ▷311 312b

```
template <class Node> static inline
Node * __bits_to_tree(BitArray & array, int & i)
{
 int bit = array.read_bit(i++);
 if (bit == 1) // ¿Es un nodo externo?
 return Node::NullPtr; // sí

 Node * p = new Node; // crear nodo interno actual
 LLINK(p) = (Node*) __bits_to_tree<Node>(array, i); // subárbol izq.
 RLINK(p) = (Node*) __bits_to_tree<Node>(array, i); // subárbol der.

 return p;
}
```

Uses BitArray 54a.

El parámetro array contiene la palabra de Lukasiewicz de un árbol abarcador. El parámetro i es el bit actual de procesamiento. El pase se hace por referencia para poder actualizarlo conforme se va leyendo la secuencia y debe inicializarse en el índice dentro de array en donde comienza la secuencia; esto permite guardar varios árboles en un BitArray, así como recuperarlos.

Como se ve, la rutina anterior es estática, pues maneja el parámetro i, por referencia, de índice actual de la cadena de bits. La rutina de interfaz se define de la siguiente manera:

312b *<Funciones de BinNode\_Utils 243a>*+≡ ▷312a 312c

```
template <class Node> inline Node * bits_to_tree(BitArray & array, int idx = 0)
{
 return __bits_to_tree <Node> (array, idx);
}
```

Uses BitArray 54a.

El arreglo sólo almacena la forma, mas no el contenido de los nodos. Para almacenar el contenido concatenamos a la palabra de Lukasiewicz cualquiera de los recorridos, en nuestro caso, el recorrido prefijo. Esto nos conduce a la siguiente rutina auxiliar, cuya función es guardar las claves en prefijo:

312c *<Funciones de BinNode\_Utils 243a>*+≡ ▷312b 313a

```
template <class Node, class Get_Key> inline
void save_tree_keys_in_prefix(Node * root, ofstream & output)
{
 if (root == Node::NullPtr)
 return;

 output << Get_Key () (root) << " ";

 save_tree_keys_in_prefix<Node,Get_Key>((Node*)LLINK(root), output);
}
```

```

 save_tree_keys_in_prefix<Node,Get_Key>((Node*)RLINK(root), output);
}

```

El rol de la clase Get\_Key es obtener el contenido del nodo que se desea almacenar.

La operación análoga, es decir, obtener las claves del archivo y colocarlas en los nodos se hace por medio de la siguiente rutina:

313a ⟨Funciones de BinNode\_Utils 243a⟩+≡ ◁312c 313b ▷

```

template <class Node, class Load_Key> inline
void load_tree_keys_in_prefix(Node * root, ifstream & input)
{
 if (root == Node::NullPtr)
 return;

 Load_Key () (root, input);

 load_tree_keys_in_prefix<Node,Load_Key>((Node*)LLINK(root), input);
 load_tree_keys_in_prefix<Node,Load_Key>((Node*)RLINK(root), input);
}

```

En este caso, la clase Load\_Key tiene como misión leer de input la clave y ponerla en el nodo. Puesto que existen diversas maneras de almacenar el contenido del nodo, la rutina no debe pretender un formato preestablecido.

Ahora tenemos todas las herramientas listas para instrumentar el almacenamiento de árboles binarios en memoria secundaria<sup>11</sup>:

313b ⟨Funciones de BinNode\_Utils 243a⟩+≡ ◁313a 315a ▷

```

template <class Node, class Get_Key> inline
void save_tree(Node * root, ofstream & output)
{
 BitArray prefix;
 tree_to_bits(root, prefix);
 prefix.save(output);
 save_tree_keys_in_prefix <Node, Get_Key> (root, output);
}

template <class Node, class Load_Key> inline
Node * load_tree(ifstream & input)
{
 BitArray prefix;
 prefix.load(input);
 Node * root = bits_to_tree <Node> (prefix);
 load_tree_keys_in_prefix <Node, Load_Key> (root, input);
 return root;
}

```

Uses BitArray 54a.

## 4.9 Árboles binarios de búsqueda

Supongamos que 1300 millones de personas están suscritas al servicio telefónico. Dado el nombre de un subscriptor, ¿cómo averiguar su número telefónico? Si los registros de aquellas 1300 millones de personas están ordenados alfabéticamente en un

---

<sup>11</sup> Así como también según sea el stream, su transmisión por red.

archivo, entonces la búsqueda binaria nos permitiría encontrar el número de celular de Zhang Shunniean Cheung Wu en a lo sumo  $\lceil \lg 13000000000 \rceil = 31$  intentos.

La búsqueda binaria exige que la secuencia de datos esté ordenada y esto es difícil de garantizar, pues es de esperar que a menudo aparezcan nuevos subscriptores y desaparezcan otros. Actualizar en línea una secuencia ordenada de números telefónicos sería muy inefficiente, pues la inserción y supresión en una secuencia ordenada es  $\mathcal{O}(n)$ .

Afortunadamente, si bien no podemos mantener sin costes importantes una secuencia ordenada, sí podemos simularla mediante un árbol binario que emule la búsqueda binaria y que nos ofrezca ordenamiento y rapidez para las operaciones de inserción, búsqueda y supresión.

**Definición 4.9 (Árbol binario de búsqueda)** Sea un árbol binario  $T = \langle L(T), \text{raiz}(T), R(T) \rangle$ , entonces,  $T$  es un árbol binario de búsqueda (ABB) si y sólo si:

$$T \in \text{ABB} \iff \begin{cases} \forall n_i \in L(T) \implies \text{KEY}(n_i) < \text{KEY}(\text{raiz}(T)) & \text{y} \\ \forall n_i \in R(T) \implies \text{KEY}(n_i) > \text{KEY}(\text{raiz}(T)) \end{cases}$$

Esta regla se conoce como la propiedad o condición de orden de un árbol binario de búsqueda.

Para disminuir la longitud del discurso, en muchos contextos utilizaremos “ABB” para denotar un “árbol binario de búsqueda”.

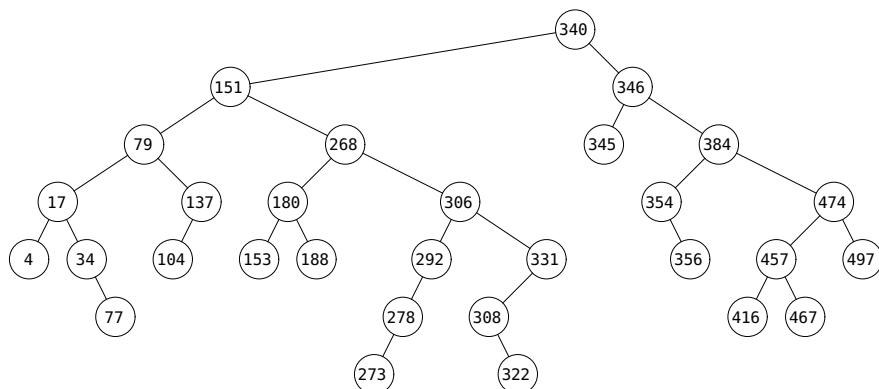


Figura 4.39: Un árbol binario de búsqueda

Dado un nodo cualquiera, todos los nodos en el subárbol izquierdo deben tener claves menores, mientras que todos los nodos en el subárbol derecho deben tener claves mayores. El árbol de la figura 4.39 satisface la definición.

El orden se destaca cuando se observa el recorrido infijo. Para el árbol de la figura 4.39, el recorrido infijo es: 4 17 34 77 79 104 137 151 153 180 188 268 273 278 292 306 308 322 331 340 345 346 354 356 384 416 457 467 474 497 . Este orden es deducible de la definición: visite los nodos a la izquierda, los cuales son menores a la raíz, luego visite la raíz y, finalmente, visite los nodos a la derecha, que son mayores que la raíz.

La propiedad de orden de un árbol binario de búsqueda permite codificarlo con el recorrido prefijo o sufijo. Es decir, podemos reconstruir completamente un ABB a partir de su recorrido prefijo.

Sea  $n_0, n_1, \dots, n_{|T|-1}$  el recorrido prefijo de un ABB  $T = < L(T), \text{raiz}(T), R(T) >$ . El recorrido prefijo nos proporciona directamente  $\text{raiz}(T) = n_0$  y  $\text{raiz}(L(T)) = n_1$ . La propiedad de orden nos garantiza que  $\text{raiz}(R(T))$  será el primer nodo en el recorrido prefijo que sea mayor o igual que  $\text{raiz}(T)$ . Así pues, aplicamos este razonamiento recursivo y obtenemos el algoritmo siguiente:

```
315a <Funciones de BinNode_Utils 243a>+≡ ◁313b 316a▷
 template <class Node> inline Node *
 preorder_to_binary_search_tree
 (DynArray<typename Node::key_type> & preorder,
 const int & l, const int & r)
 {
 if (l > r)
 return Node::NullPtr;
 Node * root = new Node(preorder[l]);
 if (l == r)
 return root;
 <Sea first_greater el primer nodo mayor que la raíz 315b>
 LLINK(root) = preorder_to_binary_search_tree<Node>(preorder, l + 1,
 first_greater - 1);
 RLINK(root) = preorder_to_binary_search_tree<Node>(preorder,
 first_greater, r);
 return root;
 }
```

Uses DynArray 34.

`right_root` es el primer nodo, de izquierda a derecha, que sea mayor que la raíz (`preorder[inf]`); este nodo es la raíz del subárbol derecho. Para localizarlo, recorremos linealmente el arreglo:

```
315b <Sea first_greater el primer nodo mayor que la raíz 315b>≡ (315a)
 int first_greater = l + 1;
 while ((first_greater <= r) and
 (preorder[first_greater] < preorder[1]))
 ++first_greater;
```

Este nodo partitiona el arreglo en dos partes. La primera comprendida entre  $l + 1$  y  $right\_root - 1$ , corresponde al recorrido prefijo del subárbol izquierdo. La segunda parte, comprendida entre `right_root` y `r`, corresponde al recorrido prefijo del subárbol derecho.

#### 4.9.1 Búsqueda en un ABB

Consideremos  $T \in \text{ABB}$  y una clave  $k$  a buscarse en el árbol. Para hacerlo inspeccionamos  $\text{KEY}(T)$ , si  $k = \text{KEY}(T)$ , entonces  $k$  ha sido encontrado. De lo contrario compara-

mos  $k < \text{KEY}(T)$ ; si el predicado es cierto, entonces buscamos en el subárbol izquierdo; de lo contrario, buscamos en el subárbol derecho. Si durante este proceso nos encontramos con un subárbol vacío, entonces concluimos que  $k$  no se encuentra dentro del árbol.

El siguiente algoritmo refleja este proceso:

316a *(Funciones de BinNode\_Utils 243a) +≡* ↳ 315a 316b ▷

```
template <class Node,
 class Compare = Aleph::less<typename Node::key_type> > inline Node *
searchInBinTree(Node * root, const typename Node::key_type & key)
{
 while (root != Node::NullPtr)
 if (Compare () (key, KEY(root))) // ¿en rama izquierda?
 root = static_cast<Node*>(LLINK(root)); // baje a la izquierda
 else if (Compare() (KEY(root), key)) // ¿en rama derecha?
 root = static_cast<Node*>(RLINK(root)); // baje a la derecha
 else
 break; // se encontró!

 return root;
}
```

Defines:  
*searchInBinTree*, used in chunk 320b.

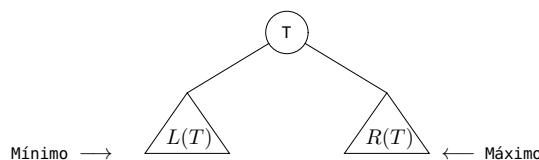
Esta primitiva busca en el árbol un nodo con valor de clave  $key$  y retorna su dirección si la clave se encontró o  $\text{Node}::\text{NullPtr}$  de lo contrario.

*searchInBinTree()* se utiliza en la mayoría de TAD fundamentados en ABB.

El desempeño de este algoritmo depende de cuan equilibrado esté el árbol. Si éste tiende a estar completo, entonces su altura tiende a  $\lceil \lg |T| \rceil$ . De este modo, si garantizamos que el árbol sea equilibrado, entonces la búsqueda en un árbol binario no requerirá más de  $\lceil \lg |T| \rceil$  comparaciones; exactamente la misma calidad de servicio de la búsqueda binaria. El truco es, pues, lograr que las inserciones y supresiones del árbol lo mantengan en equilibrio.

#### 4.9.1.1 Búsqueda del menor y del mayor elemento en un ABB

Sabemos que el recorrido infijo de un ABB muestra la secuencia ordenada. Por tanto, el menor y el mayor de todos los elementos del conjunto estarán dados por el primer y último elementos del recorrido infijo, los cuales se identifican pictóricamente, de manera general, en la siguiente figura:



Para  $T \in \text{ABB}$ , el mínimo se encuentra en el nodo más a la izquierda, mientras que el máximo en el más a la derecha. De este conocimiento se deducen los siguientes algoritmos:

316b *(Funciones de BinNode\_Utils 243a) +≡* ↳ 316a 317 ▷

```
template <class Node> inline Node * find_min(Node * root)
{
 while (LLINK(root) != Node::NullPtr)
```

```

 root = static_cast<Node*>(LLINK(root));
 return root;
 }
 template <class Node> inline Node * find_max(Node * root)
 {
 while (RLINK(root) != Node::NullPtr)
 root = static_cast<Node*>(RLINK(root));
 return root;
 }
}

```

Ambas primitivas reciben un nodo raíz de un ABB y retornan el mínimo y máximo respectivamente.

Si el ABB está equilibrado, entonces `find_min()` y `find_max()` tienen complejidad  $\mathcal{O}(\lg n)$ .

#### 4.9.1.2 Búsqueda del predecesor y sucesor

Dado  $T \in \text{ABB}$ , ¿cuál es el sucesor de  $\text{KEY}(T)$ ? Por la propiedad de orden de un ABB sabemos que este sucesor, si existe, se encuentra en el nodo más a la izquierda de la rama derecha y la condición de existencia es justamente que exista una rama derecha. De lo anterior se deduce el algoritmo siguiente:

317

*(Funciones de BinNode\_Utils 243a) +≡* ◀316b 318▶

```

template <class Node> inline Node * find_successor(Node * p, Node *& pp)
{
 pp = p;
 p = (Node*) RLINK(p);
 while (LLINK(p) != Node::NullPtr)
 {
 pp = p;
 p = (Node*) LLINK(p);
 }
 return p;
}

```

`find_successor()` retorna el nodo sucesor de  $p$  y su padre en el parámetro  $pp$ . La llamada debe hacerse sobre  $p$  y su padre. `find_successor()` no requiere un operador de comparación, pues la topología del árbol proporciona el resultado.

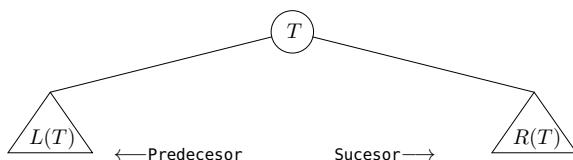


Figura 4.40: Ubicaciones generales del nodo predecesor y sucesor de la raíz de un árbol

La búsqueda del predecesor es análoga.

#### 4.9.1.3 Búsquedas especiales sobre un ABB

En algunas no tan excepcionales ocasiones se requiere “encontrar” un nodo que, en lugar de corresponderse con la clave, de alguna forma guarde otra relación con ella. En esta

situación encajan dos tipos de búsqueda: la del padre de una clave perteneciente al árbol y la del que sería el padre de una clave no perteneciente.

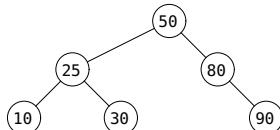
La primera búsqueda se expresa del siguiente modo:

318 *(Funciones de BinNode\_Utils 243a) +≡* «317 319a»

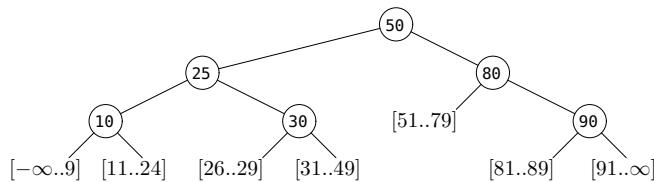
```
template <class Node,
 class Compare = Aleph::less<typename Node::key_type> > inline
Node * search_parent(Node * root, const typename Node::key_type & key,
 Node *& parent)
{
 while (true)
 if (Compare () (key, KEY(root)))
 {
 if (LLINK(root) == Node::NullPtr)
 return root;
 parent = root;
 root = static_cast<Node*>(LLINK(root));
 }
 else if (Compare () (KEY(root), key))
 {
 if (RLINK(root) == Node::NullPtr)
 return root;
 parent = root;
 root = static_cast<Node*>(RLINK(root));
 }
 else
 return root;
}
```

`search_parent()` busca la clave `key` en el ABB cuya raíz es `root`. La rutina retorna el nodo que contiene a `key` y su nodo padre se guarda en el parámetro por referencia `parent`, el cual debe inicializarse en el padre de la raíz.

Consideremos el siguiente ABB de ejemplo:



El árbol contiene las claves 10, 25, 30, 50, 80, 90 entre un conjunto infinito de posibles valores. Si el árbol creciese “naturalmente”, es decir, sólo por sus puntas, ¿en dónde colocar nuevas claves? La respuesta se evidencia en el árbol siguiente:



El único “*espacio*”<sup>12</sup> por donde puede crecer “naturalmente” un ABB es a través de sus

<sup>12</sup>En este párrafo las nociones de “*espacio*”, “*natural*” y “*físico*” se enuncian en el sentido de la ciencia física moderna. Se usan comillas y énfasis itálico porque tales ideas que si bien también son abstractas en nuestro mundo cotidiano, son aún más abstractas en el discurso de la programación de computadores, máxime cuando el concepto de árbol binario es, como tal, pura abstracción.

nodos externos. Dicho de otro modo, que el vacío que representa un nodo externo sea ocupado “físicamente” por un nuevo nodo interno.

Luego de expresado lo anterior, adquiere sentido pensar una búsqueda de un rango que no está en el árbol. O sea, buscar los puntos por donde puede crecer un ABB. Tal búsqueda la instrumenta la función:

319a *<Funciones de BinNode\_Utils 243a>* +≡ △318 321a ▷  
 template <class Node,  
 class Compare = Aleph::less<typename Node::key\_type> > inline Node \*  
 search\_rank\_parent(Node \* root, const typename Node::key\_type & key)

Cuya implementación puede observarse en la biblioteca.

La rutina busca la clave key en un árbol con raíz root y retorna el nodo que sería padre de una clave key.

#### 4.9.2 El TAD BinTree<Key>

Antes de introducir los algoritmos de inserción y supresión, es conveniente introducir nuestro TAD de base que modeliza un árbol binario de búsqueda. Tal TAD se especifica en el archivo *<tpl\_binTree.H 319b>* cuya estructura es la siguiente:

319b *<tpl\_binTree.H 319b>* ≡  
 template <template <typename> class NodeType, typename Key, class Compare>  
 class GenBinTree  
 {  
*<Definiciones públicas BinTree<Key> 320a>*  
*<miembros dato de BinTree<Key> 319c>*  
*<Métodos de BinTree<Key> 320b>*  
 };  
 template <typename Key, class Compare = Aleph::less<Key> >  
 class BinTree : public GenBinTree<BinNode, Key, Compare> {};

Uses BinNode 240.

*<tpl\_binTree.H 319b>* define tres clases:

- **GenBinTree<NodeType, Key, Compare>**: clase genérica que implanta el problema fundamental de estructuras de datos<sup>13</sup> mediante un ABB. Esta clase sólo está destinada a implantar y no es de uso del cliente.
- **BinTree<Key>**: clase parametrizada que implanta el problema fundamental a través de un árbol binario de búsqueda de nodos binarios con claves de tipo Key. Esta clase es implantada directamente por derivación pública de GenBinTree. Así pues, su interfaz es exactamente la misma que GenBinTree.
- **BinTreeVtl<Key>**: equivalente al **BinTree<Key>**, salvo que los nodos poseen destructores virtuales.

Para facilitar los algoritmos, todos los tipos abstractos basados en árboles binarios de búsqueda utilizarán un nodo cabecera:

319c *<miembros dato de BinTree<Key> 319c>* ≡ (319b)  
 Node headNode;  
 Node \* head;  
 Node \*& root;

<sup>13</sup>Recordar § 1.3 (Pág. 17).

`headNode` representa el nodo cabecera, `head` es la dirección de la cabecera y `root` es una referencia al lazo derecho de `head`. En otros términos, `RLINK(head) == root`.

Las primitivas de `GenBinTree` manipulan nodos binarios cuya especificación se exporta al usuario del siguiente modo:

320a  $\langle \text{Definiciones públicas } \text{BinTree} < \text{Key} \rangle \quad 320a \equiv \quad (319b)$

```
typedef NodeType<Key> Node;
```

El tipo será `BinTree<Key>::Node` para nodos comunes o `BinTreeVtl<Key>::Node` para nodos con destructores virtuales.

La búsqueda se remite a invocar a `searchInBinTree()`:

320b  $\langle \text{Métodos de } \text{BinTree} < \text{Key} \rangle \quad 320b \equiv \quad (319b) \quad 320c \triangleright$

```
Node * search(const Key & key)
{
 return searchInBinTree <Node, Compare>(root, key);
}
```

Uses `searchInBinTree` 316a.

Un observador esencial, sobre todo para poder liberar la memoria ocupada por el árbol binario, es la consulta de la raíz:

320c  $\langle \text{Métodos de } \text{BinTree} < \text{Key} \rangle \quad 320b \equiv \quad (319b) \quad \triangleleft 320b \quad 321b \triangleright$

```
Node*& getRoot() { return root; }
```

#### 4.9.3 Inserción en un ABB

La inserción obedece al “crecimiento natural de un árbol”, es decir, por las puntas u hojas. Dado un nodo a insertar `p`, debemos encontrar un nodo externo para sustituirlo por `p` de manera tal que no se viole la propiedad de orden de un ABB. Asumiendo que las claves no se repiten, condición que desde ahora imponemos al TAD `BinTree<Key>`, existe un sólo nodo externo para sustituir por `p`. Por ejemplo, consideremos insertar un nodo que contenga valor de clave 27 tal como se muestra en la figura 4.41. El “ex-nodo-externo”, hijo

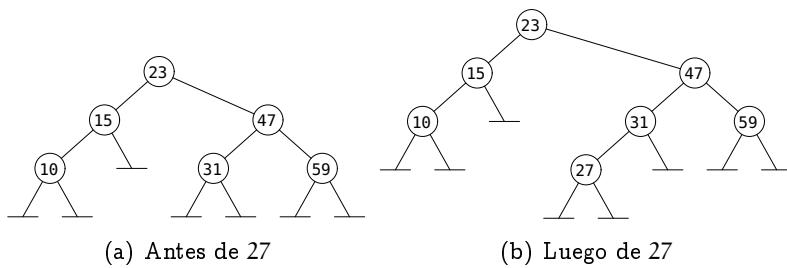


Figura 4.41: Un ejemplo de inserción en árbol binario

izquierdo del 31, fue sustituido por uno interno con valor 27, el cual, a través de sus dos nodos externos, abre dos “brechas” en el árbol. La primera está dada por el lazo izquierdo del 27 y puede albergar claves entre [24..26], mientras que la segunda brecha la representa el lazo derecho y podría albergar claves entre [28..30].

El ejercicio anterior nos indica que para efectuar una inserción es menester buscar el nodo externo a substituir. Ahora bien, a efectos de “enlazar” el nuevo nodo se requiere obtener un puntero al que sería el padre del nodo a insertar -en el ejemplo, un

puntero al nodo con clave 31-. Si bien esta acción puede instrumentarse mediante la rutina `search_rank_parent()` mencionada en § 4.9.1.3 (Pág. 317), es preferible, a efectos de la simplicidad, “memorizar” este puntero realizando una búsqueda recursiva y luego insertando el nuevo nodo. Este es el enfoque del siguiente algoritmo:

321a *(Funciones de BinNode\_Utils 243a) +≡* ◁319a 322▷

```
template <class Node,
 class Compare = Aleph::less<typename Node::key_type> > inline
Node * insert_in_binary_search_tree(Node *& root, Node * p)
{
 if (root == Node::NullPtr)
 return root = p;

 if (Compare () (KEY(p), KEY(root))) // ¿p < root?
 return insert_in_binary_search_tree<Node, Compare>((Node*&) LLINK(root), p);
 else if (Compare () (KEY(root), KEY(p))) // ¿p > root?
 return insert_in_binary_search_tree<Node, Compare>((Node*&) RLINK(root), p);

 return Node::NullPtr; // clave repetida ==> no hay inserción
}
```

Defines:

`insert_in_binary_search_tree`, used in chunks 321b and 327.

La rutina inserta el nodo `p` en el ABB con raíz `root` y retorna la raíz del árbol binario resultante en caso de que la inserción tenga éxito, es decir, si la clave contenida en `p` no se encuentra en el árbol. De lo contrario, puesto que no se permiten claves duplicadas, se retorna `Node::NullPtr`.

Notemos que el parámetro `root` se pasa por referencia, pues en el caso base de la recursión se modifica la raíz.

Mediante la anterior rutina genérica, la implantación del método `insert()` es mera cuestión de forma:

321b *(Métodos de BinTree<Key> 320b) +≡* (319b) ◁320c 323▷

```
Node * insert(Node *p)
{
 return insert_in_binary_search_tree<Node, Compare>(root, p);
}
```

Uses `insert_in_binary_search_tree` 321a.

#### 4.9.4 Partición de un ABB por clave (split)

En esta operación, un árbol  $T$  con secuencia infija genérica  $S_i = \langle k_1, k_2, k_3, \dots, k_n \rangle$  se partitiona según una clave  $k_x$  en dos árboles  $T_{<} y T_{>}$  tal que  $S_{i_{<}} = \langle k_1, k_2, k_3, \dots \rangle | \forall k_i \in T_{<} \text{, } k_i < k_x \text{ y } S_{i_{>}} = \langle \dots, k_n \rangle | \forall k_i \in T_{>} \text{, } k_i > k_x$ .

La primitiva que nos efectúa la partición se denomina `split_key_rec(T, k_x, T_{<}, T_{>})`; donde  $T$  es el árbol a partitionar,  $k_x$  es la clave de partición y  $T_{<}$  y  $T_{>}$  son los árboles resultantes de la partición.

El principio recursivo de la partición es simple y se pictorializa en el diagrama de la figura 4.42.

Llamemos  $k_x$  a la clave de partición y supongamos  $k_x > \text{KEY}(T)$ . En este caso se efectúa una llamada recursiva `split_key_rec(R(T), k_x, T'_{<}, T'_{>})` sobre  $R(T)$ , la cual

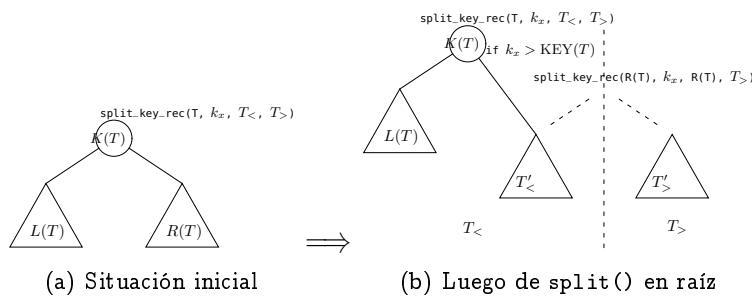


Figura 4.42: Esquema general de la partición por clave

arroja dos árboles  $T'_<$  y  $T'_>$  producto de particionar  $R(T)$  con  $k_x$ . El resultado definitivo es  $T'_< = < L(T), T, L(T)'>$ , donde  $L(T)' = T'_<$ . La otra partición,  $T'_>$ , es directamente el árbol  $T'_> = T'_>$ .

Si  $k_x < \text{KEY}(T)$ , entonces el algoritmo es simétricamente idéntico.

El caso base de la recursión es la partición sobre el árbol vacío, la cual arroja dos árboles vacíos.

Con lo anterior, el algoritmo resultante debería ser comprensible sin dificultades:

```

322 <Funciones de BinNode_Utils 243a>+≡ <321a 324>
 # define SPLIT split_key_rec<Node, Compare>
 template <class Node,
 class Compare = Aleph::less<typename Node::key_type> > inline
 bool split_key_rec(Node * root, const typename Node::key_type & key,
 Node *& ts, Node *& tg)
 {
 if (root == Node::NullPtr)
 { // key no se encuentra en árbol ==> split tendrá éxito
 ts = tg = Node::NullPtr;
 return true;
 }
 if (Compare() (key, KEY(root))) // ¿key < KEY(root)?
 if (SPLIT((Node*&) LLINK(root), key, ts, (Node*&) LLINK(root)))
 {
 tg = root;
 return true;
 }
 else
 return false;

 if (Compare() (KEY(root), key)) // ¿key > KEY(root)?
 if (SPLIT((Node*&) RLINK(root), key, (Node*&) RLINK(root), tg))
 {
 ts = root;
 return true;
 }
 else
 return false;

 return false; // clave existe en árbol ==> se deja intacto
 }

```

```
}
```

Defines:

`split_key_rec`, used in chunks 323 and 326b.

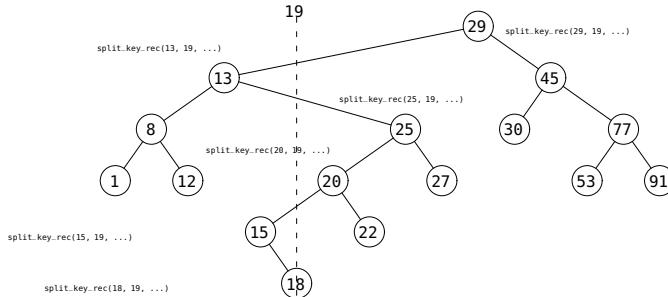


Figura 4.43: Trazado de una línea de división

`split_key_rec()` tiene cuatro parámetros, dos de entrada y dos de salida. Los de entrada son la raíz del árbol a particionar `root` y la clave de partición `key`. Los parámetros de salida son dos árboles `ts` y `tg`. Después de la llamada, `ts` contiene las claves menores que `key` y `tg` contiene las mayores o iguales.

Si la clave no se encuentra en el árbol, entonces `split_key_rec()` retorna `true` para indicar que el árbol fue particionado. De lo contrario se deja el árbol intacto y se retorna `false`.

Pictóricamente, la partición puede interpretarse como el trazado de una línea divisoria según la clave de partición. La figura 4.43 ilustra tal línea con clave 19 y se indican las llamadas recursivas de la partición. Desde el nodo raíz 29 se determina que debemos particionar la rama izquierda de raíz 13, pues 19 es menor que 29. `split_key_rec(13, 19, ts, tg)` retorna la partición del subárbol con raíz 13. El parámetro `ts` es el árbol  $T_{<}$  resultado de `split_key_rec(29, 19, ts, tg)`, mientras que el árbol  $T_{>}$  será  $\langle tg, R(29) \rangle$ .

Con la rutina anterior, ya estamos listos para escribir la versión de la partición del TAD `BinTree<Key>`:

323

(Métodos de `BinTree<Key>` 320b) +≡ (319b) ▷321b 326a▷

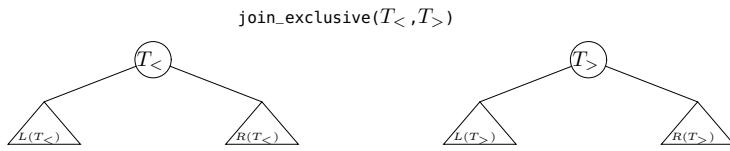
```
bool split(const Key & key, GenBinTree & l, GenBinTree & r)
{
 return split_key_rec<Node, Compare>(root, key, l.root, r.root);
}
```

Uses `split_key_rec` 322.

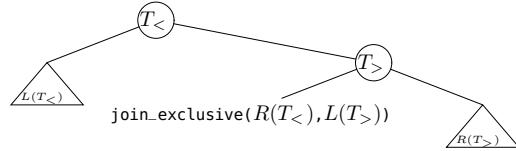
#### 4.9.5 Unión exclusiva de ABB (join exclusivo)

Consideremos dos árboles  $T_{<}$  y  $T_{>}$  resultantes de `split_key_rec()` y la operación inversa: "pegar"  $T_{<}$  y  $T_{>}$  en un árbol  $T$ . Recordemos que en este caso,  $T_{<}$  y  $T_{>}$  son excluyentes. Llámemos a esta operación `join_exclusive( $T_{<}$ ,  $T_{>}$ )`, la cual retorna el árbol correspondiente de unir  $T_{<}$  con  $T_{>}$ . Calificamos la operación de "exclusiva" porque sabemos que los rangos de claves no se solapan y que  $T_{<} \cap T_{>} = \emptyset$ , lo cual es un conocimiento muy valioso, pues nos permite hacer un algoritmo particular para este caso con mejor desempeño que el de la unión general si los intervalos se solapan o los elementos se comparten.

La siguiente figura ilustra los componentes generales de  $T_{<}$  y  $T_{>}$  que requerimos identificar para diseñar `join_exclusive( $T_{<}$ ,  $T_{>}$ )`:



Y la solución se pictoriza del siguiente modo:



El caso base del algoritmo es que alguno de los dos árboles sea vacío.

Lo anterior nos lleva al siguiente algoritmo:

```
324 <Funciones de BinNode_Utils 243a> +≡ ◁322 325 ▷
template <class Node> inline Node * join_exclusive(Node *& ts, Node *& tg)
{
 if (ts == Node::NullPtr)
 return tg;

 if (tg == Node::NullPtr)
 return ts;

 LLINK(tg) = join_exclusive(RLINK(ts), LLINK(tg));

 RLINK(ts) = tg;
 Node * ret_val = ts;
 ts = tg = Node::NullPtr; // deben quedar vacíos después del join

 return ret_val;
}
```

Defines:

`join_exclusive`, used in chunks 325 and 513.

Esta rutina nos será de suma utilidad para simplificar considerablemente el algoritmo de eliminación de un ABB.

#### 4.9.6 Eliminación en un ABB

La eliminación por clave exhibe dos fases: buscar el nodo que contenga la clave y, luego, quitarlo del árbol. Pero hay una dificultad “topológica” para “quitar” el nodo. Consideremos un nodo p a “quitarse” de un ABB y el diagrama de la figura 4.44.

Si literalmente quitamos p, entonces debemos hacer “algo” con los tres arcos que quedan “sueltos” si desaparece p: q → p, p → L(p) y p → R(p). Adicionalmente, para poder modificar correctamente a q debemos distinguir exactamente si p es hijo izquierdo o derecho.

Se conocen dos maneras de tratar con la dificultad anterior:

1. Substituimos p por su nodo predecesor en L(p), o por su sucesor en R(p).

Hay dos puntos fundamentales que debemos notar. Primero, tanto el predecesor como el sucesor de p siempre son incompletos. En el caso del predecesor, éste está

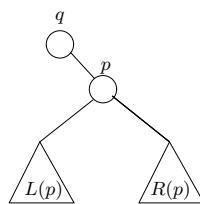


Figura 4.44: Situación general de eliminación del nodo p

dado por el nodo más a la izquierda de  $L(p)$ , el cual tiene que ser incompleto, pues si no contradiría el hecho de que el predecesor sea el más a la izquierda. El razonamiento simétrico ocurre con el sucesor.

Puesto que el nodo a sustituir está incompleto, tenemos suficiente espacio para enlazar las tres ramas si intercambiamos el nodo substituto por p, pues, por ejemplo en el predecesor, su rama izquierda faltante se substituye por  $L(p)$ .

La segunda cuestión a notar es que el predecesor y el sucesor son los dos únicos posibles nodos incompletos a intercambiar, pues de lo contrario se violaría la propiedad de orden de un ABB.

Un algoritmo “elegante” basado en este principio es bastante laborioso, pues para efectuar correctamente los intercambios se requieren mantener punteros a p, a su parent, al predecesor y al parent del predecesor. Por añadidura, si el predecesor no es hoja, hay que efectuar un paso adicional consistente en “corto-circuitar” p en su nueva posición con el árbol que éste contenga.

La “elegancia” de la que hablamos en el párrafo anterior consiste en no intercambiar los contenidos de los nodos, pues se afectaría la coherencia de eventuales estructuras de datos que apunten a ellos.

Esta forma está en desuso, pues la otra manera, que explicaremos inmediatamente, no sólo es mucho más simple, sino más eficiente. De todos modos, en § 6.4.1.2 (Pág. 479) se muestra esta técnica.

2. El árbol cuya raíz es p se sustituye por el árbol resultante de la unión exclusiva de sus ramas. En función de esto se desprende el siguiente algoritmo:

```

325 <Funciones de BinNode_Utils 243a>+≡ ◁324 326b▷
 # define REMOVE remove_from_search_binary_tree<Node, Compare>

 template <class Node,
 class Compare = Aleph::less<typename Node::key_type> > inline
 Node * remove_from_search_binary_tree(Node *& root,
 const typename Node::key_type & key)
 {
 if (root == Node::NullPtr)
 return Node::NullPtr;

 if (Compare () (key, KEY(root))) // ¿key < KEY(root)?
 return REMOVE((Node*&) LLINK(root), key);
 else if (Compare () (KEY(root), key)) // ¿key > KEY(root)?

```

```

 return REMOVE((Node*&) RLINK(root), key);

 Node * ret_val = root; // respaldar root que se va a borrar
 root = join_exclusive((Node*&) LLINK(root), (Node*&) RLINK(root));
 return ret_val;
}

Defines:
remove_from_search_binary_tree, used in chunks 326a and 327.
Uses join_exclusive 324.

```

El parámetro `root` es la raíz del ABB, el cual es por referencia, de forma tal que se modifique la raíz cuando se encuentre la clave. El segundo parámetro es la clave misma a eliminar.

Al igual que con la inserción, el método de eliminación de `BinTree<Key>` deviene muy simple:

326a *(Métodos de BinTree<Key> 320b) +≡* (319b) ◁323 328 ▷  
`Node * remove(const Key & key)`  
`{`  
 `return remove_from_search_binary_tree<Node, Compare>(root, key);`  
`}`

Uses `remove_from_search_binary_tree` 325.

#### 4.9.7 Inserción en raíz de un ABB

En el algoritmo de inserción desarrollado en § 4.9.3 (Pág. 320), el árbol crece por los nodos externos. Algunas veces es preferible que el nuevo nodo devenga raíz del árbol.

La técnica anterior favorece la localidad de referencia en el sentido de que los nodos cercanos a la raíz son los recientemente insertados.

Insertar un nuevo nodo como raíz es muy fácil luego de que se dispone de la partición `split_key_rec()` desarrollada en § 4.9.4 (Pág. 321). Todo lo que hay que hacer es particionar según la clave de inserción y luego atar a esta clave los árboles resultantes de la partición:

326b *(Funciones de BinNode\_Utils 243a) +≡* ◁325 327 ▷  
`template <class Node,`  
 `class Compare = Aleph::less<typename Node::key_type> > inline`  
`Node * insert_root(Node *& root, Node * p)`  
`{`  
 `Node * l = Node::NullPtr, * r = Node::NullPtr;`  
  
 `if (not split_key_rec<Node, Compare>(root, KEY(p), l, r))`  
 `return Node::NullPtr;`  
  
 `LLINK(p) = l;`  
 `RLINK(p) = r;`  
 `root = p;`  
  
 `return root;`  
`}`

Defines:

insert\_root, used in chunk 327.

Uses split\_key\_rec 322.

La rutina retorna `Node::NullPtr` si el árbol contiene un nodo con clave igual al nodo de inserción p.

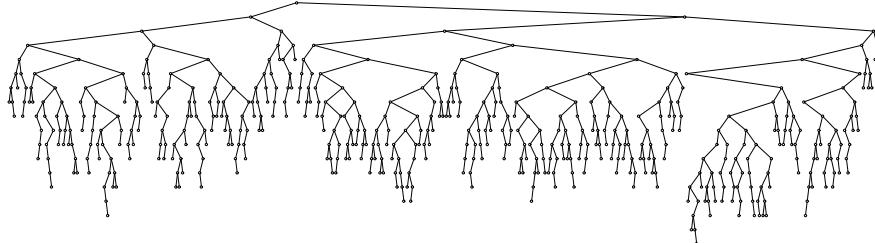


Figura 4.45: Árbol resultante de 512 inserciones aleatorias en la raíz

#### 4.9.8 Unión de ABB (join)

Consideremos  $T_1, T_2 \in \text{ABB}$  y la operación de unión  $T_1 \cup T_2$ . En nuestra interfaz, la unión se denominará  $\text{join}(T_1, T_2, d)$ , la cual retorna un ABB como producto de unir las claves de  $T_1$  con las de  $T_2$ . Puesto que hemos acordado no tener claves repetidas, las eventuales repitencias entre  $T_1$  y  $T_2$  se guardarán en un ABB auxiliar llamado d, el cual es un parámetro por referencia a  $\text{join}()$ . Debemos prever esto porque los árboles a unir pueden haberse construido independientemente.

Una primera forma de implantar el  $\text{join}()$  es recorrer en prefijo un árbol, por ejemplo,  $T_2$ , e ir insertando sus claves en el otro árbol  $T_1$ .

Otra manera, quizás más consona con la recursión inherente a un árbol binario, se ilustra en la figura 4.46.

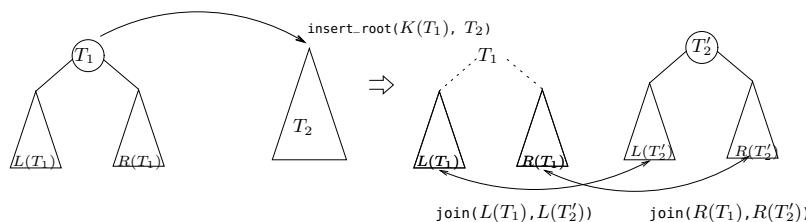


Figura 4.46: Esquema general de unión recursiva de dos árboles binarios

Tomamos la raíz de uno de los árboles y la insertamos como raíz del otro. En el caso de la figura tomamos la raíz del árbol  $T_1$  y la insertamos en  $T_2$  para obtener  $T_2'$ . Luego de insertar  $\text{root}(T_1)$ , las ramas sueltas  $L(T_1)$  y  $R(T_1)$  se unen recursivamente con las ramas izquierda y derecha de  $T_2'$ , o sea,  $L(T_2') = \text{join}(L(T_1), L(T_2'))$  y  $R(T_2') = \text{join}(R(T_1), R(T_2'))$ .

Con el principio anterior claro, podemos diseñar un algoritmo:

327 *(Funciones de BinNode\_Utils 243a) +≡* ◀326b 344▶  
 template <class Node,  
 class Compare = Aleph::less<typename Node::key\_type> > inline  
 Node \* join(Node \* t1, Node \* t2, Node \*& dup)

```

{
 if (t1 == Node::NullPtr)
 return t2;

 if (t2 == Node::NullPtr)
 return t1;

 Node * l = (Node*) LLINK(t1); // respaldos ramas de t1
 Node * r = (Node*) RLINK(t1);

 t1->reset();

 // mientras la clave raíz de t1 esté contenida en t2
 while (insert_root<Node, Compare>(t2, t1) == Node::NullPtr)
 {
 // si ==> sáquelo de t1 e insértelo en dup
 Node * p = remove_from_search_binary_tree(t1, KEY(t1));
 insert_in_binary_search_tree<Node, Compare>(dup, t1);
 }
 LLINK(t2) = join<Node, Compare>(l, (Node*) LLINK(t2), dup);
 RLINK(t2) = join<Node, Compare>(r, (Node*) RLINK(t2), dup);

 return t2;
}

```

Uses `insert_in_binary_search_tree` 321a, `insert_root` 326b, and `remove_from_search_binary_tree` 325.

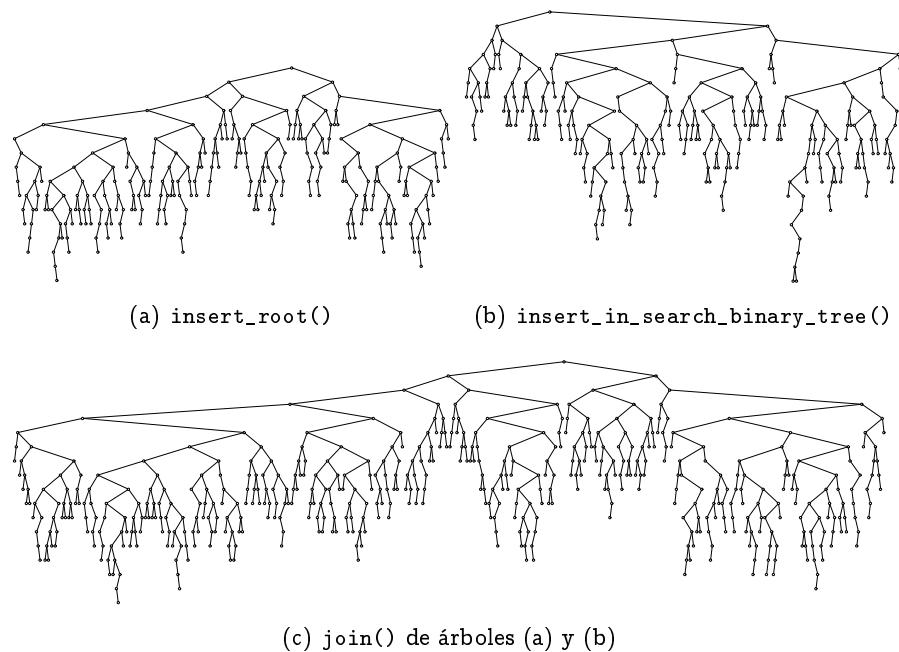


Figura 4.47: Ejemplo de `join()` de árboles binarios

Ahora incorporamos esta versión al método correspondiente en `BinTree<Key>`:

```

void join(GenBinTree & tree, GenBinTree & dup)
{

```

```

root = Aleph::join<Node, Compare> (root, tree.root, dup);
tree.root = Node::NullPtr;
}

```

#### 4.9.9 Análisis de los árboles binarios de búsqueda

Todas las operaciones estudiadas sobre un ABB deben llevar a cabo búsquedas; su tiempo de ejecución depende por tanto del tiempo de búsqueda, el cual está acotado por la altura del árbol. Consecuentemente, lo trascendental en el análisis es conocer cuál será el nivel promedio de un nodo.

En lo que sigue de nuestro análisis supondremos que el orden de inserción de las claves es aleatorio, es decir, uniformemente distribuido. Así, dado un árbol  $T$ , hay  $|T|!$  órdenes diferentes de inserción que pueden producir  $C_{|T|} = \frac{1}{|T|+1} \binom{2|T|}{|T|}$  árboles binarios diferentes. Ahora bien, es posible demostrar, preferiblemente por inducción, que  $\forall |T|$ ,  $C_{|T|} < |T|!$ . Lo cual implica que algunos árboles se repetirán para permutaciones de inserción diferentes.

**Proposición 4.12** Sea  $T$  un árbol binario de búsqueda de  $n$  nodos construido aleatoriamente según una secuencia de inserción aleatoria. No hay supresiones. Sean:

- $S_n$  el número de nodos visitados durante una búsqueda exitosa, y
- $U_n$  el número de nodos visitados durante una búsqueda fallida.

Entonces:

$$\begin{aligned}\overline{S_n} &= 2\left(1 + \frac{1}{n}\right)H_n - 3 \approx 1.386 \lg n = \mathcal{O}(\lg n) \text{ y} \\ \overline{U_n} &= 2H_{n+1} - 2 \approx 1.386 \lg(n+1) = \mathcal{O}(\lg n)\end{aligned}$$

Donde  $\overline{S_n}$  y  $\overline{U_n}$  denotan los promedios de  $S_n$  y  $U_n$  respectivamente, mientras que  $H_n$  es el  $n$ -ésimo número armónico<sup>14</sup>.

**Demostración** El número de nodos visitados en una búsqueda exitosa es la longitud del camino desde  $\text{raiz}(T)$  hasta el nodo encontrado. Si  $n_i$  es el nodo encontrado, entonces  $|\mathcal{C}_{\text{raiz}(T), n_i}| = \text{nivel}(n_i) + 1$ , pues el nivel comienza desde cero. Por definición de promedio, tenemos:

$$\begin{aligned}\overline{S_n} &= \frac{1}{n} \sum_{\forall n_i \in T} (\text{nivel}(n_i) + 1) \\ &= \frac{1}{n} \sum_{\forall n_i \in T} \text{nivel}(n_i) + 1 \\ &= \frac{1}{n} \text{IPL}(T) + 1\end{aligned}\tag{4.43}$$

Sustituimos (4.24) en (4.43) (por la proposición 4.4):

$$\overline{S_n} = \frac{\text{EPL}(T) - 2n}{n} + 1\tag{4.44}$$

---

<sup>14</sup> $H_n = \sum_{i=1}^n \frac{1}{i}$ .

Ahora planteamos una ecuación similar para una búsqueda infructuosa. Puesto que el nodo no se encuentra en el árbol, la búsqueda descenderá hasta un nodo externo. El número de nodos visitados en una búsqueda infructuosa es, pues, equivalente a la longitud del camino desde la raíz hasta el nodo externo. Podemos entonces plantear el promedio de longitudes entre todos los caminos desde la raíz hasta un nodo externo. Esto es:

$$\begin{aligned}\overline{U_n} &= \frac{1}{n+1} \sum_{\substack{\forall n_x \\ n_x \text{ nodo externo}}} \text{nivel}(n_x) \\ &= \frac{EPL(T)}{n+1} \implies \\ EPL(T) &= (n+1)\overline{U_n}\end{aligned}\quad (4.45)$$

Sustituimos (4.45) en (4.44):

$$\overline{S_n} = \frac{(n+1)\overline{U_n} - 2n}{n} + 1 \quad (4.46)$$

La ecuación (4.46) nos plantea las dos incógnitas que intentamos encontrar. Para poder resolverla requerimos una segunda ecuación, independiente de (4.46), que también nos relacione  $\overline{S_n}$  y  $\overline{U_n}$ .

Supongamos que la permutación de inserción es  $n_0 n_1 n_2 \dots n_{n-2} n_{n-1}$ . Notemos que la cantidad de nodos que se visitan cuando se busca exitosamente el nodo  $n_i$  es equivalente a uno más el número de nodos que se visitaron durante la búsqueda infructuosa que se realizó durante la inserción de  $n_i$ . Esto sólo es correcto si se asume que no hay supresiones. La ausencia de supresiones garantiza que los nodos nunca cambian de lugar en el árbol. Podemos entonces decir que:

$$\overline{S_i} = \overline{U_{i-1}} + 1 \quad (4.47)$$

Ahora, por la misma definición de promedio, podemos plantear una nueva ecuación:

$$\begin{aligned}\overline{S_n} &= \frac{(\overline{U_0} + 1) + (\overline{U_1} + 1) + (\overline{U_2} + 1) + \dots + (\overline{U_{n-1}} + 1)}{n} \\ &= 1 + \frac{1}{n} \sum_{i=0}^{n-1} \overline{U_i}\end{aligned}\quad (4.48)$$

Igualamos (4.48) con (4.46):

$$\begin{aligned}\frac{(n+1)\overline{U_n} - 2n}{n} + 1 &= 1 + \frac{1}{n} \sum_{i=0}^{n-1} \overline{U_i} \implies \\ (n+1)\overline{U_n} &= \sum_{i=0}^{n-1} \overline{U_i} + 2n\end{aligned}\quad (4.49)$$

Evaluamos (4.49) para  $n-1$  y obtenemos:

$$n\overline{U_{n-1}} = \sum_{i=0}^{n-2} \overline{U_i} + 2(n-1) \quad (4.50)$$

Restamos (4.49) menos (4.50):

$$(n+1)\bar{U}_n - n\bar{U}_{n-1} = \bar{U}_{n-1} + 2 \implies \bar{U}_n = \bar{U}_{n-1} + \frac{2}{n+1} \quad (4.51)$$

Lo que da una ecuación recurrente de  $\bar{U}_n$  que podemos resolver por expansión sucesiva hasta el último término  $\bar{U}_0 = 0$ :

$$\begin{aligned} \bar{U}_n &= \bar{U}_{n-1} + \frac{2}{n+1} \\ &= \bar{U}_{n-2} + \frac{2}{n} + \frac{2}{n+1} \\ &\dots \\ &= \bar{U}_0 + \frac{2}{2} + \frac{2}{3} + \dots + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= 2\left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-1} + \frac{1}{n} + \frac{1}{n+1}\right) \end{aligned}$$

La serie  $\sum_{i=1}^n \frac{1}{i}$  es el  $n$ -ésimo número armónico. Así pues:

$$\bar{U}_n = 2(H_{n+1} - 1) = 2H_{n+1} - 2 \approx 2(\ln(n+1) - \gamma) - 2 \approx 1,386 \lg(n+1) \quad \square \quad (4.52)$$

15

Ahora sustituimos (4.52) en (4.46):

$$\begin{aligned} \bar{S}_n &= \frac{(n+1)2(H_{n+1} - 1) - 2n}{n} + 1 \\ &= \frac{n+1}{n}(2(H_{n+1} - 1)) - 1 \\ &= 2\frac{n+1}{n}\left(H_n + \frac{1}{n+1} - 1\right) - 1 \\ &= 2\frac{n+1}{n}H_n - 3 \approx 2\ln n \approx 1,386 \lg n \quad \blacksquare \end{aligned}$$

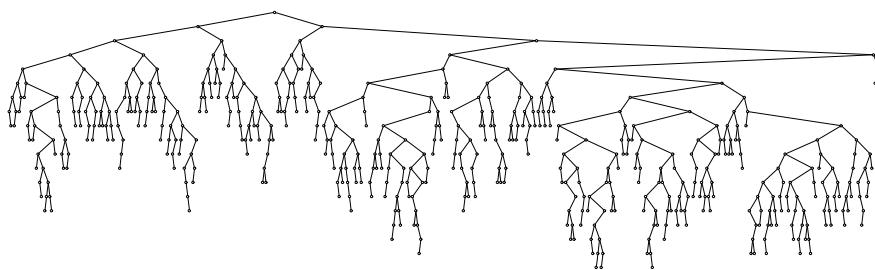


Figura 4.48: Un árbol binario de búsqueda de 512 claves aleatorias

<sup>15</sup>El símbolo  $\gamma$  denota la constante de Euler caracterizada por:

$$\gamma = \lim_{n \rightarrow \infty} (H_n - \ln n) \approx 0,5772156649\dots$$

Nuestro análisis supone que no hay supresiones intercaladas entre las inserciones. Si las supresiones son ejecutadas después de las inserciones hasta vaciar el árbol, el desempeño aún es  $\mathcal{O}(\lg n)$ , pues las premisas de la proposición 4.12 aún son ciertas.

El análisis con supresiones intercaladas entre la inserciones es un problema extremadamente difícil que a la fecha actual no ha sido resuelto satisfactoriamente. La dificultad estriba en que la eliminación efectúa una decisión que no es aleatoria cuando en la unión exclusiva escoge cuál será la raíz del subárbol resultante. No se sabe exactamente cómo manejar las probabilidades de las permutaciones ante estos desplazamientos. Además, es importante notar que el algoritmo de supresión desarrollado en § 4.9.6 (Pág. 324) plantea una simetría importante. Cuando se suprime un nodo completo y se efectúa el `join_exclusive()`, el nodo raíz del árbol resultante es el subárbol izquierdo. Esta decisión causa que el número de nodos por el lado izquierdo del punto de eliminación disminuya. Consecuentemente, la raíz del árbol tiende a desplazarse hacia la izquierda, lo cual crea un desequilibrio. Estudios empíricos arrojan una longitud promedio del camino tendiente a  $\mathcal{O}(\sqrt{n})$ .

La proposición 4.12 nos demuestra que si no se intercalan supresiones con inserciones, el desempeño esperado de la búsqueda y la inserción es  $\mathcal{O}(\lg n)$ , lo cual hace a un árbol binario una alternativa bastante aceptable para implantar tablas de símbolos en situaciones en que el orden de inserción sea aleatorio y el número de supresiones intercaladas sea relativamente pequeño; por ejemplo, la tabla de símbolos de un compilador. Lamentablemente, existen situaciones donde el orden de inserción podría ser sesgado. Por ejemplo, los apellidos en castellano están sesgados; probablemente, un apellido como León será más frecuente que Knuth.

## 4.10 El TAD DynMapTree

EL TAD `BinTree<Key>`, así como otros TAD para ABB que estudiaremos en el capítulo 6, efectúan todas sus operaciones en función de “nodos” y no en función del tipo de clave. Las razones para esto se fundamentan en el principio fin a fin y ya han sido aducidas anteriormente.

¿Cómo se planta un conjunto de claves mantenido por alguna clase de ABB? Dicho de otra manera, ¿cómo se resuelve el problema fundamental mediante un ABB?

*ALÉPH* implanta el conjunto fundamental con árboles binarios de búsqueda de dos maneras. La primera es mediante el tipo `DynSetTree<Tree, Key, Compare>`, el cual instrumenta un conjunto de claves de tipo `Key` implantado mediante alguna clase de árbol binario de búsqueda `Tree<Key>` y criterio de comparación `Compare`. La segunda manera es mediante una clase de mapeo llamada `DynMapTree`, cuya diferencia con `DynSetTree` es que la última guarda pares; o sea, maneja un conjunto “rango” del mapeo o función.

Todos los TAD fundamentados en ABB de este texto exhiben la misma interfaz que el TAD `BinTree<Key>`. Por esa razón, `DynSetTree<Tree, Key, Compare>` y `DynMapTree` pueden instrumentar cualquier clase de conjunto implementado mediante árboles binarios de búsqueda.

En esta sección desarrollaremos el más complejo de los TAD mencionados, la clase `DynMapTree`. Su función es implantar un conjunto de claves o un mapeo de claves en un dominio hacia elementos en un conjunto “rango” mediante una clase de ABB. El TAD en cuestión se define en el archivo `<tpl_dymapTree.H 332>`:

```

template <
 template <typename /* Key */, class /* Compare */> class Tree,
 typename Key, typename Range, class Compare = Aleph::less<Key> >
class DynMapTree
{
 ⟨Miembros privados de DynMapTree 333a⟩
 ⟨Miembros públicos de DynMapTree 334a⟩
};

⟨Clases dinámicas derivadas de DynMapTree 335c⟩

```

DynMapTree tiene cuatro parámetros clase. La clase Tree representa el tipo de ABB con el cual se desea implantar el mapeo. Entre los tipos posibles de los desarrollados en este texto tenemos BinTree<Key>, Rand\_Tree<Key>, Treap<Key>, Avl\_Tree<Key>, Rb\_Tree<Key> y Splay\_Tree<Key>. Con excepción de BinTree<Key>, el resto de las clases corresponde a ABB especiales, con prestaciones propias según el tipo de árbol, que implantan las mismas operaciones que BinTree<Key>. De este modo, el usuario de DynMapTree selecciona, según sus criterios de calidad y requerimiento, la clase de ABB que desea usar.

El parámetro clase Key representa el tipo de dato correspondiente al dominio del mapeo.

El parámetro Range representa el tipo de dato del rango del mapeo. Si lo que se desea es meramente mantener un conjunto de claves, entonces este parámetro se debe corresponder con una clase vacía; por ejemplo, Empty\_Node, el cual fue definido y usado para el TAD BinNode<Key>.

Finalmente, el último parámetro, constituye la clase de comparación entre claves de tipo Key.

Con lo anterior en mente podemos plantear los atributos de DynMapTree:

333a ⟨Miembros privados de DynMapTree 333a⟩≡ (332) 333b▷  
 Tree<Key, Compare> tree;  
 size\_t num\_nodes;

tree es una instancia de ABB con clave Key y num\_nodes es la cantidad de nodos que contiene el árbol tree. Esta cantidad se contabiliza en las primitivas de DynMapTree que inserten o eliminan nodos y es observable.

El TAD BinTree<Key> y demás TAD relacionados operan sobre nodos que almacenan una clave genérica de tipo Key. Ahora bien, para el mapeo requerimos de un dato de más de tipo Range, el cual se especifica por derivación del tipo genérico Tree<Key, Compare>::Node:

333b ⟨Miembros privados de DynMapTree 333a⟩+≡ (332) ▷333a  
 typedef typename Tree<Key, Compare>::Node Base\_Node;  
  
 struct Node : public Base\_Node  
 {  
 Range data;  
  
 Node() {}  
 Node(const Key & \_key) : Tree<Key, Compare>::Node(\_key) {}  
  
 Range & get\_data() { return data; }  
 };

DynMapTree opera internamente con nodos del tipo Node, los cuales se ordenan por clave de tipo Key, a nivel del árbol binario que se utilice, pero cada nodo contiene un par de tipo (Key, Range).

Los constructores por omisión y de copia pueden entonces definirse junto con el destructor:

334a *(Miembros públicos de DynMapTree 334a)≡* (332) 334b▷  
DynMapTree() : num\_nodes(0) {}

```
DynMapTree(DynMapTree & src_tree) : num_nodes(src_tree.num_nodes)
{
 Node * src_root = (Node*) src_tree.tree.getRoot();
 tree.getRoot() = copyRec(src_root);
}
virtual ~DynMapTree()
{
 if (num_nodes > 0)
 destroyRec(tree.getRoot());
}
```

Para insertar:

334b *(Miembros públicos de DynMapTree 334a)+≡* (332) ▷334a 334c▷

```
Range * insert(const Key & key, const Range & data)
{
 Node * node = new Node (key);
 node->data = data;
 if (tree.insert(node) == NULL)
 {
 delete node;
 return NULL;
 }
 ++num_nodes;

 return &node->data;
}
```

La rutina retorna la cantidad de nodos que tiene el árbol. Puesto que no se permiten elementos repetidos, esta cantidad puede cotejarse para saber si ocurrió o no la inserción.

Para la eliminación, sólo basta la clave, y ésta se hace mediante el siguiente método:

334c *(Miembros públicos de DynMapTree 334a)+≡* (332) ▷334b 335a▷

```
size_t remove(const Key & key)
{
 Node * node = (Node*) tree.remove(key);
 if (node == NULL)
 return num_nodes;

 delete node;

 return -num_nodes;
}
```

remove() retorna la cantidad de elementos, lo que permite, al igual que insert(), determinar si hubo o no eliminación.

Hay ocasiones en las cuales es deseable eliminar todos los elementos. Para ello usamos el método `empty()`:

335a *(Miembros públicos de DynMapTree 334a) +≡* (332) ◁334c 335b ▷  
`void empty()  
{  
 num_nodes = 0;  
 destroyRec(tree.getRoot());  
}`

Hay dos maneras de consultar:

335b *(Miembros públicos de DynMapTree 334a) +≡* (332) ◁335a  
`bool test_key(const Key & key)  
{  
 return (Node*) tree.search(key) != NULL;  
}  
Range * test(const Key & key)  
{  
 Node * p = (Node*) tree.search(key);  
 return p != NULL ? &(p->get_data()) : NULL;  
}`

La primera, `test_key()`, se destina para conjuntos que sólo contengan claves, mientras que la segunda, `test()`, es para mapeos.

Otra manera de insertar y buscar elementos es mediante el operador `[]`, el cual “indiza” claves y retorna imágenes dentro del rango. Por supuesto, este esquema sólo tiene sentido si se trata de un mapeo y no de un conjunto de claves.

Del TAD `DynMapTree` podemos especializar diversas clases según el tipo de árbol binario de búsqueda; por ejemplo:

335c *(Clases dinámicas derivadas de DynMapTree 335c) ≡* (332)  
`template <typename Key, typename Type, class Compare = Aleph::less<Key> >  
class DynMapBinTree : public DynMapTree<BinTree, Key, Type, Compare> {};`

`template <typename Key, typename Type, class Compare = Aleph::less<Key> >  
class DynMapAvlTree : public DynMapTree<Avl_Tree, Key, Type, Compare> {};`

El segundo tipo, `DynMapAvlTree`, es un mapeo basado en árboles AVL, una clase especial de árbol binario de búsqueda, cuya altura está determinísticamente acotada, y que estudiaremos en § 6.4 (Pág. 471). Hay más clases según los tipos de árboles binarios.

## 4.11 Extensiones a los árboles binarios

EL TAD `BinTree<Key>` y sus derivados (ver § 6 (Pág. 451)) son idóneos para la solución al problema fundamental con las operaciones de inserción, búsqueda y eliminación en  $\mathcal{O}(\lg(n))$ . Esto es un gran paso respecto a un arreglo donde la inserción y la eliminación son  $\mathcal{O}(n)$ . Sin embargo, un arreglo ordenado nos ofrece la alternativa de buscar el  $i$ -ésimo menor elemento en  $\mathcal{O}(1)$ , mientras que en un ABB, esta operación requiere un recorrido infijo que es  $\mathcal{O}(n)$ .

Existe una estructura de dato, basada en un ABB, que nos permite mantener un conjunto de claves con las siguientes operaciones y sus tiempos:

- Inserción, búsqueda y eliminación de una clave en  $\mathcal{O}(\lg(n))$ .

- Acceso al  $i$ -ésimo elemento del recorrido infijo en  $\mathcal{O}(\lg(n))$ .
- Conocimiento de la posición infija dada una clave en  $\mathcal{O}(\lg(n))$ .

El principio fundamental de la estructura de datos subyace en almacenar la cardinalidad del árbol en cada nodo. Para ello nos valdremos de la siguiente definición:

**Definición 4.10 (Árbol con rangos)** Un árbol binario con rangos es un árbol binario con un campo adicional en cada nodo, denotado  $C(n)$ , que almacena la cardinalidad y el cual satisface:

$$\forall n \in T, \quad C(n) = C(L(n)) + 1 + C(R(n)), \quad C(\emptyset) = 0$$

Un ABB puede ser extendido y lo definimos como sigue.

**Definición 4.11 (Árbol binario de búsqueda extendido)** Un árbol binario de búsqueda extendido  $T$  es un árbol binario de búsqueda con rangos.

El acrónimo ABBE refiere a un árbol binario de búsqueda extendido.

La figura 4.49 ilustra un ABBE. El campo superior es la clave y el inferior es la cardinalidad del subárbol. En la figura, cada nodo tiene una etiqueta, que no forma parte de la estructura de datos, correspondiente a su posición infija.

Dado  $n_i \in T \mid i \in \text{ABBE}$ , entonces,  $C(L(n_i))$  nos da la posición infija de  $n_i$  respecto al subárbol con raíz  $n_i$ . Este es el conocimiento que nos permitirá acceder al árbol por su posición infija.

Estamos listos para diseñar una infraestructura de abstracciones y código que nos maneje árboles extendidos. La primera fase consiste en definir la estructura de un nodo binario componente de un ABBE, la cual se define en el archivo *<tpl\_binNodeXt.H 336>*:

336 *<tpl\_binNodeXt.H 336>*≡  
 class BinNodeXt\_Data  
 {  
 size\_t count; // cardinalidad del árbol  
  
 BinNodeXt\_Data() : count(1) {}  
 BinNodeXt\_Data(SentinelCtor) : count(0) {}  
  
 size\_t & getCount() { return count; }  
 const size\_t & size() const { return count; }  
};  
DECLARE\_BINNODE\_SENTINEL(BinNodeXt, 255, BinNodeXt\_Data);  
#define COUNT(p) ((p)->getCount())  
*(Primitivas básicas sobre BinNodeXt<Key> 337)*

Uses DECLARE\_BINNODE\_SENTINEL.

La cardinalidad se guarda en el atributo `count`, el cual tiene su observador y modificador definidos en la parte pública de la clase `BinNodeXt<Key>`.

En un ABBE es deseable utilizar un nodo centinela especial que represente el árbol vacío. La ventaja del centinela es que su campo `count` siempre es cero, lo que evita la necesidad de verificar si el nodo accedido es el nulo o no en aquellos algoritmos que pregunten la cardinalidad. La cardinalidad de  $\emptyset$  es 0, que es el mismo valor de la posición infija de un nodo incompleto por la izquierda.

El nodo centinela es una instancia estática de `BinNodeXt<Key>`, cuyo espacio es reservado en el macro `DECLARE_BINNODE_SENTINEL`.

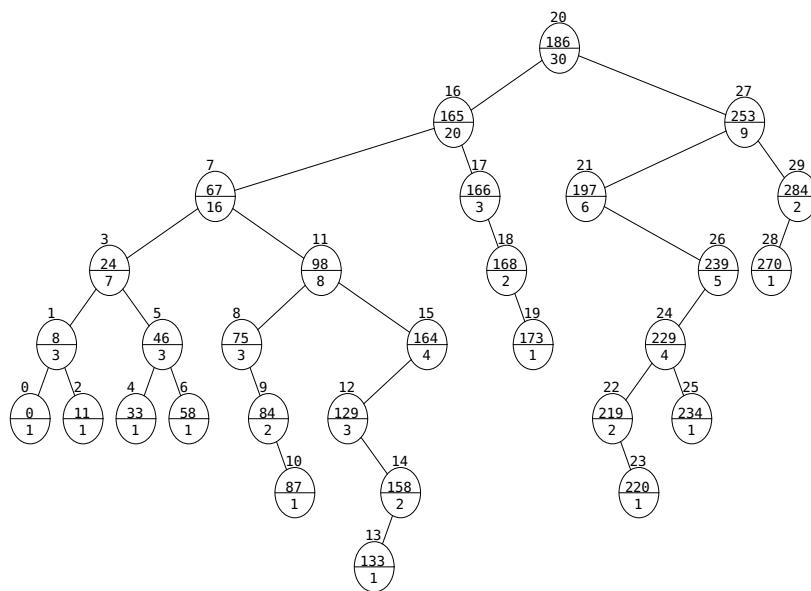
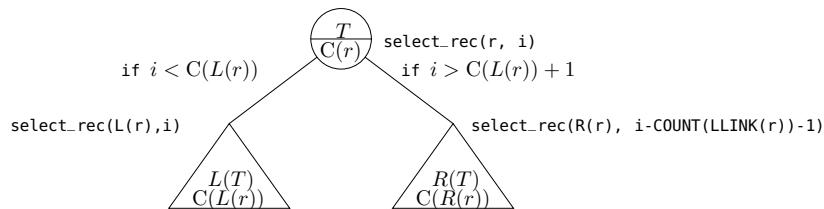


Figura 4.49: Un árbol binario extendido

#### 4.11.1 Selección por posición

Dado  $T \in \text{ABBE}$  con raíz  $r$  se desea encontrar el  $i$ -ésimo elemento en la posición infija. La estructura de datos, aunada al uso del nodo centinela, ofrece una solución completamente general, es decir, sin ningún caso particular, la cual se pictoriza del siguiente modo:



Mención particular merece la llamada recursiva por la derecha. Generalmente hablando, cuando buscamos por la derecha lo hacemos relativamente sobre un árbol que contiene  $C(R(r))$  nodos, pero la posición  $i$  de la llamada original atañe a un árbol con  $C(r)$  nodos. Debemos, pues, compensar la llamada con la cantidad de nodos que el recorrido infijo deja cuando se va por la derecha. Esto es:  $C(L(r))$  del subárbol izquierdo más la raíz.

Con lo anteriormente expuesto comprendido, el algoritmo resultante debe ser sencillo:

```

(Primitivas básicas sobre BinNodeXt<Key> 337)≡ (336) 338a▷
template <class Node> inline Node * select_rec(Node * r, const size_t & i)
{
 if (i == COUNT(LLINK(r)))
 return r;

 if (i < COUNT(LLINK(r)))
 return select_rec((Node*) LLINK(r), i);

 return select_rec((Node*) RLINK(r), i - COUNT(LLINK(r)) - 1);
}

```

```
}
```

La versión iterativa también es muy sencilla, más segura y mucho más eficiente:

338a *(Primitivas básicas sobre BinNodeXt<Key> 337) +≡* (336) ▷337 338b▷

```
template <class Node> inline Node * select(Node * r, const size_t & pos)
{
 for (size_t i = pos; i != COUNT(LLINK(r)); /* nada */)
 if (i < COUNT(LLINK(r)))
 r = static_cast<Node*>(LLINK(r));
 else
 {
 i -= COUNT(LLINK(r)) + 1;
 r = static_cast<Node*>(RLINK(r));
 }
 return r;
}
```

#### 4.11.2 Cálculo de la posición infija

Dada una clave existente en el ABB, deseamos calcular su posición infija; es decir, su orden dentro del conjunto de claves. Este problema se resuelve recursivamente del siguiente modo:

338b *(Primitivas básicas sobre BinNodeXt<Key> 337) +≡* (336) ▷338a 338c▷

```
template <class Node,
 class Compare = Aleph::less<typename Node::key_type> > inline
int inorder_position(Node * r, const typename Node::key_type & key, Node *& p)
{
 if (Compare () (key, KEY(r)))
 return inorder_position<Node, Compare>((Node*) LLINK(r), key, p);
 else if (Compare () (KEY(r), key))
 return inorder_position<Node, Compare>((Node*) RLINK(r), key, p) +
 COUNT(LLINK(r)) + 1;
 else
 {
 p = r;
 return COUNT(LLINK(r));
 }
}
```

La rutina recibe como entrada la raíz del árbol *r* y la clave *key*. El tercer parámetro, *node*, es de salida y es un nodo que contiene la clave *key*; este resultado tiene sentido sólo si la clave fue encontrada. El valor de retorno es la posición de *key* dentro del recorrido infijo. En caso de que *key* no se encuentre en el árbol, entonces se retorna un valor negativo.

#### 4.11.3 Inserción por clave en árbol binario extendido

La inserción por clave es idéntica a la de un ABB presentada en§ 4.9.3 (Pág. 320), salvo que necesitamos actualizar los contadores:

338c *(Primitivas básicas sobre BinNodeXt<Key> 337) +≡* (336) ▷338b 339▷

```
template <class Node,
```

```

 class Compare = Aleph::less<typename Node::key_type> > inline
Node * insert_by_key_xt(Node *& r, Node * p)
{
 if (r == Node::NullPtr)
 return r = p;

 Node * q;
 if (Compare () (KEY(p), KEY(r)))
 {
 q = insert_by_key_xt<Node, Compare>((Node*&) LLINK(r), p);
 if (q != Node::NullPtr)
 ++COUNT(r);
 }
 else if (Compare ()(KEY(r), KEY(p)))
 {
 q = insert_by_key_xt<Node, Compare>((Node*&) RLINK(r), p);
 if (q != Node::NullPtr)
 ++COUNT(r);
 }
 else
 return (Node*) Node::NullPtr; // clave duplicada

 return q;
}

```

La sintaxis y semántica son las mismas que para los ABB.

#### 4.11.4 Partición por clave

La partición por clave explicada en § 4.9.4 (Pág. 321) puede implantarse con la misma estructura para un ABBE:

339

*<Primitivas básicas sobre BinNodeXt<Key> 337>+≡* (336) ▷338c 340a▷

```

#define SPLIT split_key_rec_xt<Node, Compare>
template <class Node,
 class Compare = Aleph::less<typename Node::key_type> > inline
bool split_key_rec_xt(Node * root, const typename Node::key_type & key,
 Node *& l, Node *& r)
{
 if (root == Node::NullPtr)
 {
 l = r = Node::NullPtr;
 return true;
 }
 if (Compare() (key, KEY(root)))
 {
 if (not SPLIT(LLINK(root), key, l, LLINK(root)))
 return false;
 r = root;
 COUNT(r) -= COUNT(l);
 }
 else if (Compare() (KEY(root), key))
 {

```

```

 if (not SPLIT(RLINK(root), key, RLINK(root), r))
 return false;
 l = root;
 COUNT(l) -= COUNT(r);
 }
else
 return false; // clave duplicada

 return true;
}

```

Defines:  
`split_key_rec_xt`, used in chunk 340a.

#### 4.11.5 Inserción en raíz

Con la función anterior podemos implantar la inserción en la raíz bajo el mismo esquema que la desarrollada en § 4.9.7 (Pág. 326):

340a *(Primitivas básicas sobre BinNodeXt<Key> 337)+≡* (336) ▷339 340b▷

```

template <class Node,
 class Compare = Aleph::less<typename Node::key_type> > inline
Node * insert_root_xt(Node *& root, Node * p)
{
 if (root == Node::NullPtr)
 return p;

 if (not split_key_rec_xt<Node, Compare>(root, KEY(p), LLINK(p), RLINK(p)))
 return Node::NullPtr;

 COUNT(p) = COUNT(LLINK(p)) + COUNT(RLINK(p)) + 1;
 root = p;

 return p;
}
```

Uses `split_key_rec_xt` 339.

#### 4.11.6 Partición por posición

Esta operación se parece a la partición por clave, con la excepción de que el “punto” de partición es respecto a una posición  $i$  del recorrido infijo. La partición resulta en dos árboles  $T_l = \langle k_1 k_2 \dots k_l \rangle$  y  $T_r = \langle k_i \dots k_n \rangle$ .

Salvo que  $i$  esté fuera de rango, es decir,  $i \geq C(T)$ , el algoritmo siempre tiene éxito.

Para esta situación diseñamos el siguiente algoritmo reminiscente a la partición recursiva por clave:

340b *(Primitivas básicas sobre BinNodeXt<Key> 337)+≡* (336) ▷340a 341▷

```

template <class Node> inline
void split_pos_rec(Node * r, const size_t & i, Node *& ts, Node *& tg)
{
 if (i == COUNT(r)) // ¿Es la última posición (que está vacía)?
 {
```

```

 ts = r;
 tg = Node::NullPtr;
 return;
 }
 if (i == COUNT(LLINK(r))) // ¿se alcanzó posición de partición?
 {
 ts = LLINK(r);
 tg = r;
 LLINK(tg) = Node::NullPtr;
 COUNT(tg) -= COUNT(ts);
 return;
 }
 if (i < COUNT(LLINK(r)))
 {
 split_pos_rec((Node*) LLINK(r), i, ts, (Node*&) LLINK(r));
 tg = r;
 COUNT(r) -= COUNT(ts);
 }
 else
 {
 split_pos_rec((Node*) RLINK(r), i - (COUNT(LLINK(r)) + 1),
 (Node*&) RLINK(r), tg);
 ts = r;
 COUNT(r) -= COUNT(tg);
 }
}

```

Defines:

`split_pos_rec`, used in chunk 341.

#### 4.11.7 Inserción por posición

Consideremos el recorrido infijo  $k_0, k_1, k_2, \dots, k_{i-1}, k_i, k_{i+1}, \dots, k_{n_1}$ . Cuando insertamos  $k_p$  en la  $i$ -ésima posición, el recorrido resultante es  $k_0, k_1, k_2, \dots, k_{i-1}, k_p, k_i, k_{i+1}, \dots, k_{n_1}$ , es decir, a partir de la posición  $i$ , los elementos se desplazan hacia la derecha; la posición infija  $k_p$  es  $i$ , su predecesor es  $k_{i-1}$  y su sucesor es  $k_i$ . El algoritmo que proponemos es muy sencillo si nos inspiramos del de inserción en la raíz presentado en § 4.9.7 (Pág. 326):

341 ⟨*Primitivas básicas sobre BinNodeXt<Key>* 337⟩+≡ (336) ◁340b 342a▷

```

template <class Node> inline
void insert_by_pos_xt(Node *& r, Node * p, const size_t & pos)
{
 split_pos_rec(r, pos, (Node*&) LLINK(p), (Node*&) RLINK(p));
 COUNT(p) = COUNT(LLINK(p)) + 1 + COUNT(RLINK(p));
 r = p;
}

```

Uses `split_pos_rec` 340b.

Salvo que `pos` esté fuera de rango, o sea, que sea igual o sobrepase la cantidad de elementos, la inserción siempre tendrá éxito.

La inserción por posición puede violar la condición de orden de un ABB.

#### 4.11.8 Unión exclusiva de árboles extendidos

La unión exclusiva presentada en § 4.9.5 (Pág. 323) es estructuralmente idéntica en su versión para árboles extendidos. Lo único que debemos modificar es la actualización de los contadores:

342a *(Primitivas básicas sobre BinNodeXt<Key> 337) +≡* (336) ◁341 342b ▷

```

template <class Node> inline Node * join_exclusive_xt(Node *& ts, Node *& tg)
{
 if (ts == Node::NullPtr)
 return tg;

 if (tg == Node::NullPtr)
 return ts;

 LLINK(tg) = join_exclusive_xt(RLINK(ts), LLINK(tg));
 RLINK(ts) = tg;

 // actualizar contadores
 COUNT(tg) = COUNT(LLINK(tg)) + 1 + COUNT(RLINK(tg));
 COUNT(ts) = COUNT(LLINK(ts)) + 1 + COUNT(RLINK(ts));

 Node * ret_val = ts;
 ts = tg = Node::NullPtr; // deben quedar vacíos después del join

 return ret_val;
}
```

Defines:  
*join\_exclusive\_xt*, used in chunks 342b and 343.

#### 4.11.9 Eliminación por clave en árboles extendidos

Eliminar por clave en un árbol extendido también es muy similar a la estudiada en § 4.9.6 (Pág. 324), con la adición de que hay que actualizar los contadores en el camino de búsqueda:

342b *(Primitivas básicas sobre BinNodeXt<Key> 337) +≡* (336) ◁342a 343 ▷

```

#define REMOVE remove_by_key_xt<Node, Compare>
template <class Node,
 class Compare = Aleph::less<typename Node::key_type>> inline
Node * remove_by_key_xt(Node *& root, const typename Node::key_type & key)
{
 if (root == Node::NullPtr)
 return (Node*) Node::NullPtr; // clave no encontrada

 Node * ret_val = Node::NullPtr;
 if (Compare () (key, KEY(root)))
 {
 ret_val = REMOVE((Node*&) LLINK(root), key);
 if (ret_val != Node::NullPtr) // ¿hubo eliminación?
 -COUNT(root); // Sí ==> actualizar contador
 return ret_val;
 }
}
```

```

else if (Compare () (KEY(root), key))
{
 ret_val = REMOVE((Node*&) RLINK(root), key);
 if (ret_val != Node::NullPtr) // ¿hubo eliminación?
 -COUNT(root); // Sí ==> actualizar contador
 return ret_val;
}
ret_val = root; // clave encontrada ==> eliminar
root = join_exclusive_xt((Node*&)LLINK(root), (Node*&)RLINK(root));
return ret_val;
}

```

Uses join\_exclusive\_xt 342a.

#### 4.11.10 Eliminación por posición en árboles extendidos

Si la posición infija es válida, entonces, al igual que con la inserción y la partición, la eliminación siempre tiene éxito y se define como sigue:

343 ⟨Primitivas básicas sobre BinNodeXt<Key> 337⟩+≡ (336) ◁342b 345b▷

```

template <class Node> inline
Node * remove_by_pos_xt(Node *& root, const size_t & pos)
{
 if (COUNT(LLINK(root)) == pos) // ¿posición encontrada?
 {
 // Si ==> guarde nodo y realice join exclusivo
 Node * ret_val = root;
 root = join_exclusive_xt((Node*&) LLINK(root),
 (Node*&) RLINK(root));
 return ret_val;
 }
 Node * ret_val; // guarda valor de retorno de llamada recursiva
 if (pos < COUNT(LLINK(root)))
 ret_val = remove_by_pos_xt((Node*&) LLINK(root), pos);
 else
 ret_val = remove_by_pos_xt((Node*&) RLINK(root),
 pos - (COUNT(LLINK(root)) + 1));
 if (ret_val != Node::NullPtr) // ¿hubo eliminación?
 -COUNT(root); // Si ==> el árbol con raíz root perdió un nodo

 return ret_val;
}

```

Uses join\_exclusive\_xt 342a.

#### 4.11.11 Desempeño de las extensiones

Según la proposición § 4.12 (Pág. 329), una secuencia de inserción aleatoria produce un árbol que en promedio está equilibrado. Consecuentemente, si construimos un ABBE por inserción de claves aleatorias, entonces los desempeños de funciones que no modifiquen el árbol serán  $\mathcal{O}(\lg(n))$ ; este es el caso de la selección, determinación del orden infijo y de la inserción por clave.

En el capítulo 6 estudiaremos diversas técnicas para mantener equilibrados árboles binarios de búsqueda que manejan  $\mathcal{O}(\lg(n))$  en todas las operaciones. Con cualquiera de estas técnicas, siempre es posible mantener los rangos.

## 4.12 Rotación de árboles binarios

La figura 4.50 muestra dos ABB equivalentes según la propiedad de orden. Cada uno de ellos puede obtenerse a partir del otro a través de una transformación llamada “rotación”.

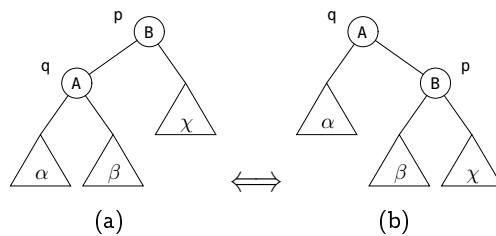


Figura 4.50: Rotación de un árbol binario

El recorrido infijo del árbol 4.50(a) es  $(\alpha A \beta) B \chi$ , mientras que el del 4.50(b) es  $\alpha A (\beta B \chi)$ . La operación de rotación no afecta la propiedad de orden de un ABB.

El árbol 4.50(b) se obtiene a partir del árbol 4.50(a) mediante una “rotación hacia la derecha” u “horaria” del nodo B. Análogamente, el árbol 4.50(a) se obtiene a partir del árbol 4.50(b) mediante una “rotación hacia la izquierda” o “antihoraria” del nodo A.

Hay dos características importantes de una rotación que exemplificaremos con la rotación hacia la derecha:

1. El nodo A disminuye en un nivel, mientras que el nivel del nodo B del árbol original aumenta en uno.
2. La rama  $\alpha$  disminuye enteramente en un nivel, mientras que la rama  $\chi$  aumenta enteramente en un nivel.

Ahora veamos cómo implantar la rotación hacia la derecha. Sea  $p$  la raíz del árbol a rotar, entonces:

344 *(Funciones de BinNode\_Utils 243a) +≡* <327 345a>

```
template <class Node> inline Node * rotate_to_right(Node * p)
{
 Node * q = static_cast<Node*>(LLINK(p));
 LLINK(p) = RLINK(q);
 RLINK(q) = p;
 return q;
}
```

`rotate_to_right()` retorna la nueva raíz del árbol luego de la rotación. Con este valor de retorno puede actualizarse el padre del árbol resultante. Por lo general, esta primitiva se utiliza en algoritmos recursivos que invocan rotaciones de subárboles.

Para algoritmos iterativos es más conveniente utilizar una versión de la rotación que efectúe la actualización del padre. Para ello debemos pasar el padre del subárbol que será

rotado:

```
345a <Funciones de BinNode_Utils 243a>+≡ ◁344
 template <class Node> inline Node * rotate_to_right(Node * p, Node * pp)
 {
 Node *q = static_cast<Node*>(LLINK(p));
 LLINK(p) = RLINK(q);
 RLINK(q) = p;
 if (static_cast<Node*>(LLINK(pp)) == p) // actualización del padre
 LLINK(pp) = q;
 else
 RLINK(pp) = q;
 return q;
 }
```

pp es el parente del nodo p.

Las versiones por la izquierda son simétricas.

#### 4.12.1 Rotaciones en árboles binarios extendidos

Con árboles extendidos hay que tener cuidado de ajustar los contadores. Retomando el caso general de rotación ilustrado en la figura 4.50, tenemos un árbol binario  $(\alpha A \beta) B \chi$  que deseamos rotar. Por la definición de árbol extendido, tenemos que:

$$\underbrace{C(B)}_{|B|} = \left( \underbrace{C(\alpha)}_{|\alpha|} + \underbrace{1}_{|\{A\}|} + \underbrace{C(\beta)}_{|\beta|} \right) + \underbrace{1}_{|\{B\}|} + \underbrace{C(\chi)}_{|\chi|} .$$

Después de rotar se debe cumplir:

$$\underbrace{C(A)}_{|A|} = \underbrace{C(\alpha)}_{|\alpha|} + \underbrace{1}_{|\{B\}|} + \left( \underbrace{C(\beta)}_{|\beta|} + \underbrace{1}_{|\{B\}|} + \underbrace{C(\chi)}_{|\chi|} \right) .$$

Así pues, la primitiva de rotación hacia la derecha será:

```
345b <Primitivas básicas sobre BinNodeXt<Key> 337>+≡ (336) ◁343
 template <class Node> inline Node * rotate_to_right_xt(Node* p)
 {
 Node * q = static_cast<Node*>(LLINK(p));
 LLINK(p) = RLINK(q);
 RLINK(q) = p;
 COUNT(p) -= 1 + COUNT(LLINK(q));
 COUNT(q) += 1 + COUNT(RLINK(p));
 return q;
 }
```

## 4.13 Códigos de Huffman

La representación de este texto consiste en una serie de símbolos latinos, arabescos, griegos y otros más convenidos desde mundos de los físicos y matemáticos. En papel, con el adecuado formato, los símbolos conforman una secuencia visualmente legible por un hispano hablante.

A la acción de reducir el espacio ocupado por un texto (o secuencia) se le denomina “comprimir”. En la elaboración de un texto hay un compromiso entre la “compresión” y la legibilidad. En aras de la legibilidad se sacrifica la compresión, y así es como debe ser. Si, por ejemplo, la letra es muy pequeña, usamos menos espacio, papel, pero el texto se hace menos legible. Por el contrario, si la letra es muy grande, entonces requerimos más espacio, ergo, más papel o pantalla.

En la representación cibernética de los símbolos de un texto se usa un “código” de bits. En nuestras circunstancias, un código es un acuerdo de representación de símbolos mediante combinaciones de bits. Un ejemplo notable lo constituye el código ASCII, el cual mapea secuencias de siete bits a un símbolo. He en la siguiente tabla algunas de sus ocurrencias:

| Símbolo | Valor Binario | Valor decimal |
|---------|---------------|---------------|
| A       | 1000001       | 65            |
| b       | 1100010       | 98            |
| {       | 1111011       | 123           |

Hay otros sistemas de codificación de caracteres, de los cuales uno muy notable es el **unicode**, extensible, de secuencias de longitud variable y con la pretensión de servir a todos los símbolos de todas las lenguas.

Designemos a  $\Sigma$  como el conjunto de todos los posibles símbolos que puede contener un texto. En este caso podemos codificar los  $|\Sigma|$  símbolos con  $\lceil \lg |\Sigma| \rceil$  bits. Acordado el código, cualquier agente (o programa) puede “interpretar” un texto.

Si consideramos a un texto particular  $T_x$  cuyos símbolos se remiten a un conjunto  $S \subset \Sigma$ , entonces podríamos codificarlo con  $\lceil \lg |S| \rceil \leq \lceil \lg |\Sigma| \rceil$  bits. Para ello construimos una tabla mapeo de símbolos a secuencias de bits y nos servimos de ella para “interpretar” el texto. Tenemos aquí una primera forma de compresión. Sin embargo, este método tiene la eventual desventaja de que el texto debe leerse por anticipado a efectos de construir la tabla. Esto puede ser problemático en algunas circunstancias, la lectura de un archivo, por ejemplo; y prohibitivo en otras, la transmisión de un texto.

Otro inconveniente del enfoque anterior es que cada símbolo ocupa la misma cantidad de bits, independientemente de su frecuencia de aparición en el texto. Si usásemos secuencias de longitud variable, entonces podríamos escoger secuencias muy cortas para símbolos muy frecuentes y dejar las más largas para símbolos de rara aparición. La idea es seleccionar los símbolos de un bit, luego, los de dos, y así sucesivamente según cuan frecuente sea la aparición del símbolo en el texto. Por ejemplo, el blanco, denotado, para distinguirlo, como “□”, y la “a”, que son muy frecuentes, podrían ser la secuencias 0 y 1 respectivamente; mientras que la “e”, “i”, “b” y “c” podrían denotarse como 00, 01, 10 y 11. Podríamos continuar con secuencias más largas hasta abarcar completamente el conjunto S. Pero así, en bruto, este enfoque plantea una ambigüedad insalvable expresada por instancia particular en la secuencia 001, pues no se puede distinguir si se trata de “□□a” o “ea” o “□i”.

La ambigüedad anterior se solventa si, en detrimento de la cantidad de combinaciones posibles de bits, usamos un código prefijo, en el sentido de la definición 4.8<sup>16</sup>, es decir, que ninguna secuencia  $s_i$  de longitud  $i$  sea prefija de alguna otra de cardinalidad superior. Si escogemos permutaciones de secuencias prefijas, entonces podemos, sin problema alguno,

<sup>16</sup>En la sección § 4.8.0.7 (Pág. 302) se usaron los símbolos “a” por “0” y “b” por “1”.

distinguir las e interpretarlas en un texto. Más aún, podemos utilizar secuencias de Dick, las cuales, como corolario de las proposiciones 4.7 y 4.10, son prefijas.

Hay una mejora aún más substancial en usar una codificación prefija: podemos mapear en  $\mathcal{O}(1)$ , en línea con la lectura del texto, su decodificación. Para ello, usamos un árbol binario cuyas hojas codifican los símbolos de  $S$ . Por ejemplo, para  $S = \{\square, a, e, i, b, c\}$ , cualquier árbol binario de 6 hojas codifica a  $S$ ; una ocurrencia se muestra en la figura 4.51.

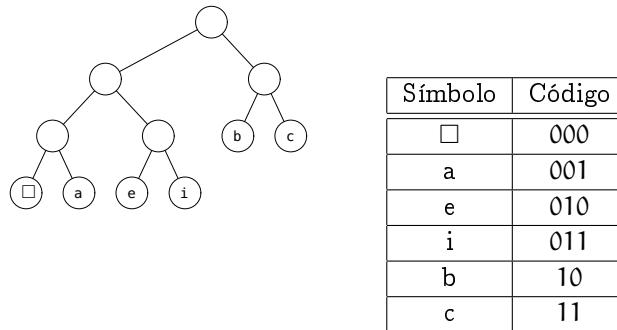


Figura 4.51: Árbol codificador de  $S = \{\square, a, e, i, b, c\}$  y su mapeo

#### 4.13.1 Un TAD para árboles de código

En el contexto de la codificación, siempre podemos distinguir dos agentes: uno codificador, que toma un texto y lo comprime, y otro decodificador, que toma el texto comprimido y lo decodifica a la secuencia original.

Para codificar usaremos la siguiente clase:

347a *(Clase codificadora 347a)≡*  

```
class Huffman_Encoder_Engine
{
 <miembros privados de codificador 347c>
 <miembros públicos de codificador 351a>
};
```

Esta clase se encarga de construir un árbol de prefijos óptimo y de codificar textos en función del árbol anterior. Por óptimo pretendemos decir que un texto codificado con los prefijos del árbol óptimo ocupa el menos espacio posible.

Para decodificar usaremos la siguiente clase:

347b *(Clase decodificadora 347b)≡*  

```
class Huffman_Decoder_Engine
{
 <miembros privados de decodificador 348a>
 <miembros públicos de decodificador 349d>
};
```

Las clases y otras definiciones se encuentran en el archivo *(Huffman.H (never defined))*.

Codificar y decodificar una secuencia requiere de varias estructuras de datos. La primera de ellas es el propio árbol de códigos, el cual se define del siguiente modo:

347c *(miembros privados de codificador 347c)≡* (347a) 348f▷  

```
BinNode<string> * root;
```

Uses BinNode 240.

348a *(miembros privados de decodificador 348a)*≡ (347b)  
`BinNode<string> * root;`  
 Uses BinNode 240.

root es, en ambas clases, la raíz de un árbol de prefijos. Posteriormente detallaremos un algoritmo para construir este árbol de manera óptima (§ 4.13.3 (Pág. 350)). Del mismo modo, también plantearemos un criterio de eficiencia y demostraremos su optimización (§ 4.13.6 (Pág. 355)). En el ínterin hay que señalar que el árbol cuya raíz es root contiene cadenas (string) y no símbolos ASCII. La razón es que en lugar de símbolos puntuales puede establecerse alguna frecuencia de aparición de frases enteras y codificarse enteramente en una secuencia de bits.

Necesitamos una “tabla de símbolos” en la cual almacenemos sus frecuencias de aparición y que nos permita construir un árbol de prefijos. El tipo de esta tabla se define así:

348b *(Declaraciones Huffman 348b)*≡ 348c▷  
`class Huffman_Node;`  
`typedef DynMapTree<Treap_Vtl, string, Huffman_Node *> Symbol_Map;`

La tabla mapea símbolos de tipo string a nodos de un heap que usaremos para determinar óptimamente los prefijos. El parámetro Treap\_Vtl es una clase especial de árbol binario de búsqueda llamado Treap, el cual será tratado en § 6.3 (Pág. 464). Si tenemos alguna dificultad en aceptar el tipo Treap\_Vtl, entonces podemos utilizar, con la misma interfaz, y probablemente con desempeño similar, el tipo BinTreeVtl previamente estudiado en § 4.9.2 (Pág. 319).

Por razones que explicaremos prontamente, un árbol de prefijos se construye a partir de un heap del cual sus nodos son de tipo Huffman\_Node y se definen de la siguiente forma:

348c *(Declaraciones Huffman 348b)*+≡ <348b 348d>  
`typedef BinNode< Aleph::pair<string, size_t> > Freq_Node;`  
`struct Huffman_Node : public BinHeap<size_t>::Node`  
`{`  
 `BinNode<string> * bin_node;`  
`};`  
 Uses BinNode 240.

Grosso modo, Huffman\_Node es un nodo binario de un heap cuya clave, accedida mediante get\_key(), es la frecuencia de aparición o estadística de un símbolo. El nodo en cuestión contiene un apuntador a un nodo dentro del árbol de prefijos.

El tipo de heap se define de la siguiente manera:

348d *(Declaraciones Huffman 348b)*+≡ <348c 348e>  
`typedef BinHeap<size_t> Huffman_Heap;`

En función de este tipo definimos las siguientes funciones auxiliares:

348e *(Declaraciones Huffman 348b)*+≡ <348d 349a>  
`static inline const size_t & get_freq(Huffman_Node * huffman_node)`  
`static inline void increase_freq(Huffman_Node * huffman_node)`  
`static inline void set_freq(Huffman_Node * huffman_node, const size_t & freq)`

Ahora podemos definir el heap dentro del codificador y su tabla de símbolos:

348f *(miembros privados de codificador 347c)*+≡ (347a) <347c 349b>  
`Huffman_Heap heap;`  
`Symbol_Map symbol_map;`

Por ejemplo, `symbol_map["a"].get_key()` retorna la frecuencia asociada al símbolo "a".

La última estructura de datos es la tabla de códigos, la cual requiere la definición del siguiente tipo:

349a *<Declaraciones Huffman 348b>+≡* ◁348e  
`typedef DynMapTree<Treap_Vtl, string, BitArray> Code_Map;`  
*Uses BitArray 54a.*

349b *<miembros privados de codificador 347c>+≡* (347a) ◁348f 349c▷  
`Code_Map code_map;`

Esta tabla se utiliza para codificar un texto. La idea es leer secuencialmente el texto y encontrar en la tabla de códigos el correspondiente código a afectos de generar el texto codificado. Al igual que con la tabla de símbolos, la de códigos se implementa con un Treap (§ 6.3 (Pág. 464)).

Hay otros atributos adicionales que usaremos en la clase `Huffman_Encoder_Engine`:

349c *<miembros privados de codificador 347c>+≡* (347a) ◁349b 351b▷  
`string end_symbol;`  
`size_t text_len;`

`end_symbol`, que también se utiliza en `Huffman_Decoder_Engine`, es un símbolo especial que denota la finalización de un texto; su definición coayuda a determinar cuándo culmina una secuencia de bits correspondiente a un texto codificado. `text_len` en la longitud del texto sin codificar.

#### 4.13.2 Decodificación

Al acto que, objetivamente, hemos denominado “interpretar”, lo podemos denominar como “decodificar”. De este modo, asumiendo una raíz `root` de un árbol de prefijos y un puntero `p`, de la siguiente forma podemos decodificar una secuencia `bit_stream` de `bit_stream_len` bits mediante la siguiente rutina:

349d *<miembros públicos de decodificador 349d>≡* (347b)  
`void decode(BitArray & bit_stream, ostream & output)`  
`{`  
 `const size_t & bit_stream_len = bit_stream.size();`  
 `BinNode<string> * p = root;`  
 `for (int i = 0; i < bit_stream_len; ++i)`  
 `{`  
 `if (bit_stream.read_bit(i) == 0)`  
 `p = LLINK(p);`  
 `else`  
 `p = RLINK(p);`  
 `if (is_leaf(p)) // ¿es hoja?`  
 `{`  
 `// si ==> escribir símbolo y reiniciar a raíz`  
 `const string & symbol = p->get_key();`  
 `if (symbol == end_symbol) // ¿se alcanza final?`  
 `break;`  
 `output << symbol;`

```

 p = root; // reiniciar a raíz, pues se leerá un nuevo código
}
}
}

Uses BinNode 240 and BitArray 54a.
```

La idea es partir desde la raíz del árbol y, según el valor de bit leído, descender hacia la izquierda o derecha hasta deparar en alguna hoja<sup>17</sup>. En ese entonces, el prefijo leído se corresponde con un símbolo.

Así pues, para decodificar una secuencia de bits, el usuario debe instanciar un objeto de tipo `Huffman_Decoder_Engine`, cuyo constructor recibe la raíz del árbol código y el símbolo considerado como fin de la entrada. Luego, se invoca al método `decode()`, el cual arroja su salida al parámetro `output`. Conocido el árbol, el proceso de decodificación es relativamente sencillo.

#### 4.13.3 Algoritmo de Huffman

Dado un conjunto de  $n$  símbolos queremos construir un árbol de códigos de exactamente  $n$  hojas. Haciendo abstracción de que estas hojas fungen de nodos externos, entonces, por la proposición 4.2, cualquier árbol de  $n - 1 + n = 2n - 1$  nodos, con cualquier permutación de símbolos como hojas, constituye un árbol de códigos. Por la proposición 4.11 sabemos que existen  $C_{2n-1} = \frac{1}{2n} \binom{2(2n-1)}{2n-1}$  diferentes árboles de código. Surgen entonces las preguntas: ¿cuál escoger?, ¿cómo construirlo?

Huffman [81] descubrió un método para construir un árbol de prefijos el cual parte de un conjunto iniciado con los símbolos y sus frecuencias de aparición. Es decir, un conjunto de nodos del tipo `Huffman_Node` ya descrito. Llamemos a este conjunto `heap`, el cual se implanta mediante un `heap`.

Para construir un nuevo nodo `huffman_node`, primero hay que seleccionar los dos nodos con menores frecuencias de la siguiente forma:

350a *(sacar del heap los dos nodos de menor frecuencia 350a)≡* (351a)  
`Huffman_Node * l_huffman_node = (Huffman_Node*) heap.getMin(); // izq  
Huffman_Node * r_huffman_node = (Huffman_Node*) heap.getMin(); // der`

Con estos dos nodos se construye un subárbol de Huffman de la siguiente forma:

350b *(crear un nuevo nodo de Huffman 350b)≡* (351a)  
`BinNode <string> * bin_node = new BinNode <string>;  
Huffman_Node * huffman_node = new Huffman_Node (bin_node);  
LLINK(bin_node) = l_huffman_node->bin_node;  
RLINK(bin_node) = r_huffman_node->bin_node;  
const size_t new_freq = get_freq(l_huffman_node) + get_freq(r_huffman_node);  
Aleph::set_freq(huffman_node, new_freq);`  
Uses BinNode 240.

El nuevo nodo se construye como padre de los dos menores contenidos en el heap. Importante notar que el nuevo subárbol es de tipo `BinNode<string>`, que es el tipo del árbol de Huffman y no de tipo `Huffman_Node`.

---

<sup>17</sup>Notemos que un prefijo guarda remembranza con el número de Deway de la hoja.

La manera definitiva para construir un árbol de prefijos se efectúa así:

351a *(miembros públicos de codificador 351a)≡* (347a) 352a▷

```

BinNode<string> * generate_huffman_tree(
)
{
 while (heap.size() > 1) // hasta que quede solo un nodo
 {
 <sacar del heap los dos nodos de menor frecuencia 350a>
 <crear un nuevo nodo de Huffman 350b>
 delete l_huffman_node;
 delete r_huffman_node;
 heap.insert(huffman_node);
 } // nodo que queda en heap es la raíz del árbol de prefijos

 Huffman_Node * huffman_root = (Huffman_Node *) heap.getMin();
 root = huffman_root->bin_node;
 delete huffman_root;
 build_encoding_map(); // construir mapeo de códigos

 return root;
}

```

Uses BinNode 240.

Una vez que se genera el árbol de Huffman debemos, a efectos de codificar, generar el mapeo de símbolos a códigos prefijos. Esta tarea la efectúa la primitiva `build_prefix_encoding()`, la cual se instrumenta mediante un recorrido prefijo sobre el árbol de Huffman:

351b *(miembros privados de codificador 347c)++* (347a) ▷349c 351c▷

```

void build_prefix_encoding(BinNode<string> * p, BitArray & array)
{
 if (is_leaf(p))
 {
 const string & str = p->get_key();
 code_map.insert(str, BitArray(array));
 return;
 }
 array.push(0); build_prefix_encoding(LLINK(p), array); array.pop();
 array.push(1); build_prefix_encoding(RLINK(p), array); array.pop();
}

```

Uses BinNode 240 and BitArray 54a.

La rutina se corresponde con un recorrido prefijo, recursivo, del árbol de Huffman con raíz `root`. A medida que se va recorriendo, el prefijo de `root`<sup>18</sup> se va manteniendo en el arreglo de bits `array` y su longitud se mantiene en el parámetro `len`. Cuando se alcanza una hoja, es decir, cuando ya tenemos el prefijo completo del símbolo, insertamos en el mapeo `code_map` la dupla `<símbolo, array>`.

`build_prefix_encoding()` es activada por la siguiente primitiva:

351c *(miembros privados de codificador 347c)++* (347a) ▷351b 352c▷

```

void build_encoding_map()
{
 BitArray array(0);
 symbol_map.empty();
}

```

<sup>18</sup>Vale la pena recordar que este prefijo es una palabra de Dick (véase § 4.8.0.8 (Pág. 306)).

```

 build_prefix_encoding(root, array);
}

Uses BitArray 54a.

```

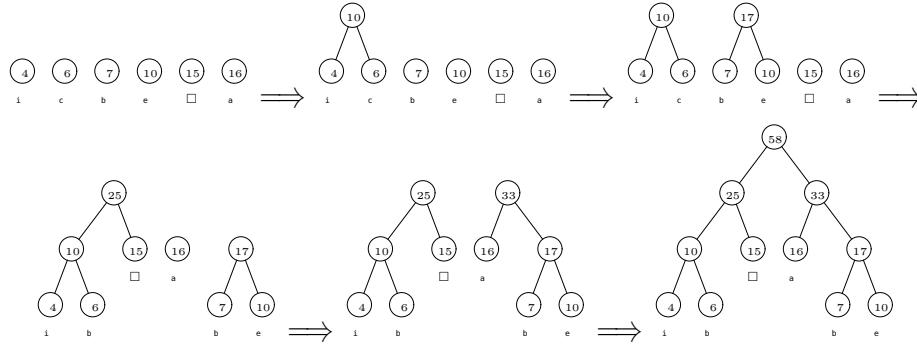


Figura 4.52: Construcción de un árbol de Huffman. Las raíces de los subárboles se encuentran en el heap

El orden de manipulación de la clase `Huffman_Encoder_Engine` es como sigue:

1. Definir símbolos con sus frecuencias
2. Generar árbol de prefijos
3. Codificar textos

En esta sección hemos llevado a cabo la segunda fase. La primera será explicada en la siguiente subsección, mientras que la tercera será explicada en § 4.13.5 (Pág. 353).

#### 4.13.4 Definición de símbolos y frecuencias

Antes de construir el árbol de prefijos según el algoritmo anterior debemos indicar los símbolos y sus respectivas frecuencias. Hay dos maneras de hacerlo. La primera consiste en especificar directamente el símbolo y su correspondiente frecuencia, lo cual se hace mediante `set_freq(str, freq)`, donde `str` es el símbolo a definir y `freq` su frecuencia asociada. El método en cuestión se especifica como sigue:

352a *(miembros públicos de codificador 351a)*+≡ (347a) ◁ 351a 352b ▷  
`void set_freq(const string & str, const size_t & freq)`

La otra forma de indicar frecuencias es a partir de un texto, identificar sus símbolos y contabilizar sus frecuencias de aparición. Esto se hace mediante `read_input(input)`, donde `input` es el texto a partir del cual se desea contabilizar las frecuencias:

352b *(miembros públicos de codificador 351a)*+≡ (347a) ◁ 352a 353b ▷  
`void read_input(char * input  
 )`

`read_input()` subyace sobre la rutina `update_freq(curr_token)`, la cual se remite a buscar `curr_token` en el mapeo de símbolos, crear una nueva entrada si no está mapeado, incrementar en uno la frecuencia y actualizar el heap:

352c *(miembros privados de codificador 347c)*+≡ (347a) ◁ 351c 354 ▷  
`void update_freq(const string & str)`

```

{
 Huffman_Node * huffman_node = NULL;
 Huffman_Node ** huffman_node_ptr = symbol_map.test(str);
 if (huffman_node_ptr == NULL) // ¿símbolo definido previamente?
 { // No ==> crear una entrada en symbol_map e insertarlo en heap
 unique_ptr<BinNode<string> > bin_node_auto(new BinNode<string>(str));
 huffman_node = static_cast<Huffman_Node*>
 (heap.insert(new Huffman_Node(bin_node_auto.get())));
 symbol_map.insert(str, huffman_node);
 bin_node_auto.release();
 }
 else
 huffman_node = *huffman_node_ptr; // ya definido, recuperarlo

 increase_freq(huffman_node);
 heap.update(huffman_node);
}

```

Uses BinNode 240.

Puesto que el fin de los métodos `read_input()` es mirar las frecuencias y construir la entrada, éstos culminan con la generación del árbol de códigos:

353a ⟨Generar árbol 353a⟩≡  
`set_end_of_stream("");`  
`generate_huffman_tree(with_freqs);`

#### 4.13.5 Codificación de texto

Una vez construido un árbol de Huffman podemos codificar el texto mediante cualquiera de los métodos `encode(input, bit_stream)`. `input` es el texto a codificar y `bit_stream` es un `BitArray` del tipo definido en § 2.1.5 (Pág. 53):

353b ⟨miembros públicos de codificador 351a⟩+≡ (347a) ◁ 352b  
`size_t encode(char * input, BitArray & bit_stream)`  
`{`  
 `char * curr_stream = input;`  
 `char curr_token[Max-Token-Size];`  
 `curr_token[1] = '\0';`  
 `while (*curr_stream != '\0')`  
 `{`  
 `curr_token[0] = *curr_stream++;`  
 `append_code(bit_stream, code_map[curr_token]);`  
 `}`  
 `append_code(bit_stream, code_map[""]);`  
  
 `return bit_stream.size();`  
`}`

Uses BitArray 54a.

El segundo parámetro de `encode()` es el arreglo de bits que contiene el texto codificado. El proceso, una vez generado el mapeo con los prefijos, es muy sencillo: buscar el símbolo leído en el mapeo y concatenar al arreglo de bits el prefijo. Tal tarea la efectúa la primitiva

Muerte De Antoñito El Camborio  
Federico García Lorca

Voces de muerte sonaron cerca del Guadalquivir.  
Voces antiguas que cercan voz de clavel varonil.  
Les clavó sobre las botas mordiscos de jabalí.  
En la lucha daba saltos jabonados de delfín.  
Baño con sangre enemiga su corbata carmesí,  
pero eran cuatro puñales y tuvo que sucumbir.  
Cuando las estrellas clavan rejonas al agua gris,  
cuando los erales sueñan verónicas de alhelí,  
voces de muerte sonaron cerca del Guadalquivir.

Antonio Torres Heredia,  
Camborio de dura crin,  
moreno de verde luna,  
voz de clavel varonil:  
¿quién te ha quitado la vida cerca del Guadalquivir?

Mis cuatro primos Heredias hijos de Benamejí.  
Lo que en otros no envidiaban, ya lo envidiaban en mí.  
Zapatos color corinto, medallones de marfil,  
y este cutis amasado con aceituna y jazmín.  
¡Ay Antoñito el Camborio,  
digno de una Emperatriz!  
Acuérdate de la Virgen porque te vas a morir.  
¡Ay Federico García,  
llama a la Guardia Civil!  
Ya mi talle se ha quebrado como caña de maíz.

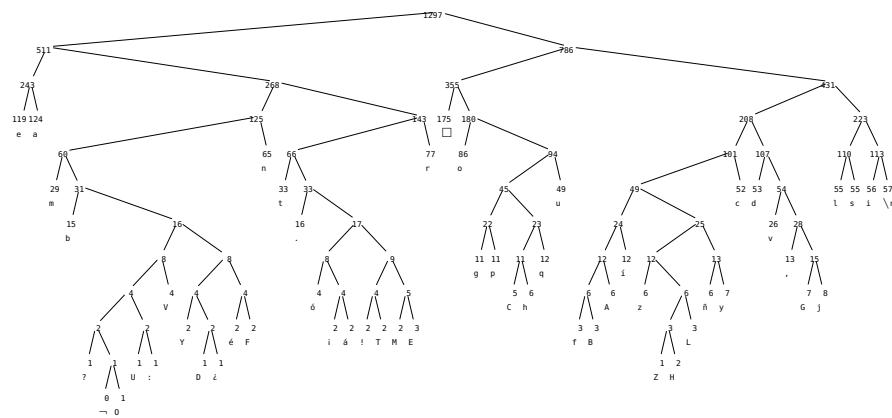
Tres golpes de sangre tuvo y se murió de perfil.  
Viva moneda que nunca se volverá a repetir.  
Un ángel marchoso pone su cabeza en un cojin.  
Otros de rubor cansado, encendieron un candil.  
Y cuando los cuatro primos llegan a Benamejí,  
voces de muerte cesaron cerca del Guadalquivir.

Figura 4.53: Un texto de García Lorca

privada append\_code():

354 *(miembros privados de codificador 347c)+≡* (347a) ↳352c  
 static void append\_code(BitArray & bit\_stream, const BitArray & symbol\_code)  
{  
 const size\_t symbol\_code\_size = symbol\_code.size();  
 for (int i = 0; i < symbol\_code\_size; i++)  
 bit\_stream.push(symbol\_code[i]);  
}

Uses BitArray 54a.



El árbol de la figura 4.54 comprime el texto mostrado en la figura 4.53 a 5942 bytes.

#### 4.13.6 Optimación de Huffman

Preguntémosnos, ¿es un árbol de Huffman óptimo?<sup>19</sup> Para confrontar la pregunta debemos plantearnos un criterio que nos pondere el coste de un símbolo en función de su frecuencia de aparición y que nos conduzca a una forma general y única de descubrir el mejor árbol de códigos.

**Definición 4.12 (Longitud ponderada del camino)** Sea un árbol binario  $T$  en el cual a cada hoja  $n_e \in T$  se le asigna un coste o ponderación  $w(n_e)$ . Entonces, la longitud ponderada del camino, denotada como  $wpl(T)$ , se define como:

$$wpl(T) = \sum_{n_e \text{ es hoja}} \text{nivel}(n_e) \times w(n_e) \quad (4.53)$$

Para el árbol de la figura 4.55-a tenemos:

$$wpl(T) = 2 \times (7 + 6) + 3 \times (15 + 16 + 10 + 4) = 161$$

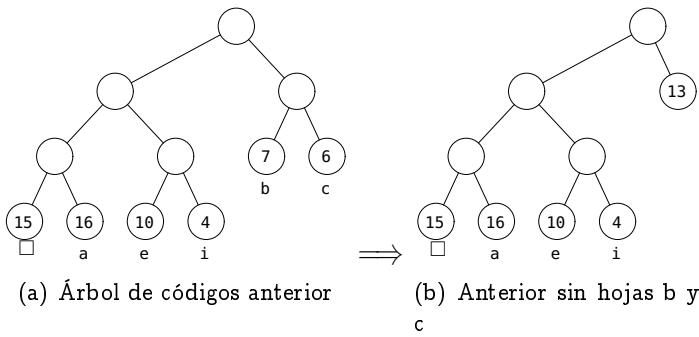


Figura 4.55: Árboles prefijos ejemplo

Cuanto menor altura tenga una hoja, menor será su incidencia sobre el  $wpl$ . La idea es construir un árbol de códigos cuyo  $wpl(T)$  sea mínimo; este es el criterio de optimización. Tal árbol se denomina “de Huffman” en honor a su descubridor, quien descubrió la siguiente proposición:

<sup>19</sup>Según D.R.A.E., “optimación” significa “acción y efecto de optimar”. El verbo “optimar” -u “optimizar”- connota, D.R.A.E. dixit, “buscar la mejor manera de realizar una actividad”. Ahora bien, en el estricto sentido de universalidad, ¿existe una mejor manera de llevar a cabo una actividad?. Responder afirmativamente y dar cuenta sobre la supuesta mejor manera, es muy delicado. Si para un cierto asunto asumimos y mostramos una optimación, entonces, en el asunto en cuestión, jamás regresaríamos a mejorarlo; ¿para qué, si el “óptimo” ya es el mejor?. Lo que es mejor es relativo a las circunstancias, y éstas atañen al momento, al lugar, pero, sobre todo, a quiénes. Asumir optimación sin consideración circunstancial conlleva peligro para las partes del contexto que no fueron sopesadas.

Quizá un indicio del desdén que a menudo conlleva una “optimación” se encuentra en su raíz etimológica. En latín “*opt̄imos*” connotaba “aristocrático”, perteneciente a los “*opt̄imates*”, quienes eran los miembros del senado romano, los precursores de los oligarcas de nuestra época. Tal parece que en la antigua Roma, los que gozaban de plenos derechos, quienes tenían mayor capacidad de elección, o sea, optaban (“*opt̄atus*”), eran los nobles.

**Proposición 4.13 (Optimación de Huffman - Huffman 1951 [80])** Sea  $\mathcal{N} = \{n_1, n_2, \dots, n_n\}$  un conjunto de nodos con pesos  $w(n_1), w(n_2), \dots, w(n_n)$ . Sea  $T$  un árbol prefijo construido según el algoritmo de Huffman (§ 4.13.3 (Pág. 350)). Entonces,  $\forall T' \neq T \implies wpl(T') \leq wpl(T)$ .

Lo anterior equivale a decir que el árbol de prefijos  $T$  es mínimo según su longitud ponderada del camino.

**Demostración** La prueba se construirá por inducción sobre la cardinalidad de  $T$ , y se valdrá del siguiente lema.

**Lema 4.5** Sea  $T$  un árbol con pesos en sus hojas. Sean  $n_1$  y  $n_2$  las dos hojas de  $T$  con peso mínimo. Consideremos un árbol  $T'$  tal que éste contenga las mismas hojas que  $T$ , excepto que las hojas  $n_1$  y  $n_2$  se substituyen por una hoja  $n_+$  con  $w(n_+) = w(n_1) + w(n_2)$ . Entonces, es posible construir un árbol  $T'$  tal que<sup>20</sup>:

$$wpl(T') \leq wpl(T) - w(n_+) \quad (4.54)$$

Por ejemplo, la figura 4.55-b ilustra un  $T'$  posible en el cual se le suprimen las hojas b y c. En este caso,  $wpl(T') = 13 + 3 \times (15 + 16 + 10 + 4) = 148$

**Demostración** Hay tres posibles situaciones a considerar para  $n_1, n_2 \in T$ :

1.  $n_1$  y  $n_2$  son hermanos: en este caso los suprimimos y ponemos en su padre  $n_3$ , que deviene en hoja,  $w(n_3) = w(n_1) + w(n_2)$ . En este árbol,  $n_3$  está en un nivel inferior  $i - 1$ , por lo que

$$\begin{aligned} wpl(T') &= wpl(T) - i \times (w(n_1) + w(n_2)) + (i - 1) \times (w(n_1) + w(n_2)) \\ &= wpl(T) - \underbrace{(w(n_1) + w(n_2))}_{w(n_3)} \end{aligned}$$

el cual difiere exactamente de  $wpl(T)$  en  $w(n_3)$ .

2.  $n_1$  y  $n_2$  están en el mismo nivel: asumamos  $s_1$  como el hermano de  $n_1$  (el caso es simétrico con  $n_2$ ). Entonces podemos intercambiar  $s_1$  con  $n_2$  y así deparar en el caso anterior.
3.  $n_1$  y  $n_2$  están en niveles distintos: al igual que en el punto anterior, asumamos  $s_1$  como el hermano de  $n_1$ . Del mismo modo, asumamos que  $\text{nivel}(n_1) > \text{nivel}(n_2)$ . Podemos pictorizar esta situación del modo mostrado en la figura 4.56.

Consideraremos la incidencia que tiene el subárbol  $s_1$  sobre  $wpl(T)$ , la cual puede expresarse así:

$$\sigma(s_1) = \sum_{\forall n_x \in \text{hoja}(s_1)} w(n_x) \times \text{nivel}(n_x)$$

Cuando hacemos el intercambio entre  $s_1$  y  $n_2$ , el valor de  $\sigma(s_1)$  disminuye a  $\sigma(s'_1)$ , pues  $\text{nivel}(s_1) = \text{nivel}(n_1)$  en  $T$ . Por tanto,

$$wpl(T') = wpl(T) + \text{nivel}(n_1) \times w(n_2) - \sigma(s_1) + \sigma(s'_1) \leq wpl(T)$$

<sup>20</sup>Es esencial distinguir la condición de posibilidad en el enunciado del lema, es decir, puede encontrarse una disposición topológica con los nodos restantes que satisfaga la desigualdad.

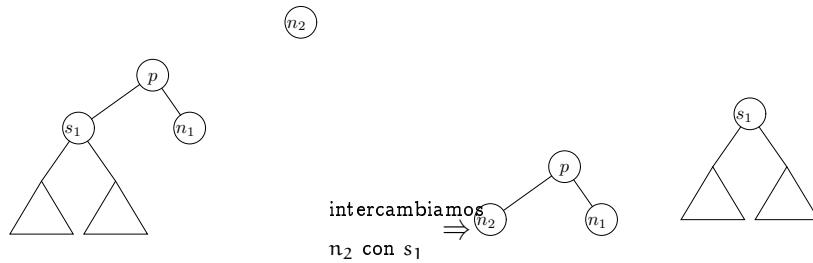


Figura 4.56: Esquema de prueba del tercer caso de la demostración del lema 4.5

El caso en que son iguales es cuando  $s_1$  es una hoja.

Ahora, cuando efectuamos la fusión de  $n_1$  y  $n_2$  en  $n_+$ , regresamos a la situación del primer caso:

$$\begin{aligned} wpl(T') &= wpl(T) + (\text{nivel}(n_1) - 1) \times (w(n_1) + w(s)) - \sigma(s_1) + \sigma(s_1') \\ &\leq wpl(T) - w(n_+) \end{aligned}$$

El caso cuando  $\text{nivel}(n_1) < \text{nivel}(n_2)$  es simétrico  $\square$

Este lema es el fundamento mostrativo que nos permitirá evidenciar que para cualquier árbol  $T' \leq T \implies wpl(T') \leq wpl(T)$ .

Retomemos la demostración de la proposición y planteemos inducción sobre la cardinalidad del árbol de Huffman  $T$ :

1. Para  $|T| \leq 2$  el resultado es directo.
2. Ahora asumamos que la proposición es cierta para todo árbol de Huffman y verifiquemos su veracidad para  $T_{n+1}$ ; es decir  $\forall T'_{n+1} \neq T_{n+1} \implies wpl(T_{n+1}) < wpl(T'_{n+1})$ . Sean  $n_1$  y  $n_2$  las dos primeras hojas que selecciona el algoritmo de Huffman. Ahora consideremos aplicar el lema 4.5 a  $T_{n+1}$  y  $T'_{n+1}$  con los nodos  $n_1$  y  $n_2$  y producir dos árboles  $T_n$  y  $T'_n$ .

Puesto que  $n_1$  y  $n_2$  son los nodos con menos peso, éstos son hermanos en el árbol de Huffman. Por tanto,  $wpl(T_n) = wpl(T_{n+1}) - w(n_1) - w(n_2)$ <sup>21</sup>.

Para el árbol  $T'_n$  sabemos, según el lema 4.5, que  $wpl(T'_n) \leq wpl(T'_{n+1}) - (w(n_1) + w(n_2))$ . Ahora bien, aunque quizás parezca obvio,  $|T_n| = |T'_n| < |T_{n+1}| = |T'_{n+1}|$ . Por tanto, podemos aplicar la hipótesis inductiva para concluir que  $wpl(T_n) \leq wpl(T'_n)$

■

#### 4.13.6.1 Conclusión

Recapitulemos el uso del algoritmo de Huffman mediante las dos clases que acabamos de presentar. El ente codificador se vale de una instancia de `Huffman_Encoder_Engine`, mientras que el decodificador de su contraparte de `Huffman_Decoder_Engine`. Obviamente, ambas instancias deben ponerse de acuerdo en usar el mismo árbol de Huffman. Sin embargo, aquí aparece un detalle interesante: para codificar o decodificar el texto no hace

<sup>21</sup> Esto se evidenció en la primera parte de la demostración del lema 4.5.

falta el árbol de Huffman. Esto es evidente en el codificador, pues, éste se remite a leer los símbolos del texto y, a través del mapeo de códigos, a emitir los correspondientes prefijos. Pero, ¿cómo haría el decodificador? La respuesta está dada por la teoría de prefijos impartida en § 4.8.0.7 (Pág. 302). En efecto, un texto codificado es una secuencia de palabras de Dick (§ 4.8.0.8 (Pág. 306)), el cual, como corolario de la proposición 4.7 (página 303), es prefijo. Esto implica que, dada una secuencia de prefijos, podemos reconocerlos entre sí. La técnica en cuestión se delega a ejercicio.

## 4.14 Árboles estáticos óptimos

Los árboles binarios son diseñados para emular la búsqueda binaria. Su bondad principal es que permiten inserciones y supresiones en  $\mathcal{O}(\lg(n))$ . Ahora bien, si el conjunto de claves fuese estático, es decir, no ocurre inserciones ni supresiones, ¿es, por ejemplo, un árbol binario de altura mínima eficiente?

Consideremos disponer las claves en un arreglo ordenado y efectuar la búsqueda binaria. Con este enfoque tenemos un árbol de longitud de camino mínimo. Además, ahorraremos espacio, pues no requerimos utilizar punteros. Como punto final, debemos remarcar que el arreglo aprovecha los caches del computador.

La búsqueda binaria asume que la frecuencia de acceso es equitativa para todas las claves y esto no es cierto en algunos casos. Algunas claves se acceden más que otras. Hay situaciones en que la frecuencia estadística de acceso puede conocerse antes de construir el árbol. En estos casos, un criterio de optimalidad consiste en minimizar la cantidad total esperada de búsquedas. Intuitivamente, esta minimización se logra colocando los nodos de acceso más frecuente cercanos a la raíz. El concepto que nos permitirá encontrar árboles óptimos subyace en la definición siguiente:

**Definición 4.13 (Longitud del camino interna ponderada)** Sea un árbol  $T$  de  $n$  nodos. Sea  $P = \{p_i \mid p_i \text{ es el peso del nodo } n_i \in T \mid \sum_{i=1}^n p_i = 1\}$ . Entonces, la longitud del camino interna ponderada se define como:

$$\begin{aligned} C_i(T) &= \sum_{i=1}^n (\text{nivel}(n_i) + 1)p_i \\ &= \sum_{i=1}^n \text{nivel}(n_i) p_i + 1 \end{aligned} \tag{4.55}$$

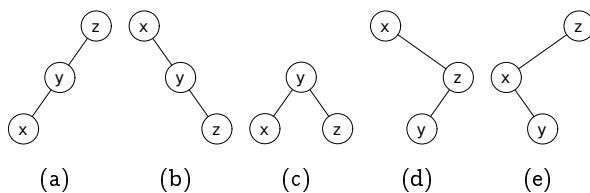


Figura 4.57: Árboles binarios de búsqueda de tres nodos

Dado un conjunto de  $n$  claves con sus probabilidades de acceso, el problema consiste en encontrar un árbol de  $n$  nodos cuyo  $C_i$  sea mínimo. Dicho de otro modo, ¿cuál, entre

todos los posibles árboles de  $n$  claves, tiene el  $C_i$  mínimo? Como ejemplo consideremos el conjunto de nodos  $\{X, Y, Z\}$  con pesos  $\{1/9, 3/9, 5/9\}$ . Según la proposición 4.11 existen  $\frac{1}{4} \binom{6}{3} = 5$  maneras de disponer tres nodos en un árbol binario de búsqueda, las cuales se ilustran en la figura 4.57. Las longitudes de camino ponderadas para cada árbol son:

$$\begin{aligned} C_i^{(a)} &= 3\frac{1}{9} + 2\frac{3}{9} + \frac{5}{9} = \frac{14}{9} \\ C_i^{(b)} &= 1\frac{1}{9} + 2\frac{3}{9} + 3\frac{5}{9} = \frac{21}{9} \\ C_i^{(c)} &= 2\frac{1}{9} + \frac{3}{9} + 2\frac{5}{9} = \frac{15}{9} \\ C_i^{(d)} &= \frac{1}{9} + 3\frac{3}{9} + 2\frac{5}{9} = \frac{20}{9} \\ C_i^{(e)} &= 2\frac{1}{9} + 3\frac{3}{9} + \frac{5}{9} = \frac{16}{9} \end{aligned}$$

Esto arroja al árbol (a) como el óptimo, resultado sorprendente y revelador de que lo óptimo no necesariamente es lo tradicionalmente bueno, pues, en nuestro ejemplo, el árbol óptimo es el más desequilibrado según los criterios tradicionales.

El ejemplo anterior nos da una idea acerca de las complejidades inherentes a un algoritmo. Podríamos generar todos los árboles de búsqueda posibles con  $n$  claves y entonces medir las longitudes de camino ponderadas para seleccionar la menor. Obviamente, esta técnica es muy costosa en tiempo y espacio.

El primer paso hacia la consecución de un algoritmo es aprehender que si un árbol es óptimo, entonces sus dos ramas también lo son. Formalizamos este hecho en la siguiente proposición:

**Proposición 4.14** Sea  $T \in \mathcal{B}$  un árbol binario de búsqueda óptimo. Entonces,  $L(T)$  y  $R(T)$  son óptimos.

**Demostración (Por contradicción)** : Sea  $k_1, k_2, \dots, k_l, \dots, k_n$  el recorrido infijo de un  $T \in \text{ABB}$  general de  $n$  nodos con raíz en clave  $k_l$ . De este modo,  $L(T)$  contiene las claves  $\{k_1, k_2, \dots, k_{l-1}\}$  y  $R(T)$  las claves  $\{k_{l+1}, \dots, k_n\}$ . Podemos entonces, según la definición 4.13, plantear  $C_i$  en función de la raíz  $k_l$ , del siguiente modo<sup>22</sup>:

$$\begin{aligned} C_i(T) &= p_l + \sum_{i=1}^{l-1} (\text{nivel}(k_i) + 1)p_i + \sum_{i=l+1}^n (\text{nivel}(k_i) + 1)p_i \\ &= p_l + C_i(L(T)) + \sum_{i=1}^{l-1} p_i + C_i(R(T)) + \sum_{i=l+1}^n p_i \\ &= \sum_{i=1}^n p_i + C_i(L(T)) + C_i(R(T)) \end{aligned} \tag{4.56}$$

Supongamos que el árbol óptimo contiene un  $L(T)$  o  $R(T)$  que no es óptimo. En este caso, según (4.56),  $C_i(T)$  aumentaría de valor, lo que contradice la suposición de que  $T$  es óptimo. La proposición es, pues, verdadera ■

<sup>22</sup>Recordemos que un nivel en  $L(T)$  o  $R(T)$  está desfasado en uno respecto a  $T$ . Por esa razón,  $\sum_{i=l+1}^n (\text{nivel}(k_i)p_i$ , en donde  $\text{nivel}(k_i)$  se refiere a  $T$ , es  $C_i(L(T))$ ; el razonamiento es el mismo para  $C_i(R(T))$

#### 4.14.1 Objetivo

Dado un arreglo ordenado  $K = [k_1, k_2, \dots, k_n]$  de  $n$  claves y un arreglo paralelo  $P = [p_1, p_2, \dots, p_n]$  de probabilidades de acceso, el problema consiste, recursivamente hablando, en determinar cuál clave  $k_l$ , entre las  $n$  posibles, debe colocarse como raíz del árbol final  $T$ . Una vez que se haga esta selección, el resto de las claves que irán en  $L(T)$  y  $R(T)$  queda totalmente definido, pues éstas deben satisfacer la propiedad de orden de un árbol binario. Los subárboles se construirían recursivamente aplicando el mismo principio sobre  $[k_1, k_2, \dots, k_{l-1}]$  y  $[p_1, p_2, \dots, p_{l-1}]$  para  $L(T)$  y  $[k_{l+1}, \dots, k_n]$  y  $[p_{l+1}, \dots, p_n]$ , para  $R(T)$ .

Esto nos conduce a una definición recursiva del costo: sea  $p_{i,j} = \sum_{l=i}^j p_l$  y  $C_{i,j}$  el costo total del árbol binario óptimo compuesto por las claves que van desde  $k_i$  hasta  $k_j$ . Entonces:

$$C_{i,j} = \min_{l=i}^j \left( \underbrace{p_{i,l-1}(1 + C_{i,l-1})}_{\text{árbol izquierdo}} + \underbrace{p_l}_{k_l \text{ como raíz}} + \underbrace{p_{l+1,j}(1 + C_{l+1,j})}_{\text{árbol derecho}} \right) \frac{1}{p_{i,j}} \quad (4.57)$$

El procedimiento es exponencial si lo aplicamos directamente. No obstante, los árboles óptimos deben satisfacer la propiedad de orden y, en este sentido, existen a lo sumo  $\frac{n(n+1)}{2}$  diferentes maneras de distribuir el conjunto  $K$  en un árbol binario. Podemos entonces diseñar un algoritmo ascendente que comience por construir todos los  $n-1$  árboles óptimos de 2 nodos. A partir de este resultado construimos los  $n-2$  árboles óptimos de 3 nodos y así sucesivamente.

Los costes de los subárboles óptimos se almacenan en una matriz  $\text{COST}[]$   $n \times n$ , donde cada entrada  $\text{COST}(i, j)$  representa el costo del árbol óptimo entre  $i$  y  $j$ . Calculamos iterativamente esta matriz por diagonales hasta llegar a la diagonal  $\text{COST}(1, n)$ , la cual contiene el costo óptimo del árbol de  $n$  nodos.

El algoritmo toma como entrada dos arreglos de dimensión  $n$ , donde  $n$  representa el número de claves. Al primero lo llamamos  $\text{keys}[]$  y contiene las claves. Al segundo lo llamamos  $[p]$  y contiene las frecuencias de acceso para cada clave.

El algoritmo requiere otra matriz  $\text{TREE}[]$  que almacene las raíces de los árboles óptimos. Dado un costo óptimo  $\text{COST}(i, j)$ ,  $\text{TREE}(i, j)$  contiene el índice de la raíz del árbol binario óptimo dentro del arreglo  $\text{keys}$ .

#### 4.14.2 Implantación

Implantamos nuestro algoritmo en el archivo  $\langle opBinTree.H 360 \rangle$  cuya estructura es la siguiente:

360       $\langle opBinTree.H 360 \rangle \equiv$   
            $\langle \text{Definición macros } opBinTree 361b \rangle$   
            $\langle \text{Funciones auxiliares } opBinTree 361d \rangle$

```
template <class Node, typename Key>
Node * build_optimal_tree(Key keys[], double p[], const int & n)
{
 <Declaración de matrices internas 361a>
 compute_optimal_costs(cost, p, n, tree);
 return compute_tree<Node, Key> (keys, tree, n, 1, n);
```

}

El archivo exporta una función principal llamada `build_optimal_tree()`. La entrada de la función es un arreglo ordenado de claves `keys[]`, un arreglo paralelo de probabilidades de acceso por cada  $i$ -ésima clave y la dimensión  $n$  de los respectivos arreglos. La salida es la raíz de un árbol binario óptimo conforme a las probabilidades dadas.

Las claves y probabilidades se ponen en arreglos estáticos. A causa de la gran escala, las matrices internas son más difíciles de mantener estáticas. Por esa razón, y, por añadidura, para no limitarnos por la fragmentación de memoria usaremos arreglos dinámicos:

361a *Declaración de matrices internas 361a*≡ (360)  
`DynArray <int> tree((n + 1)*(n + 1));  
 DynArray <double> cost((n + 1)*(n + 1));`

Uses `DynArray 34.`

`tree[]` es una matriz que almacena índices al parámetro arreglo `keys` de la función `build_optimal_tree()`. `cost[]` es una matriz que almacena costes de los árboles parciales requeridos para construir el árbol óptimo.

El acceso como matriz a una entrada de un arreglo dinámico se define del siguiente modo:

361b *Definición macros opBinTree 361b*≡ (360)  
`# define COST(i,j) (cost[(i)*(n+1) + (j)])  
# define TREE(i,j) (tree[(i)*(n+1) + (j)])`

`TREE(i,j)` almacena el índice a la raíz de un árbol óptimo parcial que contiene las claves entre  $i$  y  $j$ . Al final, la raíz del árbol óptimo definitivo será `keys[TREE(1, n)]`, la raíz del árbol izquierdo `keys[TREE(1, TREE(1, n) - 1)]`, etcétera.

`COST(i,j)` almacena el costo del árbol binario con claves entre `keys[i]` y `keys[j]`.

Como es la costumbre, todo macro debe ser indefinido al final del archivo:

361c *Indefinición macros opBinTree 361c*≡  
`# undef COST  
# undef TREE`

Ahora definimos la función `compute_optimal_cost()`, la cual se estructura en dos bloques:

361d *Funciones auxiliares opBinTree 361d*≡ (360) 362b▷  
`static inline  
void compute_optimal_costs(DynArray <double> & cost, double p[],  
 const int & n, DynArray <int> & tree)  
{  
 <Iniciar estructuras internas 361e>  
 <construir costes óptimos 362a>  
}`

Uses `DynArray 34.`

`compute_optimal_cost()` calcula las matrices de costes `COST[]` y los índices de las raíces de cada subárbol óptimo en `TREE[]`. El primer bloque *(Iniciar estructuras internas 361e)*, ubica los valores iniciales en las respectivas matrices:

361e *Iniciar estructuras internas 361e*≡ (361d)  
`int i, j, k;  
for (i = 1; i <= n; ++i)  
{  
 COST(i, i - 1) = 0;`

```

 TREE(i, i) = i;
}

```

El segundo bloque es el que calcula los costes definitivos. Primero con los subárboles de 2 nodos, luego, con los de 3, y así sucesivamente hasta obtener el valor de COST(0, n-1) y la raíz resultante en TREE(0, n-1):

362a  $\langle\text{construir costes óptimos } 362a\rangle \equiv$  (361d)

```

for (i = 0; i < n; ++i)
 for (j = 1; j <= n - i; ++j)
 {
 k = j + i;
 TREE(j, k) = min_index(cost, j, k, n);
 COST(j, k) = sum_p(p, j, k) +
 COST(j, TREE(j, k) - 1) + COST(TREE(j, k) + 1, k);
 }

```

sum\_p() efectúa la suma de las probabilidades entre j y k y se define como sigue:

362b  $\langle\text{Funciones auxiliares opBinTree } 361d\rangle \equiv$  (360) ▷361d 362c▷

```

static inline double sum_p(double p[], const int & i, const int & j)
{
 double sum = 0.0;
 for (int k = i - 1; k < j; ++k)
 sum += p[k];
 return sum;
}

```

min\_index() retorna un índice l, comprendido entre j y k, tal que COST(j, l - 1) + COST(l + 1, k) sea mínimo. Este valor se calcula mediante barrido directo de l entre j y k:

362c  $\langle\text{Funciones auxiliares opBinTree } 361d\rangle \equiv$  (360) ▷362b 363▷

```

static inline
int min_index(DynArray <double>& cost,
 const int & j, const int & k, const int & n)
{
 int ret_val;
 double min_sum = 1e32;
 for (int i = j; i <= k; ++i)
 {
 const double sum = COST(j, i - 1) + COST(i + 1, k);
 if (sum < min_sum)
 {
 min_sum = sum;
 ret_val = i;
 }
 }
 return ret_val;
}

```

Uses DynArray 34.

De la forma planteada, min\_index() es  $\mathcal{O}(n)$ , cuya llamada ocurre dentro de dos lazos  $\mathcal{O}(n)$ . Lo anterior implica que nuestro algoritmo es  $\mathcal{O}(n^3)$  un tiempo mucho mejor que el exponencial que produciría el algoritmo a fuerza bruta, pero aún cuestionable para valores de n muy grandes. La búsqueda del índice mínimo calculado en min\_index()

puede mejorarse si logramos demostrar que el subárbol mínimo se encuentra en el rango  $\text{TREE}(j, k - 1) \leq \text{TREE}(j, k) \leq \text{TREE}(j + 1, k)$ . En este caso, puede demostrarse que el algoritmo es  $\mathcal{O}(n^2)$ , lo que se delega como ejercicio.

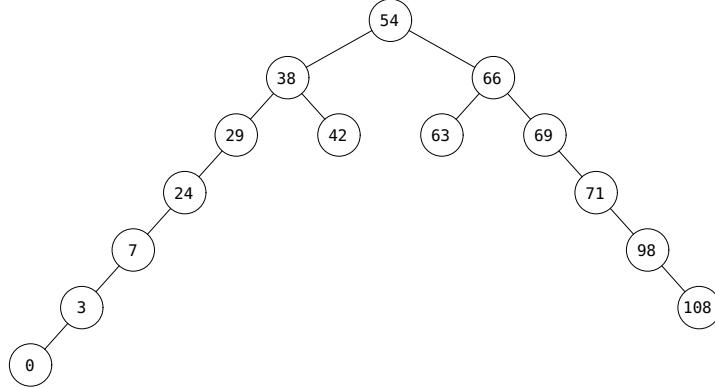


Figura 4.58: Árbol binario óptimo de 14 claves distribuidas binomialmente con  $p = 0,5$

Nos resta construir el árbol a partir de la matriz TREE. El índice de la raíz es  $\text{TREE}(1, n)$ , el de su subárbol izquierdo es  $\text{TREE}(1, \text{TREE}(1, n))$  y así sucesivamente. Estamos listos, pues, para diseñar la rutina:

363

```

⟨Funciones auxiliares opBinTree 361d⟩+≡ (360) ▷362c
 template<class Node, typename Key> static inline
 Node * compute_tree(Key keys[], DynArray<int>& tree,
 const int & n, const int & i, const int & j)
 {
 if (i > j)
 return Node::NullPtr;
 Node * root = new Node (keys[TREE(i, j) - 1]);
 LLINK(root) = compute_tree <Node, Key> (keys, tree, n, i, TREE(i, j) - 1);
 RLINK(root) = compute_tree <Node, Key> (keys, tree, n, TREE(i, j) + 1, j);
 return root;
 }

```

Uses DynArray 34.

## 4.15 Notas bibliográficas

Se podría decir que la noción matemática de árbol y sus conceptos fundamentales son resultado de varias eras y generaciones en la teoría de conjuntos. El volumen 1 del *Art of computing programming* de Knuth [97] recopila los indicios bibliográficos sobre los orígenes matemáticos de los árboles. Del mismo modo, Knuth es una de las mejores referencias para completar las matemáticas asociadas a los árboles presentadas en este texto y a otros tipos de árboles que no estudiamos. Desarrollos matemáticos más profundos pueden encontrarse en un texto formal sobre teoría de grafos. En la ocurrencia recomendamos el clásico de Harary, *Graph Theory* [70]. Desarrollos recientes pueden encontrarse en el texto *Combinatorial Algorithms* de Kreher y Stinson [102].

Los códigos de un árbol binario, el lenguaje de Lukasiewicz y la célebre notación polaca, fueron estudiados por el matemático polaco Jan Lukasiewicz [112]. La subsección § 4.8.0.7

(Pág. 302) fue inspirada a partir de los apuntes que este autor anotó y estudió del curso “*Algorithmique Combinatoire*” de 1993, de la Universidad Pierre et Marie Curie en París, Francia, e impartido por el excelente maestro Jean Berstel.

El corpus de compresión, homónimamente llamado “de Huffman”, fue descubierto por David Huffman en 1952, en aquel entonces “estudiante oficial”. Huffman jamás quiso patentar sus descubrimientos.

Según Knuth [99], los árboles binarios de búsqueda fueron descubiertos independiente-mente por varios investigadores durante la década de 1950. De nuevo, el lector es remitido a la obra de Knuth [99] para consultar las referencias exactas.

Todos los libros de estructuras de datos consagran varios capítulos a los árboles. En ese sentido, cualquier otro texto sirve de referencia adicional. Sin embargo, en lo que a árboles se refiere, ningún libro, incluido, por supuesto, el presente, puede considerarse completo. Un texto reciente, *Data Structures using C and C<sup>++</sup>* de Langsam, Augenstein y Tanenbaum [104] exhibe amplias reflexiones prácticas y presenta aplicaciones reales, notablemente, la evaluación de expresiones aritméticas mediante árboles hilados.

Los árboles hilados fueron descubiertos por A. J. Perlis y C. Thornton [141]. Aunque su uso es difícil, esta clase de árbol es usada en aplicaciones estadísticas, editores de textos y aplicaciones gráficas que manejen grandes volúmenes de información.

Los árboles con rango presentados en § 4.11 (Pág. 335) fueron desarrollados por este redactor para el presente trabajo como instrumento didáctico; con seguridad han sido desarrollados previamente. Las ideas sobre el uso de rangos para la vectorización de árboles es presentada para una clase de árbol equilibrado denominado AVL por Crane [33]. Aparentemente, el trabajo de Crane es pionero en los algoritmos de concatenación y partición.

Los heaps, junto con el heapsort, fueron presentados por primera vez por J. W. J. Williams [181] y R. W. Floyd [53].

En opinión de este redactor, la más auténtica y magistral explicación sobre heaps la ofrece Bentley en *Programming Pearls* [19]. La explicación es tan excelsa que desde hace muchos años (1988), cuando este redactor estudió los heaps, éste no puede desprenderse de su esquema de presentación de los heaps. Sirva, pues, este presente párrafo como pleitesía y cita.

El ahora clásico de Sedgewick, *Algorithms in C: Parts 1–4* [155], se caracteriza por exponer muchos algoritmos, de forma muy concisa pero a veces incompleta y en detrimento del formalismo. A pesar de ello, el Sedgewick es una de las mejores referencias para indagar algún algoritmo específico.

El volumen 3 del *Art of computer programming* de Knuth [99] contiene un análisis matemático bastante exhaustivo de los árboles binarios de búsqueda y de los árboles óptimos.

El *Introduction to Algorithms* de Cormen, Leiserson y Rivest [32] consagra un capí-tulo a las extensiones sobre árboles binarios. Aunque este capítulo se basa en una clase particular de árbol llamado “rojo-negro” (ver § 6.5 (Pág. 490)), las extensiones propuestas pueden usarse sobre prácticamente cualquier clase de árbol binario de búsqueda.

El algoritmo de inserción en la raíz presentado en § 4.9.7 (Pág. 326) fue descubierto por Stephenson y reportado en *A method for constructing binary search trees by making insertions at the root* [162]. Este trabajo marcó un hito sobre los árboles binarios, pues dio pie a nuevas ideas y usos, notablemente, los algoritmos de concatenación y partición

presentados en § 4.9.8 (Pág. 327) y § 4.11.6 (Pág. 340) y los árboles aleatorizados que serán presentados en § 6.2 (Pág. 455).

Los árboles binarios de búsqueda óptimos fueron propuestos y estudiados por Hu y Tucker en 1971 [79].

## 4.16 Ejercicios

1. Modifique el algoritmo 4.1 para que imprima un árbol arbitrariamente grande que no quepa en el medio de impresión (pantalla o impresora) (+).
2. Diseñe un algoritmo que tome como entrada una secuencia desordenada (una lista o un vector, poco importa) de pares <Valor nodo, Número de Dewey> y retorne el árbol representado con listas enlazadas y con arreglos.
3. Diseñe un algoritmo que tome de entrada un árbol cualquiera y retorne una secuencia ordenada de números de Deway. Considere los dos tipos de representación: listas enlazadas y arreglos.
4. Discuta y diseñe una estructura de dato orientada hacia la notación de Deway. Discuta algunos contextos en los cuales su diseño sería justificado.
5. Diseñe un algoritmo que busque una clave en un árbol n-ario representado mediante listas enlazadas. Discuta sobre las diferentes estrategias de búsqueda.
6. Diseñe un algoritmo que cuente el número de nodos en un árbol n-ario representado mediante listas enlazadas.
7. Diseñe un algoritmo que recorra en sufijo un árbol representado con listas.
8. Diseñe un algoritmo que escriba la representación parentizada de un árbol.
9. Diseñe un algoritmo que convierta un árbol representado con listas enlazadas en su equivalente representado con arreglos. Dé dos respuestas: la primera considerando conocido el orden del árbol, la segunda considerándolo desconocido.
10. Conciba cómo delimitar el recorrido prefijo para que contenga la información requerida para reconstruir el árbol original.
11. Conciba cómo delimitar el recorrido sufijo para que contenga la información requerida para reconstruir el árbol original.
12. Diseñe un algoritmo que construya un árbol binario a partir de los recorridos infijo y sufijo.
13. Modifique el recorrido por niveles level\_order (§ 4.4.12 (Pág. 251)) para que use una función de visita que reciba el nivel y la posición dentro del recorrido.
14. Diseñe un algoritmo no recursivo que copie de árboles binarios (+).
15. Diseñe un algoritmo no recursivo que destruya de árboles binarios (+).
16. Diseñe un algoritmo que recorra en prefijo un árbol binario hilado.

17. Diseñe un algoritmo que recorra en sufijo un árbol binario hilado.
18. Diseñe un algoritmo que hile un árbol binario. La entrada es un árbol binario normal; la salida es el mismo árbol, pero sus nodos están hilados según 4.4.15.
19. Dado un árbol binario, diseñe un algoritmo que efectúe la copia hilada.
20. Suponga un árbol binario cuyos nodos tienen un campo adicional llamado PLINK y que denota el nodo padre. Dado un nodo cualquiera p, diseñe algoritmos no recursivos y sin pila para:
  - (a) Determinar el nodo predecesor y,
  - (b) Determinar el nodo sucesor.
21. Los recorridos pseudohilados no consideran la posición del nodo en el recorrido ni su nivel. Modifique las funciones en cuestión para que se incluyan estos parámetros (++) .
22. Diseñe una rutina recursiva que recorra por niveles un árbol binario y use espacio  $\mathcal{O}(1)$  (+).
23. Diseñe un algoritmo no recursivo que efectúe el recorrido prefijo sobre una arborescencia especificada mediante el TAD Tree\_Node .
24. Diseñe un algoritmo no recursivo que efectúe el recorrido sufijo sobre una arborescencia especificada mediante el TAD Tree\_Node .
25. Considere los siguientes recorridos sobre una arborescencia:

Prefijo : 4 2 1 3 18 16 12 8 5 6 7 9 10 11 14 13 15 17 19 20

Sufijo : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Dibuje la arborescencia.
26. Defina e implante la similaridad entre dos árboles especificados mediante el TAD Tree\_Node .
27. Defina e implante la equivalencia entre dos árboles especificados mediante el TAD Tree\_Node .
28. Diseñe un algoritmo recursivo que implante la copia de un árbol especificado mediante el TAD Tree\_Node .
29. Diseñe un algoritmo no recursivo que implante la copia de un árbol especificado mediante el TAD Tree\_Node .
30. Diseñe un algoritmo no recursivo que ejecute la búsqueda por número de Dewey.
31. Complete la solución de la ecuación (4.30) (++) .
32. Complete la solución de la ecuación (4.31) (+).
33. Generalice la expresión anterior para árboles m-rios.

34. Escriba un algoritmo que transforme un árbol m-rio completo, representado con listas, en un arreglo secuencial.
35. Escriba un algoritmo que transforme un árbol binario completo en un arreglo secuencial.
36. Escriba una versión no recursiva del algoritmo que calcula la altura de un árbol binario.
37. Dibuje la arborescencia equivalente al árbol binario de la figura 4.59.

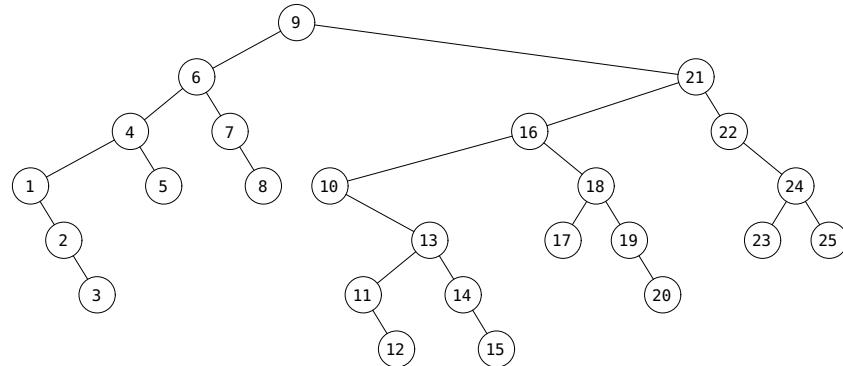


Figura 4.59: Un árbol binario

38. Dibuje el árbol binario equivalente a la arborescencia de la figura 4.60.

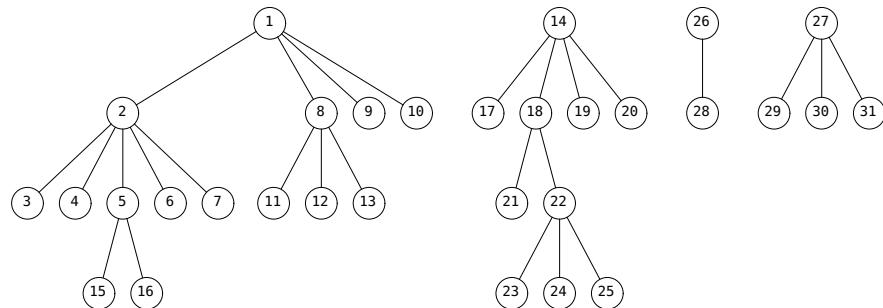


Figura 4.60: Una arborescencia

39. Considere las siguientes líneas del algoritmo presentado en § 4.4.9 (Pág. 250) para la copia de un árbol binario:

```
I(RLINK(tgt_root) == Node::NullPtr);
if (LLINK(tgt_root) != Node::NullPtr)
 destroyRec(LLINK(tgt_root));
delete tgt_root;
```

- (a) ¿Por qué el aserto tiene sentido?
- (b) El if y la destrucción del subárbol izquierdo son redundantes, ¿por qué?
40. Escriba una versión recursiva del algoritmo de destrucción mostrado en § 4.4.10 (Pág. 250) que libere los nodos en prefijo.

41. Escriba una versión recursiva del algoritmo de destrucción mostrado en § 4.4.10 (Pág. 250) que libere los nodos en infijo.
42. Escriba una versión no recursiva, sufijo, del algoritmo de destrucción mostrado en § 4.4.10 (Pág. 250) que, en caso de que ocurra un desborde de pila, deje el árbol en un estado consistente.
43. Escriba un programa que recorra un árbol binario por niveles de derecha a izquierda.
44. Implante el recorrido sufijo mediante pseudohilado (++) .
45. Modifique el recorrido infijo pseudohilado para obtener un recorrido infijo inverso, es decir, el recorrido infijo de derecha a izquierda (+).
46. Diseñe un algoritmo que reciba como entrada el número de Deway de un nodo cualquiera y el árbol binario donde reside el nodo y retorne el nodo en cuestión.
47. Diseñe un algoritmo que tome de entrada un árbol binario y construya un conjunto compuesto por todos los nodos representados en números de Deway.
48. Con base en lo planteado al inicio de § 4.8 (Pág. 302), calcule el número de arborescencias posibles que se pueden plantear con  $n$  nodos.
49. Diseñe un algoritmo que determine si una secuencia de bits corresponde a una palabra de Lukasiewicz.
50. Dado un árbol binario, diseñe un algoritmo que genere la palabra de Lukasiewicz correspondiente.
51. Considere el siguiente prototipo de función:

```
string random_luka(const size_t & m)
```

El cual genera una palabra aleatoria de Lukasiewicz de longitud  $m$ . Escriba la función en cuestión.

52. Demuestre que el lenguaje de Dick es prefijo.
53. El propósito de este ejercicio es generar una arborescencia aleatoria de  $n$  nodos.
  - (a) Demuestre que a través de la generación de un árbol binario aleatorio de  $n$  nodos se puede obtener una arborescencia aleatoria de  $n$  nodos.
  - (b) Asuma que se dispone de la función siguiente:

```
int random(int n)
```

La cual genera un número aleatorio entre comprendido en el intervalo  $[0, n]$   
 Ahora suponga el prototipo de función siguiente:

```
template <class Node> Node * random_tree(int n)
```

La cual genera un árbol binario al azar de  $n$  nodos. Escriba la función en cuestión. Puesto que sólo interesa la forma del árbol, no está permitido usar árboles binarios de búsqueda.

- (c) Considere el siguiente prototipo de función:

```
template <class Node> random_forest(int n)
```

El cual genera una arborescencia aleatoria de  $n$  nodos, y retorna la raíz del primer árbol.

Diseñe un algoritmo que instrumente la generación de una arborescencia aleatoria de  $n$  nodos e instruméntelo.

54. Diseñe un algoritmo que imprima el recorrido por niveles inverso de un árbol binario. Es decir, primero se listan de izquierda a derecha los nodos del último nivel; luego, los del penúltimo y así sucesivamente hasta llegar a la raíz.

55. Dado un árbol binario cualquiera, el recorrido complemento por niveles es definido como el listado de los nodos que faltan por nivel a partir de la raíz hasta el nodo de mayor altura. Por ejemplo, para el árbol de la figura 4.61, el recorrido complemento es 7, 8, 11, 14, 15, es decir, los nodos que faltarían para que el árbol este completo hasta su altura.

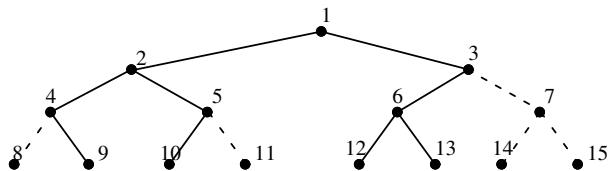


Figura 4.61: Árbol binario ejemplo

(a) Diseñe un algoritmo que calcule el recorrido complemento de un árbol binario. Asuma que los nodos están etiquetados por nivel a partir de la posición cero (+).

(b) Diseñe un algoritmo que genere un árbol complemento. Esto es, un árbol cuyas hojas encajen con los nodos faltantes del árbol para que el árbol esté completo (+).

56. Diseñe un algoritmo que “pode” un árbol binario, es decir, que elimine la mínima cantidad de nodos para volverlo un árbol completo.

57. Diseñe un algoritmo que encuentre el más largo camino por la derecha contenido en un árbol binario.

58. Diseñe un algoritmo que genere un árbol binario de  $n$  nodos donde  $n$  es un valor constante (+).

59. Diseñe un algoritmo que determine si una expresión algebraica tiene sus paréntesis balanceados (+).

60. Diseñe un algoritmo que genere un árbol binario aleatorio de  $n$  nodos. La cantidad  $n$  es un parámetro del algoritmo y todos los árboles deben tener la misma probabilidad (++).
  61. Diseñe un algoritmo que construya un árbol binario de búsqueda a partir de su recorrido sufijo.
  62. Dado un árbol binario de búsqueda, diseñe un algoritmo que construya una secuencia de inserción que reproduzca un árbol binario equivalente.
  63. Diseñe e implante un algoritmo que liste los rangos de claves que no se encuentran en el árbol.
  64. Diseñe un nodo binario que, además de la clave, tenga un enlace doble Dlink. Modifique las rutinas de inserción y eliminación en árbol binario de búsqueda para que el campo Dlink funcione de hilos explícitos. El recorrido de la lista debe corresponder al recorrido infijo en el árbol binario.
  65. Diseñe una versión iterativa, sin usar pila, del procedimiento `insert_in_binary_search_tree()`.
  66. Diseñe una versión iterativa, sin usar pila, del procedimiento `remove_from_search_binary_tree()`.
  67. Diseñe una versión iterativa, sin utilizar pila, del procedimiento `split_key_rec()`.
  68. Escriba una versión de la eliminación de un árbol binario de búsqueda que, cuando se trate de suprimir un nodo completo, escoja al azar cuál será el nodo a seleccionar como raíz del `join_exclusive()`. ¿Cuáles serían las ventajas de este enfoque?
  69. Diseñe una versión iterativa, sin usar pila, del procedimiento `remove_from_binary_tree()`.
- Ayuda: revise los fundamentos explicados en § 4.9.6 (Pág. 324)-1.
70. Demuestre que si no hay claves duplicadas, entonces `join()` y `join_preorder()` son equivalentes. Es decir, producen exactamente el mismo árbol.
  71. Dibuje los árboles binarios de búsqueda correspondientes a la partición del árbol binario de la figura 4.62 según la clave 250.
  72. Dibuje el árbol binario resultante para las siguientes secuencias de inserción:
    - (a) 220 1400 315 534 1223 1339 329 514 1207 1019 182 146 1317 435 646
    - (b) 1207 726 520 1038 10 331 316 31 1203 1277 480 755 1422 1231 574
    - (c) 170 1247 893 411 378 1371 105 186 906 689 747 115 858 453 1079
    - (d) 97 401 815 54 99 583 102 654 155 725 256 737 1193 354 854
    - (e) 649 858 699 148 57 986 1438 1194 824 996 1119 434 15 226 758
    - (f) 346 787 891 1405 1111 1204 1181 496 314 1014 529 1087 481 1146 1343
  73. Dibuje el árbol binario resultante para las siguientes secuencias de inserción en la raíz:
    - (a) 271 1450 171 93 907 123 1341 329 714 231 1079 628 1270 1214 320

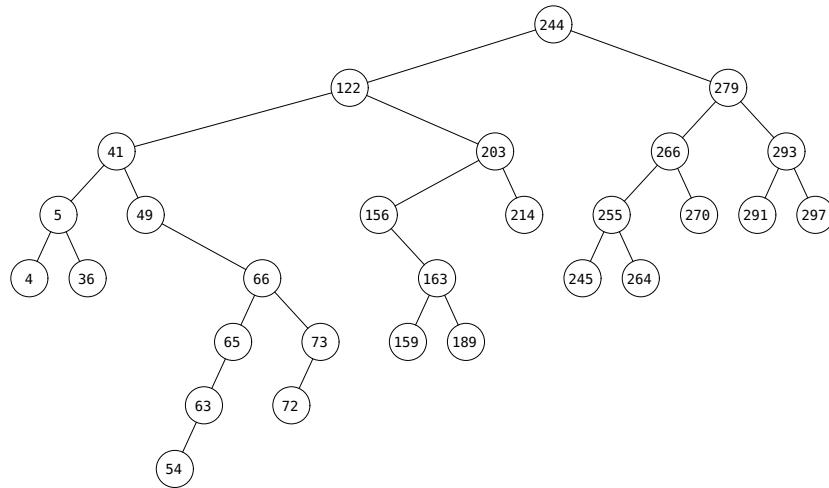


Figura 4.62: Un árbol binario de búsqueda

- (b) 990 505 834 119 329 47 1283 1303 468 970 697 239 596 794 1215
- (c) 1484 145 320 1025 609 932 270 1115 846 1211 1287 1095 855 1059 1343
- (d) 201 607 1137 1099 1197 203 840 741 526 929 1370 190 996 205 715
- (e) 420 78 221 1475 1155 1200 1218 1257 1061 1053 1124 53 665 743 1238
- (f) 429 1074 459 437 1041 979 675 1127 213 664 715 1158 117 844

74. Dados los siguientes recorridos prefijos de árboles binarios de búsqueda:

- (a) 1058 388 100 19 83 126 161 944 549 455 681 599 937 1186 1333.
- (b) 1156 3 185 119 285 225 810 409 783 420 1003 975 1351 1290
- (c) 868 527 240 11 230 562 580 605 741 972 901 1450 1270 1164 1431
- (d) 898 206 173 151 136 43 192 218 850 700 560 575 1091 1474 1325
- (e) 112 60 80 1120 625 139 277 266 942 909 981 967 1150 1439 1354

- (a) Dibuje el árbol correspondiente.
- (b) Calcule el IPL.

75. Dados los siguientes recorridos sufijos de árboles binarios de búsqueda:

- (a) 123 478 408 947 1009 831 303 1250 1363 1352 1206 1487 1369 1043 230
- (b) 15 116 512 1014 342 1049 1085 1028 1267 1294 1288 1117 1468 1340 47
- (c) 48 84 408 141 898 815 798 619 488 26 1013 1341 1410 1142 1032
- (d) 230 318 315 109 433 368 367 558 576 461 327 1193 801 1288 733
- (e) 30 83 193 456 389 495 1056 1001 565 1275 1495 1422 558 521 103

- (a) Dibuje el árbol correspondiente.
- (b) Calcule el IPL.
- (c) Demuestre que la sumatoria planteada en (4.34) (§ 4.7.4 (Pág. 289)) es, en efecto,  $\mathcal{O}(n)$ .

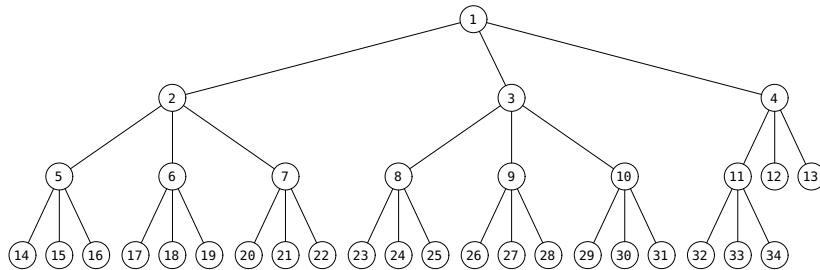


Figura 4.63: Un heap trinario

76. Considere un heap trinario, es decir, un árbol trinario completo (con la propiedad triangular) tal como el de la figura 4.63.

Y, además, con la propiedad de orden, o sea, que para cada clave del árbol, las claves descendientes siempre son mayores.

Considere la siguiente operación sobre un nodo cualquiera  $p$  de un heap trinario:

```
Node * child(Node * p, int i)
```

$p$ , como ya se dijo, es un nodo de un heap trinario.  $i$  es el ordinal del hijo; es decir:

- (a)  $i == 0$ : el hijo más a la izquierda.
- (b)  $i == 1$ : el hijo del centro.
- (c)  $i == 2$ : el hijo más a la derecha.

Puede asumir, aunque eso no es esencial al problema, que en ausencia de un hijo  $\text{child}()$  retorna `NULL`.

El objetivo de este problema es descubrir un algoritmo que encuentre la secuencia de ordinales, parámetros a  $\text{child}()$ , tales que a través de llamadas sucesivas se encontrará el último elemento del heap trinario. Por ejemplo, para la figura anterior, la secuencia es 2, 0, 2, la cual, partiendo desde la raíz 1, permite  $\text{child}(1, 2) = 4 \rightarrow \text{child}(4, 0) = 11 \rightarrow \text{child}(11, 2) = 34$ , las cuales desembocan en el nodo etiquetado con 34.

77. Generalice el problema anterior para heaps  $n$ -rios.
78. Explique cómo puede ordenarse una lista doblemente enlazada mediante el heapsort con consumo de espacio  $\mathcal{O}(1)$ .
79. Con base en la reflexión del ejercicio anterior, implante el heapsort sobre listas de tipo `Dnode<T> (+++)`.
80. Diseñe un algoritmo que recorra una secuencia de  $n$  elementos y encuentre los  $m \ll n$  menores elementos. Explique detalladamente cómo se utiliza el heap para contener y mantener, exactamente,  $m$  elementos a medida que se inspecciona la secuencia.
81. Diseñe un TAD `DynArrayHeap<T>` fundamentado en el TAD `DynArray<T>` (§ 2.1.4 (Pág. 34)). Discuta los tamaños adecuados del directorio, segmento y bloque. Decida un criterio bueno para cortar el arreglo de manera tal que el consumo de memoria sea mínimo.

82. Modifique el TAD BinHeap<Key> para que cada nodo prescinda del apuntador al padre.
83. Modifique el TAD BinHeap<Key> para prescindir de la lista de hojas. Utilice permanentemente una secuencia que albergue el número de Deway de last, de manera tal que cada modificación del heap la use para ubicar los nodos a actualizar.
84. Demuestre por inducción que un heap de  $n$  de nodos tiene  $\lceil n/2 \rceil$  hojas.
85. Considere una lista enlazada representada mediante el TAD Dnode<int>. Considere la siguiente interfaz:

```
Dnode<int> * los_mas_grandes(Dnode<int> & l, int m)
```

La cual retorna una nueva lista contentiva de las copias de los  $m$  mayores números almacenados en  $l$ .

Diseñe e instrumente la rutina en cuestión de la manera más eficiente sin que se tenga que ordenar el la lista.

86. Diseñe un algoritmo que tome como entrada el recorrido prefijo e infijo de un árbol y retorne como salida el recorrido por niveles. No se debe construir el árbol.
87. Diseñe un algoritmo que reciba como entrada el recorrido prefijo de un árbol binario de búsqueda y retorne su IPL. No se debe construir el árbol.
88. Diseñe un algoritmo que reciba como entrada el recorrido sufijo de un árbol binario de búsqueda y retorne su IPL. No se debe construir el árbol.
89. Diseñe un algoritmo que cuente la cantidad de nodos en el  $i$ -ésimo nivel de un árbol binario (+).
90. Diseñe un algoritmo que cuente la cantidad de nodos incompletos no hojas de un árbol binario.
91. Diseñe un algoritmo que cuente la cantidad de nodos llenos de un árbol binario.
92. Suponga que cada nodo posee un campo adicional next. Escriba un algoritmo que enlace todos nodos del árbol según el recorrido infijo.
93. Para los siguientes pares de árboles representados con sus recorridos prefijos, construya el árbol producto de la operación join().
  - (a) • 144 65 27 22 54 38 74 68 71 95 85 77 123 108 122
    - 5 138 46 21 33 70 55 125 86 78 76 72 109 129 143
  - (b) • 124 25 23 20 19 11 21 106 82 31 81 50 76 121 125
    - 33 29 51 119 97 84 64 68 92 89 90 110 108 149 142
  - (c) • 12 25 52 32 76 123 100 77 81 80 83 104 118 138 136
    - 62 37 31 9 26 46 61 131 110 84 69 105 116 129 143
  - (d) • 40 13 11 4 19 24 122 66 64 43 54 74 77 115 146
    - 33 23 22 2 18 7 21 39 53 44 87 63 106 118 135
  - (e) • 51 18 8 9 29 101 89 72 66 76 80 148 105 115 139

- 5 128 85 54 23 45 71 57 123 103 91 95 119 126 140
  - (f) • 123 47 45 15 33 112 97 96 80 107 105 109 121 122 149  
 • 69 20 9 19 29 53 66 138 94 82 81 78 106 95 142
94. Dados 2 árboles binarios  $T_1$  y  $T_2$ , se define la relación “es simétrico”, denotada como  $T_1 \simeq T_2$  como:
- $$T_1 \simeq T_2 \Leftrightarrow \begin{cases} T_1 = T_2 = \emptyset \Rightarrow T_1 \simeq T_2 \vee \\ T_1 \neq T_2 \neq \emptyset \Rightarrow *raiz(T_1) \simeq *raiz(T_2) \wedge L(T_1) \simeq R(T_2) \wedge L(T_1) \simeq R(T_2) \end{cases}$$
- Diseñe un algoritmo que tome como entrada dos árboles y determine si son simétricos.
95. Diseñe e implante una versión iterativa, sin usar pila, del algoritmo `inorder_position()`.
96. Modifique la primitiva `insert_in_pos()` explicada en § 4.11.3 (Pág. 338) para que valide que la clave del nodo a insertar no viole la propiedad de orden de un ABB.
97. Diseñe un algoritmo recursivo que calcule la posición infija de una clave del árbol.
98. Dado un árbol binario de búsqueda con rangos y una clave de búsqueda, diseñe un algoritmo que retorne la posición infija de la clave si ésta se encuentra en el árbol, o la posición infija que tomaría la clave después de la inserción.
99. Implante un algoritmo recursivo que efectúe la unión general de dos ABBS.
100. Implante un algoritmo iterativo, que no use pila, que haga la unión general de dos ABBS.
101. Diseñe un TAD basado en árboles con rango que modele arreglos eficientes.
102. Diseñe una versión iterativa, sin usar pila, del algoritmo `split_key_rec()` y que deje el árbol intacto si la clave ya se encuentra en el árbol (++).
103. Diseñe una versión iterativa de la función `split_pos_rec()` explicada en § 4.11.6 (Pág. 340).
104. Diseñe una rutina que busque una clave en un árbol binario y, en caso de encontrarla, rote el nodo hasta la raíz.
105. Demuestre que el árbol binario producido por `insert_in_binary_search_tree()` con la secuencia de inserción  $S = < k_1, k_2, \dots, k_n >$  es exactamente igual al producido por `insert_root()` sobre la secuencia invertida  $\bar{S} = < k_n, k_{n-1}, \dots, k_1 >$ .
106. Diseñe un algoritmo recursivo, basado en rotaciones, que efectúe la inserción por la raíz en un ABB. El algoritmo debe buscar el nodo externo que albergaría al nuevo nodo y después rotar recursivamente a su padre hasta que éste devenga en raíz.
107. (Sugerido por Andrés Arcia) Explique la operación que ejecuta la siguiente rutina:

```
Node * program(Node * root)
{
 if (root == NullPtr)
 return NullPtr;
```

```

 while (RLINK(root) != NullPtr)
 root = rotate_to_left(RLINK(root));
 program(LLINK(root));
 return root;
 }
}

```

108. Diseñe una rutina que seleccione un nodo en la  $i$ -ésima posición y lo rote el nodo hasta la raíz.
  109. Demuestre que las primitivas `insert_root_rec()` y `insert_root()` siempre producen árboles equivalentes.
  110. Dado un árbol binario de búsqueda y el algoritmo de inserción presentado en § 4.9.3 (Pág. 320), calcule la varianza sobre el número de nodos visitados en una búsqueda  $(++)$ .
  111. ¿Por qué el valor de `Node::MaxHeight = 80` es adecuado para árboles binarios de búsqueda?
  112. Dado un árbol binario de búsqueda y el algoritmo de inserción en la raíz presentado en § 4.9.7 (Pág. 326), calcule las esperanzas y varianzas de las cantidades de nodos visitados en búsquedas exitosas e infructuosas  $(++)$ .
  113. Dado un ABB, diseñe un algoritmo eficiente que extraiga un rango de claves entre  $i$  y  $j$ ,  $j > i$   $(++)$ .
  114. Dados dos árboles ABB  $A_1$  y  $A_2$ , diseñe un algoritmo eficiente que inserte  $A_1$  a partir de la posición  $i$  de  $A_2$ .
  115. Considere la primitiva `remove_by_pos()` explicada en § 4.11.9 (Pág. 342). Diseñe una versión basada en eliminación de la raíz y concatenación de los subárboles restantes.
  116. Modifique los algoritmos de inserción, eliminación, concatenación y partición para que funcionen en árboles binarios de búsqueda extendidos.
  117. Diseñe un algoritmo iterativo eficiente que efectúe la concatenación de dos árboles binarios  $(+)$ .
  118. Diseñe un algoritmo iterativo eficiente que efectúe la unión de dos árboles binarios  $(+)$ .
  119. Diseñe un algoritmo iterativo eficiente que efectúe la intersección de dos árboles binarios.  $(+)$
  120. Diseñe un algoritmo recursivo eficiente que efectúe la diferencia de dos árboles binarios  $T_1$  y  $T_2$ . Esto es,  $T_1 - T_2$  debe retornar el árbol binario  $T_1$ , excepto aquellas claves que se encuentran en  $T_2$ .
  121. Escriba un programa que inserte las  $n$  claves de manera que el árbol quede equilibrado  $(+)$ .
  122. Encuentre una expresión analítica que genere la sucesión de inserciones para que el árbol quede equilibrado  $(++)$ .
- Ayuda:** Considere  $n = 2^k$ , es decir,  $n$  es una potencia exacta de dos.

123. Escriba un algoritmo que efectúe la unión de dos árboles binarios de búsqueda extendidos. El árbol resultante no debe tener claves repetidas.
124. Escriba un algoritmo que efectúe la intersección de dos árboles binarios de búsqueda extendidos. El árbol resultante no debe tener claves repetidas.
125. Modifique las rutinas `insert_root_rec()` e `insert_root()` presentadas en 4.9.7 para que manejen árboles binarios extendidos.
126. Efectúe medidas de rendimiento para todos los recorridos implantados en `BinNode_Utils`. Dé un reporte detallado de sus experimentos que señale por cada versión de recorrido el número de nodos y el tiempo total de recorrido.
127. Diseñe e implante un tipo abstracto de dato que represente un árbol binario que ocupe espacio mínimo. Es decir, la implantación debería manejar los tres tipos de nodos señalados en § 4.4.15 (Pág. 255)-1: completo, incompleto y hoja.
128. Diseñe un TAD de uso general que modele árboles *n*-rios basados en el TAD `DynArray<T>`.
129. Dadas  $n$  claves, demuestre que existen  $\frac{n(n_1)}{2}$  árboles binarios diferentes que almacenan las  $n$  claves.
130. Dado el árbol de Huffman de la figura 4.64, decodifique la siguiente cadena de bits:

```

0 0 0 1 0 1 0 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 1 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 0 1 0 1 1 0 1 0 1 0 0 0 1 0
1 1 1 0 1 1 0 1 1 1 1 0 0 1 0 1 0 0 1 1 1 1 1 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 0 1 1 0 1 0 1 1 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 1
1 1 1 1 0 1 1 0 1 1 1 1 0 0 0 1 0 1 1 0 0 1 1 0 0 1 0 0 0 1 0 1 1 0 1 0 1 1 0 0 0 1 0 1 0 1 1 1 1 0 1 0 0 0 1 1 1 1 1 1 1 0
0 1 0 1 1 1 1 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 0 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 1 0 0 0 1 1 1 1 0 0 1 0 0 1 1 1 0 0 1 0
0 1 1 1 1 0 1 1 1 1 1 1 0 0 0 1 1 1 0 0 1 1 0 0 0 1 0 0 0 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 0 0 1 1 0 0 0 0 0 1 0 1 1 1 1 0 0 0 1
0 0 1 0 1 1 0 1 0 0 1 0 1 0 0 1 0 1 1 1 1 0 1 0 0 0 0 1 1 1 1 1 0 1 0 0 0 1 0 1 0 1 1 1 1 0 1 1 0 0 1 0 0 0 1 1 1 0 0
0 1 0 1 1 1 1 0 1 1 0 0 0 1 1 0 0 0 1 1 0 1 0 0 1 1 1 1 0 1 0 1 1 1 1 1 0 1 0 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 0 1 1 0 0
0 0 1 0 0 0 0 0 1 0 0 1 1 0 1 0 0 1 1 0 1 0 0 1 0 1 0 1 1 1 1 0 1 1 1 0 0 0 1 0 1 0 0 1 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 1 0 0 1 1 1 1 1 0 0 1 0 1 0 1 1 1 0 0 0 0 0 0 1 1 0 1 0 0 1 1 1 1 0 0 0 0 1 1 1 0 1 0 1 0 0 1 0 1 1 1 0 1 0
0 1 0 1 1 1 0 1 0 1 1 0 0 0 1 0 1 0 0 1 0 1 0 1 1 1 1 0 1 0 1 0 1 0 1 1 1 1 0 1 0 0 0 1 0 1 0 1 1 1 1 0 1 0 0 1 0 1 1 1 0 1 0
1 0 0 1 0 1 0 0 0 1 1 0 0 1 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 1 0 0 1 1 1 1 1 0 0 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 1 0 0 1 1 1 1 0 0 0 0 1 1 0 1 0 1 0 1 1 1 0 1 0 1 1 1 1 0 1 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 0 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 0 0 1 1 0 1 0 1 0 0 1 0 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 0 1 0 0 1 0 1 0 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 1 1 0 0 0 1 1 0 1 1 1 0 0 0 1 0 1 1 1 0 1 0 1 1 1 1 1 0 0 0 0 1 0 0 1 0 1 1 1 1 0 0 1 1 0 0 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1
1 1 1 1 0 0 0 0 1 1 1 0 1 0 1 1 1 1 0 1 1 1 1 0 0 1 1 1 1 1 0 0 1 0 0 1 0 1 1 1 1 0 1 0 1 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1
0 1 1 0 0 0 1 0 1 1 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 1 1 1 0 0 0 0 1 0 0 1 0 0 1 0 1 1 1 1 0 0 0 1 1 1 1 0 0 1 0 1 1 1 0 1 0 1
0 0 1 0 1 0 0 1 0 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 1 1 1 0 0 0 0 1 0 0 1 0 0 0 1 0 1 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1
0 1 0 0 0 0 0 1 1 1 0 1 0 0 1 0 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1 0 0 1 0 1 0 0 0 0 0 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1
0 0 1 0 1 1 1 0 1 1 1 0 1 0 0 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1 0 0 1 0 1 0 0 0 0 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1
0 1 1 0 0 1 0 1 0 0 1 0 1 1 1 1 0 0 0 1 1 1 0 0 1 0 1 1 1 1 0 0 1 0 0 1 0 0 0 0 1 0 0 1 1 1 1 0 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1
0 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 0 1 0 1 1 1 1 0 0 1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 0 1 0 1 1 1 1 0 1 0 1 1 1 0 1 0 1
0 0 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 1 0 0 1
1 1 0 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 1 0 0 1 0 0 1 1 1 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 0 0 1 0 1
0 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 0 1 0 1 1 1 1 0 0 1 0 1 1 1 1 0 0 1 0 1 1 1 1 0 0 1 0 1 1 1 1 0 0 1 0 1 1 1 1 0 0 1 0 1
0 0 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 1 0 0 1
1 1 0 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 1 0 0 1 0 0 1 1 1 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 0 0 1 0 1

```

```
0 1 1 1 1 0 0 0 1 1 0 1 1 0 0 0 0 1 1 0 0 1 0 0 1 0 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 1 0 1 1 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 1
1 1 1 1 0 0 1 1 0 1 0 1 1 0 1 0 1 1 1 0 0 0 0 0 1 1 0 1 0 0 1 1 1 0 0 0 0 1 1 1 0 1 0 1 1 0 1 0 1 1 1 0 1 1 1 0 0 1 0 1
1 1 1 1 1 0 1 1 1 0 0 1 0 1 1 0 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 0 1 0 1 0 1 0 1 0 1 1 1 0 0 1 0 0 0 1 0 1 1 0 1 0 0 1 0 1
1 0 1 1 0 1 0 0 1 1 1 0 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0 1 0 0 1 1 0 1 0 1 1 1 1
0 1 0 1 0 0 1 0 1 0 1 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 1 1 1 0 1 1 0 0 1 1 1 1 1 1 0 0 1 0 0 0 1 1 0 1 1 1 0 0 0 0 0 1
0 1 0 1 0 0 0 1 1 1 0 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 0 1 1 0 1 1 1 0 0 0 0 1
1 0 0 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 1 0 1 1 1
0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 1 1 1 1 0 1 1 1 1 1 1 0 0 0 1 1 1 0 1 0 1 0 0 1 1 1 0 0 0 0 1 1 0 0 1 1 0 1 0 1 0 1 1 1 1 1
1
0 1 1 0 0 1 1 0 1 0 1 1 1 1 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 0 0
```

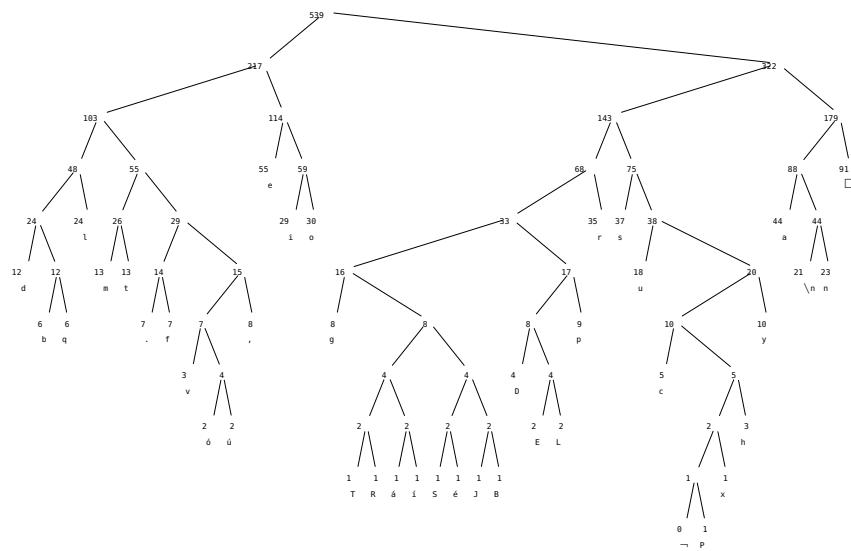


Figura 4.64: Un árbol de Huffman

131. Sea  $S$  una secuencia codificada según el algoritmo de Huffman. Diseñe un algoritmo que lea la secuencia, distinga los prefijos distintos y retorne el árbol de Huffman correspondiente a los prefijos (+).

132. ¿Cuál es la equivalencia entre una palabra de Dick y un número de Dewey?

133. Calcule el árbol óptimo para los siguientes conjuntos de claves y probabilidades:

  - (a) 10 17 30 31 63 68 74 80 88 93 99 103 114 143 14  
0.00003 0.00046 0.00320 0.01389 0.04166 0.09164 0.15274 0.19638 0.19638 0.15274 0.09164  
0.04166 0.01389 0.00320 0.00046
  - (b) 17 39 42 53 54 66 75 85 89 90 96 98 115 119 143  
0.00047 0.00470 0.02194 0.06339 0.12678 0.18594 0.20660 0.17708 0.11806 0.06121 0.02449  
0.00742 0.00165 0.00025 0.00002
  - (c) 6 8 12 40 59 62 65 69 90 103 109 118 128 134 138  
0.00475 0.03052 0.09156 0.17004 0.21862 0.20613 0.14724 0.08113 0.03477 0.01159 0.00298  
0.00058 0.00008 0.00001 0.00000
  - (d) 18 21 34 41 53 54 56 57 65 77 98 104 122 124 131  
0.00000 0.00000 0.00001 0.00008 0.00058 0.00298 0.01159 0.03477 0.08113 0.14724 0.20613  
0.21862 0.17004 0.09156 0.03052

134. Calcule las distribuciones de probabilidad para cada uno de los árboles de la pregunta anterior.
135. Los árboles estáticos óptimos presentados en § 4.14 (Pág. 358) asumen que las claves a buscar siempre están presentes. Considere el problema de construir un árbol óptimo en el cual es posible efectuar búsquedas infructuosas. En este caso, se define un arreglo adicional  $q$  de dimensión  $n + 1$  donde cada  $q_i$  representa la probabilidad de que una clave de búsqueda esté comprendida entre  $k_i$  y  $k_{i+1}$  tal que  $\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$ .
- Para esta variación del problema, defina la longitud del camino ponderada.
  - Modifique el algoritmo presentado en § 4.14 (Pág. 358) para que se consideren búsquedas infructuosas.
- Ayuda: Considere que la búsqueda infructuosa es realizada hacia un nodo externo (+).
136. (**Tomado y traducido de Lewis-Denenberg [108]**) Segundo los términos presentados en § 4.14 (Pág. 358), sea  $\text{TREE}(j, k)$ ,  $1 \leq j \leq k \leq n$  la raíz del árbol binario óptimo para el conjunto de claves  $k_j, \dots, k_k$ .
- Demuestre que si  $1 \leq j \leq k \leq n$ , entonces existen árboles óptimos tal que  $\text{TREE}(j, k - 1) \leq \text{TREE}(j, k) \leq \text{TREE}(j, k)$ . Intuitivamente, esto quiere decir que añadir una nueva clave al extremo derecho de la secuencia no causa que la raíz del árbol óptimo se desplace hacia la derecha. Análogamente, sucede lo mismo si la clave es insertada por el extremo izquierdo.
  - Demuestre que la búsqueda del subárbol óptimo es restringida a  $\text{TREE}(j, k - 1) \leq \text{TREE}(j, k) \leq \text{TREE}(j, k)$ , entonces, para cada valor de  $i$  del algoritmo presentado en § 4.14 (Pág. 358), el `for (j = ...)` es  $\mathcal{O}(n)$  y, por ende, el algoritmo global es  $\mathcal{O}(n^2)$ .
- Ayuda: No se puede efectuar la prueba si se busca un límite para el `for (j = ...)` y entonces multiplicar por  $n$ . En su lugar, deben sumarse por separado las longitudes de los rangos a buscar.

# 5

## Tablas hash

La tecnología ha transformado la faz del mundo físico, común e indispensable para la vida, que comprende a esa totalidad llamada planeta. Podríamos decir que hasta hace no menos de un siglo, toda persona asociaba lo natural a la naturaleza, es decir, al mundo lleno de aire, tierra y agua donde se distinguían los tres grandes reinos: mineral, vegetal y animal, y en el cual apenas se notaban parcelas de lo humano. Hoy en día, prácticamente no hay espacio del planeta que no esté permeado por lo humano, a tal extremo que para muchos, una estructura de concreto, con la que se convive desde el nacimiento, les parece completamente natural.

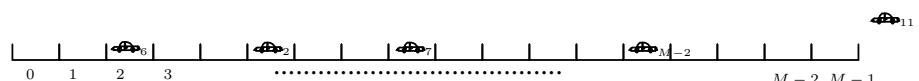
Una mirada aérea sobre la faz terrestre nos proporciona una buena idea acerca de cuánto la tecnología ha transformado el mundo. La vista está delineada por polígonos y líneas regulares, a pesar de que éstos partieron de abstracciones matemáticas, que no son naturales.

De los diversísimos objetos tecnológicos de hoy en día, uno que ha contribuido decisivamente a ese cambio del mundo lo encarna el automóvil. Detengámonos un breve momento y meditemos cómo sería el mundo si no existiese este artefacto.

El automóvil, que hoy nos es “natural”, es un medio de transporte esencial para nuestra forma de vida. Y, sin embargo, es impresionante cuán alienante nos puede ser. De manera un poco simplista podemos decir que cuando un automóvil no está en movimiento, éste ocupa un espacio. Toda actividad humana que cuente con el automóvil particular como medio de transporte, en detrimento del transporte público, debe considerar el estacionamiento. Los centros comerciales de hoy día, iconos de un tipo socialmente particular de “buena vida”, indicadores del consumismo desenfrenado de esta época y característica peculiar de un estrato social, son buenos ejemplos.

Un problema de interés cuando se diseña una obra civil en un ambiente donde no existe transporte público -cuestión que debería de ser muy excepcional- consiste en interrogar por la cantidad de superficie para el estacionamiento. Un primer paso consiste en estimar la cantidad esperada de automóviles y así decidir la capacidad en lugares del estacionamiento.

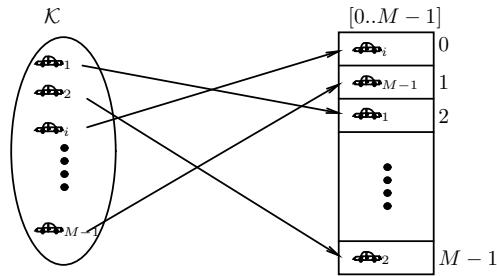
*Grosso modo*, un estacionamiento puede pictorizarse del siguiente modo:



Genéricamente, el estacionamiento tiene  $M$  puestos posibles numerados secuencialmente. Los automóviles arriban secuencialmente y se estacionan en el primer puesto disponible. El conocimiento consolidado recomienda que tal capacidad arroje un porcentaje de plenitud

entre el 75% y 90%. Es decir, según lo esperado, debe haber un sobrante entre el 25 % y 10 %.

¿Qué tiene que ver la situación del estacionamiento con los algoritmos y estructuras de datos? Consideremos un conjunto genérico  $K$  de claves, dentro de un posible dominio  $\mathcal{K}$ , y un estacionamiento con capacidad  $M$  que albergaría las claves. Ahora consideremos la existencia de una función inyectiva<sup>1</sup>  $h(k) : \mathcal{K} \rightarrow [0, M - 1]$ , la cual puede pictórizarse así:



Dicho lo anterior, podemos introducir formalmente el concepto de “tabla hash” o de “tabla de dispersión”<sup>2</sup>.

**Definición 5.1 (Tabla Hash)** Sea un conjunto de claves  $K$  pertenecientes a un dominio  $\mathcal{K}$ . Una tabla hash se define como:

1. Un arreglo de tipo  $\mathcal{K}$  de  $M$  entradas. Por lo general,  $|K| \neq M$ , aunque podrían coincidir.
2. Una función  $h(k) : \mathcal{K} \rightarrow [0, M - 1]$  llamada “función hash”.

A un registro albergante de una clave en una tabla hash se le denomina “cubeta” o su equivalente anglosajón “bucket”.

Mediante el arreglo y la función  $h(k)$  podemos usar una tabla hash para implantar el problema fundamental de estructuras de datos. Sea  $A[M]$  un arreglo de pares de tipo  $\mathcal{K} \times \{0, 1\}$ ; es decir,  $A[i].key$  indica la clave, mientras que  $A[i].busy$  indica si la entrada está ocupada o no. Entonces, *grossos modo*, las tres operaciones básicas se efectúan del siguiente modo:

**Inserción de clave  $k$ :**

1.  $i = h(k) \bmod M$
2.  $A[i].key = k$
3.  $A[i].busy = 1$

**Búsqueda de clave  $k$ :**

1.  $i = h(k) \bmod M$
2. `if (A[i].busy == 1) return true; else return false;`

---

<sup>1</sup>  $h : \mathcal{D} \rightarrow \mathcal{R}$  se dice que es inyectiva  $\iff \forall x, y \in \mathcal{D} \implies h(x) \neq h(y)$ . Una función inyectiva también es llamada “uno a uno”.

<sup>2</sup> De por sí, esta ha sido la decisión de algunos traductores al castellano. En el caso concreto, la traducción castellana [8] de “Data Structures and Algorithms de Aho, Hopcroft y Ullman” [7].

Eliminación de clave  $k$ :

1.  $i = h(k) \bmod M$
2.  $A[i].busy = 0$

Notemos que ninguna de las operaciones contiene iteraciones. Por tanto, el coste de las tres operaciones depende directamente del coste de  $h(k)$ . Consecuentemente, si  $h(k)$  es  $\mathcal{O}(1)$ , entonces las tres operaciones son  $\mathcal{O}(1)$ . He aquí, pues, la grandeza de esta estructura de datos.

Si  $h(k)$  no es inyectiva, entonces es posible que a dos claves distintas  $k_i, k_j \in \mathcal{K}$  la función hash les asigne el mismo valor, o sea,  $h(k_i) = h(k_j)$ . En este caso, decimos que existe una “colisión”. La mayoría de las veces no conocemos con exactitud el conjunto de claves  $K \subset \mathcal{K}$ , lo cual hace más posible una colisión.

Así las cosas, para que una tabla hash con una función  $h(k)$  no inyectiva satisfaga operaciones en  $\mathcal{O}(1)$ , ésta debe contemplar dos requerimientos esenciales:

1. El cómputo de la función  $h(k)$  debe hacerse en  $\mathcal{O}(1)$  con un muy poco coste constante y su comportamiento debe emular a una distribución discreta uniforme entre  $[0, M - 1]$ .

Puesto que pueden haber colisiones, éstas deben reducirse al mínimo posible; este es el sentido de que  $h(k)$  emule la aleatoriedad.

2. Por la misma posibilidad de tener colisiones, aunque fuese mínima, se debe tener una estrategia para manejarlas, es decir, se debe decidir, sistemáticamente, qué hacer con aquellas nuevas claves que colinden con otras ya presentes en la tabla. La estrategia debe ser tal que no sacrifique el tiempo  $\mathcal{O}(1)$  clamado para una tabla hash.

## 5.1 Manejo de colisiones

Como ya lo hemos señalado, si no se conoce el conjunto de claves que se insertarán en la tabla, entonces son posibles las colisiones. ¿Qué tan probables son? Una mirada a un célebre problema nos indica su probabilidad.

### 5.1.1 El problema del cumpleaños

Aprenderemos la probabilidad de colisión mediante un clásico problema de la teoría de probabilidades. Supongamos un grupo de  $n$  personas, ¿qué tan probable es que dos personas tengan la misma fecha de cumpleaños?

Algunos textos sobre probabilidades ofrecen una solución precisa a este problema conocido como el “del cumpleaños” o la “paradoja del cumpleaños” [3, 4]. Por economía en simplicidad y espacio, aquí lo abocaremos mediante una vía más corta, aunque más imprecisa, definida por Cormen *et al* en su enciclopédico texto *Introduction to Algorithms* [32].

Sea una  $x_{i,j}$  una variable aleatoria definida del siguiente modo:

$$x_{i,j} = \begin{cases} 0 & \text{si las personas } i \text{ y } j \text{ no cumplen años el mismo día} \\ 1 & \text{si las personas } i \text{ y } j \text{ cumplen años el mismo día} \end{cases} \quad (5.1)$$

Ahora definamos la siguiente variable aleatoria:

$$X = \sum_{\forall i \neq j} x_{i,j} \quad (5.2)$$

Cuya esperanza  $E(X)$ , en virtud de las definiciones planteadas, nos proporciona la cantidad esperada de pares de personas que cumplen años el mismo día.

Por definición:

$$E(X) = E\left(\sum_{\forall i \neq j} x_{i,j}\right); \quad (5.3)$$

la cual puede plantearse como:

$$E(X) = \sum_{\forall i \neq j} E(x_{i,j}). \quad (5.4)$$

Esto nos plantea la necesidad de calcular  $E(x_{i,j})$ , la cual es, por definición, así:

$$E(x_{i,j}) = 0 \times P(x_{i,j} = 0) + 1 \times P(x_{i,j} = 1) = P(x_{i,j} = 1). \quad (5.5)$$

La cuestión requiere, pues, calcular  $P(x_{i,j} = 1)$ ; es decir, la probabilidad de que dos personas particulares cumplan años el mismo día y ésta, a la incredulidad de algunos, es:

$$P(x_{i,j} = 1) = \frac{1}{365}.$$

No sorprendentemente, algunos incurren en el error de pensar que se trata de una probabilidad condicional, lo cual no es el caso. Para aprehenderlo, imaginémosnos que nos encontramos en un grupo y nos preguntamos ¿cuán probable es que tú cumplas años el mismo día que yo? La respuesta es  $1/365$ , pues el conocer mi fecha de cumpleaños revela el suceso de interés probable entre todos los 365 posibles.

Así pues:

$$E(X) = \binom{n}{2} \frac{1}{365} = \frac{n(n-1)}{730}. \quad (5.6)$$

Lo cual, al plantear la desigualdad  $E(x_{i,j}) \geq 1$ , nos arroja como raíces:

$$n_1 = -\frac{\sqrt{2921} - 1}{2}, \quad n_2 = \frac{\sqrt{2921} + 1}{2} \approx 27,523138.$$

Esto implica que para un grupo de 28 personas es esperado encontrar un par con la misma fecha de cumpleaños; para el doble de personas ( $n = 56$ ,  $E(X) = \frac{56 \times 55}{730} = 4.2192$ ) podemos esperar cuatro pares de personas con el mismo cumpleaños.

Para un suceso en apariencia aleatorio e independiente, tal como la fecha de nacimiento, la probabilidad de colisión es bastante alta, lo cual indica la necesidad de prepararse a su eventualidad. En lo que sigue discutiremos varias estrategias para almacenar colisiones en una tabla hash.

### 5.1.2 Estrategias de manejo de colisiones

Consideremos un conjunto genérico de claves colindantes  $K_c = \{k_1, k_2, \dots, k_n\}$ , o sea,  $\forall k_i, k_j \in K_c \implies h(k_i) = h(k_j)$ . Se conocen dos grupos de técnicas para lidiar con conjuntos de colisiones:

**Encadenamiento** : las colisiones se enlazan en una o varias listas enlazadas fuera o dentro de la misma tabla.

**Direccionamiento abierto**<sup>3</sup>: las colisiones se ubican directamente dentro de la propia tabla.

Cada una de estas técnicas tiene sus ventajas y desventajas y se adecúa su uso.

### 5.1.3 Encadenamiento

La idea fundamental del encadenamiento es que las colisiones se guarden en una lista simple o doblemente enlazada. Cuando los nodos de la lista se guardan fuera de la tabla, decimos que se trata de encadenamiento separado. En el sentido inverso, cuando los propios nodos de la lista son parte de la tabla, decimos que se trata de encadenamiento cerrado.

#### 5.1.3.1 Encadenamiento separado

En su forma más simple, el encadenamiento separado se implanta mediante una tabla de punteros a listas simplemente enlazadas, cuya representación puede visualizarse como en la figura 5.1, la cual muestra un conjunto de 12 claves  $k_i$  donde  $i$  representa el orden de inserción. Cada entrada en el arreglo es un apuntador al primer elemento de la lista enlazada de colisiones.

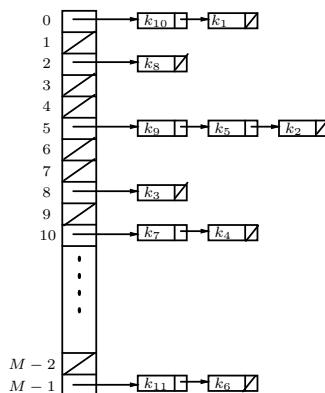


Figura 5.1: Un ejemplo de tabla hash con resolución de colisiones por encadenamiento separado

Todas las operaciones requieren calcular  $j = h(k) \bmod M$ , donde  $k$  es una clave; de esta manera, la entrada  $\text{tabla}[j]$  es el puntero a la lista enlazada. Un valor de apuntador nulo indica que la lista está vacía, es decir, que no hay ningún elemento asociado al índice  $j$  en el arreglo. De lo contrario existe una lista enlazada de claves colindantes encadenadas. En el ejemplo la entrada 11 muestra dos claves  $k_7, k_4$  cuyo valor de función hash arrojó el mismo resultado (10).

<sup>3</sup>Esta es la traducción literal del término usado en inglés “open addressing”.

En *ALÉPH* este tipo de tabla está implantado en el archivo *<tpl\_lhash.H 384a>*, cual implanta y exporta varias clases parametrizadas, de las cuales, la más importante es:

384a *<tpl\_lhash.H 384a>≡*

```
template <typename Key, class BucketType, class Cmp>
class GenLHashTable
{
 typedef BucketType Bucket;

<Prototipo de puntero a función hash 385a>
<Miembros dato de GenLHashTable 385b>
<Métodos públicos de GenLHashTable 386a>
};

<Cubetas de LHashTable<Key> 384b>
```

*GenLHashTable<Key, BucketType>* implanta todo el manejo genérico de la tabla hash, en la cual, las cubetas son de tipo *BucketType*.

### Cubetas

La manera más simple de manejar una cubeta a encadenar separadamente es mediante el tipo *Dnode<T>* estudiado en § 2.4.8 (Pág. 83):

384b *<Cubetas de LHashTable<Key> 384b>≡*

(384a) 384c▷

```
template <typename Key>
class LhashBucket : public Dnode<Key>
{
 LhashBucket(const LhashBucket & bucket) : Dnode<Key>(bucket) {}

 LhashBucket() {}

 LhashBucket(const Key & key) : Dnode<Key>(key) {}

 Key & get_key() { return this->get_data(); }
};
```

Uses *Dnode* 83a.

El empleo del tipo *Dnode<T>* nos facilita grandemente las operaciones, en particular la de eliminación, la cual, en una lista simple, requeriría mantener un puntero al elemento predecesor.

Hay dos maneras de asociar otros datos a la clave de tipo *Key* definida en la cubeta:

1. Se crea un registro, por ejemplo, de tipo *Record*, que contenga todos los datos de interés, incluida la clave de tipo *Key*. Después se declara una cubeta (*LhashBucket<Record>* o *LhashBucketVtl<Record>*) y se define el operador *Record::operator==(const Record&)* para que sólo compare la clave de tipo *Key*.

La tabla hash puede definirse mediante cualquiera de los tres tipos *GenLHashTable*, *LHashTable* o *LHashTableVtl*. Los dos últimos tipos se corresponden con tablas que manejan cubetas sin o con destructores virtuales y se definen por especialización de *GenLHashTable*:

384c *<Cubetas de LHashTable<Key> 384b>+≡*

(384a) ▷384b

```
template <typename Key, class Cmp = Aleph::equal_to<Key> >
class LHashTable : public GenLHashTable<Key, LhashBucket<Key>, Cmp>
```

2. Se deriva una clase Derivada de cualquiera de las clases cubetas LhashBucket<Key> o LhashBucketVtl<Key> en donde se incluyan los datos que se requieran asociar al tipo Key.

La tabla hash se instancia mediante LhashBucket<Key> o LhashBucketVtl<Key>, según se requiera o no destructores virtuales. La inserción y la eliminación aceptan cubetas de tipo Derivada, pues son cubetas por herencia. La búsqueda retorna una cubeta de la familia LhashBucket<Key>, la cual debe convertirse explícitamente a Derivada si se requiere conocer la cubeta Derivada.

#### Atributos de LhashTable<Key>

Obviamente, un atributo esencial, del cual hablaremos más adelante, es la función hash, que transforma una clave de tipo Key a un natural. Tal función debe tener el prototipo siguiente:

385a *(Prototipo de puntero a función hash 385a)≡* (384a)  
`typedef size_t (*Hash_Fct)(const Key &);`

El tipo LhashTable<Key> mantiene, pues, un puntero de este tipo:

385b *(Miembros dato de GenLhashTable 385b)≡* (384a) 385c▷  
`Hash_Fct hash_fct;`

La “misión” de este tipo es manejar de forma genérica la función hash. Todo constructor de tabla hash requiere esta función.

El segundo atributo es la tabla misma, es decir, el arreglo de punteros a cubetas:

385c *(Miembros dato de GenLhashTable 385b)+≡* (384a) ▷385b 385d▷  
`typedef Dnode<Key> BucketList; // Tipo lista cubetas  
 typedef typename Dnode<Key>::Iterator BucketItor; // Iterador cubetas  
 typedef Dnode<Key> Node; // Sinónimo nodo  
 BucketList * table;`

Uses Dnode 83a.

Por añadidura, hay una declaración de un iterador, la cual permite recorrer una lista de colisiones.

El resto de los atributos se definen como sigue:

385d *(Miembros dato de GenLhashTable 385b)+≡* (384a) ▷385c  
`size_t M; // Tamaño de la tabla  
 size_t N; // Número de elementos de la tabla  
 size_t busy_slots_counter; // Cantidad de entradas ocupadas  
 bool remove_all_buckets; // liberar cubetas en destructor`

El atributo busy\_slots\_counter cuenta la cantidad de entradas del arreglo table[] que están vacías. Es una información muy importante para conocer el nivel de dispersión de la función hash. Si  $N < M$ , entonces el radio  $\frac{\text{busy\_slots\_counter}}{N}$  nos da idea de la efectividad de la función hash en términos de dispersión. Un valor pequeño indica que hay muchas colisiones.

El usuario de LhashTable<Key> tiene la responsabilidad de apartar la memoria para las cubetas, así como, eventualmente, la de liberarlas. Discutimos la razón de ser de esto cuando hablamos del principio fin-a-fin (§ 1.4.2 (Pág. 22)) y lo hemos aplicado a lo largo de los distintos TAD que hemos diseñado. En ese sentido, es bastante útil tener una rutina equivalente a remove\_all\_and\_delete() de Dlink (§ 2.4.7 (Pág. 82)) o destroyRec() de los árboles binarios (§ 4.4.10 (Pág. 250)). Así pues, el atributo remove\_all\_buckets le indica al destructor de LhashTable<Key> que debe liberar toda la memoria.

Si se desea vaciar la tabla, entonces puede invocarse al método `empty()`:

386a *(Métodos públicos de GenLhashTable 386a)*≡ (384a) 386b▷

```
void empty()
{
 for (int i = 0; i < M; ++i)
 for (BucketItor itor(table[i]); itor.has_current(); /* nada */)
 delete (Bucket*) itor.del();
 busy_slots_counter = N = 0;
}
```

## Inserción

Un asunto de interés es el lugar de la lista en que debe insertarse una nueva colisión: al principio, al final u ordenar la lista. Cualquiera de estas variantes no causa gran repercusión en el desempeño, pero si es simplemente enlazada, entonces la preferencia es por el frente, de modo que la inserción sea lo más “constantemente” rápida. Si la lista es doble, entonces el hecho de que se inserte al principio o final es indiferente en tiempo.

Otro factor de interés es considerar o no repitencia de claves. A diferencia de los árboles binarios de búsqueda, en una tabla hash es preferible no verificar repitencia porque sino se requiere una búsqueda en la lista de colisiones cada vez que se efectúe una inserción.

Ahora podemos especificar la inserción en `LhashTable<Key>`, la cual se instrumenta, muy sencillamente, así:

386b *(Métodos públicos de GenLhashTable 386a)*+≡ (384a) ▷386a 386c▷

```
Bucket * insert(Bucket * bucket)
{
 const size_t i = (*hash_fct)(bucket->get_key()) % M;

 if (table[i].is_empty()) // ¿está vacía la lista table[i]?
 ++busy_slots_counter; // sí ==> actualizar contador ocupación

 table[i].append(bucket); // insertar cubeta al final
 ++N;

 return bucket;
}
```

Si la llamada a `(*hash_fct)(key)` es  $\mathcal{O}(1)$ , entonces `insert()` es, con certitud determinista,  $\mathcal{O}(1)$ .

## Búsqueda

La búsqueda consiste en calcular la entrada de la tabla mediante la función hash y luego en recorrer la  $i$ -ésima lista hasta que se encuentre  $k$  o hasta que se llegue al final, en cuyo caso se trata de una búsqueda fallida:

386c *(Métodos públicos de GenLhashTable 386a)*+≡ (384a) ▷386b 387▷

```
Bucket * search(const Key & key) const
{
 const size_t i = (*hash_fct)(key) % M;

 if (table[i].is_empty())
 return NULL;
```

```

for (BucketItor it(table[i]); it.has_current(); it.next())
{
 Bucket * bucket = static_cast<Bucket*>(it.get_current());
 if (Cmp() (bucket->get_key(), key))
 return bucket;
}
return NULL;
}

```

El desempeño de esta rutina depende de la longitud de la lista `table[i]`.

### Eliminación

La eliminación es directa y con duración exactamente constante una vez que se conoce la dirección de la cubeta:

387 *(Métodos públicos de GenHashTable 386a) +≡* (384a) ◁386c 410▷

```

Bucket * remove(Bucket * bucket)
{
 // guarda próxima colisión
 Bucket * next = static_cast<Bucket*>(bucket->get_next());
 bucket->del(); // eliminar de su lista de colisiones
 if (next->is_empty()) // ¿la lista devino vacía?
 -busy_slots_counter;// sí ==> actualizar contador listas ocupadas

 -N;

 return bucket;
}

```

#### 5.1.3.2 Análisis del encadenamiento separado

Por simplicidad crítica asumiremos una tabla consistente de un arreglo de nodos cabecera a listas de colisiones circulares y doblemente enlazadas del tipo `Dnode<T>` estudiado en § 2.4.8 (Pág. 83). Bajo el mismo espíritu, también asumiremos que la inserción siempre se hace al final de la lista.

El desempeño de la inserción es directamente  $\mathcal{O}(1)$  mediante la operación `append()` en la cabecera que arroje la función hash.

El desempeño de la eliminación depende del desempeño de la búsqueda. A su vez, el desempeño de la búsqueda, exitosa o fallida, depende de la longitud de la lista. Por tanto, el desempeño esperado estará dominado por la longitud esperada de la lista.

En lo que sigue de nuestro análisis,  $M$  denota la longitud del arreglo interno y  $N$  el número de elementos que contiene la tabla.  $\alpha = N/M$  indica el factor de carga de la tabla, es decir, qué tan llena está la tabla. Con esta terminología de base podemos introducir nuestro lema fundamental.

#### Lema 5.1 (Longitud esperada de una lista de colisiones)

Sea  $h(k) : \mathcal{K} \rightarrow [0, M-1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M-1]$ . Sea  $T$  una tabla hash de longitud  $M$  con resolución de colisiones por encadenamiento separado y que contiene  $N$  elementos. Sea  $|l_i|$  el número de nodos de la  $i$ -ésima lista en  $T[i]$ . Entonces, la longitud esperada de  $l_i$  es:

$$\bar{l}_i = \frac{N}{M} = \alpha \quad (5.7)$$

**Demostración** Puesto que  $h(k)$  se distribuye uniformemente,  $P(h(k) = i) = \frac{1}{M}$  para una clave cualquiera  $k \in \mathcal{K}$ .

Sea  $P(k \in l_i)$  la probabilidad de que la clave sea albergada en la  $i$ -ésima lista correspondiente a la entrada  $T[i]$ . Claramente, según la observación del párrafo anterior,  $P(k \in l_i) = P(h(k) = i) = \frac{1}{M}$ . Así pues, tenemos  $N$  claves uniformemente repartidas en  $M$  listas. Cada lista  $l_i$  se corresponde con una distribución binomial con parámetro  $p = \frac{1}{M}$  y  $N$  cantidad de ensayos. Por tanto, la media es  $N \frac{1}{M}$   $\square$

De este lema se derivan los dos siguientes que contabilizan la cantidad de cubetas que se visitan en una búsqueda fallida y exitosa respectivamente.

**Lema 5.2 (Cantidad de cubetas inspeccionadas en una búsqueda fallida sobre una tabla hash con resolución de colisiones por encadenamiento separado)** Sea  $h(k) : \mathcal{K} \rightarrow [0, M - 1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M - 1]$ . Sea  $T$  una tabla hash de longitud  $M$  con resolución de colisiones por encadenamiento separado y que contiene  $N$  elementos. Entonces, el número de cubetas que se inspeccionan en una búsqueda fallida es:

$$\bar{U}_N = \alpha = \frac{N}{M} \quad (5.8)$$

**Demostración** Sea  $k$  una clave que no está dentro de la tabla. El primer paso de la búsqueda es obtener  $l_i = T[h(k)]$ . De este modo, el número de cubetas a visitar se corresponde con el número de nodos de  $l_i$ , pues es necesario recorrer enteramente a  $l_i$  para estar seguros de que  $k$  no se encuentra en la tabla. Por el lema 5.1 sabemos que  $\bar{l}_i = \alpha$   $\square$

**Lema 5.3 (Cantidad de cubetas inspeccionadas en una búsqueda exitosa sobre una tabla hash con resolución de colisiones por encadenamiento separado)** Sea  $h(k) : \mathcal{K} \rightarrow [0, M - 1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M - 1]$ . Sea  $T$  una tabla de longitud  $M$  con resolución de colisiones por encadenamiento separado y que contiene  $N$  elementos. Sea  $k \in \mathcal{K}$  una clave cualquiera. Entonces, el número esperado de nodos que se visitan en una búsqueda exitosa es:

$$\bar{S}_N = 1 + \frac{\alpha}{2} - \frac{1}{2M} \quad (5.9)$$

**Demostración** Sea  $K = \{k_0, k_1, \dots, k_{N-1}\}$ , el conjunto de claves presentes en la tabla, donde  $i$  representa el orden de inserción, es decir  $k_0$  es la primera clave insertada, mientras que  $k_{N-1}$  es la última.

A efectos de la simplicidad, asumiremos que no se permiten claves duplicadas, lo cual requiere una búsqueda secuencial sobre una lista cuando se hace una inserción.

Sea  $p_i$  la probabilidad de que se busque la clave presente  $k_i$ .

Sea  $l_{k_i}$  el número de elementos que preceden a  $k_i$  en su lista de colisiones. Entonces

$$E(l_{k_i}) = \sum_{i=0}^{N-1} p_i l_{k_i}.$$

Asumiendo que cualquier clave tiene igual probabilidad de ser buscada, entonces  $p_i = 1/N$ . Después de insertar la clave  $k_i$ , hay  $i$  elementos en la tabla, lo que, según el lema 5.1, hace

que la longitud de la lista donde reside  $k_i$  sea  $l_{k_i} = i/M$ . O sea, que para encontrar  $k_i$  hay que recorrer  $\frac{i}{M}$  cubetas predecesoras, pues  $k_i$  se insertó al final de la lista y no se permiten duplicados. Esto contabiliza un total de  $\frac{i}{M} + 1$ , pues hay que acceder a la cubeta que contiene  $k_i$ .

De este modo, el número de esperado de cubetas a examinar será

$$\begin{aligned} E(l_{k_i}) &= \sum_{i=0}^{N-1} p_i \left( \frac{i}{M} + 1 \right) = \frac{1}{N} \sum_{i=0}^{N-1} \left( \frac{i}{M} + 1 \right) \\ &= \frac{1}{N} \left( \frac{1}{M} \sum_{i=0}^{N-1} i + \sum_{i=0}^{N-1} 1 \right) = \frac{N-1}{2M} + 1 = \frac{\alpha}{2} - \frac{1}{2M} + 1 \quad \square \end{aligned}$$

Con estos dos lemas, estamos listos para enunciar la proposición siguiente.

**Proposición 5.1 (Desempeño esperado de una tabla hash con resolución de colisiones por encadenamiento separado)** Sea  $h(k) : \mathcal{K} \rightarrow [0, M-1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M-1]$ . Sea  $T$  una tabla de longitud  $M$  con resolución de colisiones por encadenamiento separado y que contiene un máximo acotado de  $N$  elementos. Entonces, el desempeño esperado de las operaciones de inserción, búsqueda y eliminación es  $\mathcal{O}(\alpha) = \mathcal{O}(1)$ .

**Demostración** El punto crucial de la demostración es aprehender que tanto  $M$  como  $N$  son valores constantemente acotados. Por acotado entendemos que si bien  $N$  puede ser mayor que  $M$ , éstos valores están acotados a máximos conocidos, razón por la cual  $\alpha$  también está acotado a un valor máximo constante. Aclarado esto, podemos separar cada una de las operaciones:

**Inserción:** en este caso se efectúa una búsqueda fallida, la cual, según el lema 5.2, espera inspeccionar  $\alpha = \mathcal{O}(1)$  cubetas.

**Búsqueda:**

- Si la búsqueda es fallida, entonces el lema 5.2 aplica y la esperanza es:

$$\alpha = \mathcal{O}(1) \tag{5.10}$$

- Si la búsqueda es exitosa, entonces el lema 5.3 aplica y la esperanza es:

$$1 + \frac{\alpha}{2} - \frac{1}{2M} = \mathcal{O}(1) \tag{5.11}$$

**Eliminación:** en este caso se hace una búsqueda exitosa, la cual, en el ítem anterior se demostró que es  $\mathcal{O}(1)$  ■

Respecto al TAD LhashTable<Key>, la demostración sólo ataña a la búsqueda, pues, por su funcionalidad, en este TAD, tanto la inserción como la eliminación son  $\mathcal{O}(1)$ .

Una “ventaja” del encadenamiento separado es que el número de elementos  $N$  puede ser mayor que el tamaño de la tabla  $M$ . La inserción no tiene entonces por qué preocuparse por el desborde, lo cual nos proporciona flexibilidad de circunstancias. De los enfoques tradicionales para manejar colisiones, este es el único que permite  $N > M$ .

El desempeño del encadenamiento separado es una predicción, ¿qué tan probable es?. Un indicio de respuesta nos las proporciona la varianza, la cual es deducible de la prueba del lema 5.1 en la que determinamos que la longitud de una lista de colisiones se distribuye binomialmente con parámetro  $p = \frac{1}{M}$  y  $N$  ensayos. Por tanto, la varianza es:

$$\text{var} = \sigma^2 = \frac{N}{M} \left(1 - \frac{1}{M}\right) = \alpha - \frac{\alpha}{M} \quad (5.12)$$

Cuanto mayor sea  $M$ , más aproximada es la varianza al valor  $\alpha = \bar{l}_i$ . Cuanto mayor sea la cantidad de elementos, mayor tiende a ser la varianza. Sin embargo, esta observación desdeña el radio  $\alpha$  y, justamente, el propósito de la última expresión es reflejar el hecho de que, a mayor radio, o sea, a mayor plenitud, mayor es la varianza.

### 5.1.3.3 Encadenamiento cerrado

Al igual que en el encadenamiento separado, en esta estrategia, las colisiones se encadenan en listas enlazadas, pero la diferencia estriba en que las cubetas residen en la propia tabla. Cuando ocurre una colisión se “sondea”<sup>4</sup> linealmente la tabla en búsqueda de una cubeta vacía. La tabla ejemplo de la subsección anterior, con las mismas listas de colisiones, se vislumbra como en la figura 5.2. Notemos la colisión  $\{k_6, k_{11}\}$ , la cual requirió continuar el sondeo por el principio de la tabla hasta alcanzar la cuarta entrada de la tabla, es decir, el sondeo lineal debe ser circular.

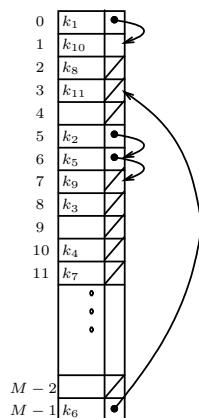


Figura 5.2: Un ejemplo de tabla hash con resolución de colisiones por encadenamiento cerrado

El sondeo requiere una forma de distinguir si una cubeta está o no vacía. La manera genérica tradicional es usar un bit con valor BUSY para indicar ocupación, o EMPTY para indicar disponibilidad.

### Búsqueda

La búsqueda es idéntica a la del encadenamiento separado.

<sup>4</sup>En inglés, a la búsqueda de una cubeta con status EMPTY se le llama “probing”. “Sondear” es el término que usaremos en castellano para referirnos a la palabra anterior.

### Eliminación

La eliminación requiere la búsqueda y se remite a marcar la entrada que contiene el elemento a eliminar como `Empty`. Si hay colisiones, entonces se debe “cortocircuitar” el enlace del elemento predecesor para que apunte al elemento sucesor.

Un caso especialmente particular ocurre cuando se elimina el primer elemento dentro de una lista de colisiones. En esta situación, el “cortocircuito” no se puede ejecutar directamente, pues no se tiene una colisión que preceda al elemento eliminado. Para resolver esto, el segundo elemento en la lista se mueve hacia la posición de eliminación y, a partir de esta posición, se efectúa el “cortocircuito” hacia el tercer elemento colindante. Para un hipotético conjunto de colisiones  $\{k_i, k_j, k_k, k_l\}$ , la situación se puede interpretar pictóricamente como en la figura 5.3. Puesto que con esta técnica los elementos pueden moverse de su posición original de inserción, este esquema de resolución de colisiones no permite mantener punteros a elementos del conjunto almacenado en la tabla, situación indeseable en algunos casos.

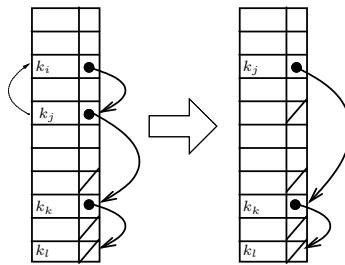


Figura 5.3: Un ejemplo general de eliminación en una tabla hash con resolución de colisiones por encadenamiento cerrado

### Inserción

La inserción es la operación más compleja y se efectúa como sigue:

1.  $i = h(k) \bmod M$
2. Si  $A[i].busy == EMPTY \Rightarrow$ 
  - (a)  $A[i].key = k$
  - (b)  $A[i].busy = BUSY$
  - (c) Terminar.
3. De lo contrario  $\Rightarrow$ 
  - (a) Recorrer la lista enlazada hasta el último elemento; sea  $k$  su posición en la tabla.
  - (b) A partir de  $k$ , buscar linealmente en la tabla la primera entrada con valor `EMPTY`; sea  $i$  el índice de esta entrada.
  - (c)  $A[i].key = k$
  - (d)  $A[i].busy = BUSY$
  - (e)  $A[j].next = A[i]$

#### 5.1.3.4 Análisis informal del encadenamiento cerrado

Debe sernos fácil asumir que si  $M \approx \infty$ , entonces el análisis es aproximado al encadenamiento separado. Esta suposición algo irrealista nos permite aprehender que para  $M \gg N$ , según los lemas (5.2) y (5.3), el análisis puede aproximarse a:

$$\overline{S_N} \approx \frac{\alpha}{2} - \frac{1}{2M} + 1 \quad (5.13)$$

$$\overline{U_N} \approx \alpha \quad (5.14)$$

$$\text{var} \approx \alpha - \frac{\alpha}{M} \quad (5.15)$$

En la subsección § 5.1.4.3 (Pág. 399) presentaremos un análisis más formal aplicable a esta estrategia.

Cuanto menor sea la carga  $\alpha$ , más precisa y realista es la aproximación. ¿Hasta qué valor de  $\alpha$  se cumple la aproximación? El análisis de otra técnica, explicado en § 5.1.4.5 (Pág. 403), nos permite vislumbrar que la aproximación es buena hasta un factor de carga  $\alpha = 0,7$ , es decir, un porcentaje de ocupación del 70%.

Lo anterior nos indica que si deseamos instrumentar el problema fundamental con una tabla hash y desempeño esperado  $\mathcal{O}(1)$ , entonces debemos estimar la cantidad esperada de elementos que vamos a guardar y, en función de la estimación, escoger  $M$  de modo que nos satisfaga el valor de  $\alpha$  para el cual los resultados son buenos, más un margen o factor de error. Si, por ejemplo, esperamos 1000 elementos, entonces escogemos  $M = 1465$ , lo cual cubre el 70% mencionado más un 5% como factor de error ingenierí<sup>5</sup>.

Una mejora para este enfoque consiste en emplear un área contigua de la tabla exclusivamente. En este caso, la tabla tiene un tamaño de  $M + C$  entradas. Cuando ocurre una colisión, entonces el sondeo lineal se hace entre el rango  $[M..C]$ . Resultados empíricos reportados por [108] indican que un valor  $C \approx 1.14M$  es suficiente para esperar cadenas de colisiones de dos elementos. Esto tiene sentido a la luz de la paradoja del cumpleaños, la cual patenta lo bastante probable que es una colisión de dos elementos, pero un análisis para colisiones de tres o más elementos nos indica mucha menos probabilidad.

A este estadio de tratamiento de las tablas hash, probablemente algún lector haya observado, como desventaja de este enfoque, la limitación que se impone sobre el máximo número de elementos que podemos guardar en una tabla hash con resolución de colisiones por encadenamiento cerrado. Bajo la misma línea de pensamiento, también se podría observar, como ventaja del encadenamiento separado, que éste permite guardar más elementos que el valor de  $M$ . ¿cuál es entonces el sentido de esta técnica?

El asombro y la sorpresa han embargado a muchos estudiantes, inclusive a algunos estudiosos, cuando se descubre que, considerando correctamente los factores involucrados, en particular el valor esperado de  $N$ , los enfoques que resuelven colisiones dentro de la misma tabla tienden a ser de mejor desempeño que el encadenamiento separado. La aprensión de esta realidad se facilita mediante las siguientes observaciones:

1. En el encadenamiento separado, las cubetas se apartan y liberan a través del manejador de memoria. El manejo de memoria dinámica, tanto para reservarla como para liberarla, no sólo toma un tiempo de más respecto a un enfoque de resolución de colisiones que las ubique dentro de la misma tabla, sino que su desempeño es variable según la carga

---

<sup>5</sup>En ingeniería, suele usarse el término "factor de seguridad" para expresar un sobrecálculo en consideración a la incertidumbre del modelo y errores de cálculo en sus diversas fases y facetas.

del manejador, carga que a su vez depende de condiciones tales como frecuencia de reservación y liberado y la cantidad actual de bloques reservados.

Para el encadenamiento separado, el apartado afecta a la operación de inserción, mientras que la liberación lo hace sobre la eliminación. Por el contrario, con el encadenamiento cerrado y demás estrategias que ubican las colisiones dentro de la misma tabla, no se usa el manejador de memoria.

Alguien podría sugerir tener preapartado un conjunto de cubetas, pero, ¿no es esto parecido a un enfoque cerrado?

2. En el encadenamiento cerrado y demás estrategias que ubiquen colisiones dentro de la misma tabla, el sondeo lineal se hace sobre cubetas que están contiguas en memoria, lo cual aprovecha el cache del hardware. Por el contrario, con el encadenamiento separado, las cubetas están en direcciones de memoria discontinuas. Por consiguiente, para el mismo conjunto de colisiones, su inspección dentro de la propia tabla tiende a ser considerablemente más rápida que su inspección a lo largo de una cadena de cubetas discontinuas en la memoria, pues el primero aprovecha el cache y el otro no.

#### 5.1.4 Direccionamiento abierto

Como ya indicamos, en el direccionamiento abierto, toda colisión se resuelve dentro de la tabla; sin apuntador hacia la colisión.

Fundamentalmente, las celdas contienen registros con el valor de clave y una marca, denominada “status”, con tres posibles valores:

**EMPTY** : la celda está disponible para almacenar una clave.

**BUSY** : la celda está ocupada por una clave.

**DELETED** : la celda contiene una entrada que en algún momento contuvo una clave y que luego fue borrada. Más adelante veremos por qué es necesaria esta marca.

Una cubeta  $\text{tabla}[i]$  puede estar ocupada como resultado directo de la función hash o porque es una colisión y fue escogida por alguna técnica de sondeo. Cualquiera de los dos casos se distingue mediante comprobación del predicado  $h(\text{tabla}[i]) = i$ . Si el predicado es cierto, entonces la clave no fue una colisión.

##### 5.1.4.1 Sondeo ideal (sondeo uniforme)

Puesto que las colisiones deben estar dentro de la propia tabla, la estrategia se remite encontrar una cubeta cuyo valor sea distinto a **EMPTY**. Puede decirse que la estrategia de dispersión cerrada (direccionamiento abierto) es una estrategia de sondeo.

¿Cuál es entonces la estrategia de sondeo ideal? Se “presume” que es aquella que inspecciona aleatoriamente las cubetas, es decir, cuando ocurre una colisión, la próxima cubeta a sondear en donde ubicar la colisión debe seleccionarse aleatoriamente en el intervalo  $[0, M - 1]$ . Típicamente, esta suposición es conocida como “hashing o dispersión universal”, y se le dice ideal porque idealiza lo que sería el mejor desempeño de cualquier estrategia cerrada de manejo de colisiones.

**Lema 5.4 (Cantidad de cubetas inspeccionadas en una búsqueda fallida sobre una tabla hash con resolución de colisiones por direccionamiento abierto y sondeo ideal - Peterson 1957 [143])** Sea  $h(k) : \mathcal{K} \rightarrow [0, M-1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M-1]$ . Sea  $T$  una tabla hash de longitud  $M$  con resolución de colisiones por direccionamiento abierto y que contiene  $N$  elementos. Sea  $k \in \mathcal{K}$  una clave cualquiera. Entonces, el promedio de cubetas que se visitan en una búsqueda fallida es:

$$\overline{U_N} = \frac{M+1}{M-N+1} \quad (5.16)$$

**Demostración** Enunciemos  $\overline{U_N}$  de la siguiente manera:

$$\overline{U_N} = \frac{\text{Suma de la cantidad de sondeos de } i \text{ cubetas en búsqueda fallida}}{\text{Total de configuraciones posibles de la tabla hash}} = \frac{\pi_i}{\pi_N} \quad (5.17)$$

Si tenemos  $M$  cubetas numeradas entre 1 y  $M$  las repartimos a las  $N$  claves insertadas, entonces el total de maneras posibles de hacerlo es  $\binom{M}{N}$ , o sea, el total de configuraciones posibles de la tabla hash; esto es:  $\pi_N$ .

Sea  $i$  la cantidad de cubetas que se inspeccionan en una búsqueda fallida con sondeo ideal.

Para que se consuman  $i$  sondeos, las primeras  $i-1$  cubetas deben tener status BUSY, mientras que el status de la última debe ser EMPTY. En  $i$  sondeos se revisan  $i-1$  cubetas que contienen claves mientras que la  $i$ -ésima cubeta sondeada está vacía. La cantidad de disposiciones en que se sondean  $i-1$  claves es equivalente a la cantidad de maneras de distribuir las  $N-i+1$  claves restantes en las  $M-k$  cubetas restantes<sup>6</sup>, es decir  $\binom{M-i}{N-i+1}$  para una disposición de claves particular. El total posible es la suma para todos los valores posibles de  $i$ :

$$\begin{aligned} \pi_i &= \sum_{i=1}^M i \binom{M-i}{N-i+1} = \sum_{i=1}^M i \binom{M-i}{M-N-1} \quad (\text{Por simetría } \binom{n}{k} = \binom{n}{n-k}) \\ &= \binom{M-1}{M-N-1} + 2 \binom{M-2}{M-N-1} + \dots + \\ &\quad (M-1) \binom{1}{M-N-1} + M \binom{0}{M-N-1} \\ &= \sum_{i=0}^{M-1} (M-i) \binom{i}{M-N-1} \\ &= M \sum_{i=0}^{M-1} \binom{i}{M-N-1} - \sum_{i=0}^{M-1} i \binom{i}{M-N-1} \end{aligned}$$

Ahora, a efectos de aplicar la propiedad de la sumatoria del índice superior de los coefi-

---

<sup>6</sup>Recuérdese que  $\binom{n}{k} = \binom{n}{n-k}$  pues de los  $k$  elementos combinados siempre sobran  $n-k$ .

cientes binomiales<sup>7</sup>, vamos a expresar el factor  $i$  de la segunda suma como  $i + 1 - 1$ :

$$\begin{aligned}\pi_i &= M \sum_{i=0}^{M-1} \binom{i}{M-N-1} - \sum_{i=0}^{M-1} (i+1-1) \binom{i}{M-N-1} \\ &= \underbrace{(M+1) \sum_{i=0}^{M-1} \binom{i}{M-N-1}}_{(A)} - \underbrace{\sum_{i=0}^{M-1} (i+1) \binom{i}{M-N-1}}_{(B)}\end{aligned}\quad (5.18)$$

(A) es calculable directamente por la propiedad de la sumatoria del índice superior:

$$(A) = (M+1) \binom{M}{M-N} = (M+1) \binom{M}{N} \quad (\text{por simetría}) \quad (5.19)$$

El cuanto a (B), podemos absorber<sup>8</sup> el factor  $(i+1)$  si multiplicamos y dividimos por  $(M-N)$  y luego aplicamos la propiedad de la sumatoria del índice superior:

$$\begin{aligned}(B) &= (M-N) \sum_{i=0}^{M-1} \frac{i+1}{M-N} \binom{i}{M-N-1} \\ &= (M-N) \sum_{i=0}^{M-1} \binom{i+1}{M-N} = (M-N) \binom{M+1}{M-N+1} \\ &= (M-N) \binom{M+1}{N} \quad (\text{por simetría})\end{aligned}\quad (5.20)$$

Sustituyendo (A) y (B) en (5.18):

$$\begin{aligned}\pi_i &= (M+1) \binom{M}{N} - (M-N) \binom{M+1}{N} \\ &= (M+1) \binom{M}{N} - (M-N) \frac{(M+1)!}{(M-N+1)!N!} \\ &= (M+1) \binom{M}{N} - (M-N) \frac{(M+1)M!}{(M-N+1)(M-N)!N!} \\ &= (M+1) \binom{M}{N} \left(1 - \frac{M-N}{M-N+1}\right) = \frac{M+1}{M-N+1} \binom{M}{N}\end{aligned}\quad (5.21)$$

Al sustituir (5.21) y el valor  $\pi_N = \binom{M}{N}$  en (5.17) obtenemos (5.16)  $\square$

Conforme  $M, N \rightarrow \infty$ , la inversa de la expresión (5.16) tiende a:

$$\overline{U_N}^{-1} = \lim_{M,N \rightarrow \infty} \frac{M-N+1}{M+1} = (1-\alpha)^{-1} \quad (5.22)$$

El resultado aproxima  $\frac{N}{M+1} \approx \alpha$ , lo que en la realidad puede ser preciso, pues suele usarse una cubeta más que el valor previsto de  $M$  que funja de centinela para detener el sondeo en todos los casos.

<sup>7</sup>La propiedad en cuestión se define como

$$\sum_{k=0}^n \binom{k}{m} = \binom{n+1}{m+1}.$$

<sup>8</sup>

$$\binom{n}{m} = \frac{n}{m} \binom{n-1}{m-1}.$$

Intuitivamente, la probabilidad de que la inserción ocurra en el primer sondeo (el directo resultante de la función hash) es  $\alpha$ . Del mismo modo, la probabilidad de requerirse dos sondeos, es  $\alpha^2$ . En fin, generalmente, la probabilidad de requerirse  $k$  sondeos ideales es  $\alpha^k$ .

La cantidad de sondeos que se hacen en una búsqueda exitosa está expresada por el lema siguiente.

**Lema 5.5 (Cantidad de cubetas inspeccionadas en una búsqueda exitosa sobre una tabla hash con resolución de colisiones por direccionamiento abierto y sondeo ideal - Peterson 1957 [143])** Sea  $h(k) : \mathcal{K} \rightarrow [0, M-1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M-1]$ . Sea  $T$  una tabla hash de longitud  $M$  con resolución de colisiones por direccionamiento abierto y que contiene  $N$  elementos. Sea  $k \in \mathcal{K}$  una clave cualquiera. Entonces, el promedio de cubetas que se visitan en una búsqueda exitosa es:

$$\overline{S_N} = \frac{M+1}{N} (H_{M+1} - H_{M-N+1}) \quad (5.23)$$

**Demostración** Sea  $\overline{S_i}$  el promedio de cubetas que se sondean en una búsqueda exitosa sobre una tabla con  $i$  elementos.

El fundamento de la demostración es expresar una búsqueda sobre una tabla que contiene  $i$  elementos en función de la búsqueda fallida que hubo de hacerse para insertar el  $i$ -ésimo elemento, o sea  $\overline{S_i} = \overline{U_{i-1}}$ .

El promedio  $\overline{S_N}$  puede expresarse, en términos de (5.16), de la siguiente manera:

$$\begin{aligned} \overline{S_i} &= \frac{1}{N} \sum_{i=0}^{N-1} \overline{U_i} \\ &= \frac{1}{N} \sum_{i=0}^{N-1} \frac{M+1}{M-i+1} = \frac{M+1}{N} \sum_{i=0}^{N-1} \frac{1}{M-i+1} \\ &= \frac{M+1}{N} (H_{M+1} - H_{M-N+1}) \quad \square \end{aligned}$$

Bajo la misma aproximación  $M, N \rightarrow \infty$  y la aproximación de los números armónicos<sup>9</sup>:

$$\begin{aligned} \lim_{M,N \rightarrow \infty} \overline{S_i} &= \lim_{M,N \rightarrow \infty} \frac{M+1}{N} (\ln(M+1) - \ln(M-N+1)) \\ &= \lim_{M,N \rightarrow \infty} \frac{1}{\alpha} \ln \frac{M+1}{M-N+1} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \quad (5.24) \end{aligned}$$

Como ya hemos indicado, el sondeo ideal es lo que se presume que arrojaría el mejor desempeño. En la práctica es irreal porque, aparte de que es demasiado difícil generar sondeos aleatorios e independientes para cualquier conjunto de claves, es posible, aunque muy improbable en la medida en que  $M$  sea más grande, que el sondeo ideal no encuentre una cubeta disponible. El sondeo ideal y las cotas expresadas por (5.22) y (5.24) nos proporcionan una referencia de desempeño para otros enfoques de sondeo que estudiaremos inmediatamente.

---

<sup>9</sup>  $H_n \approx \ln n + \mathcal{O}(1)$ .

### 5.1.4.2 Sondeo lineal

El sondeo lineal es muy similar al del encadenamiento cerrado: si ocurre una colisión, entonces la próxima celda a sondar será la siguiente disponible, cuya marca sea distinta de EMPTY, resultante de la inspección secuencial a partir del índice dado por la función hash. La tabla ejemplo que hemos mostrado para las dos estrategias anteriores se pictoriza como en la figura 5.4. Notemos que la distribución de las claves es idéntica a la del encadenamiento separado, pues son resultantes del mismo tipo de sondeo.

|         |          |   |
|---------|----------|---|
| 0       | $k_1$    | E |
| 1       | $k_{10}$ | B |
| 2       | $k_8$    | E |
| 3       | $k_{11}$ | B |
| 4       |          | E |
| 5       | $k_2$    | B |
| 6       | $k_5$    | B |
| 7       | $k_9$    | B |
| 8       | $k_3$    | B |
| 9       |          | E |
| 10      | $k_4$    | B |
| 11      | $k_7$    | B |
|         | •        | E |
|         | •        | E |
|         | •        | E |
|         |          | E |
| $M - 2$ |          | E |
| $M - 1$ | $k_6$    | B |

Figura 5.4: Un ejemplo de tabla con resolución de colisiones por direccionamiento abierto

*ALÉPH* tiene un TAD, denominado `OLHashTable`, que implanta una tabla hash con resolución de colisiones por direccionamiento abierto y sondeo lineal. El TAD en cuestión se exporta en el archivo `(tpl_olhash.H 397a)`:

397a

```
(tpl_olhash.H 397a)≡
template <typename Key, typename Record, class Cmp = Aleph::equal_to<Key> >
class OLHashTable
{
 (Miembros públicos de OLHashTable 397b)
};
```

`OLHashTable` representa una tabla hash, con resolución de colisiones abierta y sondeo lineal de cubetas, indizada por claves de tipo `Key` y que guarda registros de tipo `Record`.

### Búsqueda

La búsqueda de una clave se remite a:

397b

```
(Miembros públicos de OLHashTable 397b)≡ (397a) 398a▷
Record * search(const Key & key)
{
 // Comenzar desde índice hash y sondar hasta encontrar cubeta EMPTY
 for (int i = (*hash_fct)(key) % M, c = 0;
 c < M and table[i].status != EMPTY; ++c, ++i)
 if (table[i].status == BUSY) // ¿Hay una clave en la cubeta?
 if (Cmp() (table[i].key, key)) // Comparar la clave
 return &table[i].record;

 return NULL; // No se encuentra la clave
}
```

El lazo se detiene cuando se sondea una celda cuyo status sea `EMPTY`. Una celda con status igual a `BUSY` se revisa para verificar si contiene la clave de búsqueda. Una celda con

status DELETED no contiene clave, pero como está en el camino de una colisión, hay que continuar el sondeo sobre la siguiente cubeta.

### Inserción

La inserción se remite a encontrar la primera cubeta, sea resultante directa de la función hash o del sondeo lineal, con status distinto a BUSY. Esto se plantea mediante el siguiente método:

398a

```
(Miembros públicos de OLHashTable 397b) +≡ (397a) <397b 398b>
Record * insert(const Key & key, const Record & record)
{
 if (N >= M)
 throw std::overflow_error("Hash table is full");

 int i = (*hash_fct)(key) % M;

 while (table[i].status != BUSY) // sondeo lineal de cubetas disp.
 i = (i + 1) % M;

 // i contiene celda con DELETED o EMPTY ==> ocuparla
 table[i].key = key;
 table[i].record = record;
 table[i].status = BUSY;
 N++;

 return &table[i].record;
}
```

### Eliminación

La eliminación es la operación polémica de esta estrategia, pero es fundamentalmente simple: (1) encontrar la cubeta que contiene la clave a eliminar y (2) marcarla con DELETED.

Las cubetas marcadas como DELETED representan un coste adicional a la búsqueda, pues éstas deben inspeccionarse aunque no contengan claves. Esto plantea un problema de entropía a medida que aumenta la cantidad de eliminaciones, pudiendo ocurrir la desafortunada situación de tener que revisar todas las cubetas de la tabla en una búsqueda fallida aunque la tabla contenga muy pocos elementos. Consecuentemente, con direccionamiento abierto, las eliminaciones deben ser excepcionales.

### Mejoras a la eliminación

La entropía causada por la acumulación de cubetas con status DELETED puede contrarrestarse mediante una técnica que cierre la brecha dejada por la cubeta eliminada dentro de la cadena de colisiones. De este modo, la última cubeta de la cadena puede marcarse con el valor EMPTY. Puesto que la eliminación requiere una búsqueda, la interfaz para eliminar se plantea en función de un puntero al registro que se desea eliminar. De este modo, el usuario puede, eventualmente, ahorrar la repetición del trabajo de búsqueda. Dicho esto, no debe resultar difícil comprender la rutina de eliminación:

398b

```
(Miembros públicos de OLHashTable 397b) +≡ (397a) <398a
void remove(Record * record)
{
 Bucket * bucket = record_to_bucket(record);
```

```

const int i = bucket - &table[0]; // índice de brecha

for (int j = (i + 1) % M; true; ++j) // cerrar la brecha i
 switch (table[j].status)
 {
 case BUSY:
 if (Cmp () ((*hash_fct)(table[j].key), bucket->key))
 { // hay colisión ==> mover su contenido hacia table[i]
 table[i].key = table[j].key;
 table[i].record = table[i].record;
 table[i].status = BUSY;
 table[j].status = DELETED; // deviene DELETED, quizá
 // candidata a EMPTY
 i = j; // table[j] deviene cubeta brecha
 }
 break;
 case DELETED: // en este caso cubeta i no puede marcarse con
 // EMPTY porque si no rompe cadena de sondeo
 i = j; // table[j] deviene cubeta brecha
 break;
 case EMPTY: // en este caso j es cubeta que detiene búsqueda,
 // pero como que i está más atrás y no interrumpe
 // búsqueda se marcam EMPTY y terminamos
 table[i].status = EMPTY;
 N--;
 return; // terminar
 }
}

```

Esta rutina cierra la brecha con status DELETED y marca la última cubeta de la cadena de colisiones como EMPTY.

Aunque esta “mejora” perfecciona el proceso de búsqueda, ralentiza sin embargo el de eliminación. Si las eliminaciones son excepcionales, entonces es preferible utilizar el muy simple algoritmo de marcar la cubeta con DELETED y decrementar N.

La mejora plantea el mismo problema de movimiento de cubetas que tiene el encadenamiento cerrado. Consecuentemente, tampoco se pueden mantener punteros a elementos de la tabla.

#### 5.1.4.3 Análisis informal del sondeo lineal

El análisis que presentaremos en esta subsección está basado en los resultados presentados por Sedgewick y Flajolet [157] y se fundamentan en la siguiente proposición, cual fue el primer algoritmo que Knuth analizó exitosamente.

**Proposición 5.2 (Cantidad de cubetas accedidas en direccionamiento abierto y sondeo lineal - Knuth 1962 [99])** Sea  $h(k) : \mathcal{K} \rightarrow [0, M-1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M-1]$ . Sea T una tabla hash de longitud M con resolución de colisiones por direccionamiento abierto y sondeo lineal. Sea  $k \in \mathcal{K}$  una clave cualquiera. Entonces, el promedio de cubetas que se visitan en una

búsqueda fallida es:

$$\overline{U_N} = \frac{1}{2} + \frac{1}{2} \sum_{i=0}^{N-1} \frac{(N-1)!}{M^i(N-i-1)!} = \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right) + \mathcal{O}\left(\frac{1}{N}\right); \quad (5.25)$$

y el promedio para una búsqueda exitosa es:

$$\overline{S_N} = \frac{1}{2} + \frac{1}{2} \sum_{i=0}^{N-1} i \frac{N!}{M^i(N-i)!} = \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right) + \mathcal{O}\left(\frac{1}{N}\right). \quad (5.26)$$

**Demostración** Por su complejidad, la demostración es dejada en ejercicio. Consultese Sedgewick y Flajolet [157] y Knuth [99, 100] para los detalles ■

**Corolario 5.1 (Desempeño esperado de una tabla hash con resolución de colisiones por direccionamiento abierto y sondeo lineal)** Sea  $h(k) : \mathcal{K} \rightarrow [0, M-1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M-1]$ . Sea  $T$  una tabla de longitud  $M$  con resolución de colisiones por direccionamiento abierto y sondeo lineal. Entonces, el desempeño esperado de las operaciones de inserción, búsqueda y eliminación es  $\mathcal{O}(\alpha) = \mathcal{O}(1)$ .

**Demostración**  $M$  está acotado a un valor constante. Puesto que la tabla es cerrada,  $N \leq M$ , lo cual implica que  $\alpha$  es constante. En virtud de este hecho, las expresiones (5.25) y (5.26) son constantes.

La inserción está determinada por la búsqueda fallida y, puesto que (5.25) es constante, el desempeño esperado también es constante. Ergo, la inserción es  $\mathcal{O}(1)$ .

La búsqueda fallida encaja en la misma situación que la inserción. La búsqueda exitosa depende de (5.26), la cual también es constante.

La eliminación con el TAD `OLHashTable` es determinísticamente constante. Cualquier otro TAD basado en el sondeo lineal que requiera una búsqueda encaja en los casos anteriores ■

Ahora que hemos mostrado la proposición que nos caracteriza el desempeño del sondeo lineal, meditemos acerca de lo que ella nos revela. La figuras 5.5 y 5.6 ilustran los desempeños para del encadenamiento separado, sondeo lineal y sondeo ideal para la búsqueda fallida y exitosa respectivamente.

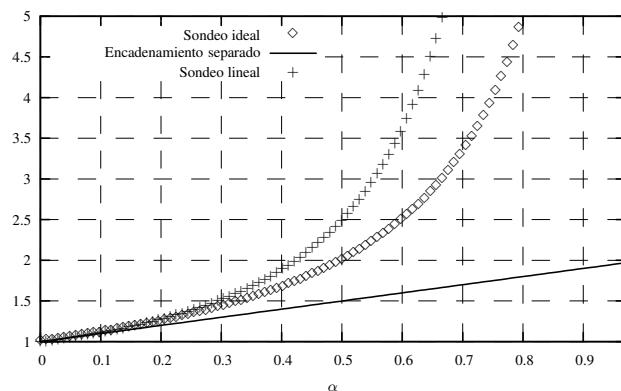


Figura 5.5: Cantidad de cubetas inspeccionadas en una búsqueda fallida

El sondeo lineal se degrada para la búsqueda exitosa cuando la tabla se encuentra en un 80% de plenitud; a partir de allí, la pendiente deviene importante y la cantidad de cubetas que se inspeccionan aumenta notablemente. Antes del 80% la cantidad de inspecciones es en promedio menor que tres, lo cual es aceptabilísimo habida cuenta del cache.

En cuanto a la búsqueda fallida, el asunto empeora antes para el sondeo lineal. A partir del 60% se inspeccionan en promedio cuatro cubetas en una búsqueda fallida, lo cual es equivalente a decir que se sondean cuatro cubetas en una inserción. Tenemos pues un margen entre el 60% y el 80% en el cual el sondeo lineal es en promedio aceptable. ¿Cuál es el porcentaje de plenitud en que podemos trabajar con el sondeo lineal de manera segura?

La pregunta anterior requiere cuestionar ¿cuál, entre las dos curvas, es la más representativa?, o quizás, mejor dicho, ¿cuál entre las dos búsquedas es la más importante? Por lo general la búsqueda o consulta de lo que ya está en un conjunto es más frecuente que la búsqueda fallida. En este sentido, la gráfica 5.6 es la más importante. Bajo este criterio y el conocimiento empírico podemos decir que el sondeo lineal es abordable hasta un máximo entre el 70 y 80% de plenitud. ¿Qué tanto cuesta el 30-20% de desocupación? Una perspectiva interesante para abordar esta pregunta nos la proporciona comparar el espacio ocupado por el encadenamiento separado.

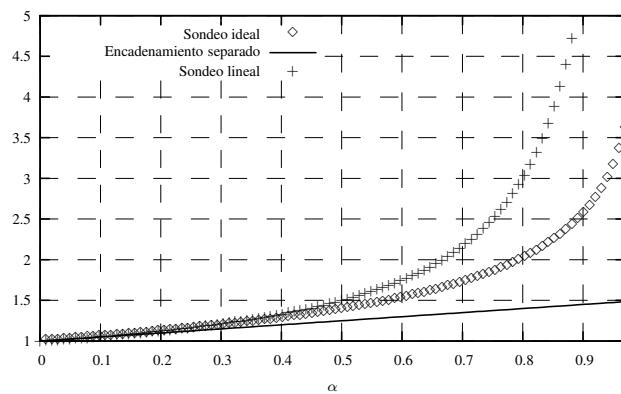


Figura 5.6: Cantidad de cubetas inspeccionadas en una búsqueda exitosa

¿Qué es lo que degrada el sondeo lineal, tan notable en las gráficas, respecto al sondeo ideal? La respuesta es conocida bajo el rótulo de “agrupamiento primario”. Es evidente que cualquier clave colisión  $h(k_1) = i$  potencia una colisión  $h(k_2) = i + 1$ . Según aumenta la cantidad de elementos se forman cadenas lineales de colisiones en torno a la posición  $i$ . Sin embargo, en la práctica, estas cadenas son tolerables hasta el porcentaje de plenitud expresado (80%), pues su contigüidad en el arreglo aprovecha el cache del computador<sup>10</sup>.

Con nuestra implantación del encadenamiento separado tenemos una ocupación en memoria de:

$$S_L = 2MS_pS_T + 2NS_pS_T . \quad (5.27)$$

Donde  $S_p$  es el espacio ocupado por un puntero y  $S_T$  es el espacio ocupado por la clave.

En el mismo sentido, con el direccionamiento abierto tenemos una ocupación de memoria de:

$$S_O = M(S_T + 1) \quad (5.28)$$

<sup>10</sup>Estos porcentajes pueden corroborarse (o refutarse) mediante cachegrind [129].

Claramente,  $S_O < S_L$ , lo que no significa que no pueda reducirse el consumo de memoria del encadenamiento separado. Por ejemplo, podemos usar listas simples y una tabla de puros punteros.

Las gráficas 5.5 y 5.6 muestran contundentemente la superioridad teórica del encadenamiento separado en cuanto a la cantidad de sondeos que se efectúan para cualquier búsqueda. Pero, como ya indicamos en § 5.1.3.4 (Pág. 392), el sondeo lineal aprovecha el cache del computador, un plano de ejecución que es varios órdenes de magnitud más veloz que el acceso fragmentado a la memoria, sin aprovechar el cache, que requeriría el encadenamiento separado. Por añadidura, la inserción e eliminación requieren apelar al manejador de memoria dinámica, mientras que el direccionamiento abierto no lo necesita.

Cuando se atina la cantidad esperada de claves a manejar, el direccionamiento abierto con sondeo lineal es, probable pero sorprendentemente, aun con los argumentos en mano, el mejor enfoque para manejar una tabla hash. Las razones se pueden resumir en:

- Es muy simple de implantar, lo cual implica un bajo coste constante.
- No utiliza el manejador de memoria dinámica.
- Aprovecha el cache del computador.

¿Cuándo es impráctico el sondeo lineal? Como desventajas podemos enunciar:

- El direccionamiento abierto está supeditado a una estimación correcta del número de elementos a manejar. Si esto falla, entonces el agrupamiento degrada considerablemente el desempeño.

En ocasiones no es posible disponer de una buena estimación. En estos casos, así como si se desea una tabla hash general, es preferible el encadenamiento separado.

- La eliminación en el sondeo lineal conlleva eventuales inconvenientes. El algoritmo de eliminación que desarrollamos tiene la desventaja de mover (por copia) los contenidos de las cubetas. No podemos, pues, mantener punteros a los elementos de una tabla con sondeo lineal.

Si apelamos a la eliminación más simple, es decir, sólo a marcar la cubeta con `DELETED`, entonces no podemos permitir muchas eliminaciones, pues éstas degradan la búsqueda.

¿Es posible superar el sondeo lineal? La cuestión equivale a encontrar una forma de evitar el agrupamiento primario. Abordajes a esta pregunta en las subsecciones siguientes.

#### 5.1.4.4 Sondeo cuadrático

El sondeo cuadrático es fundamentalmente simple. Si ocurre una colisión en la posición  $i$ , entonces la próxima cubeta a sondar está dada por  $i^2 \bmod M$ . La idea de esta estrategia es evitar el agrupamiento primario y la formación de cadenas lineales de colisiones que son las que degradan el desempeño en el sondeo lineal.

Pero esta estrategia conlleva el problema de que es difícil garantizar que todas las cubetas sean sondeadas, pues  $i^2 \bmod M$  no necesariamente cubre todo el rango  $[0, M)$ .

Cuando la tabla alcanza el 50% de plenitud, no hay garantía absoluta de que el sondeo cuadrático encuentre una cubeta disponible.

Weiss [180] muestra que es posible encontrar tal cubeta si  $M$  es primo. Por añadidura, con  $M$  primo y  $M = 4k + 3$ , el sondeo cuadrático inspecciona toda la tabla.

El sondeo cuadrático impide la formación de cadenas lineales de colisión, pero no emula el sondeo ideal porque la operación  $i^2$  no es aleatoria ni simula un comportamiento aleatorio. Consecuentemente, con el sondeo cuadrático se presenta el “agrupamiento secundario” y la formación de cadenas en torno a las colisiones  $i^2 \bmod M$ . Tiene la desventaja, además, de ser más complejo de implantar y de imponer restricción a la selección de  $M$  a un número primo, algo que no siempre es posible.

#### 5.1.4.5 Doble hash

¿Es posible instrumentar el sondeo ideal? Consideraremos disponer de  $M$  distintas funciones hash  $\{h_1(k), h_2(k), \dots, h_M(k)\}$  con comportamientos aleatorios e independientes. En esta situación podemos sondear idealmente si invocamos primigeniamente a  $h_1(k)$  y, si hay colisión, entonces sondeamos las cubetas en el orden  $h_2(k), h_3(k), \dots, h_M(k)$  hasta encontrar una cubeta disponible. Este escenario es perfectamente factible, pero ya debe sernos evidente su dificultad, no sólo en cuanto a la escala de  $M$ , sino en cuanto a su generalidad respecto al tamaño  $M$ , así como el tipo de clave para el cual se instrumente la familia de funciones hash.

A pesar de las consideraciones anteriores hay una excelente y relativamente barata manera de emular el sondeo ideal consistente en utilizar dos funciones hash para los primer y segundo sondeos respectivamente. Si los dos sondeos aleatorizados causan colisión, entonces usamos sondeo lineal. La idea fundamental es emular el sondeo ideal para la primera colisión. Intuitivamente, si recordamos la paradoja del cumpleaños, esto tiene sentido porque la probabilidad de triple o más colisión decrece exponencialmente ( $\alpha^3, \alpha^4, \dots$  y así sucesivamente).

Eventualmente, es perfectamente realizable, aunque en tiempo constante más costoso, un triple y, en general, un  $m$ -hash.

En  $\mathcal{ALEPH}$ , una tabla hash con resolución de colisiones por direccionamiento abierto y doble función hash es exportada por el TAD `ODhashTable`, el cual se especifica en el archivo `<tpl_odhash.H 403>`:

403  
`<tpl_odhash.H 403>≡`  
`template <typename Key, typename Record,`  
`class Cmp = Aleph::equal_to<Key> >`  
`class ODhashTable`  
`{`  
`(Miembros privados de ODhashTable 404)`  
`(Miembros públicos de ODhashTable 405b)`  
`};`

En la práctica, `ODhashTable` es en interfaz idéntica a `OLHashTable`, la única diferencia estriba en que el constructor requiere la segunda función hash.

#### El problema de la eliminación con el doble hash

Cuando estudiamos el sondeo lineal, explicamos el problema que plantea al desempeño el hecho de que las cubetas deban marcarse con el valor `DELETED`. En aquel entonces

desarrollamos una técnica reminiscente a “la cerradura de brecha”, que aplicamos al TAD `OLHashTable` y que nos permite paulatinamente ir eliminando el status `DELETED`. Con el doble hash no es posible esta técnica porque pueden existir claves tales que  $h_1(k) = h_2(k)$ , lo cual puede acarrear una situación en la que sea imposible determinar si la clave colisionó en el primer o segundo sondeo.

En general, el problema es el mismo si usamos tres o más funciones hash o, si se descubriese el caso, empleamos sondeo ideal.

Por añadidura, el método de cerrar brechas de cubetas con status `DELETED` (§ 5.1.4.2 (Pág. 398)) mueve los contenidos de las cubetas, movimiento indeseable en algunas situaciones.

Si usamos doble hash, ¿cómo, entonces, tratamos con la eliminación? Hay dos enfoques, descubiertos por primera vez por dos investigadores japoneses [68], que no mueven las cubetas y que a su vez eliminan la entropía causada por las cubetas `DELETED`:

1. Se mantiene la cuenta de las cubetas con status `DELETED`. Cuando el nivel de carga de cubetas `DELETED` alcance un umbral dado, por ejemplo, un 30%, todas las cubetas con valor `DELETED` se marcan con valor `EMPTY`. Luego, se recorre enteramente la tabla y se examina cuáles cubetas con status `BUSY` se encuentran en una cadena de sondeo tal que se requiera marcar cubetas intermedias con status `DELETED`.

Esta técnica tiene el inconveniente de que consume tiempo  $\mathcal{O}(M)$ , lo cual es bastante notable respecto al desempeño esperado de  $\mathcal{O}(1)$ . Además, es difícil posibilitar la ejecución de las demás operaciones mientras se efectúa esta “limpieza”.

Eventualmente, en menoscabo de tener que mover algunas cubetas, aquellas colindantes con status `BUSY`, que sean producto del sondeo lineal, pueden relocalizarse mediante simple reinserción.

2. La otra técnica, “en línea” y con duración constante, es mantener un contador de sondeos en cada cubeta que denominaremos `probe_counter`. Inicialmente, cada cubeta tiene status `EMPTY` y contador en cero. Según ocurran inserciones y las cubetas estén en una cadena de sondeo, se incrementan los contadores de las cubetas componentes de la cadena. Análogamente, cuando ocurren eliminaciones se decrementan las cubetas entre el primer sondeo y la cubeta eliminada. Cuando el contador de una cubeta con status `DELETED` deviene cero, entonces, puesto que la cubeta no rompe ninguna cadena de sondeo, ésta puede marcarse con seguridad como `EMPTY`.

Para el tipo `ODhashTable` emplearemos la segunda técnica.

### Atributos de `ODhashTable`

En función de la técnica de eliminación que recién hemos presentado y seleccionado, lo primero que debemos hacer es definir la estructura de la cubeta:

404

*(Miembros privados de `ODhashTable` 404)≡*

(403) 405a▷

```
struct Bucket
{
 Key key; // clave
 Record record; // registro
 unsigned status : 4; // status EMPTY, DELETED o BUSY
 unsigned probe_type : 4; // FIRST_PROBE SECOND_PROBE LINEAR_PROBE
```

```
 unsigned short probe_counter; // contador de sondeos
};
```

Los tres primeros atributos se definen de forma idéntica a la cubeta de OLHashTable. El atributo probe\_type indica si una clave almacenada en la cubeta es resultante del primero (FIRST\_PROBE) o segundo (SECOND\_PROBE) sondeos o del sondeo lineal (LINEAR\_PROBE) a partir de la tercera colisión.

Notemos que status y probe\_type son campos de bits (niblas) y que ambos conforman un byte exacto.

El último atributo, probe\_counter, indica, tal como acabamos de explicar, la cantidad de claves que colindan en una cadena de colisiones que parte desde el primer sondeo con la primera función hash, hasta el último sondeo con la segunda función hash o con alguna cantidad de sondeos lineales. Por ejemplo, probe\_counter == 4, entonces esto indica que esa cubeta está en el camino de sondeo que comienza por la primera función hash, luego, por la segunda función hash, más dos cubetas sondeadas linealmente.

Los atributos de la propia tabla se definen de la siguiente manera:

405a *(Miembros privados de ODHashTable 404)* +≡ (403) ◁ 404 406a ▷

```
Bucket * table; // arreglo de cubetas
Hash_Fct first_hash_fct; // primera función hash
Hash_Fct second_hash_fct; // segunda función hash
size_t M; // tamaño de la tabla
size_t N; // número de cubetas ocupadas
size_t deleted_entries_counter; // número de cubetas DELETED
size_t empty_entries_counter; // número de cubetas EMPTY
```

los cuales se inicializan en el constructor de la siguiente manera:

405b *(Miembros públicos de ODHashTable 405b)* +≡ (403) 405c ▷

```
ODHashTable(Hash_Fct __first_hash_fct, Hash_Fct __second_hash_fct,
 const size_t & len)
: table(new Bucket[len]), first_hash_fct(__first_hash_fct),
 second_hash_fct(__second_hash_fct), M(len), N(0),
 deleted_entries_counter(0), empty_entries_counter(M) {}
```

## Búsqueda

De las operaciones que requieren sondeos, la búsqueda es la más simple:

405c *(Miembros públicos de ODHashTable 405b)* +≡ (403) ◁ 405b 406b ▷

```
Record * search(const Key & key)
{
 int i = (*first_hash_fct)(key) % M; // 1er sondeo (1ra fun hash)
 if (table[i].status == EMPTY)
 return NULL;

 if (table[i].status == BUSY and Cmp() (table[i].key, key))
 return &table[i].record;

 i = (*second_hash_fct)(key) % M; // 2do sondeo (2da fun hash)
 if (table[i].status == BUSY and Cmp() (table[i].key, key))
 return &table[i].record;

 // sondeo lineal a partir de índice de 2da función hash
 for (int count = 0; count < M and table[i].status != EMPTY; ++count)
```

```

{
 advance_index(i);
 if (table[i].status == BUSY and Cmp() (table[i].key, key))
 return &table[i].record;
}
return NULL; // sondeo no encontró la clave
}

```

### Inserción

Para mejorar la compresión del algoritmo de inserción, condición esencial para la legalidad y correctitud, encapsularemos la reservación de la cubeta en la siguiente rutina:

406a *(Miembros privados de ODhashTable 404) +≡ (403) ◁ 405a 407a ▷*

```

Record* allocate_bucket(Bucket & bucket, unsigned char probe_type,
 const Key & key, const Record & record)
{
 if (bucket.status == EMPTY)
 -empty_entries_counter;
 else
 -deleted_entries_counter;

 ++N;
 bucket.key = key;
 bucket.record = record;
 bucket.status = BUSY;
 bucket.probe_type = probe_type;
 bucket.probe_counter++;

 return &bucket.record;
}

```

La rutina recibe una cubeta en la cual se desea insertar por copia el par (key, record) con sondeo de tipo probe\_type (FIRST\_PROBE, SECOND\_PROBE o LINEAR\_PROBE).

La rutina anterior permite concentrar la inserción en el asunto interés de este estudio: la manera de sondear con dos funciones hash y luego lineal si ocurren tres o más colisiones. Básicamente, independientemente de su status, a cada cubeta sondeada durante la inserción, debe incrementársele su contador de colisiones:

406b *(Miembros públicos de ODhashTable 405b) +≡ (403) ◁ 405c 408 ▷*

```

Record* insert(const Key & key, const Record & record)
{
 if (N >= M)
 throw std::overflow_error("Hash table is full");

 int i = (*first_hash_fct)(key) % M; // sondeo con 1ra función hash
 if (table[i].status != BUSY) // ¿cubeta disponible?
 return allocate_bucket(table[i], FIRST_PROBE, key, record);

 table[i].probe_counter++; // por aquí sondeará colisión
 i = (*second_hash_fct)(key) % M; // sondeo con 2da función hash
 if (table[i].status != BUSY) // ¿cubeta disponible?
 return allocate_bucket(table[i], SECOND_PROBE, key, record);

 do // sondear linealmente a partir de índice de segundo sondeo e
 // incrementar cada contador de cubeta sondeada

```

```

 { // por esta cubeta se sondará una colisión ==> ++contador
 table[i].probe_counter++;
 advance_index(i);
 }
 while (table[i].status == BUSY); // parar si DELETED o EMPTY

 return allocate_bucket(table[i], LINEAR_PROBE, key, record);
}

```

### Eliminación

La eliminación es la operación más delicada de este TAD. Ésta es, en lo que concierne al sondeo, inversa a la inserción: los contadores de las cubetas sondeadas deben decrementarse. Por añadidura, es durante esta operación cuando se detectan cubetas que pueden marcarse como EMPTY al verificar que sus contadores devienen en cero.

Así pues, a efectos de expresar la eliminación sólo en términos del sondeo, el decremento del contador de una cubeta y su eventual marcado como EMPTY se hace mediante la siguiente rutina:

407a *(Miembros privados de ODhashTable 404) +≡* (403) ◁ 406a 407b ▷

```

void decrease_probe_counter(Bucket * bucket)
{
 bucket->probe_counter--;
 if (bucket->probe_counter == 0) // ¿marcar EMPTY sobre la cubeta?
 {
 // sí
 bucket->status = EMPTY;
 -deleted_entries_counter;
 ++empty_entries_counter;
 }
}

```

Del mismo modo, la cubeta a eliminar es “des-liberada” mediante una rutina simétrica al `allocate_bucket()` empleado para la inserción:

407b *(Miembros privados de ODhashTable 404) +≡* (403) ◁ 407a ▷

```

void deallocate_bucket(Bucket * bucket)
{
 bucket->probe_counter--;
 if (bucket->probe_counter == 0)
 {
 bucket->status = EMPTY;
 bucket->probe_type = NO_PROBED;
 ++empty_entries_counter;
 }
 else
 {
 bucket->status = DELETED;
 ++deleted_entries_counter;
 }
 -N;
}

```

Al igual que en las tablas hash anteriores, la eliminación recibe un puntero al registro, lo cual hace más simple el algoritmo, pues ahorra código para la búsqueda de la cubeta. Esto, aunado a las dos rutinas anteriores, nos permitirá diseñar una eliminación que sólo

se concentre en decrementar los contadores entre el primer sondeo y la cubeta a eliminar. De aquí, pues, la utilidad del atributo `probe_type`, pues éste nos permite determinar en qué punto de la cadena de sondeo se encuentra la cubeta:

408 ⟨Miembros públicos de ODhashTable 405b⟩+≡ (403) ◁ 406b 409 ▷

```

void remove(Record * record)
{
 Bucket * bucket = record_to_bucket(record);
 if (bucket->probe_type != FIRST_PROBE)
 {
 const int i_fst_probe = (*first_hash_fct)(bucket->key) % M;
 decrease_probe_counter(&table[i_fst_probe]);
 if (bucket->probe_type == LINEAR_PROBE)
 { // cubeta apartada durante sondeo lineal ==> decrementar 2do
 // sondeo y a partir de aquí decrementar todas las cubetas
 // hasta llegar a la que vamos a eliminar
 const int i_snd_probe = ((*second_hash_fct)(bucket->key) % M);
 decrease_probe_counter(&table[i_snd_probe]);

 int i = i_snd_probe;
 const int last_index = bucket_to_index(bucket);
 for (advance_index(i); i != last_index; advance_index(i))
 decrease_probe_counter(&table[i]);
 }
 deallocate_bucket(bucket);
 }
}

```

#### 5.1.4.6 Análisis informal del doble hash

Como ya hemos reiterado, el doble hash emula el sondeo ideal. Observaciones empíricas [99] indican que esta emulación es bastante certera. Así pues, en la práctica, el comportamiento del doble hash es equiparable al del sondeo ideal y las curvas mostradas en las figuras 5.5 (pag. 400) y 5.6 (pag. 401) corroboran esa aseveración.

Al 90% de plenitud, el doble hash efectúa 10 sondeos para una búsqueda fallida y, muy importante, 2,5 sondeos para una búsqueda exitosa. Estas cotas, expresadas en función de  $\alpha$  e independientes del valor de  $M$ , hacen que el desempeño esperado de la búsqueda, exitosa o fallida, sea constante. Consecuentemente, la inserción y la búsqueda con el doble hash son  $\mathcal{O}(1)$  hasta un 90% de plenitud. Después de este umbral, el desempeño puede degradarse a  $\mathcal{O}(M)$ .

Si escogemos, por seguridad, un 90% como tolerancia de plenitud, entonces el 10-20% de diferencia respecto al sondeo lineal puede ofrecernos una ganancia importante en espacio que varía según sea el tamaño de la clave y del registro. ¿Cuál es la ganancia en tiempo?

Abordar la pregunta anterior es delicado porque el doble hash no sólo es en tiempo constante más costoso que el sondeo lineal, sino que es bastante probable que el segundo sondeo cause una ruptura del cache. Así que, por su simplicidad y buen desempeño general, el sondeo lineal es la escogencia de facto para conjuntos pequeños y medianos en los cuales sea estimable el tamaño, o en situaciones de programación rápida tales como el prototipeo o la contingencia.

La bondad del doble hash se aprecia cuanto mayores sean  $M$  y  $N$  en los cuales los 10 sondeos que toma la búsqueda fallida y la inserción, son notabilísimamente mejores que los 63 sondeos que en promedio tomaría la búsqueda fallida si la tabla estuviese al 90% de plenitud. En cuanto al espacio, a mayores valores de  $M$  y  $N$ , mayor es el desperdicio en cubetas vacías que requeriríamos con el sondeo lineal.

Pasado el 90% de plenitud, el doble hash exhibe agrupamiento secundario y se degradan sus operaciones. Es extremadamente importante garantizar esta cota.

En síntesis, los criterios que conducen a escoger doble hash como estrategia de sondeo se resumen en:

1. Gran cantidad de claves, lo cual implica que el valor a escoger de  $M$  es alto y el desperdicio en cubetas desocupadas es mayor con el sondeo lineal.
2. Poco margen de desperdicio, por ejemplo, poca memoria.

Si el estimado sobre  $N$  es preciso y no muy grande, entonces el sondeo lineal es la selección; si es  $N$  es grande, entonces, doble hash. Si el estimado de  $N$  es impreciso, entonces el encadenamiento separado es la mejor estrategia.

### 5.1.5 Reajuste de dimensión en una tabla hash

Luego de los análisis realizados, no debe costarnos aprehender que, independientemente de la estrategia para tratar con las colisiones, cuanto más pequeño sea  $\alpha$ , menor es la probabilidad de colisión y, consecuentemente, mejor es el desempeño de la estrategia. Por otra parte, aunque sea ocasional, la mala suerte existe y, eventualmente, o el conjunto de claves supera la estimación de  $N$ , o el mal azar causa una cantidad alta de colisiones. ¿Puede hacerse algo al respecto?

Hay dos maneras de tratar con la mala suerte. La primera, basada en el encadenamiento separado y que será explicada en § 5.1.7 (Pág. 412), reubica el arreglo y las cubetas para garantizar una longitud máxima de cada lista de colisiones. La segunda, objeto de esta subsección, consiste simplemente en apartar un nuevo arreglo con  $M$  mayor y reubicar las claves.

De este modo, el reajuste con el doble hash puede plantearse de la siguiente manera:

409

```
(Miembros públicos de ODhashTable 405b) +≡ (403) ▷ 408
const size_t & resize(const size_t & new_size)
{
 Bucket * new_table = new Bucket [new_size]; // aparta nuevo arreglo
 Bucket * old_table = table; // respaldar valores de antigua tabla
 const size_t old_M = M;
 table = new_table; // tabla a nuevo arreglo
 M = new_size;
 empty_entries_counter = M;
 deleted_entries_counter = N = 0;

 for (int i = 0; i < old_M; ++i) // reinserir en nueva tabla
 if (old_table[i].status == BUSY)
 insert(old_table[i].key, old_table[i].record);

 delete [] old_table;
```

```

 return M;
}

```

La operación es, conceptual y estructuralmente, sencilla e idéntica para el sondeo lineal. El criterio para invocarla lo determina el que el valor de  $\alpha$  supere o se aproxime al umbral en que se predice un buen desempeño. En el caso del doble hash, podríamos invocar el reajuste cuando nos aproximemos al 90% de plenitud.

Es plausible reajustar según una plenitud de desperdicio de cubetas; esta estrategia tiene bastante sentido si las inserciones cesan y la tabla se usa principalmente para la búsqueda. En este caso podríamos reajustar la tabla a un  $M$  inferior de manera que disminuyamos el desperdicio en cubetas y respetemos el umbral de desempeño.

Hay dos cuestionamientos a esta operación. El primero de ellos es su coste  $\mathcal{O}(M) + \mathcal{O}(M')$ , duración en la cual sería prohibitivo ejecutar cualquier otra operación. Este eventual problema se mitiga, mas no se evita del todo, contabilizando algún tiempo de ocio, es decir, una vez que  $\alpha$  devenga cercana al umbral, entonces esperamos por un tiempo prudente a ver si no ocurren operaciones; si expira el tiempo de espera, entonces procedemos a reajustar bajo la esperanza de que no estorbemos a las otras operaciones.

El segundo cuestionamiento, válido sólo para las tablas cerradas, es que se mueven los contenidos de las cubetas. Con el encadenamiento separado no tenemos este problema.

El reajuste en el encadenamiento separado tiene dos ventajas. La primera, ya mencionada, es la posibilidad de  $N > M$ . La segunda radica en simplicidad de tomar previsiones para permitir la inserción durante el proceso de reajuste.

410 (Métodos públicos de GenLhashTable 386a)+≡ (384a) ↳387

```

size_t resize(const size_t & new_size)
{
 BucketList * new_table = new BucketList [new_size];
 BucketList * old_table = table; // guardar estado tabla actual
 const size_t old_size = M;
 table = new_table; // tabla vacía con nuevo arreglo
 M = new_size;
 busy_slots_counter = N = 0;

 for (int i = 0; i < old_size; ++i) // reiniciar cubetas
 // recorrer lista colisiones en old_table[i]
 for (BucketItr it(old_table[i]); it.has_current(); /* Nada */)
 insert((Bucket*) it.del()); // eliminar e insertar en table[]

 delete [] old_table; // liberar memoria antigua tabla

 return M;
}

```

### 5.1.6 El TAD DynLhashTable (cubetas dinámicas)

¿Cuál es la manera más general de exportar un TAD basado en una tabla hash? Se trataría de un mapeo entre claves y registros, cuya interfaz y uso son bastante similares a los mapeos que realizamos con árboles binarios de búsqueda (tipo DynMapTree- § 4.10 (Pág. 332)).

Los tipos de tablas hash OLHashTable y ODhashTable satisfacen este requerimiento con la salvedad de que éstos limitan la cantidad de claves al valor de M seleccionado en construcción. El TAD LhashTable<Key> no tiene este problema, pero éste, no maneja registros directamente ni trata con la memoria dinámica.

Hay una alternativa que, en detrimento de un poco de tiempo constante, desarrollaremos en § 5.1.7 (Pág. 412). Por los momentos, en esta sección extenderemos el tipo LhashTable<Key> para que maneje un rango asociado a el conjunto de claves y trate con la memoria dinámica. El tipo en cuestión lo llamamos DynLhashTable, el cual modeliza una tabla hash, con resolución de colisiones encadenada, que asocia elementos de tipo key con registros de tipo Record y que se especifica en el archivo *<tpl\_dylhash.H 411a>*, cuya estructura general es como sigue:

```
411a <tpl_dylhash.H 411a>≡
 template <typename Key, typename Record, class Cmp = Aleph::equal_to<Key> >
 class DynLhashTable : public LhashTable<Key>
 {
 <Miembros privados de DynLhashTable 411b>

 typedef typename DynLhashTable<Key, Record>::Hash_Fct Hash_Fct;

 <Miembros públicos de DynLhashTable 411c>
 };

```

La tabla hash puede accederse como un arreglo donde los índices son claves de tipo key y los contenidos del arreglo son registros de tipo Record. A tal fin, DynLhashTable exporta sobrecargas al operador [] .

Como se ve, DynLhashTable hereda, tanto parte de su interfaz como parte de su implantación, de LhashTable<Key>. Métodos como los observadores y el reajuste ya están implantados por LhashTable<Key>. Lo mismo se puede decir para el manejo de la estrategia. La labor de DynLhashTable se remite entonces a la definición del rango y al manejo de memoria.

La definición del rango se realiza mediante una cubeta derivada de LhashTable<Key>::Bucket:

```
411b <Miembros privados de DynLhashTable 411b>≡ (411a) 412b▷
 struct DLBucket : public LhashTable<Key>::Bucket
 {
 Record record;

 DLBucket(const Key & key, const Record & _record)
 : LhashTable<Key>::Bucket(key), record(_record) { /* Empty */ }
 };

```

Una vez definida la cubeta, las operaciones principales son conceptualmente sencillas, pues el manejo de la estrategia ya está instrumentado en LhashTable<Key>. La inserción es, pues, como sigue:

```
411c <Miembros públicos de DynLhashTable 411c>≡ (411a) 412a▷
 Record * insert(const Key & key, const Record & record)
 {
 DLBucket * bucket = new DLBucket (key, record);
 LhashTable<Key>::insert(bucket);
 return &bucket->record;
 }

```

El código es lineal porque la estrategia ya está implantada en LhashTable<Key>.

La búsqueda se remite a un wrapper a la búsqueda de LhashTable<Key>:

412a *(Miembros públicos de DynLhashTable 411c) +≡* (411a) ◁ 411c ▷ 412c ▷

```
Record * search(const Key & key)
{
 DLBucket * bucket = (DLBucket*) LhashTable<Key>::search(key);
 return bucket != NULL ? &bucket->record : NULL;
}
```

La eliminación requiere transformar un registro a una cubeta:

412b *(Miembros privados de DynLhashTable 411b) +≡* (411a) ◁ 411b

```
static DLBucket * record_to_bucket(Record * rec)
{
 DLBucket * ret_val = 0;
 size_t offset = (size_t) &ret_val->record;
 return (DLBucket*) (((size_t) rec) - offset);
}
```

Con este método, la eliminación también es muy simple:

412c *(Miembros públicos de DynLhashTable 411c) +≡* (411a) ◁ 412a

```
void remove(Record * record)
{
 DLBucket* bucket = record_to_bucket(record);
 LhashTable<Key>::remove(bucket);
 delete bucket;
}
```

La facilidad con se instrumenta el TAD DynLhashTable indicia, una vez más, la bondad del principio fin-a-fin (§ 1.4.2 (Pág. 22)).

### 5.1.7 Tablas hash lineales

Una condición sobre la cual se sustentan las predicciones de desempeño que nos ofrecen las diversas estrategias para tratar con las colisiones es que el factor de carga  $\alpha$  no exceda el umbral para el cual se cumple la predicción. Cuanto mayor sea la carga, independientemente de la estrategia para tratar con las colisiones, mayor será la probabilidad de que se degrade el desempeño.

Sabemos que el factor de carga  $\alpha$  puede disminuirse si reajustamos la tabla a un mayor valor de  $M$  y reubicamos las cubetas. Pero, como ya sabemos, este reajuste tiene un precio  $\mathcal{O}(M) + \mathcal{O}(M')$ . La estructura que trataremos en esta subsección ataña a la cuestión ¿puede irse aumentado el arreglo dinámicamente? Intuitivamente debemos saber que esto es posible si usamos un arreglo dinámico del tipo estudiado en § 2.1.4 (Pág. 34). Por este lado, no tenemos costes importantes -mayores que  $\mathcal{O}(1)$ - que pagar por el hecho de relocatear la tabla. El asunto se relega a encontrar una manera de cambiar dinámicamente el valor de  $M$ .

Al enfoque de tabla hash que estudiaremos e instrumentaremos en esta subsección, cual realiza la gestión de dinámica de  $M$ , se le llama “dispersión lineal” (linear hashing). La técnica fue descubierta por vez primera para almacenamiento secundario [110] y esta presentación está basada en la de Larson [105].

El enfoque utiliza un arreglo dinámico `DynArray<T>`. El valor de  $M$  varía dinámicamente en función de  $\alpha$ . De este modo, las predicciones de desempeño siempre se cumplen sin necesidad de pagar por el reajuste.

La tabla hash lineal está implantada mediante el tipo `LinearHashTable<Key>`, el cual se especifica en el archivo `(tpl_linHash.H 413a)`, cuya especificación de base es la siguiente:

```
413a <tpl_linHash.H 413a>≡
 template <typename Key, template <class> class BucketType,
 class Cmp = Aleph::equal_to<Key> >
 class GenLinearHashTable
 {
 <Miembros privados de LinearHashTable<Key> 413b>
 <Miembros públicos de LinearHashTable<Key> 418a>
 };
 template <typename Key, class Cmp = Aleph::equal_to<Key> >
 class LinearHashTable : public GenLinearHashTable<Key, LhashBucket, Cmp>
```

Las estructuras de las cubetas son exactamente las mismas que las del tipo `LhashTable<Key>` estudiado en § 5.1.3.1 (Pág. 383).

### 5.1.7.1 Expansión/contracción de una tabla hash lineal

Siendo `table[]` un arreglo dinámico, su dimensión se aumenta y se contrae dinámicamente en tiempo  $\mathcal{O}(1)$ . Para ello se manejan los siguientes atributos umbrales:

```
413b <Miembros privados de LinearHashTable<Key> 413b>≡ (413a) 413c▷
 float upper_alpha; // factor de carga superior
 float lower_alpha; // factor de carga inferior
```

`LinearHashTable<Key>` mantiene permanentemente el factor de carga  $\alpha = N/M$ . Después de una inserción,  $\alpha$  se compara con `upper_alpha`; si  $\alpha \geq upper\_alpha$ , entonces la tabla se expande tantas unidades como sea necesario para llevar la carga a un valor inferior a `upper_alpha`; durante este proceso, algunas cubetas son reubicadas. Simétricamente, si luego de una eliminación,  $\alpha \leq lower\_alpha$ , entonces la tabla se contrae una o más veces hasta que la carga esté por debajo de `lower_alpha`; igualmente, algunas cubetas se reubican. El “truco” de la dispersión lineal estriba pues en la manera en que linealmente se expande y contrae la tabla, así como en la forma en que se maneja la función hash.

Los valores iniciales de `upper_alpha` y `lower_alpha` se especifican durante la construcción de un objeto `LinearHashTable<Key>`, el cual exporta modificadores que permiten ajustar dinámicamente los valores umbrales de carga.

El estado de expansión o contracción de la tabla se mantiene mediante los siguientes dos atributos:

```
413c <Miembros privados de LinearHashTable<Key> 413b>+≡ (413a) ▷413b 414a▷
 size_t p; // índice de la lista que se partitiona (o aumenta)
 size_t l; // cantidad de veces que se ha duplicado la tabla
```

El arreglo siempre crece o decrece por su extremo derecho. El índice `p` se referencia de dos maneras:

1. `table[p + M]` es la entrada del arreglo por donde éste se expande o contrae, según sea la operación y el valor del factor de carga  $\alpha$ .

2. `table[p]` contiene la lista de colisiones que se partitionaría si ocurriese una expansión. Cuando ésta ocurre, algunas cubetas de la lista `table[p]` se pasan a la lista `table[M + p]`. Simétricamente, cuando ocurre una contracción, todas las cubetas de `table[M + p]` se trasladan a `table[p]`.

Cuando se expande se incrementa el índice `p`; simétricamente, se decrementa cuando se contrae.

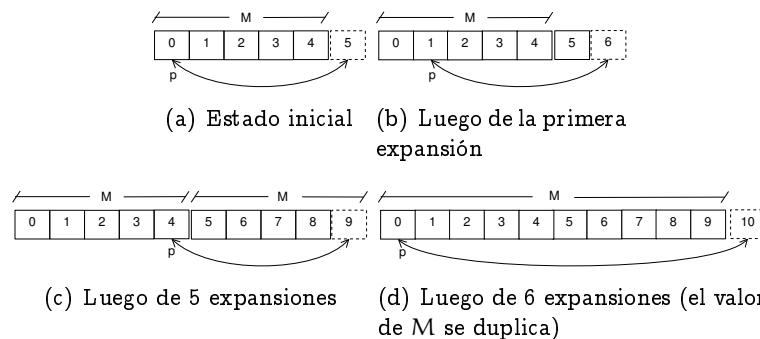


Figura 5.7: Proceso de expansión lógica de la tabla para  $M = 5$

Para poder calcular el factor de carga debemos conocer en todo momento el tamaño lógico actual de la tabla ( $M'$ ); este valor lo mantendremos en el siguiente atributo:

414a *(Miembros privados de LinearHashTable<Key> 413b) +≡ (413a) ▷ 413c 414b ▷*  
`size_t MP; // guarda el valor  $p + M$`

Por añadidura, este atributo nos pone a directa disposición el cálculo  $p + M$ .

Cuando  $p = 2M$ , el tamaño lógico de la tabla se duplica y se reinician  $M = 2M$ ,  $p = 0$  y  $MP = M$ . En todo momento,  $0 \leq p \leq 2^lM - 1$ . La función del atributo `l` es entonces acotar la contracción al valor original de  $M$  que haya sido especificado en construcción.

En cada expansión/contracción hay que estar pendiente del predicado `p == 2M`. Para ganar un poco de tiempo de cálculo, el producto  $2M$  se guarda en el atributo `MM = 2M`:

414b *(Miembros privados de LinearHashTable<Key> 413b) +≡ (413a) ▷ 414a 415b ▷*  
`size_t MM; // producto  $2 * M$`

El proceso de expansión se ilustra en la figura 5.7 para  $M = 5$ .

En virtud de lo explicado, al momento de expandir la tabla los valores de `p`, `l`, `M`, `[MP` y `MM` se actualizan de la siguiente manera:

414c *(Actualizar estado de expansión 414c) ≡ (417a)*  
`++p;  
++MP;  
if (p == M) // (p == 2*M) ¿debe duplicarse el tamaño de la tabla?  
{ // sí ==> modificar el tamaño de la tabla a 2*M  
++l; // Cantidad de veces que se ha duplicado la tabla  
p = 0;  
MP = M = MM; // se les asigna 2*M  
MM = multiply_by_two(MM);  
}`

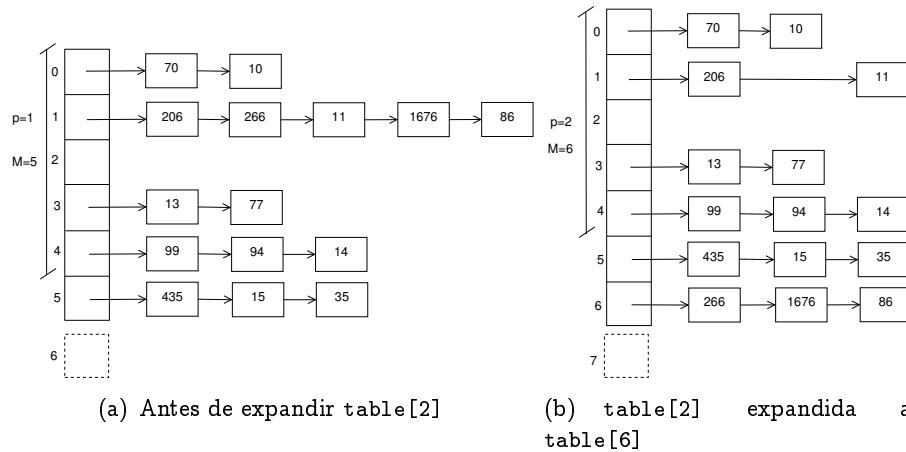


Figura 5.8: Ejemplo de partición/contracción de una lista

Simétricamente, la contracción de la tabla es el proceso inverso:

415a *(Actualizar estado de contracción 415a)* ≡ (417c)

```

if (p == 0) // ¿debe dividirse entre 2 el tamaño de la tabla?
{
 // si ==> actualizar tamaño de la tabla a M/2
 -1; // Cantidad de veces que se ha duplicado la tabla disminuye
 MM = M; // se divide entre dos
 M = divide_by_two(M);
 p = M - 1;
}
else
 -p; // no ==> sólo reducir índice p
-MP;

```

El enfoque dinámico de expansión/contracción plantea un problema con la determinación del índice en la tabla y el valor que arroje la función hash. Dada una clave  $k$ , tradicionalmente, el índice en la tabla se determina mediante:

$$h(k) \bmod M ;$$

pero este valor abarca el rango de entradas  $[0, M - 1]$  y deja por fuera el rango de entradas expandidas  $[M, M + p]$ . El problema se resuelve mediante el valor de  $p$  y modificando la llamada a la función hash de la siguiente manera:

415b *(Miembros privados de LinearHashTable<Key> 413b)* ≡ (413a) ◁ 414b 417a ▷

```

size_t call_hash_fct(const Key & key) const
{
 const size_t hash = (*hash_fct)(key);
 const size_t i = hash % M;
 return i < p ? hash % MM : i;
}

```

Defines:

`call_hash_fct`, used in chunk 418.

Recordemos que  $p$  es el índice de la lista de expansión/contracción. En el rango  $[0, M - 1]$ , la tabla contiene cubetas asignadas por

$$(*\text{hash\_fct})(\text{key}) \bmod M , \quad (5.29)$$

mientras que en el rango  $[M, M + p]$ , las cubetas se asignan por

$$(*\text{hash\_fct})(\text{key}) \bmod 2M . \quad (5.30)$$

Cuando  $p = 2M$ , entonces todo el rango  $[0, 2M - 1]$  puede ser cubierto con el módulo de  $2M$ .

Cuando ocurre una expansión se partitiona la lista de colisiones cuyo índice en la tabla es  $p$ . La lista la lista de colisiones  $\text{table}[p]$  se recorre enteramente y se invoca a `call_hash_fct()` para cada clave. Aquellas claves cuyo índice es el mismo de su entrada  $p$  permanecen en la lista, mientras que aquellas cuyo índice es distinto  $M + p$  son movidas hacia  $\text{table}[M + p]$ . La figura 5.8 ilustra los estados antes y después de una expansión.

El proceso de partición de una entrada descrito puede instrumentarse de la siguiente forma:

```

416 <Particionar lista 416>≡ (417a)
 BucketList * src_list_ptr = table.test(p);
 if (src_list_ptr != NULL) // ¿table[p] está escrita?
 if (not src_list_ptr->is_empty()) // ¿table[p] no está vacía?
 {
 BucketList * tgt_list_ptr = NULL;

 // recorrer lista colisiones y mover cubetas de table[p+M]
 for (BucketItor it(*src_list_ptr); it.has_current(); /* nada */)
 {
 Bucket * bucket = static_cast<Bucket*>(it.get_current());

 it.next(); // avance al siguiente elemento de la lista

 const Key & key = bucket->get_key();
 const int i = (*hash_fct)(key) % MM;
 if (i == p) // ¿pertenece esta clave a table[p]?
 continue; // sí ==> clave sigue en table[p] ==> siguiente

 if (tgt_list_ptr == NULL)
 tgt_list_ptr = &table.touch(MP);

 // bucket no pertenece a table[p] sino a table[p+m] ==>
 // eliminar bucket de table[i] e insertarlo en table[p+m]
 bucket->del();
 tgt_list_ptr->append(bucket);
 }
 if (src_list_ptr->is_empty()) // ¿table[p] quedó vacía?
 -busy_slots_counter; // sí ==> un slot vacío

 ++busy_slots_counter; // uno nuevo por table[p+M]
 }

```

El bloque considera la posibilidad de que la lista `table[p]` esté vacía o, inclusive, que jamás haya sido referenciada. Del mismo modo, si ninguno de los elementos de `table[p]` pasa `table[MP]`, entonces es posible que la entrada `table[MP]` no ocupe memoria.

Entendidas (se presume) las vicisitudes de la expansión, podemos escribir una rutina que la ejecute y que sea invocada cada vez que ocurre una inserción:

```
417a <Miembros privados de LinearHashTable<Key> 413b>+≡ (413a) ◁415b 417c▷
 void expand()
 { // expandir la tabla hasta que la carga esté debajo de upper_alpha
 for (float alpha = 1.0*N/MP; alpha >= upper_alpha; alpha = 1.0*N/MP)
 {
 <Particionar lista 416>
 <Actualizar estado de expansión 414c>
 }
 }
```

Defines:

expand, used in chunks 418b and 444b.

Notemos que la expansión sólo se da a lugar si se excede el factor de carga.

El proceso de contracción puede interpretarse a la inversa del de expansión: pasar los elementos de la lista `table[M + p]` hacia la lista `table[p]`; también se ilustra en la figura 5.8 si se interpreta de derecha a izquierda. En este caso, el proceso -de contracción- es mucho más rápido, pues no es necesario recorrer ninguna de las listas, sólo concatenarle `table[M + p]` a `table[p]`, la cual la efectúa directa y constantemente una operación del TAD Dlink (§ 2.4.7 (Pág. 72)):

```
417b <Fusionar lista 417b>≡ (417c)
 if (MP < table.size()) // ¿Existe table[MP]?
 {
 BucketList * src_list_ptr = table.test(MP);
 if (src_list_ptr != NULL) // ¿existe entrada para table[p+M]?
 {
 if (not src_list_ptr->is_empty()) // ¿table[p+M] está vacía?
 {
 // no ==> fusionar las listas
 BucketList & tgt_list = table.touch(p); // aparta table[p]
 tgt_list.concat_list(src_list_ptr);
 -busy_slots_counter; // table[p+M] devino vacía
 }
 table.cut(MP); // eventualmente liberar memoria de table[p+M]
 }
 }
```

La rutina de contracción, cual se invoca con cada eliminación, se define entonces de la siguiente forma:

```
417c <Miembros privados de LinearHashTable<Key> 413b>+≡ (413a) ◁417a
 void contract()
 { // contraer la tabla hasta que carga esté debajo de lower_alpha
 for (float alpha = (1.0*N)/MP; alpha <= lower_alpha and MP > len;
 alpha = (1.0*N)/MP)
 {
 <Actualizar estado de contracción 415a>
 <Fusionar lista 417b>
 }
 }
```

Defines:

contract, used in chunk 418c.

Lo novedoso de una tabla hash lineal es el proceso de expansión/contracción. Por lo demás, estructuralmente, las operaciones son idénticas a las del resto de los enfoques estudiados.

### 5.1.7.2 Búsqueda en LinearHashTable<Key>

La búsqueda, como es tradición de una tabla hash, es la operación más simple:

418a *(Miembros públicos de LinearHashTable<Key> 418a) +≡* (413a) 418b ▷

```
Bucket * search(const Key & key)
{
 const int i = call_hash_fct(key);
 BucketList * list = table.test(i);
 if (list == NULL) // ¿Ha sido escrita alguna vez table[i]?
 return NULL; // No ==> el elemento no se encuentra en la tabla

 if (list->is_empty())
 return NULL;

 // buscar key en la lista de cubetas
 for (BucketItor it(*list); it.has_current(); it.next())
 {
 Bucket * bucket = static_cast<Bucket*>(it.get_current());
 if (Cmp() (key, bucket->get_key()))
 return bucket;
 }
 return NULL;
}
```

Uses call\_hash\_fct 415b.

La añadidura de esta búsqueda respecto a los otros tipos de tabla hash es el hecho de que, puesto que el arreglo dinámico aparta memoria en función de la primera escritura, es probable que haya entradas que no hayan sido apartadas.

### 5.1.7.3 Inserción en LinearHashTable<Key>

418b *(Miembros públicos de LinearHashTable<Key> 418a) +≡* (413a) ◁ 418a 418c ▷

```
Bucket* insert(Bucket * bucket)
{
 const int i = call_hash_fct(bucket->get_key());
 BucketList & list = table.touch(i); // aparta memoria table[i]
 if (list.is_empty())
 ++busy_slots_counter;

 list.append(bucket);
 ++N;
 expand();

 return bucket;
}
```

Uses call\_hash\_fct 415b and expand 417a.

### 5.1.7.4 Eliminación en LinearHashTable<Key>

418c *(Miembros públicos de LinearHashTable<Key> 418a) +≡* (413a) ◁ 418b

```

Bucket * remove(Bucket * bucket)
{
 Bucket * next = static_cast<Bucket*>(bucket->get_next());

 bucket->del(); // elimine de lista de colisiones

 if (next->is_empty()) // ¿lista de colisiones vacía?
 -busy_slots_counter; // si ==> un slot vacío

 -N;
 contract();

 return bucket;
}

```

Uses contract 417c.

### 5.1.7.5 Análisis de la dispersión lineal

Antes de enunciar la proposición fundamental, es menester notar que, salvo las expansiones y contracciones, una tabla hash lineal es similar a una tabla hash tradicional con resolución de colisiones por encadenamiento separado. Como tal, no nos debe ser extraño el analizar la dispersión lineal a partir del análisis realizado con el encadenamiento separado.

**Proposición 5.3** (Cantidad de cubetas revisadas en una búsqueda sobre una tabla hash lineal - Larson 1988 [105] ) Sea  $h(k) : \mathcal{K} \rightarrow [0, M-1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M-1]$ . Sea  $T$  una tabla hash lineal de longitud  $M$  con  $N$  elementos. Entonces, el promedio de cubetas que se visitan en una búsqueda fallida es:

$$\alpha \leq \overline{U_N} \leq \frac{9}{16} + \alpha ; \quad (5.31)$$

y el promedio para una búsqueda exitosa es:

$$1 + \frac{1}{M} + \frac{1}{\alpha} \leq \overline{S_N} \leq 1 + \frac{1}{M} + \frac{9}{16}\alpha . \quad (5.32)$$

**Demostración** Sea  $\bar{l}$  la longitud de cada una las listas de colisiones en el encadenamiento separado. Según el lema 5.1 (ecuación (5.7), pag. 388),  $\bar{l} = N/M$ .

Los lemas 5.2 (ecuación (5.8), pag. 388) y 5.3 (ecuación (5.9), pag. 388), que estudiamos con el encadenamiento separado, plantean los promedios cuando no ocurren expansiones o contracciones. En este sentido, asumiendo un factor de carga  $y$ , podemos definir las siguientes funciones:

$$u(y) = y ; \quad (5.33)$$

para el promedio de cubetas que se inspecciona en una búsqueda infructuosa y:

$$s(y) = 1 + \frac{1}{y} + \frac{1}{M} ; \quad (5.34)$$

para el el promedio de cubetas que se inspecciona en una búsqueda exitosa.

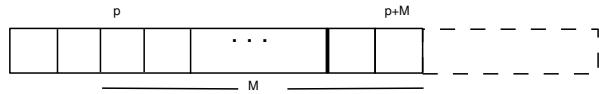
Una mirada más acuciosa permite interpretar  $u(y)$  y  $s(y)$  en el mismo sentido cuando  $p = 0$ , es decir, cuando no ha ocurrido ninguna expansión, la expansión acaba de duplicar

el valor  $M$  (véase *(Actualizar estado de expansión 414c)*) o la contracción acaba de dividir entre a  $M$  (véase *(Actualizar estado de contracción 415a)*).

Para tabla hash lineal de tamaño  $M$ , sus  $M$  listas de colisiones pueden distribuirse en:

- $p$  primeras listas que han sido particionadas.
- $M - p$  listas que no han sido particionadas, y
- $p$  listas resultantes de haber particionado las primeras  $p$  listas.

La situación se puede pictóricular de manera particular ( $p = 2$ ), así como también de manera general, así:



Las listas que no han sido particionadas contienen, en promedio,  $\bar{l}$  cubetas.

Bajo la suposición de aleatoriedad para la función hash, una lista de longitud promedio  $\bar{l}$  se partitiona en dos listas de tamaño equitativo  $\bar{l}/2$ .

En función de lo definido, el total de cubetas se reparte entonces en:

$$N = \underbrace{\frac{p\bar{l}}{2}}_{\text{particionadas}} + \underbrace{(M-p)\bar{l}}_{\text{no particionadas}} + \underbrace{\frac{p\bar{l}}{2}}_{\text{resultantes de partición}} = p\bar{l} + (M-p)\bar{l} = M\bar{l} \quad (5.35)$$

El factor de carga de una tabla hash lineal está entonces definido por:

$$\alpha = \frac{M\bar{l}}{p+M} \implies \bar{l} = \alpha \frac{p+M}{M} \quad (5.36)$$

Sea  $x = p/M$ , o sea, la proporción de listas que han sido particionadas. De este modo, (5.36) se define como:

$$\bar{l} = \alpha(1+x) \quad (5.37)$$

De (5.37) proviene la “linealidad”: la cantidad esperada de cubetas en una lista que no ha sido particionada aumenta linealmente de  $\alpha$  hacia  $2\alpha$ .

Cualquier búsqueda, exitosa o fallida, tiene probabilidad  $x$  de hacerse sobre una lista particionada o resultante de una partición. Del mismo modo, hay una probabilidad  $(1-x)$  de que la búsqueda depare en una lista que no ha sido particionada.

Si definimos  $U(x)$  como la cantidad esperada de cubetas a visitar en una búsqueda fallida cuando una proporción  $x$  de las listas de la tabla han sido particionadas, entonces, según (5.33), cual nos indica la cantidad de esperada de cubetas que se visitarán,  $U(x)$  se define como:

$$\begin{aligned} U(x) &= x u(\bar{l}/2) + (1-x) u(\bar{l}) = x u\left(\frac{\alpha(1+x)}{2}\right) + (1-x) u(\alpha(1+x)) \\ &= \frac{\alpha}{2}(2+x-x^2) \end{aligned} \quad (5.38)$$

De la misma manera, si definimos  $S(x)$  como la cantidad esperada de cubetas a visitar en una búsqueda exitosa cuando una proporción  $x$  de las listas de la tabla han sido particionadas, entonces, según (5.34), cual nos indica la cantidad de esperada de cubetas que

se visitarán,  $S(x)$  se define como:

$$\begin{aligned} S(x) &= x s(\bar{l}/2) + (1-x) s(\bar{l}) = x s\left(\frac{\alpha(1+x)}{2}\right) + (1-x) s(\alpha(1+x)) \\ &= x \left(1 + \frac{1}{M} + \frac{\alpha(1+x)}{2 \times 2}\right) + (1-x) \left(1 + \frac{1}{M} + \frac{\alpha(1+x)}{2}\right) \\ &= 1 + \frac{1}{M} + \frac{\alpha}{4}(2+x-x^2) \end{aligned} \quad (5.39)$$

En ambos tipos de búsqueda, el mínimo esperado de cubetas revisadas ocurre cuando la tabla no se ha expandido ( $x = 0$ ) o cuando la expansión ha duplicado el tamaño de la tabla ( $x = 1$ ). De este modo, las cotas inferiores están definidas por:

$$\begin{aligned} \overline{U_N} &= U(0) = \alpha \\ \overline{S_N} &= S(0) = 1 + \frac{1}{M} + \frac{\alpha}{2} \quad \square \end{aligned}$$

Notemos, como cabría esperarse, que estas son las mismas cotas que para el encadenamiento separado sin expansión/contracción.

Análogamente, el máximo esperado de cubetas revisadas ocurre cuando  $x = 1/2 \Rightarrow p = M/2$ , pues (5.38) y (5.39) son máximos cuando  $x - x^2$  es máximo. Esto equivale a decir que  $M/2$  listas han sido particionadas en  $M/2$  listas adicionales, mientras que quedan  $M/2$  listas sin particionar. Así pues:

$$\begin{aligned} \overline{U_N} &= U(0) = \frac{9}{8}\alpha \\ \overline{S_N} &= S(0) = 1 + \frac{1}{M} + \frac{9}{16}\alpha \quad \blacksquare \end{aligned}$$

El coste a lo largo de un ciclo de expansión está determinado por la proposición siguiente.

**Proposición 5.4** (Coste promedio de búsqueda en una tabla hash lineal - Larson 1988 [105]) Sea  $T$  una tabla hash lineal de longitud  $M$  con  $N$  elementos. Entonces, el coste promedio de búsqueda a lo largo de un ciclo de expansión está determinado por:

$$\overline{U_N}^M = \frac{13}{12}\alpha ; \quad (5.40)$$

para una búsqueda fallida y:

$$\overline{S_N}^M = \frac{(13\alpha + 12)M + 12}{12M} ; \quad (5.41)$$

para una búsqueda exitosa.

**Demostración** El resultado sale directamente de integrar (5.38) y (5.39):

$$\overline{U_N}^M = \int_0^1 \frac{\alpha}{2}(2+x-x^2) dx \quad (5.42)$$

y:

$$\overline{S_N}^M = \int_0^1 1 + \frac{1}{M} + \frac{\alpha}{4}(2+x-x^2) dx \quad \blacksquare \quad (5.43)$$

Así pues, en este tipo de tabla, el factor de carga siempre permanece constantemente acotado, lo cual posibilita enunciar el corolario siguiente.

**Corolario 5.2 Desempeño de las operaciones en una tabla hash lineal** Sea  $h(k) : \mathcal{K} \rightarrow [0, M - 1]$  una función hash  $\mathcal{O}(1)$  que distribuye uniformemente los elementos de  $\mathcal{K}$  hacia  $[0, M - 1]$ . Sea  $T$  una tabla hash lineal de longitud  $M$  con  $N$  elementos y factor de carga  $\alpha_l \leq \alpha \leq \alpha_h$ . Entonces, el desempeño esperado de las operaciones de inserción, búsqueda y eliminación es  $\mathcal{O}(1)$ .

**Demostración** La inserción requiere una búsqueda fallida, la cual requiere recorrer, según (5.40),  $\frac{13}{12}\alpha$  cubetas en promedio. Puesto que  $\alpha$  es constante, (5.40) es constante y la inserción es  $\mathcal{O}(1)$ . Lo mismo ocurre para la búsqueda fallida.

Para una búsqueda exitosa se recorren, según (5.43),  $\frac{(13\alpha+12)M+12}{12M}$  cubetas, cantidad que también es constante, pues  $\alpha$  es constante. La búsqueda exitosa es por tanto  $\mathcal{O}(1)$ .

Si la eliminación es como la diseñamos, entonces ésta es  $\mathcal{O}(1)$  de forma determinista. Si se basa en la búsqueda, entonces está caracterizada por la búsqueda exitosa, la cual ya demostramos es  $\mathcal{O}(1)$  ■

La gran bondad de este tipo de tabla es que no hay necesidad de ejecutar el costoso reajuste para mantener el factor de carga al valor que ofrezca el desempeño esperado; la carga es dinámicamente ajustada en tiempo constante. Independientemente de la cantidad de elementos, la carga siempre estará acotada y, por tanto, el desempeño será “constantemente esperado”. ¿Es, pues, este enfoque idóneo para todas las situaciones en que se requiera una tabla hash? Por supuesto que no, por varias razones.

En primer lugar, a causa del arreglo dinámico, por más que nos hayamos esforzado en una ejecución de alto desempeño cuando diseñamos el tipo `DynArray<T>`, éste acarrea costes constantes mayores que el del arreglo contiguo tradicional que empleamos con el encadenamiento separado. Así pues, para situaciones en las cuales se estime correctamente el valor de  $N$ , la dispersión lineal es el enfoque más lento de todos los que hemos estudiado. Dicho de otro modo, si se está en capacidad de estimar correctamente  $N$ , aun con un cierto margen de incertidumbre, entonces es preferible adoptar alguno de los enfoques que ya hemos estudiado en función de las consideraciones hechas en § 5.1.4.6 (Pág. 408). Los argumentos son simples: simplicidad y mejor desempeño.

Surge entonces la pregunta ¿cuándo debemos usar dispersión lineal?. Hay dos escenarios fundamentales:

1. Cuando no se conozca la estimación de  $N$ , cual es la situación cuando se manejan conjuntos de manera general.
2. Cuando, aunque se conozca una estimación máxima de  $N$ , éste fluctúe frecuentemente entre valores extremos.

En síntesis, la dispersión lineal tiene la doble bondad de exhibir desempeño esperado de  $\mathcal{O}(1)$  con un consumo de memoria proporcional a la cantidad de elementos que se manejan. Debe ser la opción cuando se traten conjuntos generales, por ejemplo, los mapeos de lenguajes como `python` o `perl` o la propuesta al estándar `C++` de mapeo hash, llamada `hash_map`.

## 5.2 Funciones hash

Todos los resultados analíticos de desempeño sobre los esquemas de resolución de colisiones asumen que  $h(k) : \mathcal{K} \longrightarrow [0, M - 1]$  es  $\mathcal{O}(1)$  y que ésta emula a una distribución de probabilidad uniforme. Sin estas premisas, ninguno de aquellos resultados es veraz. Es obviamente esencial entonces que la función hash cumpla estos requisitos. Pero hay mucho más que estos meros requisitos cuando se diseña una función hash.

### 5.2.1 Interfaz a la función hash

Replanteemos  $h(k) : \mathcal{K} \longrightarrow [0, M - 1]$  en términos computacionales bajo el siguiente “tipo-interfaz”:

```
template <typename Key> size_t hash_fct(const Key & key);
```

“Idealmente”, la función toma una clave genérica de tipo `Key` y la transforma a un entero entre 0 y el más grande entero que pueda representarse en el tipo `size_t`. Recordemos que el índice dentro de la tabla se calcula mediante:

```
(*hash_fct)(key) mod M;
```

La “naturaleza” de `Key` no necesariamente es numérica, en cuyo caso debemos encontrar una transformación de `key` hacia el tipo `size_t`. Así que establecemos el siguiente protocolo para el resultado de `(*hash_fct)(key)`:

1. Si `Key` no es numérico, transformarlo entonces al tipo `size_t`.
2. Asumiendo que `key` ya se encuentra en el dominio `size_t` -por vía directa o por la transformación anterior-, “dispersar” `key` lo más que se pueda, en lo posible con “emulación aleatoria”, hacia el dominio `size_t`. Esta dispersión es en sí la transformación.

En *ALÉPH* siempre culminamos la determinación de  $h(k) : \mathcal{K} \longrightarrow [0, M - 1]$  con la llamada `(*hash_fct)(key) mod M`. Así esta operación, como veremos prontamente, también puede ser parte de la dispersión.

### 5.2.2 Holgura de dispersión

El mayor entero posible está supeditado al mayor valor que se pueda representar con el tipo `size_t`. A la cantidad posible de distintos valores que puede arrojar  $h(k) : \mathcal{K} \longrightarrow [0, M - 1]$  se le llama “holgura de dispersión”; cuanto mayor sea la cantidad de números distintos que pueda generar la función hash, mayor será su holgura de dispersión. Así pues, en nuestro contexto, un requerimiento de la función hash es tener la mayor holgura de dispersión posible.

En programación, al fin de cuentas, todo tipo de dato puede representarse como una secuencia de bits. Cualquiera que sea la índole del dominio  $\mathcal{K}$ ,  $\forall k_i, k_j \in \mathcal{K}, k_i \neq k_j \implies k_i$  y  $k_j$  tendrán secuencias de bits distintas. En este sentido es bastante deseable que  $h(k) : \mathcal{K} \longrightarrow [0, M - 1]$  considere todos, los bits de una clave, pues de esta manera se facilita mejor el repartir  $\mathcal{K}$  en el espectro `size_t` y por tanto se aumenta la capacidad de dispersión. Pero no basta con considerar todos los bits. Para comprender esto, examinemos una típica situación del mundo real de programación.

Consideremos como claves simples cadenas de caracteres (`char*`) y a  $h(k)$  como la suma de los caracteres de la cadena.  $h(k)$  encaja en el rango  $[0, \sum_{i=0}^{n-1} s_i]$ . Si `size_t` está representado con 32 bits, el valor máximo a representar con `size_t` es  $2^{32} = 4294967296 \gg \sum_{i=0}^{n-1} s_i$ . Si tenemos claves alfabéticas cuya longitud es de 30 caracteres, entonces, con valores de caracteres entre  $[65, 90] \cup [61, 122]$ <sup>11</sup>, el rango de  $h(k)$  es  $[61 \times 30, 122 \times 30] = [1950, 3660]$ . Esto implica que dejamos de lado  $2^{32} - 3660 + 1950$  valores posibles que podría tomar  $h(k)$  representado con `size_t`; si  $M > 1710$ , entonces, con certitud,  $M - 1710$  entradas jamás serán accedidas y cualquiera sea la técnica que lidie con las colisiones completamente vana. Una tabla hash con esa función puede llamarse una “tabla mash”.

### 5.2.3 Plegado o doblado de clave

Considerar todos los bits de una clave está supeditado al tamaño del tipo `size_t`<sup>12</sup>, el cual está determinado por las características de hardware y el compilador. También, si efectuamos operaciones aritméticas como la exemplificada en la subsección precedente, podemos tener un desborde.

A la técnica tradicional para confrontar este problema se le conoce bajo el participio “plegado” o “doblado”. Plegar una clave consiste en seccionarla en pedazos más pequeños y combinarlos con sumas, oes exclusivos o desplazamientos.

Una clave de  $n$  bytes debe plegarse en `sizeof(size_t)` bytes. Según la índole de la clave podemos seleccionar pedazos de, por ejemplo,  $n/\text{sizeof}(\text{size\_t})$  en un orden escogido para evadir algún particular sesgo. Por ejemplo, si tenemos una clave de 20 bytes, y `sizeof(size_t) == 4`, entonces podemos considerar las niblas más significativas de los 8 bytes centrales:

424 *(Plegado de una cadena key de n bytes 424)≡*

```
char buf[sizeof(size_t)];
char * ptr = (char*) key + 6; // comienzo del centro en 20 bytes
for (int i = 0; i < 4; ++i)
{
 buf[i] = (ptr[0] >> 4) | (ptr[1] >> 4);
 ptr += 2;
}
```

En pos de una buena dispersión, en el diseño de un buen plegado es conveniente indagar la zona de la clave en la cual se presenta más diversidad, pues esa es la zona con mayor potencial para emular la deseada aleatoriedad. Simétricamente, las zonas con menor variedad son más causantes de repitencias.

### 5.2.4 Heurísticas de dispersión

En síntesis, para calcular  $h(k) : \mathcal{K} \longrightarrow [0, M - 1]$  tenemos los siguientes requerimientos:

1. Que sea muy rápida, no sólo  $\mathcal{O}(1)$ , sino que su tiempo constante sea bajo.
2. Que tenga suficiente holgura de dispersión.
3. Que los resultados caractericen una distribución de probabilidad uniforme.

<sup>11</sup>Estos son los valores ASCII para los símbolos latinos no acentuados.

<sup>12</sup>O del tipo que se utilice como rango según sea el caso.

De estos tres requerimientos, el más difícil es el tercero, pues las lecciones aprendidas en la generación de números aleatorios [98] indican que es harto difícil, por no decir imposible, generar números aleatorios a partir de datos que no los son<sup>13</sup>. Nadie hasta el presente ha descubierto una técnica de dispersión general que emule a una distribución uniforme, pero sí se conocen excelentes heurísticas que, operadas sobre una representación numérica, emulan la aleatoriedad.

#### 5.2.4.1 Dispersión por división

El método de dispersión más popular es justamente el que hemos empleado a través de todos los tipos desarrollados:

$$h(k) = k \bmod M \quad (5.44)$$

Pero en este tipo de función es crítica la selección del divisor, o sea, el valor de  $M$ , el cual, a menudo, es el propio tamaño de la tabla<sup>14</sup>.

Sabemos que cualquier entero  $k$  de  $n$  dígitos puede expresarse como

$$k = \sum_{i=0}^{n-1} d_i \times b^i ;$$

donde cada  $d_i$  es un dígito y  $b$  es la base del sistema numérico. En este sentido, la división entera  $\lfloor k/M \rfloor$  puede expresarse como:

$$\left\lfloor \frac{k}{M} \right\rfloor = \left\lfloor \frac{\sum_{i=0}^{n-1} d_i \times b^i}{M} \right\rfloor . \quad (5.45)$$

El resto de la división,  $k \bmod M$ , puede interpretarse como:  $k \bmod M = k - k \lfloor k/M \rfloor$ , lo cual, según (5.45), puede expresarse como:

$$\begin{aligned} k \bmod M &= k - \left\lfloor \frac{\sum_{i=0}^{n-1} d_i \times b^i}{M} \right\rfloor \\ &= \sum_{i=0}^{n-1} d_i \times b^i - \left\lfloor \frac{\sum_{i=0}^{n-1} d_i \times b^i}{M} \right\rfloor . \end{aligned} \quad (5.46)$$

Esta expresión nos permite ver que los sumandos menos significativos, aquellos tales que  $d_i \times b^i < M$ , siempre conforman parte de  $k \bmod M$ , pues éstos no son, en términos literales, "enteramente divisibles" por  $M$ . Así pues, (5.46) puede expresarse recursivamente como:

$$\begin{aligned} k \bmod M &= \sum_{\forall i | d_i \times b^i < M} d_i \times b^i + \left( \sum_{\forall i | d_i \times b^i \geq M} d_i \times b^i \right) \bmod M \\ &= \underbrace{\sum_{\forall i | d_i \times b^i < M} d_i \times b^i}_{\text{Indivisible}} + \underbrace{\left( \sum_{\forall i | d_i \times b^i \geq M} \left\lfloor \frac{d_i \times b^i}{M} \right\rfloor \bmod M \right) \bmod M}_{\text{Resto de restos}} , \end{aligned} \quad (5.47)$$

<sup>13</sup>En programación, decir "imposibilidad" es muy aventurado, pues lo virtual da espacio para mucho posible.

<sup>14</sup>En lo que sigue debemos estar pendientes del hecho de que  $M$  no necesariamente es el tamaño de la tabla.

o sea, todos los sumandos de  $k$  que no son enteramente divisibles por  $M$  más el resto de la suma de los restos<sup>15</sup>.

Al primer sumando de (5.47) lo llamaremos “lo indivisible” y al segundo “el resto de restos”.

Esta interpretación del resto nos facilita aprehender la trascendencia que tiene la adecuada selección del divisor  $M$  sobre la consideración de dígitos de la secuencia que expresa a  $k$ . De (5.46) se puede ver que los únicos dígitos que se contabilizarán con seguridad son los del componente indivisible. El resto de restos depende de la multiplicidad respecto a  $M$ . Según sea el valor de  $M$ , la operación módulo tomará en cuenta o no algunos dígitos de  $k$ . Un ejemplo puede ayudar a aquellos que aún no lo han realizado. Si tenemos  $M = 100$  y base decimal, entonces el resto sólo está determinado por lo indivisible, pues los restos restantes son nulos y con ellos el resto de restos. Con  $M = b^w$  no importa cuál sea el valor de  $k$ , el resto sólo considera los  $w$  últimos dígitos de  $k$ . Si, por ejemplo, los últimos dígitos de  $k$  están sesgados a estar entre 20 y 30 y  $M = 100$ , entonces  $h(k)$  estará sesgado a ese rango, lo cual derrumba todas las expectativas de desempeño, pues  $h(k)$  distará mucho de ser aleatoria.

Tampoco basta  $M \neq 2^w$ , pues podría existir algún patrón de multiplicidad en los sumandos  $d_i \times b^i$ . Por ejemplo,  $75312 \bmod 150 = 12 = 75312 \bmod 100$ ; cuando miramos que  $150 = 10 \times 5 \times 3$ , no sólo vemos una multiplicidad como para cuando  $M = 100$ , sino que, como cada sumando del resto de restos es múltiplo de 5 la incidencia de un dígito es su multiplicidad por 3; en términos más simples, cada dígito tiene probabilidad  $1/3$  de no ser considerado. Por añadidura, la suma de restos también tiene buena probabilidad de ser múltiplo de tres.

Las consideraciones sobre la multiplicidad que hemos llevado a cabo son válidas para cualquier base numérica, en particular para la binaria.

Cuando seleccionamos el método de división como función hash, debemos estar sumamente pendientes de esta multiplicidad, pues el azar puede jugarnos en contra. Si no disponemos de tiempo o ánimo para estudiar la multiplicidad, entonces podemos seleccionar a  $M$  como un número primo, lo cual asegura que ninguno de los sumandos de  $k$  tenga alguna relación de multiplicidad.

El método de división aunado a una selección de  $M$ , como divisor y a la vez como tamaño de la tabla, es el método de dispersión más popular. El enfoque de *ALEPH* conlleva directamente este método, aunque no la selección de  $M$  como primo, cual se delega al usuario del TAD. En sí, este método es bastante sencillo y rápido, pero no se adecúa a todas las circunstancias; por ejemplo, si lo usamos directamente, no siempre es fácil en función de tamaño original o del reajuste asegurar que  $M$  sea primo.

#### 5.2.4.2 Dispersión por multiplicación

Este método tiene la ventaja de que puede adaptarse para cualquier tamaño de tabla.

**Definición 5.2 (Fraccional de un número real)** Sea  $x$  un número real positivo, el

<sup>15</sup>Es de notar que

$$k \bmod M = \left( \sum_{i=0}^{n-1} (d_i \times b^i) \bmod M \right) \bmod M ,$$

o sea, el resto se puede definir simplemente como el resto de los restos. Desarrollamos la representación según  $k \bmod M = k - \lfloor k/M \rfloor$  porque consideramos que permite aprehender mejor.

fraccional de  $x$ , denotado como  $\{x\}$ , es la parte fraccional de  $x$ . Por ejemplo,  $\{1,6252345\} = 0,6252345$ .

El método de multiplicación define una función hash del siguiente modo:

$$h(k) = \lfloor \{k\theta\} M \rfloor , \quad (5.48)$$

donde  $\theta$  es un factor de multiplicación.

¿Cuál debe ser el valor del factor  $\theta$ ? Si deseamos emular la aleatoriedad, entonces tiene sentido decir que para favorecerla la escogencia debe ser algo irracional. Pues bien, resulta que la matemática nos descubre una clase infinita de números llamados “irracionales” en el sentido de que no pueden expresarse como una razón (fracción) y, ya hechas las consideraciones sobre la multiplicidad de  $M$  en la subsección anterior, sabemos que las razones (fracciones) tienden a sesgar las claves.

**Proposición 5.5 (Turán-1958)**<sup>16</sup> Sea  $\theta$  un número irracional y  $n$  un entero positivo. Entonces, los  $n + 1$  fraccionales  $\{\theta\}, \{2\theta\}, \{3\theta\}, \dots, \{n\theta\}$  tienen a lo sumo tres longitudes distintas y el próximo fraccional  $\{(n + 1)\theta\}$  se aleja en una de las mayores longitudes.

**Demostración** Ver Knuth [99] ■

El conocimiento que nos aporta este teorema se traduce en las siguientes observaciones:

1.  $\{k\theta\}$  y  $\{(k + 1)\theta\}$  pueden estar separados por tres distancias posibles y,
2.  $\{k\theta\}$  y  $\{(k + 1)\theta\}$  distan en entre sí entre las dos mayores longitudes. Esta separación es lo que emula la aleatoriedad.

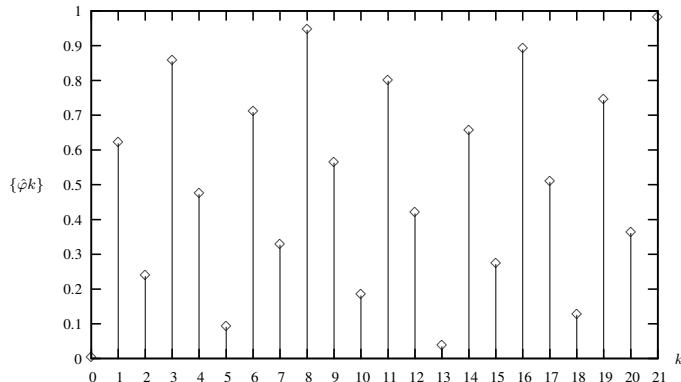


Figura 5.9: Valores del fraccional  $\{k\phi\}$

Knuth [99] sugiere que

$$\theta = \hat{\phi} = \varphi^{-1} = \frac{\sqrt{5} - 1}{2}$$

es una buena selección en muchos casos y denomina a esta dispersión “de Fibonacci”. Pero, ¿por qué es bueno este número?  $\hat{\phi}$  es el recíproco de lo que se conoce como la “proporción áurea” o el “radio divino”, pues desde tiempos inmemoriales se dice que es la

<sup>16</sup>Citado por Knuth [99].

proporción con que los dioses forjaron al mundo<sup>17</sup>. La proporción  $\varphi$ , que, como vemos, es irracional, lo que indicaría, quizá, asumiendo un Dios racional, que hay algo de irracional en lo divino, no sólo es ubicua en la naturaleza, sino que ha comprobado ser bastante buena en una amplia gama de dominios: música, arquitectura, ingenierías, etcétera. Vale la pena constatar la distribución multiplicativa, con unos pocos valores de  $k$ , a través de observación de la gráfica 5.9, de la cual constatamos que claves secuenciales distan en la dispersión: por ejemplo,  $h(5) = 0$  y  $h(6) = 4$ , pero, además, la apariencia de la propia gráfica bien podría ser la de una distribución uniforme.

Ya vemos pues que la dispersión por multiplicación reparte los productos con suficiente distancia, los intercala sin algún orden aparente y es independiente del valor de  $M$ . Por añadidura, este método tiene la ventaja de tender a ser más rápido que el de división, pues es más rápido multiplicar que dividir. Hay además un “truco” para multiplicar rápidamente, que nos ahorra el coste de apelar a la aritmética en punto flotante.

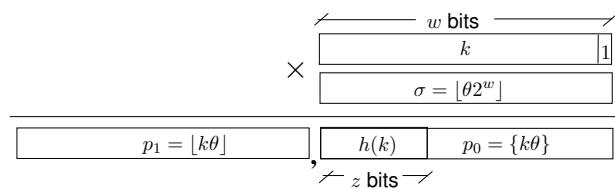


Figura 5.10: Esquema general de hash por multiplicación rápida

Sea  $w$  la longitud de la clave en bits y escojamos  $M = 2^z$ . Puesto que  $M = 2^z$ ,  $h(k)$  puede representarse con  $z$  bits.

Sea  $\sigma = \lfloor k\theta \rfloor$  representado en  $w$  bits. El producto entero  $k\sigma$  -no en punto flotante- tiene una longitud de  $2w$  bits dividida en dos palabras de  $w$  bits  $p_0$  y  $p_1$  tales que  $k\sigma = p_1 \lfloor k\theta \rfloor + p_0 \{k\theta\}$ . Pictóricamente, podemos mirar el asunto en la figura 5.10.

De este modo, los  $z$  bits más significativos de  $p_0$  representan el valor de  $h(k)$ ;  $h(k)$  se obtiene con desplazar  $z$  bits hacia la izquierda a  $p_0$  o con un “and” aritmético entre  $p_0$  y la palabra con forma:  $\underbrace{11\dots1}_{z \text{ veces}} \underbrace{00\dots0}_{w-z \text{ veces}}$ . Por grandiosa añadidura, si nos aseguramos de que  $\sigma$  sea impar, entonces este método garantiza que la palabra  $p_0$  se distribuya uniformemente, tal como evidencia el siguiente teorema.

**Proposición 5.6 (Aleatoriedad de  $\{k\theta\}$  [108])** Sea  $\sigma = \lfloor \theta 2^z \rfloor$  tal que  $\sigma$  es impar. Sea  $\mathcal{K}$  el conjunto de claves, el cual está representado con palabras de  $w$  bits. Sea  $k\sigma = p_1 \lfloor k\theta \rfloor + p_0 \{k\theta\} \mid k \in \mathcal{K}$ . Entonces, para cualquier par de claves distintas  $k_1, k_2 \in \mathcal{K}$ :

$$\sigma k_1 \bmod 2^w \neq \sigma k_2 \bmod 2^w \quad (5.49)$$

Dicho de otro modo, según la figura anterior:

$$\underbrace{\lfloor k_1 \theta \rfloor}_{p_1}, \underbrace{\{k_1 \theta\}}_{p_0} \neq \underbrace{\lfloor k_2 \theta \rfloor}_{p_1}, \underbrace{\{k_2 \theta\}}_{p_0}. \quad (5.50)$$

La proposición es equivalente a enunciar que para las palabras menos significativas  $\{k_1 \theta\} \neq \{k_2 \theta\}$ .

<sup>17</sup>En § 6.4.2.2 (Pág. 488) se explica un poco más el cálculo de este número y su relación con los números de Fibonacci. En § 6.8 (Pág. 523) se hacen algunas referencias históricas del uso de esta proporción.

**Demostración** Asumamos  $k_1 < k_2$  y sea  $D_k = k_2 - k_1$ . El argumento de la prueba estriba en demostrar que  $(\sigma k_2 \bmod 2^w) - (\sigma k_1 \bmod 2^w) = \sigma(k_2 - k_1) \bmod 2^w = \sigma D_k \bmod 2^w \neq 0$ , lo cual tiene sentido porque  $D_k \neq 0$  y el módulo exhibe la propiedad distributiva.

Sean  $\sigma = \sum_{i=0}^{w-1} \sigma_i 2^i$  y  $D_k = \sum_{i=0}^{w-1} d_i 2^i$ . El producto  $\sigma D_k$  puede expresarse como:

$$\begin{aligned}\sigma D_k &= \left( \sum_{i=0}^{w-1} \sigma_i 2^i \right) \times \left( \sum_{i=0}^{w-1} d_i 2^i \right) \\ &= (\sigma_0 2^0 + \sigma_1 2^1 + \cdots + \sigma_{w-1} 2^{w-1}) \times (d_0 2^0 + d_1 2^1 + \cdots + d_{w-1} 2^{w-1}) \\ &= \sum_{i=0}^{2w-2} \left( \sum_{j=0}^i \sigma_j d_{i-j} \right) 2^i\end{aligned}\tag{5.51}$$

Sea  $d_p$  el primer bit menos significativo de  $D_k$  cuyo valor es 1; de este modo,  $d_{p-1} = d_{p-2} = \cdots = d_0 = 0$ . Debe sernos claro que la sumatoria interna de (5.51) es nula para  $i < p$ . Ahora bien, para  $i = p$

$$\sum_{j=0}^p \sigma_j d_{p-j} = \sigma_0 d_p ,$$

pues los sumandos restantes son nulos dado que  $d_{p-j} = 0$  para  $j < p$ . De este modo, el bit  $p$  de  $\sigma D_k$  es 1, lo cual implica que  $D_k \neq 0$  y  $\sigma k_1 \bmod 2^w \neq \sigma k_2 \bmod 2^w$  para todo  $k_1 \neq k_2$

■

El teorema anterior implica que todas las claves posibles de  $w$  bits se distribuyen uniformemente en la palabra de  $p_0 = \{k\theta\}$ . Por supuesto, habrá colisiones entre los  $z$  bits más significativos que conforman a  $h(k)$ .

Esta técnica se adecua muy bien para una tabla hash lineal en la cual el tamaño  $M$  se duplica. Además, no es difícil adaptarla a palabras de bits de longitud arbitraria.

### 5.2.5 Dispersión de cadenas de caracteres

Hay una vasta cantidad de situaciones en que la indización se efectúa por una cadena de caracteres. Es pues importante meditar sobre la dispersión de cadenas, aparte de que, como lo exemplificamos en § 5.2.2 (Pág. 423), es relativamente fácil construir una pésima función hash. Si no se dispone del tiempo y ánimo, entonces es conveniente transar por bien conocidas y eficientes funciones hash. Las siguientes son públicas y de buen desempeño:

1.

429     ⟨Hash de Bernstein [20] 429⟩≡

```
size_t berstein_hash(unsigned char * str)
{
 size_t hash = 5381;
 int c;
 while (*c = *str++)
 hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
 return hash;
}
```

Las razones sobre la escogencia del valor inicial 5381 y el número “mágico” 33 aún no han sido descubiertas.

2.

430     ⟨Hash del sistema sdbm [137] 430⟩≡  
           size\_t sdbm\_hash(unsigned char \* str)  
           {  
               size\_t hash = 0;  
               int c;  
               while (c = \*str++)  
                   hash = c + (hash << 6) + (hash << 16) - hash;  
               return hash;  
           }

### 5.2.6 Dispersión universal

Por muy buena que sea una función de dispersión, ésta puede deparar en una serie de colisiones y, consecuentemente, degradar seriamente el desempeño de una tabla hash. La mala suerte, improbable en el azar pero posible, no necesariamente podría ser la causante de una serie de colisiones. Supongamos, por ejemplo, que se tiene conocimiento exacto de la función hash que se emplea en alguna determinada aplicación. Podría ocurrir que ese conocimiento se utilizase para generar inserciones colindantes que degraden el desempeño.

Curiosamente, tal como lo estudiamos con el quicksort (§ 3.2.2.4 (Pág. 179)), una manera de evitar la mala suerte, así como también, en este caso, la malicia, es mediante aleatorización, lo que quizás equivalga a decir que la mala suerte se combate con el azar<sup>18</sup>.

Una simple manera de aleatorizar es usar dos funciones de dispersión  $h_1(k)$  y  $h_2(k)$ . Cuando ocurra una inserción se selecciona al azar entre  $h_1$  y  $h_2$  la función hash que se empleará. Cuando se requiera buscar, se comienza con cualquiera de las funciones. Si la clave se encuentra, entonces aquella es la función con que pudo haberse insertado<sup>19</sup>; de lo contrario se prueba con la otra función.

Obviamente, esta técnica enlentece los algoritmos, pero combate algún mal sesgo en la secuencia de inserción de claves.

En el mismo espíritu de aleatorización, en lugar de usar y escoger simultáneamente dos o más funciones de dispersión, podríamos seleccionar una al azar cada vez que la tabla devenga vacía. Esta táctica no degrada en lo absoluto el desempeño y rompe completamente algún sesgo desafortunado o malicioso.

La siguiente definición establece un requerimiento que hace a un conjunto de funciones de dispersión “aceptables”.

#### Definición 5.3 (Familia universal de funciones hash)

Sea  $\mathcal{H} = \{h_0(k), h_1(k), \dots, h_{W-1}(k)\}$  una colección de funciones de dispersión desde un dominio  $\mathcal{K}$  al rango  $[0, M]$ . Se dice que  $\mathcal{H}$  es universal si y sólo si:

$$\forall k_1, k_2 \in \mathcal{K} \implies \frac{|\{\forall h_i, h_j \in \mathcal{H} \mid h_i(k_1) = h_j(k_2)\}|}{|\mathcal{H}|} \leq \frac{1}{M}$$

<sup>18</sup>“Azar” proviene del árabe “zahr” que significa “dado”, pero también “flores”.

<sup>19</sup>Notemos que puede ocurrir  $h_1(k) = h_2(k)$ , razón por la cual no se puede saber cuál fue la función usada para insertar.

Dicho de otro modo, el máximo permitido de colisiones que pueden ocurrir con dos pares de claves es a lo sumo  $\mathcal{H}/M$ .

Ha sido demostrado que para una familia universal de cardinalidad  $|\mathcal{H}|$ , el desempeño es completamente equiparable en todas las estrategias de resolución que hemos estudiado<sup>20</sup>. Igualmente se ha demostrado<sup>21</sup> que para  $\mathcal{K} = \{0, 1, \dots, M - 1\} \mid |\mathcal{K}| = L$  y

$$h_{i,j}(k) = ((ik + b) \bmod L) \bmod M , \quad (5.52)$$

entonces:

$$\mathcal{H} = \{h_{i,j}(k) \mid 1 \leq i \leq L \wedge 1 \leq j \leq L\} \quad (5.53)$$

Tenemos pues un método para disponer de  $L$  funciones hash y hacer aleatorización.

### 5.2.7 Dispersión perfecta

Hay ocasiones en que el conjunto de claves a indexar es finito y conocido. En este caso, es posible diseñar una función hash biyectiva y completa, con eficiencia de cómputo  $\mathcal{O}(1)$ , que mapee todo el conjunto de claves hacia el rango  $[0, M)$ . El tiempo de acceso está garantizado a ser  $\mathcal{O}(1)$  y no se requiere manejar colisiones (la función hash es biyectiva).

A una función hash como la descrita se le llama “perfecta” y existen variadas técnicas y programas consumados para generar la función en cuestión.

Si bien las situaciones en que se conoce todo el conjunto de claves son ocasionales, tampoco son excepcionales. Consideremos por ejemplo un diccionario grande a ser grabado estáticamente en un DVD. En este caso vale perfectamente la pena cargar en memoria una función hash perfecta que mapee la palabra hacia la ubicación física en el DVD.

En GNU existe un excelente y general programa, llamado gperf [5], para generar funciones hash perfectas de cualquier tipo de clave.

## 5.3 Otros usos de las tablas hash y de la dispersión

### 5.3.1 Identificación de cadenas

Según la bondad de una función de dispersión, ésta puede utilizarse, con probabilidad de atino en función de calidad de la función hash, para reconocer una subsecuencia dentro de otra. La idea fundamental es que para dos secuencias  $s_1 \neq s_2$ , la probabilidad de que  $h(s_1) = h(s_2)$  es bastante baja si la función hash es buena.

Este es el principio de algoritmos de búsqueda de patrones inspirados en un célebre algoritmo llamado de “Rabin-Karp” [91] en honor a las primeras personas que lo descubrieron. Fue originalmente concebido para la búsqueda de subcadenas de caracteres. La idea es calcular previamente una “huella dactilar” (fingerprint) en el siguiente contexto algorítmico C/C++:

431 *(Algoritmo candido de Rabin-Karp 431)≡*

```
int rabin_karp_search(char * str, char * sub)
{
 // guardar longitudes de las cadenas involucradas
 const size_t m = strlen(sub);
```

<sup>20</sup>Puede consultarse Cormen, Leiserson y Rivest [32] o Knuth [99].

<sup>21</sup>Consúltese Lewis-Denenberg [108] para detalles.

```

const size_t n = strlen(str);
⟨Calcular en fp_sub y fp_curr huellas dactilares sub y de str[0] 433a⟩
⟨Multiplicando fact de huella dactilar 433c⟩
 // recorrer la cadena str en búsqueda de la subcadena sub
for (int i = 0; i < n - m + 1; ++i)
{
 if (fp_sub == fp_curr) // coinciden las huellas digitales?
 { // sí ==> verificar si &str[i] corresponde con subcadena
 for (char * aux = str[i], int k = 0; k < m; ++k)
 if (aux[k] != sub[k])
 break; // falso positivo
 return i; // str[i] contiene la cadena
 }
 ⟨Calcular huella dactilar fp_curr para str[i+1] 433b⟩
}
return -1; // subcadena no se encuentra
}

```

Como vemos, el algoritmo es estructuralmente simple y muy similar a la búsqueda en bruto de la subcadena, la cual es  $\mathcal{O}(n \times m)$ . De esto se desprende observar que para que el algoritmo de Rabin-Karp valga la pena respecto al de fuerza bruta, se deban cumplir dos cosas:

1. El cálculo del bloque ⟨Calcular huella dactilar fp\_curr para str[i+1] 433b⟩ debe ser rápido,  $\mathcal{O}(1)$ , preferiblemente.
2. La cantidad de “falsos positivos” debe ser pequeña, pues su ocurrencia acarrea comparar los  $m$  caracteres de la subcadena. Consecuentemente, en la medida en que se detectan más falsos positivo, el algoritmo de Rabin-Karp tenderá hacia  $\mathcal{O}(n \times m)$ ; simétricamente, menos falsos positivos harán al algoritmo  $\mathcal{O}(n)$ .

El cálculo de la huella dactilar no es otra cosa que una función hash definida, sobre una cadena de caracteres genérica str de la siguiente manera:

$$h(str) = \left( \sum_{i=0}^{|str|-1} str_i \times B^{|str|-1-i} \right) \bmod M ; \quad (5.54)$$

donde  $B$  es la base del sistema de codificación (radix) de un símbolo perteneciente a una cadena de caracteres y  $str_i$  es el  $i$ -ésimo símbolo de la secuencia str.

Es menester recordar de § 5.2.4.1 (Pág. 425) la propiedad distributiva que nos indica que el resto de la suma es igual al resto de la suma de los restos<sup>22</sup>, o sea

$$(i + k) \bmod M = ((i \bmod M) + (k \bmod M)) \bmod M . \quad (5.55)$$

La propiedad distributiva del módulo también se aplica al producto; es decir

$$(ik) \bmod M = ((i \bmod M)(k \bmod M)) \bmod M . \quad (5.56)$$

<sup>22</sup>Véase también (5.47) y la nota a pie de la página 425.

Aprehender el carácter distributivo del módulo nos es muy útil porque nos permite calcular (5.54) en función de los módulos:

$$\begin{aligned}
 h(str) &= \left( \sum_{i=0}^{|str|-1} ((str_i \times B^{|str|-1-i}) \bmod M) \right) \bmod M \\
 &= \left( \sum_{i=0}^{|str|-1} (((str_i \bmod M)(B^{|str|-1-i} \bmod M)) \bmod M) \right) \bmod M \\
 &= \left( \sum_{i=0}^{|str|-1} (((str_i \bmod M)((B^{|str|-1-i} \bmod M) \bmod M)) \bmod M) \right) \bmod M \quad (5.57)
 \end{aligned}$$

lo cual es sumamente grandioso, pues nos evita preocuparnos por el eventual desborde numérico de las sumas y potencias.

Rabin y Karp [91] demuestran cómo  $h(str)$  dispersa muy bien conforme el valor de  $M$  se selecciona como un número primo.

Con el conocimiento de (5.57), podemos codificar el bloque inicial:

433a *(Calcular en fp\_sub y fp\_curr huellas dactilares sub y de str[0] 433a)≡* (431)  

```

const size_t radix = 256; // radix de un símbolo
size_t fp_sub = 0;
size_t fp_curr = 0;
for (int i = 0; i < m; ++i)
{
 fp_sub = (radix*fp_sub + sub[i]) % M;
 fp_sub = (radix*fp_curr + str[i]) % M;
}

```

El bloque toma  $\mathcal{O}(m)$ . La huella dactilar  $fp\_sub$  sólo se calcula una vez, pero la de la subcadena de  $str$  en la posición  $i$  debe calcularse  $n - i$  veces. Si usáramos el mismo procedimiento que para el bloque *(Calcular en fp\_sub y fp\_curr huellas dactilares sub y de str[0] 433a)*, entonces el algoritmo de Rabin-Karp sería  $\mathcal{O}(n \times m)$  con un coste constante mayor que el algoritmo a fuerza bruta. El truco del algoritmo reside en que la huella digital para la subcadena  $str_{i+1}$  de longitud  $m$ , que resumiremos como  $h(str_{i+1})$ , puede calcularse en tiempo independiente de  $m$  en función del valor previo de  $h(str_i)$ ; todo lo que debemos hacer es restarle a  $h(str_{i+1})$  el primer sumando de  $h(str_i)$ , pues éste ya no es parte de la huella digital, y añadirle el nuevo sumando para  $str_{i+1}$ . De esto se deduce computar  $h(str_{i+1})$  según:

$$h(str_{i+1}) = (B h(str_i) + str_{i+m} - B^m str_i) \bmod M ; \quad (5.58)$$

esta técnica es llamada “dispersión enrollada” (“hash rolled”). Bajo estos términos, el valor de  $fp\_curr$  para la  $i$ -ésima iteración se calcula así:

433b *(Calcular huella dactilar fp\_curr para str[i+1] 433b)≡* (431)  

```

fp_curr = (radix*(fp_curr - str[i]*fact) + str[i + m]) % M;

```

433c *(Multiplicando fact de huella dactilar 433c)≡* (431)  

```

const size_t fact = (size_t) pow(radix, m - 1); // radix^(m-1)

```

El método de “huella dactilar” se aplica a una amplia cantidad de situaciones. En primer lugar, el algoritmo puede hacerse más general para reconocer patrones bi o tridimensionales si los objetos en el plano o espacio se consideran como matrices o arreglos tridimensionales, los cuales, al fin de cuentas, deparan en secuencias.

Una ganancia de este algoritmo es que la longitud de la subsecuencia a buscar no incide durante el barrido de la secuencia en la que se busca; esto contrasta con otros algoritmos que se enlentecen a medida que la secuencia a buscar es de mayor longitud.

El valor de  $M$  puede seleccionarse aleatoriamente, de manera tal que un hipotético adversario no pueda atacar el algoritmo colocando como una entrada una secuencia a buscar que cause muchas colisiones.

Compartir libremente el conocimiento es una condición esencial para ser humano. Pero adjudicarse el mérito de otro, no sólo puede acarrear explotación del prójimo, sino que va contra la esencia de la vida en comunidad. El adjudicarse falsamente un descubrimiento sin haberlo hecho se le denomina “plagio”. En el descubrimiento de ideas, así como en cualquier otro obrar, se puede plagiar. Para lo escrito, y en general para toda producción representable como una secuencia, puede modificarse el algoritmo de Rabin-Karp para verificar automáticamente si eventualmente se ha cometido algún plagio. En este caso se diseña una función hash que modelice la secuencia sospechosa de plagio.

### 5.3.2 Supertraza

La computación está llena de problemas que manejan conjuntos de datos muy grandes; muchos de ellos “intratables” en el sentido de que la cantidad de datos es exponencial. Hay ocasiones en las cuales un dato o resultado debe guardarse a efectos de no repetir su cálculo y, sobre todo, de no repetir los datos que suceden al recién descubierto.

Consideremos una secuencia genérica de datos  $\langle d_1 \rightarrow d_2 \rightarrow d_3 \rightarrow \dots \rightarrow d_n \rangle$  tal que  $d_2$  depende del cálculo de  $d_1$  y así secuencial y genéricamente para todo dato  $d_{i+1}$ , que requiere calcular  $d_i$ . La idea de guardar cada dato  $d_i$  en una tabla hash es que, según la índole del problema, puede ocurrir que el cálculo de un dato  $d_i$  aparezca repetido; si éste es el caso, entonces el cálculo de toda la secuencia subsiguiente a  $d_i$  ( $d_{i+1} \rightarrow d_{i+2} \rightarrow \dots \rightarrow d_{i+k} \rangle$ ) se repite de nuevo.

Tomemos como ejemplo un algoritmo que a fuerza bruta descubra maneras de ubicar  $n$  reinas en un tablero de ajedrez sin que éstas se amenacen. Es decir, comenzamos por poner una reina en el escaque  $a1$  y proseguimos recursivamente a poner la siguiente en la secuencia de escaques de  $b$  sin que ésta amenace a  $a1$ . Comenzando por  $b1$ , vemos que la reina amenaza a la puesta en  $a1$ ; lo mismo ocurre si la ponemos en  $b2$ . Estas dos combinaciones, que definitivamente no conducen a la solución, es conveniente guardarlas a efectos de que en un tablero de  $n \times n$  escaques no repitamos el cálculo en vano. Para ello, la secuencia explorada se guarda en una tabla.

Aunque el problema anterior puede resolverse sin necesidad de una tabla, éste ilustra dos aspectos de interés: (1) la idea de guardar estado de cálculo para evitar repetición, y (2) el hecho de que conforme aumenta la dimensión del tablero y la cantidad de reinas aumenta la cantidad de estados.

La cantidad de posiciones de amenaza que debemos guardar para no repetir cálculos aumenta exponencialmente según la dimensión  $n$ . En términos prácticos, no podemos mantener una tabla que guarde todas las posiciones de amenaza, pero sí podemos “marcar” algunas posiciones mediante una función hash y anotarlas en una tabla de bits. Jamás sabremos si una posición particular contiene colisiones, pero la idea de no repetir los cálculos se mantiene.

Con esta técnica no es necesario guardar la posición de amenaza en sí. Es la función

hash y el marcaje en 1 en la tabla lo que expresa el guardar la posición.

El problema de esta técnica es que dejaremos de calcular posiciones de colisión, pero esto se puede atacar aleatoriamente repitiendo el algoritmo con distintas funciones hash. La idea es cubrir probabilísticamente, mediante diversas funciones hash, todas las posiciones posibles.

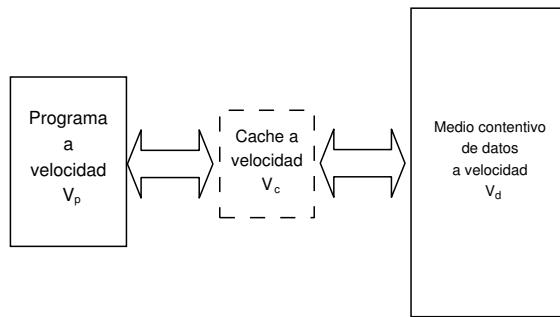
Esta muy interesante técnica, llamada “supertraza” [77], se descubrió en el dominio de la exploración de grafos dinámicos de estado. En lugar de guardar enteramente el estado de cálculo, se diseña una función de dispersión y se marca en una tabla de bits su aparición. Este marcaje aumenta considerablemente la capacidad de almacenamiento, pues sólo se requiere un bit por estado, lo que aumenta la capacidad de tratar con la escala del problema. Sin embargo, no hay manera determinista de saber si un hash almacenado en la tabla corresponde al estado actual de cálculo o se trata de otro estado que causa colisión. En otras palabras, es imposible garantizar que todos los estados se cubran.

Imposibilidad en garantizar no excluye a su probabilidad. De allí que la incertidumbre anterior se ataque ejecutando la técnica con distintas funciones hash de manera de ofrecer una buena expectativa de cubrir todos los estados posibles.

### 5.3.3 Cache (el TAD Hash\_Cache )

Uno de los principales usos de las tablas de dispersión es la instrumentación de caches.

Un cache es una tabla asociativa, finita pero de acceso rápido, que se coloca entre un programa y un conjunto de datos. Para que se recompense la utilización de un cache, éste debe ser mucho más rápido que el simple acceso al conjunto de datos. Pictóricamente, el asunto puede interpretarse del siguiente modo:



En una situación algorítmica, el programa se ejecuta a una velocidad  $v_p$  y accede datos de un conjunto que opera a velocidad  $v_d$ . El cache opera a velocidad  $v_c$  ( $v_p < v_c < v_d$ ). De este modo, cuando se desea buscar un dato se revisa primero el cache y, si se encuentra, entonces se recupera el dato sin necesidad de pagar el coste que acarrea accederlo a velocidad  $v_d$ . La idea se sustenta en la regla 80-20 mencionada en § 3.5.1 (Pág. 212).

La palabra “cache” proviene del verbo galo “cacher” que significa esconder y hace algo de alusión porque a veces éste se maneja transparentemente, es decir, el programa accede a los datos asumiendo que se encuentran en un medio esperado sin conocimiento acerca de la existencia del cache; esta es la situación, por ejemplo, en hardware. Los franceses también le llaman “antememoria” (“*antememoire*”), sobre todo para el hardware.

Hay una vasta cantidad de circunstancias en que el uso de un cache hace la diferencia entre un muy alto y pobre desempeño. Lampson [103] hace una excelente exposición al respecto desde la perspectiva del diseño de sistemas. Las situaciones típicas se pueden resumir como sigue.

### 5.3.3.1 Medio de almacenamiento de datos distinto a memoria principal

Los niveles de almacenamiento, en orden creciente a la velocidad y decreciente a la capacidad, pueden clasificarse como:

1. Memoria principal.
2. Memoria secundaria (discos duros).
3. Memoria terciaria (CD-ROM, DVD, memorias flash, cintas).
4. Memoria remota (disponible por red).

Cuando los datos que acceda un programa se encuentren en un nivel de memoria más lento que la memoria principal, entonces un cache entre la memoria principal y el medio de almacenamiento puede acelerar sustancialmente la ejecución.

### 5.3.3.2 Estructuras de datos estáticas

Por razones de diversa índole puede ocurrir que haya que emplear una estructura de datos que no es adecuada a cierto estilo de recuperación.

Por ejemplo, supongamos que mantenemos en un árbol binario de búsqueda un conjunto de registros. El árbol se indiza por alguna clave principal numérica, por instancia, un número de cédula, lo cual permite una recuperación esperada de  $\mathcal{O}(\lg(n))$  para cualquier registro.

Con menos frecuencia se requiere recuperar por alguna clave alfabética secundaria, un apellido, como clásico arquetipo. En este caso puede ser muy costoso, y probablemente innecesario, indizar mediante un segundo árbol binario. En su lugar podemos guardar en un cache, indizado por apellido, los registros más recientemente consultados, sean por cédula o por apellido, de modo tal que eventualmente nos ahorremos una búsqueda secuencial sobre el conjunto de claves.

### 5.3.3.3 Cálculos que puedan repetirse

En muchas ocasiones se requieren hacer cálculos complejos susceptibles de repetirse. En este caso podemos guardar en un cache la secuencia originaria del cálculo y su resultado. De este modo, si la dinámica del programa requiere de nuevo el cálculo, éste ya se encuentra disponible.

En esta situación suele usarse parte de la secuencia originaria de cálculo como clave de una función de dispersión.

### 5.3.3.4 Implementación

Una característica muy importante de un cache es que es finito, es decir, tiene una capacidad máxima, considerablemente menor que la del conjunto objeto de la gestión. Cuando se requiere insertar un dato en un cache lleno, entonces hay que seleccionar uno de sus datos contenidos y substituirlo por el nuevo. ¿Cuál debe seleccionarse? Si escogemos un dato que será referenciado en un futuro cercano, entonces, en lo que concierne al dato sacado, perdemos la bondad del cache. En este sentido, la localidad de referencia sugiere seleccionar el dato más antiguo.

Probablemente la tabla hash es la estructura de datos idónea para implantar un cache en memoria principal. Esencialmente por una razón: es en promedio el esquema de recuperación más rápido que se conoce. Un cache es una ayuda al desempeño, pero su bondad depende del patrón de acceso. No es, pues, crítico el tener un desempeño garantizado, pues si éste fuese el caso, entonces no deberíamos de usar un cache. Por añadidura, la función hash puede representar parte de lo que se pretenda guardar. Como ejemplos tenemos los usos descritos en las dos subsecciones anteriores y el dispersar cálculos que puedan repetirse.

*ALÉPH* contiene un TAD, llamado Hash\_Cache cuya especificación está contenida en el archivo *<tpl\_hash\_cache.H 437a>*:

437a *<tpl\_hash\_cache.H 437a>*≡  
 template <typename Key, typename Data, class Cmp = Aleph::equal\_to<Key> >  
 class Hash\_Cache  
 {  
*⟨Entrada de Hash\_Cache 437c⟩*  
*⟨Miembros privados de Hash\_Cache 437b⟩*  
*⟨Miembros públicos de Hash\_Cache 440b⟩*  
 };

Hash\_Cache implementa un cache indizado por claves de tipo Key, que guarda datos de tipo Data y está implantado mediante una tabla hash.

Puesto que un cache está destinado a ser finito y lo más rápido posible, el uso de una tabla hash lineal no parece aconsejable. Hecha esta acotación podemos decir que cualquier enfoque de resolución de colisiones (y sus tipos relacionados) puede usarse como trasfondo de implantación. Por su simplicidad, usaremos el tipo LhashTable<Key>:

437b *<Miembros privados de Hash\_Cache 437b>*≡ (437a) 438a▷  
 LhashTable<Key, Cmp> hash\_table;

Esto implica definir una clase de cubeta, la cual se define del siguiente modo:

437c *<Entrada de Hash\_Cache 437c>*≡ (437a)  
 class Cache\_Entry : public LhashTable<Key, Cmp>::Bucket  
 {  
*Data data;*  
*⟨Miembros privados de entrada de Hash\_Cache 437e⟩*  
*⟨Miembros públicos de entrada de Hash\_Cache 437d⟩*  
 }; // fin class Cache\_Entry

Defines:

*Cache\_Entry*, used in chunks 438a, 440, 441, and 443–45.

El observador de la clave se hereda de LhashTable<Key, Cmp>::Bucket, y el del dato se define como:

437d *<Miembros públicos de entrada de Hash\_Cache 437d>*≡ (437c)  
 Data & get\_data() { return data; }

Para determinar cuál de los elementos contenidos en el cache es el más antiguo, las cubetas de la tabla hash se ordenan en una cola desde la entrada más antiguamente accedida hasta la menos recientemente usada. En lugar de usar una cola separada definimos el siguiente atributo interno a la entrada del cache:

437e *<Miembros privados de entrada de Hash\_Cache 437e>*≡ (437c) 438b▷  
 Dlink dlink\_lru; // enlace a la cola lru

Defines:

*dlink\_lru*, used in chunk 438.

Si tenemos un puntero al campo dlink\_lru, entonces el siguiente macro de Dlink (§ 2.4.7 (Pág. 80)) genera una función de conversión:

438a `<Miembros privados de Hash_Cache 437b>+≡` (437a) ◁ 437b 438c ▷  
`LINKNAME_TO_TYPE(Cache_Entry, dlink_lru);`

Uses Cache\_Entry 437c and dlink\_lru 437e.

y un puntero al atributo dlink\_lru se obtiene mediante:

438b `<Miembros privados de entrada de Hash_Cache 437e>+≡` (437c) ◁ 437e 438d ▷  
`Dlink* link_lru() { return &dlink_lru; }`

Uses dlink\_lru 437e.

“LRU” es acrónimo inglés de *least recently used*, cual literalmente significa “menos recientemente utilizado”. En nuestro discurso castellano emplearemos la frase “más antigualemente accedido”. dlink\_lru es un doble enlace dentro de una lista doblemente enlazada circular, cuyo estado se mantiene desde el objeto Hash\_Cache mediante los siguientes atributos:

438c `<Miembros privados de Hash_Cache 437b>+≡` (437a) ◁ 438a 439b ▷  
`Dlink lru_list; // cabecera de la lista lru`  
`size_t num_lru; // número de elementos en lista lru`

Defines:

`lru_list`, used in chunks 440, 441, and 444b.

Por más antigua que pueda ser una entrada en el cache, hay situaciones en las cuales se requiere que ésta no se substituya cuando esté lleno e ingrese una nueva entrada. Este tipo de entrada se marca como “trancada” (locked) y se identifica mediante el siguiente atributo:

438d `<Miembros privados de entrada de Hash_Cache 437e>+≡` (437c) ◁ 438b 438e ▷  
`bool locked; // indica si la entrada está trancada`

Cuando obtenemos la entrada más antigualemente accedida necesitamos saber si está o no contenida en la tabla hash. Para eso mantenemos un atributo que indica este estado:

438e `<Miembros privados de entrada de Hash_Cache 437e>+≡` (437c) ◁ 438d 439a ▷  
`bool is_in_hash_table; // indica si entrada está contenida dentro`  
`// de tabla hash`

Los atributos de Cache\_Entry conforman la estructura mostrada en la figura 5.11. Son

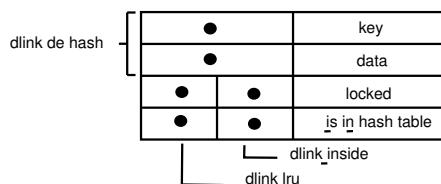


Figura 5.11: Estructura de cubeta de una entrada en el cache

tres dobles enlaces: el de la cubeta de la tabla hash (dlink de hash), dlink\_lru, que es el de la cola ordenada por tiempo de acceso, y dlink\_inside, que es el de la lista de cubetas que están dentro de la tabla hash.

A esta altura de la explicación, quizá alguien ya haya planteado como objeción el usar el encadenamiento separado en lugar de un enfoque cerrado. Si utilizásemos el sondeo

lineal, entonces tendríamos el problema de que las entradas se pueden mover y eso impide que otra estructura de dato apunte a un objeto Cache\_Entry.

Así pues, haremos que el TAD Hash\_Cache no aparte ni libere memoria para ninguna entrada cache\_entry; éstas se apartan en un arreglo contiguo en tiempo de construcción. De este modo aprovechamos el propio cache del computador y no desperdiciamos tiempo en llamadas al manejador de memoria.

Inicialmente, estas entradas están encoladas bajo el siguiente enlace:

439a *(Miembros privados de entrada de Hash\_Cache 437e) +≡ (437c) ▷438e*

```
Dlink dlink_inside; // enlace a la lista de entradas que
cuyo estado es manejado en Hash_Cache mediante los atributos:
```

439b *(Miembros privados de Hash\_Cache 437b) +≡ (437a) ▷438c 439c▷*

```
Dlink inside_list; // lista de cubetas apartadas y metidas en tabla
size_t cache_size; // máx número de entradas que puede tener cache
Defines:
```

cache\_size, used in chunks 440b and 444b.  
inside\_list, used in chunks 443a and 445.

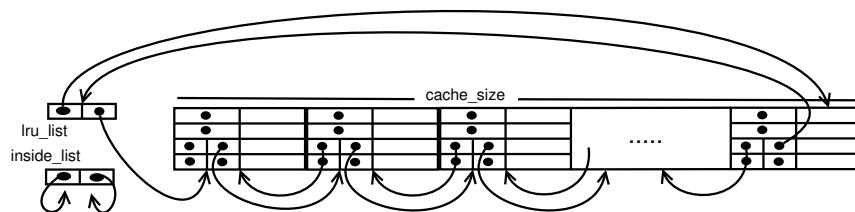


Figura 5.12: Estado inicial del arreglo contiguo de cubetas. Todas las entradas pertenecen a la lista lru.

Los enlaces a las colas de un Cache\_Entry se manejan del siguiente modo:

1. *lru\_list*: cola de entradas ordenadas desde la más antiguamente accedida hasta la más recientemente accedida. Inicialmente, el cache está vacío y todas sus entradas pre-apartadas enlazadas por esta cola.
2. *inside\_list*: lista de entradas insertadas en la tabla hash. Inicialmente, esta lista está vacía, pues el cache no contiene ningún elemento. Las entradas en esta lista también están ordenadas desde la más antiguamente accedida hasta la más recientemente accedida. Notemos sin embargo que puesto que esta cola sólo contiene entradas lógicamente pertenecientes al cache, su estado no necesariamente es el mismo que el de la cola *lru\_list*, la cual ordena entradas que pertenecen o no al cache.

La única función de la lista *inside\_list* es proveer un iterador simple sobre los elementos del cache.

Si la tabla está llena, entonces *inside\_list* deviene vacía.

3. *locked\_list*: Las entradas “trancadas” se sacan de *lru\_list* y se introducen en una tercera lista definida así:

439c *(Miembros privados de Hash\_Cache 437b) +≡ (437a) ▷439b 440a▷*  
Dlink locked\_list; // lista de entradas trancadas  
size\_t num\_locked; // número de elementos trancados

El enlace `dlink_lru` es excluyente: o la cubeta está en `lru_list` o en `locked_list`

Eventualmente, aunque no debería de ser el caso, puede aumentarse el tamaño del cache. Para eso se aparta un nuevo arreglo de `Cache_Entry`. Para poder liberarlos, los bloques de arreglos de `Cache_Entry` se enlazan en una lista que definimos del siguiente modo:

440a *(Miembros privados de Hash\_Cache 437b) +≡* (437a) ◁ 439c 440c ▷  
`typedef Dnode<Cache_Entry*> Chunk_Descriptor;`  
`Chunk_Descriptor chunk_list;`

Defines:

`chunk_list`, used in chunks 440b and 444b.

Uses `Cache_Entry` 437c and `Dnode` 83a.

`chunk_list` es el último atributo de `Hash_Cache` y define la lista de bloques (arreglos de `Cache_Entry`). Estamos listos para definir el constructor:

440b *(Miembros públicos de Hash\_Cache 440b) ≡* (437a) 443a ▷  
`Hash_Cache(size_t (*hash_fct)(const Key&),`  
`const size_t & __hash_size, const size_t & __cache_size)`  
`: hash_table(hash_fct, __hash_size, false),`  
`num_lru(0), cache_size(__cache_size), num_locked(0)`  
`{`  
`// apartar entradas del cache`  
`Cache_Entry * entries_array = new Cache_Entry [cache_size];`  
  
`// apartar el descriptor del arreglo`  
`std::unique_ptr<Chunk_Descriptor>`  
`chunk_descriptor (new Chunk_Descriptor (entries_array));`  
`chunk_list.insert(chunk_descriptor.get());`  
  
`// insertar cada Cache_Entry en lista lru`  
`for (int i = 0; i < cache_size; i++)`  
`insert_entry_to_lru_list(&entries_array[i]);`  
  
`chunk_descriptor.release();`  
`}`

Uses `Cache_Entry` 437c, `cache_size` 439b, `chunk_list` 440a, and `insert_entry_to_lru_list` 440c.

Observemos que el constructor apela a un método interno `insert_entry_to_lru_list()`, el cual, junto con otros métodos privados en torno a `lru_list`, se definen del siguiente modo:

440c *(Miembros privados de Hash\_Cache 437b) +≡* (437a) ◁ 440a 441a ▷  
`void insert_entry_to_lru_list(Cache_Entry * cache_entry)`  
`{`  
`num_lru++;`  
`lru_list.insert(cache_entry->link_lru());`  
`}`  
`void remove_entry_from_lru_list(Cache_Entry * cache_entry)`  
`{`  
`num_lru-;`  
`cache_entry->link_lru()->del();`  
`}`

Defines:

`insert_entry_to_lru_list`, used in chunks 440b, 443d, and 444b.  
`remove_entry_from_lru_list`, used in chunk 443c.

Uses Cache\_Entry 437c and lru\_list 438c.

La misma clase de rutinas se define para la lista `locked_list`.

El fin de la cola lru\_list no es otro que instrumentar el orden según el acceso. Hay varias situaciones en las cuales esta cola tiene que actualizarse:

1. Cuando se inserta un nuevo elemento en el cache, obtenemos su Cache\_Entry del frente de lru\_list, el cual contiene al elemento más antiguamente accedido -recordemos que todas las entradas Cache\_Entry se meten en esta cola durante la construcción del cache-.
  2. Cuando referenciamos a un elemento dentro del cache debemos especificar que éste es el más recientemente accedido; para eso eliminamos su Cache\_Entry a través del doble enlace dlink\_lru y lo re-insertamos en el trasero. Esto lo ejecuta la siguiente rutina:

**Defines:**

do\_mru, used in chunks 441c and 443b.  
Uses Cache\_Entry 437c and lru\_list 43

Uses Cache\_Entry::get() and I18n::list().

`do_mru()` hace a `cache_entry` la entrada más recientemente accedida.

3. Cuando eliminamos un elemento del cache lo ubicamos en el frente, acción que instrumenta la siguiente rutina:

Ahora estamos prestos para mostrar una rutina crítica, la cual selecciona la entrada

```

 do_lru(cache_entry);
 }
Cache_Entry * get_lru_entry()
{
 // obtenga entrada más antigua; menos recientemente accedida
 Dlink * lru_entry_link = lru_list.get_prev();
 Cache_Entry * cache_entry = dlink_lru_to_Cache_Entry(lru_entry_link);

 // si cache_entry contenida en tabla ==> eliminarlo
 if (cache_entry->is_in_hash_table)
 remove_entry_from_hash_table(cache_entry);

 do_mru(cache_entry); // entrada deviene más recientemente accedida

 return cache_entry;
}

```

Defines:

get\_lru\_entry, used in chunk 443a.  
remove\_entry\_from\_hash\_table, used in chunk 444a.  
Uses Cache\_Entry 437c, do\_mru 441a, and lru\_list 438c.

remove\_entry\_from\_hash\_table() elimina cache\_entry de la tabla hash y la torna como la entrada más antiguamente accedida. get\_lru\_entry() es la rutina crítica que selecciona

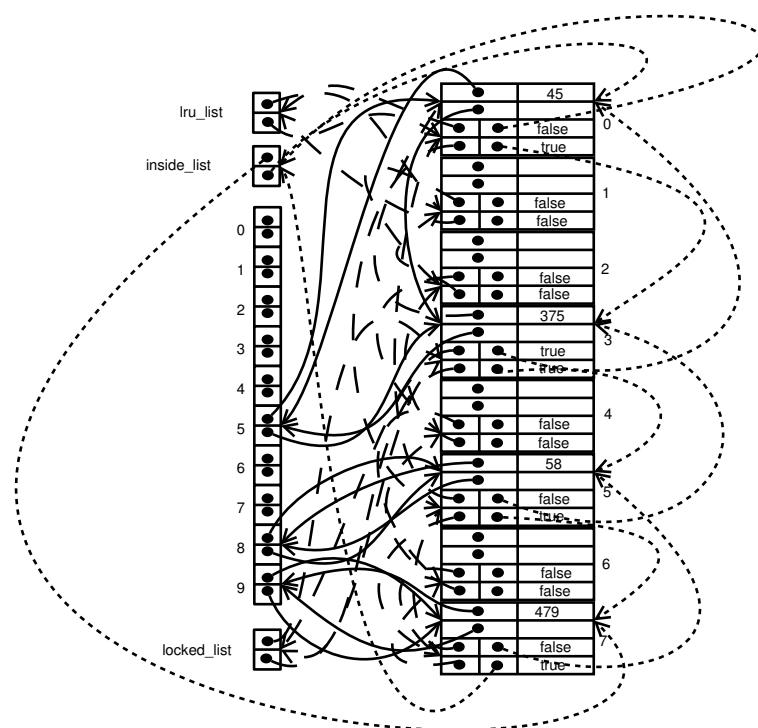


Figura 5.13: Estado de un cache de tamaño 8 con tabla hash de tamaño 10. Se muestra el arreglo de la tabla hash y el arreglo preapartido de cubetas. Los enlaces de la tabla hash son contiguos y los de la lista de entradas que están dentro de la tabla punteados y los de la lista LRU, trazados. La entrada con clave 375, que es colisión en la tabla hash con la clave 45, está “trancada”, por eso pertenece a la lista locked\_list

la entrada más antiguamente accedida y a la cual se invoca cuando se inserta un nuevo elemento en el cache.

Explicado esto, podemos mostrar la inserción:

443a *(Miembros públicos de Hash\_Cache 440b) +≡* (437a) ◁ 440b 443b ▷

```
Cache_Entry * insert(const Key & key, const Data & data)
{
 Cache_Entry * cache_entry = get_lru_entry(); // entrada más antigua
 cache_entry->get_key() = key; // escribirle el par
 cache_entry->get_data() = data;
 inside_list.insert(cache_entry->link_inside());
 hash_table.insert(cache_entry);
 cache_entry->is_in_hash_table = true;
 return cache_entry;
}
```

Uses Cache\_Entry 437c, get\_lru\_entry 441c, and inside\_list 439b.

El cache mantiene un subconjunto de los últimos cache\_size elementos accedidos con la esperanza de que, si se referencian de nuevo, entonces no se pague un coste alto por su búsqueda en el conjunto principal. Para buscar en el cache proveemos la siguiente rutina:

443b *(Miembros públicos de Hash\_Cache 440b) +≡* (437a) ◁ 443a 443c ▷

```
Cache_Entry * search(const Key & key)
{
 // buscar en la tabla hash
 Cache_Entry * cache_entry = (Cache_Entry*) hash_table.search(key);
 if (cache_entry != NULL) // ¿fue encontrada la clave?
 {
 // si ==> hacerla la más recientemente usada
 do_mru(cache_entry);
 move_to_inside_front(cache_entry);
 }
 return cache_entry;
}
```

Uses Cache\_Entry 437c and do\_mru 441a.

La búsqueda “refresca la entrada en el cache” en el sentido de que la hace la más recientemente accedida.

Para “trancar” una entrada, es decir, para asegurar que ella no sea sacada del cache cuando ocurra una inserción, se emplea el siguiente método:

443c *(Miembros públicos de Hash\_Cache 440b) +≡* (437a) ◁ 443b 443d ▷

```
void lock_entry(Cache_Entry * cache_entry)
{
 remove_entry_from_lru_list(cache_entry);
 insert_entry_to_locked_list(cache_entry);
 cache_entry->lock();
}
```

Uses Cache\_Entry 437c and remove\_entry\_from\_lru\_list 440c.

Análogamente, la entrada se puede “destrancar”:

443d *(Miembros públicos de Hash\_Cache 440b) +≡* (437a) ◁ 443c 444a ▷

```
void unlock_entry(Cache_Entry * cache_entry)
{
 remove_entry_from_locked_list(cache_entry);
 insert_entry_to_lru_list(cache_entry);
 cache_entry->unlock();
}
```

```
}
```

Uses Cache\_Entry 437c and insert\_entry\_to\_lru\_list 440c.

La eliminación de un elemento en el cache puede parecer una operación redundante e innecesaria, pues sólo basta con no acceder más a una entrada para que ésta salga del cache. Sin embargo, el conocimiento certero de que una clave jamás será accedida no sólo permite disponer su entrada para una inserción, sino que potencia a otras claves de beneficiarse del cache. Por eso, según la índole de la aplicación, puede o no ser muy valiosa la siguiente primitiva de eliminación:

444a *<Miembros públicos de Hash\_Cache 440b>+≡* (437a) ◁443d 444b▷

```
void remove(Cache_Entry * cache_entry)
{
 remove_entry_from_hash_table(cache_entry);
}
```

Uses Cache\_Entry 437c and remove\_entry\_from\_hash\_table 441c.

La selección del tamaño del cache puede ser crítica para el desempeño global de un sistema. Una capacidad insuficiente puede hacer que las inserciones reemplacen a entradas que serán referenciadas. Si esto ocurre, entonces la búsqueda siempre será vana y el desempeño del acceso global se degradará a un punto peor que si no se tuviese el cache. A este fenómeno se le llama “thrashing”<sup>23</sup> y puede ser sumamente costoso si el cache se emplea en el camino crítico de un sistema; cuestión que suele ser el caso.

Así pues, cuando se opta por el empleo de un cache, se debe, en la mayor medida posible, disponer de una estimación precisa de la relación de referencia al conjunto. En esto puede ser muy importante la relación 80-20 (§ 3.5.1 (Pág. 212)), pero cuenta habida del coste que puede acarrear un cambio o error en la estimación, es recomendable que el tamaño del cache pueda ajustarse en tiempo de ejecución. La idea es que si durante la ejecución se revela necesario hacer un ajuste, entonces éste sea posible, lo cual dista de ser un enfoque dinámico como el de una tabla hash lineal. En virtud de esto diseñamos la siguiente rutina de expansión:

444b *<Miembros públicos de Hash\_Cache 440b>+≡* (437a) ◁444a 445▷

```
void expand(const size_t & plus_size)
{
 const size_t new_cache_size = cache_size + plus_size;

 // apartar plus_size nuevas entradas
 Cache_Entry * entries_array = new Cache_Entry [plus_size];
 std::unique_ptr<Chunk_Descriptor> // apartar el descriptor
 chunk_descriptor (new Chunk_Descriptor (entries_array));

 // Calcular nuevo tamaño de tabla y relocalizar sus entradas
 const float curr_hash_ratio = 1.0*cache_size/hash_table.capacity();
 const size_t new_hash_capacity = new_cache_size/curr_hash_ratio;

 hash_table.resize(new_hash_capacity);

 // meter nuevas entradas en lru_list
```

---

<sup>23</sup>La traducción literal de este término es difícil porque pensamos que no existe un equivalente directo castellano. En el contexto del cache, “thrashing” alegoriza “flagelar”, pues, por lo general, cuando un cache entra en este estado acarrea una penalidad demasiado severa sobre el sistema que lo usa.

```

 for (int i = 0; i < plus_size; i++)
 insert_entry_to_lru_list(&entries_array[i]);

 chunk_list.insert(chunk_descriptor.release());
 cache_size = new_cache_size;
}

```

Uses Cache\_Entry 437c, cache\_size 439b, chunk\_list 440a, expand 417a, insert\_entry\_to\_lru\_list 440c, and lru\_list 438c.

Nos falta un último detalle: un iterador sobre los elementos del cache:

445 ⟨Míembros públicos de Hash\_Cache 440b⟩+≡ (437a) ▷444b

```

class Iterator : public Dlink::Iterator
{
 Iterator(Hash_Cache & cache) : Dlink::Iterator(&cache.inside_list) {}

 Cache_Entry * get_current()
 {
 Dlink * dl = Dlink::Iterator::get_current();
 return Cache_Entry::dlink_inside_to_Cache_Entry(dl);
 }
};

```

Uses Cache\_Entry 437c and inside\_list 439b.

Puesto que se itera sobre la lista inside\_list, sólo los elementos contenidos en el cache son vistos. Puesto que esta lista está ordenada por tiempo de acceso, el orden de visita es desde el más recientemente accedido hasta el menos recientemente accedido.

## 5.4 Notas bibliográficas

En su excelso libro sobre programación, van der Linden[171] menciona, metafóricamente, que si a él le tocase estar en una isla desierta y le diesen a escoger una sola estructura de datos, entonces ésta sería la tabla hash. ¿por qué un programador virtuoso considera tan vital esta estructura? Quizá porque él no sea un ser dominado por una idea determinista del mundo.

En lo concreto de la programación podemos decir que la dispersión es una técnica completamente basada en la probabilidad, es decir, crudamente hablando, en eso que ancestralmente llamamos la suerte o azar, y que hoy se mira con desdén. Cómo será de extraña esta filia por el determinismo que muy pocos programadores se percatan de que emplear una buena función de dispersión para buscar en un arreglo, sin preocuparse de las colisiones ni de marcar las celdas, tiene muchas mejores probabilidades de encontrar rápidamente una clave que la mera búsqueda secuencial.

*“El hombre que dijo: preferiría ser afortunado que bueno, tenía una profunda perspectiva de la vida. La gente teme reconocer que una gran parte de la vida depende de la suerte. Da miedo pensar en todo lo tanto sobre lo que no tenemos control. Hay momentos durante un partido en que la pelota golpea la red, y por una fracción de segundo puede seguir hacia delante o bien caer hacia atrás. Con un poco de suerte, sigue hacia adelante y ganas ... o quizás no, y pierdes”<sup>24</sup>.* Esta cita de una película de Woody Allen nos indica un poco el desdén que en esta época tenemos hacia el azar. Hace

---

<sup>24</sup>Preámbulo de la película “Match Point”.

unos 2500, años Demócrito, quizá uno de los primeros que pensaba en la idea actual de átomo, decía que “*en el azar y la espontaneidad hay conciencia*”.

Maquiavelo vio en la suerte un factor determinante para el destino de los gobernantes. Alegorizaba la vida como un río cuyo fluir cotidiano la suerte podía cambiar; una inundación, por ejemplo. Bajo ese sentido alegórico, él recomendaba prepararse para cuando la suerte tornase y lo alegorizaba con la construcción de un dique. En la vida moderna, esto puede traducirse al ahorro. En la vida de la programación, a una estrategia de resolución de colisiones.

Según Knuth, la idea de dispersión fue descubierta independientemente por H. P. Luhn y Amdahl *et al* en 1953 [99].

El encadenamiento separado y el método de división aparece primero descubierto por Dumey [39].

El sondeo lineal fue sugerido por Ershov [47], cuyo análisis, extremadamente dificultoso, fue descubierto por primera vez por Knuth en 1962, como él mismo explica en una nota a pie de página de [99] pag. 536.

El modelo de sondeo ideal, extremadamente útil como marco de análisis de las técnicas de sondeo, fue introducido por Peterson en 1957 [143].

La paradoja del cumpleaños es un problema clásico de la teoría de probabilidades. Una presentación rigurosa puede encontrarse en el clásico Feller [48, 49]. Esta paradoja fue generalizada en la célebre función Q de Ramanujan [51]. Un análisis de su aplicación en la programación puede encontrarse en [50].

La técnica presentada en este texto para eliminar las celdas marcadas DELETED con el doble hash fue presentada por primera vez en [68]. Una implantación concreta, aparte de la aquí expuesta, no es conocida públicamente.

La dispersión universal fue descubierta en 1977 por Carter y Wegman [28].

La dispersión lineal fue popularizada por el Paul Larson [105] y su descubrimiento fue reportado por primera vez en 1980 por Witold Litwin [110], en el dominio de las bases de datos.

La dispersión perfecta aparece a finales de los años setenta. Un artículo paradigmático es de Cichelli [29].

## 5.5 Ejercicios

1. Escriba un algoritmo de eliminación de una tabla hash con colisiones separadamente encadenadas en listas simplemente enlazadas.
2. La función de eliminación de los TAD LhashTable<Key> y DynLhashTable no verifica si el registro a eliminar pertenece a la tabla. Discuta diferentes enfoques para efectuar esta verificación.
3. Demuestre la proposición 5.2. (+)
4. Explique detalladamente la construcción de la expresión (5.27).
5. Asumiendo encadenamiento separado con listas simples y una tabla de puros punteros, calcule el consumo de espacio. ¿A partir de cuál valor de  $\alpha$  y  $S_T$  el direccionamiento cerrado es menos costoso en espacio?

6. Considere una tabla hash con resolución de colisiones por encadenamiento separado (con listas enlazadas), un tamaño de tabla de  $M = 13$  y una función hash  $h(k) = k \bmod M$ .
- Dibuje la tabla resultante de la siguiente secuencia de inserción  
23, 13, 20, 5, 7, 2, 40, 50, 30, 45, 12, 21, 33, 34
- (b) Asumiendo que las claves están entre 1 y 100, que no se repiten y que cada clave tiene la misma probabilidad de buscarse, calcule:
- Probabilidad de que una clave se encuentre en la tabla.
  - Probabilidad de que una clave se encuentre en la tabla y que se requiera recorrer dos (2) o más nodos durante su búsqueda.
7. Asuma una tabla hash con resolución de colisiones con encadenamiento separado. Asuma una cantidad esperada de claves de  $10^5$  con una desviación de  $10^4$ . Sugiera un tamaño de tabla tal que el factor de carga  $\alpha$  no exceda del 97 %. Calcule la longitud esperada de cada lista de colisión  $l_i$  y su desviación típica.
8. La estructura de cubeta utilizada por el TAD ODHashTable utiliza un campo especial denominado `probe_type`. Deduzca la manera en que se puede determinar en tiempo de ejecución la información aportada por este campo. Dicho de otro, ¿cómo puede prescindirse de este campo?
9. En el TAD ODHashTable, deduzca una manera de prescindir de los campos de la cubeta `status` y `probe_type`. (+)
10. Dada una tabla hash con resolución de colisiones por direccionamiento abierto, deduzca un algoritmo general de dos fases que reduzca al mínimo la cantidad de cubetas con estado `DELETED`. El algoritmo debe poder ser aplicado para cualquier estrategia de resolución por direccionamiento abierto.  
La primera fase consiste en examinar la tabla y marcar con `EMPTY` todas las entradas con valor `DELETED`. La segunda fase estudia las cubetas con valor `BUSY` y determina cuáles cubetas con estado `EMPTY` deben ser cambiadas al estado `DELETED` de manera que la clave pueda ser localizada.  
No está permitido mover los registros contenidos en las cubetas. (+)
11. Enuncie las ventajas y desventajas del algoritmo enunciado en la pregunta anterior.
12. Calcule una expresión similar a (5.27) que compare el coste en espacio del TAD ODHashTable con el de OLHashTable.
13. Sea una tabla hash de tamaño  $m = 23$  con resolución de colisiones por direccionamiento abierto y uso de dos funciones hash:
- $h_1(k) = k \bmod 23$
  - $h_2(k) = 1 + (k \bmod 19)$

Asuma como esquema de asignación de elemento la fórmula  $(h_1(k) + i \times h_2(k)) \bmod m$ ,  $m = 23$ , donde  $i$  es el número de colisiones de  $k$  al momento de la inserción. Es

decir, si  $h_1(k)$  no tiene colisión ( $i = 0$ ), entonces el elemento es insertado en la posición  $h_1(k)$ . Si hay una colisión, entonces el elemento es insertado en  $h_1(k) + h_2(k)$ ; si hay dos colisiones, entonces el elemento es insertado en  $h_1(k) + 2h_2(k)$ , y así sucesivamente.

- (a) Escriba la posición dentro de la tabla (o dibuje la tabla) luego de la siguiente secuencia de inserción

4    13    27    31    28    26    40    23    47    1

- (b) Asuma que ocurre una consulta entre un total de 20 posibles claves.

- i. ¿Cuál es la probabilidad de encontrar la clave?
- ii. ¿Cuál es la probabilidad de no encontrar la clave?
- iii. ¿Cuál es la probabilidad de que una búsqueda exitosa sea resuelta con la primera función hash?
- iv. ¿Cuál es la probabilidad de que una búsqueda exitosa sea resuelta con la segunda función hash?

14. Haga un análisis comparativo entre los TAD DynLHashTable y ODhashTable. Enuncie ventajas, desventajas, similitudes y diferencias.

15. Dada una tabla hash lineal, calcule la cantidad esperada de llamadas a la función hash cada vez que ocurre una inserción o búsqueda. (+)

16. La implantación de DynLHashTable hereda de LHashTable<Key> la implantación y parte de su interfaz. Puesto que DynLHashTable es derivada públicamente de LHashTable<Key>, hay métodos de LHashTable<Key>, que no son parte de la interfaz de DynLHashTable, que pueden llamarse desde una instancia de DynLHashTable. Por ejemplo, el usuario podría invocar la inserción de LHashTable<Key> desde un objeto de tipo DynLHashTable:

```
DynHashTable<int, Record> table;

LHashTable<int>::Bucket bucket = new LHashTable<int>::Bucket (key);

table.insert(bucket);
```

Explique un mecanismo para evitar que esto ocurra. Es decir, para impedir o, en el peor de los casos, para detectar que el usuario llamó a la inserción de LHashTable<Key> y reportar un error en tiempo de ejecución.

17. Considere el problema de la existencia de un objeto. Esto es, dado un puntero a un objeto de tipo, digamos Object, se desea verificar si la dirección de memoria apunta efectivamente a un objeto de ese tipo.

Discuta todas las alternativas posibles para diseñar un sistema general de verificación de existencia de objetos.

18. Implante las clases LHashTable<Key> y LHashTableVtl<Key> para que utilicen listas simplemente enlazadas en lugar de doblemente enlazadas.

19. Implante las clases `LHashTable<Key>` y `LHashTableVt1<Key>` para que utilicen listas enlazadas ordenadas. Haga un estudio comparativo con la implantación tradicional explicada en § 5.1.3.1 (Pág. 383).
20. Calcule la probabilidad de que la operación  $k \bmod 150$  sólo tome en cuenta los dos dígitos menos significativos.
21. Considere una tabla hash lineal con valor inicial  $M = 5$  y un factor de carga máximo permitido de  $\alpha_u = 0,9$ . Para la siguiente secuencia de inserción:

181 186 10 6 5 58 191 22 113 7 83 118 138 122 16 18 150 39 48 157

Dibuje el estado resultante de la tabla luego de las inserciones.



# 6

## Árboles de búsqueda equilibrados

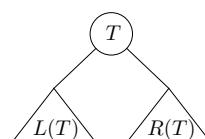
Quizá alguna vez la noción de equilibrio se nos haya aparecido bajo una idea ancestralmente inmemorable y cuyo aspecto visual es el siguiente:



La imagen pictoriza una artefacto antiquísimo, con tendencia actual al desuso, pero aún vigente en algunos escasos contextos, cuya finalidad es pesar, o sea, medir pesos. La idea es poner en uno de los platillos lo que se desea pesar, mientras que en el otro se ponen pesos de referencia actualmente llamados “plomadas”. Cuando los platillos alcanzaban igual altura se decía que había “*æquilibrium*”; de “*aequus*”, que significa “igual”, y “*libra*”, que era el nombre romano de la plomada y aún vigente como una medida real de peso o valor económico.

El artefacto en cuestión, hogaño se le denomina balanza, pero antaño los romanos también lo mentaron como “*libra*”, pues éstos refinaron el mecanismo de tal forma que el brazo de la balanza podía desplazarse y sustituir algunas plomadas de referencia. Desde entonces, la balanza o libra ha simbolizado situaciones en las cuales se anhela alguna especie de igualdad o equilibrio; en particular, ha sido el símbolo de la justicia, quizás la virtud más difícil de cultivar.

En el ámbito de los árboles binarios, el equilibrio se pictoriza bajo la ya bien conocida imagen siguiente:



Las ramas del árbol  $T$  tienen pesos equivalentes, es decir, contienen las mismas cantidades de nodos.  $T$  estaría, entonces, en equilibrio o equilibrado. Recursivamente, si todos los nodos de  $T$  están equilibrados, entonces la altura de  $T$  tiende a  $\mathcal{O}(\lg(n))$ , lo cual hace que el rendimiento de la búsqueda sobre un árbol sea también  $\mathcal{O}(\lg(n))$ . Al igual que ocurre con las virtudes, los árboles abstractos son buenos en la medida en que éstos estén equilibrados.

## 6.1 Equilibrio de árboles

En una primera instancia podemos intentar alcanzar un equilibrio basado en la siguiente definición:

**Definición 6.1 (Equilibrio fuerte de Wirth [182])** Un árbol binario T es equilibrado  $\iff \forall n_i \in T, ||L(n_i)| - |R(n_i)|| \leq 1$

Bajo esta definición podemos diseñar un primer algoritmo que nos equilibre un árbol. Nuestro algoritmo utiliza los árboles extendidos (ABBE) explicados en § 4.11 (Pág. 335). Las operaciones requeridas las incluimos en el archivo *<tpl\_balanceXt.H (never defined)>*.

Una de las ventajas de un ABBE es la operación de selección (§ 4.11.1 (Pág. 337)). Mediante esta operación podríamos seleccionar la clave del centro y hacerla raíz del árbol. El número de nodos en ambos lados diferiría a lo sumo en uno. Si aplicamos este procedimiento recursivamente obtenemos un árbol equilibrado según Wirth.

Requerimos una función que seleccione un nodo según su posición infija y lo suba hasta la raíz. Tal operación se define como sigue:

452a

*(Rutinas de equilibrio 452a)*≡

452b▷

```
template <class Node> inline
Node * select_goto_up_root(Node * root, const size_t & i)
{
 if (i == COUNT(LLINK(root)))
 return root;

 if (i < COUNT(LLINK(root)))
 {
 LLINK(root) = select_goto_up_root(LLINK(root), i);
 root = rotate_to_right_xt(root);
 }
 else
 {
 RLINK(root) = select_goto_up_root(RLINK(root),
 i - COUNT(LLINK(root)) - 1);
 root = rotate_to_left_xt(root);
 }
 return root;
}
```

*select\_goto\_up\_root()* selecciona el  $i$ -ésimo nodo del árbol con raíz *root* y lo sube hasta la raíz. Si tomamos un árbol de  $n$  nodos, seleccionamos el nodo correspondiente a la posición  $n/2$  y lo subimos hasta la raíz, entonces, a nivel del nodo  $n/2$ , la diferencia de nodos entre su subárbol izquierdo y derecho es a lo sumo uno. Si aplicamos el mismo principio recursivamente, deducimos el algoritmo siguiente:

452b

*(Rutinas de equilibrio 452a)*+≡

◀452a

```
template <class Node> inline Node * balance_tree(Node * root)
{
 if (COUNT(root) <= 1)
 return root;
```

```

root = select_gotoup_root(root, COUNT(root) / 2);
LLINK(root) = balance_tree(LLINK(root));
RLINK(root) = balance_tree(RLINK(root));

return root;
}

```

`select_gotoup_root()` puede realizarse mediante la partición por posición explicada en § 4.11.7 (Pág. 341). Esto es delegado a ejercicio.

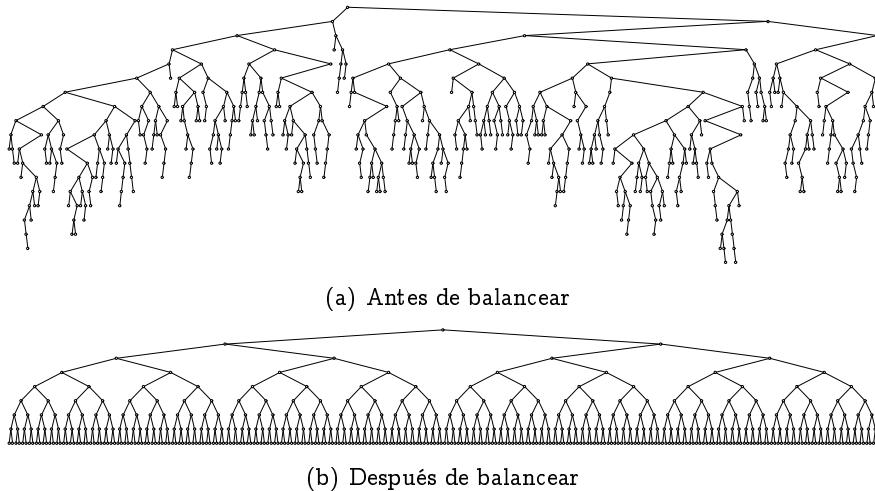


Figura 6.1: Ejemplo de equilibrado de un árbol aleatorio mediante `balance()`

Si el árbol original fue construido a partir de una secuencia aleatoria, entonces su altura tiende a  $\mathcal{O}(\lg n)$  y el desempeño de `select_gotoup_root()` llamado desde `balance_tree()` es  $\mathcal{O}(\lg n)$ . En este momento, el árbol queda particionado en dos partes equitativas, cada una con un consumo de tiempo proporcional a la mitad de la entrada. Podemos, pues, plantear la siguiente ecuación recurrente para el desempeño de `balance_tree()`:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T(n/2) + \mathcal{O}(\lg n) & \text{si } n > 1 \end{cases} .$$

Suponiendo que  $n$  es una potencia exacta de 2, realizamos la transformación  $n = 2^k \implies k = \lg n$ . Esto nos plantea:

$$T(2^k) = 2T(2^{k-1}) + k .$$

Dividiendo la ecuación entre  $2^k$ , tenemos:

$$\begin{aligned} \frac{T(2^k)}{2^k} &= \frac{T(2^{k-1})}{2^{k-1}} + \frac{k}{2^k} \\ &= \frac{T(2^{k-2})}{2^{k-2}} + \frac{k-1}{2^{k-1}} + \frac{k}{2^k} \\ &= \frac{T(2^{k-3})}{2^{k-3}} + \frac{k-2}{2^{k-2}} + \frac{k-1}{2^{k-1}} + \frac{k}{2^k} = 1 + \sum_{i=1}^k \frac{i}{2^i} \implies \end{aligned}$$

$$\frac{T(n)}{n} = 1 + \sum_{i=1}^k \frac{i}{2^i} \implies T(n) = n + n \underbrace{\sum_{i=1}^k \frac{i}{2^i}}_{S_k} . \quad (6.1)$$

La solución de la recurrencia depende, pues, del valor de la sumatoria  $S_k = \sum_{i=1}^k \frac{i}{2^i}$ , la cual puede resolverse por perturbación:

$$\begin{aligned}
 S_k &= \sum_{i=1}^k \frac{i}{2^i} \Rightarrow \\
 S_k + \frac{k+1}{2^{k+1}} &= \sum_{i=0}^k \frac{i+1}{2^{i+1}} \\
 &= \underbrace{\sum_{i=0}^k \frac{i}{2^{i+1}}}_{\frac{S_k}{2}} + \sum_{i=0}^k \frac{1}{2^{i+1}} \Rightarrow \\
 \frac{S_k}{2} &= \underbrace{\sum_{i=0}^k \frac{1}{2^{i+1}}}_{Q_k} - \frac{k+1}{2^{k+1}} . \tag{6.2}
 \end{aligned}$$

A la vez, (6.2) depende de la solución de:

$$Q_k = \sum_{i=0}^k \frac{1}{2^{i+1}} = \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k+1}} . \tag{6.3}$$

Multiplicando (6.3) por 2 tenemos:

$$2Q_k = 1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^k} . \tag{6.4}$$

Ahora restamos (6.4) menos (6.3) lo que nos da como solución de (6.3):

$$Q_k = 1 - \frac{1}{2^{k+1}} ; \tag{6.5}$$

cuyo valor, al sustituirlo en (6.2) nos arroja:

$$S_k = 2 - \frac{1}{2^k} - \frac{k+1}{2^k} = 2 - \frac{k+2}{2^k} \tag{6.6}$$

Sustituyendo (6.6) en (6.1) tenemos:

$$T(n) = n + n \left( 2 - \frac{k+1}{2^k} \right) = n + n \left( 2 - \frac{\lg n + 2}{n} \right) = 3n - \lg n - 2, \quad n \geq 1 ; \tag{6.7}$$

resultado que es  $\mathcal{O}(n \lg n)$ .

`balance_tree()` es, pues,  $\mathcal{O}(n \lg n)$ , un tiempo costoso si la función se invoca frecuentemente. Hasta el presente, nadie ha descubierto un algoritmo aceptable que modifique un ABB y garantice una condición de equilibrio fuerte, razón por la cual se ha optado por relajar las condiciones de equilibrio para hacerlas menos restrictivas. Según las técnicas algorítmicas, podemos plantear la siguiente clasificación de los mecanismos de equilibrio conocidos:

- **Equilibrio probabilístico:** este enfoque consiste en tomar decisiones aleatorias tendientes a restablecer el equilibrio. Los árboles obtenidos ofrecen un tiempo esperado de  $\mathcal{O}(\lg n)$  para todas las operaciones.

De esta clase de equilibrio estudiaremos dos estructuras: los árboles aleatorizados y los treaps.

- **Equilibrio garantizado:** en este enfoque se plantean condiciones de equilibrio, menos restrictivas que la fuerte, que permiten modificaciones  $\mathcal{O}(\lg n)$  sobre un árbol binario y que acotan la altura a  $\mathcal{O}(\lg n)$ . Los métodos de este tipo ofrecen una garantía  $\mathcal{O}(\lg n)$  en todas las operaciones.

Bajo este tipo de equilibrio estudiaremos dos estructuras: los árboles AVL y los árboles rojo-negros.

- **Equilibrio amortizado:** este enfoque consiste en ejecutar operaciones especiales durante la ejecución de cualquier operación de manera tal que el árbol tienda a estar equilibrado. Bajo esta concepción se garantiza un coste amortizado de  $\mathcal{O}(p \lg n)$  para  $p$  operaciones sucesivas sobre el árbol.

En este grupo, la única técnica amortizada conocida para equilibrar árboles es el desplegado (splay), empleada en los árboles splay.

## 6.2 Árboles aleatorizados

Un árbol binario de búsqueda aleatorizado (ABBA) es un árbol binario de búsqueda extendido (ABBE en § 4.11 (Pág. 335)) con operaciones especiales que garantizan que el ABBA sea un árbol aleatorio. Recordemos que un ABBE es un árbol en el que cada nodo almacena la cardinalidad del árbol del cual él es raíz.

Notemos la distinción que hacemos entre “aleatorio” y “aleatorizado”. Informalmente, un árbol aleatorio es equivalente a un árbol binario de búsqueda construido a partir de secuencias de inserción aleatorias, mientras que uno aleatorizado es uno tal que sus operaciones garantizan que éste siempre sea aleatorio, independientemente del tipo de operación.

**Definición 6.2 (Árbol binario de búsqueda aleatorio (ABBA))** Sea  $T$  un árbol binario de búsqueda con cardinalidad  $|T| = n$ .

- Si  $n = 0$ , entonces  $T = \emptyset$  es un ABBA.
- Si  $n > 0$ , entonces  $T$  es un ABBA si y sólo si  $L(T)$  y  $R(T)$  son ABBA independientes,

y

$$P(|L(T)| = i \mid |T| = n) = \frac{1}{n}, \quad 0 \leq i < n, \quad n > 0 \quad (6.8)$$

El punto crucial de la definición, específicamente la ecuación (6.8), es que cualquiera de las claves del árbol tiene exactamente la misma probabilidad de ser la raíz del árbol. Esta propiedad es fundamental para el diseño de los algoritmos de inserción y eliminación. Para producir un ABBA, cualquier clave a insertar debe tener alguna posibilidad de devenir la raíz del árbol, o la raíz de alguno de sus subárboles. Del mismo modo, cuando se elimina una clave, cualquiera de las restantes debe tener posibilidades de devenir raíz del árbol o de sus subárboles.

### 6.2.1 El TAD Rand\_Tree<Key>

Antes de explicar los algoritmos requerimos algunas bases para un TAD que nos modelice un ABBA. Tal TAD se denomina Gen\_Rand\_Tree<Key> y se encuentra en el archivo

*<tpl\_rand\_tree.H 456a>* cuya estructura fundamental es la siguiente:

456a *<tpl\_rand\_tree.H 456a>*≡  
 template <template <typename> class NodeType, typename Key, class Compare>  
 class Gen\_Rand\_Tree  
 {  
 typedef NodeType<Key> Node;  
*<miembros privados de Gen\_Rand\_Tree<Key> 456d>*  
*<miembros públicos de Gen\_Rand\_Tree<Key> 456c>*  
 };  
*<clases públicas de Rand\_Tree<Key> 456e>*

El primer paso para implantar *Gen\_Rand\_Tree<Key>* es definir la estructura del nodo que conforma un ABBA:

456b *<tpl\_randNode.H 456b>*≡  
 DECLARE\_BINNODE\_SENTINEL(RandNode, 80, BinNodeXt\_Data);  
 Uses DECLARE\_BINNODE\_SENTINEL.

Internamente, el TAD *Gen\_Rand\_Tree<Key>* trabaja con el mismo tipo de nodo empleado para los árboles con rango estudiados en § 4.11 (Pág. 335): los nodos guardan la cardinalidad del árbol y *NullPtr* es un nodo centinela cuya cardinalidad es cero.

La clase *Gen\_Rand\_Tree<Key>* requiere un generador de números pseudoaleatorios. Para eso nos valemos de la biblioteca *gsl*<sup>1</sup>. Usaremos el generador “tornado” (“Twisted”), actualmente conocido como el mejor generador conocido de números seudo-aleatorios [119, 118, 120]. Un puntero al objeto *gsl*, generador de números aleatorios, puede obtenerse mediante:

456c *<miembros públicos de Gen\_Rand\_Tree<Key> 456c>*≡ (456a) 457b▷  
 gsl\_rng \* gsl\_rng\_object() { return r;}

A través de este puntero, el usuario puede alterar el generador a su pleno riesgo.

Los miembros dato de *Gen\_Rand\_Tree<Key>*:

456d *<miembros privados de Gen\_Rand\_Tree<Key> 456d>*≡ (456a) 456f▷  
 Node \* tree\_root;  
 gsl\_rng \* r;

*tree\_root* es un apuntador a la raíz de árbol aleatorizado. *r* es un apuntador al objeto de generación de números pseudoaleatorios de la biblioteca *gsl*.

La clase que se exporta al usuario:

456e *<clases públicas de Rand\_Tree<Key> 456e>*≡ (456a)  
 template <typename Key, class Compare = Aleph::less<Key> >  
 class Rand\_Tree : public Gen\_Rand\_Tree<RandNode, Key, Compare> {};

### 6.2.1.1 Inserción en un ABBA

La idea básica del algoritmo de inserción es efectuar decisiones aleatorias, en función de la definición 6.2, que determinen si el nodo a insertar deviene o no raíz y garanticen que el árbol binario resultante sea aleatorio. Para eso presentamos el algoritmo siguiente:

456f *<miembros privados de Gen\_Rand\_Tree<Key> 456d>*+≡ (456a) ▷456d  
 Node \* random\_insert(Node \* root, Node \* p)  
 {  
 const long & n = COUNT(root);

---

<sup>1</sup>Gnu Scientific Library [66].

```

⟨Genera rn aleatorio entre 0 y n 457a⟩;
if (rn == n) // ¿Gana p el sorteo de ser raíz?
 return insert_root_xt <Node, Compare> (root, p); // sí

Node * result;
if (Compare () (KEY(p), KEY(root))) // KEY(p) < KEY(root) ?
{ // sí ==> insertar en árbol izquierdo
 result = random_insert(LLINK(root), p);
 if (result != Node::NullPtr) // ¿hubo inserción?
 { // si ==> actualizar rama y contadores
 LLINK(root) = result;
 ++COUNT(root);
 return root;
 }
}
else if (Compare() (KEY(root), KEY(p))) // KEY(p) > KEY(root) ?
{ // insertar en árbol derecho
 result = random_insert(RLINK(root), p);
 if (result != Node::NullPtr) // ¿hubo inserción?
 { // si ==> actualizar rama y contadores
 RLINK(root) = result;
 ++COUNT(root);
 return root;
 }
}
return Node::NullPtr; // clave duplicada ==> no hay inserción
}

```

Defines:

random\_insert, used in chunk 457b.

El punto central de la inserción es el sorteo efectuado en ⟨Genera rn aleatorio entre 0 y n 457a⟩, el cual, en consonía con la definición 6.2, decide si el nodo a insertar devendrá raíz o no:

457a ⟨Genera rn aleatorio entre 0 y n 457a⟩≡ (456f)
const size\_t rn = gsl\_rng\_uniform\_int(r, n + 1);

La inserción en la raíz es realizada por la rutina insert\_root\_xt() explicada en § 4.11.5 (Pág. 340).

random\_insert() implanta la inserción aleatorizada. La versión pública de la inserción para el tipo Gen\_Rand\_Tree<Key> se especifica como sigue:

457b ⟨miembros públicos de Gen\_Rand\_Tree<Key> 456c⟩+≡ (456a) ▷456c 458a>
Node \* insert(Node \* p)
{
 Node \* result = random\_insert(tree\_root, p);
 if (result == Node::NullPtr)
 return NULL;
 return tree\_root = result;
}

Uses random\_insert 456f.

### 6.2.1.2 Eliminación en un ABBA

En la eliminación en un árbol binario de búsqueda, la decisión de escoger la raíz resultante de la unión exclusiva siempre es la raíz del sub-árbol izquierdo (§ 4.11.8 (Pág. 342)). Esto introduce un sesgo en el equilibrio probabilístico en contra de la aleatoriedad. Sortear uniformemente cuál de las raíces -izquierda o derecha- debe reemplazarse no funciona, pues no se pondera la cantidad de nodos que tenga cada rama.

El truco para que la unión exclusiva produzca un árbol aleatorio es sortear, en función de las cardinalidades, cuál de las raíces devendrá la raíz del resultado. Esta idea se plasma en el algoritmo siguiente:

458a   *(miembros públicos de Gen\_Rand\_Tree<Key> 456c) +≡ (456a) ▷ 457b 458c ▷*

```

Node * random_join_exclusive(Node * tl, Node * tr)
{
 if (tl == Node::NullPtr)
 return tr;

 if (tr == Node::NullPtr)
 return tl;

 const size_t & m = COUNT(tl);
 const size_t & n = COUNT(tr);
 <Genera rn aleatorio entre 1 y m + n 458b>
 if (rn <= m)
 { // rama izquierda gana sorteo
 COUNT(tl) += COUNT(tr);
 RLINK(tl) = random_join_exclusive(RLINK(tl), tr);
 return tl;
 }
 else
 {
 COUNT(tr) += COUNT(tl);
 LLINK(tr) = random_join_exclusive(tl, LLINK(tr));
 return tr;
 }
}

Defines:
random_join_exclusive, used in chunk 458c.

458b (Genera rn aleatorio entre 1 y m + n 458b) ≡ (458a)
```

const size\_t rn = 1 + gsl\_rng\_uniform\_int(r, m + n);

Una vez realizada la unión exclusiva aleatorizada, la eliminación aleatoria en árbol binario de búsqueda con rangos es estructuralmente idéntica a la de un ABB:

458c   *(miembros públicos de Gen\_Rand\_Tree<Key> 456c) +≡ (456a) ▷ 458a 459a ▷*

```

Node * random_remove(Node *& root, const Key & key)
{
 if (root == Node::NullPtr)
 return Node::NullPtr;

 Node * ret_val;
 if (Compare() (key, KEY(root)))
```

```

{
 ret_val = random_remove(LLINK(root), key);
 if (ret_val != Node::NullPtr)
 COUNT(root)-;
 return ret_val;
}
else if (Compare() (KEY(root), key))
{
 ret_val = random_remove(RLINK(root), key);
 if (ret_val != Node::NullPtr)
 COUNT(root)-;
 return ret_val;
}
// clave encontrada
ret_val = root;
root = random_join_exclusive(LLINK(root), RLINK(root));
ret_val->reset();

return ret_val;
}

```

Defines:

random\_remove, used in chunk 459a.

Uses random\_join\_exclusive 458a.

La lógica del algoritmo es descender recursivamente hasta el nodo a eliminar. Una vez encontrado este nodo se efectúa una concatenación “aleatoria” entre la rama izquierda y la derecha, la cual es el resultado de la eliminación.

Culminamos esta sección con la estructura de la rutina pública de eliminación:

459a *(miembros públicos de Gen\_Rand\_Tree<Key> 456c) +≡ (456a) ▷458c 459b▷*

```

Node * remove(const Key & key)
{
 Node * ret_val = random_remove(tree_root, key);
 return ret_val != Node::NullPtr ? ret_val : NULL;
}

```

Uses random\_remove 458c.

### 6.2.1.3 Acceso por posición

Una ventaja directa de los árboles aleatorizados es que ellos soportan el acceso por posición estudiado en § 4.11 (Pág. 335). En este sentido, las implantaciones de las operaciones select() y position() son directas:

459b *(miembros públicos de Gen\_Rand\_Tree<Key> 456c) +≡ (456a) ▷459a*

```

Node * select (const size_t & i)
{
 return Aleph::select(tree_root, i);
}
size_t size() const { return COUNT(tree_root); }

Aleph::pair<int,Node*> position (const Key & key)
{
 Aleph::pair<int,Node*> ret_val;

```

```

 ret_val.first =
 Aleph::inorder_position <Node, Compare> (tree_root, key, ret_val.second);
 return ret_val;
}

```

A destacar el valor de retorno `Aleph::pair<int,Node*>` para la rutina `position()`, el cual es un par cuyo primer campo es la posición y segundo el nodo que alberga la clave.

### 6.2.2 Análisis de los árboles aleatorizados

La idea central en este análisis es demostrar que los algoritmos de inserción 6.2.1.1 y de eliminación 6.2.1.2 producen árboles aleatorios en el sentido de la definición 6.2.

**Lema 6.1** (Martínez y Roura 1998 [117]) Sean  $T_<$  y  $T_>$  los árboles binarios de búsqueda producidos por la llamada `split_key_rec_xt(T, x, T_<, T_>)` implantada según el algoritmo explicado en § 4.11.4 (Pág. 339). Si  $T$  es un árbol aleatorio, entonces  $T_<$  y  $T_>$  son árboles aleatorios independientes.

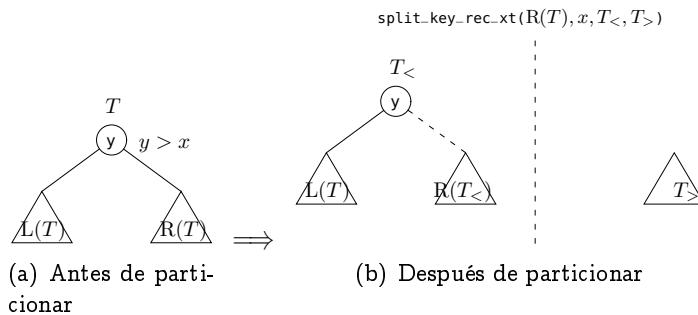


Figura 6.2: `split_key_rec_xt(T, x, T_<, T_>)`

#### Demostración (Por inducción sobre $n = |T|$ )

- $n = 0$ : si  $T = \emptyset$ , entonces `split_key_rec_xt(T, x, T_<, T_>)` produce  $T_< = T_> = \emptyset$ , el cual es por definición un árbol aleatorizado. El lema es, pues, cierto para  $n = 0$ .
- $n > 0$ : ahora asumimos que el lema es cierto para todo  $n$  y verificamos si el lema es cierto para  $n + 1$ .

Sea  $y = \text{KEY(raiz}(T))$ . Si  $x > y$ , entonces  $\text{raiz}(T_<) = \text{raiz}(T)$  y el árbol  $T_>$  y el subárbol derecho  $R(T_<)$  se calculan recursivamente con la llamada `split_key_rec_xt(R(T), x, T_<, T_>)`. Por premisa del lema,  $L(T)$  es aleatorio e independiente, pues  $T$  también lo es. Por la hipótesis inductiva, los árboles  $R(T_<)$  y  $T_>$  son aleatorios e independientes, pues son el resultado de la llamada `split_key_rec_xt(R(T), x, T_<, T_>)`.

Del razonamiento anterior podemos concluir que que  $T_>$  es aleatorio e independiente. ¿Qué acerca de  $T_<$ ? Sabemos que sus ramas izquierda y derecha son aleatorias e independientes. Nos falta entonces demostrar que la ecuación 6.8 se satisface para  $T_<$ . Para eso debemos verificar que  $\forall z \in T_<, P(\text{raiz}(T_<) = z) = \frac{1}{m}$  donde  $m = |T_<|$ . Esto es

equivalente a plantear:

$$\begin{aligned} P(\text{raiz}(T_<)) &= P(\text{raiz}(T) = z \mid \text{raiz}(T) < x) = \frac{P(\text{raiz}(T) = z \cap \text{raiz}(T) < x)}{P(\text{raiz}(T) < x)} \\ &= \frac{1/n}{m/n} = \frac{1}{m} \end{aligned}$$

Simétricamente, el mismo razonamiento se aplica si  $x < y$ , intercambiando los roles de  $T_<$  y  $T_>$ , respectivamente  $\square$

El lema 6.1 es útil porque la partición es una operación interna de la inserción, cuya validez está demostrada por la proposición siguiente.

**Proposición 6.1 (Martínez y Roura 1998 [117])** Sea  $T$  un árbol aleatorio y sea  $p$  un nodo a insertar con clave  $k$ . Entonces, la llamada  $\text{insert}(T, p)$  implantada en § 6.2.1.1 (Pág. 456) produce un árbol aleatorio.

**Demostración (Por inducción sobre  $n = |T|$ )**

- $n = 0$ : si  $T = \emptyset$ , entonces  $\text{insert}(\emptyset, p) = p$ . Los subárboles  $L(p)$  y  $R(p)$  son vacíos, los cuales, por definición, son aleatorios. Sobre  $p$  se verifica que  $P(\text{raiz}(p) = k) = 1 = \frac{1}{|p|}$ . Así pues,  $\text{insert}(\emptyset, p) = p$  es un árbol aleatorio y el lema es cierto para  $n = 0$ .
- $n > 0$ : ahora asumimos que el lema es cierto para todo  $n$  y verificamos si lo es para  $n + 1$ . Sea  $T' = \text{insert}(T, p)$ , sea  $x = \text{KEY}(p)$  e  $y = \text{raiz}(T)$ . Antes de insertar  $x$ ,  $P(\text{raiz}(T) = y) = \frac{1}{n}$ , pues  $T$  es aleatorio.

A partir de estas suposiciones podemos distinguir dos casos después de la inserción: (1) que  $y$  continúe como raíz de  $T'$ , o (2) que  $x$  sea la raíz de  $T'$ :

1. Si  $\text{raiz}(T') = y$ , entonces  $x$  debe perder en el sorteo (*Genere rn aleatorio entre 0 y n* 457a) efectuado por el algoritmo de inserción y el predicado  $\text{rn} == n$  debe ser falso. La probabilidad de que el predicado  $\text{rn} == n$  sea falso es  $\frac{n}{n+1}$  (el complemento probabilístico de  $\frac{1}{n+1}$ , que es la probabilidad de que  $x$  sea raíz de  $T'$ ). Así pues:

$$\forall y \in T, P(\text{raiz}(T') = y) = \underbrace{\frac{1}{n}}_{P(\text{raiz}(T)=y)} \times \underbrace{\frac{n}{n+1}}_{P(\text{raiz}(T')=y)} = \frac{1}{n+1}$$

En este caso,  $x$  será insertado en alguno de los subárboles de  $T$  y, por la hipótesis inductiva, el resultado será un árbol aleatorio.

2. Si  $\text{raiz}(T') = x$ , entonces  $x$  el predicado  $\text{rn} == n$  debe ser cierto; esto sucede con probabilidad  $P(\text{raiz}(T') = x) = \frac{1}{n+1}$ , que es lo esperado según el algoritmo. Por otra parte,  $T' = \langle T_<, x, T_> \rangle$ , donde  $T_<$  y  $T_>$  son los árboles resultantes de  $\text{random\_join}(T, x, T_<, T_>)$  los cuales son, por el lema 6.1, aleatorios ■

Del resultado anterior podemos establecer el corolario siguiente.

**Corolario 6.1** Sea  $K = \{k_1, k_2, \dots, k_n\}$  un conjunto de claves cualquiera. Sea  $T$  un árbol aleatorizado. Entonces, cualquier permutación de inserción de  $K$  sobre  $T$ , según el algoritmo desarrollado en § 6.2.1.1 (Pág. 456), produce un árbol aleatorizado.

El corolario conlleva implicaciones importantes. En primer lugar, cualquiera que sea el orden de inserción, siempre obtendremos un árbol aleatorizado. En otras palabras, el árbol resultante es equivalente a un ABB construido a partir de una secuencia de inserción aleatoria. Consecuentemente, todos los resultados conocidos para un ABB son aplicables; el más importante de ellos es la proposición 4.12 que nos promedia la altura del árbol a  $\mathcal{O}(\lg n)$ .

En cuanto a la eliminación, el primer paso para analizarla es estudiar la unión exclusiva aleatoria, la cual se analiza en el lema siguiente.

**Lema 6.2 (Martínez y Roura 1998 [117])** Sean  $T_<$  y  $T_>$  dos árboles aleatorios independientes tal que  $\forall k_l \in T_<, \forall k_r \in T_>, k_l < k_r$ , es decir, todas las claves de  $T_<$  son estrictamente menores que las claves de  $T_>$ . Entonces,  $T' = \text{random\_join\_exclusive}(T_<, T_>)$  es un árbol aleatorio.

**Demostración (por inducción sobre  $m = |T_<|$  y  $n = |T_>|$ )**

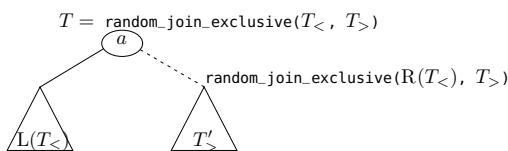
- $m = 0$  o  $n = 0$ : bajo este predicado, tenemos dos casos:

1. Si  $m = 0$  y  $n = 0$ , entonces  $\text{random\_join\_exclusive}(T_<, T_>) = \emptyset$ , el cual es, por definición, aleatorio.
2. Si  $m = 0$  o  $n = 0$ , entonces  $\text{random\_join\_exclusive}(T_<, T_>)$  retorna el árbol no vacío del par, el cual es, por premisa del lema, aleatorio. El lema es, pues, cierto para  $m = 0$  o  $n = 0$ .

- $m > 0$  y  $n > 0$ : sea

- $a = \text{KEY}(\text{raiz}(T_>))$ ,
- $b = \text{KEY}(\text{raiz}(T_>))$  y
- $T' = \text{random\_join\_exclusive}(T_<, T_>)$

Asumamos que  $a$  gana el sorteo *Genere rn aleatorio entre 1 y m + n* efectuado en el `random_join_exclusive()` desarrollado en § 6.2.1.2 (Pág. 458), o sea, que el predicado `rn <= m` es cierto. Esto implica que  $a$  será la raíz de  $T'$ . Podemos bosquejar el árbol resultante del siguiente modo:



Por la estructura del algoritmo y el resultado del sorteo, la rama izquierda de  $T'$  es  $L(T)$ , mientras que la derecha es la llamada recursiva `random_join_exclusive(R(T<), T>)`. Ahora bien,  $L(T)$  es un árbol aleatorio pues, por premisa del lema,  $T$  también lo es. Igualmente, por la hipótesis inductiva,  $T'$  también es aleatorio.  $L(T)$  y  $T'$  son independientes, pues  $L(T)$  y  $R(T)$  son, por premisa del lema, independientes y  $T' = \text{random\_join\_exclusive}(R(T<), T_>)$  es, por la hipótesis inductiva, independiente. Habiendo demostrado que las ramas de  $T'$  son aleatorias e independientes, sólo nos falta demostrar que  $\forall x \in T_<, P(\text{raiz}(T') = x) = \frac{1}{m+n}$ . Esto es:

$$\begin{aligned}
 P(\text{raiz}(T') = x) &= P(\text{raiz}(L(T)) = x) \times P(\text{rn} \leq M \text{ sea cierto}) \\
 &= \frac{1}{m} \times \frac{m}{m+n} = \frac{1}{m+n} \quad \square
 \end{aligned}$$

Con este lema estamos en capacidad de estudiar la eliminación, la cual es analizada en la proposición siguiente.

**Proposición 6.2** (Martínez y Roura 1998 [117]) Sea  $T$  un árbol aleatorio y  $x$  una clave contenida en  $T$ . Sea la llamada a  $\text{remove}(T, x)$  y sea  $T'$  el árbol aleatorio resultante después de la llamada a  $\text{remove}()$ . Entonces,  $T'$  es un árbol aleatorio.

**Demostración** (por inducción sobre  $n = |T|$ )

- $n = 1$ : entonces  $T' = \emptyset$ , el cual es, por definición, aleatorio.
- $n > 1$ : entonces asumimos que el lema es cierto para todo  $|T| < n$  y verificamos si también es cierto para  $n$ . Aquí podemos separar dos casos:
  1. Si  $\text{raiz}(T) \neq x$ , entonces  $x$  es eliminado de uno de los subárboles de  $T$  y, por la hipótesis inductiva, esta eliminación arroja un árbol aleatorio.
  2. Si  $\text{raiz}(T) = x$ , entonces  $\text{remove}(T, x)$  efectúa la llamada  $\text{random\_join}(L(T), R(T))$  la cual, por el lema 6.2, es un árbol aleatorio. Podemos afirmar entonces que  $L(T')$  y  $R(T')$  son árboles aleatorios independientes. Nos falta por comprobar que  $\forall y \in T', P(\text{raiz}(T') = y) = \frac{1}{n-1}$ , lo cual se verifica como sigue:

$$\begin{aligned}
 P(\text{raiz}(T') = y) &= P(\text{raiz}(T') = y \mid \text{raiz}(T) = x) \times P(\text{raiz}(T) = x) + \\
 &\quad P(\text{raiz}(T') = y \mid \text{raiz}(T) \neq x) \times P(\text{raiz}(T) \neq x) \\
 &= P(\text{raiz}(\text{random\_join}(L(T), R(T))) = y) \times \frac{1}{n} + \\
 &\quad P(\text{raiz}(T') = y \mid \text{raiz}(T) \neq x) \times \frac{n-1}{n} \\
 &= \frac{1}{n-1} \times \frac{1}{n} + \frac{1}{n-1} \times \frac{n-1}{n} = \frac{1}{n-1} \quad \blacksquare
 \end{aligned}$$

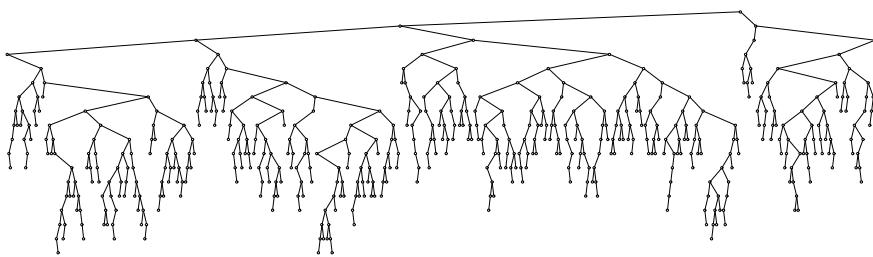


Figura 6.3: Un árbol aleatorizado de 512 nodos

De las proposiciones mostradas podemos enunciar el siguiente corolario.

**Corolario 6.2** Si  $T$  es un árbol aleatorizado cualquiera, entonces cualquier secuencia de inserciones y eliminaciones produce un árbol aleatorio.

Según el corolario anterior, cualquiera que sea la secuencia de inserción, con eliminaciones intercaladas arbitrarias, el árbol resultante siempre es aleatorio. Podemos, pues, concluir que en promedio la altura del árbol aleatorizado es  $\mathcal{O}(\lg n)$ . Puesto que el tiempo de las operaciones de inserción y eliminación depende de la altura, podemos concluir también que en promedio la inserción y la eliminación son  $\mathcal{O}(\lg n)$ .

Los árboles aleatorizados tienen dos costes adicionales respecto a los binarios clásicos. En primer lugar, la estructura de nodo de un ABBA requiere espacio adicional para almacenar la cardinalidad. En segundo lugar, un ABBA requiere operaciones adicionales respecto a un ABB. Durante la inserción, es necesario efectuar una partición y actualizar las cardinalidades.

A pesar de las restricciones anteriores, un ABBA tiene la ventaja de eliminar el peligro representado por el sesgo en la secuencia de inserción y las inserciones intercaladas. Por añadidura, el contador presente en cada nodo facilita perfecta y “naturalmente” el acceso por posición, con todas sus operaciones asociadas, tal como se explicó en § 4.11 (Pág. 335). Esto último es quizá la gran bondad de este esquema de equilibrio.

### 6.3 Treaps

Ahora estudiaremos otro enfoque de aleatorización radicalmente diferente a un ABBA: la estructura treap.

**Definición 6.3 (Treap)** Sea  $T$  un árbol binario en el cual cada nodo  $n_i$  tiene dos campos a saber:

1.  $\text{KEY}(n_i) \in K$  es la clave de búsqueda, donde  $K$  es un conjunto ordenable cualquiera.
2.  $\text{PRIO}(n_i) \in P$  es la prioridad del nodo, donde  $P$  es un conjunto ordenable cualquiera.

Entonces,  $T$  es un treap si y sólo si:

- $T \in \text{ABB}$  según las claves  $K$ .
- $\forall n_i \in T, \text{PRIO}(n_i) \leq \text{PRIO}(L(n_i))$  y  $\text{PRIO}(n_i) \leq \text{PRIO}(R(n_i))$ .

La segunda propiedad se llama “relación del ancestro”. Si las prioridades son únicas, es decir, si éstas no se repiten, entonces el treap es denominado “puro”.

La primera propiedad corresponde a la de orden de un ABB; la segunda a la de orden de un heap. De allí, pues, el nombre “treap”: tree y heap.

Podemos caracterizar un treap como una secuencia de pares (clave, prioridad) correspondiente a los contenidos de cada nodo. Por ejemplo, el treap de la figura 6.4 puede caracterizarse como:  $(6, 792), (8, 538), (10, 602), (11, 478), (13, 704), (17, 844), (20, 807), (21, 663), (22, 459), (23, 652), (24, 622), (26, 600), (35, 499), (37, 536), (38, 434), (43, 666), (46, 656), (53, 761), (54, 542), (59, 696), (60, 642), (61, 803), (66, 852), (69, 526), (76, 538), (80, 609), (81, 655), (96, 533), (97, 440), (100, 435)$ .

Un treap tiene una propiedad muy interesante caracterizada por el siguiente lema.

**Lema 6.3 (Lema de la unicidad del treap -Seidel - Aragon 1996 [158])** Sea  $K = \{k_1, k_2, \dots, k_n\}$  un conjunto de claves y  $P = \{p_1, p_2, \dots, p_n\}$  un conjunto de prioridades. Entonces, el conjunto  $\{(k_1, p_1), (k_2, p_2), \dots, (k_n, p_n)\} \subset K \times P$  tiene un único treap.

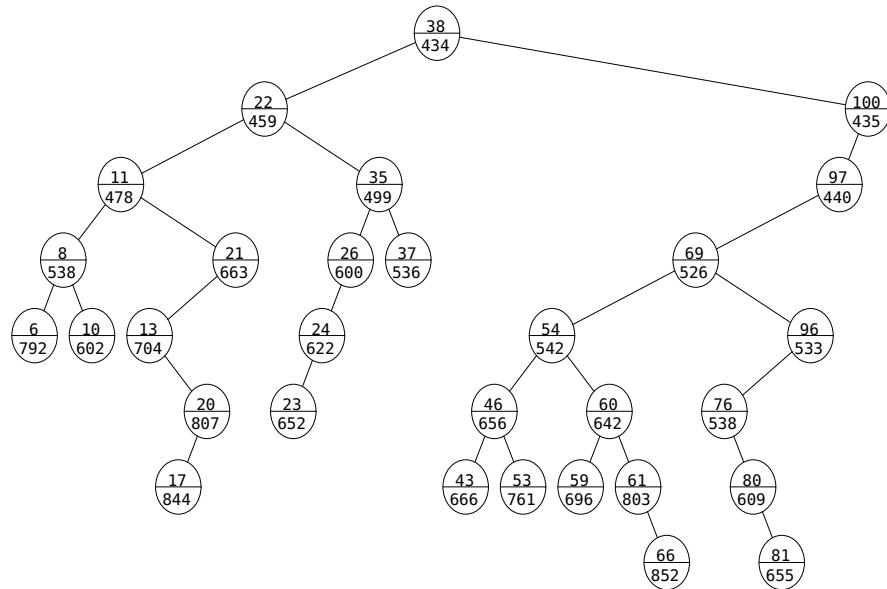


Figura 6.4: Ejemplo de treap. Campos superiores representan las claves; inferiores prioridades

#### Demostración (Por inducción sobre $n$ )

- $n = 1$ : En este caso existe un único treap conformado por un par único. El lema es cierto para  $n = 1$ .
- $n > 1$ : Ahora asumimos que el lema es cierto para todo  $n$  y verificamos su veracidad para  $n + 1$ . Puesto que las prioridades son únicas, el treap de  $n + 1$  nodos sólo puede tener una raíz cuya prioridad es la menor de todas. Sea  $(k_i, p_i)$  el nodo raíz del treap. Una vez determinada la raíz del treap y a causa de la propiedad de orden de un ABB, el conjunto de nodos de las ramas izquierda y derecha queda determinado en  $K_{<} = \{(k_1, p_1), \dots, (k_{i-1}, p_{i-1})\}$  y  $K_{>} = \{(k_{i+1}, p_{i+1}), \dots, (k_n, p_n)\}$ , respectivamente. Por la hipótesis inductiva,  $K_{<}$  y  $K_{>}$ , cuyas cardinalidades son inferiores a  $n + 1$ , tienen treaps únicos. El lema es cierto para todo  $n$   $\square$

¿En qué consiste la aleatoriedad de un treap? La respuesta se encuentra en la selección de la prioridad. Cuando se crea un nuevo nodo, se selecciona la prioridad aleatoriamente, se inserta el nodo según el algoritmo clásico de inserción en ABB y las violaciones eventuales a la relación del ancestro se corrigen mediante rotaciones.

##### 6.3.1 El TAD Treap<Key>

El primer paso en el diseño de un TAD que nos modelice un treap es determinar la estructura del nodo que lo conforma. Hay dos maneras generales de implantar un treap: recursiva o iterativamente. Ambos enfoques son muy sencillos y comparten la estructura del nodo. Por esa razón mantendremos la estructura del nodo en un archivo separado denominado `<treapNode.H 465>`, el cual se estructura como sigue:

```
<treapNode.H 465>≡
const long Max_Priority = ULONG_MAX;
```

```

const long Min_Priority = 0;

class TreapNode_Data
{
 unsigned long priority;
 TreapNode_Data () : priority (Max_Priority) {}
 unsigned long & getPriority () { return priority; }
};

DECLARE_BINNODE_SENTINEL(TreapNode, 80, TreapNode_Data);
#define PRIO(node) ((node) -> getPriority ())

Uses DECLARE_BINNODE_SENTINEL.

```

Aquí se define el nodo binario `TreapNode<Key>` que almacena una prioridad accedida mediante el método `getPriority()` o el macro `PRIO`. El archivo exporta dos constantes `Min_Priority` y `Max_Priority` que corresponden a los valores mínimo y máximo que puede tener una prioridad.

Un `TreapNode<Key>` tiene un valor de `Node::NullPtr` apuntado a un centinela con prioridad máxima. Este nodo funge de centinela para detener rotaciones descendentes hacia hojas del árbol.

Nuestra implantación del treap requiere de un nodo cabecera padre de la raíz. Tal nodo también funge de centinela para detener rotaciones ascendentes hacia la raíz.

Ahora procedemos a desarrollar el TAD `Treap<Key>`:

466a *(tpl\_treap.H 466a)*≡  
`template <template <typename> class NodeType, typename Key, class Compare>`  
`class Gen_Treap`  
`{`  
*(Miembros de Gen\_Treap<Key> 466b)*  
`};`  
*(Clases públicas de Gen\_Treap<Key> 467a)*

Antes de implantar los constructores es necesario definir los atributos internos de `Gen_Treap<Key>`:

466b *(Miembros de Gen\_Treap<Key> 466b)*≡ (466a) 466c▷  
`Node head;`  
`Node * head_ptr;`  
`Node *& tree_root;`

`head` es un nodo cabecera, padre centinela de la raíz, `head_ptr` es un apuntador al nodo cabecera y `tree_root` es una referencia apuntador a la raíz del treap.

Un treap requiere una función que genere números aleatorios correspondientes a las prioridades. Para eso nos valemos de la biblioteca `gsl`<sup>2</sup>. Usaremos el generador “tornado” (“Twisted”), actualmente conocido como el mejor generador de números seudo-aleatorios [119, 118, 120]:

466c *(Miembros de Gen\_Treap<Key> 466b) +≡* (466a) ▷466b 468a▷  
`gsl_rng * r;`

La clase `Gen_Treap<Key>` no es para uso directo del cliente, pues su rol es implantar un treap genérico que sea independiente de que sus nodos sean virtuales o no. El cliente

---

<sup>2</sup>Gnu Scientific Library [66].

debe usar alguna de las siguientes clases:

467a *(Clases públicas de Gen\_Treap<Key> 467a)≡* (466a)  
`template <typename Key, class Compare = Aleph::less<Key> >  
 class Treap : public Gen_Treap<TreapNode, Key, Compare> {};`

### 6.3.2 Inserción en un treap

Conceptualmente, la inserción de un nodo p puede caracterizarse en dos partes:

1. Efectuar una inserción como en ABB, es decir, substituir el nodo externo por p.
2. Rotar p, de forma que éste suba de nivel, hasta que su prioridad no viole la relación ancestral.

Estamos preparados para implantar el algoritmo de inserción:

467b *(Inserción en Gen\_Treap<Key> 467b)≡*  
`static Node * insert(Node * root, Node * p)  
{  
 if (root == Node::NullPtr)  
 return p;  
  
 Node * insertion_result = NULL;  
 if (Compare() (KEY(p), KEY(root)))  
 {  
 insertion_result = insert(LLINK(root), p);  
 if (insertion_result == Node::NullPtr)  
 return Node::NullPtr;  
  
 LLINK(root) = insertion_result;  
 if (PRI0(insertion_result) < PRI0(root))  
 return rotate_to_right(root);  
 else  
 return root;  
 }  
 else if (Compare() (KEY(root), KEY(p)))  
 {  
 insertion_result = insert(RLINK(root), p);  
 if (insertion_result == Node::NullPtr)  
 return Node::NullPtr;  
  
 RLINK(root) = insertion_result;  
 if (PRI0(insertion_result) < PRI0(root))  
 return rotate_to_left(root);  
 else  
 return root;  
 }  
 return Node::NullPtr;  
}`

`insert()` efectúa una búsqueda recursiva de la clave contenida en el nodo p. Si se encuentra un nodo con tal clave, entonces se retorna `Node::NullPtr`.

Si, por el contrario, no se encuentra la clave, entonces la recursión se detiene en el nodo externo donde inicialmente se inserta el nodo. La secuencia recursiva de retornos verifica si hay una violación de la relación ancestral, la cual es eventualmente corregida con una rotación.

La versión anterior de `insert()` es un miembro privado, estático, cuyo rol es hacer la inserción y corregir las violaciones de la relación ancestral. La selección aleatoria de la prioridad y la invocación inicial al método recursivo son efectuados por la versión pública de `insert()` cuya definición es como sigue:

468a *(Miembros de Gen\_Treap<Key> 466b) +≡* (466a) ◁ 466c 468b ▷  
 Node \* insert(Node \* p)  
 {  
 PRIO(p) = gsl\_rng\_get(r); // selección aleatoria de prioridad  
 Node \* result = insert(tree\_root, p);  
 if (result == Node::NullPtr)  
 return NULL;  
 tree\_root = result;  
 return p;  
}

### 6.3.3 Eliminación en treap

La eliminación es conceptualmente más simple que la clásica en un ABB. Primero se busca el nodo con la clave de interés. Una vez ubicado, el nodo se rota de manera que su hijo de menor prioridad suba de nivel. El proceso continua hasta que el nodo a eliminar devenga hoja; en ese momento el nodo se substituye por `Node::NullPtr`.

Sorprendentemente, una versión iterativa de la eliminación es fácilmente realizable, la cual se estructura en dos partes que se presentan como sigue:

468b *(Miembros de Gen\_Treap<Key> 466b) +≡* (466a) ◁ 468a  
 Node \* remove(const Key & key)  
{  
*(Buscar nodo a eliminar p 468c)*  
if (p == Node::NullPtr)  
return NULL; // clave no fue encontrada  
*(Rotar p hasta que devenga hoja 469a)*  
p->reset();  
return p;  
}

El primer bloque *(Buscar nodo a eliminar p 468c)* se encarga de ubicar el nodo a suprimir `p` y se define como sigue:

468c *(Buscar nodo a eliminar p 468c) ≡* (468b)  
Node \*\* pp = &RLINK(head\_ptr);  
Node \* p = tree\_root;  
while (p != Node::NullPtr)

```

if (Compare() (key, KEY(p)))
{
 pp = &LLINK(p);
 p = LLINK(p);
}
else if (Compare() (KEY(p), key))
{
 pp = &RLINK(p);
 p = RLINK(p);
}
else
 break;

```

Este bloque declara dos variables. La variable *p* guarda la dirección del nodo a suprimir. Puesto que en la segunda fase el nodo *p* será rotado hasta que devenga hoja, es necesario en principio guardar la dirección del nodo padre de tal que manera éste se actualice en cada rotación. Como el algoritmo es iterativo, es preferible guardar la dirección de la celda dentro del nodo padre que apunta a *p*; tal apuntador es mantiene en la variable *pp*.

El segundo bloque *<Rotar p hasta que devenga hoja 469a>* se especifica del siguiente modo:

469a *<Rotar p hasta que devenga hoja 469a>*≡ (468b)

```

while (<p no sea hoja 469b>)
 if (PRI0(LLINK(p)) < PRI0(RLINK(p)))
 {
 *pp = rotate_to_right(p);
 pp = &RLINK(*pp);
 }
 else
 {
 *pp = rotate_to_left(p);
 pp = &LLINK(*pp);
 }
*pp = Node::NullPtr;

```

La última línea es la que finalmente elimina el nodo *p* al asignarle a la celda del nodo que apunta a *p* el nodo externo *Node::NullPtr*.

El predicado *<p no sea hoja 469b>* consiste en verificar si los subárboles de *p* son externos. Esto se lleva a cabo de la siguiente manera:

469b *<p no sea hoja 469b>*≡ (469a)

```

not (LLINK(p) == Node::NullPtr and RLINK(p) == Node::NullPtr)

```

Es el momento de hacer una observación interesante: la eliminación de un nodo ocurre exactamente en el mismo sitio donde éste se hubiese insertado. Recordemos que la inserción comienza por poner el nuevo nodo como hoja. Si decidimos eliminar una clave y luego insertarla, la hoja del nodo de inserción es exactamente la misma que cuando fue eliminada. Un ejemplo de esta situación se ilustra con el nodo 38 de la figura 6.5.

Intuitivamente, este fenómeno es evidenciado por el lema de la unicidad del treap (lema 6.3). Para que el treap sea único a través de todas las modificaciones, la inserción y eliminación siempre deben ver el mismo treap.

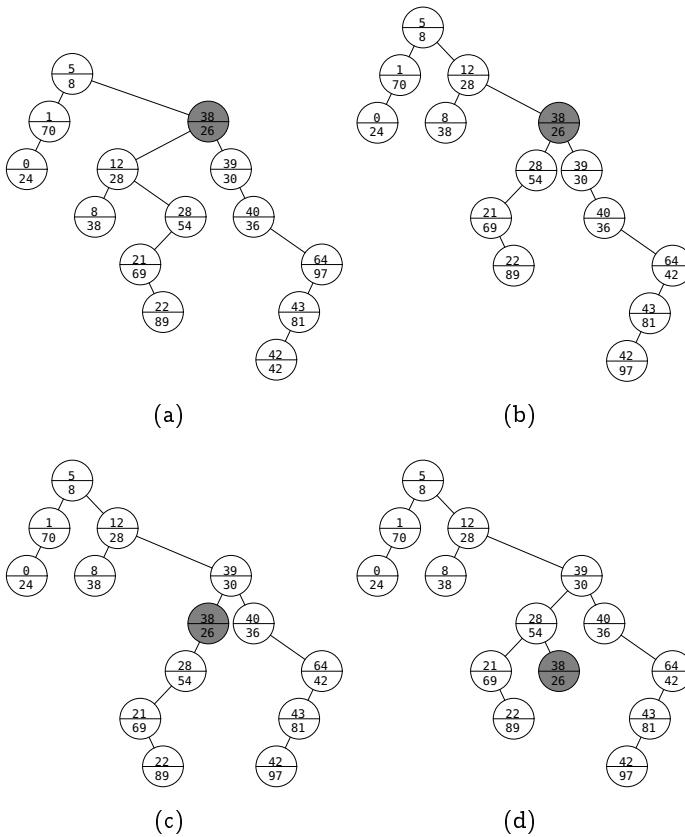


Figura 6.5: Ejemplo de eliminación de la clave 38 en un treap. La posición final del 38 antes de suprimirlo es la misma que la posición inicial de inserción

#### 6.3.4 Análisis de los treaps

En análisis de los treaps requiere establecer una propiedad muy asociada a la aleatoriedad: la propiedad de ausencia de memoria en las decisiones aleatorias. Tal propiedad está dada por la siguiente proposición:

**Proposición 6.3** Un treap  $T$  es un árbol binario aleatorio.

En otras palabras, un treap es equivalente a un ABB construido a partir de una secuencia de inserción aleatoria.

**Demostración** Puesto que las prioridades son seleccionadas aleatoria e independientemente, podemos asumir que todas las prioridades son escogidas antes de que se inicie la secuencia de inserción.

Sea  $K = \{k_1, k_2, \dots, k_n\}$  una secuencia de inserción cualquiera y sea  $P = \{p_1, p_2, \dots, p_n\}$  un conjunto de prioridades seleccionadas aleatoria e independientemente que son asignadas a las claves  $K$  tal que cada par  $(k_i, p_i)$  compondrá un nodo del treap.

Por el lema 6.3 (de la unicidad), sabemos que dadas las claves y sus prioridades el treap es único. Esto implica que el orden de inserción no afecta el treap resultante. De este modo, sin pérdida de generalidad, podemos asumir una secuencia de inserción  $S_{ABB}$  que va desde la menor hasta la mayor prioridad de la cual podemos realizar las siguientes observaciones:

1. Respecto al conjunto de claves  $K$ , la probabilidad de que  $S_{ABB}$  sea el valor de una permutación específica es  $\frac{1}{|K|!}$  pues el orden de la permutación está dado por el orden de las prioridades que son asignadas aleatoriamente a cada clave. Por tanto, la secuencia  $S_{ABB}$  ordenada por prioridad es aleatoria, pues las prioridades fueron escogidas aleatoria e independientemente.
2.  $S_{ABB}$  no causa rotaciones en el treap, pues cada nodo es insertado como hoja y, puesto que fue insertado por prioridad ascendente, la relación ancestral jamás es violada. De lo anterior podemos concluir que la inserción de  $S_{ABB}$  ocurre de la misma manera que en un ABB.

Puesto que  $S_{ABB}$  es una secuencia aleatoria y que la inserción ocurre como en un ABB, podemos concluir que un treap es equivalente a un ABB construido a partir de una secuencia de inserción aleatoria ■

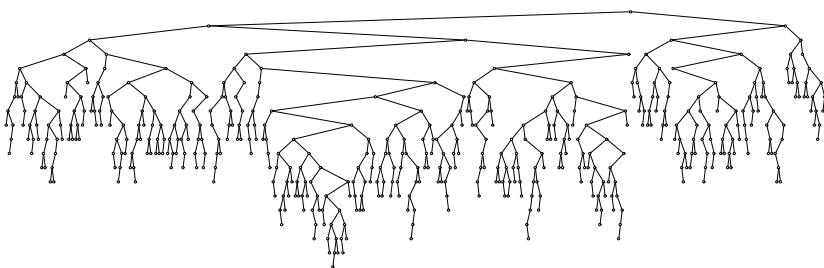


Figura 6.6: Un treap de 512 nodos

La proposición anterior nos garantiza que el tiempo esperado para la búsqueda en un treap es  $\mathcal{O}(\lg n)$ . ¿Qué acerca de las inserciones y supresiones? Observemos que ambas operaciones están dominadas por la altura del árbol y que éstas resultan en treaps. La cantidad máxima de rotaciones también es función de la altura. De esta manera, podemos concluir que las operaciones de inserción, búsqueda y eliminación son  $\mathcal{O}(\lg n)$ .

### 6.3.5 Prioridades implícitas

Los treaps son árboles aleatorios. Consecuentemente, el desempeño de la búsqueda es equivalente a los árboles aleatorizados. En cuanto al espacio, los costes también son equivalentes, pues ambos árboles requieren almacenar un entero adicional por nodo: en los árboles aleatorizados es la cardinalidad, en los treaps, la prioridad.

Sorprendentemente, en un treap podemos obviar la prioridad si la substituimos por una función hash que transforme la clave a un valor de prioridad. De esta manera, la prioridad no necesita almacenarse, pues puede calcularse implícitamente cada vez que se requiera. El treap ocuparía entonces un espacio equivalente al de un ABB.

Para que este enfoque sea eficaz, es necesaria una buena función hash. En este sentido, toda la teoría de funciones hash impartida en § 5.2 (Pág. 423) es aplicable.

## 6.4 Árboles AVL

Un árbol AVL, o un AAVL, es una clase especial de árbol binario de búsqueda definido como sigue.

**Definición 6.4 (Árbol AVL)** Sea  $T$  un árbol binario de búsqueda. Se dice que  $T$  es AVL si y sólo si:

$$\forall n_i \in T, \quad -1 \leq h(R(n_i)) - h(L(n_i)) \leq 1 \quad (6.9)$$

Es decir, para todo nodo, la diferencia de altura entre la rama izquierda y derecha no debe exceder de la unidad.

Para futuros discursos, la diferencia de altura de un nodo  $n_i$  se llama  $\delta(n_i) = h(R(n_i)) - h(L(n_i))$ . En contextos de código,  $\delta(n_i)$  se denota como  $\text{DIFF}(n)$ .

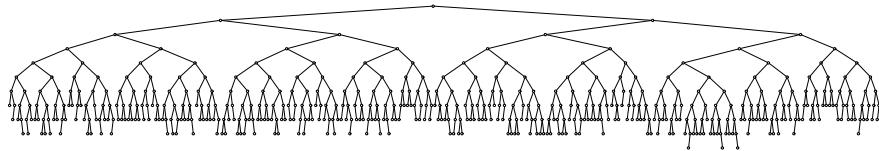


Figura 6.7: Un árbol AVL de 512 nodos

La condición (6.9) se llama condición AVL y es menos fuerte que el equilibrio fuerte de Wirth definido en § 6.1 (Pág. 452). La idea es perder un poco de equilibrio en aras de algoritmos más eficientes que garanticen la condición AVL. Más adelante demostraremos que la altura de un AAVL está logarítmicamente acotada. La ganancia estriba entonces en lograr algoritmos de modificación logarítmicos. Esto es factible si almacenamos en cada nodo información sobre el equilibrio.

Se conocen dos formas de implantar un AAVL. La primera, que es la más popular y eficiente, pero también la más difícil, consiste en almacenar en cada nodo la diferencia de alturas. Para ello sólo se requieren dos bits, los cuales, bajo los requerimientos de alineación de palabra impuestos por las arquitecturas modernas, deben extenderse hasta la longitud de la palabra de la arquitectura.

La segunda implantación consiste en almacenar en cada nodo su altura. Puesto que el árbol es logarítmicamente acotado, un byte es suficiente. Esta alternativa conduce a algoritmos más sencillos, pero menos eficiente, pues cuando se modifica el árbol se requieren actualizar más nodos con su altura.

En este texto adoptaremos el primer estilo de implantación; es decir, almacenaremos la diferencia de alturas en cada nodo. La estructura de un nodo AVL se define en el archivo `avlNode.H 472`, el cual tiene la estructura siguiente:

472 `<avlNode.H 472>≡`

```

class AvlNode_Data
{
 signed char diff; // diferencia de altura
 signed char & getDiff() { return diff; }
};

DECLARE_BINNODE(AvlNode, 40, AvlNode_Data);
#define DIFF(p) ((p)->getDiff())

```

Uses AvlNode and DECLARE\_BINNODE.

#### 6.4.1 El TAD Avl\_Tree<Key>

El TAD `Gen_Avl_Tree<Key>` define una clase genérica de árbol AVL independiente de que el nodo sea virtual o no. La clase en cuestión se define en el archivo `<tpl_avl.H 473a>`,

que se define a continuación:

473a *(tpl\_avl.H 473a)≡*

```
template <template <typename> class NodeType, typename Key, class Compare>
class Gen_Avl_Tree
{
 typedef NodeType<Key> Node;
<miembros privados de Gen_Avl_Tree<Key> 473b>
<miembros públicos de Gen_Avl_Tree<Key> 474a>
};

template <typename Key, class Compare = Aleph::less<Key> >
class Avl_Tree : public Gen_Avl_Tree<AvlNode, Key, Compare> {};
```

Uses AvlNode.

Aunque por lo general los algoritmos sobre árboles son naturalmente expresables en forma recursiva, los árboles AVL no encajan en esta categoría. Las implantaciones recursivas que se conocen están basadas en almacenar en cada nodo su altura y actualizarla recursivamente en cada modificación. En nuestro caso, en aras de lograr una implantación más eficiente que la recursiva, utilizaremos una pila, la cual se especifica como sigue:

473b *(miembros privados de Gen\_Avl\_Tree<Key> 473b)≡* (473a) 473c ▷

```
FixedStack<Node *, Node::MaxHeight> avl_stack;
```

Uses FixedStack 101a.

avl\_stack siempre guardará el camino de búsqueda desde la raíz hasta el nodo insertado o eliminado. Puesto que la altura de un AAVL está acotada, la pila no requiere verificaciones de desborde.

473c *(miembros privados de Gen\_Avl\_Tree<Key> 473b)+≡* (473a) ▲ 473b 473d ▷

```
Node head_node;
Node * head_ptr;
Node *& root;
Compare cmp;
```

head\_node es un nodo auxiliar que será padre del nodo raíz. head\_ptr es un apuntador al nodo auxiliar. El uso del nodo cabecera es muy útil para generalizar las rotaciones. Si no usásemos esta cabecera, entonces la rotación de la raíz debería tratarse separadamente.

head\_ptr siempre estará insertado en la pila. A efectos algorítmicos, se considerará que la pila está vacía cuando ésta sólo contenga el nodo cabecera. Planteamos, entonces, las siguientes funciones:

473d *(miembros privados de Gen\_Avl\_Tree<Key> 473b)+≡* (473a) ▲ 473c 474b ▷

```
bool avl_stack_empty() { return avl_stack.top() == head_ptr; }

void clean_avl_stack() { avl_stack.popn(avl_stack.size() - 1); }
```

avl\_stack\_empty() retorna cierto si avl\_stack está lógicamente vacía. clean\_avl\_stack() limpia la pila; es decir, extrae todos sus elementos.

Por convención, la raíz root será la hija derecha del nodo cabecera head\_ptr. Notemos que root es una referencia a RLINK (head\_ptr) y no un puntero. Este ardid nos permite trabajar directamente con RLINK (head\_ptr) mediante el nombre root.

#### 6.4.1.1 Inserción en un árbol AVL

Consideremos un AAVL en el cual se realiza una inserción clásica en un ABB; es decir, el nodo se inserta como hoja en el lugar ocupado por el nodo externo resultado de la búsqueda. Tal inserción la realiza el algoritmo siguiente:

474a *(miembros públicos de Gen\_Avl\_Tree<Key> 474a)≡* (473a) 479a▷

```

Node * insert(Node * p)
{
 if (root == Node::NullPtr)
 return root = p;

 Node *pp = search_and_stack_avl(KEY(p));
 if (cmp (KEY(p), KEY(pp)))
 LLINK (pp) = p;
 else if (cmp (KEY(pp), KEY(p)))
 RLINK(pp) = p;
 else
 { // clave duplicada
 clean_avl_stack();
 return NULL;
 }
 restore_avl_after_insertion(p);

 return p;
}

```

La función `search_and_stack_avl()` busca la clave contenida en `p` y a la vez guarda todo el camino de búsqueda en la pila `avl_stack`:

474b *(miembros privados de Gen\_Avl\_Tree<Key> 473b)≡* (473a) ▷473d 476a▷

```

Node * search_and_stack_avl(const Key & key)
{
 Node * p = root;
 do // desciende en búsqueda de key y empila camino de búsqueda
 {
 avl_stack.push(p);
 if (cmp(key, KEY(p))) // ¿key < KEY(p)?
 p = LLINK(p);
 else if (cmp(KEY(p), key)) // ¿key > KEY(p)?
 p = RLINK(p);
 else
 return p; // clave duplicada
 }
 while (p != Node::NullPtr);

 return avl_stack.top();
}

```

Si `KEY(p)` ya se encuentra en el árbol, entonces `search_and_stack_avl()` retorna el nodo que contiene la clave. La duplicación se detecta en el primer `if`. En caso contrario, `pp` es el padre del nodo a insertar `p`. El nodo `p` se inserta físicamente y se actualiza el factor de balance de `pp`.

#### Restauración de la condición AVL

Después de la inserción física se verifica si ocurre una violación de la condición AVL, la cual se caracteriza en dos casos generales. El primero de ellos se ilustra en la figura 6.8(a). Inicialmente, el nodo  $y$  se encuentra perfectamente equilibrado, mientras que su padre  $x$  está ligeramente desequilibrado hacia la derecha. Ocurre entonces la inserción de un valor mayor que  $y$ , el cual hace que la rama  $\chi$  aumente de altura. El nodo  $x$  sufre un desequilibrio que viola la condición AVL.

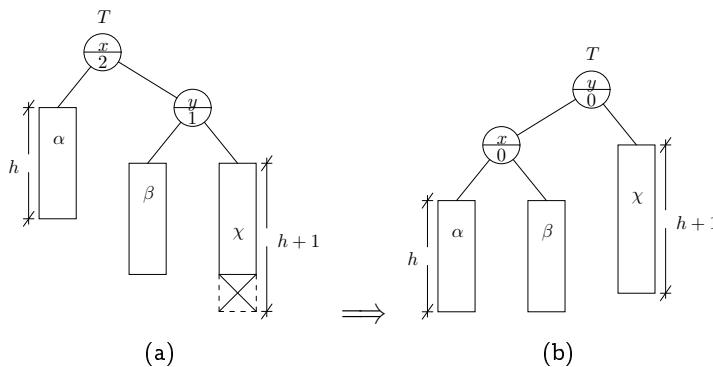


Figura 6.8: Primer caso de inserción en un árbol AVL

La violación descrita en la figura 6.8(a) se corrige mediante una rotación hacia la izquierda del nodo  $y$ , cuyo resultado general se ilustra en la figura 6.8(b). Notemos que en las dos figuras -6.8(a) y 6.8(b)- el valor de altura global  $h + 2$  permanece intacto. Por lo tanto, si la ascendencia de  $T$  es AVL antes de la inserción, también lo es después. En otras palabras, para el caso de desequilibrio de la figura 6.8(a), la corrección resultante después de la rotación produce un árbol enteramente AVL.

El desequilibrio ilustrado en la figura 6.8(a) tiene un caso simétrico que ocurre cuando la inserción se produce por la izquierda. Gráficamente, el desequilibrio y su corrección se ven igual que las figuras 6.8(a) y 6.8(b) reflejadas en un espejo.

Los valores de los factores permiten identificar la situación de desequilibrio mostrada en la figura 6.8(a). Dado un nodo  $x$  con un desequilibrio podemos caracterizar las siguientes situaciones y sus correspondientes acciones:

1. Si  $\text{DIFF}(x) == 2 \text{ and } \text{DIFF}(y) == 1 \Rightarrow$  se efectúa una rotación simple hacia la izquierda.
2. Si  $\text{DIFF}(x) == -2 \text{ and } \text{DIFF}(y) == -1 \Rightarrow$  se efectúa una rotación simple hacia la derecha.

Los métodos `rotateLeft()` y `rotateRight()` rotan el nodo  $p$  hacia la izquierda y derecha respectivamente. Los métodos deben actualizar los factores de equilibrio de los nodos involucrados. Al respecto existe un caso que sólo ocurre durante la eliminación, evaluado en el if y que explicaremos posteriormente. De lo contrario, los factores de balance se ponen en cero. Es posible deducir los cambios en los factores y ajustarlos a través de operaciones aritméticas, empero se requieren varios if para hacerlo. Por lo tanto, es más simple diferenciar los casos y asignar los factores directamente.

El segundo caso de violación de la condición AVL se ilustra en la figura 6.9(a). El árbol AVL  $T$  sufre un desequilibrio por la derecha en el nodo  $x$ , causado por un aumento

de altura del nodo  $z$  que estaba perfectamente equilibrado. La inserción puede ocurrir por cualquiera de los lados de  $z$ .

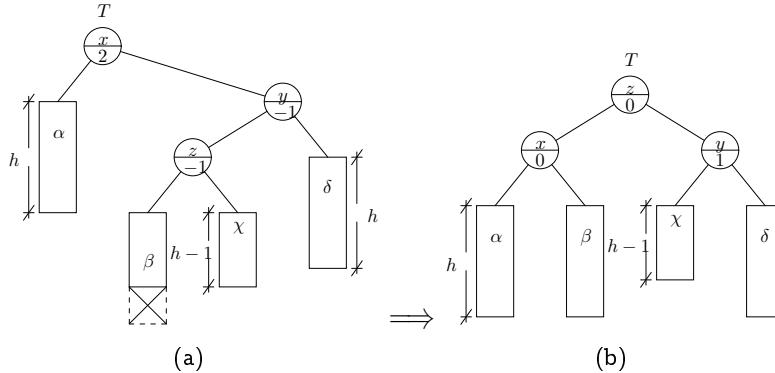


Figura 6.9: Segundo caso de inserción en un árbol AVL

El desequilibrio se corrige mediante dos rotaciones sucesivas del nodo  $z$ . Primero, el nodo  $z$  se rota hacia la derecha, luego, hacia la izquierda. La corrección se ilustra en la figura 6.9(b). Antes y después de la corrección, la altura global del árbol,  $h + 2$ , es la misma. Consecuentemente, si la ascendencia de  $T$  es AVL, entonces, después de la inserción y posterior corrección, la ascendencia también es AVL.

El caso ilustrado en las figuras 6.9(a) y 6.9(b) tiene su correspondiente simétrico por la izquierda, el cual es equivalente a reflejar las figuras en un espejo.

La identificación de la situación de desequilibrio de la figura 6.9(a), junto con su corrección, se caracteriza de la siguiente manera:

1. Si  $\text{DIFF}(x) == 2$  and  $\text{DIFF}(y) == -1 \Rightarrow$  primero se rota el nodo  $z$  hacia la derecha y luego hacia la izquierda.
2. Si  $\text{DIFF}(x) == -2$  and  $\text{DIFF}(y) == 1 \Rightarrow$  primero se rota el nodo  $z$  hacia la izquierda y luego hacia la derecha.

Los métodos `doubleRotateLeft()` y `doubleRotateRight()` realizan las dobles rotaciones a la izquierda y derecha respectivamente.

Los casos presentados en las figuras 6.8(a) y 6.9(a), junto con sus equivalentes simétricos, caracterizan situaciones de violación de la condición AVL que ocurren durante la inserción. Clasificaremos las acciones de corrección en el siguiente rango:

476a  $\langle \text{miembros privados de } \text{Gen_Avl_Tree} < \text{Key} > \text{ 473b} \rangle + \equiv \quad (473a) \triangleleft 474b \ 476b \triangleright$   
`enum Rotation_Type  
{ ROTATE_LEFT, ROTATE_RIGHT, DOUBLE_ROTATE_LEFT, DOUBLE_ROTATE_RIGHT };`

Este tipo de valor es devuelto por una rutina que examina un nodo desequilibrado y determina, en función del tipo de desequilibrio, cual clase de rotación debe ser aplicada:

476b  $\langle \text{miembros privados de } \text{Gen_Avl_Tree} < \text{Key} > \text{ 473b} \rangle + \equiv \quad (473a) \triangleleft 476a \ 477a \triangleright$   
`static Rotation_Type rotation_type(Node * p)  
{  
 Node * pc; // guarda hijo de p  
 if (DIFF(p) == 2) // hacia la izquierda  
 {  
 pc = RLINK(p);`

```

 if (DIFF(pc) == 1 or DIFF(pc) == 0)
 return ROTATE_LEFT;

 return DOUBLE_ROTATE_LEFT;
}

pc = LLINK(p);
if (DIFF(pc) == -1 or DIFF(pc) == 0)
 return ROTATE_RIGHT;

return DOUBLE_ROTATE_RIGHT;
}

```

`rotation_type()` es llamada por una función de uso general que restaura la condición AVL. Su estructura general corresponde a verificar los cuatro tipos de rotación en función del nodo violatorio de la condición AVL y del factor de su hijo. Notemos que `rotation_type()` efectúa una verificación adicional, `DIFF(pc) == 0`, correspondiente a otro caso que sólo se presenta en la eliminación y que explicaremos posteriormente.

Lo anterior proporciona toda la utilería requerida para restaurar la condición AVL en cualquiera de los casos de inserción:

477a *(miembros privados de Gen\_Avl\_Tree<Key> 473b)+≡ (473a) ▷ 476b 477b ▷*

```

static Node * restore_avl(Node * p, Node * pp)
{
 Node ** link = LLINK(pp) == p ? &LLINK(pp) : &RLINK(pp);
 switch (rotation_type(p))
 {
 case ROTATE_LEFT: return *link = rotateLeft(p);
 case ROTATE_RIGHT: return *link = rotateRight(p);
 case DOUBLE_ROTATE_LEFT: return *link = doubleRotateLeft(p);
 case DOUBLE_ROTATE_RIGHT: return *link = doubleRotateRight(p);
 }
 return NULL;
}

```

Uses `doubleRotateLeft`, `doubleRotateRight`, `rotateLeft`, and `rotateRight`.

El método `restore_avl()` toma dos nodos como parámetros. El primero, `p`, es el nodo que sufre el desequilibrio; el segundo, `pp`, es el padre de `p`. En función del tipo de desequilibrio determinado por `rotation_type()`, `restore_avl()` emprende las acciones necesarias para restaurar la condición AVL.

Hasta el presente hemos desarrollado una serie de rutinas encargadas de corregir una violación de la condición AVL en un nodo. Nos falta determinar cuál es el nodo que sufre el desequilibrio. Para la inserción, basta con memorizar el último nodo desequilibrado en el camino de búsqueda, pues éste es el único susceptible de sufrir un desequilibrio. Ubicado este nodo, se actualiza su factor y se verifica si se requiere emprender acciones correctivas con el método `restore_avl()`.

Ahora sólo nos resta una rutina que busque dentro de la pila el nodo desequilibrado, actualice el factor y, eventualmente, efectúe la corrección:

477b *(miembros privados de Gen\_Avl\_Tree<Key> 473b)+≡ (473a) ▷ 477a 480 ▷*

```

void restore_avl_after_insertion(Node * p) // ERROR const Key & key
{
 Node * pp = avl_stack.pop(); // padre del nodo insertado
}

```

*(Ajustar factor del padre del nodo insertado 478a)*

```
if (avl_stack_empty())
 return; // pp es raíz

Node *gpp; // padre de pp
(Actualizar ancestros de pp 478b)
```

```
 clean_avl_stack();
}
```

El bloque *(Ajustar factor del padre del nodo insertado 478a)* extrae el primer nodo en la pila, actualiza su factor y verifica si éste no es la raíz:

478a *(Ajustar factor del padre del nodo insertado 478a)≡* (477b)

```
if (LLINK(pp) == p) // ERROR cmp (key, KEY(pp)) // ajuste el factor del padre del nodo insertado 0
 DIFF(pp)-;
else
 DIFF(pp)++;

if (DIFF(pp) == 0)
{ // en este caso, altura del ascendiente de pp no aumenta
 clean_avl_stack();
 return;
}
```

*(Actualizar ancestros de pp 478b)* revisa cada ancestro gpp de pp y actualiza su factor. Si el factor de algún gpp es cero, entonces la altura de gpp permanece igual y, puesto que la ascendencia de gpp es AVL, el resto del árbol es AVL y el algoritmo termina. En caso contrario se revisa si hay una violación de la condición AVL, la cual se corrige. En este último caso, el algoritmo también termina, pues sabemos que en la inserción el árbol completo es AVL luego de efectuar la primera corrección:

478b *(Actualizar ancestros de pp 478b)≡* (477b)

```
do // buscar nodo con factor igual a 0
{
 gpp = avl_stack.pop();
 // actualizar factores de equilibrio
 if (LLINK(gpp) == pp) // ERROR cmp (KEY(p), KEY(gpp)) // ERROR if (Compare () (key, KEY(gpp)))
 DIFF(gpp)-;
 else
 DIFF(gpp)++;

 if (DIFF(gpp) == 0)
 break; // no se necesita reajuste
 else if (DIFF(gpp) == -2 or DIFF(gpp) == 2)// ¿es AVL?
 { // sí ==> se requiere reajuste
 Node *ggpp = avl_stack.pop();
 restore_avl(gpp, ggpp);
 break;
 }
 pp = gpp; // ERROR; añadir
}
while (not avl_stack_empty());
```

#### 6.4.1.2 Eliminación en un árbol AVL

Estructuralmente, la eliminación se divide en tres pasos: (1) buscar el nodo a eliminar, (2) eliminarlo y (3) revisar la condición AVL y eventualmente restaurarla. En ese sentido, el procedimiento se plantea del siguiente modo:

479a *(miembros públicos de Gen\_Avl\_Tree<Key> 474a) +≡* (473a) ▷ 474a

```

Node * remove(const Key & key)
{
 if (root == Node::NullPtr)
 return NULL;

 Node * p = search_and_stack_avl(key);
 if (no_equals<Key, Compare> (KEY(p), key))
 {
 // clave no fue encontrada
 clean_avl_stack();
 return NULL;
 }
 (Eliminación física de p 479b)

 restore_avl_after_deletion(left_deficit); // ERROR

 return p;
}

```

#### Eliminación física del nodo

*(Eliminación física de p 479b)* se fundamenta en el esquema alternativo de eliminación explicado en § 4.9.6-1 (página 324), el cual, recordemos, consiste en garantizar que el nodo a eliminar sea incompleto, y en el caso de que sea completo, entonces sustituirlo por el sucesor o el predecesor. En nuestro caso optamos por el sucesor y planteamos el siguiente algoritmo:

479b *(Eliminación física de p 479b) ≡* (479a)

```

Node * pp = avl_stack.top(1); // obtener padre de p
bool left_deficit; // ERROR Key removed_key = KEY(p);
while (true)
{
 left_deficit = LLINK(pp) == p; // ERROR Añadir
 if (LLINK(p) == Node::NullPtr) // ¿incompleto por la izquierda?
 {
 // Sí, ate a pp el hijo de p
 if (LLINK(pp) == p)
 LLINK(pp) = RLINK(p);
 else
 RLINK(pp) = RLINK(p);
 break;
 }
 if (RLINK(p) == Node::NullPtr) // ¿incompleto por la izquierda?
 {
 // Sí, ate a pp el hijo de p
 if (LLINK(pp) == p)
 LLINK(pp) = LLINK(p);
 else
 RLINK(pp) = LLINK(p);
 break;
 }
}

```

```

 // aquí p es un nodo completo ==> intercambiar por sucesor
 swapWithSuccessor(p, pp);
 // removed_key = KEY(succ); // ERROR eliminar
 }
 if (pp == head_ptr) // verifique si se eliminó la raíz
 { // factores quedan inalterados ==> no se viola condición AVL
 clean_avl_stack();
 return p;
 }
}

```

Si ocurre un intercambio del nodo a eliminar con su sucesor infijo, entonces el nodo eliminado no sirve para actualizar el factor del nodo sucesor, pues existe una violación temporal de la propiedad de orden de un ABB. Por esta razón, la variable `removed_key` almacena la clave según la cual se actualizarán los factores en el camino de búsqueda.

Como su nombre expresa, la rutina `swapNodeWithSuccessor()` intercambia `p` con su sucesor infijo. A la excepción de los aspectos pertinentes a los árboles AVL, la siguiente implantación es estructuralmente idéntica para la eliminación en toda clase de árbol binario de búsqueda que subyazca en el intercambio del nodo eliminado con su sucesor (o predecesor) infijo:

480 *(miembros privados de Gen\_Avl\_Tree<Key> 473b)+≡ (473a) ▷477b 483▷*

```

Node* swapWithSuccessor(Node * p, Node *& pp)
{
 // Referencia al tope de la pila, pues p será intercambiado con
 // sucesor y la posición en la pila la ocupará el sucesor de p
 Node *& ref_to_stack_top = avl_stack.top();
 // encuentre el sucesor a la vez que actualiza la pila
 Node *fSucc = p; // padre del sucesor
 Node *succ = RLINK(p); // búsqueda comienza desde RLINK(p)

 avl_stack.push(succ);
 while (LLINK(succ) != Node::NullPtr) // descender lo más a la izq
 {
 fSucc = succ;
 succ = LLINK(succ);
 avl_stack.push(succ);
 }
 // actualice antigua entrada de pila ocupada por p. Equivale
 // a intercambiar antiguo tope (antes de buscar suc) con actual
 ref_to_stack_top = succ;
 avl_stack.top() = p;
 if (LLINK(pp) == p) // actualice el nuevo hijo de pp (sucesor)
 LLINK(pp) = succ;
 else
 RLINK(pp) = succ;

 LLINK(succ) = LLINK(p); // intercambie las ramas izquierdas
 LLINK(p) = Node::NullPtr;
 if (RLINK(p) == succ) // actualice ramas derechas
 { // sucesor es exactamente el hijo derecho de p
 RLINK(p) = RLINK(succ);
 RLINK(succ) = p;
 pp = succ;
 }
}

```

```

 }
else
{ // sucesor es el descendiente más a la izquierda de RLINK(p)
 Node *succr = RLINK(succ);
 RLINK(succ) = RLINK(p);
 LLINK(fSucc) = p;
 RLINK(p) = succr;
 pp = fSucc;
}
DIFF(succ) = DIFF(p); // intercambie factores de equilibrio

return succ;
}

```

### Restauración de la condición AVL

La figura 6.10(a) ilustra el primer caso de desequilibrio causado por una eliminación. La figura supone una eliminación en la rama  $\alpha$  que le causa una pérdida de altura. La situación es idéntica a la figura 6.8(a) y su corrección similar, es decir, una rotación hacia la izquierda cuyo resultado, idéntico a la figura 6.8(b), se presenta en la figura 6.10(b).

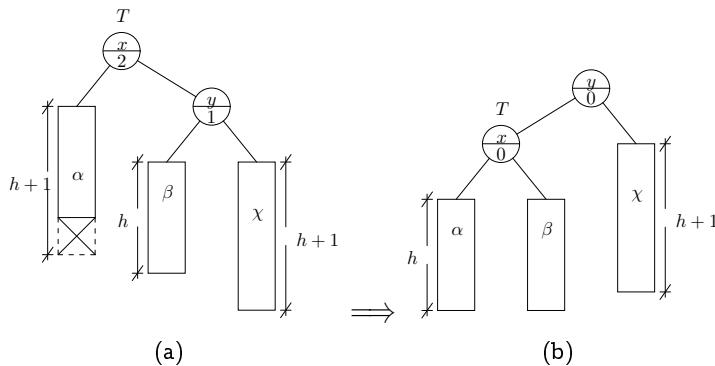


Figura 6.10: Primer caso de eliminación en un árbol AVL

Dado el nodo en el cual ocurre la violación, la identificación del primer caso es idéntica a la de la inserción, es decir, factores con el mismo signo y una unidad de diferencia. Para este caso, entonces, podemos reutilizar la rutina `rotation_type()`. Sin embargo, a diferencia de la inserción, el árbol T sufre una pérdida general de altura. En efecto, antes de la eliminación, el árbol T tiene altura general  $h + 3$ , mientras que después de eliminar la altura general es  $h + 2$ . Consecuentemente, la ascendencia de T puede sufrir un desequilibrio violatorio de la condición AVL.

La figura 6.11(a) muestra el segundo caso de eliminación en un AAVL, el cual también es similar al segundo caso de la inserción mostrado en la figura 6.9(a). En la figura 6.11(a), la rama  $\alpha$  sufre una pérdida de altura debida a una eliminación o a un ajuste de equilibrio. A diferencia de la figura 6.9(a), el nodo  $z$  puede estar perfectamente equilibrado o sufrir un ligero y no violatorio desequilibrio. En todo caso, la altura de  $z$  es  $h + 1$ . La corrección se presenta en la figura 6.11(b), que es la misma que para la inserción presentada en la figura 6.9(b): una rotación doble, cruzada, del nodo  $z$  hacia la derecha y luego hacia la izquierda.

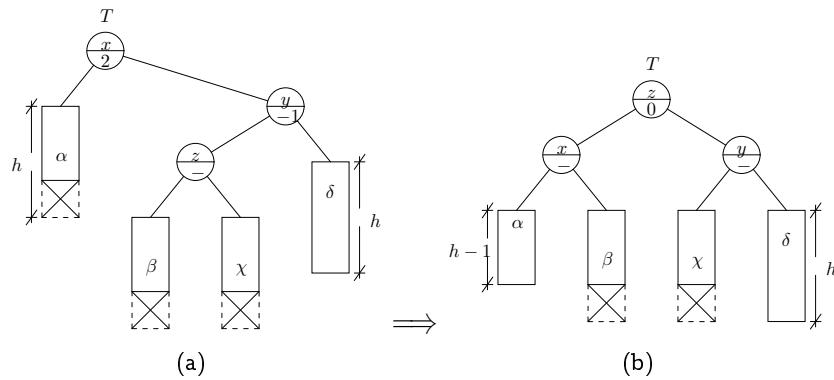


Figura 6.11: Segundo caso de eliminación en un árbol AVL

Similar al primer caso, el segundo también reduce la altura general del árbol T de  $h+3$  a  $h+2$ . Consecuentemente, también es necesario revisar la ascendencia de T en búsqueda de violaciones de la condición AVL causadas por el ajuste. Dado un nodo violatorio de la condición AVL, el segundo caso es únicamente identificable: los signos del nodo violatorio y su hijo son inversos. Puesto que esta es la misma situación que con el segundo caso de violación en la inserción, también podemos reutilizar la rutina `rotation_type()`.

El tercer y último caso de violación causado por una eliminación se ilustra en la figura 6.12(a) (Pág. 482). Aquí, la rama  $\alpha$  pierde altura debido a una eliminación o a un ajuste previo. Este caso y su solución son parecidos al primer caso ilustrado en las figuras 6.10(a) (Pág. 481). La diferencia estriba en que el nodo y está equilibrado. Este caso también es únicamente identifiable: el hijo del nodo violatorio está equilibrado y fue considerada, pero hasta ahora no explicada, en el diseño de la rutina `rotation_type()`.

La corrección del tercer caso consiste en una rotación simple hacia la izquierda que arroja como solución la situación descrita en la figura 6.12(b). Debido a que  $y$  está inicialmente equilibrado, la altura general de  $T$  no cambia. Por lo tanto, si la ascendencia de  $T$  es AVL, el árbol resultante de la corrección también lo es y la eliminación puede culminar ante este caso.

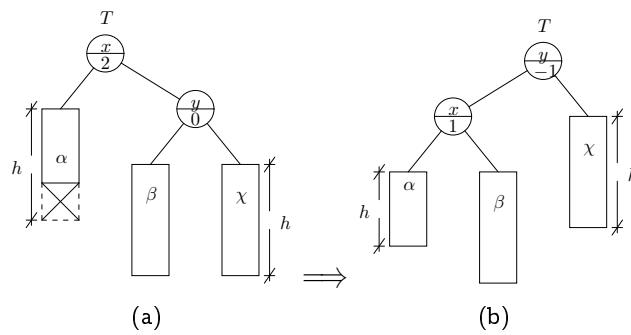


Figura 6.12: Tercer caso de eliminación en un árbol AVL

Cada uno de los tres casos tiene sus equivalentes simétricos fácilmente interpretables imaginándolos reflejados en un espejo.

De manera general, después de la eliminación física debemos retroceder en el camino de búsqueda, actualizar cada factor y revisarlo para ver si existe alguna violación. Si

se encuentra un factor que represente una violación, entonces debemos corregirla según los casos presentados. Si la corrección corresponde al tercer caso, entonces el algoritmo termina. De lo contrario debemos repetir el procedimiento hasta encontrar un factor que no represente una violación o hasta encontrarnos con la raíz. Esta conducta se expresa en la siguiente función:

```
483 <miembros privados de Gen_Avl_Tree<Key> 473b>+≡ (473a) ▷480
 void restore_avl_after_deletion(bool left_deficit) // ERROR const Key & key)
 {
 Node * pp = avl_stack.top(1); // padre de p
 Node * ppp = avl_stack.popn(3); // elimina de pila p, padre y abuelo
 while (true)
 { // actualice factores de equilibrio
 if (left_deficit) // ERROR Compare () (key, KEY(pp)))
 DIFF(pp)++;
 else
 DIFF(pp)--;
 if (DIFF(pp) == -2 or DIFF(pp) == 2) // ¿es válido?
 pp = restore_avl(pp, ppp); // no!
 if (DIFF(pp) != 0 or pp == root)
 break; // altura global de árbol no ha cambiado ==> terminar
 left_deficit = LLINK(ppp) == pp;
 pp = ppp; // avance al próximo ascendiente
 ppp = avl_stack.pop();
 }
 clean_avl_stack();
 }
```

#### 6.4.2 Análisis de los árboles AVL

Comenzaremos nuestro análisis por la presentación de la siguiente proposición y discusión de sus consecuencias.

**Proposición 6.4 (Adelson Velsky - Landis, 1962 [6])** Sea  $T$  un árbol AVL con  $n$  nodos, entonces:

$$\lg(n + 1) \leq h(T) < 1.4404 \lg(n + 2) - 0.3277 \quad (6.10)$$

**Demostración** Ver § 6.4.2.2 (Pág. 488).

La demostración es algo dificultosa y se desarrollará en detalle en las secciones subsiguientes. Pero sus consecuencias para el análisis de las operaciones de inserción y eliminación son mucho más evidentes.

La primera observación que conlleva la proposición 6.10 es que la altura máxima de un árbol AVL es  $\mathcal{O}(\lg n)$ . No importa cuánto crezca el árbol, el aumento de la altura siempre es logarítmico.

Respecto al coste constante tenemos, en el peor de los casos, una altura un 44% mayor que la de un árbol perfectamente equilibrado. En las situaciones restantes, la altura es

menor y el tiempo de búsqueda es mejor. Ahora bien, ¿qué tan costoso es este 44%? Si recordamos los 1300 millones de chinos considerados en § 4.9 (Pág. 313), el nombre de Zhang Shunniean Cheung Wu puede ser localizado en a lo sumo  $\lceil 1.44 \lg 1300000000 \rceil = 44$  intentos, 13 más que con un árbol perfectamente equilibrado, y esto sólo si somos muy desafortunados.

La cota impuesta por la proposición 6.10 ataña directamente a la búsqueda. ¿Qué acerca de la inserción y eliminación? La inserción requiere una búsqueda que, por la proposición 6.10, es  $\mathcal{O}(\lg n)$ . Luego, se retrocede en el camino de búsqueda para verificar si hay algún desequilibrio, que no toma más de tres nodos, y que lo hace, pues, constantemente acotado. Si se encuentra un desequilibrio, entonces éste se corrige en, a lo sumo, dos rotaciones, las cuales también están constantemente acotadas. Podemos concluir entonces que la inserción es  $\mathcal{O}(\lg n) + \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(\lg n)$ .

El primer paso de la eliminación es una búsqueda, la cual ya concluimos que toma  $\mathcal{O}(\lg n)$ . Después, debemos retroceder en el camino de búsqueda, verificar si hay desequilibrio y, si es el caso, corregirlo. Recordemos que una corrección de desequilibrio puede causar otro desequilibrio en la ascendencia del nodo corregido. En el peor de los casos, la primera corrección puede causar una cadena de correcciones sucesivas que puede llegar hasta la raíz. Puesto que la altura es  $\mathcal{O}(\lg n)$ , la cadena de correcciones es a lo sumo  $\mathcal{O}(\lg n)$  si ésta llegase hasta la raíz. La eliminación es, pues,  $\mathcal{O}(\lg n) + \mathcal{O}(\lg n) = \mathcal{O}(\lg n)$ , más costosa que la inserción, pero también  $\mathcal{O}(\lg n)$ .

En conclusión, todas las operaciones de un árbol AVL son  $\mathcal{O}(\lg n)$  para el peor caso. Si se requieren garantías de desempeño y se dispone de una implementación, los árboles AVL son una buena escogencia.

#### 6.4.2.1 Árboles de Fibonacci

Para demostrar la proposición 6.10, es conveniente estudiar una clase especial de árbol denominado de “Fibonacci”.

**Definición 6.5 (Árbol de Fibonacci)** Un árbol de Fibonacci de orden  $k$ , denominado  $T_k$ , se define recursivamente como:

$$T_k = \begin{cases} \emptyset & \text{si } k = 0 \\ \langle \emptyset, 0, \emptyset \rangle & \text{si } k = 1 \\ \langle T_{k-1}, \text{Fib}(k+1) - 1, T_{k-2} + \text{Fib}(k+1) \rangle & \text{si } k \geq 2 \end{cases}$$

El último término,  $\langle T_{k-1}, \text{Fib}(k+1) - 1, T_{k-2} + \text{Fib}(k+1) \rangle$ , debe interpretarse cuidadosamente. La rama izquierda,  $T_{k-1}$  es el árbol de Fibonacci de orden  $k-1$ . La raíz de  $T_k$  está etiquetada con  $\text{Fib}(k+1) - 1$  donde  $\text{Fib}(i)$  es el  $i$ -ésimo número de Fibonacci. La rama derecha es el árbol de Fibonacci de orden  $k-2$ , pero con todos sus nodos sumados en  $\text{Fib}(k+1)$ .

La figura 6.13 muestra el árbol de Fibonacci para  $k = 8$ . La rama izquierda es el árbol  $T_7$ . A su vez, la rama izquierda de  $T_7$  es el árbol  $T_6$  quien a su vez tiene de rama izquierda a  $T_5$  y así sucesivamente. La rama derecha de  $T_k$  tiene exactamente la misma forma que  $T_{k-1}$ , pero cada nodo es incrementado en  $\text{Fib}(k+1)$ . Por ejemplo, en la figura 6.13 se puede verificar que la rama derecha principal tiene exactamente la misma forma que  $T_6$ , excepto que los valores de cada nodo son incrementados en  $\text{Fib}(9) = 34$ .

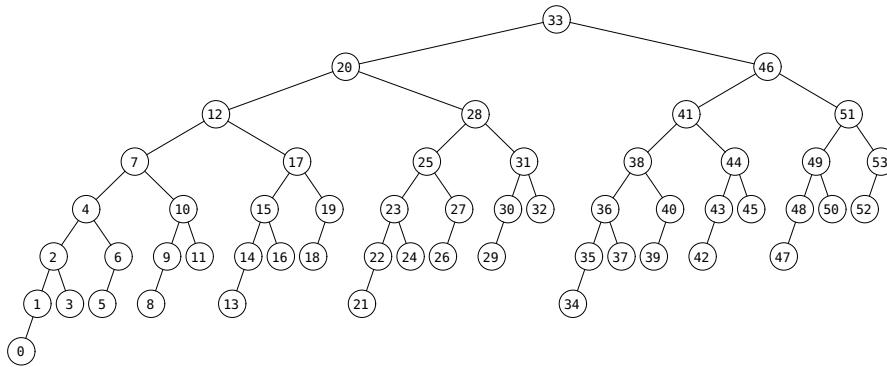


Figura 6.13: Árbol de Fibonacci de orden 8

De la figura 6.13 se observa que la etiqueta de cada nodo es exactamente su posición infija. Este “truco”<sup>3</sup> nos permite intuir la cardinalidad de un árbol de Fibonacci. La raíz del árbol de Fibonacci de orden 8 mostrado en la figura 6.13 es  $\text{Fib}(8+1)-1 = 33$ . Puesto que los nodos están etiquetados con su posición infija,  $\text{Fib}(7+2) = 33$  es la cantidad de nodos que preceden al árbol izquierdo. Pero, por definición,  $L(T_8) = T_7$ , por lo que podemos deducir que  $|T_7| = \text{Fib}(7+2) - 1$ . Podemos observar diferentes árboles de Fibonacci y corroborar esta intuición mediante la siguiente proposición:

**Proposición 6.5** El número de nodos de un árbol de Fibonacci de orden  $k$  es  $\text{Fib}(k+2) - 1$ .

**Demostración (por inducción sobre  $k$ )**

- $k = 0$ : En este caso,  $|T_0| = \text{Fib}(0) = 0$ . La proposición es cierta para  $k = 0$ .
- $k > 0$ : Ahora asumimos que la proposición es cierta para todo  $k$  y verificamos si aún lo es para  $k + 1$ .

Por definición:

$$|T_{k+1}| = |T_k| + 1 + |T_{k-1}| \quad (6.11)$$

es decir, el número de nodos de la rama izquierda más la raíz más el número de nodos de la rama derecha. Aplicando la hipótesis inductiva a la ecuación (6.11), tenemos:

$$\begin{aligned} |T_{k+1}| &= \text{Fib}(k+1) - 1 + 1 + \text{Fib}(k+1) - 1 \\ &= \text{Fib}(k+2) + \text{Fib}(k+1) - 1 \\ &= \text{Fib}(k+3) - 1 \end{aligned}$$

La proposición es entonces cierta para todo  $k$  ■

La altura de un árbol de Fibonacci también es fácilmente sugerible por observación. En el caso del árbol de orden 8 mostrado en la figura 6.13 es 8; es decir, su orden.

**Proposición 6.6** La altura del árbol de Fibonacci de orden  $k$  es  $k$ .

<sup>3</sup>En realidad, el interés primario de los árboles de Fibonacci es topológico, pero la manera en que los etiquetamos nos hace más aprensible las demostraciones.

### Demostración (por inducción sobre $k$ )

- $k = 0$ : este caso es directo, pues el árbol vacío tiene altura nula.
- $k > 0$ : ahora asumimos que la proposición es cierta para todo  $k$  y verificamos para  $k + 1$ . Sea  $T_{k+1}$  el árbol de Fibonacci de orden  $k + 1$ . Entonces,  $h(T_{k+1}) = 1 + \max(h(T_k), h(T_{k-1}))$ . Aplicando la hipótesis inductiva,  $\max(h(T_k), h(T_{k-1})) = h(T_k) = k$ , por lo que  $h(T_{k+1}) = 1 + k$  ■

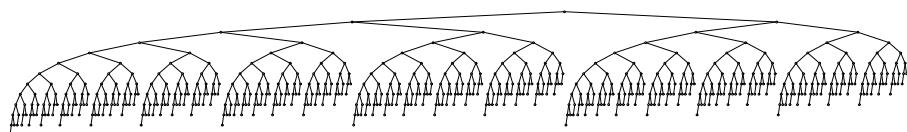


Figura 6.14: Árbol de Fibonacci de orden 13

La proposición anterior es fundamental para demostrar que un árbol de Fibonacci es AVL, cuestión que responde la proposición que sigue:

**Proposición 6.7** Un árbol de Fibonacci es AVL.

**Demostración** Sea  $T_k$  el árbol de Fibonacci de orden  $k$ . Construiremos la demostración por inducción sobre  $k$ :

- $k = 0$ : este caso es directo, pues el árbol vacío es AVL.
- $k = 1$ : en este caso,  $T_1$ , compuesto por un solo nodo, es un árbol AVL.
- $k > 0$ : ahora asumimos que la proposición es cierta para todo  $k$  y verificamos para  $T_{k+1}$ .

Para que  $T_{k+1}$  sea AVL, la diferencia de alturas de todos sus nodos no debe exceder de uno. Por definición, las ramas de  $T_{k+1}$  son árboles de Fibonacci, los cuales, por la hipótesis inductiva, son AVL. Así pues, el único nodo en que hay que verificar la condición AVL es en la raíz de  $T_{k+1}$ . La diferencia de alturas de  $raiz(T_{k+1})$  está dada por  $\delta(raiz(T_{k+1})) = h(R(T_{k+1})) - h(L(T_{k+1})) = h(T_{k-1}) - h(T_k)$ , la cual, por la proposición 6.6, es  $k - 1 - k = -1$ . Puesto que todos los nodos de  $T_{k+1}$  satisfacen la condición AVL,  $T_{k+1}$  es AVL y la proposición es cierta para todo  $k$  ■

Cualquier árbol de Fibonacci es AVL. Desde esta perspectiva, los árboles de Fibonacci son de alto interés, pues nos caracterizan árboles AVL de altura máxima. Para apreciar esta cuestión requerimos la siguiente definición:

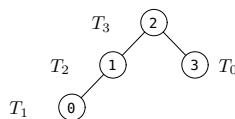
**Definición 6.6 (Árbol AVL crítico)** Sea  $T$  un árbol AVL de altura  $h$ . Se dice que  $T$  es crítico si, y sólo si, al eliminar un nodo, el árbol deja de ser AVL o pierde altura.

La siguiente proposición nos establece la criticidad de un árbol de Fibonacci.

**Proposición 6.8** Un árbol de Fibonacci es un árbol AVL crítico.

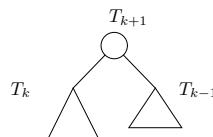
**Demostración (por inducción sobre  $k = |T|$ )**

- $k = 0$ : en este caso,  $T_0 = \emptyset$  que, si bien es por definición AVL, no admite eliminación.
- $k = 1$ : en este caso,  $T_1$  es un árbol conformado por un sólo nodo. Al efectuar la única eliminación posible se depara en  $T_0$ , el cual, por la proposición 6.6, tiene menor altura que  $T_1$ . El lema es, pues, cierto para este caso.
- $k = 2$ : en este caso se trata de  $T_2$ , del cual sólo es posible eliminar dos nodos. Independientemente del nodo que eliminemos, el árbol depara en  $T_1$ , o sea, que se pierde altura.
- $k = 3$ : entonces tenemos  $T_3$ :



De aquí tenemos cuatro casos:

1. Si eliminamos el nodo 0, entonces el árbol pierde altura.
  2. Si eliminamos el nodo 3, entonces el árbol deja de ser AVL y cualquier ajuste que se haga conlleva una pérdida de altura.
  3. Si eliminamos el nodo 1, entonces al atar la raíz 2 con la hoja 0 el árbol pierde altura.
  4. Finalmente, si eliminamos la raíz 2, entonces tenemos dos alternativas para sustituirla:
    - (a) Colocamos el nodo 1 como raíz, lo que acarrea que el árbol pierda altura.
    - (b) Colocamos el nodo 3 como raíz, lo que causa que el árbol deje de ser AVL.
- $k > 3$ : Ahora asumimos que el lema es cierto para todo  $k$  y verificamos si aún lo es para  $k + 1$ . El árbol  $T_{k+1}$  puede pictorizarse como:



Por la hipótesis inductiva, los árboles  $T_k$  y  $T_{k-1}$  son críticos, así que al eliminar cualquier nodo de ellos el árbol dejaría de ser AVL o perdería altura. Así pues, el único punto de duda lo constituye la eliminación de la raíz de  $T_{k+1}$ .

Al eliminar la raíz, que es un nodo completo, debemos sustituirla por alguno de los nodos de  $T_k$  o  $T_{k-1}$ . Tenemos dos casos:

1. Sustituimos la raíz por un nodo de  $T_k$ : si  $T_k$  pierde altura, entonces  $T_{k+1}$  también pierde altura.  
Si  $T_k$  deja de ser AVL, entonces  $T_{k+1}$  también deja de ser AVL.
2. Sustituimos la raíz por un nodo de  $T_{k-1}$ : si  $T_{k-1}$  deja de ser AVL, entonces  $T_{k+1}$  también deja de ser AVL.  
Si  $T_{k-1}$  pierde altura, entonces ocurre una violación de la condición AVL en la raíz de  $T_{k+1}$ , pues  $\delta(T_{k+1}) = -2$ .  $T_{k+1}$  deja pues de ser AVL.

Puesto que  $T_{k+1}$  es un AVL crítico, entonces la proposición es cierta para todo  $k$  ■

Estudiados los árboles de Fibonacci, tenemos todas las herramientas requeridas para la demostración de la proposición 6.4.

#### 6.4.2.2 Demostración de la proposición 6.4

La altura mínima de un árbol AVL es la misma que para cualquier árbol binario, la cual fue demostrada en la proposición 4.1. Por tanto, nos concentraremos en estudiar la altura máxima que puede alcanzar un árbol AVL.

Estamos interesados en encontrar una manera de disponer  $n$  nodos en un árbol AVL de manera tal que su altura sea máxima. En otras palabras, debemos disponer los  $n$  nodos en un árbol crítico. Como sabemos de la proposición 6.8, un árbol de Fibonacci es crítico. Del mismo modo, por la proposición 6.7, un árbol de Fibonacci es AVL. Con seguridad podemos buscar la cota de la altura máxima a través de un árbol de Fibonacci.

Dada una altura nominal  $h$ , por la proposición 6.6, el árbol de Fibonacci  $T_h$ , AVL y crítico, tiene, según la proposición 6.5,  $\text{Fib}(h + 2) - 1$  nodos. Así pues, planteamos:

$$n \geq |T_h| = \text{Fib}(h + 2) - 1 \quad (6.12)$$

Cualquier árbol AVL de altura  $h$  tiene que tener igual o más nodos que  $|T_h|$ , pues  $T_h$  es crítico. De este modo, lo único que nos resta es encontrar una expresión que nos denote el  $n$ -ésimo número de Fibonacci y despejar  $h$  de la desigualdad (6.12). A partir del conocimiento de que  $\text{Fib}(0) = 0$ ,  $\text{Fib}(1) = 1$ , debemos resolver la ecuación de recurrencia:

$$\text{Fib}(k) = \text{Fib}(k - 1) + \text{Fib}(k - 2) \Rightarrow \quad (6.13)$$

$$\text{Fib}(k + 2) = \text{Fib}(k + 1) + \text{Fib}(k) \Rightarrow \quad (6.14)$$

La ecuación (6.14) es una ecuación recurrente homogénea de segundo orden. Se llama de esta manera porque es reminiscente a una ecuación diferencial homogénea de segundo orden. De hecho, las ecuaciones de recurrencias a menudo se llaman "ecuaciones de diferencias" por su similitudes con las ecuaciones diferenciales. En este sentido, la ecuación (6.14) es reminiscente a una ecuación diferencial lineal homogénea de segundo orden con coeficientes constantes, es decir, de forma  $\ddot{y} + a_1 \dot{y} + a_2 y = 0$ .

Ahora podemos resolver la ecuación recurrente (6.14) mediante un método análogo de resolución de una ecuación diferencial. Para ello, al igual que con una ecuación diferencial homogénea de segundo orden, aplicamos la transformación siguiente:

$$\text{Fib}(k) = c \omega^k, \quad c, \omega \neq 0, k \geq 2 . \quad (6.15)$$

Sustituimos (6.15) en (6.14), lo que proporciona la ecuación característica siguiente:

$$c \omega^{k+2} = c \omega^{k+1} + c \omega^k , \quad (6.16)$$

la cual, al dividir por  $c \omega^k$ , puede simplificarse a la legendaria "ecuación divina":

$$\omega^2 - \omega - 1 = 0 , \quad (6.17)$$

cuyas raíces son:

$$\omega = \frac{1 \pm \sqrt{5}}{2} . \quad (6.18)$$

Estos números son legendarios, el número  $\varphi$  y  $\hat{\varphi}$  cuyos valores son:

$$\varphi = \frac{1 + \sqrt{5}}{2} \approx 1.61803 , \quad (6.19)$$

$$\hat{\varphi} = \frac{1 - \sqrt{5}}{2} \approx 0.61803 \quad (6.20)$$

De esta manera, la solución general es:

$$\text{Fib}(k) = c_1 \varphi^k + c_2 \hat{\varphi}^k \quad (6.21)$$

Para encontrar los valores de  $c_1$  y  $c_2$ , hacemos lo mismo que con una ecuación diferencial: planteamos un sistema de ecuaciones con valores conocidos de  $\text{Fib}(k)$ . En este caso utilizamos  $\text{Fib}(0) = 0$  y  $\text{Fib}(1) = 1$ :

$$\text{Fib}(0) = 0 = c_1 \varphi^0 + c_2 \hat{\varphi}^0 = c_1 + c_2 \quad (6.22)$$

$$\text{Fib}(1) = 1 = c_1 \varphi^1 + c_2 \hat{\varphi}^1 = c_1 \varphi + c_2 \hat{\varphi} \quad (6.23)$$

De la ecuación (6.22) tenemos:

$$c_2 = -c_1 \quad (6.24)$$

y sustituyendo (6.24) en (6.23):

$$c_1 \varphi - c_1 \hat{\varphi} = 1 \implies c_1(\varphi - \hat{\varphi}) = 1 \implies c_1 = \frac{1}{\varphi - \hat{\varphi}} = \frac{1}{\sqrt{5}} \quad (6.25)$$

Combinando (6.25) y (6.24) en (6.21):

$$\text{Fib}(k) = \frac{1}{\sqrt{5}} \varphi^k - \frac{1}{\sqrt{5}} \hat{\varphi}^k = \frac{\varphi^k - \hat{\varphi}^k}{\sqrt{5}} \quad (6.26)$$

Puesto que  $\hat{\varphi} < 1$ , el término  $\hat{\varphi}^k$  deviene exponencialmente muy pequeño conforme crece  $k$ , ergo  $\hat{\varphi}$  es despreciable en (6.26) para valores de  $k$  muy grandes. Consecuentemente,  $\text{Fib}(k)$  es bastante cercano a  $\varphi^k/\sqrt{5}$  conforme crece  $k$ . Así pues, podemos tener una expresión más simple para calcular el  $k$ -ésimo número de Fibonacci:

$$\text{Fib}(k) = \frac{\varphi^k}{\sqrt{5}} \quad \text{redondeado al entero más próximo} \quad (6.27)$$

lo cual nos permite completar la demostración de la proposición 6.4.

Ahora retomamos la desigualdad (6.12):

$$n \geq |\text{T}_h| = \text{Fib}(h+2) - 1 \implies \quad (6.28)$$

$$> \left( \frac{\varphi^{h+2}}{\sqrt{5}} \text{redondeado al entero más cercano} \right) - 1 \implies \quad (6.29)$$

$$n > \frac{\varphi^{h+2}}{\sqrt{5}} - 2 \quad (6.30)$$

Notemos que la ecuación (6.27) establece que la función redondeo al entero más cercano debe aplicarse. Para despejar  $h$  de la inecuación (6.29), sería necesario conocer la inversa

de la función de redondeo. Por esa razón restamos una unidad del lado derecho de la inecuación (6.29) y, a partir de allí, expresamos una desigualdad estricta:

$$\begin{aligned}\varphi^{h+2} &< \sqrt{5}(n+2) \implies \\ h+2 &< \log_{\varphi}[\sqrt{5}(n+2)] \implies \\ h &< \log_{\varphi}(n+2) + \log_{\varphi}(\sqrt{5}) - 2 \implies \\ h &< \frac{\lg(n+2)}{\lg \varphi} - 0.3277 \implies \\ h &< 1.4404 \lg(n+1) - 0.3277 \quad \blacksquare\end{aligned}$$

## 6.5 Árboles rojo-negro

Ahora estudiaremos una clase de árbol binario cuyo esquema de balance también es garantizado. El esquema es “cromático” en el sentido de que los nodos son “coloreados” de rojo o negro, respectivamente. Las reglas de coloración proporcionan argumentos combinatorios, en función de las posibles combinaciones de colores, que garantizan un equilibrio.

**Definición 6.7 (Árbol rojo-negro)** Un árbol rojo-negro, o ABRN, es un árbol binario de búsqueda, con un atributo adicional en cada uno de sus nodos llamado “color”, que satisface las siguientes condiciones denominadas “cromáticas”:

- **Condición cromática primaria:** un nodo es rojo o negro.
- **Condición roja:** si un nodo es rojo, entonces sus dos hijos deben ser negros. Esto garantiza que dos nodos rojos nunca pueden estar contiguos en el camino de búsqueda.
- **Condición negra:** cualquier camino desde la raíz hasta un nodo externo contiene el mismo número de nodos negros.
- **Nodo externo negro:** los nodos externos son negros.

Esta definición y los algoritmos resultantes están fuertemente inspirados de la excelsa presentación de los árboles rojo-negro de Derick Wood [184]. Su definición difiere ligeramente de otros estudiosos que exigen que la raíz siempre sea negra.

La figura 6.15 ilustra un ejemplo de un árbol rojo-negro en la cual los nodos rojos están sombreados.

La condición negra justifica un nuevo concepto: la altura negra, la cual se define como sigue.

**Definición 6.8 (Altura negra en un nodo rojo-negro)** Sea  $T$  un árbol rojo-negro y  $n_i \in T$  un nodo cualquiera de  $T$ . Se define la “altura negra” de  $n_i$ , denotada como  $bh(n_i)$ , como el número de nodos negros desde  $n_i$  hasta cualquier nodo externo de  $T$ .

Notemos que la altura negra está definida para cualquier camino desde  $raiz(T)$ . Esto es una consecuencia directa de la condición negra que nos garantiza que esta altura es la misma independientemente de la hoja que se considere.

El primer paso hacia el diseño de un tipo abstracto de dato es especificar un nodo rojo-negro:

```
491 <rbNode.H 491>≡
 typedef unsigned char Color;
 # define COLOR(p) ((p)->getColor())
 # define RED (0)
 # define BLACK (1)
 class RbNode_Data
 {
 Color color; // RED o BLACK

 RbNode_Data() : color(RED) {}
 RbNode_Data(SentinelCtor) : color(BLACK) {}

 Color & getColor() { return color; }
 };
 DECLARE_BINNODE_SENTINEL(RbNode, 128, RbNode_Data);
Defines:
RbNode, used in chunk 492a.
Uses DECLARE_BINNODE_SENTINEL.
```

La clase RbNode está diseñada para cumplir las condiciones de base. El constructor por omisión siempre crea un nodo rojo. Un nuevo nodo rojo no altera la altura negra, por consiguiente, se preserva la condición negra. A efectos de satisfacer la última condición y facilitar los algoritmos, los árboles rojo-negro usan un nodo nulo centinela que es negro.

Intuitivamente podemos apreciar el equilibrio de un árbol rojo-negro: todo nodo está perfectamente “negro”-equilibrado, es decir, para todo nodo, la diferencia de alturas negras entre sus dos ramas es nula. De hecho, el árbol rojo-negro mostrado en la figura 6.16 está aceptablemente equilibrado.

### 6.5.1 El TAD Rb\_Tree<Key>

EL TAD Gen\_Rb\_Tree<Key> define un árbol rojo-negro cuyas operaciones principales son en función de nodos rojo-negro definidos anteriormente. La estructura principal es como

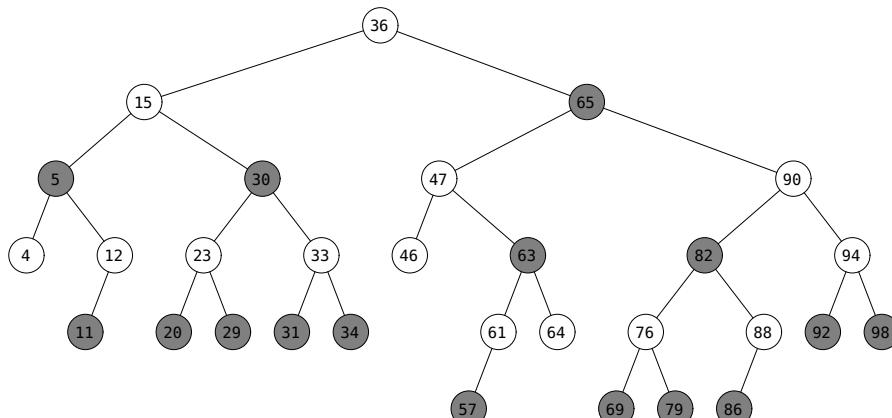


Figura 6.15: Un árbol rojo-negro

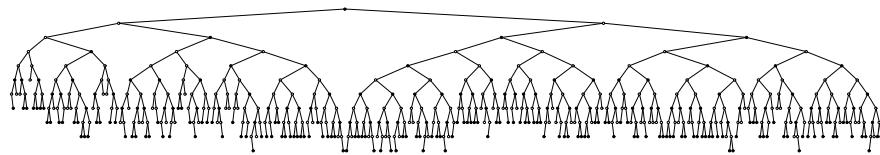


Figura 6.16: Un árbol rojo-negro de 512 nodos

sigue:

```
492a <tpl_rb_tree.H 492a>≡
 template <template <typename> class NodeType, typename Key, class Compare>
 class Gen_Rb_Tree
 {
 typedef NodeType<Key> Node;
 <miembros privados de Gen_Rb_Tree<Key> 492b>
 <miembros públicos de Gen_Rb_Tree<Key> 493>
 };
 template <typename Key, class Compare = Aleph::less<Key> >
 struct Rb_Tree : public Gen_Rb_Tree<RbNode, Key, Compare> {};
Uses RbNode 491.
```

Los fundamentos de implantación son muy similares a los demás árboles binarios estudiados: un nodo cabecera, uno centinela y dilucidación de uso o no de la recursividad. Los algoritmos son más sencillos que los de un árbol AVL, pero no lo suficiente como para permitir una implantación naturalmente recursiva.

Una de las decisiones que tomamos a priori es hacer una implantación iterativa asistida de una pila, la cual justifica los siguientes atributos:

```
492b <miembros privados de Gen_Rb_Tree<Key> 492b>≡ (492a) 492c▷
 Node head_node; // cabecera centinela
 Node * head; // puntero a centinela
 Node *& root;
 FixedStack<Node*, Node::MaxHeight> rb_stack;
Uses FixedStack 101a.
```

La inserción y eliminación requieren empilar el camino de búsqueda. Esta acción es efectuada por una rutina específica:

```
492c <miembros privados de Gen_Rb_Tree<Key> 492b>+≡ (492a) ▷492b 494▷
 Node * search_and_stack_rb(const Key & key)
 {
 Node * p = root;
 rb_stack.push(head);
 do
 {
 rb_stack.push(p);
 if (Compare () (key, KEY(p)))
 p = LLINK(p);
 else if (Compare () (KEY(p), key))
 p = RLINK(p);
 else
 return p;
 }
```

```

 while (p != Node::NullPtr);

 return rb_stack.top();
}

Defines:
search_and_stack_rb, used in chunks 493 and 497.

```

### 6.5.1.1 Inserción en un árbol rojo-negro

En principio, la inserción es exactamente igual a la inserción en un árbol binario estándar. Luego, si es necesario, se efectúan los ajustes para preservar el balance:

```

493 <miembros públicos de Gen_Rb_Tree<Key> 493>≡ (492a) 497▷
 Node * insert(Node * p)
 {
 if (root == Node::NullPtr)
 return root = p; // inserción en árbol vacío

 Node * q = search_and_stack_rb(KEY(p));
 if (Compare () (KEY(p), KEY(q)))
 LLINK(q) = p;
 else if (Compare () (KEY(q), KEY(p)))
 RLINK(q) = p;
 else
 {
 rb_stack.empty();
 return NULL; // clave duplicada
 }
 fix_red_condition(p);

 return p;
 }
 if (root == Node::NullPtr)
 return root = p; // inserción en árbol vacío

 Node * q = search_and_stack_rb(KEY(p));
 if (Compare () (KEY(p), KEY(q)))
 LLINK(q) = p;
 else if (Compare () (KEY(q), KEY(p)))
 RLINK(q) = p;
 else
 {
 rb_stack.empty();
 return q; // clave duplicada
 }
 fix_red_condition(p);

 return p;
}
Node * insert_dup(Node * p)
{
 I(COLOR(p) == RED);

```

```

if (root == Node::NullPtr)
 return root = p; // inserción en árbol vacío

Node * q = search_dup_and_stack_rb(KEY(p));
if (Compare () (KEY(p), KEY(q)))
 LLINK(q) = p;
else
 RLINK(q) = p;

fix_red_condition(p);

return p;
}

```

bool verify() { return is\_red\_black\_tree(root) ; }

Uses fix\_red\_condition 494, is\_red\_black\_tree, and search\_and\_stack\_rb 492c.

La rutina implanta la inserción física clásica. fix\_red\_condition() verifica violaciones de las condiciones cromáticas y las corrige si es el caso.

En el caso de la inserción, el truco es insertar un nodo rojo, pues éste no altera la condición negra, ergo, el balance de alturas negras. Empero, la inserción de un nodo rojo puede causar una violación de la condición roja si el padre del nodo insertado es rojo. Si el padre del nodo insertado es negro, entonces el árbol resultante es rojo-negro y el algoritmo termina. De lo contrario es necesario restablecer la condición roja sin alterar la condición negra.

La estructura de fix\_red\_condition() es como sigue:

494 ⟨miembros privados de Gen\_Rb\_Tree<Key> 492b⟩+≡ (492a) ◁ 492c 499 ▷

```

void fix_red_condition(Node * p)
{
 while (p != root)
 {
 Node * pp = rb_stack.pop(); // padre de p
 if (COLOR(pp) == BLACK) // ¿padre de p negro?
 break; // sí ==> no hay rojos consecutivos ==> terminar

 if (root == pp) // ¿p es hijo directo de la raíz?
 {
 // sí ==> colorear raíz de negro y terminar
 COLOR(root) = BLACK;
 break;
 }
 Node * spp = rb_stack.pop(); // abuelo de p
 Node * spp = LLINK(ppp) == pp ? RLINK(ppp) : LLINK(ppp); // tío
 if (COLOR(spp) == RED) // ¿tío de p rojo?
 {
 // intercambiar colores entre los niveles
 COLOR(ppp) = RED;
 COLOR(pp) = BLACK;
 COLOR(spp) = BLACK;
 p = ppp;
 continue; // ir próximo ancestro, verificar violaciones
 }
 }
}

```

```

Node * pppp = rb_stack.pop(); // bisabuelo de p
if (LLINK(pp) == p and LLINK(ppp) == pp)
{
 rotate_to_right(ppp, pppp);
 COLOR(pp) = BLACK;
}
else if (RLINK(pp) == p and RLINK(ppp) == pp)
{
 rotate_to_left(ppp, pppp);
 COLOR(pp) = BLACK;
}
else
{
 if (RLINK(pp) == p)
 {
 rotate_to_left(pp, ppp);
 rotate_to_right(ppp, pppp);
 }
 else
 {
 rotate_to_right(pp, ppp);
 rotate_to_left(ppp, pppp);
 }
 COLOR(p) = BLACK;
}
COLOR(ppp) = RED;
break; // árbol es rojo-negro ==> terminar
}
rb_stack.empty();
}

```

Defines:

`fix_red_condition`, used in chunk 493.

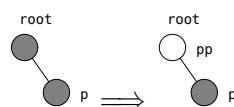
El algoritmo toma como entrada un nodo `p`, que es el nodo a insertar. La salida es un árbol rojo-negro con el nodo `p` insertado.

Al inicio del `while`, `p` es un nodo rojo recién insertado. Algunas correcciones pueden causar otras violaciones sobre la ascendencia de `p`. Cuando esto ocurre, `p` se actualiza a que apunte sobre un nodo rojo que viole la condición roja.

Si el padre de `p` es negro, entonces el nuevo nodo no causa ninguna violación y el algoritmo termina. Esta es la primera verificación del `while`.

Si, por el contrario, `pp` es rojo, entonces tenemos dos nodos rojos consecutivos que violan la condición roja. La corrección se efectúa según los siguientes casos:

1. `p` es hijo directo de la raíz: esta situación se verifica en el segundo `if` y se resuelve coloreando la raíz de negro:



El árbol resultante es rojo-negro y el algoritmo termina.

2. Tío de p es rojo: esta circunstancia, detectada por el tercer if, se sorteá con un intercambio de colores entre los niveles. Es decir, el abuelo p deviene rojo y sus dos hijos devienen negros. Los dos casos posibles se muestran en la figura 6.17 (Pág. 496).

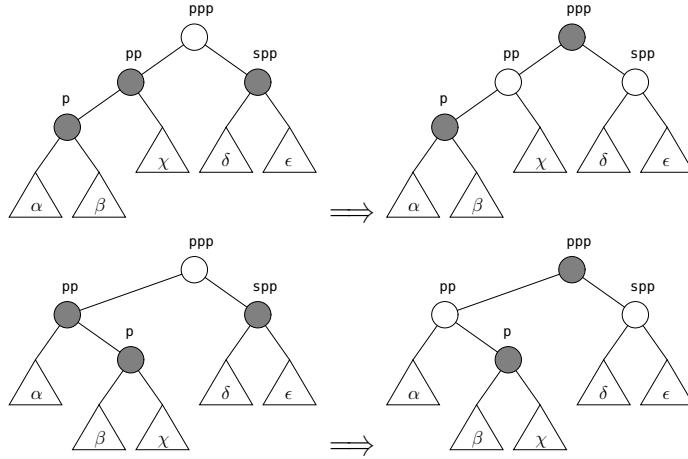


Figura 6.17: Intercambio de colores entre los niveles

Luego de este ajuste el padre de ppp es podría ser rojo, lo que acarrearía una violación de la condición roja. Por esa razón, ejecutamos  $p = \text{ppp}$  y avanzamos a una nueva iteración.

3. Tío de p es negro: en este caso hacemos rotaciones que transformen el problema en uno de los anteriores.

Hay dos posibles situaciones generales:

- (a) Esta situación se muestra en la figura 6.18 (Pág. 496) y se detecta porque p, pp y ppp están completamente alineados, sea hacia la izquierda o hacia la derecha. Su detección y procesamiento ocupan los cuarto y quinto if.

La corrección consiste en efectuar una rotación simple de pp hacia el lado de spp; esto hace que pp devenga la raíz del subárbol. Por último, se intercambian los colores de pp y ppp.

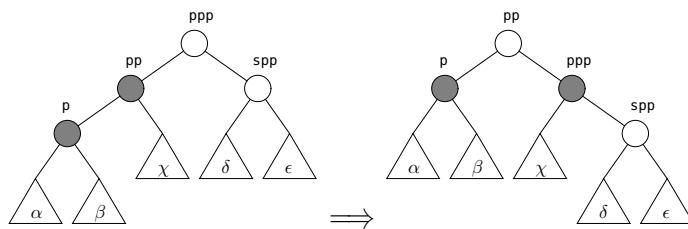


Figura 6.18: Rotación y coloración para restaurar condición roja

- (b) Esta situación se ilustra en la figura 6.19 (Pág. 497), y se presenta cuando p, pp y ppp no están completamente alineados. La solución consiste en una doble rotación cruzada de p hacia el lado de spp. En este caso, p deviene la raíz del subárbol y los colores de p y de ppp son intercambiados.

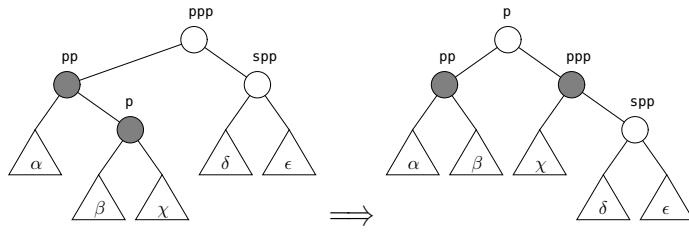


Figura 6.19: Doble rotación de p y coloración para restaurar condición roja

En las dos situaciones anteriores, el árbol resultante es rojo-negro y el algoritmo termina.

Algunas iteraciones pueden ocurrir cuando se cae en el segundo caso, pudiendo ser necesario subir hasta la raíz. Cuando esto ocurre, la eventual violación de la condición roja disminuye dos niveles. De este modo, el número máximo de iteraciones está acotado por la altura del árbol dividida entre dos ( $h/2$ ).

### 6.5.1.2 Eliminación en un árbol rojo-negro

En principio, la eliminación es similar a la de un árbol binario estándar. Luego, si es necesario, se hacen los ajustes que restauren las condiciones cromáticas.

497

```
<miembros públicos de Gen_Rb_Tree<Key> 493) +≡ (492a) ▷ 493
Node* remove(const Key & key)
{
 if (root == Node::NullPtr)
 return NULL;

 Node * q = search_and_stack_rb(key);
 if (no_equals<Key, Compare> (KEY(q), key)) // ¿clave no encontrada?
 {
 rb_stack.empty();
 return NULL;
 }

 Node * pq = rb_stack.top(1); // padre de 1
 Node * p; // hijo de q luego de que éste ha sido eliminado
 while (true) // eliminación clásica árbol binario de búsqueda
 {
 if (LLINK(q) == Node::NullPtr)
 {
 if (LLINK(pq) == q)
 p = LLINK(pq) = RLINK(q);
 else
 p = RLINK(pq) = RLINK(q);
 break; // goto end;
 }
 if (RLINK(q) == Node::NullPtr)
 {
 if (LLINK(pq) == q)
 p = LLINK(pq) = LLINK(q);
```

```

 else
 p = RLINK(pq) = LLINK(q);
 break; // goto end;
 }
 find_succ_and_swap(q, pq);
 }
 if (COLOR(q) == BLACK) // ¿se eliminó un nodo negro?
 fix_black_condition(p);

 q->reset();
 rb_stack.empty();

 return q;
}

```

Uses `find_succ_and_swap`, `fix_black_condition` 499, and `search_and_stack_rb` 492c.

La rutina busca la clave a la vez que empila el camino de búsqueda. El nodo a eliminar es llamado `q` y su padre `pq`. El `while` implanta la eliminación física del nodo, la cual, a estas alturas del estudio, debe ser comprensible para el lector. Después de la eliminación física existe el riesgo de violación de la condición negra, pues el nodo eliminado puede ser sido negro, lo que provoca una pérdida en altura negra. En adelante llamaremos `p` a ese nodo eliminado, el cual puede tener un déficit en nodos negros.

`find_succ_and_swap()` intercambia `q` con su sucesor infijo. La rutina es estructuralmente idéntica a la versión para árboles AVL (§ 6.4.1.2 (Pág. 479)). El único punto a considerar es que si se elimina un nodo completo, entonces el sucesor -o predecesor- que se sustituye debe heredar el mismo color del nodo eliminado. De esta manera, la violación eventual ocurre sobre un nodo que con certeza es incompleto.

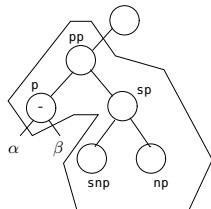


Figura 6.20: Zona de familiares donantes de nodo rojo

La eliminación estándar en un ABB siempre se remite a la eliminación de un nodo cuyo parente sea incompleto. La idea es efectuar un corto-circuito -§ 4.9.6 (Pág. 324)- hacia el hijo de `q` al cual denominamos `p`. Si `q` es rojo, entonces el árbol resultante es rojo-negro. El problema surge cuando `q` es negro, pues se viola la condición negra. En este caso diremos que existe un déficit de un nodo negro en el camino desde el parente del nodo eliminado hasta `p`. De manera general, la estrategia de restauración de la condición negra es encontrar un nodo rojo en las inmediaciones de `p` tal como se indica en la figura 6.20. Si se encuentra tal nodo, entonces éste puede pasarse hacia el lado del déficit y colorearlo de negro para así compensar el déficit. El trabajo es efectuado por la rutina `fix_black_condition()`, la cual toma como entrada un nodo `p` con un déficit en un nodo negro que debe ser restaurado. La salida es un árbol rojo-negro.

La estructura de la rutina es la siguiente:

```
499 ⟨miembros privados de Gen_Rb_Tree<Key> 492b⟩+≡ (492a) ▷ 494
 void fix_black_condition(Node * p)
 {
 if (COLOR(p) == RED) // ¿p es rojo?
 {
 // si ==> lo pintamos de negro y terminamos
 COLOR(p) = BLACK; // esto compensa el déficit
 return;
 }

 Node * pp = rb_stack.popn(2); // padre de p
 while (p != root)
 {
 Node * sp = LLINK(pp) == p ? RLINK(pp) : LLINK(pp); // hermano p
 if (COLOR(sp) == RED) // ¿hermano de p es rojo?
 ⟨rotar sp hacia el lado de p e intercambiar sus colores 500a⟩

 ⟨calcular los sobrinos de p (hijos de sp) 500b⟩

 if (COLOR(np) == RED) // ¿np es rojo?
 {
 ⟨rotar sp hacia p, heredar color pp y pintar negro a np y pp 501a⟩
 return;
 }
 if (COLOR(snp) == RED) // ¿snp es rojo?
 {
 ⟨rotar doble a snp y pintar de negro a pp 501b⟩;
 return;
 }
 if (COLOR(pp) == RED) // ¿pp es rojo?
 {
 ⟨intercambiar colores de p y pp 502⟩;
 return;
 }
 // no hay nodo rojo en las adyacencias de p ==> desplazar el
 // déficit hacia pp y repetir la iteración
 COLOR(sp) = RED;
 p = pp;
 pp = rb_stack.pop();
 }
 }
```

Defines:

`fix_black_condition`, used in chunk 497.

La rutina comienza por evaluar si el nodo deficitario es rojo; en ese caso basta con pintar a p de negro y el árbol general es rojo-negro; situación y corrección ilustradas en la figura 6.21.

En el caso contrario, el algoritmo entra en un ciclo que busca un nodo rojo en la zona enmarcada en la figura 6.20 (Pág. 498) para intentar sustituir al nodo negro y así compensar el déficit. La iteración examina los nodos de la zona en el orden sp, np, snp y pp, según los siguientes casos:

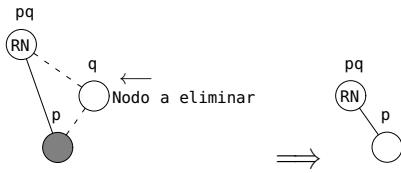


Figura 6.21: Eliminación de un nodo negro con hijo rojo

1. sp rojo: la situación y su corrección se muestran en la figura 6.22.

500a  $\langle \text{rotar sp hacia el lado de } p \text{ e intercambiar sus colores 500a} \rangle \equiv$  (499)

```

{
 Node *& ppp = rb_stack.top(); // abuelo de p

 if (LLINK(pp) == p)
 {
 sp = LLINK(sp);
 ppp = rotate_to_left(pp, ppp);
 }
 else
 {
 sp = RLINK(sp);
 ppp = rotate_to_right(pp, ppp);
 }

 COLOR(ppp) = BLACK;
 COLOR(pp) = RED;
}

```

Esta acción aumenta el nodo deficitario en un nivel, pero hace que el nuevo hermano de p sea negro y que entre en la zona de familiares donantes descrita en la figura 6.20 (Pág. 498).

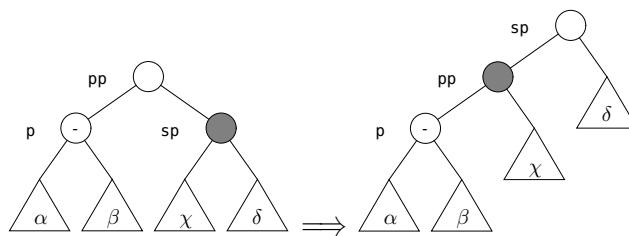


Figura 6.22: Caso cuando sp es rojo

Si sp es negro, entonces hay que revisar sus sobrinos, pero antes es menester calcularlos de la siguiente manera:

500b  $\langle \text{calcular los sobrinos de } p \text{ (hijos de sp) 500b} \rangle \equiv$  (499)

```

Node * np, * snp; // sobrinos de nodo p

if (LLINK(pp) == p) // ¿p es hijo izquierdo?

```

```

 {
 // si ==> que sp es hijo derechos
 np = RLINK(sp);
 snp = LLINK(sp);
 }
else
{
 np = LLINK(sp);
 snp = RLINK(sp);
}

```

2. np rojo: la situación se ilustra en la figura 6.23 y se identifica porque np está alineado hacia la derecha con sp. La solución consiste en rotar sp hacia el lado de p y colorear de negro np y pp. Con esto ganamos el nodo negro pp en el camino hacia p, lo cual compensa el déficit.

501a  $\langle\text{rotar } sp \text{ hacia } p, \text{ heredar color } pp \text{ y pintar negro a } np \text{ y } pp \text{ 501a}\rangle \equiv \quad (499)$

```

Node * ppp = rb_stack.top();

if (RLINK(sp) == np)
 rotate_to_left(pp, ppp);
else
 rotate_to_right(pp, ppp);

COLOR(sp) = COLOR(pp);
COLOR(pp) = BLACK;
COLOR(np) = BLACK;

```

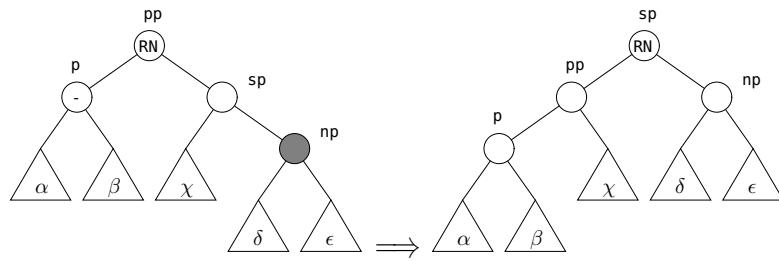


Figura 6.23: Caso cuando np es rojo

3. snp rojo: la situación se ilustra en la figura 6.24 (Pág. 502); es muy similar a la anterior excepto que no hay alineación completa hacia la derecha. La solución también es similar: rotar snp dos veces hacia el lado de p y luego colorear de negro pp y sp. Al igual que en el caso anterior, la ganancia del nodo negro pp en el camino hacia p compensa el déficit.

501b  $\langle\text{rotar doble a } snp \text{ y pintar de negro a } pp \text{ 501b}\rangle \equiv \quad (499)$

```

Node * ppp = rb_stack.top();

if (LLINK(sp) == snp)
{
 rotate_to_right(sp, pp);
}

```

```

 rotate_to_left(pp, ppp);
 }
else
{
 rotate_to_left(sp, pp);
 rotate_to_right(pp, ppp);
}
COLOR(snp) = COLOR(pp);
COLOR(pp) = BLACK;

```

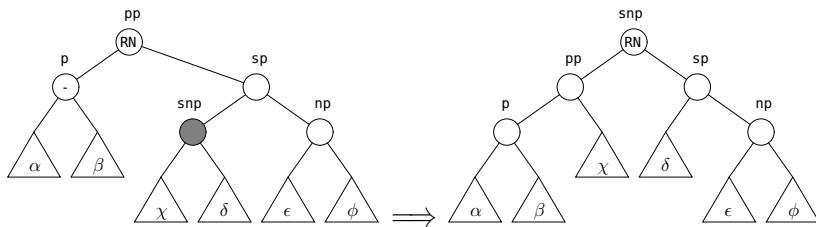


Figura 6.24: Caso cuando snp es rojo

4. pp rojo: la situación se muestra en la figura 6.25 (Pág. 502) y se resuelve intercambiando los colores de pp con sp. Después de este ajuste, el camino hacia p gana un nodo negro a través de su padre, mientras que el camino hacia sp permanece con la misma cantidad de nodos negros. Este ajuste sólo puede hacerse si se han evaluado los casos anteriores, pues se podría violar la condición roja si alguno de los sobrinos de p fuese rojo.

502       $\langle intercambiar\ colores\ de\ p\ y\ pp\ 502 \rangle \equiv$  (499)  
           COLOR(pp) = BLACK;  
           COLOR(sp) = RED;

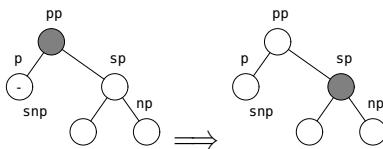


Figura 6.25: Caso cuando el padre de  $p$  es rojo

5. Ninguno de los familiares es rojo: si los casos anteriores fallan, entonces no hay nodos rojos en la zona enmarcada en la figura 6.20 (Pág. 498). Ante esta situación, la única salida es desplazar el déficit hacia el padre de  $p$  y luego repetir el procedimiento anterior con  $p = pp$ . El desplazamiento, ilustrado en la figura 6.26, consiste simplemente en colorear  $sp$  de rojo. Con esto eliminamos un nodo negro a partir de  $sp$  y trasladamos el déficit hacia  $pp$ .

El algoritmo itera si no se encuentran nodos rojos en la zona enmarcada en la figura 6.20 (Pág. 498). La cantidad máxima de iteraciones depende de si hay o no nodos rojos en las inmediaciones del camino de búsqueda a la clave eliminada. En el peor de los casos

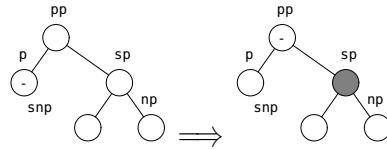


Figura 6.26: Desplazamiento del déficit hacia pp

puede ser necesario subir hasta la raíz. En esta situación, el déficit ocurre para todos los caminos desde la raíz hasta cualquier nodo nulo, por lo que no hay violación de la condición negra.

El camino más largo se da cuando la eliminación ocurre en una de las hojas más profundas en altura negra. Esto implica que del lado opuesto a p el árbol debe estar perfectamente balanceado, pues si no, la zona no sería completamente negra. El lema 6.7 nos mostrará que la altura de la zona completamente negra es  $\lfloor h/2 \rfloor$  para el peor árbol. La proposición 6.9 nos mostrará que la altura máxima es  $2 \lg(n+2) - 2$ . Así pues, la eliminación es  $\mathcal{O}(\lg n)$  para el peor caso.

### 6.5.2 Análisis de los árboles rojo-negro

Comencemos por recordar que la condición negra garantiza por si misma un balance a nivel de nodos negros. Si sólo se cuentan los nodos negros, entonces, para cada nodo, la diferencia de alturas entre las dos ramas es cero. Bajo la altura negra, el árbol está perfectamente equilibrado. El análisis se enfoca entonces en estudiar el impacto de los nodos rojos sobre la altura global del árbol.

Por la condición roja, la altura debe expresarse en función de nodos negros, pues no puede haber dos nodos rojos consecutivos. El número máximo de nodos rojos involucrados no puede exceder, pues, al número de nodos negros más uno. De manera intuitiva podemos decir que en el peor caso, la altura máxima de un árbol rojo-negro está determinada por la longitud máxima en nodos negros multiplicada por dos. Es decir, asumimos que el camino de la rama más larga está repleto de nodos rojos intercalados entre nodos negros.

Con estas observaciones en mente, estamos listos para enunciar la proposición siguiente.

**Proposición 6.9** (Guibas y Sedgewick - 1978) Sea  $T$  un árbol rojo-negro con  $n$  nodos y altura  $h$ . Entonces:

$$h \leq 2 \lg(n+2) - 2 \quad (6.31)$$

**Demostración** El enfoque es similar al utilizado en los árboles AVL (proposición 6.4). Es decir, vamos a determinar cuál es el árbol rojo-negro de altura  $h$  con el máximo número de nodos. Para eso debemos identificar un árbol rojo-negro crítico.

**Definición 6.9 (Árbol rojo-negro crítico)** Sea  $T$  un árbol rojo-negro de altura  $h(T)$ . Se dice que  $T$  es crítico si, y sólo si, al quitarle una hoja el árbol deja de ser rojo-negro o pierde altura.

Al igual que con los árboles AVL, la cardinalidad de un árbol rojo-negro crítico es mínima a la altura  $h(T)$ , es decir, si  $T$  es crítico, no es posible tener un árbol a la misma altura con menor cantidad de nodos.

Ahora procedemos a estudiar cuántos nodos tiene un árbol crítico en función de su altura. Primero comenzemos por observar las diferencias de altura en los nodos de un árbol crítico.

**Lema 6.4** Sea  $T$  un árbol rojo-negro crítico de altura  $h(T) > 1$  con ramas  $L(T)$  y  $R(T)$  respectivamente. Entonces  $h(L(T)) \neq h(R(T))$ .

**Demostración (por reducción al absurdo)** Supongamos que existe un árbol rojo-negro crítico con  $h(L(T)) = h(R(T))$ . Puesto que  $h(T) > 1$ , podemos reemplazar cualquiera de las ramas por otro árbol rojo-negro más pequeño en altura y cardinalidad, lo cual arrojaría un árbol rojo-negro de cardinalidad inferior. Esto causa una contradicción, pues hemos dicho que  $T$  es crítico. El lema es pues cierto  $\square$

Ahora veamos cuántos subárboles críticos hay dentro de un árbol rojo-negro crítico.

**Lema 6.5** Sea  $T$  un árbol rojo-negro crítico de altura  $h(T) > 1$ . Entonces,  $h(R(T)) = h(T) - 1 \implies R(T)$  es crítico y por lo tanto mínimo.

La situación simétrica, es decir, si  $h(L(T)) = h(T) - 1$ , es equivalente.

**Demostración (por reducción al absurdo)** Supongamos que  $T$  es crítico y que  $R(T)$  no lo es. Entonces,  $R(T)$  podría ser reemplazado por un subárbol  $T'$ ,  $h(T') = h(T) - 1$  tal que  $|T'| < |R(T)|$ . Esto implicaría que  $T$  tiene menos nodos a la misma altura, lo cual es una contradicción con la suposición de que  $T$  es crítico  $\square$

Con este lema podemos estudiar la composición cromática de cada uno de los subárboles de un árbol rojo-negro crítico.

**Lema 6.6** Sea  $T$  un árbol rojo-negro crítico de altura  $h(T) > 1$ . Entonces,  $h(R(T)) = h(T) - 1 \implies$ :

1.  $L(T)$  no tiene nodos rojos.
2.  $L(T)$  es un árbol perfectamente equilibrado y completo.

#### Demostración

1. Si  $L(T)$  tiene un nodo rojo, entonces este nodo puede ser suprimido y sustituido por el árbol vacío. El árbol resultante es rojo-negro pero tiene menos nodos, lo cual viola la suposición de que  $T$  es crítico.
2. Puesto que  $L(T)$  sólo contiene nodos negros, la única forma de preservar la condición negra es que  $L(T)$  sea perfectamente equilibrado y completo  $\square$

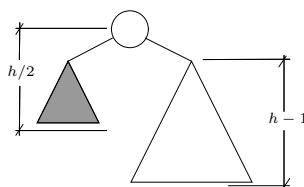


Figura 6.27: Forma de un árbol rojo-negro crítico

Este lema demuestra que la forma de un árbol rojo-negro crítico es la de la figura 6.27. Es decir, la rama de menor altura es enteramente negra y perfectamente balanceada. Note que el lema 6.6 también indica que la forma del subárbol derecho de la figura 6.27 es, en términos recursivos, similar a la figura 6.27. Esta observación es muy importante porque nos permitirá calcular una cota a la cantidad de nodos de nodos que tiene un árbol crítico en función de su altura.

Si  $T$  es rojo-negro, entonces  $bh(L(T)) = bh(R(T))$ . Si, además,  $T$  es crítico, entonces  $bh(R(T)) = h(L(T))$ , pues, por el lema 6.6,  $L(T)$  sólo tiene nodos negros. Por el lema 6.5 sabemos que  $h(R(T)) = h(T) - 1$ . Del mismo modo, también por el lema 6.5, si  $R(T)$  es crítico, entonces éste debe tener la forma demostrada por el lema 6.6.

**Lema 6.7** Sea  $T$  un árbol rojo-negro crítico de altura  $h(T) = h$ . Entonces,

$$bh(T) = \left\lfloor \frac{h}{2} \right\rfloor \quad (6.32)$$

**Demostración (por inducción sobre la altura  $h$ )**

- $h = 0$ : en este caso  $\lfloor 0/2 \rfloor = 0$  y la hipótesis de base es válida.
- $h > 0$ : ahora suponemos que el lema es válido para todo árbol  $T_h$  rojo-negro crítico de altura  $h$  y verificamos si aún lo es para el árbol  $T_{h+1}$  de altura  $h+1$ . La demostración dependerá de que  $h$  sea par o impar:
  1. Si  $h$  es impar  $\Rightarrow h+1$  es par. En este caso, si coloreamos de negro la raíz de  $T_{h+1}$  tenemos:

$$\begin{aligned} bh(T_{h+1}) &= bh(T_h) + 1 \Rightarrow \\ &= \left\lfloor \frac{h}{2} \right\rfloor + 1 = \frac{h-1}{2} + 1 \Rightarrow \\ &= \frac{h+1}{2} = \left\lfloor \frac{h+1}{2} \right\rfloor \end{aligned} \quad (6.33)$$

El lema es válido para  $h$  impar.

2. Si  $h$  es par  $\Rightarrow h+1$  es impar. En este caso, si coloreamos de rojo la raíz de  $T_{h+1}$  tenemos:

$$\begin{aligned} bh(T_{h+1}) &= bh(T_h) \Rightarrow \\ &= \frac{h}{2} = \left\lfloor \frac{h+1}{2} \right\rfloor \end{aligned} \quad (6.34)$$

El lema es válido para  $h$ , par o impar  $\square$

La demostración del lema 6.7 hace aprensible el hecho de que el color de la raíz de un árbol rojo-negro puede ser negro solamente si la altura del árbol es par.

El lema 6.7 permite completar la prueba del teorema porque ahora podemos calcular una expresión que indique la cantidad total de nodos de un árbol rojo-negro crítico.

Si  $T_h$  es un árbol rojo-negro crítico de altura  $h$ , entonces:

$$|T_h| = 1 + |L(T_h)| + |R(T_h)| \quad (6.35)$$

Por los lemas 6.6 y 6.7 sabemos que  $L(T_h)$  sólo tiene nodos negros, por ende:

$$|L(T_h)| = 2^{\lfloor(h-1)/2\rfloor} - 1 \quad (6.36)$$

Por otra parte, por el lema 6.5, sabemos que  $R(T_h)$  es crítico y que su altura es  $h-1$ . Sustituimos (6.36) en (6.35) y obtenemos una ecuación recurrente fácilmente resoluble:

$$|T_h| = |T_{h-1}| + 2^{\lfloor(h-1)/2\rfloor} \quad (6.37)$$

la cual, al expandir  $|T_{h-1}|$ :

$$|T_h| = |T_{h-2}| + 2^{\lfloor(h-2)/2\rfloor} + 2^{\lfloor(h-1)/2\rfloor} \quad (6.38)$$

- Si  $h$  es par  $\Rightarrow$

$$|T_h| = |T_{h-2}| + 2^{(h-2)/2} + 2^{(h-2)/2} = |T_{h-2}| + 2^{h/2} \Rightarrow$$

$$|T_h| = \sum_{i=1}^{h/2} 2^i = 2^{h/2+1} - 2 \quad (6.39)$$

- Si  $h$  es impar  $\Rightarrow$

$$|T_h| = |T_{h-2}| + 2^{(h-3)/2} + 2^{(h-1)/2} \Rightarrow$$

$$|T_h| = |T_{h-2}| + 2^{-1} \cdot 2^{(h-1)/2} + 2^{(h-1)/2} \Rightarrow$$

$$|T_h| = |T_{h-2}| + 3 \cdot 2^{(h-3)/2} \Rightarrow$$

$$|T_h| = |T_{h-2}| + 3 \cdot \sum_{i=0}^{(h-3)/2} 2^i = 1 + 3 \cdot 2^{(h-3)/2} - 3 = 3 \cdot 2^{(h-3)/2} - 2 \quad (6.40)$$

Para completar la demostración, sea  $n$  el número de nodos de un árbol rojo-negro de altura  $h$ .  $n$  debe ser mayor o igual al número de nodos del árbol rojo-negro crítico de altura  $h$ . Tomando el menor término derecho de las ecuaciones (6.39) y (6.40) planteamos:

$$\begin{aligned} n &\geq 2^{h/2+1} - 2 \Rightarrow \\ n + 2 &\geq 2^{h/2+1} \Rightarrow \\ \lg(n + 2) &\geq \frac{h}{2} + 1 \Rightarrow \\ h &\leq 2 \lg(n + 2) - 2 \blacksquare \end{aligned}$$

La máxima altura de un árbol rojo-negro es aproximadamente un 28% más grande que la máxima altura de un árbol AVL ( $2 \lg(n + 2)$  versus  $1.4404 \lg(n + 2)$ ). Así pues, analíticamente, el desempeño de la búsqueda debería ser superior en los árboles AVL que en los árboles rojo-negro. Empero, esta diferencia no es significativa. El desempeño de ambos árboles es muy similar.

## 6.6 Árboles splay

Los árboles aleatorizados y los treaps basan sus equilibrios sobre la toma de decisiones aleatorias. En promedio, la altura de estos árboles deviene logarítmica, con probabilidades muy escazas de tener un mal caso. Los árboles AVL y rojo-negro basan sus equilibrios en reglas especiales que se mantienen en tiempos logarítmicos. En esta clase de árboles, la altura máxima y, por ende, el tiempo de acceso, están logarítmicamente acotados. Existe una tercera alternativa de equilibrio basada en efectuar modificaciones estructurales sobre el árbol, que “tiendan” a volverlo equilibrado. Tal alternativa es el objeto de estudios de esta sección: los árboles splay.

Consideremos las tres operaciones clásicas sobre una tabla de símbolos y un arreglo que almacena los elementos desordenadamente. Como vimos en § 2.1.1 (Pág. 29), en un arreglo, las tres operaciones son  $\mathcal{O}(n)$ . Ahora consideremos una modificación sobre la implantación tradicional: cada vez que se efectúa un acceso, sea por búsqueda, inserción o eliminación, el elemento es ubicado en la primera posición. Bajo esta regla, la inserción y eliminación devienen  $\mathcal{O}(1)$  para todos los casos y la búsqueda “ $\mathcal{O}(n)$ ” para el peor caso. Si la aplicación exhibe localidad de referencia, entonces la búsqueda tiende a ser  $\mathcal{O}(1)$ , pues los elementos recientemente accedidos se encuentran en las primeras posiciones del arreglo. ¿Qué tan buena es esta técnica? Los análisis formales y empíricos demuestran que es muy buena en muchos casos. Cuando no hay localidad de referencia, la búsqueda es  $\mathcal{O}(n)$  para los casos promedio y peor, un desempeño aceptable hasta escalas medianas.

¿Se puede aplicar la técnica anterior para los árboles binarios de búsqueda? La respuesta es afirmativa y la primera tentación es mover el nodo recientemente accedido hasta la raíz. La inserción consistiría en insertar en la raíz según los algoritmos estudiados en § 4.9.7 (Pág. 326). Del mismo modo, podríamos diseñar una rutina similar al `select()` estudiado en § 4.11.1 (Pág. 337) que busque una clave y mediante rotaciones sucesivas suba el nodo encontrado hasta la raíz. Al igual que con los arreglos, esta técnica ha probado tener un desempeño bueno para aplicaciones que exhiban localidad de referencia. Lamentablemente, puesto que la técnica no lleva a cabo ningún ajuste de equilibrio sobre el árbol, éste, probablemente, no será muy equilibrado. Peor aún, si los accesos son sesgados hacia un extremo del orden de las claves, el árbol puede devenir severamente desequilibrado.

Sleator y Tarjan [160] descubrieron una técnica de ajuste que garantiza que el costo promedio de  $n$  operaciones es  $\mathcal{O}(\lg n)$ . Su técnica es llamada “splaying” y los árboles resultantes son denominados “árboles splay”<sup>4</sup>. Básicamente, el “splaying” consiste en llevar un nodo accedido hasta la raíz mediante operaciones especiales que favorecerán futuros accesos.

La primera de las operaciones se llama zig-zig y se ilustra en la figura 6.28. Vista de izquierda a derecha, la operación se llama `zig_zig_right`, mientras que en el sentido contrario se llama `zig_zig_left`. La operación sube el nodo A tres niveles y se define como sigue:

<sup>4</sup>En castellano no existe un término que adecuadamente refleje la palabra “splay”. Encontramos en “plegado” y en “desplegado” las aproximaciones más representativas, pero, a nuestro juicio, “árboles desplegados” o “árboles plegados” no expresan los mismos significados que “splay trees”. Por esa razón preferimos utilizar el término original.

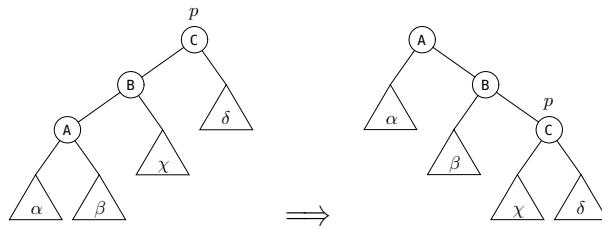


Figura 6.28: Operación zig-zig

```
static Node * zig_zig_right(Node* p)
{
 Node * q = LLINK(p);
 Node * r = LLINK(q);
 LLINK(p) = RLINK(q);
 LLINK(q) = RLINK(r);
 RLINK(q) = p;
 RLINK(r) = q;
 return r;
}
```

La operación simétrica es equivalente a observar la figura 6.28 de derecha a izquierda y se llama `zig_zig_left()`, la cual se instrumenta simétricamente similar.

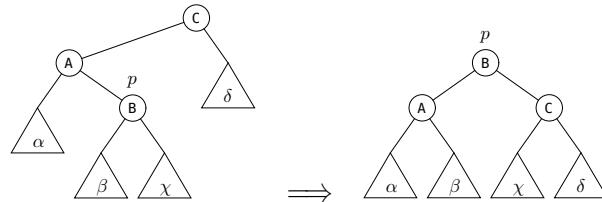


Figura 6.29: Operación zig-zag

La segunda operación se llama `zig-zag` y se ilustra en la figura 6.29. Esta operación es equivalente a una doble rotación. Al igual que el `zig-zig`, el nodo `p` sube tres niveles y se define como sigue:

508

`(operaciones zig 507) +≡` (509a) ↳ 507

```
static Node * zig_zag_right(Node* p)
{
 LLINK(p) = rotate_to_left(LLINK(p));
 return rotate_to_right(p);
}
```

El equivalente simétrico se denomina `zig_zag_left()`.

La última operación, que se llama “`zig`”, consiste en efectuar una rotación simple y sólo se lleva a cabo si el nodo que devendrá raíz se encuentra en el nivel 1, es decir, exactamente un nivel inferior a la raíz<sup>5</sup>.

Dado un nodo `x` en un árbol binario de búsqueda, el “`splaying`” del nodo `x` consiste en subirlo hasta la raíz mediante las operaciones `zig`, `zig-zig` y `zig-zag`. Los análisis posteriores

<sup>5</sup>Véase § 4.50 (Pág. 344).

demonstrarán que si esto se efectúa por cada acceso, entonces el costo promedio de  $n$  accesos es  $\mathcal{O}(\lg n)$ .

### 6.6.1 El TAD Splay\_Tree<Key>

El TAD Gen\_Splay\_Tree<Key> modeliza un árbol binario de búsqueda en el que cualquiera de las operaciones causa un splaying del nodo accedido. El TAD se define en el archivo *<tpl\_splay\_tree.H 509a>*, el cual se define como sigue:

509a *<tpl\_splay\_tree.H 509a>*≡  
 template <template <typename> class NodeType, typename Key, class Compare>  
 class Gen\_Splay\_Tree  
 {  
 typedef NodeType<Key> Node;  
  
*<operaciones zig 507>*  
*<miembros privados de Gen\_Splay\_Tree<Key> 509b>*  
*<miembros públicos de Gen\_Splay\_Tree<Key> 512a>*  
};  
 template <typename Key, class Compare = Aleph::less<Key> >  
 struct Splay\_Tree : public Gen\_Splay\_Tree<BinNode, Key, Compare> {};  
Uses BinNode 240.

El TAD Gen\_Splay\_Tree<Key> no requiere definir una nueva clase de nodo binario, pues no es necesario almacenar información de estado en cada nodo.

Las operaciones zig-zig y zig-zag involucran 4 nodos que son examinados bajo dos perspectivas. Primero requerimos determinar cuál de las seis operaciones posibles debe ejecutarse. Segundo, debemos ejecutar la operación en cuestión. El splaying de un nodo involucra entonces a todos los nodos del camino de búsqueda desde la raíz hasta el nodo donde se realiza el splaying. Por tanto, usamos una pila que almacene el camino íntegro de búsqueda:

509b *<miembros privados de Gen\_Splay\_Tree<Key> 509b>*≡ (509a) 509c▷  
 struct Node\_Desc  
{  
 Node \* node;  
 Node \*\* node\_parent;  
};  
 ArrayStack<Node\_Desc, Node::MaxHeight> splay\_stack;  
Uses ArrayStack 101a.

Cada entrada de la pila guarda registros de tipo Node\_Desc, el cual a su vez guarda dos valores: el nodo parte del camino de búsqueda y un doble apuntador a su parente. El puntero doble permite efectuar las rotaciones sin necesidad de preguntar de cuál lado, izquierdo o derecho, se encuentra el parente.

La acción de splay tiende a equilibrar el árbol, pero no excluye un caso muy desafortunado. Eventualmente, splay\_stack puede tener un desborde que debe ser manejado. Por esa razón definimos un método especial que inserta un nodo en la pila y que verifica el desborde:

509c *<miembros privados de Gen\_Splay\_Tree<Key> 509b>*+≡ (509a) ↳509b 510a▷  
 void push\_in\_splay\_stack(Node \* p, Node \*\* pp)  
{

```

try { splay_stack.push(Node_Desc(p, pp)); }
catch (std::overflow_error)
{
 if (splay_stack.empty())
 throw;
}
}

```

Defines:

`push_in_splay_stack`, used in chunks 510-12.

La única responsabilidad de `push_in_splay_stack()` es atrapar la excepción `std::overflow_error`. Si esto ocurre, lo único que se hace es vaciar la pila antes de propagar la excepción.

`push_in_splay_stack()` es utilizada cada vez que se efectúe una búsqueda, la cual es efectuada por la rutina siguiente:

510a *(miembros privados de Gen\_Splay\_Tree<Key> 509b) +≡ (509a) <509c 510b>*

```

void search_and_push_in_splay_stack(const Key & key)
{
 if (splay_stack.empty())
 Node ** pp = &RLINK(head);
 Node * p = root;
 while (p != NULL) // buscar iterativamente key
 {
 push_in_splay_stack(p, pp);
 if (Compare () (key, KEY(p)))
 {
 pp = &LLINK(p);
 p = LLINK(p);
 }
 else if (Compare() (KEY(p), key))
 {
 pp = &RLINK(p);
 p = RLINK(p);
 }
 else
 return;
 }
}

```

Defines:

`search_and_push_in_splay_stack`, used in chunks 512 and 513.

Uses `push_in_splay_stack` 509c.

`search_and_push_in_splay_stack()` es una rutina muy importante, la cual es indispensable comprender. Básicamente, la rutina busca un nodo con clave `key` y empila el camino de búsqueda independientemente de que se encuentre o no la clave. Si la búsqueda es exitosa, entonces el tope de la pila contiene el nodo encontrado. De lo contrario, el tope contiene el último nodo antes del externo detuvo la búsqueda. La rutina se emplea para las tres operaciones básicas: inserción, búsqueda y eliminación.

Los atributos restantes ya son familiares al lector:

510b *(miembros privados de Gen\_Splay\_Tree<Key> 509b) +≡ (509a) <510a 511a>*

```

Node head_node;

```

```
Node *head;
Node *&root;
```

La esencia de la implantación subyace en una rutina `splay()` que efectúa el splaying del nodo situado en el tope de la pila. `splay()` requiere determinar el tipo operación ejecutar según la alineación del camino de búsqueda. Esto es llevado a cabo por la siguiente rutina:

```
511a <miembros privados de Gen_Splay_Tree<Key> 509b>+≡ (509a) ◁510b 511b>
enum Zig_Type { ZIG_LEFT, ZIG_RIGHT, ZIG_ZAG_LEFT, ZIG_ZAG_RIGHT,
 ZIG_ZIG_LEFT, ZIG_ZIG_RIGHT };
Zig_Type zig_type(Node *& p, Node **& pp)
{
 Node_Desc first = splay_stack.pop();
 Node_Desc second = splay_stack.pop();
 Zig_Type ret = RLINK(second.node) == first.node ? ZIG_LEFT : ZIG_RIGHT;

 if (splay_stack.is_empty())
 {
 IG(LLINK(second.node) == first.node, ret == ZIG_RIGHT);
 p = second.node;
 pp = second.node_parent;
 return ret;
 }
 Node_Desc third = splay_stack.pop();
 pp = third.node_parent;
 p = third.node;
 if (ret == ZIG_LEFT)
 return RLINK(third.node) == second.node ? ZIG_ZIG_LEFT : ZIG_ZAG_RIGHT;
 else
 return LLINK(third.node) == second.node ? ZIG_ZIG_RIGHT : ZIG_ZAG_LEFT;
}
```

Defines:

`zig_type`, used in chunk 511b.

`zig_type()` retorna el tipo de operación que debe hacerse sobre el nodo en el tope de la pila. Adicionalmente, la rutina devuelve los valores `p` y `pp` correspondientes al último nodo en la cadena de la operación y su parent. Estamos listos para la rutina `splay()`, la cual es muy sencilla una vez definida la maquinaria necesaria:

```
511b <miembros privados de Gen_Splay_Tree<Key> 509b>+≡ (509a) ◁511a
void splay()
{
 if (splay_stack.is_empty() or splay_stack.top().node == root)
 return;

 Node **pp, *p;
 while (true)
 {
 switch (zig_type(p, pp))
 {
 case ZIG_LEFT: *pp = rotate_to_left(p); break;
 case ZIG_RIGHT: *pp = rotate_to_right(p); break;
 case ZIG_ZIG_LEFT: *pp = zig_zig_left(p); break;
 case ZIG_ZIG_RIGHT: *pp = zig_zig_right(p); break;
 }
 }
}
```

```

 case ZIG_ZAG_LEFT: *pp = zig_zag_left(p); break;
 case ZIG_ZAG_RIGHT: *pp = zig_zag_right(p); break;
 }
 if (splay_stack.is_empty())
 break;
 push_in_splay_stack(*pp, pp);
}
}

```

Defines:

splay, used in chunks 512 and 513.

Uses push\_in\_splay\_stack 509c and zig\_type 511a.

splay() es la rutina fundamental del TAD Gen\_Splay\_Tree<Key>. Ella asume que se efectuó una llamada previa a search\_and\_push\_in\_splay\_stack() y que el tope de la pila contiene el nodo sobre el cual se desea efectuar el splay. Cada vez que se ejecuta cualquier operación de inserción, búsqueda o eliminación, se debe hacer un splay del árbol. La idea de esta operación es amortizar los casos desafortunados y equilibrar el árbol.

De las tres operaciones fundamentales, la más simple es la de búsqueda, la cual se define como sigue:

512a <miembros públicos de Gen\_Splay\_Tree<Key> 512a>≡ (509a) 512b▷

```

Node * search(const Key & key)
{
 if (root == NULL)
 return NULL;

 search_and_push_in_splay_stack(key);

 splay();

 if (are_equals<Key, Compare>(key, KEY(root)))
 return root;

 return NULL;
}

```

Uses search\_and\_push\_in\_splay\_stack 510a and splay 511b.

La búsqueda es muy sencilla porque toda la infraestructura se encuentra implantada por los métodos privados search\_and\_push\_in\_splay\_stack() y splay(). Note que el splay siempre se efectúa, independientemente de que la búsqueda sea fallida o no.

La inserción es ligeramente más complicada pero al igual que con la búsqueda todo el trabajo es implantado por search\_and\_push\_in\_splay\_stack() y splay():

512b <miembros públicos de Gen\_Splay\_Tree<Key> 512a>+≡ (509a) ◁512a 513▷

```

Node * insert(Node * p)
{
 if (root == NULL)
 return root = p;

 search_and_push_in_splay_stack(KEY(p));
 Node * pp = splay_stack.top().node;
 if (Compare () (KEY(p), KEY(pp)))
 {

```

```

 LLINK(pp) = p;
 push_in_splay_stack(p, &LLINK(pp));
}
else if (Compare () (KEY(pp), KEY(p)))
{
 RLINK(pp) = p;
 push_in_splay_stack(p, &RLINK(pp));
}
else
{
 // clave duplicada
 splay(); // de todos modos hacemos el splay
 return NULL;
}
splay();

return root;
}

```

Uses `push_in_splay_stack` 509c, `search_and_push_in_splay_stack` 510a, and `splay` 511b.

Para la eliminación se tienen dos alternativas: la eliminación clásica que ante un nodo completo selecciona el sucesor o predecesor, o mediante la operación `join()` desarrollada en § 4.9.8 (Pág. 327). Por ser la más sencilla y posiblemente la más eficaz, escogemos la segunda alternativa:

513     ⟨miembros públicos de Gen\_Splay\_Tree<Key> 512a⟩+≡ (509a) □ 512b

```

Node * remove(const Key & key)
{
 search_and_push_in_splay_stack(key);

 splay(); // Suba el nodo hasta la raíz mediante el splay

 if (no_equals <Key, Compare> (KEY(root), key))
 return NULL; // key no está

 Node * p = root;
 root = join_exclusive(LLINK(root), RLINK(root));

 p->reset();
}

```

Uses join\_exclusive 324, search\_and\_push\_in\_splay\_stack 510a, and splay 511b.

El algoritmo efectúa la búsqueda del nodo a eliminar y hace un splay independientemente de que la búsqueda haya sido exitosa o no. Si la clave se encuentra en el árbol, entonces la raíz contiene el nodo a eliminar y el árbol resultante es la concatenación de sus ramas. En caso contrario, el algoritmo termina, pues no es necesaria otra operación.

### 6.6.2 Análisis de los árboles splay

El análisis de un árbol splay es paradigmático del amortizado.

Todas las operaciones llevan a cabo una búsqueda, cuyo desempeño depende de la profundidad del nodo; posteriormente, se hace el splay del nodo, cuyo desempeño también depende de la longitud del camino entre la raíz y el nodo. La inserción y la búsqueda tienen  $\mathcal{O}(\text{nivel}(n_i)) + \mathcal{O}(\text{nivel}(n_i)) = \mathcal{O}(\text{nivel}(n_i))$ , mientras que la eliminación tiene  $\mathcal{O}(\text{nivel}(n_i)) + \mathcal{O}(\text{nivel}(n_i)) + \mathcal{O}(\text{nivel}(n_i)) = \mathcal{O}(\text{nivel}(n_i))$ , siendo  $\text{nivel}(n_i)$  el nivel del nodo sujeto a la operación splay. Esto nos sugiere que la función potencial deba considerar el nivel de los nodos como parte del coste.

La longitud del camino interno, presentada en la definición § 4.5 (Pág. 277), nos arroja una métrica que considera el nivel. Ahora bien, el carácter de emulación a la búsqueda binaria, así como las cinco clases anteriores de árboles binarios que hemos estudiado, sugieren que consideremos al logaritmo en la función potencial que nos pondere el nivel.

**Definición 6.10 (Rango de un árbol)** Sea un árbol binario  $T$  de  $n$  nodos. Sea  $C(n_i)$  la cantidad de nodos del sub-árbol con raíz en  $n_i$ . Entonces, el “rango de  $T$ ”, denotado como  $r(T)$ , se define como:

$$r(T) = \lg C(T) \quad (6.41)$$

Sleator y Tarjan [160] seleccionaron una función potencial basada en el rango de un árbol del siguiente modo:

$$\Phi(T) = \sum_{\substack{\forall n_i \in T \\ n_i \text{ es interno}}} r(n_i) = \sum_{\substack{\forall n_i \in T \\ n_i \text{ es interno}}} \lg C(n_i) \quad (6.42)$$

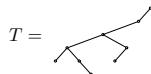
En cierta forma, este potencial es reminiscente al  $IPL(T)$  en el sentido de que pondera cuán equilibrado está  $T$ . La diferencia consiste en que  $\Phi(T)$  está acotado por  $\mathcal{O}(n \lg(n))$ , mientras que el  $IPL(T)$  por  $\mathcal{O}(n^2)$ .

Para un árbol completo, el rango es pequeño, por ejemplo, para:



$$\Phi(T) = \lg(9) + \lg(5) + 2\lg(3) + 5\lg(1) \approx 8,66 ,$$

mientras que con un árbol más desequilibrado, el rango es mayor; por ejemplo, para:



$$\Phi(T) = \lg(9) + \lg(8) + \lg(7) + \lg(4) + 2\lg(2) + 3\lg(1) \approx 17,62 .$$

Cuanto más equilibrado esté un árbol, menor es su potencial; análogamente, cuanto menos equilibrado, mayor éste es.

**Proposición 6.10 (Lema de acceso a un árbol splay - Sleator y Tarjan 1985 [160])**

Sea  $T$  un árbol splay y  $k \in T$  una clave. Sean  $C_{i-1}(k)$  y  $C_i(k)$  las cardinalidades del sub-árbol cuya raíz es  $k$  antes y después de un paso de un splay de un total de  $m$  pasos. Sean  $r_{i-1}(k)$  y  $r_i(k)$  los rangos de  $k$  antes y después de un paso del splay de un total de  $m$  pasos. Entonces, el coste amortizado en el  $i$ -ésimo paso, definido como  $\hat{c}_i$ , está acotado como:

$$\hat{c}_i \leq \begin{cases} 3r_i(k) - 3r_{i-1}(k) & \text{si } i < m \\ 3r_i(k) - 3r_{i-1}(k) + \mathcal{O}(1) & \text{si } i = m \end{cases} \quad (6.43)$$

**Demostración** La demostración consiste en calcular el coste amortizado para cada tipo de operación que ocurre durante un paso del splay entre los  $m$  requeridos para subir  $k$  hasta la raíz. De este modo:

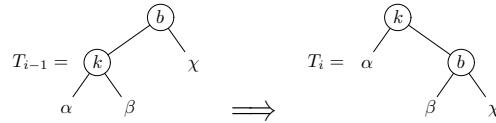
$$\hat{c}_i = t_i + \Phi(T_i) - \Phi_{i-1}(T_{i-1}) \quad (6.44)$$

Donde  $T_{i-1}$  y  $T_i$  son los árboles antes y después de un paso en el splay entre los tres posibles tipos definidos. Sea  $k$  el nodo que subirá de nivel en uno de los pasos del splay, entonces, a efectos de esta demostración, plantearemos una presentación menos precisa del lema de acceso

$$\hat{c}_i \leq 3r_i(k) - 3r_{i-1}(k) + \mathcal{O}(1), \quad 1 \leq i \leq m \quad (6.45)$$

En lo que sigue trataremos los tres posibles pasos que ocurren en un splay:

1. zig:



En este caso, la duración constante equivale a una rotación, razón por la cual asumimos  $t_i = 1$ . El zig no altera las cardinalidades de las ramas  $\alpha$ ,  $\beta$  y  $\chi$ , por lo que sus rangos se cancelan en la diferencia de potencial. Por lo tanto, la diferencia de potencial sólo contiene a los rangos de los nodos  $k$  y  $b$ . De este modo:

$$\hat{c}_i = 1 + r_i(k) + r_i(b) - r_{i-1}(k) - r_{i-1}(b) \quad (6.46)$$

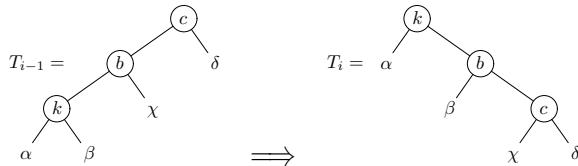
Notemos, por observación de la figura del zig, que  $C_{i-1}(b) = C_i(k)$ , por lo que  $r_i(k) - r_{i-1}(b) = 0$ , lo que nos deja:

$$\hat{c}_i = 1 + r_i(b) - r_{i-1}(k) \quad (6.47)$$

Del mismo modo,  $r_{i-1}(b) \geq r_i(k)$ , por lo que podemos plantear la desigualdad siguiente:

$$\begin{aligned} \hat{c}_i &\leq 1 + r_i(k) - r_{i-1}(k) \\ &\leq 3r_i(k) - 3r_{i-1}(k) + \mathcal{O}(1) \quad \square \end{aligned} \quad (6.48)$$

2. zig-zig:



En este caso tampoco cambian las cardinalidades de  $\alpha$ ,  $\beta$ ,  $\chi$  y  $\delta$ , por lo que los rangos permanecen iguales entre  $T_{i-1}$  y  $T_i$  y se cancelan mutuamente en el cálculo del diferencial potencial. Por otra parte, el zig-zig es más costoso en duración que el zig. En complejidad de tiempo se puede decir que el zig-zig efectúa una rotación más, la cual

es proporcional a 2, o sea, el doble del zig. Por lo tanto, el coste amortizado se define como:

$$\hat{c}_i = 2 + r_i(k) + r_i(b) + r_i(c) - r_{i-1}(k) - r_{i-1}(b) - r_{i-1}(c)$$

En este caso,  $C_{i-1}(c) = C_i(k)$ , lo que hace que  $r_{i-1}(c) - r_i(k) = 0$ , por lo que la expresión anterior deviene en:

$$\hat{c}_i = 2 + r_i(b) + r_i(c) - r_{i-1}(k) - r_{i-1}(b) \quad (6.49)$$

Igual que para el zig,  $r_{i-1}(b) \geq r_{i-1}(k)$  y  $r_i(k) \geq r_i(b)$ . Substituimos en la ecuación anterior y la hacemos una desigualdad:

$$\hat{c}_i \leq 2 + r_i(k) + r_i(c) - 2r_{i-1}(k) \quad (6.50)$$

Sean:

$$x = \frac{C_{i-1}(k)}{C_i(k)}, \quad y = \frac{C_i(c)}{C_i(k)} \quad (6.51)$$

Puesto que  $C_{i-1}(k) > 0$ ,  $C_i(c) > 0$  y  $C_i(k) > 0$ , entonces  $x > 0$  e  $y > 0$ . Más aún, puesto que  $C_i(k) > C_{i-1}(k)$  y  $C_i(k) > C_i(c)$ , entonces  $x < 1$  e  $y < 1$ . Por añadidura,  $x + y < 1$ , lo cual se evidencia expresando las cardinalidades en función de sus sub-árboles:

$$x + y = \frac{1 + C(\alpha) + C(\beta) + C(\chi) + C(\delta)}{3 + C(\alpha) + C(\beta) + C(\chi) + C(\delta)} = \frac{2 + C(\alpha) + C(\beta) + C(\chi) + C(\delta)}{3 + C(\alpha) + C(\beta) + C(\chi) + C(\delta)} \quad (6.52)$$

¿Cómo se acotará  $\lg(x) + \lg(y)$ ? o, dicho de otro modo, ¿cuál será el máximo valor que tomará  $\lg(x) + \lg(y)$ ? La respuesta se trata en el siguiente lema:

**Lema 6.8** Sean  $x, y \in \mathcal{R} \mid x > 0, y > 0, x + y \leq 1$ , entonces,  $\lg(x) + \lg(y) \leq -2$ .

**Demostración** Por una propiedad ya conocida,  $\lg(x) + \lg(y) = \lg(xy)$ . La curva del logaritmo es monótonamente creciente y, dentro del dominio de nuestro interés, exhibe la forma de la figura 6.30. El valor de  $\lg(xy)$  es máximo cuando el producto  $xy$  es máximo. Dadas las condiciones de  $x$  e  $y$ ,  $xy$  es máximo cuando  $x = y = 1/2 \Rightarrow \lg(xy) = -2 \quad \square$

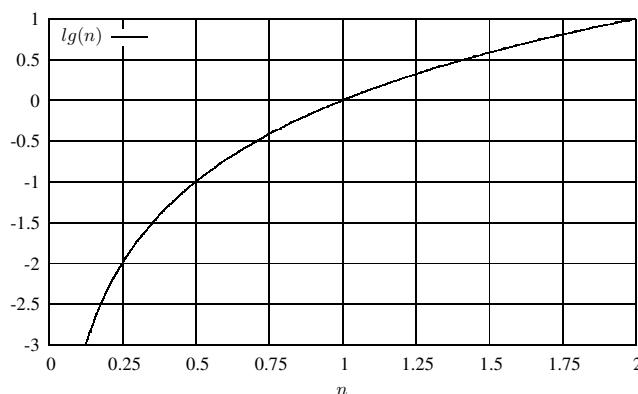


Figura 6.30: Curva  $\lg n$  para valores menores que 1

Ahora expandimos  $\lg(x) + \lg(y)$ :

$$\begin{aligned}
 \lg(x) + \lg(y) &= \lg \frac{C_{i-1}(k)}{C_i(k)} + \lg \frac{C_i(c)}{C_i(k)} \\
 &= \lg C_{i-1}(k) - \lg C_i(k) + \lg C_i(c) - \lg C_i(k) \\
 &= \lg C_{i-1}(k) + \lg C_i(c) - 2\lg C_i(k) \\
 &= r_{i-1}(k) + r_i(c) - 2r_i(k)
 \end{aligned} \tag{6.53}$$

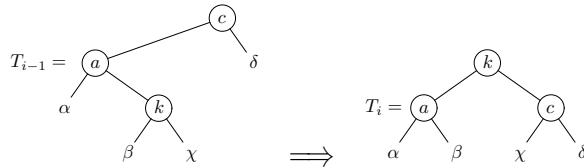
En función del lema 6.8 planteamos la siguiente desigualdad:

$$\begin{aligned}
 r_{i-1}(k) + r_i(c) - 2r_i(k) &\leq -2 \Rightarrow \\
 2r_i(k) - r_i(c) - r_{i-1}(k) - 2 &\geq 0
 \end{aligned} \tag{6.54}$$

Como el resultado es positivo lo podemos sumar a la desigualdad (6.50):

$$\begin{aligned}
 \hat{c}_i &\leq 2 + r_i(k) + r_i(c) - 2r_{i-1}(k) + 2r_i(k) - r_i(c) - r_{i-1}(k) - 2 \\
 &\leq 3r_i(k) - 3r_{i-1}(k) \\
 &\leq 3r_i(k) - 3r_{i-1}(k) + \mathcal{O}(1) \quad \square
 \end{aligned} \tag{6.55}$$

### 3. zig-zag:



El razonamiento en este caso es completamente similar al anterior hasta (6.49).

Observando que  $r_{i-1}(a) \geq r_{i-1}(k)$ , y que  $r_i(k) \geq r_i(c)$ , planteamos la siguiente desigualdad:

$$\hat{c}_i \leq 2 + r_i(a) + r_i(c) - 2r_{i-1}(k) \tag{6.56}$$

Sean:

$$x = \frac{C_i(a)}{C_i(k)}, \quad y = \frac{C_i(c)}{C_i(k)} \tag{6.57}$$

En este momento debe ser claro que  $x + y < 1$ , razón por la cual, aplicando el lema 6.8 tenemos que:

$$\begin{aligned}
 \lg x + \lg y &\leq -2 \Rightarrow \\
 \lg \frac{C_i(a)}{C_i(k)} + \lg \frac{C_i(c)}{C_i(k)} &\leq -2 \Rightarrow \\
 2r_i(k) - r_i(a) - r_i(c) - 2 &\geq 0
 \end{aligned} \tag{6.58}$$

Finalmente, sumando el lado izquierdo de (6.58) al derecho de (6.56), tenemos:

$$\begin{aligned}
 \hat{c}_i &\leq 2r_i(k) - 2r_{i-1}(k) \\
 &\leq 3r_i(k) - 3r_{i-1}(k) + \mathcal{O}(1) \quad \square
 \end{aligned} \tag{6.59}$$

Los tres casos posibles de un paso del splay están acotados por  $3r_i(k) - 3r_{i-1}(k) + \mathcal{O}(1)$

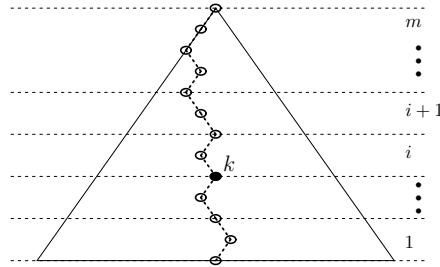
■ El lema de acceso es la base demostrativa de unos cuantos teoremas sobre los árboles splay, entre ellos el siguiente:

**Proposición 6.11** El coste amortizado  $\hat{c}_s$  de una operación `splay()` sobre un nodo  $k$  perteneciente a un árbol splay es  $\mathcal{O}(\lg n)$ .

**Demostración** La operación splay puede imaginarse como una secuencia de pasos zig según la orientación del nodo cuyo costo amortizado total puede definirse como:

$$\hat{c}_s = \sum_{i=1}^m \hat{c}_i .$$

Esta secuencia de operaciones puede pictórizarse del siguiente modo:



El nodo  $k$  sube hacia la raíz mediante  $m - 1$  operaciones zig-zig o zig-zag y una  $m$ -ésima operación que podría ser cualquiera de las tres posibles. Más precisamente sabemos, por el lema de acceso, que los  $m - 1$  primeros pasos están acotados por  $3r_i(k) - 3r_{i-1}(k)$ , pues este valor acota tanto al zig-zig como al zig-zag; mientras que la última operación podría ser cualquiera de las tres, razón por la cual es necesario acotarla al máximo, es decir, a  $3r_m(k) - 3r_{m-1}(k) + 1$ .

En base a lo anterior podemos definir:

$$\begin{aligned} \hat{c}_s &= \sum_{i=1}^{m-1} (3r_i(k) - 3r_{i-1}(k)) + 3r_m(k) - 3r_{m-1}(k) + 1 \\ &= \left( \sum_{i=1}^{m-1} 3r_i(k) \right) - \left( \sum_{i=1}^{m-1} 3r_{i-1}(k) \right) + 3r_m(k) - 3r_{m-1}(k) + 1 \\ &= (3r_1(k) + 3r_2(k) + \dots + 3r_{m-1}(k)) - (3r_0(k) + 3r_1(k) + \dots + 3r_{m-2}(k)) + \\ &\quad 3r_m(k) - 3r_{m-1}(k) + 1 \\ &= 3r_m(k) - 3r_0(k) + 1 \\ &\leq 3r_m(k) + 1 = 1 + \lg n = \mathcal{O}(\lg n) \blacksquare \end{aligned}$$

Como corolario a la proposición 6.11 podemos enunciar que  $p$  operaciones sobre un árbol splay tienen coste amortizado de  $\mathcal{O}(p \lg n)$ . En efecto, como ya expresamos anteriormente, la función potencial considera la calidad de equilibrio del árbol, la cual cuenta para el tiempo de búsqueda que efectúa una operación de un árbol splay.

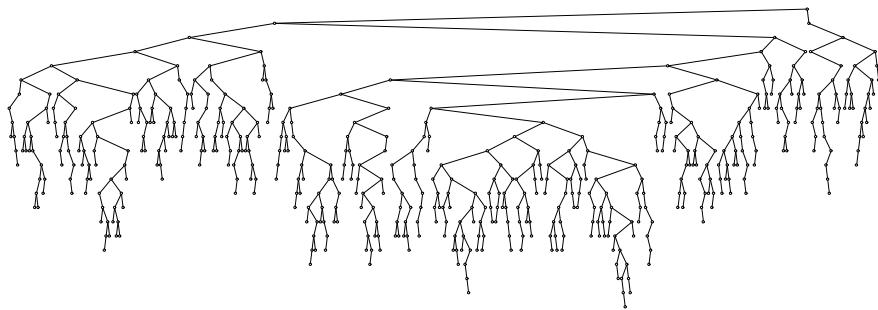


Figura 6.31: Un árbol splay de 512 nodos

## 6.7 Conclusión

Este capítulo trató el equilibrio de árboles binarios de búsqueda. La idea subyacente es minimizar la cantidad de nodos a inspeccionar en una búsqueda.

A tenor de lo anterior estudiamos varias clases de equilibrio junto con sus árboles correspondientes. En primer lugar consideramos el equilibrio fuerte de Wirth y desarrollamos un algoritmo  $\mathcal{O}(n \lg n)$  que equilibra un ABB cualquiera. El desarrollo de este algoritmo nos permitió percarnos de las vicisitudes y dificultades inherentes al equilibrio.

El resto del capítulo se consagró a estudiar cinco clases de árboles equilibrados: aleatorizados, treaps, AVL, rojo-negro y splay. Cada uno con sus ventajas y desventajas a considerar según la aplicación.

El subscriptor del presente texto ha llevado un estudio empírico sobre los diferentes tipos de árboles. Los resultados, sin rigor estadístico, mucho menos sin expresar intervalos de confianza, se muestran en las figuras 6.33, 6.34, 6.35 y 6.36 respectivamente<sup>6</sup>. Las graficas en cuestión corresponden a medidas sobre el siguiente experimento:

1. Se genera una secuencia aleatoria de claves de inserción.
2. Para cada tipo de árbol se hace 3 veces la inserción de la secuencia anterior y, a cada potencia exacta de 2, se efectúan las medidas.

La menor duración medida entre las tres repeticiones se considera como el tiempo de duración.

Lo anterior se repitió 5 veces, es decir, se generaron 5 secuencias aleatorias de inserción. Las gráficas corresponden a los promedios.

La figura 6.33 ilustra las alturas de los seis tipos de ABB estudiados para secuencias de inserción aleatorias. Esta medida es útil porque nos da una idea del costo máximo involucrado en la búsqueda. En este sentido, la figura 6.33 evidencia y confirma la teoría acerca de que los árboles rojo-negro y AVL son los que tienen menos altura. La sorpresa de esta gráfica es que, al menos para entradas aleatorias, los árboles rojo-negro distan de alcanzar su cota teórica  $2\lg(n + 1) - 2$ .

<sup>6</sup>Las medidas en las cuales se expresa tiempo corresponden a los valores más cercanos al centro del intervalo de confianza y fueron efectuados sobre una Intel Pentium III a 800 Mhz.

Medidas sobre arquitecturas diferentes -Sparc y MIPS-, estudios sobre sesgos para verificar bondades del splay, otras métricas -número de rotaciones, por ejemplo- y análisis más exhaustivos aún están pendientes y es un campo fértil para trabajo.

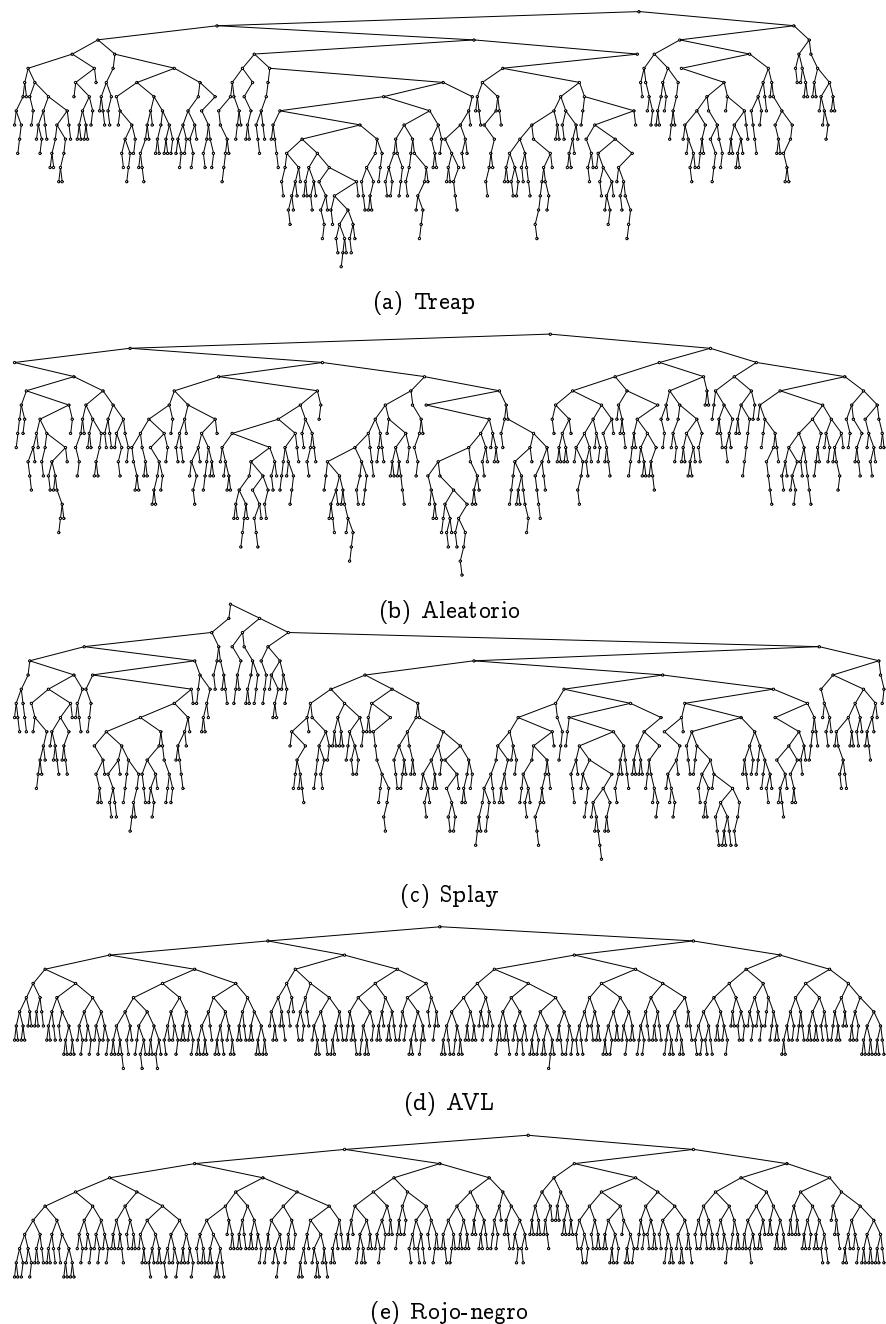


Figura 6.32: Diferentes modelos de equilibrio y sus árboles resultantes para la misma secuencia de 512 claves aleatorias

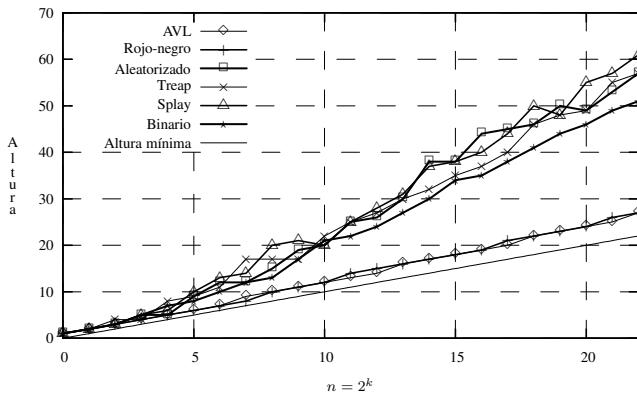


Figura 6.33: Altura de diferentes ABB

La longitud del camino interno nos indica cuán equilibrado está el árbol. En este sentido, la figura 6.34 evidencia que la calidad de todos los árboles es equivalente, siendo ligeramente favorable hacia los árboles AVL y rojo-negro.

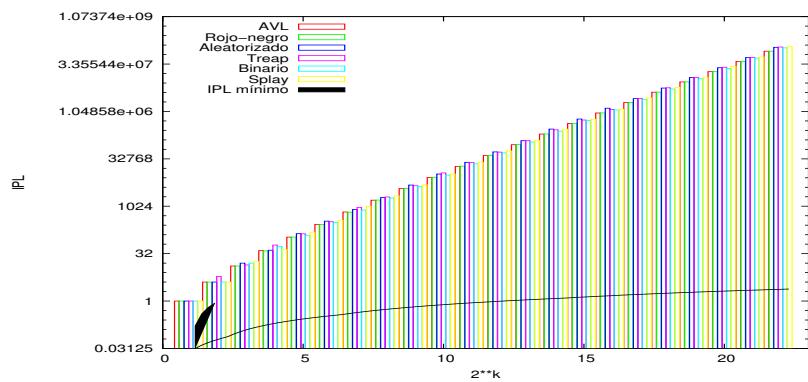


Figura 6.34: IPL para diferentes ABB (en escala logarítmica)

Las figuras 6.33 y 6.34 esconden el costo de la operación splay para los árboles del mismo nombre, pues el experimento no considera ni emula la localidad de referencia. En este sentido, parece que si cualquiera de las claves tiene la misma probabilidad de buscarse, entonces el árbol splay es el de más pobre desempeño. Pero esto puede ser falacioso, pues la localidad de referencia cumple un rol fundamental en la mayoría de las aplicaciones. En los árboles de altura acotada, los probabilísticos y los binarios clásicos, las claves más recientes tienden a encontrarse hacia las hojas: mientras cuanta joven es una clave, mayor es el tiempo esperado de búsqueda. Esto no sucede con un árbol splay: las claves recientemente accedidas se encuentran cercanas a la raíz, por lo que el desempeño debería de ser considerablemente mejor para aplicaciones con localidad de referencia.

Las duraciones de inserción y eliminación para los árboles splay no se muestran porque son tan notablemente superiores al resto que su visualización obstaculizaría apreciar las diferencias entre las demás curvas.

Si bien los cinco enfoques de equilibrio ofrecen desempeños  $\mathcal{O}(\lg n)$  para las tres operaciones fundamentales, es crucial observar que para la inserción y eliminación todos estos

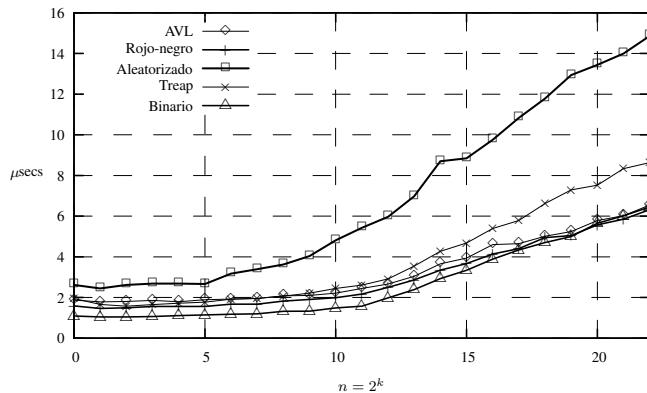


Figura 6.35: Tiempo de inserción para diferentes ABB

enfoques de equilibrio acarrean costes constantes severamente mayores que los de un ABB clásico. Las figuras 6.35 y 6.36 evidencian dramáticamente este aspecto: el ABB clásico es el más rápido hasta  $2^{19}$  nodos para la inserción y  $2^{14}$  nodos para la eliminación. A partir de estos valores, el ABB clásico es ligeramente inferior pero casi tan bueno como el rojo-negro y el AVL. Para esta última observación no se debe olvidar que los ordenes de inserción son aleatorios.

En la selección de un árbol no se debe despreciar el uso de un ABB clásico. En tal sentido, un ABB debe considerarse antes que cualesquiera de los métodos de equilibrio desarrollados en este capítulo. La selección de un ABB debe considerar los siguientes aspectos:

1. La escala en función del número de nodos es pequeña o mediana.
2. Las secuencias de inserción no deben estar considerablemente sesgadas, pues un ABB es  $\mathcal{O}(\lg n)$  para inserciones aleatorias. Si las secuencias de inserción son aleatorias, entonces se puede considerar un ABB para grandes escalas.
3. Puesto que las eliminaciones degradan un ABB, el número de eliminaciones debe ser mucho menor que el de inserciones.
4. El desempeño  $\mathcal{O}(\lg n)$  es probabilístico. Aun con inserciones aleatorias y ausencia de eliminaciones, un ABB puede exhibir un caso desafortunado. Por lo tanto, un ABB no debe utilizarse si existen requerimientos absolutos de desempeño.

La idea subyacente de los enfoques aleatorizados es eliminar cualquier sesgo en la inserción o eliminación mediante decisiones aleatorias que aleatorizan el árbol haciéndolo equivalente a un ABB construido a partir de una secuencia al azar. Consecuentemente, se puede emplear un árbol aleatorizado o un treap cuando se teme sesgo en la entrada. Puesto que el desempeño es probabilístico, se debe garantizar la calidad del generador de números pseudoaleatorios. Aun con un buen generador de números aleatorios, el desempeño de un enfoque aleatorizado es susceptible al infiutnío. Así pues, los equilibrios probabilísticos no deben usarse si se presentan restricciones fuertes de desempeño.

Las figuras 6.35 y 6.36 evidencian al árbol aleatorizado como uno de los más costosos. De hecho, para la inserción, se aprecia una degradación apreciable según aumenta

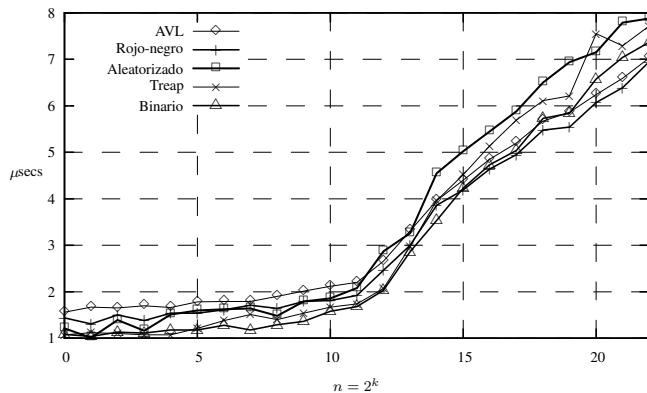


Figura 6.36: Tiempo de eliminación para diferentes ABB

el número de nodos, degradación que no ocurre con la otra alternativa aleatorizada: el treap. En opinión del autor, esto es explicable por las siguientes razones:

1. Los árboles aleatorios son más sensibles a la calidad del generador de números aleatorios que los treaps. Los ABBA requieren números al azar entre cero y la cardinalidad del árbol; mientras que las prioridades de los treaps utilizan todo el espectro del generador de números aleatorios. Este hecho es importante de considerar porque el rango restringido de los árboles aleatorios lo hace más sensible a fallas, sesgos y demás imperfecciones del generador de números aleatorios.

Otro aspecto a considerar es que el treap efectúa muchas menos llamadas al generador de números aleatorios que el ABBA. El treap genera un aleatorio una sola vez por inserción; mientras que el ABBA efectúa sorteos en la inserción y en la eliminación. Además, el ABBA puede efectuar varios sorteos en cada operación. La cantidad esperada de sorteos depende de la cardinalidad del árbol: a mayor cantidad de nodos, mayor el número esperado de sorteos. Esta consideración es esencial, pues muchos generadores sufren de “fatiga” en el sentido de que su aleatoriedad disminuye según aumenta el número de llamadas.

2. Los algoritmos del treap son más simples y tienen menos costes constantes que los del ABBA.

La discusión anterior evidencia claramente que si existen restricciones insalvables de desempeño y si no hay espacio para la mala suerte, entonces la escogencia de facto es un árbol con altura acotada: un AVL o un rojo-negro. Si se desea una implementación sencilla y que exhiba buen tiempo, entonces, en opinión del autor, los treaps son la mejor alternativa. Finalmente, si se tiene conocimiento de una buena localidad de referencia, entonces el desempeño los árboles splay puede ser espectacular.

## 6.8 Notas bibliográficas

La historia de los enfoques de equilibrio de un árbol comienza por los árboles binarios de búsqueda clásicos. A principio de 1960, los investigadores ya sabían que las promesas

de desempeño de un ABB sólo eran vigentes a condición de tener secuencias de inserción aleatorias y de no ocurrir excesivas eliminaciones.

En 1962 se publicaron los árboles AVL por Adelson Velsky y Landis [6]. Hasta recientemente, estos árboles permanecieron la escogencia de facto en la implantación de tablas de símbolos en memoria principal. Aún hoy en día, esta clase de árbol se usa frecuentemente y conforma el estándar de muchas aplicaciones.

El número  $\varphi$  es muy importante en muchos lugares de las matemáticas. Es probable que su uso provenga del mundo artístico, pues desde tiempos inmemoriales  $\varphi$  ha sido considerada la más bella proporción entre dos diferentes tamaños. Edsger W. Dijkstra expone los números de Fibonacci y la sección áurea orientada hacia las ciencias computacionales [37]<sup>7</sup>.

A finales de 1960, investigaciones en recuperación de claves en almacenamiento secundario condujeron a una nueva y popular clase de árbol: el B, descubierto por Bayer [15]. Esta clase de árbol generó una corriente de investigación que condujo a diversas estructuras de búsqueda en memoria principal, notablemente el árbol 2-3, descubierto por Hopcroft y descrito por Aho, Hopcroft y Ullman [9], y el árbol binario B simétrico, que es un caso particular del árbol B descubierto por Bayer [14]. Los árboles rojo-negro son un equivalente binario de un tipo de árbol cuaternario denominado por Bayer 2-3-4. Implantar árboles 2-3-4 en memoria es complicado porque en un nodo deben mantenerse claves ordenadas y punteros a los hijos. Esta aclaratoria es importante porque las demostraciones acerca la garantía de balance pueden efectuarse directamente con los árboles 2-3-4.

En 1978, Guibas y Sedgewick [67] aprovecharon un isomorfismo entre los árboles 2-3-4 y los árboles binarios de búsqueda y enunciaron el árbol objeto de esta sección: el árbol rojo-negro. Aparte de que es más fácil implantar un árbol rojo-negro que un árbol 2-3-4, la definición 6.7 establece un enfoque combinatorio, natural, para ver por qué los árboles rojo-negro son balanceados. Wood [184] ofrece una excelentísima perspectiva de los árboles rojo-negro basada en sus propiedades "cromáticas", bajo la cual se constituyó el discurso de este texto.

Los treaps fueron descubiertos por Vuillemin en 1978 [176] y estudiados en profundidad por Seidel y Aragon [158]. Por su desempeño y sencillez, esta clase de árbol es la selección favorita de muchas aplicaciones.

Muchas grandes ideas son simples y naturales a condición de que ya hayan sido reveladas. Un notable ejemplo lo constituyen los árboles aleatorizados descubiertos por Mar-

---

<sup>7</sup> Esta proporción está onminiscentemente presente a lo largo de toda la naturaleza, la anatomía humana inclusive. Como ejemplos vanidosos, el ombligo es considerado por los escultores como la división natural de todo el cuerpo. Pues bien, la distancia desde el suelo al ombligo, dividida entre la distancia desde el ombligo hasta la cabeza, es  $\varphi$ . Del mismo modo, exactamente la misma proporción es preservada entre la cabeza y la quijada, donde  $\varphi$  está ubicado en la punta de la nariz. Todas las partes del cuerpo humano respetan esta proporción.

La proporción en cuestión es que al dividir un segmento en dos partes, digamos  $x$  e  $y$ , se satisfaga la siguiente ecuación  $\frac{x}{x+y} = \frac{y}{x}$ . Si se efectúa la transformación  $\varphi = \frac{x}{y}$  se tiene la ecuación divina:  $\varphi^2 - \varphi - 1 = 0$ .

Puesto que la mayoría somos creyentes, en honor a Dios, creador del Universo -y del Hombre inclusive-,  $\varphi$  es llamado el número divino, el número de Dios, el radio de oro o, artísticamente, la sección áurea, pues parece que fue la proporción que Dios utilizó para crear nuestro mundo.

El nombre  $\varphi$  es en honor a Fidias, considerado el más grande escultor de la antigua Grecia, autor de la estatua de Zeus en el monte Olimpo, elaborada de marfil y oro aproximadamente en el año 450 A.C. Esta obra fue incluida por el poeta Antípater de Sidón en su célebre lista de las siete maravillas del mundo antiguo.

tinez y Roura [117]. Durante décadas, los teóricos buscaron una estructura aleatoria natural que resolviese la asimetría de la eliminación de un ABB. A pesar del treap de Vuillemin [176] y de que Seidel y Aragon hayan demostrado su aleatoriedad y equivalencia con el ABB [158], el árbol aleatorizado es una estructura natural que pondera probabilísticamente según el equilibrio fuerte de Wirth [182].

Desde principios de la computación, los diseñadores de sistemas han utilizado felizmente heurísticas autoajustantes que mueven elementos recientemente accedidos a posiciones de rápido acceso. Desde el descubrimiento de la inserción por la raíz [162], muchos programadores usaron este aspecto para aprovechar la localidad de referencia. Pero no fue sino hasta 1985, cuando Sleator y Tarjan publicaron los árboles splay [160], que se dispuso de un árbol logarítmico en el sentido amortizado. Por añadidura, este fue uno de los primeros estudios convincentes sobre el análisis amortizado.

## 6.9 Ejercicios

1. ¿Cuántos nodos tiene el árbol de la figura 6.1(a)?
2. Diseñe la rutina `select_goto_root()` presentada en § 6.1 (Pág. 452) en función del procedimiento `insert_by_pos_xt()` explicado en § 4.11.7 (Pág. 341).
3. Ejecute las siguientes secuencias de inserción en un árbol aleatorio.
  - (a) 127 193 30 71 158 185 40 12 35 94 116 110 139 57 81 95 181 114 112 171
  - (b) 143 118 34 111 125 61 101 45 126 38 12 41 199 54 21 166 136 154 188 174
  - (c) 148 176 182 108 141 44 173 128 102 65 139 192 37 188 116 88 165 162 9 91
  - (d) 108 67 109 76 83 175 140 136 35 168 62 181 100 112 197 92 1 173 180 33
  - (e) 123 193 13 15 50 115 160 25 45 136 79 3 1 14 178 108 62 57 101 75

Para todas las secuencias, asuma la siguiente secuencia de números aleatorios entre 1 y 100:

```
43 19 46 43 6 28 11 5 10 35 4 38 23 48 3 17 30 31 41 12 35 32 44 47 40 46 3 44 32 19 9 39 38
19 28 12 13 18 36 11 50 9 33 44 6 8 29 6 48 13 30 21 16 48 3 14 39 9 1 31 23 48 40 46 34 30 43
3 40 4 15 36 40 41 30 26 44 23 34 50 33 10 49 22 9 37 32 47 26 19 40 39 12 10 5 10 26 36 49 24
21 28 1 48 24 7 14 39 38 42 45 20 5 40 5 31 20 9 50 7 35 13 7 13 19 50 35 45 9 39 3 34 32 17 45
19 41 40 32 8 24 9 17 48 12 44 24 7 33 25
```

4. Para los siguientes árboles aleatorios, definidos por sus recorridos prefijos, ejecute las secuencias de eliminación dadas:
  - (a) **Prefijo:** 46 11 18 151 102 83 55 70 61 63 79 103 125 124 140 148 173 162 160 181  
**Secuencia de eliminación:** 125 148 18 11 79 124 140 181 102 173
  - (b) **Prefijo:** 195 193 105 40 33 5 31 22 50 61 68 62 76 159 151 146 112 137 165 199  
**Secuencia de eliminación:** 31 68 165 22 159 105 112 61 199 151
  - (c) **Prefijo:** 132 48 37 28 12 9 95 76 55 70 82 106 102 97 153 135 195 171 165 177  
**Secuencia de eliminación:** 165 177 171 9 76 28 195 48 55 37
  - (d) **Prefijo:** 96 89 59 28 21 13 46 30 83 87 174 147 132 109 101 99 162 150 157 19

**Secuencia de eliminación:** 28 132 13 99 101 150 87 174 157 96

(e) **Prefijo:** 190 63 43 37 13 3 28 38 55 51 184 177 90 124 122 168 138 162 187 196

**Secuencia de eliminación:** 90 37 124 190 51 28 122 168 3 13

Para todos los ejercicios asuma la siguiente secuencia de números aleatorios entre 1 y 100:

1 83 9 84 64 46 100 65 7 43 20 63 26 61 100 95 95 60 75 50 19 51 21 29 90 69 80 93 71 52 34 40  
34 29 67 63 39 57 93 70 31 82 67 93 47 12 99 67 4 59 75 41 93 93 21 93 52 49 50 93 100 39 58 6  
95 68 45 63 10 13 4 100 85 1 3 70 77 43 94 35 8 56 71 69 96 80 30 30 85 59 68 35 29 34 97 15  
29 44 79 71 26 47 12 67 94 27 30 80 39 79 32 3 49 65 66 89 45 66 93 68 69 81 85 99 86 44 18 80  
50 19 69 90 96 35 12 1 72 21 26 27 29 55 52 62 50 19 50 46 68 71

5. Dado un nodo  $p$  a eliminar de un árbol aleatorizado, considere el siguiente algoritmo de eliminación:

- (a) Sea  $r = \text{número aleatorio entre } [1, |L(p)| + |R(p)|]$ .
- (b) Si  $r \leq |L(p)| \implies$ 
  - La raíz del subárbol resultante es el predecesor de  $p$ .

De lo contrario  $\implies$

- La raíz del subárbol resultante es el sucesor de  $p$ .

Pruebe o desapruebe que el árbol resultante es aleatorio. (+)

6. Instrumente el algoritmo anterior.

7. Analice cuidadosamente el algoritmo de inserción en un árbol aleatorizado y estime la cantidad esperada de sorteos junto con su varianza.

8. Analice cuidadosamente el algoritmo de eliminación en un árbol aleatorizado y estime la cantidad esperada de sorteos junto con su varianza.

9. Como se ha mencionado, uno de los problemas de los árboles aleatorizados es que se efectúan muchos sorteos que hacen que el generador de números aleatorios alcance su punto de fatiga más rápidamente. Proponga un esquema que minimice las posibilidades de fatiga del generador de números aleatorios.

10. Diseñe un algoritmo que genere un árbol de búsqueda aleatorio, es decir, que corresponda a la definición de árbol aleatorio dada en § 6.2 (Pág. 455).

11. Considere el siguiente prototipo de función:

```
template <class Node> luka_to_tree(char *& cod)
```

El cual retorna el árbol binario correspondiente a la palabra de Lukasiewicz almacenada en  $cod$  (ver § 4.7 (Pág. 302)). Asuma que la palabra es correcta. Escriba la función en cuestión.

12. Haga las pruebas estadísticas pertinentes para verificar si el sorteo de inserción en la raíz se distribuye con probabilidad  $p = \frac{1}{n+1}$ .
13. Haga las pruebas estadísticas pertinentes para verificar si el sorteo para la selección de la raíz del algoritmo de concatenación aleatoria se distribuye con probabilidad  $p = \frac{1}{m}$ , donde  $m$  es la cantidad de nodos del árbol izquierdo.
14. Diseñe un algoritmo de unión de dos árboles aleatorios  $T_1$  y  $T_2$  tal que claves pertenecientes a  $T_1$  pueden ser mayores que algunas claves pertenecientes a  $T_2$ . El algoritmo debe ser  $\mathcal{O}(n \lg(n))$ .
15. Diseñe un algoritmo de unión de dos árboles aleatorios  $T_1$  y  $T_2$  tal que claves pertenecientes a  $T_1$  pueden ser mayores que algunas claves pertenecientes a  $T_2$ . El algoritmo debe ser  $\mathcal{O}(n \lg(n))$ .
16. Diseñe un algoritmo que efectúe la inserción por posición en un árbol aleatorizado. El árbol resultante debe ser aleatorio. Demuestre su respuesta. (+)
17. Diseñe un algoritmo que efectúe la eliminación por posición en un árbol aleatorizado. El árbol resultante debe ser aleatorio. Demuestre su respuesta. (+)
18. Diseñe un TAD equivalente a `Gen_Rand_Tree<Key>` en el cual todas las operaciones no sean recursivas.
19. Ejecute las siguientes secuencias de inserción en un treap:

- (a) 49 97 82 142 187 85 14 123 194 129 190 114 152 109 66 33 99 37 86 73
- (b) 11 24 126 10 17 85 100 194 29 97 167 18 48 80 12 186 6 3 84 74
- (c) 14 171 41 120 172 11 123 15 71 74 176 148 149 31 104 184 83 182 81 47
- (d) 130 176 8 142 182 129 125 153 0 122 13 190 148 184 30 84 16 1 58 39
- (e) 4 161 3 120 71 145 68 140 191 134 29 116 118 21 104 108 190 159 158 148

Para todos los ejercicios, use la siguiente secuencia de números aleatorios:

33 81 23 16 75 60 14 89 93 99 97 66 38 34 47 49 52 47 21 40

20. Para los siguientes treaps, definidos por sus recorridos prefijos, ejecute las secuencias de eliminación dadas:
- (a) **Prefijo:** 148 22 16 8 94 51 41 25 30 66 63 128 124 116 113 111 190 173 186 175  
**Secuencia de eliminación:** 157 23 85 165 118 111 70 184 186 19
  - (b) **Prefijo:** 86 58 15 29 32 41 47 83 164 129 90 87 102 126 114 106 136 196 174 192  
**Secuencia de eliminación:** 100 31 190 158 84 105 149 10 52 32
  - (c) **Prefijo:** 54 89 34 137 180 44 133 112 71 51 7 39 41 84 161 15 69 113 93 59  
**Secuencia de eliminación:** 133 5 51 155 77 178 6 166 186 115
  - (d) **Prefijo:** 140 66 38 8 25 15 46 76 110 79 102 83 129 161 145 151 156 155 164 194  
**Secuencia de eliminación:** 22 36 73 63 115 140 123 45 123 26
  - (e) **Prefijo:** 122 113 42 22 7 26 62 49 97 172 133 127 153 137 151 167 156 158 168 198  
**Secuencia de eliminación:** 199 49 70 62 150 7 26 118 158 75

21. Diseñe un algoritmo que construya un treap dados sendos arreglos con claves y prioridades. El algoritmo no puede usar las primitivas del treap.
22. Diseñe un TAD equivalente a Gen\_Treap<Key> en el cual todas las operaciones no sean recursivas.
23. Diseñe un algoritmo  $\mathcal{O}(n \lg(n))$  que efectúe la unión de dos treaps.
24. Diseñe un algoritmo  $\mathcal{O}(n \lg(n))$  que efectúe la intersección de dos treaps.
25. Diseñe un algoritmo  $\mathcal{O}(n \lg(n))$  que efectúe la partición (split) de un treap.
26. Considere un treap en el cual la prioridad está dada por el número de accesos. A mayor número de accesos, menor es la prioridad. De este modo, el treap devendría auto ajustante en el sentido de que las claves recientemente accedidas se encontrarían en la raíz. Estudie y compare este método con la selección aleatoria de prioridades.
27. Considere la inserción por la raíz de un treap:

```
Node * insert_root(Node * r, Node * p)
```

La cual inserta el nodo p, con clave K(p) y prioridad escogida aleatoriamente P(p), como raíz del treap r.

- (a) Instrumente el algoritmo que implanta insert\_root().
- (b) Asumiendo que el valor de P(p) es único en el treap resultante, demuestre que el algoritmo de la pregunta anterior genera un treap.
28. Ejecute las siguientes secuencias de inserción en un árbol AVL:
  - (a) 235 206 185 156 194 108 137 207 141 93 148 231 75 144 6 239 2 138 30 187 58 244 0 83 55
  - (b) 207 76 4 226 216 83 129 221 230 81 29 46 201 208 212 56 168 38 78 87 138 215 66 122 154
  - (c) 247 168 113 141 1 176 52 179 223 106 122 181 100 118 81 21 27 41 228 197 2 123 22 244 212
  - (d) 144 77 246 193 213 230 82 44 114 195 148 88 231 0 163 109 228 66 28 12 132 18 135 79 74
  - (e) 39 234 1 242 172 156 108 30 170 241 232 3 140 5 180 100 133 60 24 139 129 19 235 118 236
29. Para los siguientes árboles AVL, definidos por sus recorridos prefijos, ejecute las secuencias de eliminación dadas:
  - (a) **Prefijo:** 136 103 62 22 19 12 55 45 86 70 99 123 121 126 134 181 152 142 147 158 193 189  
**Secuencia de eliminación:** 99 123 121 181 134 193 12 22 45 205 62 189
  - (b) **Prefijo:** 128 41 28 21 8 24 29 93 54 49 84 117 94 207 158 146 143 157 182 190 225 211  
**Secuencia de eliminación:** 41 54 29 211 28 235 128 84 24 182 146 8
  - (c) **Prefijo:** 110 49 29 11 1 21 39 91 80 77 102 97 191 173 156 151 172 184 181 219 210 209  
**Secuencia de eliminación:** 80 21 49 226 181 39 11 91 29 172 184 97

(d) **Prefijo:** 151 47 44 7 1 35 46 115 93 71 94 140 126 218 203 184 183 201 192 213 207 232  
231 227 243

**Secuencia de eliminación:** 184 227 213 207 46 47 151 44 231 115 94 7

(e) **Prefijo:** 172 131 89 69 27 110 106 116 155 141 137 152 158 194 185 176 193 190 222 201  
217 244 236 231 245

**Secuencia de eliminación:** 185 131 201 176 69 194 190 172 217 236 152 158

30. Si cada nodo de un árbol AVL almacena su altura, ¿cuántos bits se requieren?
31. Calcule la complejidad de tiempo de la función `is_avl()`.
32. Implante enteramente un TAD que modele árboles AVL en el cual cada nodo guarde su altura. (+)
33. Diseñe un algoritmo  $\mathcal{O}(n \lg(n))$  que efectúe la unión de dos árboles AVL.
34. Diseñe un algoritmo  $\mathcal{O}(n \lg(n))$  que efectúe la partición (split) de dos árboles AVL.
35. Diseñe un algoritmo  $\mathcal{O}(n \lg(n))$  que efectúe la intersección de dos árboles AVL.
36. Escriba un algoritmo que determine si un árbol binario es de Fibonacci. Calcule su complejidad de tiempo.
37. Escriba un algoritmo que genere el árbol de Fibonacci de orden  $k$ .
38. (De Moivre - 1718) Demuestre que

$$\lim_{n \rightarrow \infty} \frac{\text{Fib}(n+1)}{\text{Fib}(n)} = \frac{1 + \sqrt{5}}{2} .$$

Donde  $\text{Fib}(i)$  es el  $i$ -ésimo número de Fibonacci.

39. Demuestre que:

$$\sum_{i=0}^n F_i = F_{n+2} - 1$$

Donde  $F_i$  es el  $i$ -ésimo número de Fibonacci.

40. Obtenga una expresión que denote el número de hojas de un árbol de Fibonacci.
41. Demuestre que el árbol de Fibonacci  $T_k$  contiene  $F_{k+2}$  nodos externos.
42. Calcule la longitud del camino interno del árbol de Fibonacci de orden 8.
43. Deduzca una expresión general para la longitud del camino interno del árbol de Fibonacci de orden  $T_k$ . (++)
44. Considere  $\text{Fib}_l$  como el conjunto de todos los árboles de Fibonacci "libres". Un árbol de Fibonacci "libre"  $T_k$  de orden  $k$  se define como:
  - (a) Caso base  $k = 0$ :

$$T_0 = \emptyset \in \text{Fib}_l \quad (6.60)$$

(b) Caso base  $k = 1$ :

$$T_1 = \cdot \in \text{Fib}_l \quad (6.61)$$

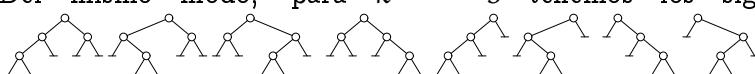
(c) Caso general para  $k > 1$ :

$$T_k = \langle T_i, \cdot, T_j \rangle \quad (6.62)$$

tal que  $i \neq j$ ,  $i < k$ ,  $j < k$ ,  $((i = k - 1 \wedge j = k - 2) \vee (i = k - 2 \wedge j = k - 1))$ ,  $L(T_k) \in \text{Fib}_l$  y  $R(T_k) \in \text{Fib}_l$

Por ejemplo, y para señalar una diferencia con los árboles de Fibonacci tradicionales, para  $k = 2$  tenemos los siguientes dos árboles posibles: .

Del mismo modo, para  $k = 3$  tenemos los siguientes árboles posibles:



(a) Considere el siguiente prototipo de función:

```
template <class Node> bool is_free_fib_tree(Node * r)
```

La cual retorna true si el árbol cuya raíz es  $r$  es un árbol de Fibonacci libre.

Implemente la función en cuestión.

(b) Determine la cantidad de árboles de Fibonacci libres distintos de orden  $k$ . (++)

(c) Considere el siguiente prototipo de función:

```
template <class Node> Node random_free_fib_tree(int k)
```

La cual retorna un árbol de Fibonacci libre de orden  $k$  aleatorio. (++)

45. Demuestre que la inserción en un árbol AVL causa a lo más una rotación doble.

46. Explique y desarrolle un método para reconstruir un árbol AVL dado el recorrido infijo de las claves y los valores de diferencia de altura de cada nodo.

47. Demuestre que para todo árbol AVL existe una coloración rojo-negro. (+)

48. Dibuje el árbol rojo-negro correspondiente a la siguiente secuencia de inserción:

247 184 240 238 275 28 108 34 201 147 41 113 93 110 98 101 164 253 186 70 198 145  
250 158 292 273 257 243 255 171

49. Diseñe un algoritmo que examine un árbol binario cualquiera y determine si es coloreable.

50. Diseñe un algoritmo que coloree de rojo-negro un árbol cualquiera que haya pasado el test del ejercicio anterior.

51. Ejecute las siguientes secuencias de inserción en un árbol rojo-negro:

(a) 14 52 107 169 66 44 15 116 68 196 51 177 79 171 218 59 9 197 125 84 89 1 213 195 163

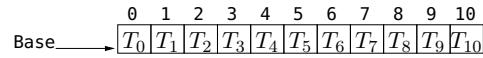
(b) 110 199 147 180 79 10 127 240 0 38 104 195 207 49 89 9 153 191 101 54 206 181

- (c) 152 15 36 31 124 239 57 90 86 26 129 169 91 133 137 42 99 32 35 102 200 220 95 80  
 (d) 228 226 97 175 185 161 21 240 227 237 194 74 84 90 193 27 72 51 192 142 55 229 167  
 (e) 197 25 107 0 62 114 124 221 122 179 105 128 78 29 58 41 208 217 46 158 213 32 66 246 152
52. Para los siguientes árboles AVL, definidos por sus recorridos prefijos, ejecute las secuencias de eliminación dadas:
- (a) **Prefijo:** 195 143 90 8 0 1 83 58 125 107 156 148 153 188 187 186 193 264 218 217 211 232  
 225 238 267 265 277 270 278  
**Secuencia de eliminación:** 8 265 193 148 225 107 195 58 0 218 186 125 270 217 277
- (b) **Prefijo:** 160 89 35 20 1 26 32 77 133 95 149 135 159 282 213 175 167 178 190 262 240 269  
 289 285 297 294 299  
**Secuencia de eliminación:** 262 285 160 269 159 32 240 282 133 299 213 294 26 178 167
- (c) **Prefijo:** 129 62 38 12 6 31 42 84 73 107 93 128 241 157 148 140 143 150 195 173 227 229  
 276 244 247 291 281 296  
**Secuencia de eliminación:** 195 291 84 42 31 12 281 173 157 129 241 140 276 107
- (d) **Prefijo:** 156 67 23 13 10 22 53 30 64 115 73 89 138 136 151 272 185 177 169 178 229 192  
 247 292 278 291 294 299  
**Secuencia de eliminación:** 247 229 272 278 294 13 292 177 136 192 73 151 138 64 89
- (e) **Prefijo:** 106 46 39 21 3 37 40 41 69 55 53 90 70 93 92 229 165 128 125 110 130 159 183  
 174 224 196 293 238 260 297  
**Secuencia de eliminación:** 159 125 130 55 37 183 297 293 165 41 3 46 196 92 21
53. (Tomado y traducido de Parberry [139]) Dibuje un árbol AVL para el cual la eliminación de un nodo requiera dos rotaciones dobles. Dibuje el árbol, identifique el nodo a eliminar y explique por qué dos rotaciones son necesarias y suficientes. (+)
54. Considere el siguiente prototipo de búsqueda:

```
template <typename T> int search(T a[], const T & x, int n)
```

a es un arreglo ordenado de n elementos en el cual se busca la clave x.

El fin de este problema es implantar un esquema de búsqueda denominado de “Fibonacci”, el cual estructura las observaciones sobre el arreglo de la siguiente manera general:



$F_k$  es el k-ésimo número de Fibonacci. i es la posición de inspección actual, mientras que p y q son los números de Fibonacci de orden inmediatamente inferior a i. La idea del esquema de búsqueda es comparar x con a[i] y, si  $x \neq a[i]$ , entonces, según x sea menor o mayor a a[i], desplazar los intervalos de búsqueda hacia un intervalo menor o mayor que i.

- (a) Asumiendo que  $n + 1 = F_{k+1}$ , escriba un algoritmo que implante la búsqueda de Fibonacci.

Ayuda 1: los índices q y p siempre son números consecutivos de Fibonacci, mientras que ese no siempre es el caso para el índice i.

Ayuda 2: El valor inicial de i es  $F_k$ .

Ayuda 3: La condición de parada es que  $x \neq a[i]$  y  $q == 0$  or  $p == 1$ . Es decir, el elemento no se encuentra y no es posible desplazar el intervalo sin que se "salga" del arreglo.

(b) Analice el tiempo de ejecución de la búsqueda de Fibonacci.

(c) ¿Cómo puede hacerse si  $n + 1 \neq F_{k+1}$

55. (Tomado y traducido de Parberry [139]) Demuestre que  $\forall n \in \mathbb{N}$  existe un árbol AVL para el cual la eliminación de un nodo requiere exactamente n rotaciones. Indique cómo se construye tal árbol, indique de manera general el nodo a eliminar y explique por qué n rotaciones son necesarias.

56. Diseñe un algoritmo  $\mathcal{O}(n \lg(n))$  que efectúe la unión de dos árboles rojo-negro.

57. Diseñe un algoritmo  $\mathcal{O}(n \lg(n))$  que efectúe la partición (split) de dos árboles rojo-negro.

58. Diseñe un algoritmo  $\mathcal{O}(n \lg(n))$  que efectúe la intersección de dos árboles rojo-negro.

59. Ejecute las siguientes secuencias de inserción en un árbol splay:

(a) 173 220 204 95 138 208 247 264 154 7 244 167 283 188 187 97 12 168 238 197 166 253 270  
291 186 160 59 60 250 117

(b) 6 37 144 285 105 259 80 65 106 205 32 141 24 150 3 30 64 7 57 16 187 101 293 20 73 14  
147 93 79 153

(c) 139 200 91 141 37 237 221 63 33 214 224 297 21 274 195 278 243 100 196 256 148 252 163  
206 3 77 8 276 99 281

(d) 112 129 96 71 237 86 79 45 255 226 36 64 225 240 167 276 7 239 37 232 115 175 264 138  
22 33 299 229 161 111

(e) 271 181 204 288 121 128 105 254 81 162 247 35 56 117 115 174 253 255 208 61 239 236 66  
161 145 70 137 258 148 107

60. Para los siguientes árboles AVL, definidos por sus recorridos prefijos, ejecute las secuencias de eliminación dadas:

(a) **Prefijo:** 77 74 18 6 64 21 38 72 286 82 83 88 227 205 98 130 196 147 284 275 271 236 230  
265 240 259 270 291 295 298

**Secuencia de eliminación:** 286 298 295 196 83 227 77 64 147 236 38 88 205 6 21

(b) **Prefijo:** 277 275 10 3 157 156 59 40 47 93 64 78 87 150 234 160 231 190 162 165 186 209  
225 211 218 255 270 290 291 293

**Secuencia de eliminación:** 10 40 47 255 186 162 87 150 93 157 160 231 277 290 293

(c) **Prefijo:** 216 214 188 186 171 130 122 23 16 10 7 67 98 92 75 90 83 113 166 159 165 202  
200 213 234 262 251 266 269 292

**Secuencia de eliminación:** 266 213 165 122 75 7 83 166 23 269 292 186 16 67 216

(d) **Prefijo:** 64 60 55 10 6 15 26 19 30 67 72 247 245 166 93 90 159 143 131 117 173 238 195  
181 248 252 261 253 256 284

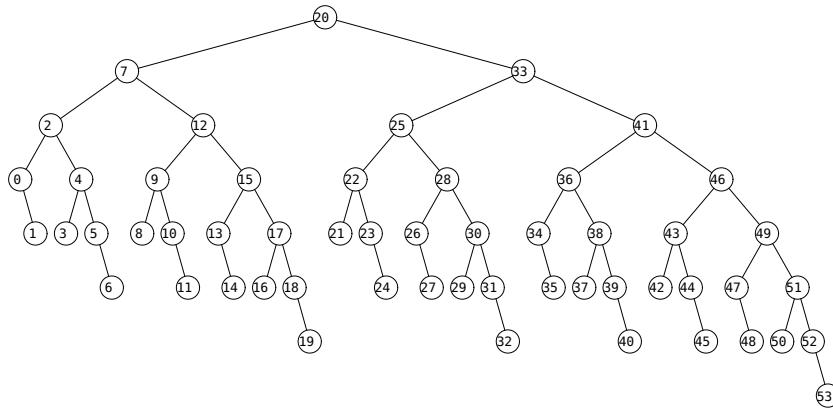


Figura 6.37: Árbol AVL a eliminar el nodo 0

**Secuencia de eliminación:** 15 10 90 26 60 166 238 67 248 64 173 256 117 72 6

(e) **Prefijo:** 60 27 20 15 12 5 266 182 73 136 89 134 97 90 105 115 168 241 222 213 190 188  
184 198 199 211 259 251 286 293

**Secuencia de eliminación:** 198 27 60 266 15 293 89 20 241 115 213 5 73 136 105

61. Dado el árbol AVL de la figura 6.37, Dibuje el árbol resultante de eliminar el nodo 0 (cero).
62. Dibuje el árbol AVL resultante de eliminar el nodo 53 del árbol de la figura 6.13.
63. Diseñe un algoritmo que efectúe la partición de un árbol splay. Demuestre que la altura del árbol resultante es logarítmica.
64. Dibuje el árbol splay correspondiente a la siguiente secuencia de inserción:  
127 209 226 65 259 95 120 21 240 272 210 122 84 33 297 105 160 40 222 115 248 17 262  
194 135 91 152 199 169 219



# Grafos

Entre nuestros sentidos sensoriales, parece que la vista es el que ocupa la mayor parte de nuestra percepción. Nuestra memoria y recuerdo están impregnados en gran medida de imágenes visuales. En muchas ocasiones preferimos tratar con imágenes porque nos son más concretas a nuestra percepción y entendimiento que los conceptos abstractos. Cuando lidiamos con lo puramente abstracto, es muy posible que asociemos una imagen visual. Cuando escuchamos “tres”, por ejemplo, es posible que se nos aparezca una imagen relacionada a una triplicidad o al dígito “3”.

En la construcción del conocimiento humano suelen utilizarse imágenes artificiales para expresar mejor ideas y conceptos, o para sintetizarlas en una imagen que englobe y resuma los asuntos de interés. Las imágenes de este tipo son llamadas comúnmente “gráficos”. Los gráficos cartesianos, es decir, las curvas matemáticas dibujadas en el eje cartesiano de coordenadas, permiten mirar ampliamente el comportamiento de una función, comportamiento éste que puede estar oculto en la mera fórmula. Esta clase de gráfico es artificial en el sentido de que no se encuentra naturalmente; o sea, no se corresponde con una imagen percibida en el mundo natural.

Existe una amplia variedad de problemas en la cual se deben expresar relaciones entre cosas de algún tipo. Quizá un buen ejemplo para este tiempo lo constituya un mapa vial que represente una red de carreteras entre ciudades. En la jerga computacional, a este tipo de gráfico se le llama formalmente “grafo”.

El término “grafo” proviene del griego γράφος (grafos), el cual significa escritura. La escritura es expresión sensorial<sup>1</sup> de eso tan maravilloso y misterioso como el lenguaje. Cuando leemos, no requerimos la presencia del escritor<sup>2</sup>, sino algún sentido de percepción y entrenamiento como lector. Los gráficos constituyen una de las primeras formas humanas de representar conocimiento y la computación y programación no escapan a la riqueza expresiva de lo gráfico.

En términos matemáticos, una relación es un subconjunto ( $\mathcal{R} \subseteq A \times B$ ) de algún producto cartesiano entre dos conjuntos  $A$  y  $B$ . En esta sección nos interesan relaciones entre cosas de un mismo tipo, por lo que los conjuntos origen y destino son los mismos y la relación, calificada de “binaria”, se refiere al producto cartesiano  $A \times A$ . Hay varios esquemas gráficos para expresar una relación binaria. En matemática es frecuente utilizar los diagramas sagitales, pero éstos son engorrosos para

---

<sup>1</sup>Bien sea visual, táctil o de alguna otra clase.

<sup>2</sup>Aunque pueden perderse algunas emociones e interpretaciones involucradas cuando se escucha directamente al lector.

las relaciones binarias. En lo que concierne este estudio existe una representación gráfica que permite mirar y entender mejor una relación. La figura 7.1 ilustra la relación  $R = \{(a, b), (a, c), (a, d), (b, a), (b, c), (b, d), (b, f), (b, e), (c, a), (c, b), (c, g), (c, f), (c, h), (d, a), (d, b), (d, e), (g, c), (g, h), (f, c), (f, b), (f, h)\} \subseteq A \times A$ ,  $A = \{a, b, c, d, e, f, g, h\}$ , de una manera diferente y, en general más simple, que un diagrama sagital.

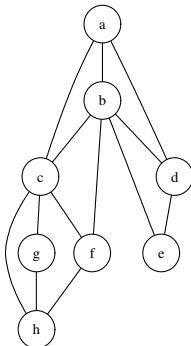


Figura 7.1: Ejemplo de relación expresada con un gráfico

Figuras como la 7.1-a se llaman “grafos”. En el ejemplo en cuestión se aprecian visualmente las conexiones entre los elementos del conjunto  $A$ . Los elementos se dibujan encerrados entre círculos, mientras que las relaciones entre ellos se indican mediante líneas que conectan los círculos. Es muy posible que al lector le sea más sencillo apreciar la relación mediante la figura 7.1 que en su representación matemática  $R = \{(a, b), (a, c), (a, d), (b, a), (b, c), (b, d), (b, f), (b, e), (c, a), (c, b), (c, g), (c, f), (c, h), (d, a), (d, b), (d, e), (g, c), (g, h), (f, c), (f, b), (f, h)\}, A = \{a, b, c, d, e, f, g, h\}$ .

En la jerga de grafos, los elementos de relación se llaman vértices o nodos, mientras que las relaciones se llaman aristas o arcos. En el ejemplo de la figura 7.1-a los vértices se dibujan con círculos etiquetados con los elementos del conjunto  $A = \{a, b, c, d, e, f, g, h\}$ , pero no existe ninguna restricción sobre la forma visual que deba tener un nodo; podría haberse dibujado con cuadrados, triángulos o cualquier otra figura.

Por lo general, en matemática, la ambigüedad no es bien vista. Pero a veces es justamente en la capacidad de expresar formas gráficas diferentes para una misma relación donde reside el poder de abstracción y representación de un grafo. Un grafo es, pues, una estructura de datos orientada hacia la representación gráfica de las relaciones entre elementos de un mismo tipo. La razón que motiva a los programadores, ingenieros, matemáticos y demás practicantes a utilizar grafos, es la comodidad que conlleva para el entendimiento el manejar abstracciones gráficas. Acabamos de presentar el para qué es la idea de grafo: una abstracción general que pretende abarcar una vasta gama de problemas en los cuales se puedan expresar relaciones entre elementos de un mismo tipo y nos sea más cómodo visualizarlas en un gráfico.

Si bien la idea de grafo es visual en su sentido sensorial, su tratamiento tiene dos perspectivas de estudio que no lo son: la matemática combinatoria y la programación.

El concepto de grafo y sus problemas asociados existen desde mucho antes que la programación. Puesto que la mayoría de los problemas que modelizan los grafos requieren mucho cómputo, durante mucho tiempo su interés y estudio estuvo reservado a los matemáti-

cos. La perspectiva matemática tiende a investigar el grafo como una relación, es decir, combinatorialmente.

La otra perspectiva del análisis la constituye la programación de algoritmos sobre grafos. Dichos algoritmos plantean desafíos de labor de una escala y complejidad que aún no ha sido abordada hasta el presente. Los algoritmos sobre grafos conjugan el uso de diversas estructuras de datos y algoritmos. Por adelantar un ejemplo, el cálculo de caminos óptimos puede requerir algoritmos de ordenamiento y heaps binarios.

## 7.1 Fundamentos

Como intuitivamente ya se ha mencionado, un grafo  $G$  se define formalmente mediante una dupla  $G = \langle V, A \rangle$  compuesta de dos conjuntos:  $V$  es el conjunto de vértices o nodos y  $A$  es el conjunto de aristas o arcos<sup>3</sup>. En el ejemplo de la figura 7.2,  $V$  se compone por las letras desde la  $A$  hasta la  $M$ , mientras que  $A$  lo componen las conexiones enumeradas desde el 1 hasta el 29.

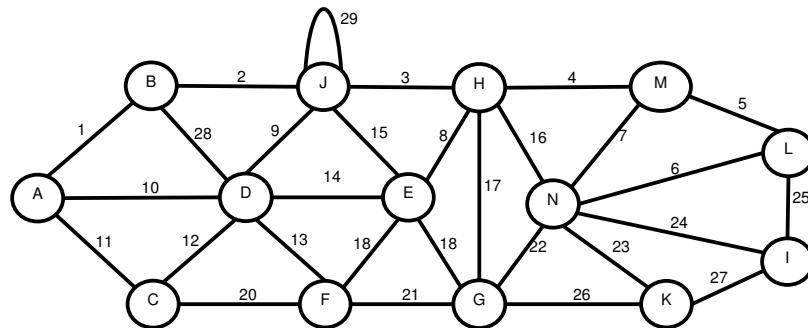


Figura 7.2: Un grafo

La numeración de los arcos de las figuras 7.2 y 7.3 no representa magnitudes o tipos. Según las características de la aplicación, a los nodos y arcos pueden asociárseles tipos de datos. Por ejemplo, un “grafo vial” de carreteras guardaría en sus nodos los nombres de las ciudades junto con información turística y de servicios básicos (estaciones de gasolina, hoteles, etcétera). Los arcos representarían las carreteras y almacenarían las distancias entre las ciudades junto con un calificador que indicase su calidad (autopista, carretera, trocha, etcétera). A menudo, a esta clase de grafo, en particular a aquel cuyos arcos guardan cantidades, se le llama “grafo con pesos” porque las cantidades se ponderan y se utilizan en algoritmos, por ejemplo, para calcular la distancia más corta entre dos ciudades.

En el grafo de la figura 7.2, cualquier arco relaciona a sus nodos en los dos sentidos. Por ejemplo, el arco 1 representa las relaciones  $A \rightarrow B$  y  $B \rightarrow A$  respectivamente. En términos matemáticos, el arco 1 representa  $(A, B), (B, A) \in A \times A$ . En palabras matemáticas, la relación representada por el grafo de la figura 7.2 es simétrica.

<sup>3</sup>Por lo general, los programadores usan el término “nodo”, mientras que los matemáticos prefieren el término “vértice”. Del mismo modo, los programadores utilizan el vocablo “arco” mientras que los matemáticos se inclinan por usar “arista”. Puesto que este texto es más para programadores, se usarán en preferencia los términos **nodo** y **arco** respectivamente.

Cuando la relación no es simétrica, entonces el grafo es “dirigido” y a los arcos se les ponen flechas que indican el sentido de la relación entre dos nodos. Un grafo dirigido también se denomina “digrafo”. La figura 7.3 muestra un ejemplo. Nótese que el arco 1 dirigido desde el nodo A hacia el nodo B representa  $(A, B) \in \mathcal{R}$ . Del mismo modo, nótese que  $(B, A) \notin \mathcal{R}$  se expresa por la ausencia del arco dirigido desde B hacia A.

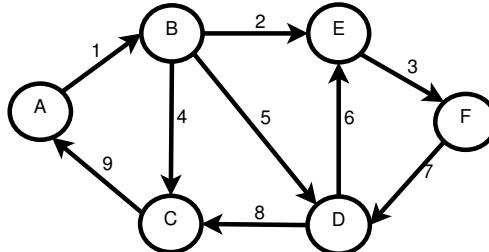


Figura 7.3: Un digrafo o grafo dirigido

El número de arcos conectados a un nodo es llamado el **grado del nodo**. El grado del grafo es el mayor grado entre todos sus nodos. Dado un nodo  $v$ , los nodos directamente conectados desde  $v$  a través de sus arcos se llaman nodos adyacentes. Por ejemplo, en el grafo de la figura 7.3, los nodos adyacentes a B son {C, D, E}. Los nodos desde los cuales se llega directamente a un nodo  $v$  se denominan nodos incidentes. Para el ejemplo anterior, A es el único nodo incidente a B. Un arco es incidente sobre el(los) nodo(s) que él conecta. Esta jerga es aplicable a un grafo no dirigido, pero puede complementarse si a los números de arcos entrantes y salientes se les llaman “grado de entrada” y “grado de salida” respectivamente.

Dado un arco dirigido, a su nodo de origen se le denomina “adyacente” y al destino “incidente”.

Un “camino”  $P \in G$  de un grafo  $G$  se define como una secuencia intercalada de nodos y arcos de  $G$ . Hay varias formas de expresar un camino, las cuales dependen del tipo de grafo y de la aplicación. En el caso del grafo de la figura 7.2, la secuencia  $P = A \rightarrow B \rightarrow J \rightarrow H \rightarrow N \rightarrow L \rightarrow I$  indica un camino desde el nodo A hasta el nodo I. Otra manera puede ser directamente A, B, J, H, N, L, I, o mediante las etiquetas de los arcos: 1, 2, 3, 16, 6, 25; esta última forma está supeditada a que los arcos estén etiquetados; requisito que no se circunscribe en la definición formal de grafo. La longitud de un camino se compone por el número de arcos que éste contenga, o sea, el número de nodos menos uno.

Una de las aplicaciones más importantes de los grafos es la búsqueda de caminos según alguna restricción dada. Quizá el problema más popular sea la búsqueda del camino mínimo entre un par de nodos.

Un camino cuyo primer y último nodo sean iguales se denomina **ciclo** o **circuito**. Si la longitud del ciclo es uno, entonces al camino se le califica de “simple”, de “bucle” o de “lazo”. El grafo de la figura 7.2 contiene un sólo ciclo simple en el nodo J; es decir, el par  $(J, J)$ .

Un grafo  $G = \langle V, A \rangle$  se califica de **conexo** si para todos los pares ordenados de nodos  $(v_1, v_2) \in G$  existe un camino desde  $v_1$  hasta  $v_2$ . Expresado de otra manera, un grafo es conexo si para todo nodo del grafo existe un ciclo que no sea simple.

Un grafo se dice que es **total**, **completamente conexo** o, simplemente, **completo**,

si cada nodo del grafo es adyacente al resto. El grado de un grafo completo es su número de nodos. La figura 7.4 ilustra el grafo completo de grado 5.

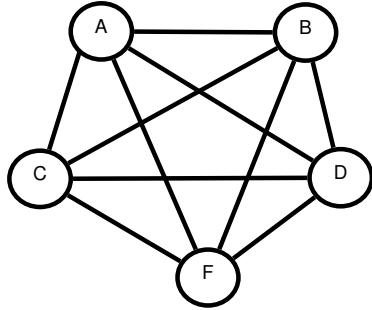


Figura 7.4: Un grafo completo de grado 5

**El grafo complemento** de un grafo  $G$ , que se denota  $\bar{G}$ , es el grafo compuesto por los arcos que se requerirían para que  $G$  deviniese completo. Dicho de otro modo, el grafo completo de grado  $|V|$  menos los arcos de  $G$ . La figura 7.5 ilustra un ejemplo. La unión de un grafo y su complemento es el grafo completo de grado  $|V|$ . Algunos algoritmos requieren calcular el grafo complemento; otros requieren el grafo completo, el cual puede calcularse si se tiene el complemento mediante la unión  $G \cup \bar{G}$ .

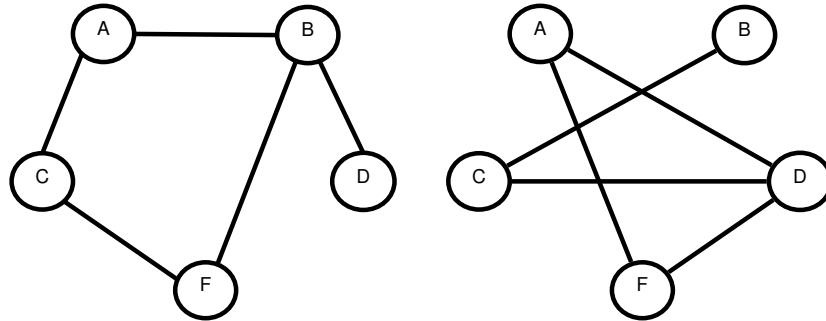


Figura 7.5: Un grafo y su complemento

Las escalas en número de nodos y arcos son los factores principales que inciden en el tiempo de ejecución de un algoritmo sobre un grafo. En las expresiones de cálculos, para facilitar el discurso, las cardinalidades de los conjuntos  $V$  y  $A$  suelen representarse directamente sin las barras.

A un grafo que tenga dos o más arcos que conecten los mismos nodos se le denomina, en paráfrasis con los conjuntos, “multigrafo”. La figura 7.6 ilustra un ejemplo. Un multigrafo puede ser dirigido, en cuyo caso se le llama **multidigrafo** o **multigrafo dirigido**. Un arco redundante recibe el nombre de “paralelo”.

Dado un grafo  $G = < V, A >$ , cualquier subconjunto  $G' = < V', A' > | (V' \subseteq V)(A' \subseteq A | \forall a = (v_1, v_2) \in A' \implies v_1, v_2 \in V')$  se llama **subgrafo** de  $G$ . La figura 7.7 ilustra dos subgrafos del ilustrado en la figura 7.2. Cualquier subgrafo completo se llama **clique**.

A un grafo que no es conexo se le llama “**inconexo**”. Un grafo inconexo  $G$  se compone de grafos conexos, los cuales se denominan “**subgrafos conexos de  $G$** ”.

Un grafo conexo sin ciclos se llama “**árbol**”. Menester notar que ésta es una interpretación diferente a la de árbol tratada en § 4.1 (Pág. 226), pues en este caso no es requisito denotar un nodo raíz, aunque nada impide hacerlo.

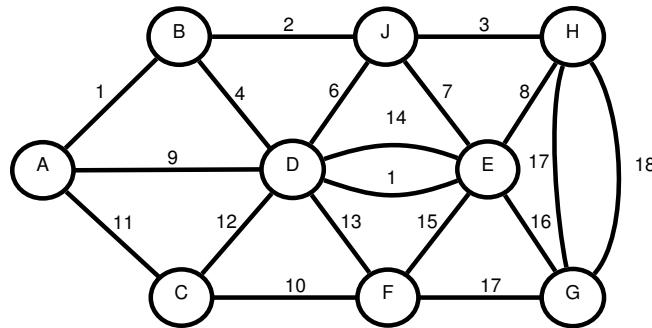


Figura 7.6: Un multigrafo

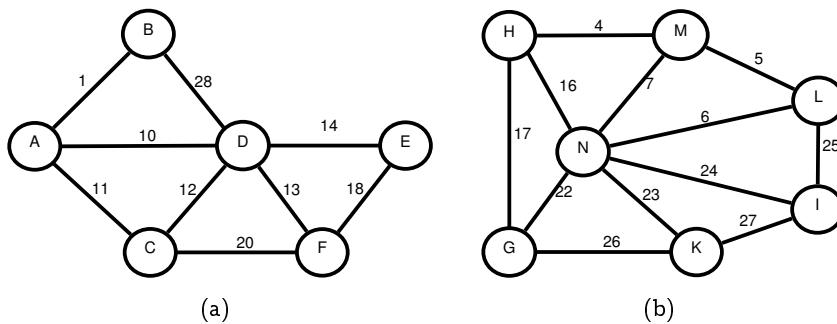


Figura 7.7: Dos subgrafos del grafo en 7.2

Un “árbol abarcador”<sup>4</sup> de un grafo conexo  $G$  es un árbol compuesto con arcos de  $G$  que contiene todos los nodos de  $G$ . La figura 7.8 ilustra un árbol abarcador del grafo mostrado en la figura 7.2.

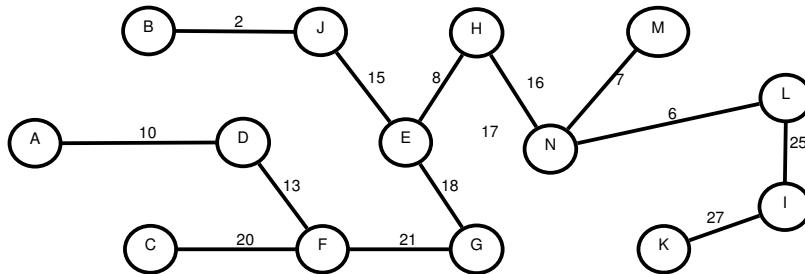


Figura 7.8: Un árbol abarcador del mostrado en la figura 7.2

Algunos algoritmos requieren verificar si el grafo es “bipartido” o encontrarlo. Un grafo es “bipartido” si puede dividirse en dos conjuntos disjuntos de nodos  $B$  y  $C$  tal que los arcos de un conjunto conecten nodos de  $B$  con nodos de  $C$  y que no exista conexión entre los nodos de un mismo conjunto. La figura 7.9 muestra un ejemplo.

Del sentido “visual” del concepto de grafo surge la idea de “coloración” si asociamos colores a los nodos o arcos. El problema de colorear nodos consiste en “pintar” los nodos de manera tal que dos nodos adyacentes no tengan el mismo color. En el mismo sentido,

<sup>4</sup>En inglés: “spanning tree”.

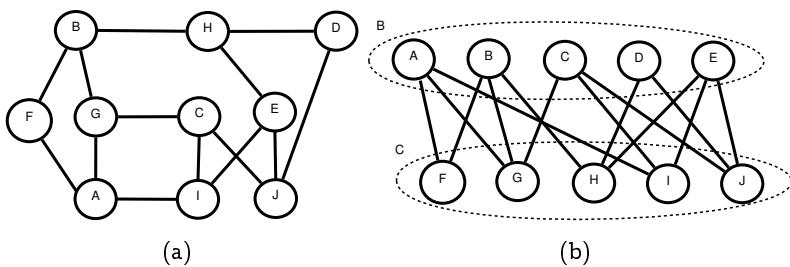


Figura 7.9: Un grafo bipartido

colorear arcos consiste en pintarlos de forma tal que dos arcos que inciden sobre el mismo nodo tengan colores diferentes.

Dos grafos  $G_1 = \langle V_1, A_1 \rangle$  y  $G_2 = \langle V_2, A_2 \rangle$  son isomorfos si es posible cambiar de las etiquetas de  $V_1$  por las de  $V_2$  tal que  $A_1 = A_2$ . La figura 7.10 muestra un ejemplo.

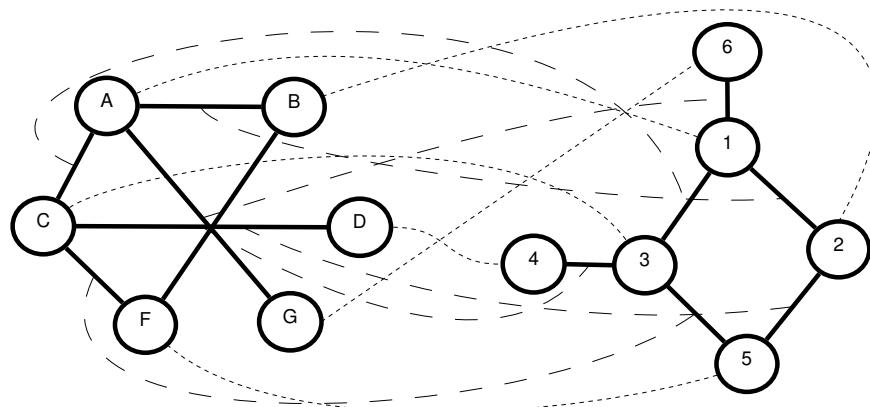


Figura 7.10: Dos grafos isomorfos

Un grafo es llamado planar si es posible dibujarlo en un plano sin que sus arcos se crucen.

Un camino “euleriano” es uno que pasa por todos los arcos exactamente una vez. Este tipo de camino tipifica algunas situaciones de optimización: por ejemplo, el recorrido de un cartero. Un camino “hamiltoniano” es uno que pasa por todos los nodos exactamente una vez. Al igual que la clase anterior, este tipo de camino también modeliza situaciones de optimización; por ejemplo, el camino que debe hacer un visitador médico para cubrir todas las ciudades de su responsabilidad.

Los grafos representan una clase de problemas más compleja que hace uso intensivo de implantaciones del problema fundamental de estructura de datos. La índole de problemas con grafos puede clasificarse en cuatro tipos.

El primer tipo de problema se clasifica en “de existencia”. Dado un grafo y unos requerimientos sobre éste, ¿existe un grafo que satisfaga los requerimientos? Por ejemplo, ¿es un grafo planar?

El segundo tipo se tipifica en “de construcción”. Una vez que se conoce la existencia de un grafo especial, entonces, ¿cómo construirlo?. Alguna veces no es necesario determinar la existencia porque ésta se conoce de antemano, por ejemplos, para calcular el árbol abarcador o algún camino. Otras veces sí conviene determinar la existencia primero,

antes de abordar la construcción, por instancia, el dibujado de un grafo planar.

Al tercer tipo se le denomina en “de enumeración” o “conteo”. Una vez que se determina la existencia, entonces se desea conocer la cantidad de soluciones. Por lo general, estos problemas atañen más al interés matemático que aplicativo.

Finalmente, el último tipo de problema se llama “de optimización” y consiste en construir la mejor solución según algún criterio de optimización. En lo que sigue de este texto se encontrarán muchos ejemplos variando desde el árbol abarcador mínimo hasta la construcción de un camino hamiltoniano.

## 7.2 Estructuras de datos para representar grafos

Se conocen dos esquemas para representar grafos en la memoria de un computador: matrices o listas enlazadas.

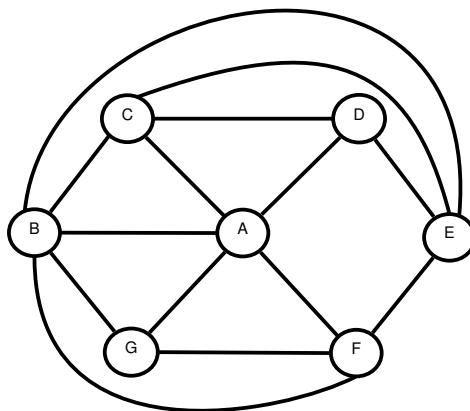


Figura 7.11: Un grafo ejemplo

### 7.2.1 Matrices de adyacencia

Puesto que un grafo representa visualmente una relación, las matrices parecen ser la estructura “natural”. De hecho, éstas tienen una tradición de uso exitosa en la matemática para expresar y manipular relaciones binarias y, consecuentemente, grafos.

Una matriz que represente a un grafo se llama “de adyacencia”. Se trata de una matriz cuadrada  $M[V \times V]$  cuyas entradas  $M(i, j)$  representan la conexión entre el nodo  $i$  y  $j$ . La mínima información necesaria en cada entrada es un bit que indique la existencia o no del arco. El grafo de la figura 7.11 se representa mediante la matriz mostrada en la figura 7.12. Un cero indica la ausencia de arco, mientras que un uno su presencia.

Si el tipo de dato asociado a un nodo no es entero, entonces hay que implantar un mapeo de nodos hacia índices de la matriz. La manera más simple de hacerlo es mediante un arreglo de nodos cuyos índices se corresponden con índices en la matriz. Si el arreglo está ordenado, entonces la búsqueda binaria explicada en § 3.1.9 (Pág. 165) encuentra muy eficazmente un índice dado un nodo.

Si los arcos no tienen algún tipo asociado, entonces el TAD BitArray, estudiado en § 2.1.5 (Pág. 53), puede utilizarse como base de implantación de la matriz de adyacencia. Por el contrario, si los arcos asocian algún tipo, entonces cada entrada de la

$$\begin{pmatrix} & \begin{matrix} A & B & C & D & E & F & G \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \\ G \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{pmatrix}$$

Figura 7.12: Matriz de adyacencia del grafo en la figura 7.11

matriz podría representar la instancia del tipo que estaría asociada al arco. En este caso, requerimos una instancia especial del tipo para indicar la ausencia del arco.

Una matriz de adyacencia ocupa  $\mathcal{O}(V^2)$  celdas, coste alto aun para grafos de mediana escala. Por esta razón resulta interesante el uso de estructuras de representación de matrices esparcidas que no ocupen espacio por entradas nulas, en nuestro caso, por ausencia de arcos.

Una primera manera de ahorrar memoria la constituye considerar el tipo de grafo. Si la matriz no representa a un digrafo, entonces basta con sólo guardar los elementos sobre o debajo de la diagonal. Del mismo modo, si el grafo no contiene lazos, entonces no es necesario considerar la diagonal.

Una matriz tradicional requiere un bloque de memoria contiguo proporcional a  $\mathcal{O}(V^2)$ . A medida que aumente  $V$  será más difícil para un manejador de memoria encontrar un bloque contiguo. Un esquema funcionalmente equivalente consiste en apartar un arreglo de arreglos cuya representación pictórica se muestra en la figura 7.13. En ese caso, el manejador de memoria requiere apartar  $V$  bloques contiguos de tamaño  $V$ , lo cual es más fácil que uno solo de  $V \times V$ . En C++, una matriz de adyacencia de arcos del tipo  $T$  se declara e inicia del siguiente modo:

543

*(inicializar arreglo de arreglos 543)*≡

```
T ** matriz = new T * [V];
for (int i = 0; i < V; ++i)
 matriz[i] = new T [V];
```

Este código puede modificarse para usar el tipo `BitArray`, lo cual se deja como ejercicio.

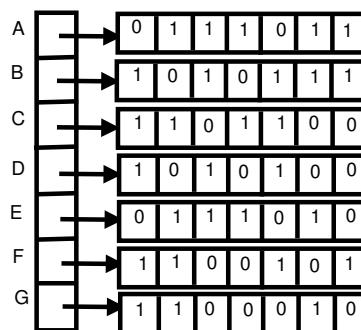


Figura 7.13: Arreglo de arreglos correspondiente a la matriz de adyacencia de la figura 7.11

Hay algoritmos que se desempeñan mejor con matrices de adyacencia que con su contraparte que usa listas enlazadas. Algunos algoritmos con matrices de adyacencia requieren otras matrices para mantener estado de sus cálculos, lo cual incrementa el coste en memoria.

A parte del aprovechamiento del tipo de grafo, otra forma de ahorrar espacio es usar un arreglo esparcido del tipo `DynArray<T>` estudiado en § 2.1.4 (Pág. 34).

Una matriz de adyacencia requiere conocer a priori el número de nodos. Por esta razón, esta representación no es conveniente para situaciones en las cuales al grafo se le inserten y eliminan nodos o arcos dinámicamente.

### 7.2.2 Listas de adyacencia

Dado un nodo  $v$  perteneciente a un grafo, su lista de adyacencia se compone por los nodos a los cuales  $v$  está conectado mediante arcos. De este modo, para representar un grafo en memoria se utilizan listas de adyacencia para cada nodo del grafo. La figura 7.14 ilustra un ejemplo.

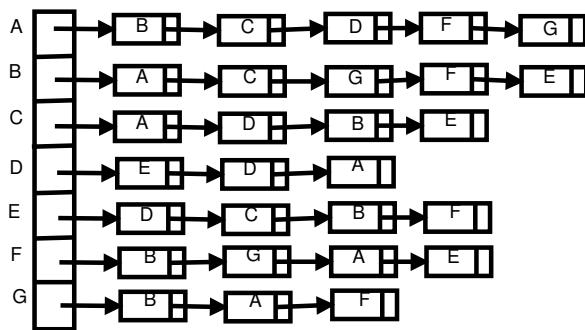


Figura 7.14: Listas de adyacencia del grafo en la figura 7.11

Cada nodo de una lista de adyacencia guarda la dirección de memoria de un vértice en el grafo<sup>5</sup> correspondiente a la existencia de un arco. Puesto que en un grafo, los arcos son bidireccionales, éstos se reflejan dos veces en la representación con listas de adyacencia, o sea, una vez por cada nodo extremo de cada arco. Por ejemplo, en el grafo de la figura 7.11, el arco  $A \leftrightarrow B$  aparece en las listas de adyacencia de los nodo A y B.

La duplicidad de arcos en las listas de adyacencia acarrea un sobrecoste de espacio cuando los arcos asocian algún tipo de dato, pues se duplica información. Por otra parte, algunos algoritmos guardan estado de cálculo en los arcos, lo que requiere, en caso de mantenerse la duplicidad mencionada, que el estado se actualice en dos direcciones de memoria diferentes. Por esta razón es preferible abstraer un arco y guardar direcciones a éste en las listas de adyacencia, en lugar de guardar direcciones de nodos.

De esta manera, la lista de adyacencia deviene una lista de arcos. La distinción del nodo destino se hace según el nodo propietario de la lista. En el ejemplo de la figura 7.11, si nos encontramos en la lista de adyacencia del nodo B y encontramos el arco  $A \leftrightarrow B$ , entonces sabemos que se trata del arco con sentido  $B \leftrightarrow A$ .

Con el esquema anterior se puede mantener estado en los arcos sin necesidad de lidiar

<sup>5</sup>Se usa “vértice” en lugar de “nodo” para distinguirlo de nodo en la lista de adyacencia.

con las duplicidades de nodos. Este esquema también funciona con digrafos a expensas de un costo en memoria ligeramente superior que el de guardar los nodos.

Las listas de adyacencia tienen varias ventajas respecto a las matrices. La primera de ellas es que el coste en espacio es exactamente proporcional al número de arcos. Esto se traduce en ahorros de memoria cuando el grafo es espaciado.

La segunda ventaja es que se facilita operar el grafo en función de su carácter gráfico; es decir, en función de nodos y arcos. Esto no sucede en general con un algoritmo que use una matriz de adyacencia, el cual tiene probablemente sentido matemático, pero no el gráfico que señalamos al principio del capítulo.

La última ventaja que apreciamos en la representación con listas es su flexibilidad para el manejo de memoria y dinamismo. Puesto que se aparta memoria en función de nodos y arcos, es más simple para el manejador de memoria obtener bloques, aun en escenarios en que haya poca memoria o ésta esté muy fragmentada. Por otro lado, por su carácter de listas, esta representación es mucho más dinámica para operaciones intercaladas de inserción y eliminación, sobre todo si las listas son doblemente enlazadas.

Como desventaja podemos identificar que la verificación de existencia de un arco es en el promedio o peor de los casos  $\mathcal{O}(V)$ , pues es necesaria una búsqueda en una lista de adyacencia. Con una matriz, en cambio, esta verificación es  $\mathcal{O}(1)$  o  $\mathcal{O}(\lg(n))$ .

Con una matriz de adyacencia, los arcos aparecen en el orden dado por sus filas (o columnas). En el ejemplo de la figura 7.11, los arcos hacia el nodo D siempre aparecen antes que los que van hacia el nodo E. Con listas de adyacencia, el orden de aparición de los arcos y, por tanto, de su orden de procesamiento, es relativo a su posición dentro de la lista de adyacencia. A la vez, el orden de aparición depende del orden de inserción en el grafo. Aparte de la suerte, por lo general, esto no parece tener implicaciones de desempeño.

### 7.3 Un TAD para grafos (List\_Graph)

En esta sección presentaremos el diseño e implantación del TAD List\_Graph, el cual pretende modelizar grafos generales que puedan usarse para la mayoría de clases de aplicaciones conocidas sobre grafos.

A pesar de que en su sentido visual, un grafo parece ser muy general, esta abstracción de concepto se usa para resolver problemas muy distintos. Consecuentemente, encontrar un patrón general que permita modelizar un TAD que funcione para todas las clases de problemas sobre grafos, es algo difícil. Además, como ya debe ser bien sabido, luego de insistir en el principio fin-a-fin (§ 1.4.2 (Pág. 22)), la generalidad tiene sus costes en el sentido de que algunas aplicaciones no requieren la maquinaria desplegada para tratar con todos los casos posibles.

El código pertinente a los TAD fundamentales sobre grafos se especifica en el archivo `tpl_graph.H`.

Podría decirse que una de las principales dificultades para modelizar un TAD de grafos es el hecho de que éste involucra otras clases de objeto. Esencialmente, un grafo maneja nodos y arcos, los cuales, en muchos casos, hay que tratarlos como una abstracción separada, aunque relacionada, del propio grafo. Un aspecto esencial para la compresión es la observancia de que es en el momento en que se define el grafo, es decir, cuando éste se declare y se compile, que se conocerán completamente los tipos de nodos y arcos.

Debido a su alto dinamismo, `List_Graph` constituye el principal TAD pertinente a los grafos. Este TAD permite copiar, asignar y modificar su topología; características necesarias para una amplia gama de algoritmos. Algoritmos basados en matrices de adyacencia usarán una familia de TAD orientados a matrices: `Map_Matrix_Graph<GT>`, `Matrix_Graph<GT>` y `Ady_Mat`, los cuales serán detalladamente desarrollados en § 7.6 (Pág. 628).

### 7.3.1 Grafos

Un `List_Graph` es una clase que modeliza un grafo implementado con listas de adyacencia. Sus parámetros tipo son el nodo y el arco, y se define del siguiente modo:

546a *(Grafos 546a)≡*

```
template <typename __Graph_Node, typename __Graph_Arc>
class List_Graph
{
 <Tipos de List_Graph 546b>
 <Miembros privados de List_Graph 559e>
 <Miembros públicos de List_Graph 547b>
 <Iteradores de List_Graph 550a>
};
```

Defines:  
`List_Graph`, used in chunks 547a, 550a, 555, 561, 562, 564–66, 571–74, 576a, and 577b.

Para poder usar un objeto `List_Graph` es necesario haber definido los tipos de nodos y de arcos, cuestión a la que nos abocaremos en las próximas subsecciones.

`List_Graph` será ampliamente utilizado para la programación genérica, es decir, para programar algoritmos que funcionen para clases generales de grafos. En la programación genérica es a menudo necesario conocer los subtipos de `List_Graph`, los cuales se definen a continuación

546b *(Tipos de List\_Graph 546b)≡* (546a)

```
typedef __Graph_Node Node;
typedef __Graph_Arc Arc;
typedef typename Node::Node_Type Node_Type;
typedef typename Arc::Arc_Type Arc_Type;
```

Consideremos el siguiente fragmento de código, el cual ejemplifica el uso de estos tipos:

```
template <class GT> void fct(GT & g)
{
 typename GT::Arc * arc;

 typename GT::Arc_Type dato;
 // ...
}
```

La función es genérica y recibe un `List_Graph` llamado `GT`. La primera línea declara un puntero a un arco, mientras que la siguiente declara una variable del mismo tipo que el atributo asociado al arco. Notemos que este código es genérico en el sentido de que debería de operar sobre cualquier combinación de tipos de nodos y arcos.

En lo que sigue de las siguientes subsecciones mencionaremos operaciones sobre `List_Graph` a nivel de interfaz y no de implantación.

### 7.3.2 Digrafos (List\_Digraph)

El TAD que modeliza digrafos se denomina List\_Digraph. Tanto a nivel de estructura de datos, como a nivel de interfaz, el List\_Graph tiene casi todo lo requerido para manejar digrafos. Por esa razón definiremos List\_Digraph por herencia pública de List\_Graph tal como se presenta a continuación:

547a *(Digrafos 547a)≡*

```
template <typename __Graph_Node, typename __Graph_Arc>
class List_Digraph : public List_Graph<__Graph_Node, __Graph_Arc>
{
 List_Digraph();
 List_Digraph(const List_Digraph & dg);
 List_Digraph & operator = (List_Digraph & dg);
};
```

Defines:  
*List\_Digraph*, used in chunks 562d and 711b.  
 Uses List\_Graph 546a.

La interfaz de List\_Digraph es idéntica a la de List\_Graph; de hecho, es, por derivación de clases, la misma. A nivel de implementación, la diferencia reside en que en List\_Graph se redunda el arco, sin afectar la coherencia de los algoritmos y con un ligero consumo de espacio, para representar la bidireccionalidad del arco, mientras que en List\_Digraph sólo se pone el arco en la lista de adyacencia de su nodo adyacente.

Puede haber situaciones en las cuales es necesario determinar si un objeto de tipo List\_Graph es o no un digrafo. Para eso se provee la primitiva siguiente:

547b *(Miembros públicos de List\_Graph 547b)≡* (546a) 548c▷

```
bool is_digraph() const;
```

la cual retorna true si el objeto es un digrafo y false de lo contrario.

### 7.3.3 Nodos

Para representar a un nodo de un grafo se utiliza el TAD Graph\_Node, cuya especificación general es como sigue:

547c *(Definición de nodo de grafo 547c)≡*

```
template <typename Node_Info> class Graph_Node : public Dlink
{
 typedef Graph_Node Node;
 typedef Node_Info Node_Type;
 <Miembros de Graph_Node 548a>
};
```

*(Definición de nodo de grafo 547c)* modeliza un nodo perteneciente a un grafo implementado mediante listas de adyacencia. Un Graph\_Node es una clase plantilla cuyo parámetro es el tipo de objeto asociado al nodo y que se denomina Node\_Info. Si se desea, por ejemplo, un grafo de ciudades, entonces la línea siguiente:

```
Graph_Node<Ciudad> * nodo;
```

Declara un puntero a nodo con atributo de tipo Ciudad.

Eventualmente, si no se requiere ningún atributo para un nodo o si se prefiere colocarlo en una derivación, entonces podemos especificarlo con atributos vacíos de la siguiente forma:

```
Graph_Node<Empty_Class> node;
```

El atributo de un nodo se guarda en el miembro dato `node_info`, el cual se declara como sigue:

548a *(Miembros de Graph\_Node 548a)* ≡  
 Node\_Info node\_info;

(547c) 548b▷

y que se observa y modifica mediante:

548b *(Miembros de Graph\_Node 548a)* +≡  
 Node\_Info & get\_info() { return node\_info; }

(547c) ◁548a 557a▷

Dentro de un `List_Graph`, cada nodo pertenece a una lista circular doblemente enlazada de nodos. El enlace dentro de esa lista es `this` mediante herencia pública de `Dlink`.

### 7.3.3.1 Inserción de nodos

Para la comprensión de esta subsección y de las subsiguientes es necesario aprehender que un grafo `List_Graph` usa objetos de tipo `List_Graph::Node` y no estrictamente de tipo `Graph_Node`.

Aunque sin duda deberá basarse en `Graph_Node`, `List_Graph::Node` podría ser una derivación. Por esta razón, a efectos de la completitud de tipos es necesario insistir en que en un `List_Graph` el nodo es de tipo `List_Graph::Node`.

Hecha la aclaratoria anterior podemos hablar de la inserción de un nodo en un `List_Graph`. Hay dos operaciones:

548c *(Miembros públicos de List\_Graph 547b)* +≡  
 inline virtual Node \* insert\_node(Node \* node);

(546a) ◁547b 548d▷

inline virtual Node \* insert\_node(const Node\_Type & node\_info);

La primera versión toma un nodo ya creado, es decir, cuya construcción ya fue efectuada, y lo inserta dentro del grafo. La segunda aparta la memoria para un objeto de tipo `List_Graph::Node`, le asigna el atributo `node_info` y luego lo inserta dentro del grafo.

Si no hay memoria suficiente para crear el nodo, entonces se genera la excepción estándar `bad_alloc`.

A efectos de no ralentizar el desempeño dinámico de `List_Graph`, `insert_node()` no hace ninguna validación de correctitud, por ejemplo, la doble inserción.

### 7.3.3.2 Eliminación de nodos

Para eliminar un nodo de un grafo sólo basta tener un puntero a él:

548d *(Miembros públicos de List\_Graph 547b)* +≡  
 inline virtual void remove\_node(Node \* node);

(546a) ◁548c 549a▷

`remove_node()` elimina del grafo el nodo `node` junto a todos sus arcos incidentes y adyacentes. Toda la memoria ocupada por el nodo y sus arcos es liberada.

Bajo el mismo espíritu de no gastar ciclos en validación, `remove_node()` no efectúa verificación de correctitud; por ejemplo, la eliminación de un nodo que no pertenezca al grafo<sup>6</sup>.

---

<sup>6</sup>Las primeras versiones de `List_Graph` contenían invariantes que en cierto modo permitían validar su uso. No obstante, se eliminaron porque la degradación de desempeño para algoritmos prototípicos era tan severa que impedía una codificación productiva.

### 7.3.3.3 Acceso a los nodos de un grafo

Si bien las operaciones de inserción devuelven el puntero al nodo insertado, lo cual permite “recordarlo”, puede requerirse acceso a los nodos desde un List\_Graph. La primitiva básica para obtener un nodo cualquiera del grafo es:

549a *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 548d 549b ▷  
 inline Node \* get\_first\_node();

La cual retorna un nodo cualquiera contenido dentro del grafo.

`get_first_node()` es un punto cualquiera de acceso al grafo. No debe hacerse ninguna consideración acerca del nodo que retornará esta primitiva.

El número de nodos que contiene un List\_Graph puede conocerse mediante:

549b *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 549a 549c ▷  
 inline const size\_t & get\_num\_nodes() const;

### Búsqueda de nodos

Dado un grafo, puede plantearse la búsqueda de algún nodo que reuna algunas características. List\_Graph ofrece tres formas de búsqueda, secuenciales, es decir cuyo coste es  $\mathcal{O}(n)$  ( $|V| = n$ ), por lo que no se recomienda su uso si la búsqueda es muy frecuente.

Es menester señalar que si la aplicación requiere búsquedas frecuentes, entonces es preferible indizar los nodos en alguna estructura de datos especial, una tabla hash o alguna clase de árbol binario de búsqueda.

El primer tipo de búsqueda está dado por:

549c *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 549b 549d ▷  
 template <typename T, class Equal> Node \* search\_node(const T & data);

En este caso se hace una búsqueda secuencial mediante invocación al criterio de igualdad `Equal()` (`p, data`), donde `p` es una referencia al atributo asociado al nodo. La clase `Equal` permite establecer algún criterio selectivo sobre alguna parte de los atributos. Si se desean comparar enteramente otros, entonces se puede usar la siguiente versión por omisión:

549d *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 549c 549e ▷  
 Node \* search\_node(const Node\_Type & node\_info);

la cual invoca al operador `==` de la clase `Node_Type`.

Otro tipo de búsqueda consiste en verificar si algún nodo particular es o no parte del grafo:

549e *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 549d 549f ▷  
 bool node\_belong\_to\_graph(Node \* node);

`node_belong_to_graph()` retorna `true` si `node` es parte del grafo, y `false` de lo contrario.

A veces se requiere buscar un nodo con características que no son parte de sus atributos típicos (clase `Node::Node_Type` o `Node_Info`). Por ejemplo, alguna información incluida por derivación de `Graph_Node`. Puesto que no hay manera de conocer a priori estas clases de circunstancias, la única forma que se nos ocurre para comunicar información sobre la búsqueda es con un puntero opaco. Así pues, nuestra última clase de búsqueda se efectúa de la siguiente forma:

549f *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 549e 552b ▷  
 template <class Equal> Node \* search\_node(void \* ptr);

la cual, usando el criterio de igualdad `bool Equal::operator() (Node * curr, void * ptr)`, busca un nodo con características especificadas en el apuntador `ptr`.

### Iterador de nodos

Dado un `List_Graph` hay una manera de recorrer todos sus nodos. Para eso, `List_Graph` exporta la siguiente subclase:

550a *(Iteradores de List\_Graph 550a)≡* (546a) 550b▷

```
class Node_Iterator : public Dlink::Iterator
{
 inline Node_Iterator() {}
 inline Node_Iterator(List_Graph & _g);
 inline Node_Iterator(const Node_Iterator & it);
 inline Node_Iterator & operator = (const Node_Iterator & it);

 inline Node * get_current_node();
 Node * get_current() { return get_current_node(); }
};
```

Uses `List_Graph` 546a.

La operación `get_current_node()` retorna el nodo actual en donde se encuentra el iterador. Por razones de compatibilidad, el método tradicional `get_current()` se sobrecarga para que invoque a `get_current_node()`.

`Node_Iterator` recorre todos los nodos del grafo independientemente de sus relaciones de conectividad. Puesto que deriva de `Dlink::Iterator`, las operaciones de desplazamiento y ubicación son las mismas: `next()`, `prev()` y `has_current()`.

El orden de visita de `Node_Iterator` debe considerarse indeterminado.

La siguiente función exemplifica el recorrido de todos los nodos del grafo y la impresión, para cada uno de ellos, del número de arcos:

```
template <class GT> void recorrer(GT & g)
{
 for (typename GT::Node_Iterator it(g); it.has_current(); it.next())
 print("%d\n", g.get_num_arc(it.get_current_node()));
```

### Iterador de arcos de un nodo

Existe una clase de iterador el cual recorre los arcos adyacentes de un nodo. Posiblemente, este es el iterador más popular e importante, pues es la base de cualquier recorrido de cómputo sobre un grafo. Su especificación es como sigue:

550b *(Iteradores de List\_Graph 550a)+≡* (546a) ▷550a 555▷

```
class Node_Arc_Iterator : public Dlink::Iterator
{
 <Miembros privados de iterador de Node 570b>
 <Miembros públicos de iterador de Node 550c>
};
```

Las formas de declarar (construir) un objeto `Node_Arc_Iterator` se expresan mediante los siguientes constructores:

550c *(Miembros públicos de iterador de Node 550c)≡* (550b) 551a▷

```
inline Node_Arc_Iterator();
```

```
inline Node_Arc_Iterator(Node * _src_node);
inline Node_Arc_Iterator(const Node_Arc_Iterator & it);
```

Interesante observar que el objeto asociado a un Node\_Arc\_Iterator es un nodo perteneciente a un grafo y no el grafo mismo.

Node\_Arc\_Iterator requiere el operador de asignación, de modo tal que se puedan copiar estados temporales de iteración. Para eso requerimos el operador =:

551a *(Miembros públicos de iterador de Node 550c) +≡ (550b) ↣ 550c 551b ▷*  
`inline Node_Arc_Iterator & operator = (const Node_Arc_Iterator & it);`

Puesto que Node\_Arc\_Iterator es, por derivación, del tipo Dlink::Iterator, éste contiene todos sus métodos asociados (next(), prev(), has\_current(), etcétera). El objeto actual del iterador se accede mediante:

551b *(Miembros públicos de iterador de Node 550c) +≡ (550b) ↣ 551a 570c ▷*  
`typedef Arc * Item_Type;
typedef Node * Set_Type;`

```
inline Arc * get_current_arc();
Arc * get_current() { return get_current_arc(); }
inline Node * get_tgt_node();
```

get\_current\_arc() retorna el arco actual (de tipo Arc), mientras que get\_tgt\_node() retorna el nodo destino del arco actual (el nodo origen es el aquél sobre el cual se itera). get\_tgt\_node() es muy útil porque en este caso sí se distingue el nodo destino, es decir, aquél que está conectado al extremo del arco actual del nodo origen sobre el cual se está iterando.

Similar al iterador sobre nodos, el método get\_current() está sobrecargado para que invoque a get\_current\_arc().

Node\_Arc\_Iterator es el principal mecanismo para manipular grafos implantados mediante listas enlazadas. Es importante destacar que los arcos de un nodo se visitan en un orden indeterminado, el cual debe considerarse aleatorio por cualquier aplicación que haga uso de Node\_Arc\_Iterator.

### 7.3.4 Arcos

Un arco de un grafo se define mediante el TAD Graph\_Arc como sigue:

551c *(Definición de arco de grafo 551c) ≡*  
`template <typename Arc_Info> class Graph_Arc : public Dlink
{
 typedef Graph_Arc Arc;
 typedef Arc_Info Arc_Type;`

*(Atributos de Graph\_Arc 557b)*  
*(Métodos de Graph\_Arc 552a)*  
`};`

Graph\_Arc modeliza un arco perteneciente a un grafo implantado mediante listas de adyacencia. Al igual que Graph\_Node, Graph\_Arc es una clase parametrizada cuyo tipo asociado, Arc\_Info, constituye la información relacionada al arco y a la cual se accede

mediante:

552a *(Métodos de Graph\_Arc 552a)* ≡  
*Arc\_Info & get\_info() { return arc\_info; }* (551c) 566c▷  
 Si se desea, por ejemplo, un grafo de carreteras, entonces la siguiente línea:

*Arc<Carretera> \* arco;*

Declara un apuntador a un arco que usa como atributo un tipo Carretera.

Un arco conecta dos nodos llamados origen y destino que pueden accederse de la siguiente manera:

552b *(Miembros públicos de List\_Graph 547b)* +≡  
*inline Node \* get\_src\_node(Arc \* arc);* (546a) ◁549f 552c▷  
*inline Node \* get\_tgt\_node(Arc \* arc);*

Es momento de aclarar dos cosas en relación a los métodos sobre un arco:

1. En el caso de un grafo, estos métodos no necesariamente tienen sentido según la dirección del arco; simplemente conforman una manera de conocer los nodos que éste conecta. En el caso de un digrafo, por supuesto, *get\_src\_node()* retorna el nodo adyacente y *get\_tgt\_node()* el incidente.
2. Ambas primitivas, así como la mayoría de las de acceso a un arco, son parte de la clase *List\_Graph* y no de la clase *Graph\_Arc*. Esto implica que debe conocerse el grafo para poder invocar a uno de estos métodos.

Algo que sí tiene sentido es conocer cuál es el nodo que conecta un arco dado un nodo origen. Para eso se provee la siguiente primitiva:

552c *(Miembros públicos de List\_Graph 547b)* +≡  
*inline Node \* get\_connected\_node(Arc \* arc, Node \* node);* (546a) ◁552b 552d▷

La cual siempre retorna el nodo conectado a *node*, por el arco *arc*, independientemente de que sea un grafo o digrafo.

Ocasionalmente puede ser necesario indagar si un nodo está o no relacionado con otro a través de un arco. Para eso se proporciona la primitiva siguiente:

552d *(Miembros públicos de List\_Graph 547b)* +≡  
*inline bool node\_belongs\_to\_arc(Arc \* arc, Node \* node) const;* (546a) ◁552c 552e▷

La cual retorna *true* si *node* está conectado al arco *arc*.

#### 7.3.4.1 Inserción de arcos

Para definir un arco en un grafo deben haberse definido e insertado en el grafo sus dos nodos mediante las primitivas correspondientes definidas en § 7.3.3.1 (Pág. 548). Con las direcciones de los nodos puede invocarse la siguiente primitiva:

552e *(Miembros públicos de List\_Graph 547b)* +≡  
*inline virtual Arc \**  
*insert\_arc(Node \* src\_node, Node \* tgt\_node,*  
*const typename Arc::Arc\_Type & arc\_info);* (546a) ◁552d 553a▷

`src_node` y `tgt_node` son los nodos que relaciona el arco. Si se trata de un grafo, entonces el orden entre los nodos no tiene importancia. Si, por el contrario, se trata de un digrafo, entonces `src_node` es el nodo adyacente y `tgt_node` el incidente. `arc_info` es el valor de atributo del arco con que se desea crearlo.

`insert_arc()` aparta la memoria requerida para el arco, construye el arco para los valores especificados por los parámetros, lo inserta en el grafo y retorna la dirección de memoria del nuevo arco. Si no hay suficiente memoria se genera la excepción `bad_alloc`.

#### 7.3.4.2 Eliminación de arcos

Para eliminar un arco sólo se requiere conocer su dirección y entonces valerse del método siguiente:

553a *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 552e 553b ▷  
 inline virtual void remove\_arc(Arc \* arc);

el cual elimina del grafo el arco `arc` y libera la memoria.

El método no efectúa ninguna verificación de correctitud; por ejemplo, la eliminación de un arco que no existe en el grafo.

Los métodos que alteran la topología, entiéndase, `insert_node()`, `remove_node()`, `insert_arc()` y `remove_arc()` son virtuales. Esto es deseable para su reuso funcional por parte de extensiones o especializaciones derivadas. Un uso de ello es presentado en § 7.10.4 (Pág. 715).

Algunos algoritmos sobre grafos requieren eliminar conjuntos de arcos, hacer un cálculo parcial y luego restaurarlos. En estas situaciones es deseable que la eliminación de arco sea topológica, de carácter temporal, y no completa como se plantea con `remove_arc()`. Para eso se provee la siguiente primitiva:

553b *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 553a 553c ▷  
 inline virtual void disconnect\_arc(Arc \* arc);

Un arco desconectado mediante `disconnect_arc()` puede conectarse de nuevo a través de:

553c *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 553b 553d ▷  
 inline virtual Arc \* connect\_arc(Arc \* arc);

#### 7.3.4.3 Acceso a los arcos de un grafo

Una manera primigenia de obtener un arco cualquiera de un grafo la conforma el método:

553d *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 553c 553e ▷  
 inline Arc \* get\_first\_arc();

el cual retorna un arco cualquiera del grafo. No deben hacerse suposiciones acerca de cuál arco será retornado. Sin embargo, los arcos dentro del grafo pueden ordenarse mediante:

553e *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 553d 554a ▷  
 template <class Compare> inline void sort\_arcs();

`sort_arcs<Compare>()` ordena los arcos según el criterio de comparación `Compare`, el cual invoca al operador `bool Compare::operator () (Arc * a1, Arc * 2)`. El programador es responsable de implantar el acceso al atributo del arco por el cual se compara y de la comparación misma. El ordenamiento se efectúa en  $\mathcal{O}(n \lg(n))$  con consumo de espacio  $\mathcal{O}(1)$ .

El número de arcos de un grafo puede conocerse mediante:

554a  $\langle Miembros\ públicos\ de\ List\_Graph\ 547b \rangle + \equiv$  (546a)  $\triangleleft 553e\ 554b \triangleright$   
 $\text{inline const size\_t\ & get\_num\_arcs()\ const;}$

Y el número de arcos que tiene un nodo, mediante

554b  $\langle Miembros\ públicos\ de\ List\_Graph\ 547b \rangle + \equiv$  (546a)  $\triangleleft 554a\ 554c \triangleright$   
 $\text{inline const size\_t\ & get\_num\_arcs(Node\ *\ node)\ const;}$

### Búsqueda de arcos

Para los arcos existen cuatro tipos de búsqueda, los cuales se enuncian a continuación:

1. Búsqueda por atributo `Arc::Arc_Type` o `Arc_Info`<sup>7</sup>: en este caso podemos usar:

554c  $\langle Miembros\ públicos\ de\ List\_Graph\ 547b \rangle + \equiv$  (546a)  $\triangleleft 554b\ 554d \triangleright$   
 $\text{template <typename T, class Equal> Arc\ *\ search\_Arc(const\ T\ &\ data);}$

La primera versión usa el criterio de comparación `Equal`, lo cual permite discernir alguna parte de los atributos en la búsqueda. La segunda versión invoca directamente al operador `bool Node_Type::operator == (const Node_Type & info)`.

2. Prueba de pertenencia realizada por:

554d  $\langle Miembros\ públicos\ de\ List\_Graph\ 547b \rangle + \equiv$  (546a)  $\triangleleft 554c\ 554f \triangleright$   
 $\text{bool arc\_belong\_to\_graph(Arc\ *\ arc);}$

Se retorna `true` si un arco con dirección `arc` pertenece al grafo, y `false` de lo contrario.

3. Existencia de un arco entre dos nodos: para ello se usa la siguiente función:

554e  $\langle Funciones\ de\ List\_Graph\ 554e \rangle \equiv$  560b  
 $\text{template <class GT, class SA> typename GT::Arc\ *\}$   
 $\text{search\_arc(GT\ &\ g, typename GT::Node\ *\ src\_node,}$   
 $\text{typename GT::Node\ *\ tgt\_node);}$

El cual retorna la dirección de un arco en caso de existir alguno entre los nodos `src_node` y `tgt_node`.

A diferencia de sus rutinas predecesoras, que son métodos de `List_Graph`, esta versión es una función externa a la clase. Esto permite especializar versiones particulares de `search_arc()` que disciernan los arcos según algún criterio.

La primitiva funciona para digrafos.

En caso de tratarse de un multigrafo (o multidigrafo) y haber más de un arco entre los nodos involucrados, se retorna cualquiera entre los redundantes sin consideración de algún criterio específico.

4. Búsqueda especializada: este es el caso cuando en la búsqueda se desea distinguir algún dato que no es parte de los atributos del arco (`Arc_Type`).

554f  $\langle Miembros\ públicos\ de\ List\_Graph\ 547b \rangle + \equiv$  (546a)  $\triangleleft 554d\ 557c \triangleright$   
 $\text{template <class Equal> Arc\ *\ search\_arc(void\ *\ ptr);}$

<sup>7</sup>Recordemos que son sinónimos.

La rutina retorna la dirección del arco que satisface igualdad dada por bool Equal::operator () (Arc \* arc, void \*ptr), o NULL en caso de que se hayan recorrido todos los arcos sin satisfacer el criterio de igualdad.

Todas las búsquedas son  $\mathcal{O}(E)$  para el peor caso (búsqueda fallida).

## Iterador sobre los arcos de un grafo

Los arcos de un grafo pueden observarse mediante el siguiente iterador:

```

555 <Iteradores de List_Graph 550a>+≡ (546a) ◊550b
 class Arc_Iterator : public Dlink::Iterator
 {
 Arc_Iterator() {}
 Arc_Iterator(List_Graph & _g);
 Arc_Iterator(const Arc_Iterator & it);
 Arc_Iterator & operator = (const Arc_Iterator & it);

 Arc * get_current_arc();
 Arc * get_current() { return get_current_arc(); }

 Node * get_src_node() { return (Node*) get_current_arc()->src_node }
 Node * get_tgt_node() { return (Node*) get_current_arc()->tgt_node
 };

```

Al igual que con los iteradores de este texto, `Arc_Iterator` contiene las funciones típicas `next()`, `prev()` y `has_current()` (por derivación de `Dlink::Iterator`). La obtención del arco de visita actual se lleva a cabo mediante `get_current_arc()`.

Si los arcos no han sido ordenados, entonces el orden de visita es indeterminado; de lo contrario, el iterador visitará los arcos según el orden establecido en la última invocación a `sort_arcs()`.

### 7.3.5 Atributos de control de nodos y arcos

En muchos casos, los algoritmos sobre grafos requieren mantener un estado de cálculo en sus nodos y arcos. Quizá el caso más común sea “marcar” o “pintar” como visitado un nodo o un arco.

En este diseño contemplamos tres maneras de llevar estado, tanto en los nodos como en los arcos: bits de control, contadores y “cookies”. Cada clase de estado se guarda directamente en el nodo o en el arco.

### 7.3.5.1 Bits de control

A efectos de “pintar” nodos y arcos sólo se requiere de un bit. Podríamos mantener un bit por cada nodo y arco en un valor cero y, al procesarlo según el interés del algoritmo, pintarlo con uno. El problema de esta técnica es que los algoritmos que usen el bit devienen no-reentrantes. Por ejemplo, una prueba de conectividad puede inspeccionar los nodos a través de los arcos y pintarlos a medida que los visita. El grafo sería conexo si se visitan todos los nodos, es decir, si al final de la iteración están todos los nodos pintados. Ahora bien, otro algoritmo que invoque pruebas de conectividad no puede usar el mismo

bit usado por la prueba de conectividad para pintar sus nodos, pues comprometería la consistencia del test de conectividad.

Para tratar con la situación anterior usaremos varios bits clasificados según el algoritmo que los utilice. Éstos se especifican como sigue:

556a *(Definición de Bits de control 556a)≡*  
*(Número de bit 556d)*  
 class Bit\_Fields  
 {  
 unsigned int depth\_first : 1;  
   // .. declaación de los siguientes bits  
  
*(Métodos bits control 556b)*  
 };

Defines:

Bit\_Fields, used in chunk 557.

Bit\_Fields es un atributo que maneja como mínimo 16 bits destinados a llevar estado de visita.

Cualquier bit de control puede consultarse mediante:

556b *(Métodos bits control 556b)≡* (556a) 556c▷  
 bool get\_bit(const int & bit) const  
 {  
 switch (bit)  
 {  
 case Aleph::Depth\_First: return depth\_first;  
 // ...  
 }  
 }

Así como también, eventualmente, puede modificarse a través de:

556c *(Métodos bits control 556b)+≡* (556a) ▷556b  
 void set\_bit(const int & bit, const int & value)  
 {  
 switch (bit)  
 {  
 case Aleph::Depth\_First: depth\_first = value; break;  
 // ...  
 }  
 }

Los bits de control están enumerados “mágicamente”<sup>8</sup> según su fin pretendido por otros algoritmos:

556d *(Número de bit 556d)≡* (556a)  
 enum Graph\_Bits  
 {  
 Depth\_First,  
 Breadth\_First,  
 // ...  
 };

---

<sup>8</sup>En la jerga de programación, un número mágico es uno que es nombrado con un identificador.

Por ejemplo, el algoritmo de prueba de ciclo invocará:

```
control_bits.set_bit(Test_Cycle, 1);
```

a efectos de pintar una visita.

Cada nodo y arco de un grafo contiene bits de control especificados como sigue:

557a *(Miembros de Graph\_Node 548a) +≡* (547c) ◁ 548b 558a ▷  
 Bit\_Fields control\_bits;  
 Uses Bit\_Fields 556a.

557b *(Atributos de Graph\_Arc 557b) +≡* (551c) 558b ▷  
 Bit\_Fields control\_bits;  
 Uses Bit\_Fields 556a.

En un nodo los bits de control pueden accederse mediante:

557c *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 554f 557d ▷  
 inline Bit\_Fields & get\_control\_bits(Node \* node);  
 Uses Bit\_Fields 556a.

También pueden colocarse todos los bits en cero mediante:

557d *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 557c 557e ▷  
 inline void reset\_bit(Node \* node, const int & bit);

O asignarse un bit en particular a través de:

557e *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 557d 557f ▷  
 inline void set\_bit(Node \* node, const int & bit, const int & value);

Análogamente, existen las mismas operaciones para un arco:

557f *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 557e 557g ▷  
 inline Bit\_Fields & get\_control\_bits(Arc \* arc);

```
inline void reset_bit(Arc * arc, const int & bit);
```

```
inline void set_bit(Arc * arc, const int & bit, const int & value);
```

Uses Bit\_Fields 556a.

Se pueden reiniciar los bits de control de todos los nodos o arcos de un grafo mediante:

557g *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁ 557f 558c ▷  
 void reset\_bit\_nodes(const int & bit)  
 {  
 for (Node\_Iterator itor(\*this); itor.has\_current(); itor.next())  
 reset\_bit(itor.get\_current\_node(), bit);  
 }  
 void reset\_bit\_arcs(const int & bit)  
 {  
 for (Arc\_Iterator itor(\*this); itor.has\_current(); itor.next())  
 reset\_bit(itor.get\_current\_arc(), bit);  
 }

Los cuales colocan en cero, sea para los nodos o arcos, al “bit-ésimo” bit de control.

### 7.3.5.2 Contadores

El segundo nivel en manejo de estado para un nodo o arco es un contador entero. Por lo general, éste se usa para determinar la cantidad de visitas o para marcar “colores”. Su definición para un nodo es como sigue:

```
558a <Miembros de Graph_Node 548a>+≡ (547c) ◁557a 559a▷
 long counter;

y para un arco:
558b <Atributos de Graph_Arc 557b>+≡ (551c) ◁557b 559b▷
 long counter;

 El contador puede accederse o reiniciarse (colocarse en cero) para los nodos:
558c <Miembros públicos de List_Graph 547b>+≡ (546a) ◁557g 558d▷
 inline long & get_counter(Node * node);
 inline void reset_counter(Node * node);

O para los arcos:
558d <Miembros públicos de List_Graph 547b>+≡ (546a) ◁558c 558e▷
 inline long & get_counter(Arc * arc);

 inline void reset_counter(Arc * arc);

 Igualmente podemos reiniciar los contadores (colocarlos en cero) de todos los nodos o
arcos:
558e <Miembros públicos de List_Graph 547b>+≡ (546a) ◁558d 559c▷
 void reset_counter_nodes()
 {
 for (Node_Iterator itor(*this); itor.has_current(); itor.next())
 reset_counter(itor.get_current_node());
 }
 void reset_counter_arcs()
 {
 for (Arc_Iterator itor(*this); itor.has_current(); itor.next())
 reset_counter(itor.get_current_arc());
 }
```

### 7.3.5.3 Cookies

Existen muchos algoritmos que requieren asociar estado de cálculo de una manera muy particular al tipo de algoritmo y para los cuales los bits de control y contadores no son suficientes. La búsqueda del camino más corto según Dijkstra, por ejemplo, necesita guardar cierta información temporal en los arcos. La forma y uso de este estado es particular al algoritmo de Dijkstra y posiblemente no sirve para otros algoritmos.

Requerimos entonces de una forma más amplia de asociar estado genérico. Puesto que en muchas ocasiones, el estado es temporal, es decir, sólo se requiere durante la ejecución del algoritmo, no es conveniente parametrizarlo en una plantilla que se le pase al nodo o arco cuando éstos se instancien.

Una forma genérica, pero delicada, para asociar genéricamente datos a los nodos es a través de un “cookie”. El término cookie proviene de la programación de sistemas y connota a un parámetro opaco que se le transmite a una función. En nuestro caso, un cookie es un puntero opaco que se asocia a un nodo o a un arco.

Así pues, los nodos y los arcos poseen un atributo cookie, cuya especificación es la siguiente:

```
559a <Miembros de Graph_Node 548a>+≡ (547c) ◁558a 567b▷
 void * cookie;

559b <Atributos de Graph_Arc 557b>+≡ (551c) ◁558b 566a▷
 void * cookie;
```

Los cookies pueden accederse mediante las siguientes primitivas:

```
559c <Miembros públicos de List_Graph 547b>+≡ (546a) ◁558e 559d▷
 inline void *& get_cookie(Node * node);
 inline void *& get_cookie(Arc * arc);

get_cookie() retorna una referencia con privilegios completos.
```

Los cookies de todos los nodos o arcos pueden reiniciarse mediante:

```
559d <Miembros públicos de List_Graph 547b>+≡ (546a) ◁559c 559f▷
 void reset_cookie_nodes()
 {
 for (Node_Iterator itor(*this); itor.has_current(); itor.next())
 itor.get_current_node()->cookie = NULL;
 }
 void reset_cookie_arcs()
 {
 for (Arc_Iterator itor(*this); itor.has_current(); itor.next())
 itor.get_current_arc()->cookie = NULL;
 }
```

Supongamos por ejemplo que necesitamos guardar en cada nodo una lista dinámica de valores reales. Si tenemos un apuntador a un nodo, llamado node, entonces, una forma de hacerlo es como sigue:

```
node->get_cookie() = new DynDlist<float>;
```

Cuando el usuario requiera acceder a la lista, puede hacerlo mediante un casting:

```
static_cast<DynDlist<float>*>(node->get_cookie()) ...
```

Por supuesto, no debe olvidarse de entregar memoria al sistema cuando ésta ya no se requiera. Tal acción se efectúa como sigue:

```
delete static_cast<DynDlist<float>*>(node->get_cookie())
```

Si el usuario requiere un estado estructurado, entonces éste utiliza una estructura, struct o class, según sea el caso.

A veces es necesario guardar alguna información asociada al grafo, no a los nodos o arcos; por ejemplo, algún nodo o arco centinela de valor temporal. Para estos casos disponemos de un cookie en el grafo:

```
559e <Miembros privados de List_Graph 559e>≡ (546a) 561b▷
 void * cookie;

559f <Miembros públicos de List_Graph 547b>+≡ (546a) ◁559d 560a▷
 void *& get_cookie() { return cookie; }
```

### Mapeo entre nodos y arcos

Un uso importante del cookie es implantar el mapeo entre grafos homomorfos o isomorfos. En ciertos algoritmos, por ejemplo, el cálculo del árbol abarcador, se debe calcular un grafo adicional correspondiente al árbol abarcador. Para identificar dentro del árbol abarcador cuál es su nodo en el grafo original, el cookie almacena la imagen del nodo dentro del árbol y viceversa. A este tipo de algoritmos les puede ser muy útil las siguientes primitivas

560a *(Miembros públicos de List\_Graph 547b) +≡* (546a) ◁559f 561a▷  
 template <class N1, class N2> static void map\_nodes(N1 \* p, N2 \* q);  
  
 template <class A1, class A2> static void map\_arcs(A1 \* p, A2 \* q);  
 map\_nodes() mapea entre sí los nodos p y q respectivamente. Si ya existe un mapeo previo, o sea, si p->get\_cookie() contiene una dirección, entonces map\_nodes() hace un mapeo compuesto. Por ejemplo, supongamos que p y q están mapeados entre sí y que realizamos la siguiente llamada:

```
map_nodes(q, t);
```

Entonces, el mapeo resultante se puede interpretar como  $p \longleftrightarrow q \longleftrightarrow t \longleftrightarrow p$ .

El mapeo entre arcos efectuado mediante map\_arcs() tiene semántica similar al de los nodos.

Puesto que los cookies no tienen tipo, cuando se manejan mapeos a través de ellos hay que hacer conversiones engorrosas. Para facilitar tales conversiones se plantean dos primitivas que asumen que sobre el cookie existe un mapeo:

560b *(Funciones de List\_Graph 554e) +≡* ◁554e 562e▷  
 template <class GT> typename GT::Node \* mapped\_node(typename GT::Node \* p)  
 {  
 return (typename GT::Node \*) NODE\_COOKIE(p);  
}  
 template <class GT> typename GT::Arc \* mapped\_arc(typename GT::Arc \* a)  
{  
return (typename GT::Arc \*) ARC\_COOKIE(a);  
}  
 Defines:  
 mapped\_node, used in chunks 605, 607, 612, 626b, 628, 638b, 646c, 663, 683b, 701, 704, 781b, 787, 790, and 791.

Las primitivas de mapeo permiten mantener cadenas de mapeos y son una manera de simplificar muchos algoritmos. La situación más común ataña a las clases de algoritmos que modifican el grafo para efectuar sus cálculos, lo cual acarrea pérdida del grafo original. Para evitar eso hacemos una copia con mapeo y operamos sobre la copia. Los cookies del grafo copia mantienen la imagen del grafo original. Por supuesto, esto también requiere que se mapeen los arcos, lo cual se hace de manera similar a los nodos.

El uso de los cookies, aunque bastante flexible, es muy delicado porque no están sujetos al sistema de tipos del compilador. Por esa razón es altamente recomendable asegurarse de que los cookies se accedan mediante funciones que efectúen la conversión de tipo.

### 7.3.5.4 Reinicio de nodos y arcos

Todos los atributos de control de un nodo o de un arco, es decir, los bits, el contador y el cookie pueden reiniciarse completamente mediante:

561a *<Miembros públicos de List\_Graph 547b>+≡* (546a) ◁560a 561c▷  
 inline void reset\_node(Node \* node);  
 inline void reset\_arc(Arc \* arc);

También pueden reiniciarse todos los atributos de control de todos los nodos o arcos del grafo. En este sentido podemos ilustrar un uso concreto de las clases `Operate_On_Nodes()` y `Operate_On_Arcs()`.

Básicamente, lo que deseamos es recorrer cada nodo e invocar `reset()` para cada atributo de control. Esta actividad se puede especificar mediante la siguiente clase operación:

561b *<Miembros privados de List\_Graph 559e>+≡* (546a) ◁559e 564a▷  
 struct Reset\_Node  
 {  
 void operator () (List\_Graph&, Node \* node) const  
 {  
 node->control\_bits.reset();  
 node->counter = 0;  
 node->cookie = NULL;  
 }  
 };  
 Uses List\_Graph 546a.

Ahora escribimos simplemente la reiniciación de los nodos mediante invocación a `operate_on_nodes()` con `Reset_Node` como operación:

561c *<Miembros públicos de List\_Graph 547b>+≡* (546a) ◁561a 562a▷  
 void reset\_nodes()  
 {  
 Operate\_On\_Nodes <Graph\_Class, Reset\_Node> () (\*this);  
}

Análogamente, la misma técnica se aplica para reiniciar los arcos.

Es típico, antes de hacer un cálculo que involucre los atributos de control, tanto de nodos como de arcos, invocar a `reset_nodes()` y `reset_arcs()` para “limpiarlos” de valores utilizados por cálculos previos.

### 7.3.6 Macros de acceso a nodos y arcos

A efectos de facilitar la legibilidad de código, aunque en detrimento del diagnóstico sintáctico del compilador, la implantación de `List_Graph` y algunos algoritmos usan los siguientes macros:

561d *<Macros para grafos 561d>≡*  
 # define NODE\_BITS(p) ((p)->control\_bits)  
 # define NODE\_COUNTER(p) ((p)->counter)  
 # define IS\_NODE\_VISITED(p, bit) (NODE\_BITS(p).get\_bit(bit))  
 # define NODE\_COOKIE(p) ((p)->cookie)  
 # define ARC\_COUNTER(p) ((p)->counter)  
 # define ARC\_BITS(p) ((p)->control\_bits)

```
define IS_ARC_VISITED(p, bit) (ARC_BITS(p).get_bit(bit))
define ARC_COOKIE(p) ((p)->cookie)
```

### 7.3.7 Construcción y destrucción de List\_Graph

Básicamente, un grafo puede construirse por omisión o por copia desde otro grafo:

562a *(Miembros públicos de List\_Graph 547b)+≡* (546a) ◁ 561c 562b ▷  
     inline List\_Graph();  
     inline List\_Graph(const List\_Graph & g);  
 Uses List\_Graph 546a.

El primer constructor crea un grafo vacío (sin nodos y arcos), el segundo una copia de grafo g.

Es posible asignar a un grafo otro grafo:

562b *(Miembros públicos de List\_Graph 547b)+≡* (546a) ◁ 562a 562c ▷  
     inline List\_Graph& operator = (List\_Graph & g);  
 Uses List\_Graph 546a.

En este caso, el grafo destino se destruye completamente y se copia el fuente g.

El destructor:

562c *(Miembros públicos de List\_Graph 547b)+≡* (546a) ◁ 562b 562d ▷  
     inline virtual ~List\_Graph();  
 Uses List\_Graph 546a.

elimina del grafo todos los nodos y arcos que éste contiene. Toda la memoria es liberada.

Se puede construir un grafo a partir de un digrafo, así como también asignar a un grafo un digrafo. En ese caso, los arcos dirigidos son sustituidos por arcos bidireccionales:

562d *(Miembros públicos de List\_Graph 547b)+≡* (546a) ◁ 562c  
     inline List\_Graph(List\_Digraph<Node, Arc> & g);  
  
     inline List\_Graph & operator = (List\_Digraph<Node, Arc> & g);  
 Uses List\_Digraph 547a and List\_Graph 546a.

En ocasiones no tan excepcionales se requiere, además de la copia del grafo, un mapeo isomorfo entre las copias. En esta situación especial puede usarse la siguiente función:

562e *(Funciones de List\_Graph 554e)+≡* (546a) ◁ 560b 563a ▷  
     template <class GT>  
     void copy\_graph(GT & gtgt, GT & gsrc, const bool cookie\_map = false);

el cual, en caso de que el parámetro cookie\_map sea cierto, hace un mapeo biyectivo entre los dos grafos a través de los cookies de sus nodos y arcos. Al final de la copia, el puntero cookie de cada nodo contendrá la imagen en el grafo copia; igual se aplica para los arcos.

La copia mapeada es una función externa a la clase, es decir, no es un método de List\_Graph. La razón es que es imposible conocer el tipo final de los nodos y arcos de una clase basada en List\_Graph. Al hacerla externa a la clase se está en posición de conocer los tipos en cuestión y de hacer las copias adecuadas. Consiguentemente, no es recomendable que la copia de grafos subyazca en el constructor copia o el operador de asignación.

La copia mapeada es muy útil para llevar a efecto muchos algoritmos. Por ejemplo, para calcular el complemento de un grafo podemos, en primer lugar, copiar el grafo, luego, insertamos sobre la copia los arcos necesarios para volverlo completamente conexo y, finalmente, eliminamos de la copia aquellos arcos que hayan sido mapeados.

No todas las veces es conveniente mapear las copias mediante los cookies. La generalidad del caso la representa toda situación en la cual el grafo a copiar ya contenga datos en los cookies. Esta es la razón por la cual el mapeo entre los cookies es opcional a la copia.

Es posible “borrar” explícitamente un grafo, es decir, eliminar todos sus nodos (y arcos). Para ello se usa el siguiente método:

563a *<Funciones de List\_Graph 554e>+≡* △562e 563b ▷  
*template <class GT> inline void clear\_graph(GT & g);*

Esta primitiva es muy valiosa cuando se usan grafos temporales correspondientes a cálculos intermedios.

### 7.3.8 Operaciones genéricas sobre nodos

En ocasiones se requiere hacer una operación específica sobre todos los nodos; por ejemplo, poner algún valor inicial de atributo. Para tales efectos se proveen las siguientes rutinas, las cuales, por su sencillez, se implantan directamente:

563b *<Funciones de List\_Graph 554e>+≡* △563a 563c ▷  
*template <class GT, class Operation, class SN = Default\_Show\_Node<GT> >*  
*class Operate\_On\_Nodes*  
*{*  
*void operator () (GT & g, Operation op = Operation()) const*  
*{*  
*for (Node\_Iterator<GT,SN> it(g); it.has\_current(); it.next())*  
*op (g, it.get\_current\_node());*  
*}*  
*void operator () (GT & g, void \* ptr, Operation op = Operation()) const*  
*{*  
*for (Node\_Iterator<GT,SN> it(g); it.has\_current(); it.next())*  
*op (g, it.get\_current\_node(), ptr);*  
*}*  
*};*

Como se ve, sendas clases recorren cada nodo del grafo g de tipo genérico GT y efectúan la operación Operation() () sobre cada nodo. La segunda versión permite pasar parámetros a través del puntero ptr.

### 7.3.9 Operaciones genéricas sobre arcos

Al igual que para los nodos, es posible efectuar operaciones genéricas sobre todos los nodos del grafo. Para eso se provee, por ejemplo, la siguiente operación:

563c *<Funciones de List\_Graph 554e>+≡* △563b  
*template <class GT, class Operation,*  
*class SA = Default\_Show\_Arc<GT> >*  
*class Operate\_On\_Arcs*  
*{*  
*void operator () (GT & g, Operation op = Operation()) const*  
*{*  
*for (Arc\_Iterator<GT,SA> it(g); it.has\_current(); it.next())*  
*op (g, it.get\_current\_arc());*  
*}*  
*void operator () (GT & g, typename GT::Node \* p,*

```

 Operation op = Operation() const
 {
 for (Node_Arc_Iterator<GT,SA> it(p); it.has_current(); it.next())
 op (g, it.get_current_arc());
 }
};

```

La semántica es similar a la de los nodos (ver § 7.3.8 (Pág. 563)). La segunda operación sólo se circumscribe a los arcos adyacentes a un nodo.

### 7.3.10 Implantación de List\_Graph

List\_Graph maneja dos listas circulares doblemente enlazadas: una de sus nodos y otra de sus arcos. Cada lista asocia un contador de sus elementos:

564a *(Miembros privados de List\_Graph 559e)*+≡ (546a) ◁ 561b 564b ▷  
Dlink node\_list; // lista de nodos  
size\_t num\_nodes; // cantidad de nodos  
Dlink arc\_list; // lista de arcos  
size\_t num\_arcs; // cantidad de arcos

num\_nodes y num\_arcs contabilizan las cantidades totales de nodos y arcos del grafo.

Estos atributos se actualizan en la primitivas de inserción y eliminación.

node\_list y arc\_list son las cabeceras, de tipo Dlink, de las listas de nodos y arcos, respectivamente, que contiene el grafo. Recordemos que las clases Graph\_Node y Graph\_Arc derivan de Dlink. La base Dlink, pues, conforma en cada clase el enlace doble de la listas node\_list y arc\_list.

Como node\_list y arc\_list son, por derivación, de tipo Dlink, planteamos las siguientes funciones de conversión de Dlink a Graph\_Node y a Graph\_Arc:

564b *(Miembros privados de List\_Graph 559e)*+≡ (546a) ◁ 564a 564c ▷  
static Node \* dlink\_to\_node(Dlink \* p) { return (Node\*) p; }  
static Arc \* dlink\_to\_arc(Dlink \* p) { return (Arc\*) p; }

#### 7.3.10.1 Acceso a nodos y arcos

Con estas conversiones ya podemos codificar “legiblemente” los métodos básicos de acceso a nodos y arcos:

564c *(Miembros privados de List\_Graph 559e)*+≡ (546a) ◁ 564b 568a ▷

564d *(Implantación de List\_Graph 564d)*+≡ (565a) ▷

```

template <typename Node, typename Arc>
Node * List_Graph<Node, Arc>::get_first_node()
{
 return dlink_to_node(node_list.get_next());
}

template <typename Node, typename Arc>
Arc * List_Graph<Node, Arc>::get_first_arc()
{
 return dlink_to_arc(arc_list.get_next());
}

```

Uses List\_Graph 546a.

Tanto la búsqueda de nodos como la de arcos requieren iterar sobre la correspondiente lista (node\_list o arc\_list, según sea el caso). Por esa razón nos conviene implementar estos iteradores a efectos de que las búsquedas se efectúen mediante ellos.

### Iterador de nodos

La clase Node\_Iterator se implanta directamente por derivación pública de Dlink::Iterator.

### Iterador de arcos

La misma consideración que para el iterador sobre los nodos aplica sobre el iterador sobre los arcos, pues éstos también se fundamentan en Dlink.

### Búsqueda de nodos

Los diferentes estilos de búsqueda de nodos y arcos se instrumentan a través de algunos de los dos iteradores presentados en las secciones previas. La implementación de una de las primitivas ilustra el esquema general:

565a *(Implantación de List\_Graph 564d) +≡* △564d 565b ▷  
 template <typename Node, typename Arc>  
 template <typename T, class Equal>  
 Node \* List\_Graph<Node, Arc>::search\_node(const T & data)  
 {  
 for (Node\_Iterator it(\*this); it.has\_current(); it.next())  
 {  
 Node \* p = it.get\_current\_node();  
 if (Equal () (p->get\_info(), data))  
 return p;  
 }  
 return NULL;  
 }

Uses List\_Graph 546a.

A la excepción de la especialización por omisión, cada rutina se fundamenta en el iterador sobre nodos del grafo Node\_Iterator.

### Búsqueda de arcos

La búsqueda de un arco dentro del grafo es reminiscente a la de un nodo: recorrer mediante el iterador de arcos hasta satisfacer la búsqueda o haberlos recorrido enteramente, en cuyo caso, la búsqueda es fallida. La implementación es como sigue:

565b *(Implantación de List\_Graph 564d) +≡* △565a 566b ▷  
 template <typename Node, typename Arc>  
 template <typename T, class Equal>  
 Arc \* List\_Graph<Node, Arc>::search\_Arc(const T & data)  
 {  
 for (Arc\_Iterator it(\*this); it.has\_current(); it.next())  
 {  
 Arc \* a = it.get\_current\_arc();  
 if (Equal () (a->get\_info(), data))  
 return a;  
 }  
}

```

 return NULL;
 }

```

Uses List\_Graph 546a.

Más adelante, cuando definamos el iterador de arcos sobre un nodo implementaremos la búsqueda de un arco que conecte a dos nodos.

### 7.3.10.2 Arcos

Un arco conecta dos nodos llamados origen y destino, declarados de la siguiente manera:

566a *(Atributos de Graph\_Arc 557b) +≡* (551c) <559b 569>

```

void * src_node; // nodo origen
void * tgt_node; // nodo destino

```

Estos atributos son de tipo void\* porque desde un Graph\_Arc no se puede conocer el tipo exacto de Graph\_Node; éste se conocerá cuando se instancie el grafo. La implantación del acceso a estos atributos se hace desde la clase List\_Graph de la siguiente manera:

566b *(Implantación de List\_Graph 564d) +≡* <565b 566d>

```

template <typename Node, typename Arc>
Node * List_Graph<Node, Arc>::get_src_node(Arc * arc)
{
 return (Node*) arc->src_node;
}
template <typename Node, typename Arc>
Node * List_Graph<Node, Arc>::get_tgt_node(Arc * arc)
{
 return (Node*) arc->tgt_node;
}

```

Uses List\_Graph 546a.

Para la implantación de get\_connected\_node() diseñamos una rutina homónima, parte de Graph\_Arc, que nos sirva de intermediaria:

566c *(Métodos de Graph\_Arc 552a) +≡* (551c) <552a

```

void * get_connected_node(void * node)
{
 return src_node == node ? tgt_node : src_node;
}

```

Y, mediante ella, llevar a cabo el método de List\_Graph:

566d *(Implantación de List\_Graph 564d) +≡* <566b 571b>

```

template <typename Node, typename Arc>
Node * List_Graph<Node, Arc>::get_connected_node(Arc * arc, Node * node)
{
 return (Node*) arc->get_connected_node(node);
}

```

Uses List\_Graph 546a.

### Listas de adyacencia

Como ya ha sido decidido y es sabido, la representación en memoria de List\_Graph se basa en listas de adyacencia. Aparte de un mayor consumo de espacio para grafos muy densos, un problema de este enfoque es la redundancia de nodos en las listas de adyacencia.

Esto es, por cada arco  $u \rightarrow v$  se requiere un nodo en la lista de adyacencia de  $u$ , y de otro en la lista de  $v$ . Esta redundancia exige acceder las dos listas de adyacencia cada vez que de alguna forma se requiera modificar un arco.

Para evadir el problema de coherencia anterior, y por mayor eficiencia en rapidez, diseñaremos una estructura de datos especializada cuya definición comienza por el tipo Arc\_Node:

567a *<Definición de Arco-Nodo 567a>*≡

```
struct Arc_Node : public Dlink
{
 void * arc;
 Arc_Node() : arc(NULL) {}
 Arc_Node(void * __arc) : arc(__arc) {}
};
```

Defines:

Arc\_Node, used in chunks 568–70 and 572–74.

Arc\_Node deriva de Dlink, por lo que éste es parte de una lista doblemente enlazada. El campo arc es un puntero a un Graph\_Arc (o familiar de él por derivación). Pictóricamente, un Arc\_Node se representa en memoria de la manera ilustrada en la figura 7.15.

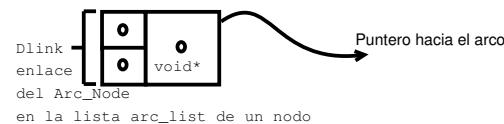


Figura 7.15: Representación en memoria de un objeto Arc\_Node

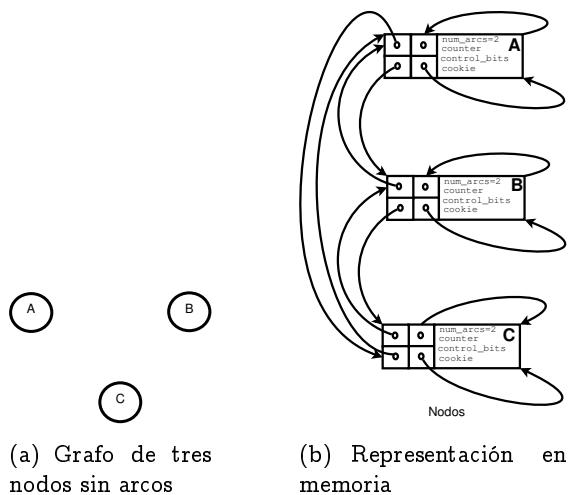


Figura 7.16: Un grafo simple y el estado de su estructura de datos

Los objetos del tipo Arc\_Node son los componentes de la lista de adyacencia de cada nodo. De este modo, la lista de adyacencia de un nodo se define como sigue:

567b *<Miembros de Graph\_Node 548a>*+≡

```
Dlink arc_list;
```

(547c) ◀559a

Así, un nodo se representa en memoria del modo ilustrado en la figura 7.17.

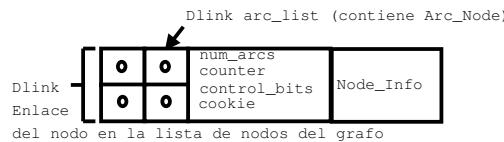


Figura 7.17: Representación en memoria de un nodo

Para un grafo de tres nodos, sin arcos, el estado de la estructura de datos es el correspondiente al de la figura 7.16, es decir, tres nodos enlazados a través de `node_list` en la clase `List_Graph`. Puesto que no hay arcos, las listas de adyacencia de cada nodo están vacías.

En síntesis, manejamos tres tipos de listas circulares doblemente enlazadas: (1) la lista de todos los nodos del grafo, (2) la lista de todos los arcos del grafo, y (3) la lista por nodo de sus arcos. Tales listas son accedidas mediante la clase `Dlink`. Por esa razón plantearemos funciones de conversión desde enlaces de tipo `Dlink` hacia el nodo.

Comencemos con la lista de adyacencia de un nodo, la cual es de tipo `Dlink`, pero sus nodos<sup>9</sup> de tipo `Arc_Node`:

568a *(Miembros privados de List\_Graph 559e) +≡* (546a) ◁ 564c 568b ▷  
 static Arc\_Node \* dlink\_to\_arc\_node(Dlink \* p) { return (Arc\_Node\*) p; }  
 Uses Arc\_Node 567a.

Si tenemos, por ejemplo, un puntero a un objeto `Node`, podemos acceder al primero de sus arcos en su lista de `Arc_Node`:

```
Arc_Node * p = dlink_to_arc_node(node->arc_list.get_next());
```

El cual debe transformarse a un objeto de tipo `Arc` mediante:

```
Arc * arc = static_cast<Arc*>(p->arc);
```

pues `p->arc` es de tipo `void*`. Para endulzar un poco esta conversión escribimos la siguiente primitiva:

568b *(Miembros privados de List\_Graph 559e) +≡* (546a) ◁ 568a 570a ▷  
 static Arc \* void\_to\_arc(Arc\_Node \* arc\_node)  
 {  
 return (Arc\*) arc\_node->arc;  
 }  
 Uses Arc\_Node 567a.

Recordemos que un nodo tiene una lista `arc_list` de elementos de tipo `Arc_Node` enlazados mediante punteros dobles `Dlink`. A su vez, un `Arc_Node` tiene un puntero a un arco de tipo `Graph_Arc`. Esta configuración nos permitirá una inserción  $\mathcal{O}(1)$  a la vez que no se redundará el arco. Pero, así las cosas, la eliminación de un arco todavía requiere recorrer la lista de arcos del nodo destino para encontrar su `Arc_Node` y eliminarlo. Por esta razón ubicaremos en `Graph_Arc` dos punteros al `Arc_Node`, uno al del nodo origen y otro al del nodo destino. El `Graph_Arc` tendrá entonces la representación en memoria mostrada en la figura 7.18.

<sup>9</sup>Téngase cuidado con la ambigüedad. En esta frase nos referimos a los nodos de una lista enlazada y no a los de un grafo.

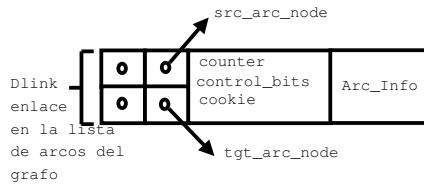


Figura 7.18: Representación en memoria de un arco

Lo que conduce a las siguientes declaraciones para un arco:

569 *(Atributos de Graph\_Arc 557b) +≡ (551c) ▷ 566a*  
`Arc_Node * src_arc_node; // puntero al Arc_Node del nodo fuente  
Arc_Node * tgt_arc_node; // puntero al Arc_Node del nodo destino`

Uses Arc\_Node 567a.

y cuya configuración nos permite resolver dos problemas:

1. Al momento de eliminar un arco en un grafo, requerimos eliminar el Arc\_Node de las listas del nodo origen y destino respectivamente. Esto se logra en  $\mathcal{O}(1)$  mediante el acceso a los campos src\_arc\_node y tgt\_arc\_node.
2. Cuando se trate de un digrafo, el nodo destino del arco no contiene un Arc\_Node que apunte al arco. Del mismo modo, el campo tgt\_arc\_node se mantiene en NULL.

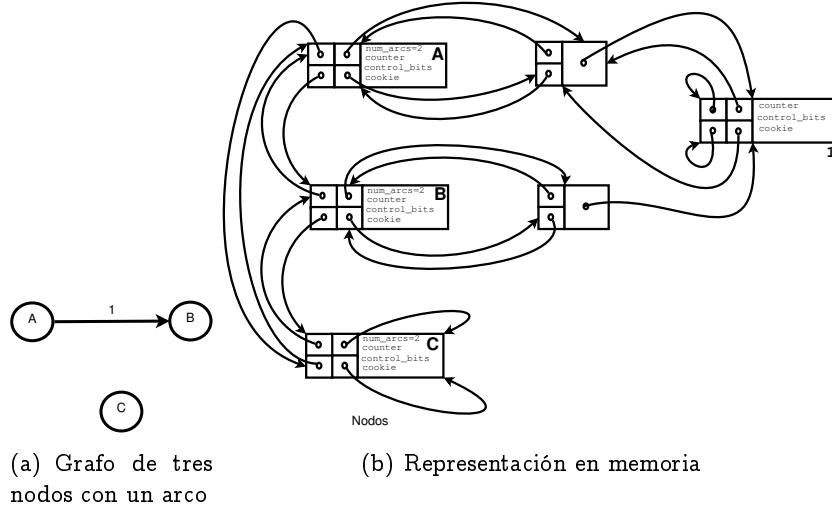


Figura 7.19: Un grafo simple y el estado de su estructura de datos

Si al grafo de la figura 7.19-a le insertamos un arco desde el nodo A hacia el B, entonces el estado de la estructura de datos depara en el de la figura 7.19-b. Como se ve, aunque hay un solo arco, existen dos objetos de tipo Arc\_Node relacionados con él.

En este estado es simple apreciar el sentido de la estructura de datos, sobre todo al momento de eliminar un arco. Supongamos que deseamos eliminar el único arco del grafo 7.19-a. A través de su dirección tenemos acceso directo a los objetos Arc\_Node de las listas de adyacencia de A y B. Puesto que cada objeto Arc\_Node está doblemente enlazado, la eliminación de la lista de adyacencia de cada nodo es directa.

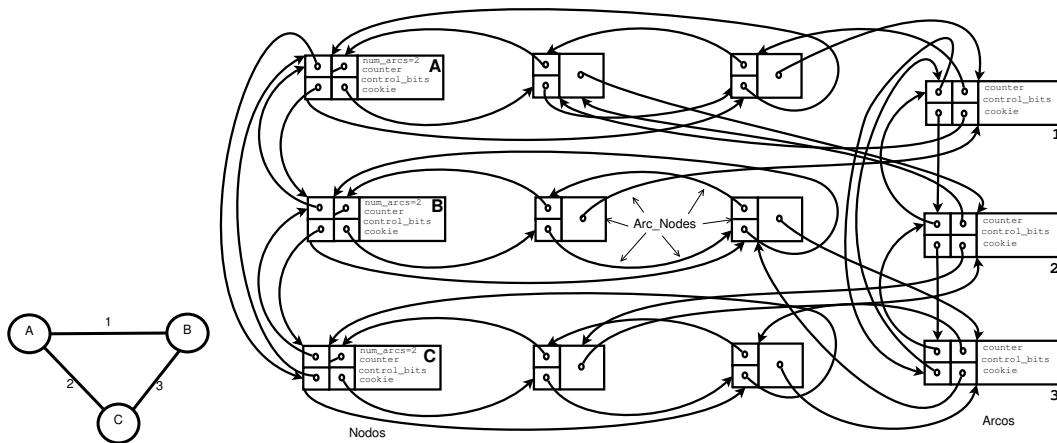


Figura 7.20: Un grafo simple y el estado de su estructura de datos

La lista de arcos de todo el grafo, manejada a través del atributo `arc_list` de `List_Graph`, se muestra mejor en la figura 7.20. Aquí también se evidencia la complejidad de la estructura de datos.

List\_Digraph hereda toda la interfaz de List\_Graph y casi toda su implantación. En algunas operaciones requeriremos determinar si se trata de un grafo o un digrafo. Para eso utilizaremos una bandera lógica:

570a

*<Miembros privados de List\_Graph 559e>* +≡ (546a) <568b  
 bool digraph;

El valor de `digraph` es `true` si se trata de un digrafo; `false` de lo contrario.

### Iterador de arcos sobre un nodo

Como ya se dijo, un iterador de arcos a partir de un nodo se implementa a partir de Dlink::Iterator, pues Arc\_Node es por derivación de tipo Dlink.

A parte de guardar el grafo y el nodo desde el cual se itera, `Node_Arc_Iterator` mantiene la función que determina cuál arco mostrar:

570b

*(Miembros privados de iterador de Node 570b)≡* (550b)  
**Node \***      **src\_node;**

Puesto que `Node_Arc_Iterator` es por derivación de tipo `Dlink::Iterator`, éste tiene todos los métodos asociados (`next()`, `prev()`, `has_current()`, etcétera). El método `Dlink::Iterator::get_current()` no puede usarse directamente porque éste retorna un `Dlink` y necesitamos conocer en principio el `Arc_Node`. Por tanto, diseñamos esta conversión:

570c

```

<Miembros públicos de iterador de Node 550c>+≡ (550b) <551b
 Arc_Node * get_current_arc_node()
{
 return dlink_to_arc_node(Dlink::Iterator::get_current());
}

```

Uses Arc Node 567a.

### Búsqueda de un arco dados dos nodos

Con el iterador de arcos sobre un nodo podemos implantar fácilmente la búsqueda de un arco que conecte dos nodos. La idea es tomar uno de los nodos, preferiblemente el que contenga menos arcos, y desde allí recorrer sus arcos hasta encontrar el otro nodo o hasta que no hayan más arcos que recorrer, en cuyo caso, la búsqueda es fallida.

571a *(Implantación de funciones de List\_Graph 571a)≡* 575▷

```

template <class GT, class SA>
typename GT::Arc * search_arc(GT & g, typename GT::Node * src_node,
 typename GT::Node * tgt_node)
{
 Node_Arc_Iterator<GT, SA> itor;
 typename GT::Node * searched_node;

 // escoger nodo con menos arcos
 if (g.is_digraph() or src_node->num_arcs < tgt_node->num_arcs)
 {
 searched_node = tgt_node;
 itor = Node_Arc_Iterator<GT, SA>(src_node);
 }
 else
 {
 searched_node = src_node;
 itor = Node_Arc_Iterator<GT, SA>(tgt_node);
 }
 for /* nada */; itor.has_current(); itor.next())
 if (itor.get_tgt_node() == searched_node)
 return itor.get_current_arc();

 return NULL;
}

```

#### 7.3.10.3 Inserción de nodos

El primer tipo de inserción asume que la memoria para el nodo ya ha sido apartada y se instrumenta de la siguiente manera:

571b *(Implantación de List\_Graph 564d)++≡* ◁566d 571c▷

```

template <typename Node, typename Arc>
Node * List_Graph<Node, Arc>::insert_node(Node * node)
{
 ++num_nodes;
 node_list.append(node);
 return node;
}

```

Uses List\_Graph 546a.

El segundo tipo extiende el trabajo anterior para apartar la memoria para el nodo y colocarle la información dada de la forma siguiente:

571c *(Implantación de List\_Graph 564d)++≡* ◁571b 572▷

```

template <typename Node, typename Arc> Node *
List_Graph<Node, Arc>::insert_node(const typename Node::Node_Type & node_info)
{

```

```

 return List_Graph<Node, Arc>::insert_node(new Node (node_info));
}

Uses List_Graph 546a.
```

### 7.3.10.4 Inserción de arcos

Para insertar un arco deben conocerse los nodos que éste relaciona, los cuales deben haberse insertado previamente en el grafo. La inserción de un arco se resume en los siguientes pasos:

1. Apartar memoria para el arco (de tipo Arc) y asignarle sus datos.
2. Si la inserción ocurre en un grafo (no un digrafo), apartar entonces memoria para un objeto de tipo Arc\_Node correspondiente al nodo destino tgt\_node, ponerlo a apuntar hacia el arco creado en (1) e insertarlo en la lista de adyacencia de tgt\_node.  
En este punto debemos estar pendientes de que el arco a insertar no represente un lazo, y así, de ese modo, no duplicar el Arc\_Node.
3. Apartar memoria para un objeto de tipo Arc\_Node correspondiente al nodo origen src\_node, ponerlo a apuntar hacia el arco creado en (1) e insertarlo en la lista de adyacencia de src\_node
4. Insertar el arco en la lista de arcos del grafo.

El algoritmo anterior se codifica como sigue:

572   *(Implantación de List\_Graph 564d)* +≡                                                          △571c 573 ▷

```

template <typename Node, typename Arc> Arc *
List_Graph<Node, Arc>::insert_arc(Node * src_node, Node * tgt_node)
{
 // paso 1: apartar memoria para Arc e iniciar
 unique_ptr<Arc> arc (new Arc); // apartar memoria para arco
 arc->src_node = src_node; // Iniciar sus campos
 arc->tgt_node = tgt_node;

 // paso 3: (parcial): apartar Arc_Node de src_node
 unique_ptr<Arc_Node> src_arc_node (new Arc_Node (arc.get()));

 // paso 2: si es grafo ==> apartar Arc_Node de tgt_node
 if (not digraph) // si es digrafo ==> no insertar en otro nodo
 {
 // inserción en nodo destino
 if (src_node == tgt_node) // ¿es un lazo?
 arc->tgt_arc_node = src_arc_node.get();
 else
 { // apartar arco nodo para tgt_node
 unique_ptr<Arc_Node> tgt_arc_node(new Arc_Node(arc.get()));
 // inserción en lista de adyacencia de tgt_node
 arc->tgt_arc_node = tgt_arc_node.get();
 tgt_node->arc_list.append(tgt_arc_node.get());
 tgt_node->num_arcs++;
 tgt_arc_node.release();
 }
 }
}
```

```

 // paso 3 (resto): inserción en lista adyacencia src_node
 arc->src_arc_node = src_arc_node.get();
 src_node->arc_list.append(src_arc_node.get());
 src_node->num_arcs++;
 arc_list.append(arc.get()); //paso 4: insertar en lista arcos grafo
 ++num_arcs;
 src_arc_node.release();

 return arc.release();
}

```

Uses Arc\_Node 567a and List\_Graph 546a.

La codificación es ligeramente diferente al algoritmo en castellano porque en ella se considera la posibilidad de que ocurra una falla de memoria. Para mantener el código lo más parecido al algoritmo en castellano usamos autopunteros<sup>10</sup>. Por eso la estrategia es apartar los tres bloques de memoria que requerimos (el del Arc y los dos Arc\_Node), y si no ha ocurrido excepción, entonces podemos insertar en todas las listas sin riesgo de falla.

#### 7.3.10.5 Eliminación de arcos

La eliminación de un arco es más sencilla porque no hay puntos eventuales de excepción<sup>11</sup>. El procedimiento se resume en los siguientes pasos:

1. Eliminar de la lista de adyacencia del nodo origen el Arc\_Node y liberar su memoria.  
El acceso a este Arc\_Node se da por el atributo arc->src\_arc\_node.
2. Si se trata de un grafo (no un digrafo), eliminar entonces de la lista de adyacencia del nodo destino el Arc\_Node y liberar su memoria.  
El acceso a este Arc\_Node se da por el atributo arc->tgt\_arc\_node.  
En este paso hay que prestar atención a que el arco sea un ciclo, en cuyo caso no hay que ni eliminar de la lista de adyacencia ni liberar la memoria; ambas acciones ya fueron hechas en el paso anterior.
3. Finalmente, eliminar el arco de la lista de arcos del grafo y liberar su memoria.

Los pasos anteriores se reflejan en el siguiente código:

573 <*Implantación de List\_Graph 564d*>+≡ ▷572 574▷

```

template <typename Node, typename Arc>
void List_Graph<Node, Arc>::remove_arc(Arc * arc)
{
 // paso 1: eliminar Arc_node de src_node
 Node * src_node = get_src_node(arc);
 Arc_Node * src_arc_node = arc->src_arc_node;

 src_arc_node->del(); // desenlaza src_node de la lista de nodos
 src_node->num_arcs--; // decrementa contador de arcos de src_node

```

<sup>10</sup>Un autopuntero es una clase de objeto apuntador que maneja la liberación automática de memoria cuando se invoca al destructor. De este modo se ahorra la inclusión de un manejador de excepciones que prevea una falla de memoria y que requiera "limpiar" estado intermedio. Los autopunteros se liberan cuando se ejecuta el método release(); en este caso, el destructor no efectuará el delete.

<sup>11</sup>Salvo que se trate de un error del usuario, por ejemplo, eliminación doble o un bug en la implantación.

```

 delete src_arc_node; // entrega memoria
 if (not digraph)
 {
 // eliminación arco en nodo destino
 Node * tgt_node = get_tgt_node(arc);
 if (src_node != tgt_node) // verificar eliminación de ciclo
 {
 // paso 2: eliminar Arc_node de tgt_node
 Arc_Node * tgt_arc_node = arc->tgt_arc_node;
 tgt_arc_node->del();
 tgt_node->num_arcs--;
 delete tgt_arc_node;
 }
 }
 // eliminación de arco del grafo
 arc->del(); // desenlazar arc de lista de arcos de grafo
 -num_arcs;
 delete arc;
}
Uses Arc_Node 567a and List_Graph 546a.

```

### 7.3.10.6 Eliminación de nodos

La eliminación de un nodo puede parecer complicada ante el requerimiento de que deben eliminarse todos sus arcos incidentes y adyacentes. A pesar de todo, si no se trata de un digrafo, resulta sencilla si nos basamos en la eliminación del arco implantada en la subsección anterior:

574      *(Implantación de List\_Graph 564d)* +≡                          ◁ 573 576a ▷

```

template <typename Node, typename Arc>
void List_Graph<Node, Arc>::remove_node(Node * node)
{
 if (not digraph)
 // Eliminar arcos adyacentes a node
 while (not node->arc_list.is_empty()) // mientras queden arcos
 {
 // obtener Arc_Node
 Arc_Node * arc_node =
 dlink_to_arc_node(node->arc_list.get_next());
 Arc * arc = void_to_arc(arc_node); // obtener el arco
 remove_arc(arc); // eliminarlo del grafo
 }
 else
 for (Arc_Iterator it(*this); it.has_current();)
 {
 Arc * arc = it.get_current();
 if (get_src_node(arc) == node or get_tgt_node(arc) == node)
 {
 it.next();
 remove_arc(arc);
 }
 else
 it.next();
 }
 // en este punto el nodo ya no tiene arcos
}

```

```
node->del(); // desenlazar nodo de lista de nodos del grafo
-num_nodes;
delete node;
}
```

Uses Arc Node 567a and List Graph 546a.

Si se trata de un digrafo, entonces la cuestión es más difícil e ineficiente, pues con la estructura de datos diseñada no queda otra alternativa que mirar todos los arcos y eliminar aquellos que apunten a `node`. Esto puede ser extremadamente ineficiente y hay que estar consciente de este coste a la hora de alterar la topología de un digrafo.

### 7.3.10.7 Limpieza de grafos

`remove_node()` nos hace la mayor parte del trabajo necesario para limpiar un grafo (operación `clear_graph()`), pues ésta se remite a eliminar todos los nodos del grafo:

```
575 <Implantación de funciones de List_Graph 571a>+≡ ◁571a 577a▷
 template <class GT> void clear_graph(GT & g)
 {
 for (Arc_Iterator<GT> it(g); it.has_current();) // eliminar arcos
 {
 typename GT::Arc * arc = it.get_current();
 it.next();
 g.remove_arc(arc);
 }
 for (Node_Iterator<GT> it(g); it.has_current();) // eliminar nodos
 {
 typename GT::Node * p = it.get_current();
 it.next(); // avanzar antes de borrar (consistencia iterador)
 g.remove_node(p); // eliminarlo del grafo
 }
 }
```

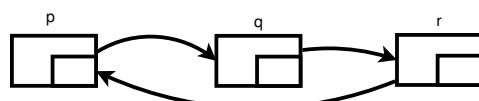
Puesto que `remove_arc()` y `remove_node()` son virtuales, puede haber toda una cadena de derivación. Por eso es más seguro eliminar los arcos y luego los nodos, en lugar de eliminar directamente los nodos (que también eliminan intrínsecamente los arcos).

### 7.3.10.8 Mapeo de nodos y arcos

Recordemos que estipulamos el mapeo de nodos y de arcos a través de los cookies. Concentrémonos en `map_node()`, el cual mapea los cookies de dos nodos y preserva la composición de mapeos en el sentido de que si los cookies son distintos de `NULL`, entonces el mapeo se propaga como si los cookies fuesen una lista enlazada circular. Inicialmente, si no hay mapeo, entonces `node_map(p, q)` construye la siguiente configuración de enlaces:



Luego, si ocurre otra llamada `node map(q,r)`, entonces los enlaces devienen en:



Claramente, los mapeos deben comportarse como si tratase de una inserción en una lista circular, simplemente enlazada, sin nodo cabecera, lo que nos conduce a:

576a

```
Implantación de List_Graph 564d +≡
template <typename Node, typename Arc>
template <class N1, class N2>
void List_Graph<Node, Arc>::map_nodes(N1 * p, N2 * q)
{
 if (NODE_COOKIE(p) == NULL)
 {
 NODE_COOKIE(p) = q;
 NODE_COOKIE(q) = p;
 return;
 }
 NODE_COOKIE(q) = NODE_COOKIE(p);
 NODE_COOKIE(p) = q;
}
template <typename Node, typename Arc>
template <class A1, class A2>
void List_Graph<Node, Arc>::map_arcs(A1 * p, A2 * q)
{
 if (ARC_COOKIE(p) == NULL)
 {
 ARC_COOKIE(p) = q;
 ARC_COOKIE(q) = p;
 return;
 }
 ARC_COOKIE(q) = ARC_COOKIE(p);
 ARC_COOKIE(p) = q;
}
```

&lt;574 577b&gt;

Uses List\_Graph 546a.

### 7.3.10.9 Copia de grafos

De manera general, la copia puede separarse en dos fases: (1) copia de nodos, y (2) copia de arcos. Copiar nodos es relativamente fácil: por cada nodo del grafo origen se crea una copia y se inserta en el grafo destino.

La copia de arcos es similar: por cada arco del grafo origen se crea una copia y se inserta en el grafo destino. Recordemos que para insertar un arco hay que especificar sus nodos. Cuando inspeccionamos un arco del grafo origen, los nodos que examinamos refieren al grafo origen, pero el arco a insertar en el grafo destino requiere nodos respecto al grafo destino y no respecto al origen. Se nos presenta entonces el siguiente problema: dado un nodo del grafo origen, ¿cómo conocer su equivalente en el grafo copia destino? Una manera de hacerlo sería mediante un mapeo de nodos entre los dos grafos a través de sus cookies, pero la semántica de la interfaz `copy_graph()` puede impedir este tipo de mapeo (cuando el parámetro `cookie_map == false`). Por esa razón haremos el mapeo mediante una tabla instrumentada con un árbol AVL:

576b

*Declaración de tabla mapeo 576b* ≡

(577a)

```
DynMapAvlTree<typename GT::Node*, typename GT::Node*> map;
```

map mapea nodos del grafo origen `src_graph` a nodos del grafo destino `this`.

El método de copia se especifica como sigue:

577a *(Implantación de funciones de List\_Graph 571a) +≡* ◁575  
 template <class GT> void copy\_graph(GT & gtgt, GT & gsrc, const bool cookie\_map)  
 {  
     clear\_graph(gtgt); // limpiar this antes de copiar  
*(Declaración de tabla mapeo 576b)*  
  
     // fase 1: recorrer nodos de src\_graph e insertar copia en this  
     for (typename GT::Node\_Iterator it(gsrc);  
         it.has\_current(); it.next())  
     {  
         typename GT::Node \* src\_node = it.get\_current\_node();  
         unique\_ptr<typename GT::Node> tgt\_node(new typename GT::Node(src\_node));  
         map.insert(src\_node, tgt\_node.get());  
  
         typename GT::Node \* tgt = tgt\_node.release();  
         gtgt.insert\_node(tgt); // insertar en grafo destino  
  
         if (cookie\_map)  
             GT::map\_nodes(src\_node, tgt);  
     }  
     // fase 2: por cada arco de src\_graph, crear en this un  
     // arco que conecte a los nodos mapeados de map  
     for (typename GT::Arc\_Iterator it(gsrc);  
         it.has\_current(); it.next())  
     {  
         typename GT::Arc \* src\_arc = it.get\_current\_arc();  
  
         // obtener imágenes de nodos en el grafo destino y crear arco  
         typename GT::Node \* src\_node = map[gsrc.get\_src\_node(src\_arc)];  
         typename GT::Node \* tgt\_node = map[gsrc.get\_tgt\_node(src\_arc)];  
         typename GT::Arc \* tgt\_arc = gtgt.insert\_arc(src\_node, tgt\_node);  
         \*tgt\_arc = \*src\_arc;  
         if (cookie\_map)  
             GT::map\_arcs(src\_arc, tgt\_arc);  
     }  
 }

Aparte de su utilización pública en el contexto de una aplicación, `copy_graph()` es usado por el constructor copia y el operador de asignación.

#### 7.3.10.10 Ordenamiento de arcos

Los arcos están contenidos en la lista doblemente enlazada `arc_list`. Para ordenarlos nos valdremos del mergesort implementado en § 3.2.1.5 (Pág. 172). Apelamos al mergesort por varias razones. En primer lugar, a diferencia, por instancia, del quicksort, el desempeño  $\mathcal{O}(n \lg(n))$  del mergesort está garantizado y no supeditado a la permutación. En segundo lugar, el consumo de espacio del mergesort es a lo sumo  $\mathcal{O}(\lg(n))$ , a diferencia del  $\mathcal{O}(n)$ , que es el peor caso de consumo para el quicksort con listas enlazadas.

Así pues, el ordenamiento de los arcos se remite a:

577b *(Implantación de List\_Graph 564d) +≡* ◁576a  
 template <typename Node, typename Arc>  
 template <class Compare>

```
void List_Graph<Node, Arc>::sort_arcs()
{
 mergesort<Cmp_Arc<Compare> >(arc_list);
}
```

Uses List\_Graph 546a and mergesort 173a.

`sort_arcs()` recibe la clase de comparación `Compare` sobre punteros a arcos de tipo `Arc`; esta clase, que la debe proveer el usuario interesado en ordenar los arcos, hace la comparación `bool operator () (Arc *, Arc *)`. Pero el `mergesort` sobre listas enlazadas invoca a `bool operator () (Dlink *, Dlink *)` (§ 3.2.1.5 (Pág. 172)), para comparar. Debemos, pues, escribir la clase de comparación para `mergesort()`, la cual se denomina `Cmp_Arc`. Su función es convertir los punteros de tipo `Dlink` a `Arc*` e invocar a la comparación `Compare`.

## 7.4 TAD camino sobre un grafo (Path<GT>)

La búsqueda de caminos es una de las actividades más comunes en la manipulación sobre un grafo. Por eso vale la pena un TAD que modelice caminos y facilite su construcción.

El TAD Path<GT> modeliza un camino sobre un List\_Graph, su estructura general se define como sigue:

Defines:  
Path, used in chunks 579c, 597–602, 650, 682, 683, 687, 692, 704–6, 726, 727a, 770, 771, 798, and 801.  
El parámetro tipo GT debe ser de tipo List\_Graph, List\_Digraph o de alguna clase descendiente

Path<GT> guarda la instancia del grafo en un atributo:

el cual es consultable mediante:

La clase Path<GT> lee los tipos concernientes a los nodos y arcos del grafo de la siguiente manera:

578d     $\langle$ Tipos de path 578d $\rangle \equiv$  (578a)  
        **typedef typename GT::Node\_Type Node\_Type;**  
        **typedef typename GT::Arc\_Type Arc\_Type;**

Lo mismo se aplica para los punteros a nodos y arcos, con la excepción de que éstos se manejan de manera privada:

Nuestra manera de representar un camino es mediante los arcos que lo componen. Esta forma es independiente de idiosincrasias tales como que se trate un multigrafo, un digrafo, etcétera. Puesto que en List\_Graph y en su derivado List\_Digraph, un arco no expresa la dirección, es necesario indicar cuál es el nodo origen. De este modo, el otro nodo del arco sería el sucesor en el camino.

La observación anterior sugiere la siguiente representación de estructura de datos de cada punto del camino:

579a *(Miembros privados de path 578b) +≡* (578a) ◁ 578e 579b ▷  
 struct Path\_Desc  
 {  
 Node \* node; // nodo origen  
 Arc \* arc; // arco adyacente  
 Path\_Desc(Node \* \_node = NULL, Arc \* \_arc = NULL) : node(\_node), arc(\_arc) {}  
};  
*Defines:*  
 Path\_Desc, used in chunks 579–81.

De este modo, un camino se representa como una lista de Path\_Desc:

579b *(Miembros privados de path 578b) +≡* (578a) ◁ 579a ▷  
 DynDList<Path\_Desc> list;  
*Uses DynDList 85a and Path\_Desc 579a.*

La forma “tradicional”<sup>12</sup> de declarar un camino es uno vacío que refiere a un determinado grafo:

579c *(Miembros públicos de path 578c) +≡* (578a) ◁ 578c 579d ▷  
 Path(GT & \_g, void \* \_\_cookie = NULL) : g(&\_g), cookie(\_\_cookie) {}  
*Uses Path 578a.*

El constructor inicia un camino vacío sobre el grafo g.

Para comenzar a operar sobre un camino vacío hay que indicar cuál es su nodo inicial. Esto se hace mediante la siguiente primitiva:

579d *(Miembros públicos de path 578c) +≡* (578a) ◁ 579c 579e ▷  
 void init(Node \* start\_node) { list.append(Path\_Desc(start\_node)); }  
*Uses Path\_Desc 579a.*

donde start\_node es nodo inicio del camino.

Se puede hacer lo mismo en tiempo de construcción.

La longitud de un camino es el número de arcos que lo componen. En nuestra estructura de datos, esta información está dada por el número de elementos del atributo list:

579e *(Miembros públicos de path 578c) +≡* (578a) ◁ 579d 579f ▷  
 const size\_t & size() const { return list.size(); }

La “limpieza” de un camino consiste en borrar todos sus nodos y arcos, lo cual se efectúa con la operación siguiente:

579f *(Miembros públicos de path 578c) +≡* (578a) ◁ 579e 580a ▷  
 void clear\_path()  
 {  
 while (not list.is\_empty())  
 list.remove\_first();  
}

<sup>12</sup>En realidad, la tradición “aún” no está instituida.

La cual vacía la lista de Path\_Desc.

Luego de tener un camino iniciado, es decir, con su primer nodo, la construcción del camino se remite a añadirle los arcos. Para eso se dispone de la siguiente operación:

580a *(Miembros públicos de path 578c) +≡* (578a) ◁ 579f 580b ▷  
 void append(Arc \* arc)  
 {  
 Path\_Desc & last\_path\_desc = list.get\_last();  
 last\_path\_desc.arc = arc;  
 list.append(Path\_Desc(g->get\_connected\_node(arc, last\_path\_desc.node)));  
}

Uses Path\_Desc 579a.

Otra forma de añadir un componente al camino es especificar el siguiente nodo en lugar del arco. Para eso se provee el append() por nodo:

580b *(Miembros públicos de path 578c) +≡* (578a) ◁ 580a 580c ▷  
 void append(Node \* node)  
{  
 if (list.is\_empty())  
 {  
 init(node);  
 return;  
 }  
 Node \* last\_node = get\_last\_node();  
 Arc \* arc = search\_arc(\*g, last\_node, node);  
 append(arc);  
}

La rutina es delicada porque puede acarrear ambigüedades sobre multigrafos y multidiagramas.

Lo común en la construcción de caminos es por el extremo denominado “último nodo”. Sin embargo, ocasionalmente es posible extender el camino por el primer nodo, es decir, añadir un nuevo nodo o arco tal que éste devenga el primero. Esto se hace mediante las contrapartes insert().

Los extremos de un camino se consultan mediante las primitivas siguientes:

580c *(Miembros públicos de path 578c) +≡* (578a) ◁ 580b 581a ▷  
 Node \* get\_first\_node() { return list.get\_first().node; }  
  
 Node \* get\_last\_node() { return list.get\_last().node; }  
  
 Arc \* get\_first\_arc() { return list.get\_first().arc; }  
  
 Arc \* get\_last\_arc()  
{  
 typename DynDlist<Path\_Desc>::Iterator it(list);  
 it.reset\_last();  
 it.prev();  
 return it.get\_current().arc;  
}

Uses DynDlist 85a and Path\_Desc 579a.

De estas rutinas, quizá la que merece una explicación, es get\_last\_arc(). Recordemos primero que un DynDlist<T> sólo puede accederse por sus extremos. Si se desea un

elemento diferente, entonces es necesario recorrer la secuencia a través de un iterador. Ahora bien, `get_last_arc()` debe retornar el último arco del camino, pero éste no se encuentra en el último `Path_Desc` de `list`, sino en el penúltimo. Esto es lo que motiva que el iterador se posicione en el último `Path_Desc` y luego retroceda una posición. De esta forma, desde el penúltimo `Path_Desc` se accede al último arco.

Muchos algoritmos, sobre todo los que efectúan retroceso (backtracking), construyen caminos cuyos extremos finales deben ser borrados. Por esta razón es indispensable la siguiente primitiva:

581a *(Miembros públicos de path 578c) +≡*  
`void remove_last_node()  
{  
 list.remove_last();  
 list.get_last().arc = NULL;  
}`

(578a) ◁580c 581b ▷

la cual borra el último arco (junto con su nodo) del camino.

Luego de construido un camino, puede requerirse recorrerlo. Para eso se provee un iterador:

581b *(Miembros públicos de path 578c) +≡*  
`class Iterator : public DynDlist<Path_Desc>::Iterator  
{  
 // ...  
 Path_Desc & get_curr_path_desc()  
 {  
 return this->DynDlist<Path_Desc>::Iterator::get_current();  
 }  
 Node * get_current_node() { return this->get_curr_path_desc().node; }  
  
 Arc * get_current_arc()  
 {  
 return this->get_curr_path_desc().arc;  
 }  
 bool has_current_arc() const  
 {  
 return this->has_current() and not this->is_in_last();  
 }  
 bool has_current_node() const { return this->has_current(); }  
};`

(578a) ◁581a

Uses `DynDlist 85a` and `Path_Desc 579a`.

## 7.5 Recorridos sobre grafos

Se han identificado dos patrones arquetípicos de exploración, llamados búsqueda en profundidad y en amplitud respectivamente. La mayoría de los algoritmos sobre grafos exhibe una u otra o ambas maneras de explorarlo.

La idea de los recorridos es similar a los de los árboles estudiados en el capítulo 4, es decir, un patrón de procesamiento presente en muchos algoritmos.

Los recorridos y otras primitivas basadas en ellos se definen en el archivo `tpl_graph_utils.H`.

### 7.5.1 Iteradores filtro

Es probable que la idea de iterador se encuentre entre los más útiles y versátiles patrones de diseño [57]. Hasta el presente, cualquiera de los iteradores que hemos presentado comprende la totalidad del conjunto al que refieren.

Prontamente constataremos que muchos problemas modelizados sobre grafos requieren sofisticadas estructuras de datos, así como el empleo de distintos algoritmos. A causa de esto, por lo común, diseñar e instrumentar algoritmos sobre grafos no es una tarea sencilla, máxime cuando pretendemos la generalidad y genericidad. Por añadidura, los grafos requieren bastante espacio de memoria, por lo que su ahorre puede ser crítico en muchas situaciones.

A veces es importante procesar algunas clases de nodo o arco de un grafo y descartar otros. Una manera genérica de filtrar la secuencia presentada por un iterador consiste en usar un “iterador filtro”, el cual en interfaz es muy similar a un iterador tradicional. Para un iterador filtro debe especificarse: el conjunto sobre el cual se itera, un iterador sobre aquel conjunto y una función que decide si un elemento devuelto por el iterador debe o no ser mostrado por el iterador filtro. Su especificación reside en el archivo `filter_iterator.H`, el cual se define como sigue:

582a

```
<filter_iterator.H 582a>≡
 template <class Container, class It, class Show_Item>
 class Filter_Iterator : public It
 {
 <Miembros privados de iterador filtro 582b>
 <Miembros públicos de iterador filtro 583b>
 };
```

Defines:

`Filter_Iterator`, used in chunks 583–85.

Cuando se instancia un iterador filtro (`Filter_Iterator`), se especifican tres clases parametrizadas según lo ya mencionado:

1. `Container`: el tipo de conjunto sobre el cual se itera.
2. `It`: un iterador sobre el conjunto `Container`.

Internamente, el iterador filtro instancia un iterador de tipo `Container::It`, con el cual se recorren los elementos de `Container`.

3. `Show_Item`: una clase que determina si un elemento del conjunto debe o no ser mostrado por el iterador. Esto se efectúa mediante la llamada lógica `bool Show_Item::operator()(Container&, Container::Item_Type&)`, la cual debe retornar `true` si el elemento debe mostrarse o `false` de lo contrario.

La implantación es relativamente sencilla y está fundamentada en la derivación pública de `Container::It`. Para instrumentar la llamada convenida a `Show_Item()()` requerimos guardar el conjunto en puntero como atributo de la clase:

582b

```
<Miembros privados de iterador filtro 582b>≡
 Container * cont;
 Show_Item show_item;
```

(582a) 583a▷

Además, emplearemos dos tipo de rutinas, una para poner el iterador en uno de los extremos de la secuencia y otra para avanzarlo; éstas se realizan de la siguiente manera:

583a *<Miembros privados de iterador filtro 582b>*≡ (582a) ▷582b

```

void goto_first_valid_item()
{
 try
 { // colocarse en el primer elemento que acepte show_item
 for (It::reset_first(); true; It::next())
 if (not It::has_current() or
 show_item (*cont, It::get_current()))
 return;
 }
 catch (std::overflow_error) { /* seguir overflow; no propagar */ }
}
void forward()
{
 It::next();
 try
 { // avanzar hasta el siguiente item que acepte show_item
 for (;true; It::next())
 if (not It::has_current() or
 show_item (*cont, It::get_current()))
 return;
 }
 catch (std::overflow_error) { /* seguir overflow; no propagar */ }
}
```

`goto_first_valid_item()` pone el iterador filtro en el primer elemento válido que determine `show_item(*cont, It::get_current())`. `forward()` lo avanza hasta la primera posición válida según el mismo criterio. En ambas operaciones existe la posibilidad de que no haya más elementos que mostrar. En este caso, el iterador debe dejarse en el estado de desborde `std::overflow_error` sin propagar la excepción (a menos que se ejecute un `next()` sobre un iterador desbordado, y esta es la razón por la cual el primer `next()` sobre `forward()` no atrapa la excepción).

Hay dos rutinas análogas para poner el iterador en el último elemento y avanzar hacia atrás: `goto_last_valid_item()` y `backward()`.

Las rutinas de avance definidas ocultan el “enmascaramiento” de elementos a través de `show_item`. Mediante ellas, la instrumentación debe ser fácilmente comprensible:

583b *<Miembros públicos de iterador filtro 583b>*≡ (582a)

```

typedef typename It::Item_Type Item_Type;
```

```

Filter_Iterator(Show_Item si = Show_Item()) : cont(NULL), show_item(si) {}

Filter_Iterator(Container & cont, Show_Item si = Show_Item())
 : It(cont), cont(&cont), show_item(si)
{
 goto_first_valid_item();
}
void next() { forward(); }
```

```
void prev() { backward(); }

void reset_first() { goto_first_valid_item(); }

void reset_last() { goto_last_valid_item(); }

Uses Filter_Iterator 582a.
```

Mediante esta primitiva un iterador filtro ofrece la ilusión de ser un iterador de *ALEPH* (ofrece exactamente su misma interfaz).

Los iteradores de *ALEPH* fueron diseñados pensando en el rendimiento y legibilidad. Principalmente por esa razón, éstos no exportan una interfaz reminiscente a la de la biblioteca `stdc++`, pues ésta última consume tiempos en copias temporales<sup>13</sup>.

#### 7.5.1.1 Iterador filtro de arcos de un nodo

El iterador filtro que acabamos de definir es aplicable a prácticamente cualquier contenedor de *ALÉPH*, los relacionados a los grafos inclusive. Consecuentemente, según las circunstancias algorítmicas, algunas veces necesitaremos iterar de manera que los algoritmos no vean algunas clases de arcos; los usados para cálculo parciales, por ejemplo. Por esta razón definiremos un iterador externo a la clase `List_Graph` que opera sobre los arcos de un nodo. Tal clase se especifica del modo siguiente:

584a

*⟨Iteradores filtros de List\_Graph 584a⟩* ≡

584b ▷

```
template <class GT, class Show_Arc = Default_Show_Arc<GT> >
class Node_Arc_Iterator : public Filter_Iterator<typename GT::Node*,
 typename GT::Node_Arc_Iterator,
 Show_Arc>
```

Uses Filter\_Iterator 582a.

Siendo derivada de `Filter_Iterator` con el iterador de nodos del grafo, la interfaz de `Node_Arc_Iterator` es idéntica a `List_Graph::Node_Arc_Iterator`.

La clase global `Node_Arc_Iterator` tiene una especialización por omisión que no efectúa ningún filtrado y que es implantado por la clase `Default_Show_Arc`. Es decir, si no se especifica una clase filtro de arcos, entonces `Node_Arc_Iterator` muestra todos los arcos.

Este iterador será ampliamente utilizado por los algoritmos de grafos. Su versión por omisión es en desempeño equivalente al iterador `List_Graph::Node_Arc_Iterator`, pues el carácter “inline” de los métodos permite al compilador realizar directamente la llamada.

### 7.5.1.2 Iterador filtro de arcos

Similar al iterador anterior, pero para todos los arcos de grafo, definimos un iterador filtro:

584b

*⟨Iteradores filtros de List\_Graph 584a⟩* +≡

△584a 585a▷

```
template <class GT, class Show_Arc = Default_Show_Arc<GT> >
class Arc_Iterator :
 public Filter_Iterator<GT, typename GT::Arc_Iterator, Show_Arc>
```

Uses Filter Element

El uso de este iterador es menos frecuente que Node Arc Iterator.

<sup>13</sup>Un ejemplo típico:

```
for (typename set<T>::iterator = s.begin(); it < s.end(); it++)
```

### 7.5.1.3 Iterador filtro de nodos

Mucho menos frecuente de uso, pero plausible, definimos un iterador filtro sobre los nodos:

585a *(Iteradores filtros de List\_Graph 584a) +≡* 584b  
 template <class GT, class Show\_Node = Default\_Show\_Node<GT> >  
 class Node\_Iterator :  
 public Filter\_Iterator<GT, typename GT::Node\_Iterator, Show\_Node>  
 Uses Filter\_Iterator 582a.

### 7.5.2 Recorrido en profundidad

En un recorrido en profundidad, el grafo se visita tratando de conformar el camino más largo posible antes de retornar a un nodo ya visitado.

Es un recorrido de índole recursiva. Si `__dft(p)` fuese la primitiva de recorrido sobre el nodo `p`, entonces el algoritmo general sería como sigue.

**Algoritmo 7.1 (Recorrido en profundidad desde el nodo p)** El algoritmo es recursivo con prototipo `__dft(p)`, donde `p` es el nodo actual de visita.

Se usa una variable global, `num_nodes`, cuyo valor inicial es cero, para señalar la cantidad de nodos visitados.

1. Si `p` está pintado, es decir, ya fue marcado como visitado  $\Rightarrow$  retorne.
2. Pintar `p`.
3. Incrementar `num_nodes` en uno.
4. Si `num_nodes == |V|`  $\Rightarrow$  todos los nodos del grafo han sido recorridos y el algoritmo termina.
5. Para todo arco `a` adyacente a `p`:
  - (a) Sea `q` el nodo incidente desde `p` a través del arco `a`.
  - (b) Llamar a `__dft(q);`

Se le dice en “profundidad” porque, dado el nodo `p`, el recorrido no revisa un nuevo arco `p → r` hasta no haber recorrido completamente en profundidad el nodo `q` desde el arco actual `p → q`.

Muchas veces, un problema sobre grafos consiste en buscar un nodo con algunas características que representan la solución. Por esta razón, a los recorridos sobre grafos también se les llama, indistintamente, “búsquedas”. Emplearemos como sinónimo, entonces, la expresión “búsqueda en profundidad” para referir al recorrido homónimo.

Veamos ahora cómo codificar el recorrido en profundidad. Para eso examinemos el prototipo de la función recursiva que nos implantará el algoritmo 7.1:

585b *(Recorrido en profundidad 585b) ≡* 586 ▷  
 template <class GT, class SA> inline static bool  
 \_\_depth\_first\_traversal(GT & g, typename GT::Node \* node,  
 typename GT::Arc \* arc,  
 bool (\*visit)(GT & g, typename GT::Node \*,  
 typename GT::Arc \*),  
 size\_t & count);

Esta es la rutina recursiva que implanta el algoritmo 7.1.

`__depth_first_traversal<GT,SA>()` opera sobre un objeto derivado de `List_Graph`, el cual está expresado por el parámetro tipo `GT`. El parámetro tipo `SA` representa el criterio de mirada de los arcos; internamente, `__depth_first_traversal()` utiliza un iterador filtro sobre los arcos. De este modo se puede configurar el recorrido para considerar o no arcos del grafo según algún criterio.

Los parámetros de la función se describen así:

1. `g`: el grafo sobre el cual se hace el recorrido.
2. `node`: el nodo que se está visitando.
3. `arc`: el arco desde el cual se llega al nodo `node`.
4. `(*visit)()`: puntero a la función de visita. Si este puntero es distinto de nulo, entonces la función de visita se invoca la primera vez que se ve un nodo.

La función de visita tiene los siguientes parámetros:

- (a) `g`: el grafo que se está visitando.
- (b) `_n`: puntero al nodo visitado.
- (c) `_a`: puntero al arco que conduce al nodo visitado `_n`.

`(*visit)()` retorna un valor lógico. Si es `true`, entonces se asume que la búsqueda ha terminado; de lo contrario, la búsqueda prosigue hasta que `(*visit)()` retorne `true` o hasta que se hayan visitado todos los arcos y nodos del grafo según su conectividad.

5. `node_counter`: parámetro por referencia que contabiliza la cantidad de nodos visitados.

`__depth_first_traversal()` recorre recursivamente el grafo a partir del nodo `node`. El valor de retorno de `(*visit)()` permite detener la búsqueda según algún criterio específico embebido en la función.

El recorrido en profundidad se invoca por primera vez desde la siguiente rutina de uso público:

```
586 <Recorrido en profundidad 585b>+≡ <585b 587>
 template <class GT, class SA = Default_Show_Arc<GT>> inline size_t
 depth_first_traversal(GT & g, typename GT::Node * start_node,
 bool (*visit)(GT & g, typename GT::Node *,
 typename GT::Arc *),
 SA sa = SA())
 {
 g.reset_bit_nodes(Depth_First); // reiniciar Depth_First de nodos
 g.reset_bit_arcs(Depth_First); // reiniciar Depth_First de arcos
 size_t counter = 0; // inicialmente no se ha visitado ning n nodo

 __depth_first_traversal<GT,SA>(g, start_node, NULL, visit, counter, sa);

 return counter;
 }
```

la cual explora en profundidad el grafo  $g$  a partir del nodo `start_node` y retorna la cantidad de nodos visitados. Si se especifica una función de visita, entonces ésta se invoca cada vez que durante la exploración se descubra un nodo.

Hay otra versión, que no requiere especificar el nodo de inicio y que arranca la visita sobre un nodo cualquiera del grafo (el que arroja `get_first_node()`).

Antes de implantar `_depth_first_traversal()` es conveniente hacer algunas aclaratorias que nos ayudarán a comprender el código de éste y otros algoritmos sobre grafos.

En primer lugar acordemos la manera de marcar los nodos, la cual será mediante un bit de control, en nuestro caso, el bit `Depth_First`. En segundo lugar, a efectos de acelerar el recorrido, vamos a marcar también los arcos vistos. Esto nos ahorra llamadas recursivas que van a caer sobre nodos previamente visitados desde otro camino. Hechas las aclaratorias pertinentes podemos presentar la rutina recursiva:

```
587 <Recorrido en profundidad 585b>+≡ ◁586
 template <class GT, class SA> inline static bool
 _depth_first_traversal(GT & g, typename GT::Node * node,
 typename GT::Arc * arc,
 bool (*visit)(GT & g, typename GT::Node *,
 typename GT::Arc *),
 size_t & count,
 SA sa = SA())
 {
 if (IS_NODE_VISITED(node, Depth_First))
 return false;

 NODE_BITS(node).set_bit(Depth_First, true); // marca nodo visitado
 count++;

 /* Aquí se visita el nodo si se desea que el orden sea prefijo */

 if (count == g.get_num_nodes()) // ¿se visitaron todos los nodos?
 return true;

 // recorrer recursivamente arcos de node
 for (Node_Arc_Iterator<GT, SA> it(node, sa); it.has_current(); it.next())
 {
 typename GT::Arc * arc = it.get_current_arc();
 if (IS_ARC_VISITED(arc, Depth_First))
 continue;

 ARC_BITS(arc).set_bit(Depth_First, true); // visitado
 if (_depth_first_traversal<GT, SA>(g, it.get_tgt_node(), arc,
 visit, count, sa))
 return true; // ya se exploró cabalmente it.get_tgt_node()
 }
 /* Aquí se visita si se desea que el orden sea sufijo */

 return false; // retorne y siga explorando
 }
```

La rutina está configurada para detenerse cuando se hayan visitado todos los nodos. Si las circunstancias del cálculo lo ameritan, puede ser necesario recorrer todos los arcos.

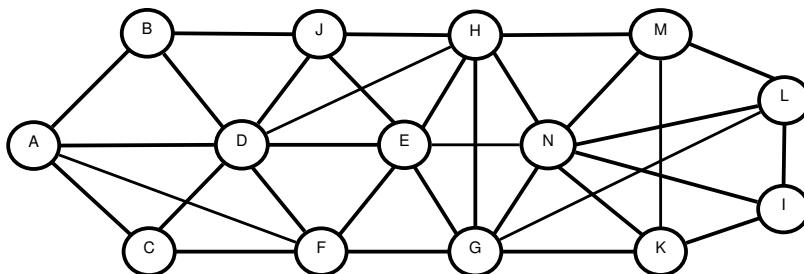


Figura 7.21: Un grafo ejemplo

En este caso basta con eliminar el `if (count == g.get_num_nodes())` para asegurar que se vean todos los nodos y arcos conexos.

La figura 7.22 ilustra dos interpretaciones de recorrido para el grafo de la figura 7.21, un recorrido que comienza en el nodo A y otro en N. Una vez establecido el nodo de inicio, el orden de visita depende de la topología del grafo y del orden en que se presenten los arcos en las listas de adyacencia de cada nodo.

El recorrido en profundidad puede plantearse como prefijo o sufijo. Si el procesamiento o visita del nodo se efectúa antes del for, entonces el recorrido es prefijo. Si, por el contrario, se hace después del for, entonces es sufijo. En ambos tipos, el orden de visita es el mismo que si se hiciese sobre el árbol abarcador que caracteriza el recorrido.

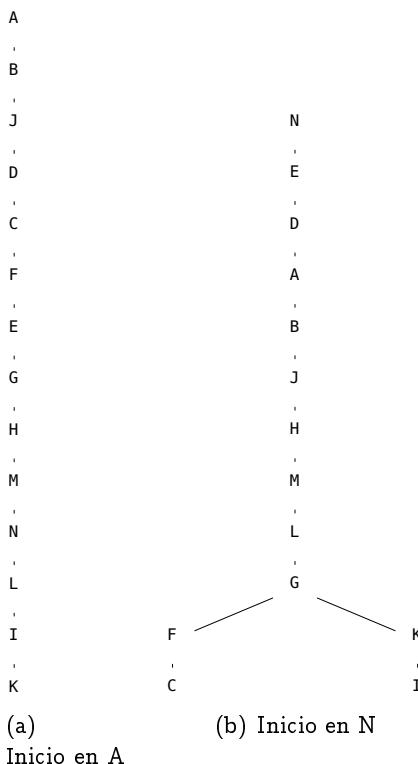


Figura 7.22: Árboles abarcadores de profundidad del grafo de la figura 7.21

El recorrido en profundidad se usa cuando se estima que el nodo solución está “lejano” del inicio. Notemos que los árboles de profundidad de la figura 7.22 son “largos y delgados”, con pocas ramas. El primero es completamente secuencial porque a partir de A es posible

visitar enteramente el grafo sin necesidad de regresar a un nodo ya visitado. En el segundo árbol vemos un regreso al nodo G.

Si el grafo es conexo, entonces la búsqueda en profundidad requiere a lo sumo  $\mathcal{O}(V) + \mathcal{O}(E) = \mathcal{O}(\max(V, E))$  pasos para inicializar los bits de los nodos y arcos. Posteriormente, la rutina comienza en el primer nodo del grafo y a lo sumo llama recursivamente V veces a `__depth_first_traversal()`. Durante cada llamada recursiva se recorre enteramente la lista de adyacencia del nodo de visita, lo cual implica que, en el peor de los casos, se visitarán todos los arcos. Por tanto, la búsqueda en profundidad con listas enlazadas es, para el peor caso,  $\mathcal{O}(V) + \mathcal{O}(E)$ .

El peor coste en espacio de la exploración en profundidad ocurre si el grafo es fuertemente conexo. Una indicación de esto se muestra mirando el árbol abarcador en profundidad de la figura 7.22-(a). Observemos que tiene una sola rama, lo que es equivalente a una lista enlazada. En este caso, la profundidad recursiva es proporcional a  $\mathcal{O}(V)$ .

Si el grafo no es conexo, entonces el recorrido culmina sin haber visitado todos los nodos; aquéllos que sean visitados dependerán del nodo por el cual se inicie la búsqueda.

El recorrido que hemos presentado busca nodos, por eso podemos detenerlo cuando se han visitado todos los nodos. En ocasiones, la búsqueda se plantea en función de arcos, en cuyo caso, el recorrido tiende a ser más costoso, pues la detención debe hacerse cuando se hayan visitado todos los arcos.

### 7.5.3 Conectividad entre grafos

Una de las aplicaciones más simples de la búsqueda en profundidad es la prueba de conectividad. La idea básica es recorrer el grafo y verificar si se visitan todos los nodos, en cuyo caso podemos concluir con certitud que el grafo es conexo.

Hay una consideración adicional que hace nuestro algoritmo, y que consiste en revisar la cantidad de arcos, la cual, si es menor al número de nodos menos uno, entonces podemos concluir que el grafo es inconexo y ahorrarnos el recorrido.

La rutina resultante se implanta entonces de la siguiente manera:

```
589 <Prueba de conectividad 589>≡
 template <class GT, class SA = Default_Show_Arc<GT>> inline
 bool test_connectivity(GT & g)
 {
 if (g.get_num_arcs() < g.get_num_nodes() - 1)
 return false;

 return depth_first_traversal<GT, SA>(g, NULL) == g.get_num_nodes();
 }
```

`test_connectivity()` retorna true si el grafo es conexo, false de lo contrario.

La prueba sobre el número de arcos toma tiempo  $\mathcal{O}(1)$  y puede ahorrar tiempo de ejecución. Empero, esta no es correcta si tratamos con un multigrafo.

`test_connectivity()` no opera sobre digrafos. Esta clase de conectividad se estudia en § 7.7.1 (Pág. 638).

### 7.5.4 Recorrido en amplitud

Otra forma de procesar el recorrido sobre un grafo que privilegia los nodos más cercanos sobre los más lejanos y que, según la índole del grafo, puede ser más conveniente, es el recorrido en “amplitud”. En esta clase de búsqueda no se procesa un nodo de un nivel i

hasta que no se hayan procesado todos los nodos del nivel anterior  $i - 1$ , es decir, una especie de recorrido por niveles. El medio para guardar los nodos por niveles es el mismo que para los árboles: una cola. En nuestro caso usamos una cola de punteros a arcos.

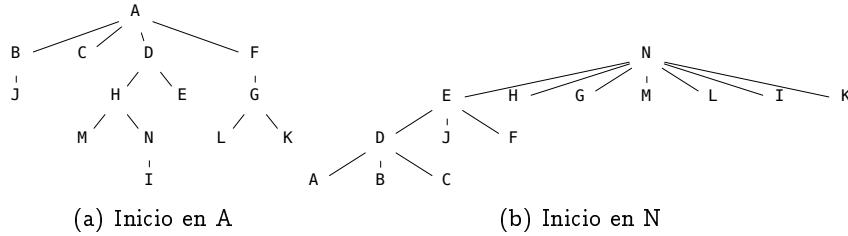


Figura 7.23: Árboles abarcadores de amplitud del grafo de la figura 7.21

La diferencia esencial entre la búsqueda en amplitud y la de profundidad es que la de amplitud explora los caminos más cortos primero, mientras que la de profundidad tiende a ir por los más largos. Este hecho se indica perfectamente contrastando los árboles de recorridos, los cuales son más “frondosos” para el recorrido en amplitud (ver figuras § 7.22 (Pág. 588) y § 7.23 (Pág. 590)).

La primitiva no es recursiva. Tiene los mismos parámetros que su contraparte en profundidad y se instrumenta del siguiente modo:

```
590 <Recorrido en amplitud 590>≡ 600d▷
 template <class GT, class SA = Default>Show_Arc<GT>> inline size_t
 breadth_first_traversal(GT & g, typename GT::Node * start,
 bool (*visit)(GT &, typename GT::Node *,
 typename GT::Arc *));
{
 g.reset_bit_nodes(Breadth_First);
 g.reset_bit_arcs(Breadth_First);
 DynListQueue<typename GT::Arc*> q; // cola de arcos pendientes

 // ingresar a la cola los arcos de nodo start
 for (Node_Arc_Iterator<GT, SA> it(start); it.has_current(); it.next())
 q.put(it.get_current_arc());

 NODE_BITS(start).set_bit(Breadth_First, true);
 size_t node_counter = 1; // contador de nodos visitados

 // mientras queden arcos en cola y resten todos por visitar
 while (not q.is_empty() and node_counter < g.get_num_nodes())
 {
 typename GT::Arc * arc = q.get(); // saque de cola más cercano

 ARC_BITS(arc).set_bit(Breadth_First, true);

 typename GT::Node * src = g.get_src_node(arc);
 typename GT::Node * tgt = g.get_tgt_node(arc);
 if (IS_NODE_VISITED(src, Breadth_First) and
 IS_NODE_VISITED(tgt, Breadth_First))
 continue;
```

```

typename GT::Node * visit_node = // próximo a visitar
 IS_NODE_VISITED(src, Breadth_First) ? tgt : src;

/* Aquí se visita el nodo mediante la función de visita */

NODE_BITS(visit_node).set_bit(Breadth_First, true);
node_counter++;

// insertar en cola arcos del nodo recién visitado
for (Node_Arc_Iterator<GT, SA> it(visit_node); it.has_current(); it.next())
{
 typename GT::Arc * curr_arc = it.get_current_arc();
 if (IS_ARC_VISITED(curr_arc, Breadth_First))
 continue;

 // revise nodos del arcos para ver si han sido visitados
 if (IS_NODE_VISITED(g.get_src_node(curr_arc), Breadth_First) and
 IS_NODE_VISITED(g.get_tgt_node(curr_arc), Breadth_First))
 continue; // nodos ya visitados

 q.put(curr_arc);
}
}

return node_counter;
}

```

La búsqueda en amplitud puede ser más costosa en tiempo y en espacio que la de profundidad, pues el encolamiento mantiene arcos que no han sido visitados, lo que no sucede con el empilamiento de la búsqueda en profundidad que almacena arcos visitados.

El peor caso de ejecución de la búsqueda en amplitud ocurre cuando es necesario visitar todos los arcos antes de haber visto todos los nodos, o sea,  $\mathcal{O}(E)$ , cual es el mismo de la búsqueda en profundidad.

Para analizar el coste en espacio es conveniente observar la dinámica de inserción y eliminación de arcos en la cola. Cuando se saca de la cola un arco perteneciente al nivel  $i$  (respecto al nodo de inicio), ésta puede contener arcos del nivel  $i$  y del superior  $i + 1$ . Si interpretamos el recorrido como el recorrido por niveles de un árbol, entonces podemos percatarnos de que a lo sumo se tendrán  $E/2$  arcos en la cola, lo cual equivale a un coste  $\mathcal{O}(E)$ , cual puede ser bastante mayor que el coste en espacio  $\mathcal{O}(V)$  del recorrido en profundidad.

¿Cuándo recorrer en profundidad o en amplitud? La respuesta depende de varios factores, de los cuales el principal es la presunción de cercanía que se tenga acerca del nodo solución considerando que al criterio de cercanía incide la topología del grafo y su densidad. Si el nodo solución se presume cercano, entonces el criterio de amplitud tiende a encontrarlo más rápido que el de profundidad.

La búsqueda en amplitud de un nodo particular encuentra el camino más corto en cantidad de arcos.

Por costumbre, no generalidad, se tiende a hacer primero una búsqueda en profundidad para encontrar una solución o una aproximación y, posteriormente, para afinarla, se hace una en amplitud.

### 7.5.5 Prueba de ciclos

Planteemos el problema de determinar si existe o no un ciclo a partir de un nodo particular `src_node`. En este caso, un criterio en profundidad parece preferible al de amplitud porque no gasta tiempo ni espacio en los arcos parciales que aún no son procesados.

Una búsqueda en profundidad parcial, es decir, que no necesariamente pretenda visitar todos los nodos, sirve para detectar si existe un ciclo a partir de un nodo. La idea es explorar recursivamente hasta encontrar el nodo de partida, en cuyo caso la prueba es positiva, o culminar la búsqueda, en cuyo caso es negativa.

Nuestra prueba de ciclo se apoya sobre la siguiente rutina recursiva:

592a *(Búsqueda de ciclo 592a)≡* 592b▷

```
template <class GT, class SA> inline static
bool __test_cycle(GT & g, typename GT::Node * src_node,
 typename GT::Node * curr_node);
```

`__test_cycle()` es de uso privado, efectúa la exploración recursiva en profundidad por el nodo `curr_node` en búsqueda del nodo `src_node` y maneja tres parámetros:

1. `g`: el grafo sobre el cual se realiza la prueba.
2. `src_node`: el nodo para el cual se verifica si existe un ciclo.
3. `curr_node`: es el nodo de visita de la llamada recursiva.

La rutina retorna `true` si desde `curr_node` se alcanzó el nodo inicial `src_node`; en este caso existe un ciclo. Si se recorre entera y recursivamente el componente conectado a `curr_node`, entonces podemos concluir que pasando por `curr_node` no es posible alcanzar a `src_node` y que, por lo tanto, por esta vía no hay ciclo. En este caso retornamos `false`.

La interfaz pública se plantea y se instrumenta de la siguiente forma:

592b *(Búsqueda de ciclo 592a)+≡* ▷592a 593▷

```
template <class GT, class SA = Default_Show_Arc<GT> > inline
bool test_for_cycle(GT & g, typename GT::Node * src_node)
{
 g.reset_bit_nodes(Test_Cycle); // reiniciar Test_Cycle para nodos
 g.reset_bit_arcs(Test_Cycle); // reiniciar Test_Cycle para arcos

 // explorar recursivamente por arcos adyacentes a src_node
 for (Node_Arc_Iterator<GT, SA> it(src_node); it.has_current(); it.next())
 {
 typename GT::Arc * arc = it.get_current_arc();
 if (IS_ARC_VISITED(arc, Test_Cycle))
 continue;

 ARC_BITS(arc).set_bit(Test_Cycle, true); // pintar arco

 if (__test_cycle<GT,SA>(g, src_node, it.get_tgt_node()))
 return true; // ciclo detectado
 }
 // Se han explorado todos los caminos desde src_node
 // sin encontrar de nuevo a src_node ==> no hay ciclo
 return false;
}
```

`test_for_cycle()` retorna true si existe un ciclo desde `src_node`; false de lo contrario.

Un punto esencial en este algoritmo es que, a diferencia de lo que planteamos en las subsecciones anteriores concernientes a las b usquedas, no debemos detenernos cuando hayamos visitado todos los nodos, pues alg un ciclo podr a encontrarse por alg un arco que no haya sido visitado. No tenemos en este caso otra alternativa que explorar todos los arcos, raz n por la cual el algoritmo es  $\mathcal{O}(E)$  para el peor caso en que no exista ciclo.

Dicho lo anterior, estamos listos para implantar la exploración recursiva:

593

*(Búsqueda de ciclo 592a) +≡*

```
template <class GT, class SA> inline static
bool __test_cycle(GT & g, typename GT::Node * src_node,
 typename GT::Node * curr_node)
{
 if (src_node == curr_node)
 return true; // ciclo detectado!

 if (IS_NODE_VISITED(curr_node, Test_Cycle))
 return false;

 NODE_BITS(curr_node).set_bit(Test_Cycle, true); // marque nodo

 // buscar caminos desde current_node a ver si llega a src_node
 for (Node_Arc_Iterator<GT, SA> it(curr_node); it.has_current(); it.next())
 {
 typename GT::Arc * arc = it.get_current_arc();
 if (IS_ARC_VISITED(arc, Test_Cycle))
 continue;

 ARC_BITS(arc).set_bit(Test_Cycle, true); // marque arco

 if (__test_cycle<GT,SA>(g, src_node, it.get_tgt_node()))
 return true; // ciclo encontrado desde el arco actual
 }
 // En este punto se han explorado caminos desde curr_node
 // sin encontrar src_node ==> no existe ciclo por curr_node
 return false;
}
```

### 7.5.6 Prueba de aciclicidad

Un grafo es acíclico si éste no contiene ciclos. Como muchos algoritmos requieren verificar esta condición sobre cálculos de grafos parciales, es importante que esta prueba se haga eficientemente.

Una variante del algoritmo anterior de prueba de ciclos puede utilizarse para verificar aciclicidad sobre un grafo inconexo. Podemos invocar a `test_for_cycle()` para cada nodo del grafo, y si todas las ejecuciones arrojan como resultado falso, entonces el grafo es acíclico.

El principio de la prueba de aciclicidad es que si durante la exploración del grafo, pasando por un arco no visitado, vemos a un nodo ya visitado, entonces el grafo tiene un

ciclo (el que permite regresar al nodo visitado por un camino diferente). En virtud de esta observación podemos plantear la siguiente rutina, privada, recursiva, de exploración en profundidad:

594a *(Prueba de aciclicidad 594a)≡* 594b▷

```
template <class GT, class SA> inline static
bool __is_graph_acyclique(GT & g, typename GT::Node * curr_node)
{
 if (IS_NODE_VISITED(curr_node, Is_Acyclique))
 return false;

 NODE_BITS(curr_node).set_bit(Is_Acyclique, true); // marcar nodo

 for (Node_Arc_Iterator<GT, SA> i(curr_node); i.has_current();
 i.next())
 {
 typename GT::Arc * arc = i.get_current_arc();
 if (IS_ARC_VISITED(arc, Is_Acyclique))
 continue;

 ARC_BITS(arc).set_bit(Is_Acyclique, true);

 if (not __is_graph_acyclique<GT,SA>(g, i.get_tgt_node()))
 return false;
 }
 // todos los arcos recorridos sin encontrar ciclo ==>
 // el grafo es acíclico pasando por curr_node
 return true;
}
```

`__is_graph_acyclique()` realiza una prueba de aciclicidad a partir del nodo `curr_node`. Se retorna `true` si hay aciclicidad; `false` de lo contrario.

La detención se efectúa cuando se detecta un ciclo; en este caso `__is_graph_acyclique()` retorna `false`. El que no se encuentre ciclo por algún arco adyacente de `curr_node` no implica que no exista alguno por otro de sus arcos. Sólo cuando se hayan explorado todos los arcos de `curr_node` sin encontrar ciclo podemos concluir que el grafo es acíclico.

Si un subgrafo de un grafo conexo es cíclico, entonces, sin importar desde cuál nodo se inicie, una búsqueda en profundidad debe detectar un ciclo por encuentro de un nodo ya marcado. Si eso no ocurre, es decir, si no podemos regresar al nodo origen de la búsqueda, entonces, con toda certitud, el subgrafo es acíclico. Si eso sucede para todos los nodos conexos, entonces el grafo es acíclico.

Hay una observación crucial en el caso de un grafo conexo (no un digrafo, multigrafo o multidigrafo): si la cantidad de arcos es menor que la de nodos, entonces el grafo no puede contener ningún ciclo. Este hecho nos acota la prueba de aciclicidad a  $\mathcal{O}(V)$ . La prueba puede iniciarse desde cualquier nodo y la siguiente versión pública es perfectamente aplicable:

594b *(Prueba de aciclicidad 594a)+≡* ▷594a 595▷

```
template <class GT, class SA = Default_Show_Arc<GT> > inline
bool is_graph_acyclique(GT & g, typename GT::Node * start_node)
{
```

```

 if (g.get_num_arcs() >= g.get_num_nodes())
 return false;

 g.reset_bit_arcs(Is_Acyclique);
 g.reset_bit_nodes(Is_Acyclique);

 return __is_graph_acyclique<GT,SA>(g, start_node);
}

```

`__is_graph_acyclique()` realiza la exploración recursiva en profundidad; `start_node` es el nodo de inicio de la prueba, el cual debe pertenecer a un componente de grafo “sospechoso” de tener un ciclo. La utilidad de la interfaz anterior es cuando `start_node` haya sido añadido en algún cálculo y se requiera verificar si la adición causa o no un ciclo. Esta rutina funciona para cualquier grafo conexo (sin importar el nodo de inicio) o para verificar aciclicidad desde un nodo particular `start_node`.

Si no se tiene conocimiento acerca de la conectividad del grafo, entonces es necesario inspeccionar los eventuales componentes conexos del grafo, cuestión que se puede efectuar del siguiente modo:

595 *<Prueba de aciclicidad 594a>+≡* ◁594b

```

template <class GT, class SA = Default_Show_Arc<GT> > inline
bool is_graph_acyclique(GT & g)
{
 if (g.get_num_arcs() >= g.get_num_nodes())
 return false;

 g.reset_bit_arcs(Is_Acyclique);
 g.reset_bit_nodes(Is_Acyclique);
 for (Node_Iterator<GT> it(g); it.has_current(); it.next())
 {
 typename GT::Node * current_node = it.get_current_node();
 if (IS_NODE_VISITED(current_node, Is_Acyclique))
 continue;

 if (not __is_graph_acyclique<GT,SA>(g, current_node))
 return false;
 }
 return true;
}

```

Después de la primera llamada a `__is_graph_acyclique(g, curr_node)`, algunos nodos ya quedan marcados con el valor `Is_Acyclique`; sobre éstos no se debe repetir la prueba de aciclicidad.

Puesto que `is_graph_acyclique()` detecta ciclo verificando si un nodo ha sido o no visitado, ésta no opera para digrafos.

Si tenemos un digrafo, entonces podemos llamar a `test_for_cycle()` para cada nodo del grafo. Si encontramos un ciclo nos detenemos y sabemos que el digrafo no es acíclico. Si probamos para todos los nodos, entonces podemos concluir que el grafo es acíclico. Esta técnica acarrea un coste de  $\mathcal{O}(V \times E)$ .

En § 7.7.4 (Pág. 648) desarrollaremos una técnica que determina la aciclicidad en  $\mathcal{O}(V + E)$ .

### 7.5.7 Búsqueda de caminos por profundidad

La búsqueda de caminos es un problema común y crítico de los grafos. En esta subsección sólo desarrollaremos prueba de existencia y determinación de caminos. En la sección § 7.9 (Pág. 674) estudiaremos varios algoritmos para determinar el camino de coste mínimo.

#### 7.5.7.1 Prueba de existencia

Dado un par de nodos, ¿cómo determinar si existe un camino entre ellos? Llamemos `__test_for_path()` a la rutina recursiva, que explora en profundidad el grafo, en búsqueda de un nodo destino `end_node` y cuyo prototipo se enuncia a continuación:

596a *(Prueba de camino 596a)*≡

```
template <class GT, class SA> inline static
bool __test_for_path(GT & g, typename GT::Node * curr_node,
 typename GT::Node * end_node);
```

596b▷

`g` es el grafo. `curr_node` es el nodo actual que se está visitando y `end_node` es el nodo destino del camino. Si se encuentra a `end_node`, entonces existe el camino pasando por `curr_node` y la rutina retorna `true`. Si se exploran todas las vías por `curr_node` sin encontrar a `end_node`, entonces se concluye que pasando por `curr_node` no existe camino a `end_node` y la rutina retorna `false`.

Para este problema, el conocimiento que tengamos sobre la conectividad del grafo nos ayuda un poco. El grafo puede ser inconexo y aun así existir camino entre los nodos considerados. Así las cosas, en esta situación no tenemos más alternativa que explorar el grafo. Pero si el grafo es conexo y no es dirigido, entonces con certitud existe un camino entre cualquiera de sus nodos y no requerimos explorarlo.

Veamos ahora cómo se aplica la rutina anterior para la prueba de existencia de camino entre un nodo inicial `start_node` y otro final `end_node`:

596b *(Prueba de camino 596a)*+≡

△596a 597a▷

```
template <class GT, class SA = Default_Show_Arc<GT>> inline
bool test_for_path(GT& g, typename GT::Node * start_node,
 typename GT::Node * end_node)
{
 // si el grafo es conexo ==> existe camino
 if (not g.is_digraph() and g.get_num_arcs() >= g.get_num_nodes())
 return true;

 g.reset_bit_nodes(Test_Path);
 g.reset_bit_arcs(Test_Path);

 // buscar recursivamente por arcos adyacentes a start_node
 for (Node_Arc_Iterator<GT, SA> i(start_node); i.has_current(); i.next())
 {
 typename GT::Arc * arc = i.get_current_arc();
 ARC_BITS(arc).set_bit(Test_Path, true); // marcar arco
 if (__test_for_path<GT, SA>(g, i.get_tgt_node(), end_node))
 return true;
 }
 // todos los arcos de start_node han sido explorados sin
 // encontrar camino hasta end_node ==> no existe camino
 return false;
```

```

 }

 test_for_path() retorna true si existe un camino entre start_node y end_node;
 false de lo contrario.

```

La prueba de camino parcial, es decir, determinar si hay un camino desde un nodo intermedio curr\_node hasta uno final end\_node se implementa según el patrón de búsqueda en profundidad:

597a *<Prueba de camino 596a>+≡* ◁596b

```

template <class GT, class SA> inline static
bool __test_for_path(GT & g, typename GT::Node * curr_node,
 typename GT::Node * end_node)
{
 if (curr_node == end_node)
 return true; // se alcanzó a end_node

 if (IS_NODE_VISITED(curr_node, Test_Path)) // ¿se visitó curr_node?
 return false; // sí, no explore

 NODE_BITS(curr_node).set_bit(Test_Path, true); // pintar curr_node

 // buscar recursivamente a través de arcos de curr_node
 for (Node_Arc_Iterator<GT, SA> i(curr_node); i.has_current(); i.next())
 {
 typename GT::Arc * arc = i.get_current_arc();
 if (IS_ARC_VISITED(arc, Test_Path))
 continue;

 ARC_BITS(arc).set_bit(Test_Path, true); // pintar arco
 if (__test_for_path<GT, SA>(g, i.get_tgt_node(), end_node))
 return true;
 }
 // todos los arcos adyacentes de curr_node explorados sin
 // encontrar a end_node ==> no existe camino por curr_node
 return false;
}

```

Observemos que, en detrimento de la eficiencia, a través de `__test_for_path()` podemos encontrar ciclos o lazos.

El tiempo de ejecución de `test_for_path()` es, en el peor caso, proporcional a la cantidad de arcos, pues éste no iterará más de  $E$  veces, o sea,  $\mathcal{O}(E)$ .

Con la representación matricial hay una manera de calcular las relaciones de conectividad entre todos los nodos de un grafo. Tal relación se llama “clausura transitiva” y se calcula mediante un algoritmo llamado de “Warshall”, que será presentado en § 7.6.3 (Pág. 635).

### 7.5.7.2 Búsqueda de camino entre dos nodos

En ocasiones, lo que se desea es encontrar y construir un camino cualquiera entre dos nodos. Para eso diseñaremos una primitiva, muy similar a la de la subsección anterior, cuyo fin sea obtener un objeto de tipo Path contentivo de un camino entre dos nodos y con la interfaz siguiente:

597b *<Camino de grafo 578a>+≡* ◁578a 598a▷

```
template <class GT, class SA = Default_Show_Arc<GT> > inline
bool find_path_depth_first(GT & g, typename GT::Node * start_node,
 typename GT::Node * end_node, Path<GT> & path);
```

Uses Path 578a.

`find_path_depth_first()` encuentra y construye, mediante una búsqueda en profundidad, un camino entre `start_node` y `end_node` del grafo `g`. El camino resultante se guarda en el parámetro `path`. La rutina retorna `true` si se logra encontrar un camino; `false` de lo contrario, en cuyo caso, el valor de `path` debe considerarse indeterminado.

La función contempla los siguientes parámetros tipo:

1. `GT`: el tipo de grafo.
2. `SA`: el filtro del iterador de arcos.

Muchos algoritmos requieren buscar caminos entre pares de nodos. Por eso puede ser útil parametrizar el método `find_path_depth_first()`, o sea, que éste sea pasado como parámetro a un método. Desde el lenguaje C, el medio tradicional para esto es un puntero a función, pero el estándar C++ no permite punteros a funciones plantillas. Para contornear esta situación, y a la vez dar un estilo más hacia los objetos, exportaremos `find_path_depth_first()` bajo el nombre de una clase:

598a

*(Camino de grafo 578a) +≡* △597b 598b ▷

```
template <class GT, class SA = Default_Show_Arc<GT> >
class Find_Path_Depth_First
{
 bool operator () (GT & g, typename GT::Node * start_node,
 typename GT::Node * end_node, Path<GT> & path) const
 {
 return find_path_depth_first<GT,SA>(g, start_node, end_node, path);
 }
};
```

Uses Path 578a.

`find_path_depth_first()` es la interfaz pública, la cual se apoya sobre la rutina siguiente, que es la que efectúa la exploración recursiva en profundidad:

598b

*(Camino de grafo 578a) +≡* △598a 599 ▷

```
template <class GT, class SA> inline
bool __find_path_depth_first(GT& g, typename GT::Node * curr_node,
 typename GT::Arc * curr_arc,
 typename GT::Node * end_node, Path<GT> & curr_path)
{
 if (curr_node == end_node) // ¿se alcanzó nodo final?
 {
 // sí, terminar añadir el arco y terminar
 curr_path.append(curr_arc);
 return true;
 }
 if (IS_NODE_VISITED(curr_node, Find_Path)) // ¿ha sido visitado?
 return false; // sí ==> desde él no hay camino

 curr_path.append(curr_arc); // añadir curr_arc al camino
 NODE_BITS(curr_node).set_bit(Find_Path, true);
```

```

 // buscar recursivamente a través de arcos de curr_node
 for (Node_Arc_Iterator<GT, SA> i(curr_node); i.has_current(); i.next())
 {
 typename GT::Arc * next_arc = i.get_current_arc();
 if (IS_ARC_VISITED(next_arc, Find_Path))
 continue;

 ARC_BITS(next_arc).set_bit(Find_Path, true);
 typename GT::Node * next_node = i.get_tgt_node();
 if (_find_path_depth_first<GT, SA> (g, next_node, next_arc,
 end_node, curr_path))
 return true; // se encontró camino
 }
 curr_path.remove_last_node();

 return false;
}

```

Uses Path 578a.

La estructura es la misma que la de las diferentes exploraciones en profundidad que hemos presentado. La diferencia estriba en que antes de visitar recursivamente los arcos de curr\_node, lo añadimos al camino. Si recorremos todos los arcos de curr\_node sin encontrar un camino hacia end\_node, entonces sacamos a curr\_node del camino. Dada la índole recursiva del algoritmo, path funge como pila. Por tanto, el máximo tamaño de path podría alcanzar  $\mathcal{O}(E)$ .

\_find\_path\_depth\_first() retorna true si se logra construir un camino hacia end\_node proveniendo desde current\_node. Hay dos parámetros adicionales respecto a \_\_test\_for\_path(): el arco actual current\_arc y el camino path. path es un parámetro de salida que contiene el valor del camino encontrado. current\_arc es un arco cuyo nodo origen es current\_node que se requiere porque en la clase Path se inserta por arcos y no por nodos.

Nos falta implantar la primitiva pública:

599 ⟨Camino de grafo 578a⟩+≡ ◁598b

```

template <class GT, class SA = Default_Show_Arc<GT>> inline
bool find_path_depth_first(GT & g, typename GT::Node * start_node,
 typename GT::Node * end_node, Path<GT> & path)
{
 path.clear_path();
 path.init(start_node);
 g.reset_bit_nodes(Find_Path);
 g.reset_bit_arcs(Find_Path);
 NODE_BITS(start_node).set_bit(Find_Path, true);

 // explorar recursivamente cada arco de start_node
 for (Node_Arc_Iterator<GT, SA> i(start_node); i.has_current(); i.next())
 {
 typename GT::Arc * arc = i.get_current_arc();
 ARC_BITS(arc).set_bit(Find_Path, true);
 typename GT::Node * next_node = i.get_tgt_node();
 if (IS_NODE_VISITED(next_node, Find_Path))

```

```

 continue;

 if (_find_path_depth_first<GT, SA>(g, next_node, arc, end_node, path))
 return true;
 }
 return false;
}

```

Cuando se

nodos es preferible valerse, en una primera instancia, de una exploración en profundidad. Luego, si se requiere uno más corto en arcos, o que consuma menos memoria, entonces puede hacerse por amplitud. Otra situación en la cual es preferible la exploración en profundidad es cuando se busque un camino sobre un grafo acíclico o un árbol.

### 7.5.8 Búsqueda de caminos por amplitud

El fin de esta subsección es construir un camino por amplitud entre dos nodos `start` y `end`. Al comenzar por `start` exploramos y encolamos todos sus arcos en búsqueda de `end`. Si no hemos hallado `end`, entonces continuamos por todos los caminos de longitud dos. Este proceso continúa hasta encontrar el camino en cuestión.

Para llevar a cabo la estrategia anterior, la cola debe guardar caminos parciales desde el nodo de inicio hasta el del último nivel que se haya explorado, lo cual se especifica del siguiente modo:

600a    *Declaración de cola de caminos 600a* ≡ DynListQueue<Path<GT>\*> q; // cola de caminos parciales (601)

Nótese que es una cola de punteros a caminos, lo cual implica que la memoria de éstos debe apartarse y, sobre todo, cuando se vacíe la cola, liberarse. Como se trata de punteros Path\*, el destructor de la cola no liberará la memoria ocupada por los caminos, lo cual requiere la siguiente acción explícita para liberar la cola:

600b      *\vaciar cola y liberar caminos 600b*  $\equiv$  (601)  
               while (not q.is\_empty())  
               delete q.get();

Esta acción debe hacerse al final del procedimiento de la búsqueda, se haya encontrado o no un camino, o si ocurre una excepción.

Para manejar el camino actual de exploración, manejaremos la siguiente declaración de camino actual 600c ≡ Path<GT> \* path\_ptr = NULL; Uses Path 578a.

La inserción en la cola requiere (1) apartar memoria para el camino y (2) la propia acción de insertar en la cola. Puesto que hay manejo de memoria en ambas operaciones, usaremos una rutina con auto-punteros:

```

{
 unique_ptr<Path<GT>> path_auto;
 if (path_ptr == NULL) // ¿iniciar o copiar*
 path_auto = unique_ptr<Path<GT>>(new Path<GT>(g, node)); // iniciar
 else
 path_auto = unique_ptr<Path<GT>>(new Path<GT>(*path_ptr));// copiar

 path_auto->append(arc); // añadir el nuevo arco
 q.put(path_auto.get()); // insertar el nuevo camino en cola
 path_auto.release();
}

```

Uses Path 578a.

`__insert_in_queue()` crea un nuevo camino con el arco añadido `arc` y lo encola en la cola `q`. La rutina maneja una variable `Path` llamada `path_auto`, la cual, según el valor de `path_ptr`, se procesa de dos posibles formas:

1. Si `path_ptr == NULL`, entonces `path_auto` es un nuevo camino con nodo de inicio `node`.
2. Si `path_ptr != NULL`, entonces el camino contenido en `*path_ptr` se copia a `path_auto`.

En cualquiera de los dos casos, el arco `arc` se le añade a `path_auto` y éste se inserta en la cola.

Con lo anterior podemos diseñar la siguiente búsqueda de camino en amplitud:

601

```

<Recorrido en amplitud 590>+≡ <600d 602>
 template <class GT, class SA = Default_Show_Arc<GT>> inline
 bool find_path_breadth_first(GT& g, typename GT::Node * start,
 typename GT::Node * end, Path<GT> & path)
 {
 path.clear_path(); // limpiamos cualquier cosa que esté en path
 g.reset_bit_nodes(Find_Path);
 g.reset_bit_arcs(Find_Path);

 <Declaración de cola de caminos 600a>
 <Declaración de camino actual 600c>

 // insertar los caminos parciales con nodo inicio start
 for (Node_Arc_Iterator<GT, SA> i(start); i.has_current(); i.next())
 __insert_in_queue<GT>(g, q, start, i.get_current_arc());

 NODE_BITS(start).set_bit(Find_Path, true); // márquelo visitado
 while (not q.is_empty()) // mientras queden arcos por visitar
 {
 path_ptr = q.get(); // extraiga camino actual

 typename GT::Arc * arc = path_ptr->get_last_arc();
 ARC_BITS(arc).set_bit(Find_Path, true); // marcar arco

 typename GT::Node * tgt = path_ptr->get_last_node();
 if (IS_NODE_VISITED(tgt, Find_Path)) // ¿visitó último nodo?

```

```

 { // si ==> camino no conduce a end ==> borrar camino
 delete path_ptr;
 continue;
 }
 if (tgt == end) // ¿se encontró un camino?
 {
 // si ==> path_ptr contiene camino buscado
 path.swap(*path_ptr); // copiar resultado al parámetro path
 ⟨vaciar cola y liberar caminos 600b⟩
 return true;
 }
 NODE_BITS(tgt).set_bit(Find_Path, true); // marcar último nodo

 // insertar en cola arcos del nodo recién visitado
 for (Node_Arc_Iterator<GT,SA> i(tgt); i.has_current(); i.next())
 {
 typename GT::Arc * curr_arc = i.get_current_arc();
 if (IS_ARC_VISITED(curr_arc, Find_Path)) // ¿arco visitado?
 continue; // si ==> avanzar al siguiente

 // revise nodos del arco para ver si han sido visitados
 if (IS_NODE_VISITED(g.get_src_node(curr_arc), Find_Path) and
 IS_NODE_VISITED(g.get_tgt_node(curr_arc), Find_Path))
 continue; // nodos ya visitados ==> no meter arco

 _insert_in_queue<GT>(g, q, NULL, curr_arc, path_ptr);
 }
 delete path_ptr; // borrar camino extraído de la cola
} // fin while (not q.is_empty())
⟨vaciar cola y liberar caminos 600b⟩

return false;
}

```

Uses Path 578a.

Se debe prestar mucha atención a liberar los caminos que sean vistos y desechados, bien sea porque el nodo ya haya sido visitado o porque path\_ptr ya haya sido copiado.

Como se debe apreciar, `find_path_breadth_first()` consume más memoria que su contraparte en profundidad. Aparte de que la cola tiende a crecer más que una pila, cuando es el caso en el recorrido en profundidad, la cola guarda caminos enteros desde el nodo de inicio.

Al igual que para `find_path_depth_first()`, `find_path_breadth_first()` también puede invocarse mediante una instancia especial de clase:

|     |                                                                |      |
|-----|----------------------------------------------------------------|------|
| 602 | <i>&lt;Recorrido en amplitud 590&gt;+≡</i>                     | <601 |
|     | template <class GT, class SA = Default_Show_Arc<GT> >          |      |
|     | class Find_Path_Breadth_First                                  |      |
|     | {                                                              |      |
|     | bool operator () (GT & g, typename GT::Node * start,           |      |
|     | typename GT::Node * end, Path<GT> & path) const                |      |
|     | {                                                              |      |
|     | return find_path_breadth_first <GT, SA> (g, start, end, path); |      |
|     | }                                                              |      |

```
};
```

Uses Path 578a.

### 7.5.9 Árboles abarcadores de profundidad

El objeto de esta subsección es desarrollar un algoritmo que, a partir de un nodo de inicio, explore en profundidad un grafo y construya otro grafo correspondiente a un árbol abarcador. Tal rutina tiene la siguiente forma general:

603a *(Árboles abarcadores 603a)≡* 604▷  
`template <class GT, class SA = Default_Show_Arc<GT> > inline  
bool find_depth_first_spanning_tree(GT & g, typename GT::Node * gnode, GT & tree)  
{  
(Inicializar construcción árbol abarcador 603b)  
(Recorrer en profundidad nodos adyacentes a gnode 603c)  
 return true;  
}  
find_depth_first_spanning_tree() maneja tres parámetros:`

1. *g*: un grafo conexo sobre el cual se requiere construir un árbol abarcador.
2. *gnode*: el nodo inicial desde el cual se desea comenzar la construcción del árbol abarcador.
3. *tree*: un grafo de salida donde se colocará el árbol abarcador resultante.

El valor de retorno es *true* si existe un árbol abarcador; *false* de lo contrario.

Por añadidura, *find\_depth\_first\_spanning\_tree()* mapea los nodos y arcos a través de los cookies. De este modo, el usuario puede conocer, dentro del grafo, cuáles nodos o arcos pertenecen al árbol abarcador.

Antes de comenzar la exploración recursiva de *g*, es necesario limpiar sus bits de control y asegurarse de que *tree* esté vacío:

603b *(Inicializar construcción árbol abarcador 603b)≡* (603a)  
`g.reset_nodes();  
g.reset_arcs();  
clear_graph(tree); // asegurar que árbol destino esté vacío  
NODE_BITS(gnode).set_bit(Scaling_Tree, true); // marcar gnode  
typename GT::Node * tnode = tree.insert_node(gnode->get_info());  
GT::map_nodes(gnode, tnode);`

Usamos el bit *Scaling\_Tree* para pintar un nodo o arco que haya sido visitado.

Con lo anterior en mente podemos plantear el recorrido en profundidad a partir de *gnode*:

603c *(Recorrer en profundidad nodos adyacentes a gnode 603c)≡* (603a 604)  
`for (Node_Arc_Iterator<GT, SA> i(gnode); i.has_current(); i.next())  
{  
 typename GT::Arc * arc = i.get_current_arc();  
 if (IS_ARC_VISITED(arc, Scaling_Tree))  
 continue;  
 typename GT::Node * arc_tgt_node = i.get_tgt_node();`

```

if (IS_NODE_VISITED(arc_tgt_node, Spanning_Tree))
 continue; // destino ya visitado desde otro arco

if (_find_depth_first_spanning_tree<GT,SA>(g, arc_tgt_node,
 arc, tree, tnode))
 return false; // ya el árbol está calculado
}

```

La estructura es similar a la de otros algoritmos basados en el recorrido en profundidad. La rutina recursiva es `_find_depth_first_spanning_tree()`, la cual acepta los siguientes parámetros en el orden presentado:

1. g: el grafo a recorrer.
2. arc\_tgt\_node: un nodo dentro de g que aún no ha sido visitado y que no tiene imagen en tree.
3. garc: el arco actual de g cuyo nodo origen es gnode. Notemos que el arco aún no tiene imagen en tree; éste será insertado al comienzo de `_find_depth_first_spanning_tree()`.
4. gnode: un nodo de g, ya visitado, que tiene imagen en tree.
5. tree: el árbol abarcador que se está construyendo.
6. tnode: la imagen de gnode en tree. Notemos que en este caso funge de nodo destino.

La rutina recursiva resultante tiene entonces la siguiente implantación:

604      *(Árboles abarcadores 603a) +≡*                                          <603a 605>

```

template <class GT, class SA> inline static
bool _find_depth_first_spanning_tree(GT & g, typename GT::Node * gnode,
 typename GT::Arc * garc,
 GT & tree, typename GT::Node * tnode)
{
 NODE_BITS(gnode).set_bit(Spanning_Tree, true); // marcar nodo
 ARC_BITS(garc).set_bit(Spanning_Tree, true); // marcar arco

 typename GT::Node * tree_tgt_node = tree.insert_node(gnode->get_info());
 GT::map_nodes(gnode, tree_tgt_node);

 typename GT::Arc * tarc =
 tree.insert_arc(tnode, tree_tgt_node, garc->get_info());
 GT::map_arcs(garc, tarc);

 tnode = tree_tgt_node;
 if (tree.get_num_nodes() == g.get_num_nodes()) // ¿grafo abarcado?
 return true; // tree ya contiene el árbol abarcador

 <Recorrer en profundidad nodos adyacentes a gnode 603c>

 return false;
}

```

De este algoritmo es fundamental entender su condición de parada, cual se alcanza cuando se han abarcado todos los nodos de  $g$ , es decir, cuando la cantidad de nodos de  $tree$  sea la misma que la de  $g$ . También es importante notar que al evitar la añadidura de un arco o nodo ya visitado en  $tree$ , es imposible causar un ciclo, razón que garantiza que  $tree$  sea, en efecto, un árbol.

### 7.5.10 Árboles abarcadores de amplitud

En el mismo sentido que para un árbol abarcador en profundidad podemos plantear una rutina que nos extraiga del grafo el árbol abarcador correspondiente al recorrido en amplitud desde un nodo dado. Tal rutina se efectúa de la siguiente manera:

605

```
Árboles abarcadores 603a>+≡ ◁604 607▷
template <class GT, class SA = Default_Show_Arc<GT> > inline
void find_breadth_first_spanning_tree(GT & g, typename GT::Node * gp,
 GT & tree)
{
 g.reset_bit_nodes(Scaling_Tree);
 g.reset_bit_arcs(Scaling_Tree);
 clear_graph(tree);
 unique_ptr<typename GT::Node> tp_auto(new typename GT::Node(gp));
 tree.insert_node(tp_auto.get());
 GT::map_nodes(gp, tp_auto.release());

 DynListQueue<typename GT::Arc*> q; // insertar en cola arcos gp
 for (Node_Arc_Iterator<GT, SA> i(gp); i.has_current(); i.next())
 q.put(i.get_current_arc());

 NODE_BITS(gp).set_bit(Scaling_Tree, true);

 while (not q.is_empty())
 {
 typename GT::Arc * garc = q.get();
 ARC_BITS(garc).set_bit(Scaling_Tree, true);
 typename GT::Node * gsrc = g.get_src_node(garc);
 typename GT::Node * gtgt = g.get_tgt_node(garc);

 if (IS_NODE_VISITED(gsrc, Scaling_Tree) and
 IS_NODE_VISITED(gtgt, Scaling_Tree))
 continue; // los dos nodos de garc ya fueron visitados

 if (IS_NODE_VISITED(gtgt, Scaling_Tree)) // ¿gtgt visitado?
 std::swap(gsrc, gtgt); // sí, intercámboleo con gsrc

 typename GT::Node * tsrc = mapped_node<GT>(gsrc);
 NODE_BITS(gtgt).set_bit(Scaling_Tree, true); // gtgt visitado

 // crear copia de gtgt, insertarlo en tree y mapearlo
 unique_ptr<typename GT::Node> ttgt_auto(new typename GT::Node(gtgt));
 tree.insert_node(ttgt_auto.get());
 typename GT::Node * ttgt = ttgt_auto.release();
```

```

GT::map_nodes(gtgt, ttgt);

 // insertar nuevo arco en tree y mapearlo
 typename GT::Arc * tarc = tree.insert_arc(tsrc, ttgt, garc->get_info());
 GT::map_arcs(garc, tarc);
 if (tree.get_num_nodes() == g.get_num_nodes()) // ¿abarca a g?
 break;

 // insertar en cola arcos de gtgt
 for (Node_Arc_Iterator<GT, SA> i(gtgt); i.has_current(); i.next())
 {
 typename GT::Arc * current_arc = i.get_current_arc();
 if (IS_ARC_VISITED(current_arc, Spanning_Tree))
 continue;

 // revise nodos de arcos para ver si han sido visitados
 if (IS_NODE_VISITED(g.get_src_node(current_arc), Spanning_Tree) and
 IS_NODE_VISITED(g.get_tgt_node(current_arc), Spanning_Tree))
 continue; // nodos ya visitados ==> no meter arco
 q.put(current_arc);
 }
}
}

Uses mapped_node 560b.

```

Puesto que de la cola se obtienen arcos de los cuales un solo nodo está contenido en tree y, además, tree es permanentemente conexo, entonces es imposible que la inclusión de un nuevo arco cause un ciclo. tree es, por tanto, un árbol abarcador.

### 7.5.11 Árboles abarcadores en arreglos

Una manera muy simple de representar un árbol, que no explicamos en el capítulo 4, consiste en mantener un arreglo de padres a cada nodo del árbol. Cada entrada  $i$  del arreglo contiene el parent del nodo cuyo índice o número es  $i$ . La idea es pictorizada mediante los ejemplos de la figura 7.24.

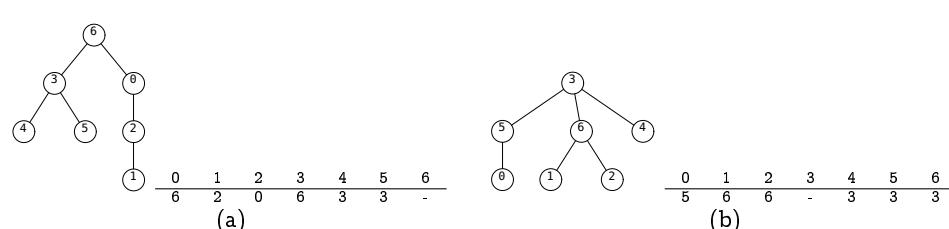


Figura 7.24: Dos ejemplos de árboles representados con arreglos

Para el caso de un árbol abarcador de un grafo, y en general para árboles de cualquier índole, la representación pictorizada asume que los nodos están enumerados, lo cual nos restringe la representación. Para evitar esto empleamos un arreglo de punteros a nodos, en el cual la entrada contentiva de NULL representa la raíz.

Como el TAD List\_Graph permite representar multigrafos, el mero arreglo de padres no es suficiente para distinguir multiarcos. Esta ambigüedad puede resolverse mediante un

arreglo de arcos en lugar del de nodos, pero esto sólo es válido si el árbol abarcador refiere a un digrafo. Así que, en aras de la generalidad, emplearemos dos arreglos de punteros, uno a nodos y otro a arcos.

Si  $g$  es un grafo y de alguna manera tenemos un árbol abarcador representado en los arreglos de punteros a nodos  $\text{pred}[]$  y arcos  $\text{arcs}[]$  respectivamente, entonces podemos obtener un grafo tree contentivo del árbol abarcador mediante la siguiente rutina:

607

```
Árboles abarcadores 603a +≡ ▷605
template <class GT>
void build_spanning_tree(GT & g, GT & tree, DynArray<typename GT::Node> * pred,
 DynArray<typename GT::Arc> * arcs,
 const bool with_map = false)
{
 tree.clear();
 for (int i = 0; i < g.get_num_nodes(); ++i)
 {
 typename GT::Arc * garc = arcs[i];
 if (garc == NULL)
 continue;

 typename GT::Node * gsrc = g.get_src_node(garc);
 typename GT::Node * tsr = NULL;
 if (IS_NODE_VISITED(gsrc, Spanning_Tree))
 tsr = mapped_node <GT> (gsrc);
 else
 {
 NODE_BITS(gsrc).set_bit(Spanning_Tree, true);
 tsr = tree.insert_node(gsrc->get_info());
 }

 typename GT::Node * gtgt = g.get_tgt_node(garc);
 typename GT::Node * ttgt = NULL;
 if (IS_NODE_VISITED(gtgt, Spanning_Tree))
 ttgt = mapped_node <GT> (gtgt);
 else
 {
 NODE_BITS(gtgt).set_bit(Spanning_Tree, true);
 ttgt = tree.insert_node(gtgt->get_info());
 }

 typename GT::Arc * tarc = tree.insert_arc(tsr, ttgt, garc->get_info());
 ARC_BITS(garc).set_bit(Min, true);
 if (with_map)
 {
 GT::map_nodes(gsrc, tsr);
 GT::map_nodes(gtgt, ttgt);
 GT::map_arcs(garc, tarc);
 }
 }
}
```

Uses `DynArray` 34 and `mapped_node` 560b.

El parámetro `with_map` indica si el árbol abarcador debe o no mapearse con el grafo `g`.

A parte de la economía de espacio, representar árboles abarcadores de grafos mediante arreglos es particularmente útil para algoritmos de búsqueda de caminos mínimos, específicamente los algoritmos de Floyd y Bellman-Ford, los cuales estudiaremos en § 7.9.2 (Pág. 685) y § 7.9.3 (Pág. 692) respectivamente.

### 7.5.12 Conversión de un árbol abarcador a un Tree\_Node

En § 4.5 (Pág. 264) desarrollamos el TAD `Tree_Node`, el cual modeliza un árbol `m-rio`, mientras que en las subsecciones precedentes acabamos de desarrollar algoritmos para

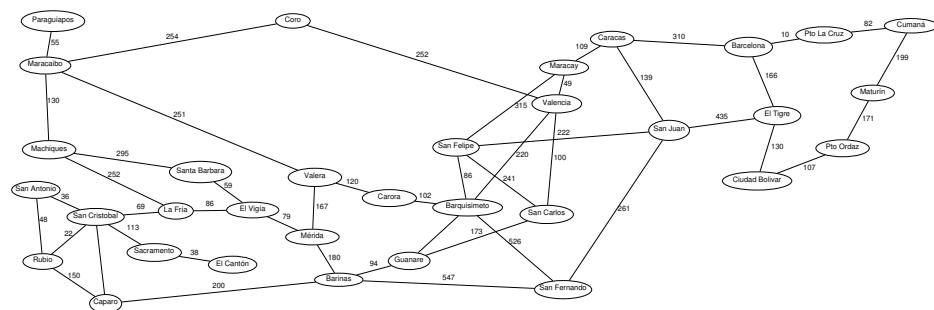


Figura 7.25: Un grafo vial de ciudades y distancias

calcular árboles abarcadores correspondiente a los recorridos en profundidad y amplitud de un grafo. En esta subsección desarrollaremos un algoritmo que nos convierte un árbol abarcador de tipo `List_Graph` a un árbol de tipo `Tree_Node`. Aparte de su valor didáctico, esta rutina nos permitirá apelar al dibujado automático de árboles `m-rios` y así esquematizar, a lo largo de este texto, diferentes árboles que se descubren en muchos de los algoritmos que existen sobre grafos.

Nuestro algoritmo de conversión reside en el archivo `graph_to_tree.H`.

La conversión se invoca a través de la siguiente interfaz:

608 `<Conversión de List_Graph a Tree_Node 608>` 609a▷

```

template <class GT, typename Key, class Convert> static
Tree_Node<Key> * graph_to_tree_node(GT & g, typename GT::Node * groot);

```

La cual retorna un árbol de tipo `Tree_Node<Key>*` correspondiente al árbol abarcador `g` con raíz en `groot`<sup>14</sup>.

`graph_to_tree_node()` tiene tres parámetros tipo:

1. `GT`: El tipo de `List_Graph`, que contiene un árbol abarcador y que se desea convertir a `Tree_Node`.
2. `Key`: La clave que se almacenará en el `Tree_Node`.
3. `Convert`: Una clase de transformación desde `typename GT::Node*` hacia `Tree_Node<Key>*`.

En el operador `()` de esta clase se delega la conversión a través de la siguiente llamada:

<sup>14</sup>Los árboles dibujados en este texto son procesados con una rutina llamada `generate_tree()` residente en el archivo `generate_tree.H`, la cual genera un árbol de entrada al programa de dibujado `nt`.

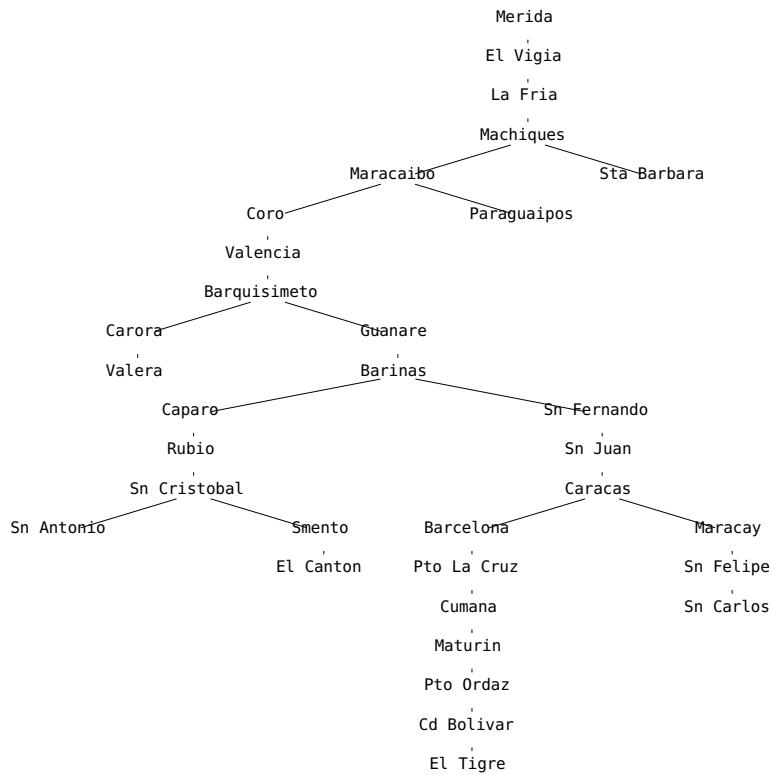


Figura 7.26: Árbol abarcador en profundidad del grafo de la figura § 7.25 (Pág. 608) con nodo de inicio “Mérida”

```
void operator () (typename GT::Node * groot, Tree_Node<Key>* troot)
```

Cada vez que se crea y se mapea un nodo de tipo groot de tipo typename GT::Node \*, se invoca a:

```
Convert () (groot, troot);
```

La cual extrae alguna información de groot, la convierte al tipo Key y asigna este resultado a troot->get\_key().

#### 4. SA: el filtro de arcos.

graph\_to\_tree\_node() se apoya sobre la siguiente rutina privada:

609a ⟨Conversión de List\_Graph a Tree\_Node 608⟩+≡ 609b ⟣  
 template <class GT, typename Key, class Convert> static void  
 \_\_graph\_to\_tree\_node(GT & g, typename GT::Node \* groot,  
 Tree\_Node<Key> \* troot);

la cual se encarga de recorrer recursivamente el árbol abarcadorg a partir del nodo groot y mantiene en troot el equivalente de tipo Tree\_Node<Key>.

Mediante \_\_graph\_to\_tree\_node() podemos implantar la rutina principal de la siguiente manera:

609b ⟨Conversión de List\_Graph a Tree\_Node 608⟩+≡ 609a 610a ⟣  
 template <class GT, typename Key,  
 class Convert, class SA = Default\_Show\_Arc<GT> > inline  
 Tree\_Node<Key> \* graph\_to\_tree\_node(GT & g, typename GT::Node \* groot)

```

{
 Tree_Node<Key> * troot = new Tree_Node<Key>; // apartar memoria raíz

 Convert () (groot, troot); //convertir de groot y copiar a troot

 __graph_to_tree_node <GT, Key, Convert, SA> (g, groot, troot);

 return troot;
}

```

`graph_to_tree()` inicializa la raíz `Tree_Node<Key>`. El resto del trabajo lo efectúa la exploración recursiva de `groot` implantada por `__graph_to_tree()`:

610a *(Conversión de List\_Graph a Tree\_Node 608)≡* 609b

```

template <class GT, typename Key, typename Convert, class SA> static inline
void __graph_to_tree_node(GT & g, typename GT::Node * groot,
 Tree_Node<Key> * troot)
{
 typedef typename GT::Node Node;
 typedef typename GT::Arc Arc;

 // recorrer arcos de groot y construir recursivamente
 for (Node_Arc_Iterator<GT, SA> it(groot); it.has_current(); it.next())
 {
 Arc * arc = it.get_current_arc();
 if (IS_ARC_VISITED(arc, Convert_Tree))
 continue;

 ARC_BITS(arc).set_bit(Convert_Tree, true); // arc visitado
 Node * gtgt = it.get_tgt_node();
 Tree_Node<Key> * ttgt = new Tree_Node<Key>;

 Convert () (gtgt, ttgt); // asignarle la clave

 troot->insert_rightmost_child(ttgt); // insertarlo como hijo

 __graph_to_tree_node <GT, Key, Convert, SA> (g, gtgt, ttgt);
 }
}

```

Es suficiente con sólo marcar los arcos que ya se han visitado, pues, como `g` es un árbol, no hay posibilidad de visitar al nodo desde otro arco. El bit de marca se denomina `Convert_Tree`.

### 7.5.13 Componentes inconexos de un grafo

Dado un grafo, se desea determinar la cantidad de subgrafos inconexos que lo conforman. A tales efectos, consideremos la siguiente primitiva:

610b *(Componentes inconexos 610b)≡* 611a▷

```

template <class GT, class SA = Default_Show_Arc<GT>> inline
void inconnected_components(GT & g, DynDlist<GT> & list);

```

Uses `DynDlist 85a` and `inconnected_components 611b`.

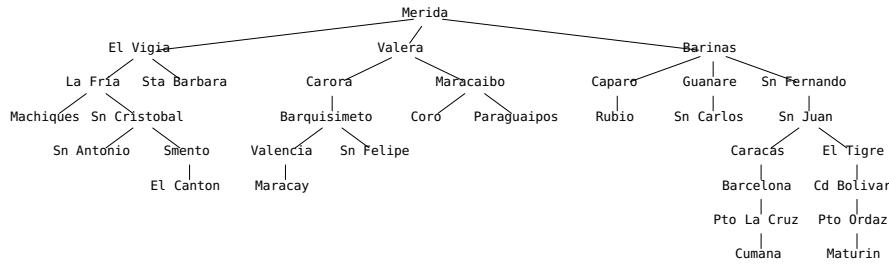


Figura 7.27: Árbol abarcador en amplitud del grafo de la figura § 7.25 (Pág. 608) con nodo de inicio “Mérida”

`inconnected_components()` recibe un grafo  $g$  y una lista de grafos vacía  $list$ . Luego de la ejecución de la primitiva,  $list$  contiene los subgrafos mapeos de  $g$  correspondientes a sus componentes inconexos. Si  $g$  es conexo, entonces  $list$  contiene un solo grafo.

Hay varias maneras de aprovechar el recorrido en profundidad para resolver este problema, algunas más eficientes y otras más simples de implantar. En todos los casos, la idea básica es recorrer los nodos del grafo según el nodo examinado se haya visitado o no.

Un algoritmo simple consiste en efectuar dos pasadas. La primera recorre los arcos por profundidad y pinta los nodos según la conectividad. Al final de esta pasada, los nodos y arcos quedan pintados según el color del componente inconexo desde el cual se inició su recorrido en profundidad. Durante esta pasada se insertan los nodos, mas no los arcos, en los subgrafos resultados.

Posteriormente, en una segunda pasada, se recorren los arcos y éstos, según su color, se insertan en su correspondiente subgrafo resultado. La primera pasada del algoritmo toma  $\mathcal{O}(E)$ , mientras que la segunda  $\mathcal{O}(E)$ . Por tanto, el algoritmo es  $\mathcal{O}(E)$ .

Se puede lograr un algoritmo más eficiente que ahorre la segunda pasada si se construye directamente el subgrafo durante el recorrido en profundidad en lugar de postergarlo para la segunda pasada. Este es el enfoque que emplearemos.

Consideremos la primitiva siguiente:

```
611a <Componentes inconexos 610b>+≡ ◁610b 611b▶
 template <class GT, class SA = Default_Show_Arc<GT>> inline
 void build_subgraph(GT & g, GT & sg,
 typename GT::Node * g_src, size_t & node_count);
```

`build_subgraph()` recorre en profundidad el grafo  $g$  y construye un componente  $sg$  según la conectividad del nodo que se esté visitando.  $g\_src$  es el “nodo actual” de  $g$  que se intenta visitar.  $node\_count$  es un contador sobre el número de nodos visitados que permite detectar cuando se ha inspeccionado todo el grafo.

`build_subgraph()` es invocado por `inconnected_components()`, el cual recorre todos los nodos del grafo  $g$  y, para cada nodo  $curr\_node$  que no haya sido visitado, hace una llamada a `build_subgraph()`, la cual construirá el componente inconexo correspondiente al recorrido en profundidad desde  $curr\_node$ . Para poder mapear todo el componente inconexo, la detención debe hacerse cuando se hayan recorrido todos los arcos.

Bajo las reciente premisas podemos enunciar el algoritmo definitivo:

```
611b <Componentes inconexos 610b>+≡ ◁611a 612▶
 template <class GT, class SA = Default_Show_Arc<GT>> inline
 void inconnected_components(GT & g, DynDlist <GT> & list)
 {
```

```

g.reset_nodes();
g.reset_arcs();
size_t count = 0; // contador de nodos visitados
for (typename GT::Node_Iterator i(g); // recorrer nodos de g
 count < g.get_num_nodes() and i.has_current(); i.next())
{
 typename GT::Node * curr = i.get_current_node();
 if (IS_NODE_VISITED(curr, Build_Subtree))
 continue;

 // crear subgrafo componente inconexo conectado por curr_node
 list.append(GT()); // crea subgrafo y lo inserta en lista
 GT & subgraph = list.get_last(); // grafo insertado en list

 build_subgraph <GT, SA> (g, subgraph, curr, count);
}
}

Defines:
 interconnected_components, used in chunk 610b.
Uses DynDlist 85a.

```

Ahora podemos implantar `build_subgraph()`, la cual se remite a explorar en profundidad desde un nodo `g_src` a la búsqueda de todos los arcos y nodos asequibles desde `g_src`, y que conformarán un componente conexo del grafo:

612   *<Componentes inconexos 610b>+≡*                                                                   $\triangleleft 611b$

```

template <class GT, class SA = Default_Show_Arc<GT> > inline
void build_subgraph(GT & g, GT & sg,
 typename GT::Node * g_src, size_t & node_count)
{
 if (IS_NODE_VISITED(g_src, Build_Subtree))
 return;

 NODE_BITS(g_src).set_bit(Build_Subtree, true); // g_src visitado
 ++node_count;

 typename GT::Node * sg_src = mapped_node <GT> (g_src);
 if (sg_src == NULL) // ¿está mapeado g_src?
 {
 // No, cree imagen de g_src en el subgrafo sg y mapee
 sg_src = sg.insert_node(g_src->get_info());
 GT::map_nodes(g_src, sg_src);
 }

 for (Node_Arc_Iterator<GT, SA> i(g_src); // explore desde g_src
 node_count < g.get_num_nodes() and i.has_current(); i.next())
 {
 typename GT::Arc * arc = i.get_current_arc();
 if (IS_ARC_VISITED(arc, Build_Subtree))
 continue; // avance próximo arco

 ARC_BITS(arc).set_bit(Build_Subtree, true); // arc visitado
 typename GT::Node * g_tgt = i.get_tgt_node(); // destino de arc
 typename GT::Node * sg_tgt = mapped_node <GT> (g_tgt);
 }
}

```

```

if (sg_tgt == NULL) // ¿está mapeado en sg?
{
 // no, hay que mapearlo e insertarlo en el subgrafo sg
 sg_tgt = sg.insert_node(g_tgt->get_info());
 GT::map_nodes(g_tgt, sg_tgt);
}
// tenemos nodos en subgrafo, insertamos arco y mapeamos
typename GT::Arc * sg_arc = sg.insert_arc(sg_src, sg_tgt, arc->get_info());
GT::map_arcs(arc, sg_arc);

build_subgraph<GT,SA>(g, sg, g_tgt, node_count);
}
}

Uses mapped_node 560b.

```

#### 7.5.14 Puntos de articulación de un grafo

Dado un grafo conexo, un **punto de articulación**, o **de corte**, es un nodo tal que al eliminarse divide al grafo en al menos dos subgrafos disjuntos. La figura 7.28 muestra un ejemplo en el cual sus puntos de articulación están resaltados.

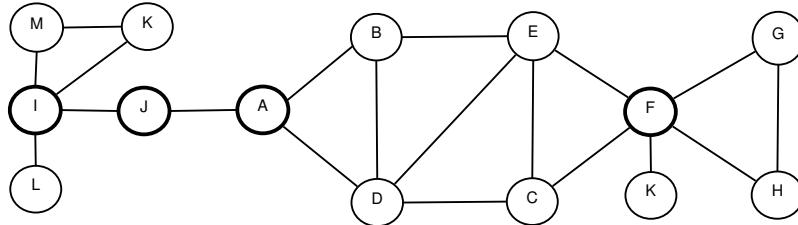


Figura 7.28: Un grafo y sus puntos de articulación

Los subgrafos disjuntos resultantes de la eliminación de los puntos de corte de un grafo se denominan “**bloques**” o “**componentes conexos**”.

A un grafo sin puntos de articulación se le califica de “**grafo biconexo**”. Los componentes conexos de un grafo son biconexos.

La determinación de los puntos de articulación es fundamental para la prueba de planaridad y para algunos algoritmos de dibujado, a la vez que constituye otra aplicación más de la búsqueda en profundidad. También revela en cierto modo los “**puntos críticos**” de un grafo. Por ejemplo, en una red eléctrica, los puntos de articulación indican los puntos que al fallar pueden comprometer la red; lo mismo aplica a la mayoría de clases de redes: de transporte, de paquetes de red, etcétera.

En esta subsección presentaremos un algoritmo lineal basado en la búsqueda en profundidad, el cual permite encontrar eficientemente los puntos de corte de un grafo.

**Definición 7.1 (Número df)** : Sea un grafo conexo  $G = \langle V, E \rangle$  sobre el cual se efectúa un recorrido en profundidad. Sea  $v \in V$  un nodo. Entonces,  $df(v)$  se define como el ordinal de visita de un recorrido en profundidad. En otras palabras,  $df(v)$  denota el orden de visita en un recorrido en profundidad.

El valor de  $df(v)$  para cualquier nodo es directamente calculable por el recorrido en profundidad prefijo.

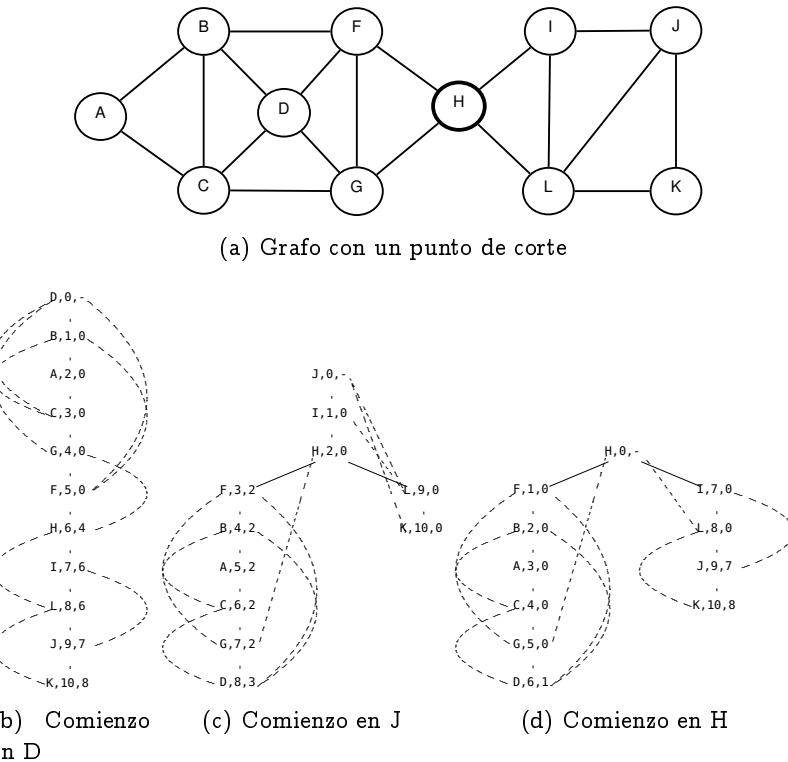


Figura 7.29: Un grafo con puntos de corte y sus árboles abarcadores con sus números df, low y arcos no-abarcadores punteados

Cuando se efectúa un recorrido en profundidad, los arcos por los cuales se descubren nuevos nodos son llamados “arcos abarcadores”, pues éstos son parte del árbol abarcador que caracteriza el recorrido. Análogamente, el resto de los arcos, es decir, aquellos que no son parte del árbol abarcador, son tildados de “no-abarcadores”.

**Definición 7.2 (Número low)** : Sea  $G = \langle V, E \rangle$  un grafo conexo sobre el cual se hace un recorrido en profundidad. Sea  $v \in V$  un nodo. Entonces:

$$\text{low}(v) = \begin{cases} \text{df}(v) \\ \text{mínimo de } \begin{cases} \text{df}(x) & | x \text{ nodo conectado a } v \text{ por arco no-abarcador} \\ \text{low}(w) & | w \text{ un hijo de } v \end{cases} \end{cases} \quad (7.1)$$

El valor  $\text{low}(v)$  puede calcularse progresivamente durante un recorrido en profundidad.

Para un nodo  $p$  cualquiera usaremos el contador de cada nodo para almacenar  $\text{df}(p)$ , y el puntero cookie para el de  $\text{low}(p)$ :

614a

$\langle \text{Puntos de corte } 614a \rangle \equiv$

```
template <class GT> inline static long & df(typename GT::Node * p)
template <class GT> inline static long & low(typename GT::Node * p)
```

Esta rutinas retornan los valores de  $\text{df}(p)$  y  $\text{low}(p)$ , respectivamente. Estos valores son inicializados al principio de la rutina que nos calcule los puntos de corte:

614b

$\langle \text{Inicializar nodos y arcos } 614b \rangle \equiv$

```
Operate_On_Nodes <GT, Init_Low <GT> > () (g);
g.reset_arcs();
```

615a▷

(616a)

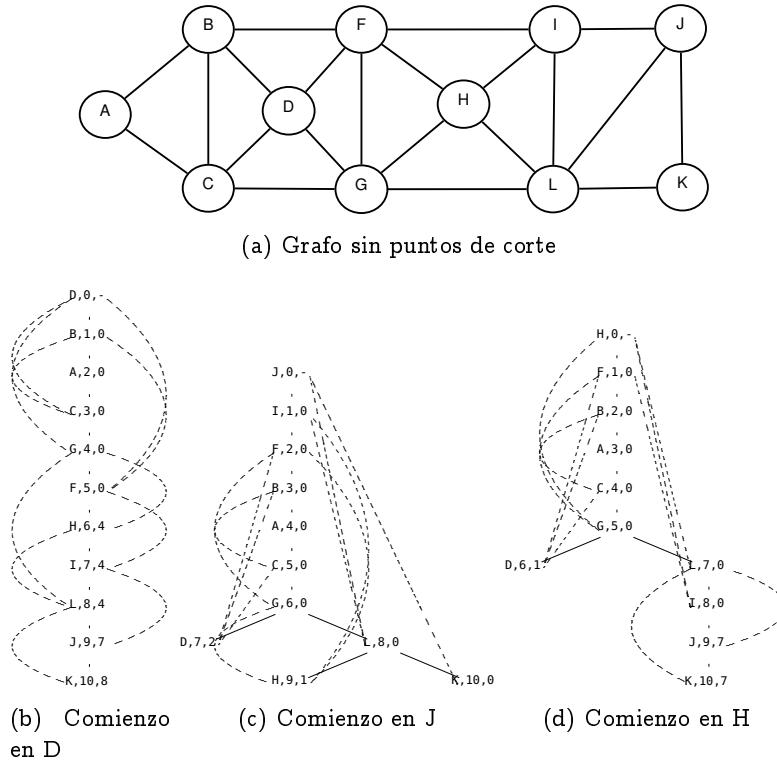


Figura 7.30: Un grafo sin puntos de corte y sus árboles abarcadores con sus números df, low y arcos no-abarcadores punteados

Donde la inicialización de un nodo está instrumentada por la clase `Init_Low`, cuya especificación es como sigue:

615a  $\langle Puntos\ de\ corte\ 614a \rangle + \equiv$  614a 615b ▷  
`template <class GT> struct Init_Low`  
`{`  
 `void operator () (GT & g, typename GT::Node * node)`  
 `{`  
 `g.reset_counter(node); // inicializa df`  
 `low <GT> (node) = -1; // inicializa low`  
 `}`  
`};`

La interfaz de cálculo de los puntos de corte se denomina `compute_cut_nodes()`; sus parámetros son el grafo, el nodo inicio de búsqueda y una lista sobre la cual se almacenan sus puntos de articulación:

615b  $\langle Puntos\ de\ corte\ 614a \rangle + \equiv$  615a 623 ▷  
`template <class GT, class SA = Default_Show_Arc<GT> >`  
`void compute_cut_nodes(GT & g, typename GT::Node * start,`  
 `DynList<typename GT::Node *> & list)`  
`{`  
 `(Inicializar detección de nodos de corte 616a)`  
 `(Explorar recursivamente las ramas de start 616b)`  
 `(Verificar si start es un nodo de corte 617a)`  
`}`

Uses `DynList` 85a.

#### 7.5.14.1 Detección de los puntos de corte

El algoritmo de cálculo de puntos de corte efectúa un recorrido en profundidad, calcula progresivamente los valores  $df(v)$  y  $low(v)$  para cada nodo  $v$  y determina si  $v$  es o no un punto de corte según:

1. Si  $v$  es raíz y tiene dos o más ramas, entonces  $v$  es un nodo de corte.

Esto será demostrado en la proposición 7.1 (Pág 620).

2.  $v$  no es raíz y  $v$  tiene un hijo  $w$  tal que  $low(w) \geq df(v)$ , entonces  $v$  es un punto de corte

Esto será demostrado en la proposición 7.2 (Pág 620).

En el ínterin puede verificarse la aplicabilidad examinando las figuras 7.29 y 7.30.

#### 7.5.14.2 Desarrollo del algoritmo

`compute_cut_nodes` tiene tres fases cuyas funciones están claramente enunciadas. Aparte de reiniciar los arcos y nodos requerimos el contador global de visitas que determinará el  $df$  de cada nodo, el nodo inicial por donde comenzar a buscar y un contador de llamadas a la exploración recursiva, la cual será implantada por `_compute_cut_nodes()`:

616a *(Iniciarizar detección de nodos de corte 616a)≡* (615b)  
*(Iniciarizar nodos y arcos 614b)*  
`long current_df = 0; // contador global de visitas  
NODE_BITS(start).set_bit(Depth_First, true); // marcar start  
df <GT> (start) = current_df++;`  
`int call_counter = 0; // contador llamadas recursivas`

`call_counter` cuenta la cantidad de llamadas efectuadas a `_compute_cut_node()`, lo cual permitirá luego, según el conocimiento dado en § 7.5.14.1 (Pág. 616), determinar si `start` es o no un nodo de corte.

La siguiente fase es la exploración recursiva a todos los nodos conectados por `start`:

616b *(Explorar recursivamente las ramas de start 616b)≡* (615b)  
`// Recorra los arcos de start mientras g no haya sido abarcado  
for (Node_Arc_Iterator<GT, SA> i(start);  
 i.has_current() and current_df < g.get_num_nodes(); i.next())`  
`{  
 typename GT::Node * tgt = i.get_tgt_node();  
 if (IS_NODE_VISITED(tgt, Depth_First))  
 continue;  
  
 typename GT::Arc * arc = i.get_current_arc();  
 if (IS_ARC_VISITED(arc, Depth_First))  
 continue;  
  
 ARC_BITS(arc).set_bit(Depth_First, true);  
 _compute_cut_nodes <GT, SA> (g, list, tgt, arc, current_df);  
 ++call_counter;  
}`

Según el corolario, start es un nodo de corte si el árbol abarcador de profundidad con raíz start tiene más de un hijo. En los términos de este algoritmo, esto está determinado por la cantidad de veces que se explore recursivamente un nodo conectado a start; dicho de otro modo, la cantidad de veces que se llame a `__compute_cut_nodes()`. Si se llama una sola vez, start no es un nodo de corte. De lo contrario, sin importar el valor exacto de `call_counter`, start es de corte. Lo anterior se expresa entonces del siguiente modo:

617a *(Verificar si start es un nodo de corte 617a)≡* (615b)  
`if (call_counter > 1) // ¿es la raíz un punto de articulación?  
{  
 NODE_BITS(start).set_bit(Cut, true);  
 list.append(start);  
}`

Cada vez que se detecte un nodo de corte, éste se marcará con el bit Cut, el cual nos servirá luego para marcar los componentes conexos asociados los puntos de corte.

La interfaz de la exploración recursiva, `__compute_cut_nodes()`, se expresa así:

617b *(Definición del recorrido recursivo para los puntos de corte 617b)≡*  
`template <class GT, class SA> inline static  
void __compute_cut_nodes(GT & g, DynDlist<typename GT::Node *> & list,  
 typename GT::Node * p, typename GT::Arc * a,  
 long & curr_df);`

Uses `DynDlist` 85a.

cuyos parámetros se definen como sigue:

1. g: el grafo.
2. list: la lista de nodos de corte. `__compute_cut_nodes()` añade cada punto de corte encontrado a esta lista.
3. p: el nodo actualmente siendo visitado.
4. a: el arco desde el cual se alcanza a p. Este es, en el árbol abarcador de profundidad, el arco que conecta a su nodo padre.
5. curr\_df: el valor actual del contador de visitas que será asignado al nodo siendo visitado como su valor df.

La implantación de `__compute_cut_nodes()` añade la recursión, el cálculo de los valores low de cada nodo y la distinción de que el nodo visitado sea o no una hoja dentro del eventual árbol abarcador de profundidad:

617c *(Implantación del recorrido recursivo para los puntos de corte 617c)≡*  
`template <class GT, class SA> inline static  
void __compute_cut_nodes(GT & g, DynDlist<typename GT::Node *> & list,  
 typename GT::Node * p, typename GT::Arc * a,  
 long & curr_df)`  
`{  
 NODE_BITS(p).set_bit(Depth_First, true); // pinte p visitado  
 low <GT> (p) = df <GT> (p) = curr_df++; // asignele df  
 // recorrer arcos de p mientras no se abarque a g`

```

bool p_is_cut_node = false;
for (Node_Arc_Iterator <GT, SA> i(p); i.has_current(); i.next())
{
 typename GT::Arc * arc = i.get_current_arc();
 if (arc == a)
 continue; // a es el padre de arc ==> ignorarlo

 typename GT::Node * tgt = i.get_tgt_node();
 if (IS_NODE_VISITED(tgt, Depth_First))
 {
 if (not IS_ARC_VISITED(arc, Depth_First)) // no abarcador?
 if (df<GT>(tgt) < low<GT>(p)) // sí, verificar valor low
 low<GT>(p) = df<GT>(tgt); // actualizar low(p)
 continue;
 }
 if (IS_ARC_VISITED(arc, Depth_First))
 continue;

 ARC_BITS(arc).set_bit(Depth_First, true); // marque arco

 __compute_cut_nodes<GT, SA>(g, list, tgt, arc, curr_df);

 if (low<GT>(tgt) < low<GT>(p))
 low<GT>(p) = low<GT>(tgt); // actualizar low(p)

 if (low<GT>(tgt) >= df<GT>(p) and df<GT>(tgt) != 0) // ¿de corte?
 p_is_cut_node = true;
}
// aquí, p ya fue explorado recursivamente
if (p_is_cut_node)
{
 NODE_BITS(p).set_bit(Cut, true);
 list.append(p);
}
}

Uses DynDlist 85a.

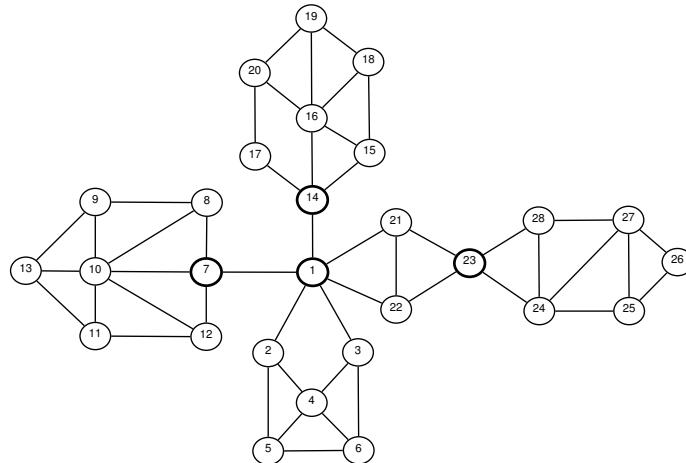
```

La rutina aprovecha el propio recorrido sobre los arcos para calcular  $\text{low}(p)$  exactamente según la fórmula (7.1). Inicialmente,  $\text{low}(p) = \text{df}(p)$ , luego, durante la inspección de los arcos de  $p$ , se determina si se trata del arco padre, de un arco no-abarcador o de un hijo. Cada mirada sobre un arco hijo o uno no-abarcador verifica si hay un valor menor para colocarle  $\text{low}(p)$ .

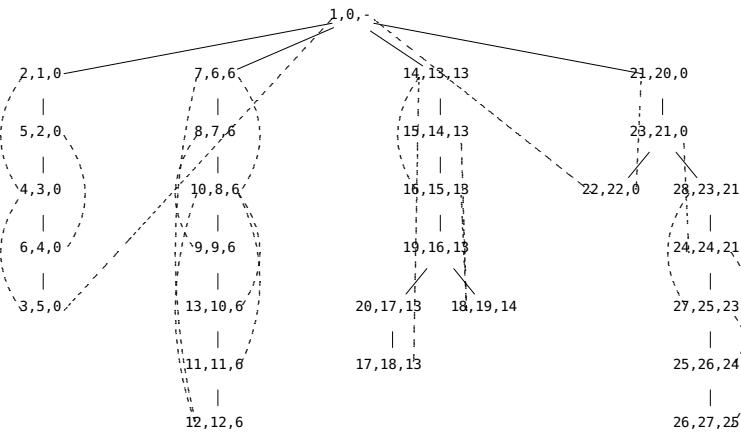
Los arcos no-abarcadores se identifican porque no son marcados con el bit `Depth_First`. El arco padre se distingue mediante el parámetro `a`. Finalmente, los arcos del árbol abarcador sí son marcados justo antes de la llamada recursiva.

Puesto que para determinar si  $p$  es un nodo de corte es necesario inspeccionar todos sus arcos (abarcadores y no-abarcadores), la detención del recorrido debe hacerse cuando se hayan mirado todos los arcos y no todos los nodos.

Después de una llamada recursiva se sabe que  $tgt$  es un hijo de  $p$ . Por esa razón se compara  $\text{low}(tgt) \geq \text{df}(p)$  para detectar si  $p$  es un nodo de corte. El algoritmo propaga el menor valor de  $\text{low}$  desde las hojas hasta sus ancestros, pues la idea es memorizar si existe un arco dentro de una descendencia que cruce o no hacia una ascendencia.



(a) Un grafo con varios puntos de corte



(b) Árbol abarcador

Figura 7.31: Un grafo y su árbol abarcador de profundidad junto con sus df y low

### 7.5.14.3 Análisis y correctitud del algoritmo

#### Análisis

Puesto que `compute_cut_nodes()` es esencialmente una búsqueda en profundidad que examina todos los arcos, el coste del algoritmo es  $\mathcal{O}(E) + \mathcal{O}(V)$ .

#### Correctitud

Nuestra prueba se orientará a demostrar que los dos criterios enunciados en § 7.5.14.1 (Pág. 616) efectivamente detectan puntos de corte.

El primer paso para analizar la correctitud es caracterizar una propiedad fundamental de un punto de corte, la cual está expresada por el siguiente lema.

**Lema 7.1** Si  $v$  es un punto de corte de un grafo  $G$  tal que al eliminar  $v$  de  $G$  este se divide en al menos dos subgrafos  $G_1$  y  $G_2$ , entonces todo camino desde cualquier nodo en  $G_1$  hacia cualquier otro nodo en  $G_2$  contiene a  $v$ .

**Demostración (Por contradicción)** Puesto que  $v$  es un punto de corte, el grafo puede verse así de manera general como en la figura 7.32.

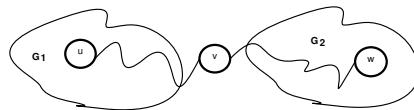


Figura 7.32: Esquema general de división de un grafo en torno a punto de corte

Si existiese un camino  $G$  de  $u$  hacia  $w$  que no pase por  $v$ , entonces  $v$  no sería un punto de corte, lo que es una contradicción  $\square$

Para el caso en el que un nodo sea raíz de un árbol abarcador de profundidad, la detección del punto de corte, tal como se expresa en 7.5.14.1-a se verifica mediante la siguiente proposición.

**Proposición 7.1** Sea  $T$  el árbol abarcador de profundidad de un grafo  $G = \langle V, E \rangle$  generado a partir de un nodo  $v$ . Si  $v$  tiene más de un hijo, entonces  $v$  es un nodo de corte.

**Demostración** Supongamos un nodo  $v$  que no es de corte y es raíz de un árbol abarcador en profundidad. Sean  $u$  y  $w$  dos hijos de  $v$  tal como en la figura 7.33.

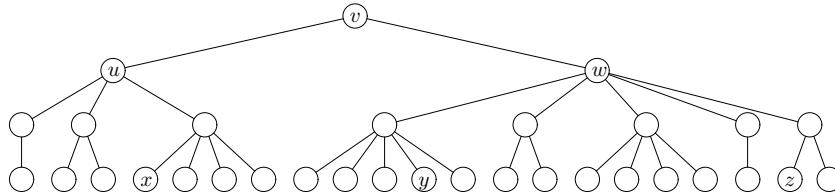


Figura 7.33: Caso genérico negado de la proposición 7.1

Recordemos que el recorrido en profundidad (`y __compute_cut_nodes()`) marca los nodos visitados en prefijo (justo en la primera línea). Así, el primer nodo visitado es  $v$ , después  $u$  y seguidamente todos los nodos que sean alcanzables desde  $u$  sin pasar por  $v$ , pues éste último está marcado como visitado. Después de haber visitado  $u$ , el recorrido regresa a  $v$  (se regresa a `compute_cut_nodes()`<sup>15</sup>) y prosigue hacia los nodos adyacentes a  $v$  que aún no han sido visitados; en nuestro caso hacia  $w$ . Por tanto, no existe ningún arco que conecte a un nodo en la rama cuya raíz es  $u$  con un nodo en la rama cuya raíz es  $w$ , pues si no hubiese sido visto desde  $u$  o desde uno de sus descendientes. Consecuentemente, todo camino  $x \rightsquigarrow y$  tal que  $x$  es igual o descendiente de  $u$  e  $y$  es igual o descendiente de  $w$  pasa por el nodo  $v$ , lo cual implica, según el lema 7.1, que  $v$  es un nodo de corte ■

Para comprobar la correctitud de la segunda condición de detección (7.1) enunciada en § 7.5.14.1 (Pág. 616) nos valdremos de la siguiente proposición.

**Proposición 7.2** Sea  $G = \langle V, E \rangle$  un grafo. Sea  $T = \langle V, E' \rangle$ ,  $E' \subset E$  un árbol abarcador de profundidad de  $G$  y  $v$  un nodo cualquiera de  $T$  distinto a la raíz. Si  $v$  tiene un hijo  $w$  tal que  $\text{low}(w) \geq \text{df}(v)$ , entonces  $v$  es un nodo de corte.

<sup>15</sup>Note que se trata de `compute_cut_nodes()` y no del recorrido recursivo `__compute_cut_nodes()`.

**Demostración (Por contradicción)** Lo primero que debemos hacer es recordar la definición de  $\text{low}(w)$  según la fórmula 7.1:

$$\text{low}(w) = \min \left\{ \begin{array}{l} \text{df}(w) \\ \text{df}(x) \mid x \text{ nodo conectado a } v \text{ por arco no-abarcador} \\ \text{low}(y) \mid y \text{ un hijo de } w \end{array} \right\} \quad (7.2)$$

Ahora debemos mirar el algoritmo, específicamente la rutina `__compute_cut_nodes()`, y estudiar cómo ésta efectivamente satisface el cálculo según (7.2).

El valor inicial de  $\text{low}(w)$  se asigna en la segunda línea de `__compute_cut_nodes()`:

```
low <GT> (p) = df <GT> (p) = curr_df++; // asigne df
```

De aquí se desprende que inicialmente  $\text{low}(w) > \text{df}(v)$ . Por tanto, de no afectarse  $\text{low}(w)$  durante el resto de la ejecución de `__compute_cut_nodes()`,  $w$  será detectado como un punto de corte.

Para el siguiente análisis es conveniente mantener en cuenta que un componente conexo no contiene puntos de corte.

En orden secuencial, las partes de `__compute_cut_nodes()` con parámetro  $w$  en las cuales se puede afectar a  $\text{low}(w)$  son:

```
1. if (IS_NODE_VISITED(tgt, Depth_First))
{
 if (not IS_ARC_VISITED(arc, Depth_First)) // no abarcador?
 if (df<GT>(tgt) < low<GT>(w)) // sí, verificar valor low
 low<GT>(w) = df<GT>(tgt); // actualizar low(p)
 // ...
}
```

Se deben reunir varias condiciones para que ocurra esta afectación. En primer lugar, los nodos  $w$  y  $tgt$  ya deben estar visitados. En segundo lugar el arco  $w \rightarrow tgt$  no ha sido visto por el recorrido, o sea, el arco  $w \rightarrow tgt$  es no abarcador. Finalmente, la afectación sólo ocurre si  $\text{df}(tgt) < \text{low}(w)$ .

Claramente existe un camino  $tgt \rightsquigarrow v \rightarrow w$ ; pero el que exista un arco directo  $w \rightarrow tgt$  y  $\text{low}(tgt) < \text{low}(w)$  implica que existe un ciclo  $tgt \rightsquigarrow v \rightarrow w \rightarrow tgt$ , pues  $\text{df}(tgt) < \text{df}(w)$ . Por tanto,  $tgt, v$  y  $w$  pertenecen al mismo componente conexo, lo cual implica que  $v$  no es un nodo de corte.

```
2. if (low<GT>(tgt) < low<GT>(w))
 low<GT>(w) = low<GT>(tgt); // actualizar low(p)
```

Esta afectación sólo puede ocurrir después de haber visitado enteramente a  $tgt$ . Para que  $\text{low}(tgt) < \text{low}(w)$  sea cierto, un descendiente de  $tgt$  debe estar conectado a un ancestro de  $v$ . La situación es genéricamente pictorizada en la figura 7.34. Llamemos  $x$  a tal descendiente. El valor  $\text{low}(x)$  es afectado durante la ejecución de la línea anterior ya discutida. Luego, los ancestros entre  $w$  y  $x$  van heredando el valor  $\text{low}(x)$  durante el regreso recursivo.

Este caso es una extensión del anterior en el cual existe un ciclo  $tgt \rightsquigarrow v \rightarrow w \rightsquigarrow x \rightarrow tgt$ . Por tanto, bajo el mismo razonamiento,  $tgt, v, w$  y  $x$  pertenecen al mismo componente conexo, lo cual implica que  $v$  no es un nodo de corte.

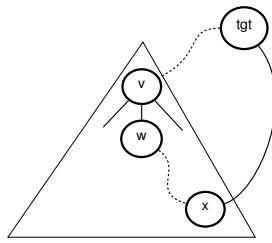


Figura 7.34: Segundo caso de modificación del valor  $\text{low}(w)$

En síntesis, inicialmente  $\text{low}(w) \geq \text{df}(v)$  y  $v$  se asume como de corte. Luego se transforma a  $\text{low}(w) < \text{df}(v)$  si  $v$  no es de corte.

Ahora neguemos la proposición y supongamos que  $v$  tiene un hijo  $w$  tal que  $\text{low}(w) \geq \text{df}(v)$  pero  $v$  no es de corte, lo cual no es posible según lo que acabamos de analizar del algoritmo. La proposición es pues cierta ■

Hasta el presente, hemos demostrado que los dos criterios de detección de puntos de corte mostrados en § 7.5.14.1 (Pág. 616) son correctos. Pero nuestra demostración no prueba que `compute_cut_nodes()` detecta todos los puntos de corte. Para hacerlo, debemos comprobar que cualquier punto de corte encaja en una de las dos condiciones. El camino lógico para hacerlo es demostrar la suficiencia de cada una de las proposiciones 7.1 y 7.2 respectivamente, lo cual es delegado en ejercicio.

### 7.5.15 Componentes conexos de los puntos de corte

Una vez calculados los puntos de corte puede ser deseable conocer cuáles son sus “componentes conexos” o, también, “bloques”, es decir, los subgrafos resultantes de suprimir los puntos de corte.

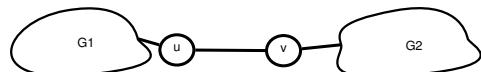
Un primer y muy simple algoritmo para determinar los componentes conexos consiste en tomar una copia mapeada del grafo y sobre éste eliminar sus nodos de corte. Los subgrafos resultantes son directamente los componentes conexos del grafo. Un problema con este algoritmo es su coste en espacio.

Puede ser interesante no sólo tener los componentes conexos sino el resto del grafo original, de modo tal que pudiéramos reconstruirlo a partir de sus componentes conexos, puntos de corte y arcos que contengan al menos un punto de corte. Es momento de introducir dos nuevas definiciones.

**Definición 7.3 (Arco de corte)** Sea un grafo  $G = \langle V, E \rangle$  y dos nodos de corte  $u, v$  tales que éstos conforman un arco  $a = (u, v) \in E$ . Entonces el arco  $(u, v)$  es denominado “de corte”.

Dicho de otro modo, un arco de corte es aquel conformado por dos nodos de corte.

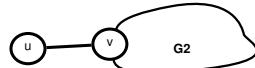
Pictóricamente, un arco de corte se generaliza del siguiente modo:



Por ejemplo, los arcos  $(1, 7)$  y  $(1, 14)$  del grafo de la figura 7.31 son de corte.

**Definición 7.4 (Arco de cruce)** Sea un grafo  $G = \langle V, E \rangle$  y un arco  $a \in E$  tal que  $a = (u, v)$  está compuesto por un nodo de corte  $u$  y por un nodo  $v$  perteneciente a algún componente conexo. Entonces, el arco  $a = (u, v)$  es llamado “de cruce”.

Pictóricamente, un arco de cruce se generaliza del siguiente modo:



Por ejemplo, los arcos  $(1, 2), (1, 21)$  y  $(1, 22)$  del grafo de la figura 7.31 son, entre otros más, de cruce.

Obtendremos los componentes conexos en dos fases. La primera se encarga de “pintar” los componentes conexos, los nodos y arcos de corte y los arcos de cruce con diferentes colores de modo tal que en la fase siguiente se les distinga. Para esta fase emplearemos la rutina `paint_subgraphs()`.

Dada una lista de nodos de corte calculada mediante `compute_cut_nodes()`, `paint_subgraphs()` pinta los componentes conexos del grafo según los cortes. Esta técnica, más compleja pero con menor consumo de espacio, consiste en “pintar” el grafo a partir de un punto de corte e ir cambiando los colores según se regrese al punto de partida o se alcance otro punto de corte. Esto presume que las marcas añadidas (el bit `Cut`) por `compute_cut_nodes()` aún siguen presentes. Llamaremos a nuestra primitiva básica `paint_subgraphs()`, la cual tiene la siguiente interfaz:

```
template <class GT, class SA>
long paint_subgraphs(GT & g,
 const DynDlist<typename GT::Node*> & cut_node_list);
```

$g$  es un grafo con sus nodos de corte calculados y guardados en la lista `cut_node_list`, mientras que `cut_arc_list` es un parámetro de salida que retorna los arcos de corte del grafo. El valor de retorno es la cantidad de colores encontrada, o sea, la cantidad de componentes conexos o bloques.

`paint_subgraphs()` asume que previamente se ejecutó sobre el grafo la rutina `compute_cut_nodes()`.

Luego de una llamada a `paint_subgraphs()`, el grafo  $g$  deviene coloreado de la siguiente forma:

1. Componentes conexos alrededor de un nodo de corte son coloreados con un color entre 1 y  $n$ , donde  $n$  es la cantidad total de componentes conexos que tiene el grafo. Por “coloreados”, insistimos, queremos decir que todos los nodos y arcos pertenecientes a un color  $i$  tienen su valor de contador en  $i$ .
2. Los nodos de corte tienen color 0.
3. Los arcos de corte tienen color 0 y el bit `Cut` en `true`.
4. Los arcos de cruce son coloreados con el color especial `Cross_Arc`. Notemos que esta clase de arco no es de corte ni pertenece a un componente conexo.

El color de un arco de cruce se define como:

623     $\langle$  Puntos de corte 614a  $\rangle + \equiv$   
 $\quad \quad \quad \text{const long Cross_Arc} = -1;$

$\triangleleft$  615b 624  $\triangleright$

Defines:

`Cross_Arc`, used in chunks 624 and 625b.

Para verificar si un arco es de cruce, emplearemos la primitiva siguiente:

```
624 <Puntos de corte 614a>+≡ ◁623 625a▷
 template <class GT> inline static bool is_a_cross_arc(typename GT::Arc * a)
 {
 return ARC_COUNTER(a) == Cross_Arc;
 }
 Uses Cross_Arc 623.
```

La segunda fase consiste en mapear el grafo previamente pintado según alguno de los colores empleados en la primera fase. Para esta fase se emplean dos primitivas:

1. 

```
template <class GT, class SA>
void map_subgraph(GT & g, GT & sg, const long & color);
```

Se utiliza para obtener los componentes conexos del grafo según sus puntos de articulación.

*g* es un grafo con sus nodos de corte previamente calculados y marcado mediante `compute_cut_nodes()`. *sg* es el grafo en donde se desea obtener una copia mapeada del componente conexo de color *color* previamente pintado mediante `paint_subgraphs()`.

Notemos que debemos hacer tantas llamadas sobre subgrafos *sg* distintos como componentes conexos tenga *g*. O sea, la cantidad de colores, cual es el valor de retorno de `paint_subgraphs()`.

2. 

```
template <class GT, class SA>
void map_cut_graph(GT & g, DynDlist<typename GT::Node*> & cut_node_list,
 GT & cut_graph,
 DynDlist<typename GT::Arc*> & cross_arc_list);
```

Obtiene el “grafo de corte”, es decir, el grafo compuesto por todos los nodos y arcos de corte.

*g* es un grafo con sus nodos de corte previamente calculados y marcado mediante `compute_cut_nodes()`. *cut\_node\_list* es la lista de nodos de corte también obtenida luego de llamar a `compute_cut_nodes()`. *cut\_graph* es el grafo donde se desea una copia mapeada del grafo de corte. Finalmente, *cross\_arc\_list* es la lista de arcos de cruce en *g*, los cuales no pertenecen ni a los componentes conexos, ni al grafo de corte.

### 7.5.15.1 Pintado de componentes conexos

Grosso modo, la técnica de pintado consiste en iniciar un recorrido en profundidad desde un nodo conectado a punto de corte. Cuando el recorrido recursivo alcance un nodo de corte, que puede ser el mismo de partida, entonces se hace un cambio (incremento) de color. Durante el recorrido de un componente, tanto los nodos como los arcos son coloreados con el color actual. Una vez que todos los arcos hayan sido recorridos sin cruzar un punto de corte, entonces el grafo estará coloreado según los componentes asociados a sus puntos de articulación.

Sabiendo que un bloque se accede desde un arco de cruce, su pintado se origina en el nodo perteneciente al arco de cruce:

625a *<Puntos de corte 614a>+≡* ◁624 625b▷

```
template <class GT, class SA> inline static
void __paint_subgraph(GT & g, typename GT::Node * p, const long & current_color)
{
 if (is_node_painted <GT> (p))
 return;

 paint_node <GT> (p, current_color);

 for (Node_Arc_Iterator<GT, SA> it(p); it.has_current(); it.next())
 {
 typename GT::Arc * arc = it.get_current_arc();
 if (is_arc_painted <GT> (arc))
 continue;

 typename GT::Node * tgt = it.get_tgt_node();
 if (is_a_cut_node <GT> (tgt))
 continue;

 paint_arc <GT> (arc, current_color);

 __paint_subgraph <GT, SA> (g, tgt, current_color);
 }
}
```

La llamada inicial a `__paint_subgraph()` se ejecuta desde un arco de cruce por la siguiente rutina que revisa un punto de corte:

625b *<Puntos de corte 614a>+≡* ◁625a 626a▷

```
template <class GT, class SA> inline static
void __paint_from_cut_node(GT & g, typename GT::Node * p, long & current_color)
{
 // pintar recursivamente con dif colores bloques conectados a p
 for (Node_Arc_Iterator<GT, SA> it(p); it.has_current(); it.next())
 {
 typename GT::Arc * arc = it.get_current_arc();
 typename GT::Node * tgt_node = it.get_tgt_node();
 if (is_a_cut_node <GT> (tgt_node)) // ¿ es un arco de corte?
 {
 ARC_BITS(arc).set_bit(Cut, true); // marque como de corte
 continue; // avance a próximo arco
 }
 else
 {
 paint_arc <GT> (arc, Cross_Arc); // marque como de cruce
 if (is_node_painted <GT> (tgt_node))
 continue;
 }
 // pintar recursivamente nodo conectado a arc
 }
}
```

```

 --paint_subgraph <GT, SA> (g, tgt_node, current_color);

 current_color++; // cambiar color (sig arco en otro bloque)
}
}

Uses Cross_Arc 623.

```

Como se ve, `_paint_from_cut_node()` toma un nodo de corte y se encarga de invocar a `_paint_subgraph()` sobre el arco actual o a pintar el arco de corte.

Finalmente no falta implementar la rutina principal:

626a *(Puntos de corte 614a) +≡* △625b 626b▷

```

template <class GT, class SA = Default_Show_Arc<GT> > inline long
paint_subgraphs(GT & g, const DynDlist<typename GT::Node*> & cut_node_list)
{
 g.reset_counter_nodes();
 g.reset_counter_arcs();
 long current_color = 1;

 // Recorrer cada nodo de corte y pintar sus bloques
 for (typename DynDlist<typename GT::Node*>::Iterator
 i(cut_node_list); i.has_current(); i.next())
 _paint_from_cut_node<GT,SA>(g, i.get_current(), current_color);

 return current_color;
}

```

Uses DynDlist 85a.

Después de calcular los puntos de corte y tenerlos en `cut_node_list`, el usuario invoca a `paint_subgraphs()` para pintar con distintos colores los bloques del grafo. Cada bloque puede distinguirse mediante su color. Los arcos de cruce se distinguen a través del color especial `Cross_Arc`. Los nodos y arcos de corte se distinguen mediante el bit `Cut`. Hasta el presente no se ha hecho una copia adicional del grafo.

### 7.5.15.2 Copia mapeada de componentes conexos

En algunas circunstancias es conveniente copiar los distintos bloques de un grafo. Por ejemplo, los algoritmos de dibujado o de detección de planaridad simplifican sus cálculos trabajando sobre los bloques. Puesto que a menudo es necesario modificar los bloques, es preferible hacerlo sobre copias que sobre el grafo original.

El proceso de identificación, copia y mapeo a un subgrafo, dado un color es simple una vez que se tienen pintados los componentes conexos: buscar y encontrar el primer nodo con el color de interés y, a partir de él, hacer una exploración en profundidad que visite, copie y mapee los nodos y arcos con el mismo color.

Sobre un nodo del grafo `gsrc`, con el color de interés “color”, ya copiado y mapeado a un subgrafo `sg`, el resto del mapeo se completa recursivamente de la siguiente manera:

626b *(Puntos de corte 614a) +≡* △626a 627▷

```

template <class GT, class SA> inline static
void _map_subgraph(GT & g, GT & sg, typename GT::Node * gsrc,
 const long & color)
{
 typename GT::Node * tsrcc = mapped_node<GT>(gsrc); // gsrc en sg

```

```

// recorrer arcos de gsrc y añadir a sg los del color de interés
for (Node_Arc_Iterator <GT, SA> i(gsrc); i.has_current(); i.next())
{
 typename GT::Arc * garc = i.get_current_arc();
 if (get_color <GT> (garc) != color or IS_ARC_VISITED(garc, Build_Subtree))
 continue; // arco es de otro color o ya está visitado

 ARC_BITS(garc).set_bit(Build_Subtree, true);

 typename GT::Node * gtgt = i.get_tgt_node();
 typename GT::Node * ttgt = NULL; // imagen gtgt en sg
 if (IS_NODE_VISITED(gtgt, Build_Subtree)) // ¡gtgt en sg?
 ttgt = mapped_node<GT> (gtgt);
 else
 { // gtgt no está en sg ==> copiarlo y mapearlo
 unique_ptr<typename GT::Node> ttgt_auto(new typename GT::Node(gtgt));
 sg.insert_node(ttgt_auto.get());
 GT::map_nodes(gtgt, ttgt_auto.get());
 NODE_BITS(gtgt).set_bit(Build_Subtree, true);
 ttgt = ttgt_auto.release();
 }
 typename GT::Arc * tarc = sg.insert_arc(tsrc, ttgt, garc->get_info());
 GT::map_arcs(garc, tarc);

 __map_subgraph<GT, SA> (g, sg, gtgt, color);
}
}

Uses mapped_node 560b.

```

`__map_subgraph()` marca los nodos y arcos visitados con el bit `Build_Subtree`; de este modo se distingue que un nodo o arco con el color de interés haya sido o no mapeado.

La rutina de interfaz `map_subgraph()` busca sobre el grafo `g` el primer nodo con color “color” y mapea el componente conexo al subgrafo `sg`. Se especifica como sigue:

627 ⟨Puntos de corte 614a⟩+≡ ◁626b 628▷

```

template <class GT, class SA = Default_Show_Arc<GT> >
void map_subgraph(GT & g, GT & sg, const long & color)
{
 clear_graph(sg);
 typename GT::Node * first = NULL; // busque primer nodo con color
 for (typename GT::Node_Iterator it(g); it.has_current(); it.next())
 if (get_color <GT> (it.get_current_node()) == color)
 first = it.get_current_node();

 // cree first, insértelo en sg y mapéelo
 unique_ptr<typename GT::Node> auto_tsrc(new typename GT::Node(first));
 sg.insert_node(auto_tsrc.get());
 GT::map_nodes(first, auto_tsrc.release());
 NODE_BITS(first).set_bit(Build_Subtree, true);

 __map_subgraph <GT, SA> (g, sg, first, color); // mapee first
}

```

Nos falta por construir el “grafo de corte” y guardar los arcos de cruce. Para eso con una pasada sobre la lista de nodos de corte creamos y mapeamos sus imágenes en el grafo de corte `cut_graph`. Después, efectuamos un barrido sobre todos los arcos. Los que están pintados los ignoramos, pues pertenecen a los componentes conexos. Los arcos de corte los copiamos e insertamos en `cut_graph`, mientras que los de cruce los insertamos en la lista `cross_arc_list`. Un detalle esencial es percibirse de que, con excepción de los componentes conexos y del grafo de corte, los arcos de cruce no son copiados.

```
628 <Puntos de corte 614a>+≡ <627
 template <class GT, class SA = Default_Show_Arc<GT> >
 void map_cut_graph(GT & g, DynDlist<typename GT::Node*> & cut_node_list,
 GT & cut_graph,
 DynDlist<typename GT::Arc*> & cross_arc_list)
 {
 clear_graph(cut_graph);
 // recorra lista de nodos de corte e insértelos en cut_graph
 for (typename DynDlist<typename GT::Node*>::Iterator
 it(cut_node_list); it.has_current(); it.next())
 {
 typename GT::Node * gp = it.get_current();
 unique_ptr<typename GT::Node> tp_auto(new typename GT::Node(gp));
 cut_graph.insert_node(tp_auto.get());
 GT::map_nodes(gp, tp_auto.release());
 }
 // recorra arcos de g ==> cut_graph = {arcos no corte} U
 // cross_arc_list = {arcos de cruce}
 for (Arc_Iterator <GT, SA> it(g); it.has_current(); it.next())
 {
 typename GT::Arc * garc = it.get_current_arc();
 if (is_a_cross_arc <GT> (garc))
 {
 cross_arc_list.append(garc);
 continue;
 }
 if (not is_an_cut_arc <GT> (garc))
 continue;

 typename GT::Node * src = mapped_node<GT>(g.get_src_node(garc));
 typename GT::Node * tgt = mapped_node<GT>(g.get_tgt_node(garc));
 typename GT::Arc * arc = cut_graph.insert_arc(src, tgt, garc->get_info());
 GT::map_arcs(garc, arc);
 }
 }
```

Uses `DynDlist 85a` and `mapped_node 560b`.

El grafo `cut_graph` es inconexo y sus componentes pueden hallarse mediante `build_subgraph()` implantada en *<Componentes inconexos 610b>* (§ 7.5.13 (Pág. 610))

## 7.6 Matrices de adyacencia

Existe un amplio corpus de la teoría de grafos, así como una extensa variedad de circunstancias de desempeño, en las cuales puede ser preferible usar la matriz de adyacencia como

representación de un grafo.

*ALÉPH* exporta los siguientes TAD vinculados a las matrices de adyacencia:

1. `Map_Matrix_Graph<GT>`: modeliza una matriz de adyacencia “mapeo” de un grafo de tipo GT basado sobre `List_Graph`. Este TAD emplea como objetos las clases `Graph_Node` y `Graph_Arc` utilizadas para `List_Graph`.

El acceso a una entrada de la matriz retorna punteros a arcos de un grafo implementado mediante una variante de `List_Graph`. Pueden especificarse punteros a nodos o sus índices en la matriz de adyacencia.

`Map_Matrix_Graph<GT>` no está destinada en sí a cálculos, simplemente constituye el primer paso para obtener una matriz de adyacencia. Sin embargo, desde este TAD es posible modificar los contenidos de los nodos y de los arcos.

2. `Matrix_Graph<GT>`: modeliza una matriz de adyacencia a partir de un `List_Graph` cuyas entradas contienen el tipo `Graph_Arc::Arc`. La idea de este TAD es obtener una copia de una grafo basado sobre `List_Graph` que pueda modificarse sin afectar el grafo original.
3. `Ady_Mat`: modeliza una matriz de adyacencia en la cual el usuario especifica el tipo de dato que se desea como entrada de la matriz.
4. `Bit_Mat_Graph<GT>`: el cual modeliza una matriz de adyacencia de bits.

Las cuatro clases anteriores se definen en el archivo `tpl_matgraph.H`, de las cuales presentaremos con cierto detalle `Ady_Mat` (§ 7.6.1 (Pág. 629)) y `Bit_Mat_Graph<GT>` (§ 7.6.2 (Pág. 633)).

### 7.6.1 El TAD `Ady_Mat`

En muchas circunstancias se requiere una matriz homomorfa a una de adyacencia que contenga valores distintos y, muy probablemente, de tipos diferentes a los del grafo. Por ejemplo, podría necesitarse una matriz contentiva de duraciones de trayectos entre arcos.

Para la situación anterior se tiene el tipo `Ady_Mat`, el cual modeliza una matriz elementos de tipo `Entry_Type` y especificada de la siguiente forma:

629 *(Ady\_Mat 629)≡*

```
template <class GT, typename __Entry_Type,
 class SA = Default_Show_Arc<GT> >
class Ady_Mat
{
 <Miembros privados de Ady_Mat 630a>
 <Miembros públicos de Ady_Mat 630b>
};

<Implantación de métodos de Ady_Mat 632b>
```

`Ady_Mat` debe estar estrechamente ligado a un objeto `List_Graph`. El punto esencial en la clase `Ady_Mat` está dado por el hecho de que el usuario puede decidir el tipo de datos que albergarán las entradas de la matriz. Consecuentemente, `Ady_Mat` no necesariamente representa una matriz de adyacencia.

Los atributos de `Ady_Mat` son muy similares a los de las matrices anteriores:

630a *(Miembros privados de Ady\_Mat 630a)≡* (629)  
`GT * lgraph;`  
`DynArray<Node*> nodes;`  
`DynArray<Entry_Type> mat;`  
`mutable size_t num_nodes;`  
`mutable Entry_Type Null_Value;`

Uses `DynArray` 34 and `Null_Value`.

Los accesos a un objeto `Ady_Mat` son similares a los de `Map_Matrix_Graph<GT>`. Los nodos pueden referirse por punteros a nodos en un `List_Graph` o por sus índices enteros respecto a la matriz.

El consumo de memoria de la matriz es proporcional a la cantidad de entradas que hayan sido escritas. Una entrada  $(i, j)$  que no haya sido escrita puede retornar `false` cuando se invoque a `mat.exist(index)`. Por este medio se detecta si la entrada es nula.

El resto de los atributos son observables:

630b *(Miembros públicos de Ady\_Mat 630b)≡* (629) 630c▷  
`GT & get_list_graph() { return *lgraph; }`  
  
`const Entry_Type & null_value() const { return Null_Value; }`  
  
`const size_t & get_num_nodes() const { return num_nodes; }`

Uses `Null_Value`.

El uso de `Null_Value` es el mismo que el de la clase `Matrix_Graph<GT>`: definir el cero en la matriz y permitir un ahorro de espacio por los arcos ausentes.

Puesto que el tipo de dato que contiene la matriz es independiente del objeto `List_Graph`, no hay necesidad de ocupar espacio a priori para el contenido de la matriz. Por eso tiene sentido el siguiente constructor:

630c *(Miembros públicos de Ady\_Mat 630b)+≡* (629) <630b 630d▷  
`Ady_Mat(GT & g) : lgraph(&g), num_nodes(lgraph->get_num_nodes())`  
`{`  
 `copy_nodes(g);`  
`}`

En este caso, la matriz no ocupa memoria inicialmente; ésta se apartará a medida que las entradas se vayan escribiendo. Sin embargo, si se opta por construir de esta forma un objeto `Ady_Mat`, entonces el usuario debe encargarse él mismo de escribir todas las entradas de la matriz (a través del operador  $(i, j)$ ), pues el valor `Null_Value` no se ha definido, o definir `Null_Value` antes de efectuar cualquier escritura. El TAD no verifica este orden.

Hay dos maneras de ahorrar memoria por ausencia de arcos o porque la aplicación maneje algún elemento nulo, la primera consiste en definir `Null_Value` en tiempo de construcción mediante:

630d *(Miembros públicos de Ady\_Mat 630b)+≡* (629) <630c 631a▷  
`Ady_Mat(GT & g, const Entry_Type & null)`  
 `: lgraph(&g), num_nodes(lgraph->get_num_nodes()), Null_Value(null)`  
 `{ copy_nodes(*lgraph); }`

Uses `Null_Value`.

En este caso, `Entry_Type` debe ser el mismo tipo que `GT::Arc_Type`. Un error de compilación debe generarse si no es así. La segunda manera es mediante la siguiente primitiva:

631a *<Miembros públicos de Ady\_Mat 630b>+≡* (629) ◁630d 631b▷  
 void set\_null\_value(const `Entry_Type` & null) { `Null_Value` = null; }  
*Uses Null\_Value.*

El tipo `Ady_Mat` está destinado para guardar datos de cualquier tipo, sobre todo para cálculos temporales requeridos por algoritmos de grafos diseñados para matrices de adyacencia. Como tal, es muy común inicializar matrices de tipo `Ady_Mat`. Para esta tarea se proveen dos clases de interfaces:

1.

631b *<Miembros públicos de Ady\_Mat 630b>+≡* (629) ◁631a 631c▷  
 template <class `Operation`> void operate\_all\_arcs\_list\_graph();

Esta primitiva ejecuta la operación `Operation()` sobre cada entrada de la matriz que contenga un arco en `List_Graph`.

Notemos que la operación no se invoca para arcos ausentes.

El formato de la clase `Operation()` es `Operation()(mat, arc, i, j, entry)`, cuyos parámetros se describen así:

- (a) `mat`: es una referencia al objeto `Ady_Mat` sobre el cual se está realizando la operación.
- (b) `arc`: es un apuntador al arco dentro del `List_Graph` asociado a la matriz.
- (c) `i`: es el índice del nodo origen en la matriz.
- (d) `j`: es el índice del nodo destino en la matriz.
- (e) `entry`: es una referencia al contenido de la entrada  $(i, j)$  en la matriz.

Eventualmente, si se requiere transmitir algún parámetro adicional a la operación, entonces puede usarse la siguiente versión:

631c *<Miembros públicos de Ady\_Mat 630b>+≡* (629) ◁631b 632a▷  
 template <class `Operation`> void operate\_all\_arcs\_list\_graph(void \* ptr);

La cual invoca a `Operation()(mat, arc, i, j, entry, ptr)`.

Como ejemplo, consideremos inicializar una matriz con los valores de distancias guardados en cada arco. Suponiendo que la distancia de un arco es accesible mediante un método del arco llamado `get_distance()`, la inicialización de un arco en la matriz puede especificarse de este modo:

```
struct Init_Arc
{
 void operator () (AdyMat<Grafo> & mat, Grafo::Arc * arc,
 int & i, int & j, Entry & entry)
 {
 entry = arc->get_distance();
 }
};
```

De este modo pueden iniciarse todas las entradas mediante:

```
mat.operate_all_arcs_list_graph <Init_Arc> ();
```

Puesto que sólo se inicializan entradas para arcos dentro del List\_Graph asociado, aquellas entradas en donde no haya arco contienen el valor Null\_Value.

2. La segunda interfaz está basada en índices de Ady\_Mat:

632a <*Miembros públicos de Ady\_Mat* 630b>+≡ (629) <631c  
template <class Operation> void operate\_all\_arcs\_matrix();  
  
template <class Operation>  
void operate\_all\_arcs\_matrix(void \* ptr);

El esquema es similar al anterior, salvo que en este caso se recorren todas las entradas y, por tanto, se aparta la memoria para todas. Consecuentemente, el consumo de espacio es  $\mathcal{O}(n^2)$ .

El esquema de parámetros de la clase Operation() es el siguiente:

- (a) mat: referencia a la matriz.
  - (b) src\_node: puntero al nodo origen dentro del List\_Graph asociado.
  - (c) tgt\_node: puntero al nodo destino dentro del List\_Graph asociado.
  - (d) s: índice del nodo origen.
  - (e) t: índice del nodo destino.
  - (f) entry: referencia a entrada dentro de la matriz.
  - (g) ptr (opcional según la interfaz): puntero opaco.

Clarificada la interfaz, podemos mostrar algunas de las implementaciones de los métodos anteriores:

```

 Operation () (*this, arc, src_idx, tgt_idx, entry);
 }
}
template <class GT, typename __Entry_Type, class SA>
template <class Operation>
void Ady_Mat<GT, __Entry_Type, SA>::operate_all_arcs_matrix()
{
 const long & n = num_nodes;
 for (int s = 0; s < n; ++s)
 {
 Node * src_node = get_node<GT>(nodes, s);
 for (int t = 0; t < n; ++t)
 {
 Node * tgt_node = get_node<GT>(nodes, t);
 __Entry_Type & entry = mat.touch(index_array(s, t, num_nodes));

 Operation () (*this, src_node, tgt_node, s, t, entry);
 }
 }
}

```

### 7.6.2 El TAD Bit\_Mat\_Graph<GT>

La última clase de matriz se llama `Bit_Mat_Graph<GT>` y representa una muy simple matriz de adyacencia de bits. Un valor uno indica presencia de arco; uno nulo, ausencia. Por supuesto, el tipo en cuestión está basado en el TAD `BitArray` desarrollado en § 2.1.5 (Pág. 53) y se define de la siguiente manera:

633a *(Métodos privados de Bit\_Mat\_Graph<GT> 633a)*≡ (633b) 633c▷  
*BitArray bit\_array;*  
*Uses BitArray 54a.*

Mientras que el `Bit_Mat_Graph<GT>` se especifica mediante la siguiente clase:

633b *(Bit\_Mat\_Graph<GT> 633b)*≡  
*template <class GT, class SA = Default\_Show\_Arc<GT> >*  
*class Bit\_Mat\_Graph*  
*{*  
 *<Métodos privados de Bit\_Mat\_Graph<GT> 633a>*  
 *<Métodos públicos de Bit\_Mat\_Graph<GT> 634a>*  
*};*

Un `Bit_Mat_Graph<GT>` representa una matriz de adyacencia cuyos bits sólo expresan presencia o ausencia de arco dentro de un grafo representando con un `List_Graph`. `Bit_Mat_Graph<GT>` no sirve para manejar multigrafos o multidigrafos.

`Bit_Mat_Graph<GT>` mantiene un apuntador al `List_Graph` asociado, un arreglo dinámico de nodos que permitirá calcular rápidamente el índice mediante la búsqueda binaria y la cantidad de nodos:

633c *(Métodos privados de Bit\_Mat\_Graph<GT> 633a)+≡* (633b) ▷633a  
*GT \**  
 *lgraph;*  
*DynArray<typename GT::Node\*> nodes;*

```

 mutable size_t n;
Uses DynArray 34.

 Hay varias maneras de construir un Bit_Mat_Graph<GT>:

634a <Métodos públicos de Bit_Mat_Graph<GT> 634a>≡ (633b) 634b▷
 Bit_Mat_Graph() : lgraph(NULL) {}

 Bit_Mat_Graph(GT & g)
 : bit_array(g.get_num_nodes()*g.get_num_nodes()), lgraph(&g)
 {
 copy_list_graph(g);
 }
 Bit_Mat_Graph(const Bit_Mat_Graph & bitmat)
 : bit_array(bitmat.bit_array), lgraph(bitmat.lgraph),
 nodes(bitmat.nodes), n(bitmat.n) {}

 Bit_Mat_Graph(const size_t & dim)
 : bit_array(dim*dim), lgraph(NULL), nodes(dim), n(dim) {}

Uses copy_list_graph.
```

El cuarto constructor no requiere un List\_Graph porque está destinado a declarar matrices de bits a usarse para cálculos parciales. Esto, por supuesto, impide el acceso por direcciones de nodos en un List\_Graph.

Eventualmente se puede especificar el List\_Graph por separado, así como también consultarla:

```

634b <Métodos públicos de Bit_Mat_Graph<GT> 634a>+≡ (633b) ▷634a 634c▷
 void set_list_graph(GT & g)
 {
 const size_t & n = g.get_num_nodes();
 bit_array.set_size(n*n);
 lgraph = &g;
 copy_list_graph(g);
 }
 GT * get_list_graph() { return lgraph; }

Uses copy_list_graph.
```

Finalmente, debemos implantar las cuatro formas de acceso, dos de las cuales pueden instrumentarse directamente de forma similar a las clases matrices que ya hemos tratado:

```

634c <Métodos públicos de Bit_Mat_Graph<GT> 634a>+≡ (633b) ▷634b
 Node * operator () (const long & i)
 {
 return Aleph::get_node<GT>(nodes, i);
 }
 long operator () (Node * node) const
 {
 return node_index<GT>(nodes, n, node);
 }
```

La implantación del acceso a una entrada matricial  $(i, j)$  es más compleja que las anteriores, pues el compilador y la mayoría de hardware no manejan bits como una unidad. El compilador maneja palabras compuestas por bits. No podemos, pues, implantar el operador  $(i, j)$  para que retorne un bit. Tampoco nos sirve que el operador  $(i, j)$  nos

retorne una palabra de tipo int, pues el acceso podría ser de escritura. Debemos entonces distinguir los accesos de escritura de los de lectura y para eso se usa una clase Proxy en el mismo estilo que otros tipos ya estudiados.

### 7.6.3 Algoritmo de Warshall

Ilustraremos nuestra primera utilización de una matriz de adyacencia para implantar un célebre algoritmo que calcula todas las relaciones de conectividad de un grafo a través de los distintos caminos. El algoritmo en cuestión tiene más sentido sobre grafos dirigidos y es conocido como “de Warshall” [178] en honor a su descubridor.

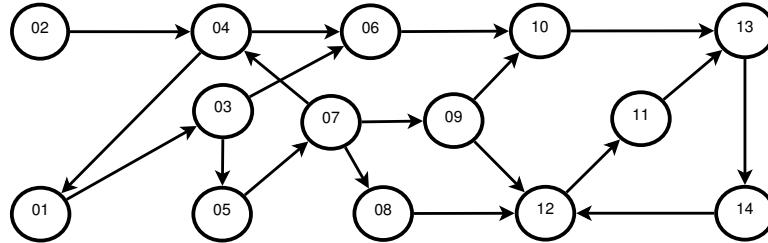


Figura 7.35: Un grafo dirigido

Como enunciamos al principio, un grafo es una manera de representar una relación matemática. En este sentido, una relación binaria  $E$  sobre un conjunto  $V$  (lo cual conforma el grafo  $\langle V, E \rangle$ ) es transitiva si  $\forall x, y, z \in V \implies (x, y), (y, z) \in E$ . Un grafo transitivo es todo aquel que representa a una relación transitiva.

**Definición 7.5 (Clausura transitiva)** La clausura transitiva de una relación  $\mathcal{R}$  es la relación  $\mathcal{R}^*$  definida por todo par  $(x, y) \in \mathcal{R}^*$  tal que existe un camino entre  $x$  e  $y$ .

La clausura transitiva  $\mathcal{R}^*$  de una relación  $\mathcal{R}$  es la mínima relación que contiene a  $\mathcal{R}$ .

Si una relación  $\mathcal{R}$  se representa matricialmente, entonces la matriz correspondiente a la relación  $\mathcal{R}^*$  representa todas las relaciones posibles de conectividad entre cada par de nodos. Una entrada  $\mathcal{R}_{(i,j)}^* \neq 0$  indica que existe un camino entre los nodos  $i$  y  $j$ . Calcular  $\mathcal{R}^*$  nos permite, pues, implementar una prueba de existencia de camino entre cualquier par de nodos.

El algoritmo de Warshall para calcular la clausura transitiva se sirve de la construcción de relaciones entre caminos de longitud  $i + 1$ . La idea es construir una secuencia de relaciones intermedias (o digrafos)  $\mathcal{R}^0, \mathcal{R}^1, \mathcal{R}^2, \dots, \mathcal{R}^n$ . En cada iteración que calcula  $\mathcal{R}^k$ , se examinan todos los tríos  $(v_k, v_i, v_j)$  de nodos de  $\mathcal{R}^{k-1}$ . Si la transitoriedad  $v_i \rightarrow v_j \rightarrow v_k$  existe en  $\mathcal{R}^{k-1}$ , entonces se añade el arco  $v_i \rightarrow v_k$  a  $\mathcal{R}^k$ .

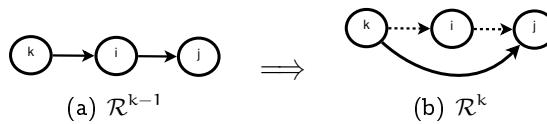


Figura 7.36: Añadidura de arco en  $\mathcal{R}^k$

Por supuesto, si dentro de un trío  $(v_k, v_i, v_j)$  en  $\mathcal{R}^{k-1}$  ya existe un arco  $v_i \rightarrow v_k$ , entonces la entrada  $\mathcal{R}_{i,j}^k = 1$ . La fórmula resultante es entonces:

$$\mathcal{R}_{i,j}^k = \mathcal{R}_{i,j}^{k-1} \vee (\mathcal{R}_{i,k}^{k-1} \wedge \mathcal{R}_{k,j}^{k-1}) \quad (7.3)$$

El proceso se ejecuta  $n$  veces hasta llegar a los caminos de longitud  $n$ . Lo que nos arroja

$$\mathcal{R}^* = \mathcal{R}^n = \prod_{k=1}^n \mathcal{R}^{k-1} \times \mathcal{R}^k \quad (7.4)$$

La operación  $\mathcal{R}^{k-1} \times \mathcal{R}^k$  se efectúa según (7.3).

|    | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| 1  | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  |
| 2  | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  |
| 3  | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  |
| 4  | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  |
| 5  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0  | 0  | 0  | 0  | 0  |
| 6  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 0  | 0  | 0  | 0  |
| 7  | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0  | 0  | 0  | 0  | 0  |
| 8  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 1  | 0  | 0  |
| 9  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 0  | 1  | 0  | 0  |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 1  | 0  |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 1  | 0  |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 1  | 0  | 0  | 0  |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 1  |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 1  | 0  | 0  |

Figura 7.37: Matriz de adyacencia de bits del grafo de la figura 7.35

El cálculo de la clausura transitiva de un grafo representado mediante un objeto de tipo `List_Graph` se instrumenta en el archivo `<warshall.H 636a>` especificado de la siguiente forma:

636a `<warshall.H 636a>≡`

```
template <class GT, class SA = Default_Show_Arc<GT> >
void warshall_compute_transitive_clausure(GT & g,
 Bit_Mat_Graph<GT, SA> & mat)
{
 Calcular clausura transitiva 636b
}
```

g es un `List_Graph` sobre el cual se desea calcular la clausura transitiva, mientras que mat es el resultado de la clausura en cuestión según el algoritmo de Warshall.

A efectos de ahorro de espacio diseñaremos un algoritmo que sólo use dos matrices de manera tal de no redundar el espacio de las  $n$  matrices. Para eso denominamos a mat como la matriz de bits  $\mathcal{R}^k$  en la iteración k y a mat\_prev como  $\mathcal{R}^{k-1}$ , el cual hay que iniciar  $\mathcal{R}^0$ :

636b `<Calcular clausura transitiva 636b>≡` (636a) 637▷
`Bit_Mat_Graph<GT, SA> mat_prev(g);`

Sólo nos queda implantar el algoritmo en si según (7.4):

```
637 <Calcular clausura transitiva 636b>+≡ (636a) ▷636b
const size_t & n = mat.get_num_nodes();
for (int k = 0; k < n; k++)
{
 for (int i = 0; i < n; i++)
 for (int j = 0; j < n; j++)
 mat(i, j) = mat_prev(i, j) or
 (mat_prev(i, k) and mat_prev(k, j));

 mat_prev = mat;
}
```

|    | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| 1  | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  |
| 2  | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  |
| 3  | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  |
| 4  | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  |
| 5  | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  |
| 6  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 1  | 1  | 1  | 1  |
| 7  | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  |
| 8  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 1  | 1  | 1  | 1  |
| 9  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 1  | 1  | 1  | 1  |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 1  | 1  | 1  | 1  |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 1  | 0  |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 1  | 0  | 1  | 0  |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 1  | 1  | 1  | 1  |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 1  | 1  | 0  |    |

Figura 7.38: Clausura transitiva del grafo del la figura 7.35 (pág. 635).

## 7.7 Grafos dirigidos

Al inicio de este capítulo introducimos la noción de grafo dirigido o digrafo. En § 7.3.2 (Pág. 547) presentamos el TAD que lo representa, List\_Digraph, cuya implantación es fundamentalmente idéntica a la de List\_Graph, pues ésta última es su base por herencia.

Casi todos los algoritmos que hasta el presente hemos diseñado operan con digrafos. Muchos algoritmos matriciales, en la ocurrencia el de Warshall para la clausura transitiva, y el de Floyd-Warshall para el cálculo de caminos mínimos (ver § 7.9.2 (Pág. 685)), fueron concebidos para digrafos. Operan perfectamente sobre grafos porque un grafo es también un digrafo<sup>16</sup>.

<sup>16</sup>Aquí puede revelársenos una inconsistencia expresada por el hecho de que List\_Digraph hereda de List\_Graph y no al revés, como pudiera sugerir la línea de razonamiento que estamos tomando. La inversión se debe a la estructura de datos y la reutilización. En lo que concierne a la implantación, la única diferencia entre los dos tipos se encuentra en la implantación del método `insert_arc()` (§ 7.3.10.4 (Pág. 572)). Si observamos bien este método podemos percatarnos de que podríamos implantar a los digrafos como base de los grafos. La decisión obedeció a que los algoritmos sobre grafos son más generales que los de digrafos y, sobre todo, más simples. Parece, pues, con más sentido común que los grafos sean base objetiva (tienen los mismos métodos) y subjetiva de los digrafos y no al contrario.

### 7.7.1 Conectividad entre digrafos

En digrafos, la idea de conectividad puede ser confusa. Dos nodos son conexos o están “directamente conectados” si entre ellos existe un arco. Del mismo modo, son conexos o están “indirectamente conectados” si entre ellos existe un camino que los conecte. Similarmente, un digrafo es conexo si todos sus nodos son conexos. En un digrafo, empero, el que un nodo  $p$  esté conectado a  $q$  no implica que  $q$  esté conectado a  $p$ . Este hecho tiene consecuencias en los algoritmos. En la ocurrencia de esta subsección, nos revela que la rutina `test_connectivity()` estudiada en § 7.5.3 (Pág. 589) no nos sirve para digrafos.

Para digrafos es preferible a menudo emplear la noción de “alcanzabilidad” entre dos nodos: un nodo  $q$  es “alcanzable” desde otro  $p$  si existe un camino  $p \rightarrow q$ .

Así las cosas, podemos hacer una prueba de conectividad basándonos en el recorrido en profundidad que verifique si desde cada nodo se pueden alcanzar los restantes, la cual ya fue previamente instrumentada:

638a *(Algoritmos para digrafos 638a)*≡

```
template <class GT, class SA = Default_Show_Arc<GT> >
bool is_reachable(GT & g, typename GT::Node * src, typename GT::Node * tgt)
{
 return test_for_path <GT, SA> (g, src, tgt);
```

638b▷

La misma rutina basada en la búsqueda en profundidad puede usarse sin problema alguno para digrafos representados con listas.

### 7.7.2 Inversión de un digrafo

En ocasiones puede ser útil obtener el “digrafo inverso”  $\vec{G} = \langle V, \vec{E} \rangle$ , es decir, un digrafo con los mismos nodos pero sus arcos invertidos. Esto se logra muy fácilmente recorriendo el digrafo e insertando arcos invertidos en el digrafo destino:

638b *(Algoritmos para digrafos 638a)*+≡

◁638a

```
template <class GT, class SA = Default_Show_Arc<GT> >
void invert_digraph(GT & g, GT & gi)
{
 // recorrer todos los arcos del digrafo g
 for (Arc_Iterator<GT, SA> it(g); it.has_current(); it.next())
 {
 typename GT::Arc * arc = it.get_current();

 // procesar nodo origen
 typename GT::Node * ssrc = g.get_src_node(arc);
 typename GT::Node * rsrc = mapped_node<GT> (ssrc);
 if (rsrc == NULL) // ¿ya está creado ssrc en gi?
 {
 // no == crearlo, insertarlo y mapearlo
 unique_ptr<typename GT::Node> rsrc_auto(new typename GT::Node(ssrc));
 gi.insert_node(rsrc_auto.get());
 GT::map_nodes(ssrc, rsrc_auto.get());
 rsrc = rsrc_auto.release();
 }
 // procesar nodo origen
 typename GT::Node * stgt = g.get_tgt_node(arc);
```

```

typename GT::Node * rtgt = mapped_node<GT> (stgt);
if (rtgt == NULL) // ¿ya está creado ssrc en gi?
{ // no == crearlo, insertarlo y mapearlo
 unique_ptr<typename GT::Node> rtgt_auto(new typename GT::Node(stgt));
 gi.insert_node(rtgt_auto.get());
 GT::map_nodes(stgt, rtgt_auto.get());
 rtgt = rtgt_auto.release();
}
typename GT::Arc * ai = gi.insert_arc(rtgt, rsrc, arc->get_info());
GT::map_arcs(arc, ai);
}

```

Uses `mapped_node` 560b.

`invert_digraph()` es una rutina muy simple. Claramente, su desempeño es  $\mathcal{O}(E)$  y a menudo se usa como etapa inicial de otros algoritmos. Una aplicación es conocer el grado de entrada de un nodo. Dado un nodo  $p$ , `g.get_num_arcs(p)` nos proporciona el grado de salida, mientras que `gi.get_num_arcs(p)` el de entrada, siendo `gi` el grafo inverso de `g`.

En expresiones formales suele designarse al grado de entrada como  $g_{in}(v)$  y a  $g_{out}(v)$  como el de salida.

En la jerga de digrafos suelen usarse los términos “grado de adyacencia” para referir al grado de salida y “grado de incidencia” para el de entrada. En ese sentido se distinguen dos clases de nodos especiales. A uno al cual sólo le lleguen arcos y desde el cual no salga ninguno, es decir, a uno cuyo grado de adyacencia sea cero, se le tilda de “terminal” o “sumidero”. En el sentido simétricamente inverso, a uno desde el cual sólo salgan arcos y no le llegue ninguno, o sea, a uno con grado de incidencia cero, se le llama “fuente” u “origen”.

### 7.7.3 Componentes fuertemente conexos de un digrafo

Una idea similar a la conectividad en digrafos es la de “conectividad fuerte”. Decimos que dos nodos  $u$  y  $v$  están fuertemente conectados si existen los caminos  $u \rightsquigarrow v$  y  $v \rightsquigarrow u$ .

En función de su alcanzabilidad decimos que un digrafo es “fuertemente conexo” si todos sus nodos son alcanzables entre sí. Contrariamente, decimos que es “débilmente conexo” si no es fuertemente conexo.

Una indicación de la “fuerza de conectividad” de un grafo puede conocerse mediante el algoritmo de Warshall (§ 7.6.3 (Pág. 635)). Si el grafo es fuertemente conexo, entonces su clausura transitiva no contiene ceros. Encontramos pues en este algoritmo una técnica  $\mathcal{O}(V^3)$  para averiguar por la conectividad fuerte, así como también indagar sobre la alcanzabilidad entre cualquier par de nodos. Disponiendo de la clausura transitiva, la prueba de alcanzabilidad toma a lo sumo  $\mathcal{O}(\lg V)$ , si se requiere indizar los nodos.

Para grafos enormes y densos, el algoritmo de Warshall puede ser inaplicable en espacio; en estos casos podemos adoptar una estrategia basada en la rutina `is_reachable()`; invocarla para cada par de nodos nos hace el mismo trabajo en  $\mathcal{O}(V^2) \times \mathcal{O}(V E) = \mathcal{O}(V^3 E)$  pasos; más costoso en tiempo pero menos en espacio.

En función de la conectividad fuerte, en un digrafo pueden identificarse sus “componentes fuertemente conexos”; es decir, bloques o subgrafos débilmente

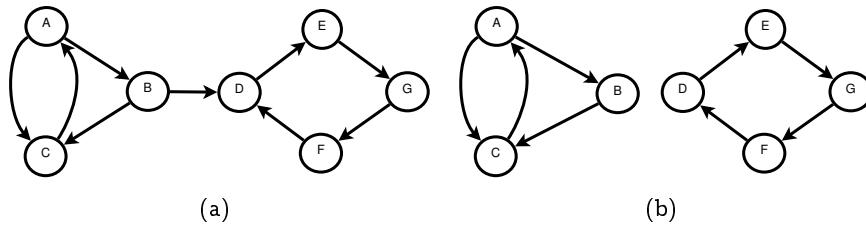


Figura 7.39: Un digrafo y sus dos componentes fuertemente conexos

conexos entre si, pero fuertemente conexos en sus nodos internos.

Hay varios y célebres algoritmos para calcular los componentes fuertemente conexos.

#### 7.7.3.1 Algoritmo de Kosaraju

Al más antiguo algoritmo conocido para determinar los componentes fuertemente conexos de un digrafo, y probablemente el más simple de comprender, se le conoce como el de Kosaraju.

##### Algoritmo 7.2 (Algoritmo de Kosaraju)

La entrada es un digrafo  $G = \langle V, E \rangle$ .

La salida es una lista  $l$  de componentes fuertemente conexos.

1.  $l = \emptyset$
2. Haga una búsqueda en profundidad en  $G$  en la cual los nodos se enumeran en sufijo según su orden de terminación. Sea  $df(v)$  este número.
3. Construya el grafo invertido  $\vec{G} = \langle V, E' \rangle$  mapeado con  $G$ , es decir, los campos  $df(v)$  en  $\vec{G}$  son los mismos calculados en el paso 2.
4. Sea  $S$  la secuencia de nodos ordenada descendente por  $df$ . Sea  $i = |V|$ .
5.  $\forall v \in S \implies$  (los nodos aparecen ordenados por  $df(v)$  desde el mayor hasta el menor)
  - (a) Si  $v$  no está marcado  $\implies$ 
    - i. Sea  $T = \emptyset$
    - ii.  $T =$  grafo alcanzable por el recorrido en profundidad desde  $v$  en  $\vec{G}$  sobre todo nodo que no esté visitado.

Al final de ciclo,  $T$  es un componente fuertemente conexo de  $G$ .

6. Marque todos los nodos y arcos de  $T$ .

7.  $l = l \cup T$ .

Podemos identificar tres fases principales, todas  $\mathcal{O}(V) + \mathcal{O}(E)$ : el recorrido en profundidad que enumera los nodos por su orden de terminación recursiva, la construcción de  $\vec{G}$  y el recorrido final ordenado descendente en el que se descubren los componentes fuertemente conexos. Dado un bloque fuertemente conexo, la enumeración inversa de los nodos refleja el orden de visita del recorrido en profundidad. En el ejemplo de la figura 7.40-a identificamos a los nodos E (10), G (5), H (9) e I (8) como los últimos nodos

en marcarse en sufijo de los bloques identificados en el grafo inverso de la figura 7.40-b como  $\alpha$ ,  $\beta$ ,  $\gamma$  y  $\delta$  respectivamente. En el caso del primer nodo E (10), éste descubre el componente  $\alpha$  sin posibilidad de mirar algunos de los bloques restantes. Luego, desde H (9), puede alcanzarse el bloque  $\alpha$ , pero éste ya está marcado como visitado, lo que hace que sólo se vea el nodo H (9). Similarmente, desde el nodo I (8) se pueden alcanzar los bloques  $\gamma$  y  $\alpha$ , pero una vez más, esto se evita al ya estar visitados.

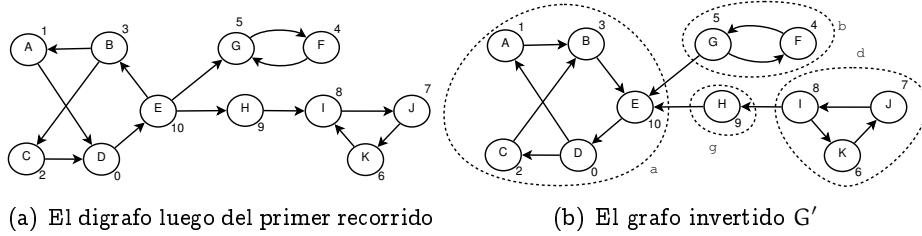


Figura 7.40: Las dos fases del algoritmo de Kosaraju. Los nodos están etiquetados según su orden de terminación recursiva en profundidad. Se asume que la llamada inicial ocurre en el nodo E

El cálculo de  $df(v)$  se efectúa en profundidad, pero la asignación del contador de visita se hace el sufijo, o sea, al término de la llamada recursiva, tal como se indica en la siguiente rutina:

641 *⟨Asignación de df en sufijo 641⟩≡*

```

template <class GT, class SA> inline static
void __dfp(GT & g, typename GT::Node * p, DynArray<typename GT::Node*> & df)
{
 if (IS_NODE_VISITED(p, Depth_First))
 return;

 NODE_BITS(p).set_bit(Depth_First, true);

 // recorre en profundidad los arcos de p
 for (Node_Arc_Iterator<GT,SA> it(p); it.has_current(); it.next())
 {
 typename GT::Arc * a = it.get_current_arc();
 if (IS_ARC_VISITED(a, Depth_First))
 continue;

 ARC_BITS(a).set_bit(Depth_First, true);

 __dfp<GT,SA>(g, it.get_tgt_node(), df);
 }
 df.append(p); // asignación de df(p) en sufijo
 NODE_COUNTER(p) = df.size();
}

```

Uses DynArray 34.

En lugar de pasar un contador, la rutina recibe un arreglo dinámico de punteros a nodos. La idea es guardar en el arreglo la secuencia de asignación del  $df(v)$  para un nodo dado  $v$ . El elemento  $df[0]$  representa el primer nodo al cual se asigna su  $df$ , el cual es el más profundo en la llamada recursiva. Análogamente,  $df[1]$  es el segundo nodo y así sucesivamente hasta

terminar con el nodo inicio de la llamada recursiva. El uso del arreglo `df[]` evita ordenar explícitamente los nodos por su enumeración o utilizar un heap para conocer el mayor.

La instrumentación final del algoritmo de Kosaraju, así como su verificación de correctitud, se delegan en ejercicio.

### 7.7.3.2 Algoritmo de Tarjan

El algoritmo de Tarjan [78, 166] tiene la virtud de identificar los bloques de un digrafo en un solo recorrido en profundidad y sin necesidad de copiar el grafo. Para eso la rutina emplea una variante de los números `df` y `low` definidos en § 7.5.14 (Pág. 613), (7.1) (pag. 614) acerca del cálculo de los puntos de corte o articulación.

En aquel entonces definimos a  $df(v)$  como el orden en que es visitado el nodo  $v$  por un recorrido en profundidad y a  $low(v)$  como el mínimo  $df$  entre todos los nodos conectados a  $v$  por caminos conformados por arcos abarcadores y no-abarcadores. Puesto que se trata de un digrafo, no requerimos pensar en arcos no-abarcadores, sino más bien en nodos alcanzables. De este modo,  $low(v)$  puede definirse más simple como:

$$\text{low}(v) = \text{mínimo } df \text{ del componente fuertemente conexo donde se encuentra } v \quad (7.5)$$

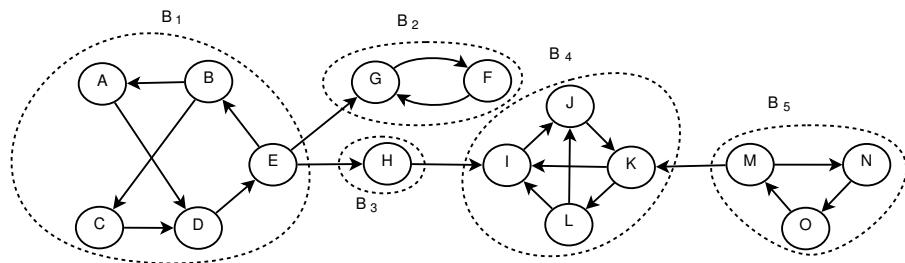


Figura 7.41: Un digrafo con sus bloques

Consideremos un nodo cualquiera  $v$  y una rutina recursiva en profundidad `--scc(v, ...)`, la cual siempre, a su inicio, fija  $df(v) = \text{low}(v) = \text{contador de visita actual}$ . Por la propia definición de conectividad fuerte (§ 7.7.1 (Pág. 638)), es claro que cualquier llamada recursiva, sin importar cuál sea el nodo, descubrirá todos los nodos pertenecientes al bloque en el cual se encuentra  $v$ .

Ahora observemos el digrafo de la figura 7.41 y ejemplifiquemos varias llamadas. Si la llamada ocurre sobre el nodo  $M$ , entonces ésta sólo se descubre un solo bloque, el  $B_5$ . Si la llamada ocurre sobre  $H$ , entonces se descubren los bloques  $B_3$ ,  $B_4$  y  $B_5$ . Si la llamada ocurre sobre  $D$ , entonces se descubren todos los bloques. Surge pues la pregunta ¿existe alguna manera de delimitar el bloque?, es decir, cuando inspeccionamos un arco  $w \rightarrow x$ , ¿cómo podemos discernir si  $w$  y  $x$  están o no en el mismo bloque?

Independientemente del nodo por donde iniciemos un recorrido en profundidad, el valor  $\text{low}(v)$ , para cualquier nodo  $v$ , debe corresponderse con el valor  $df(v)$  del primer nodo visitado dentro del bloque. Retomando el ejemplo de la figura 7.41, un eventual orden de visita puede examinarse en la figura 7.42. El orden de primeras llamadas a `--scc(v, ...)` se muestra en la figura 7.43. Cada árbol corresponde a una llamada desde el `for` de `tarjan_connected_components()`, mientras que cada árbol muestra los bloques que se descubren recursivamente en profundidad.

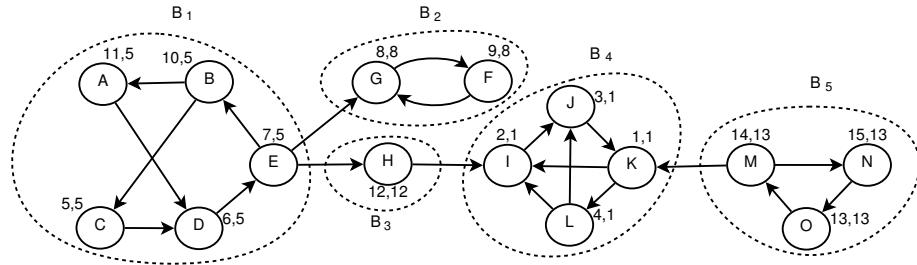


Figura 7.42: Un recorrido en profundidad del digrafo de la figura 7.41. Cada nodo está etiquetado con el par  $df, low$ .

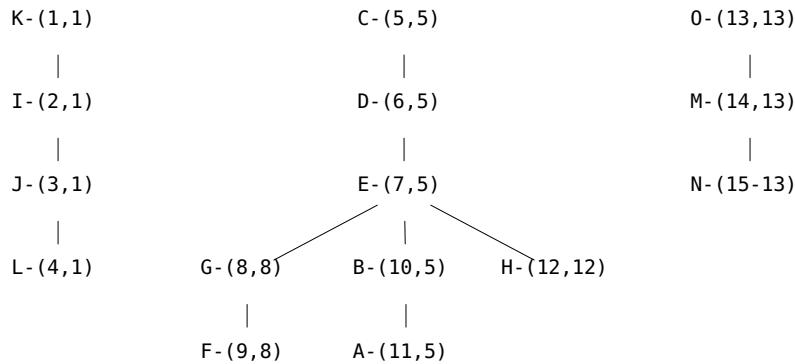


Figura 7.43: Árboles de recorrido en profundidad para el digrafo de la figura 7.42. Cada nodo muestra la etiqueta del nodo y sus números  $df$  y  $low$ .

De la figura 7.43 podemos hacer algunas observaciones en pos de discernir si un par de nodos están o no dentro del mismo bloque:

1. Si el árbol tiene una sola rama, entonces éste contiene un solo bloque. Este es el caso de los bloques  $B_4$  y  $B_5$ .
2. Si el árbol contiene varias ramas, entonces hay tantas ramas como bloques. Este es el caso del segundo árbol, cuya raíz es  $C$ , el cual contiene a los bloques  $B_1$ ,  $B_2$  y  $B_3$ .

En este caso, ¿cómo distinguimos un bloque del otro? Una observación más detallada nos permite identificar que los bloques  $B_2$  y  $B_3$  se encuentran en ramas hijas del nodo  $E$ . De aquí nos aparece la pregunta ¿cómo identificamos el inicio de una rama?. La respuesta está dada por el par  $\langle df(v), low(v) \rangle$ . Cada inicio de rama, que se corresponde al inicio de un bloque, se identifica por el hecho de que  $df(v) = low(v)$ . Esto es independiente de la forma de la rama.

El valor  $df(v)$  es calculable mediante un contador de visitas, mientras que el valor  $low(v)$  es calculable por retorno de recursión (backtrack) y por inspección de arcos no-abarcadores. Conformémonos por los momentos con asumir que se calcula el valor de  $low(v)$  y estudiemos un primer y simple lema.

**Lema 7.2** Sea un digrafo  $G = \langle V, E \rangle$  y  $T$  un árbol abarcador en profundidad cualquiera. Sea  $v \in V$  un nodo cualquiera de  $G$  tal que  $df(v) = low(v)$ . Sea  $B$  el componente fuertemente conexo al que pertenece  $v$ . Entonces,

$$\forall w \in V \mid low(w) = low(v) \iff w \in B \quad (7.6)$$

Dicho de otra forma, todo nodo  $w$  perteneciente al bloque de  $v$ , y sólo ellos, tienen valor  $\text{low}(w) = \text{low}(v)$ .

### Demostración

**Necesidad  $\implies$**  : por la propia definición de  $\text{low}$  (7.5) pag. 642,  $\text{low}(w) = \text{low}(v)$ , pues si no contradiría la definición.

**Suficiencia  $\iff$**  : supongamos un nodo  $x \in B \mid \text{low}(x) \neq \text{low}(v)$ . Esta suposición contradice la definición (7.5) pag. 642  $\square$

Si tenemos un digrafo con sus valores  $\text{df}$  y  $\text{low}$  calculados, entonces podemos distinguir sus bloques del siguiente modo:

1. Cada nodo  $v$  con  $\text{df}(v) = \text{low}(v)$  se encuentra en un bloque distinto. Existen tantos bloques como nodos que satisfagan  $\text{df}(v) = \text{low}(v)$ .
2. Por cada bloque identificado mediante un nodo  $v$  con  $\text{df}(v) = \text{low}(v)$ , los nodos  $w$  con  $\text{low}(w) = \text{low}(v)$  pertenecen al mismo bloque de  $v$

El próximo lema indica que el recorrido en profundidad descubre los bloques.

**Lema 7.3** Sea un digrafo  $G = \langle V, E \rangle$  y  $T$  un árbol abarcador en profundidad cualquiera. Sea  $v \in V$  un nodo cualquiera en  $G$  tal que  $\text{df}(v) = \text{low}(v)$ , entonces el bloque al que pertenece  $v$  se abarca enteramente desde  $T$ .

**Demostración** Por la definición (7.5) pag. 642,  $v$  es el primer nodo visitado en el recorrido en profundidad. Así, en su árbol abarcador,  $v$  tiene que ser el ancestro de todos los nodos del bloque al que pertenece  $v$ . Consecuentemente, el resto de los nodos del bloque se abarca desde  $v$   $\square$

Ahora consideremos el algoritmo de Tarjan. La rutina que funge de interfaz del algoritmo, y que invoca al recorrido recursivo en sí, es:

644

*(Componentes fuertemente conexos 644)  $\equiv$* 

647a▷

```
template <class GT, class SA> inline void
tarjan_connected_components(GT & g, DynDlist <GT> & blk_list,
 DynDlist<typename GT::Arc*> & arc_list)
{
 (Inicializar Tarjan 645a)
 for (typename GT::Node_Iterator it(g); curr_df < n; it.next())
 {
 typename GT::Node * v = it.get_current();
 if (not IS_NODE_VISITED(v, Aleph::Depth_First))
 __scc<GT, SA>(g, blk_list, stack, v, curr_df);
 }
 (Construir y mapear los bloques 646c)
}
```

Uses DynDlist 85a.

$g$  es el digrafo sobre el cual se desea calcular los bloques.  $\text{blk\_list}$  es una lista de digrafos mapeados correspondiente a los distintos bloques de  $g$ . Finalmente, los arcos de  $g$  que cruzan entre los bloques se guardan en la lista  $\text{arc\_list}$ .

La rutina emplea algunas variables internas, de las cuales ahora es menester clarificar las siguientes:

645a *(Inicializar Tarjan 645a)≡* (644 647a 653)  
`long curr_df = 0; // contador de visitas  
const long & n = g.get_num_nodes();`

$\text{curr\_df}$  es el contador de nodos visitados, el cual sólo se incrementa al descubrir un nodo no visitado y se emplea para asignar el valor de  $\text{df}(v)$  y el inicial de  $\text{low}(v)$  a lo largo de las llamadas recursivas de  $\_\text{scc}()$ .

El `for` revisa los nodos, invoca a  $\_\text{scc}()$  para cada nodo no visitado y se detiene cuando se han visitado todos los nodos, lo cual se detecta cuando  $\text{curr\_df} == n$ .

$\_\text{scc}()$  se define del siguiente modo:

645b *(Rutinas estáticas de Tarjan 645b)≡* 647b ▷  
`template <class GT, class SA> inline  
void _scc(GT & g, DynDlist <GT> & list,  
 DynListStack<typename GT::Node*> & stack,  
 typename GT::Node * v, long & df_count)  
{  
 <Inicio de recorrido sobre v 645c>  
 <Recorrer en profundidad bloque conexo a v 646b>  
 <Verificar si df(v) = low(v) y calcular bloque 645d>  
}`

Uses `DynDlist 85a` and `DynListStack 105c`.

Esta rutina recorre en profundidad el nodo  $v$  del grafo  $g$  y guarda en la lista  $\text{list}$  sus bloques;  $\text{stack}$  es una pila de nodos visitados cuyo sentido explicaremos más adelante,  $v$  es el nodo actual que se recorre en profundidad y  $\text{df\_count}$  es el contador de visitas.

El primer paso al entrar a  $\_\text{scc}(v, \dots)$  es empilar  $v$  en la pila  $\text{stack}$ :

645c *(Inicio de recorrido sobre v 645c)≡* (645b 647b) 646a ▷  
`push_in_stack<GT>(stack, v);`

Una vez recorridos en profundidad los nodos adyacentes a  $v$ , se pregunta por  $\text{df}(v) = \text{low}(v)$ . Si es cierto, entonces la pila contiene la secuencia  $z, y, \dots, w, v$  (invertida), por lo que podemos, desempilar con seguridad el bloque hasta que saquemos a  $v$ :

645d *(Verificar si df(v) = low(v) y calcular bloque 645d)≡* (645b)  
`if (low <GT> (v) == df <GT> (v)) // ¿primer nodo visitado del bloque?  
{  
 // sí ==> saque los nodos del bloque que están en pila  
 const size_t & blk_idx = list.size();  
 GT & blk = list.append(GT());  
 while (true) // sacar el bloque de la pila hasta sacar a v  
 {  
 typename GT::Node * p = pop_from_stack<GT>(stack);  
 typename GT::Node * q = blk.insert_node(p->get_info());  
 GT::map_nodes(p, q);  
 NODE_COUNTER(p) = NODE_COUNTER(q) = blk_idx;  
 if (p == v)  
 break;  
 }
}`

```
}
```

El valor `blk_idx` es el “número de bloque”. Este índice permite, luego de haber recorrido enteramente el digrafo, reconocer los distintos bloques.

El conocimiento que nos proporciona el lema 7.3 nos asegura que el bloque está entera y secuencialmente contenido en la pila, pues  $v$  es raíz del bloque en el árbol abarcador en profundidad.

La determinación de  $\text{low}(v)$  se hace por retroceso (backtracking). Cuando visitamos un nodo  $v$ , el valor de  $\text{low}(v)$  se inicia en  $\text{df}(v)$ :

646a  $\langle \text{Inicio de recorrido sobre } v \text{ 645c} \rangle \equiv$  (645b 647b)  $\triangleleft 645c$   
`NODE_BITS(v).set_bit(Aleph::Depth_First, true);`  
`df<GT>(v) = low<GT>(v) = df_count++;`

Luego, se revisan los arcos de  $v$  y se invoca recursivamente a `__scc(w, ...)` por cada nodo  $w$  que sea conexo a  $v$  y que no haya sido visitado:

646b  $\langle \text{Recorrer en profundidad bloque conexo a } v \text{ 646b} \rangle \equiv$  (645b 647b)  
`// recorrer en profundidad todos los nodos conectados a v`  
`for (Node_Arc_Iterator<GT, SA> it(v); it.has_current(); it.next())`  
`{`  
 `typename GT::Node * w = it.get_tgt_node();`  
 `if (not IS_NODE_VISITED(w, Aleph::Depth_First))`  
 `{`  
 `__scc <GT, SA> (g, list, stack, w, df_count);`  
 `low <GT> (v) = std::min(low <GT> (v), low <GT> (w));`  
 `}`  
 `else if (is_node_in_stack<GT>(w))`  
 `// si está en pila ==> v fue visitado antes que p`  
 `low <GT> (v) = std::min(low <GT> (v), df <GT> (w));`  
`}`

Al salir de `__scc(g, list, stack, w, df_count)`,  $w$  ya tiene calculado su  $\text{low}(w)$ , por lo que éste se coteja con  $\text{low}(v)$  para verificar si se encuentra un valor menor que el actual. Si  $w$  ya ha sido visitado desde otra iteración de `tarjan_connected_components()`, entonces debemos verificar su  $\text{low}(w)$  sólo si  $w$  pertenece al bloque de  $v$ , lo cual verificamos por la presencia de  $w$  en la pila.

Para almacenar los valores  $\text{df}$  y  $\text{low}$  emplearemos el contador y el cookie, exactamente de la misma manera, y con las mismas rutinas que empleamos en § 7.1 (Pág. 614).

Para culminar, la última parte del algoritmo de Tarjan consiste en construir los subdigrafos mapeados de los bloques:

646c  $\langle \text{Construir y mapear los bloques 646c} \rangle \equiv$  (644)  
`// recorrer cada uno de subgrafos parciales y añadir sus arcos`  
`for (typename DynDlist<GT>::Iterator i(blk_list); i.has_current(); i.next())`  
`{`  
 `// recorrer todos los nodos del bloque`  
 `GT & blk = i.get_current();`  
 `for (typename GT::Node_Iterator j(blk); j.has_current(); j.next())`  
 `{`  
 `typename GT::Node * gsrc = j.get_current();`  
 `// recorrer arcos de gsrc`  
 `for (Node_Arc_Iterator<GT, SA> k(gsrc); k.has_current(); k.next())`  
 `{`

```

 typename GT::Node * gtgt = k.get_tgt_node();
 typename GT::Arc * ga = k.get_current_arc();
 if (NODE_COUNTER(gsrc) != NODE_COUNTER(gtgt))
 { // arco inter-bloque ==> añádalo a arc_list
 arc_list.append(ga);
 continue;
 }
 // insertar y mapear el arco en el sub-bloque
 typename GT::Node * bsrc = mapped_node<GT>(gsrc);
 typename GT::Node * btgt = mapped_node<GT>(gtgt);
 typename GT::Arc * ba = blk.insert_arc(bsrc, btgt, ga->get_info());
 GT::map_arcs(ga, ba);
 }
}
}

Uses DynDlist 85a and mapped_node 560b.

```

`tarjan_connected_components()` hace un trabajo laborioso, pero garantiza la obtención de componentes fuertemente conexos como subgrafos. Sin embargo, a veces sólo es preferible identificar los componentes sin necesidad de mantenerlos como grafos. En ese caso podemos plantear la siguiente versión:

647a *<Componentes fuertemente conexos 644>+≡* ◁644 649a▷

```

template <class GT, class SA> inline void
tarjan_connected_components(GT & g,
 DynDlist<DynDlist<typename GT::Node*>> & blks)
{
 <Inicializar Tarjan 645a>
 for (typename GT::Node_Iterator it(g); curr_df < n; it.next())
 {
 typename GT::Node * v = it.get_current();
 if (not IS_NODE_VISITED(v, Aleph::Depth_First))
 __scc <GT, SA> (g, blks, stack, v, curr_df);
 }
}

```

Uses DynDlist 85a.

La diferencia respecto a la versión anterior estriba en que en lugar de calcular y retornar una lista de subgrafos, trabajamos con una lista de lista de nodos; cada lista contiene los nodos pertenecientes a un componente.

Esta versión sobrecargada también requiere sobrecargar `__scc()` para que reciba la lista de listas:

647b *<Rutinas estáticas de Tarjan 645b>+≡* ◁645b

```

template <class GT, class SA> inline
void __scc(GT & g, DynDlist<DynDlist<typename GT::Node*>> & list,
 DynListStack<typename GT::Node*> & stack,
 typename GT::Node * v, long & df_count)
{
 <Inicio de recorrido sobre v 645c>
 <Recorrer en profundidad bloque conexo a v 646b>
 <Verificar si df(v) = low(v) y eventualmente llenar lista 648>
}

```

Uses DynDlist 85a and DynListStack 105c.

La estructura es esencialmente idéntica a la anteriormente desarrollada, salvo por el último bloque  $\langle$  Verificar si  $df(v) = low(v)$  y eventualmente llenar lista 648 $\rangle$ , el cual se describe del siguiente modo:

648  $\langle$  Verificar si  $df(v) = low(v)$  y eventualmente llenar lista 648 $\rangle \equiv$  (647b)  
 if ( $low < GT > (v) == df < GT > (v)$ ) // ¿primer nodo visitado del bloque?  
 { // sí ==> saque los nodos del bloque que están en pila  
 list.append(DynDlist<typename GT::Node\*>());  
 DynDlist<typename GT::Node\*> & l = list.get\_last();  
 while (true) // sacar bloque de pila hasta llegar a v  
 {  
 typename GT::Node \* p = pop\_from\_stack<GT>(stack);  
 l.append(p);  
 if (p == v)  
 break;  
 }  
 }  
 }

Uses DynDlist 85a.

Como no se requiere construir subgrafos mapeados de los bloques, esta versión es más eficiente en tiempo y espacio. Además, el mero hecho de separar los nodos del digrafo en conjuntos según su pertenencia al componente fuertemente conexo, arroja bastante información sobre la conectividad. En la ocurrencia, todo el trabajo que ya hicimos de mapear los bloques en subgrafos puede hacerse con la lista de listas. El punto aquí es que con tan solo tener un nodo del componente fuertemente conexo ya se tiene posibilidad de procesarlo sin afectar a los demás componentes.

#### 7.7.4 Prueba de aciclicidad

En § 7.5.6 (Pág. 593) diseñamos la primitiva `is_graph_acyclique()`, destinada a verificar por la existencia de ciclo en un grafo. Como aquella rutina se basa en el pintado de los nodos, no funciona para digrafos. Cuando se encuentra un nodo que ya ha sido pintado, no hay manera de saber si lo fue por un ciclo o por procedencia desde un nodo perteneciente a otro bloque.

Otra primitiva que desarrollamos en § 7.5.5 (Pág. 592) fue `test_for_cycle()`. Podemos verificar por la aciclicidad llamando `test_for_cycle()` por cada nodo del grafo hasta cubrir todos los nodos o encontrar un ciclo. Este enfoque tomaría  $\mathcal{O}(V \times E)$ .

El algoritmo de Tarjan, que es  $\mathcal{O}(V) + \mathcal{O}(E)$ , puede emplearse para determinar eficientemente si un digrafo tiene o no ciclos. Si partimos del hecho de que un bloque debe tener forzadamente ciclos, el algoritmo de Tarjan responde por su existencia. En otras palabras, si la cantidad de bloques de un digrafo es menor que la de nodos, entonces existe al menos un ciclo, pues un componente fuertemente conexo contiene un ciclo. Interpretado de otro modo: basta con encontrar un bloque de dos o más nodos para concluir que el grafo tiene un ciclo.

En términos del algoritmo, esto equivale a detectar si una secuencia de pops en el bloque  $\langle$  Verificar si  $df(v) = low(v)$  y calcular bloque 645d $\rangle$  contiene más de un nodo. Si el lazo extrae al menos dos nodos, entonces tenemos un componente fuertemente conexo de dos o más nodos, el cual, por definición, contiene un ciclo.

Según las consideraciones hechas, la rutina recursiva deviene en:

649a ⟨Componentes fuertemente conexos 644⟩+≡

```

template <class GT, class SA> inline static
bool __ida(GT & g, DynListStack<typename GT::Node*> & stack,
 typename GT::Node * v, long & df_count)
{
 push_in_stack <GT> (stack, v);
 NODE_BITS(v).set_bit(Aleph::Depth_First, true);
 df<GT>(v) = low<GT>(v) = df_count++;

 // recorrer en profundidad todos los nodos conectados a v
 for (Node_Arc_Iterator <GT, SA> it(v); it.has_current(); it.next())
 {
 typename GT::Node * w = it.get_tgt_node();
 if (not IS_NODE_VISITED(w, Aleph::Depth_First))
 {
 if (__ida <GT, SA> (g, stack, w, df_count))
 return true;

 low<GT>(v) = std::min(low<GT>(v), low<GT>(w));
 }
 else if (is_node_in_stack<GT>(w))
 // si está en pila ==> v fue visitado antes que p
 low<GT>(v) = std::min(low<GT>(v), df<GT>(w));
 }
 if (low<GT>(v) == df<GT>(v)) // ¿primer nodo visitado de bloque?
 {
 // sí ==> verifique si tiene dos o más nodos
 int i = 0;
 for (; true; ++i) // sacar bloque de la pila hasta sacar a v
 {
 typename GT::Node * p = pop_from_stack <GT> (stack);
 if (p == v)
 break;
 }
 return i > 1; // si sacamos dos o más nodos entonces hay ciclo
 }
 return false;
}

```

Uses DynListStack 105c.

`__ida()` es una rutina estática. La de uso público se define como:

649b ⟨Componentes fuertemente conexos 644⟩+≡

```

template <class GT, class SA> inline bool is_digraph_acyclique(GT & g)
{
 long curr_df = 0; // contador de visitas
 const long & n = g.get_num_nodes();
 DynListStack<typename GT::Node*> stack; // pila que define rama
 for (Node_Iterator <GT, SA> it(g); curr_df < n; it.next())
 {
 typename GT::Node * v = it.get_current();
 if (not IS_NODE_VISITED(v, Aleph::Depth_First))
 if (__ida<GT, SA>(g, stack, v, curr_df)) // visitar v
 return true;
 }
}

```

◁647a 649b▷

```
 }
 return false;
}
```

La cual tiene la misma estructura que tarjan\_connected\_components().

Este algoritmo determina la existencia de ciclos en un digrafo en tiempo  $\mathcal{O}(V) + \mathcal{O}(E)$  y con un máximo consumo de espacio de  $\mathcal{O}(V)$ .

### 7.7.5 Cálculo de ciclos en un digrafo

En el mismo sentido de la variante de la subsección precedente podemos emplear el algoritmo de Tarjan para calcular efectiva y eficientemente un ciclo en un digrafo.

Cuando sacamos de la pila en el bloque *(Verificar si  $df(v) = \text{low}(v)$  y calcular bloque 645d)*, construimos un digrafo auxiliar que mapea al componente conexo. Sobre este componente buscamos un ciclo mediante una búsqueda en profundidad. Finalmente, mediante el mapeo, construimos el camino correspondiente al ciclo:

```

650 <Componentes fuertemente conexos 644>+≡ ◁649b 652b▷
 template <class GT, class SA> inline static
bool __cc(GT & g, DynListStack<typename GT::Node*> & stack,
 typename GT::Node * v, long & df_count, Path<GT> & path)
{
 push_in_stack <GT> (stack, v);
 NODE_BITS(v).set_bit(Aleph::Depth_First, true);
 df<GT>(v) = low<GT>(v) = df_count++;

 // recorrer en profundidad todos los nodos conectados a v
 for (Node_Arc_Iterator <GT, SA> it(v); it.has_current(); it.next())
 {
 typename GT::Node * w = it.get_tgt_node();
 if (not IS_NODE_VISITED(w, Aleph::Depth_First))
 {
 if (__cc<GT, SA> (g, stack, w, df_count, path))
 return true;

 low<GT>(v) = std::min(low<GT>(v), low<GT>(w));
 }
 else if (is_node_in_stack<GT> (w))
 // si está en pila ==> v fue visitado antes que p
 low<GT>(v) = std::min(low<GT>(v), df<GT>(w));
 }

 if (low<GT>(v) == df<GT>(v)) // ¿primer nodo visitado del bloque?
 { // saque nodos de pila e insertelos en bloque auxiliar
 GT blk; // grafo auxiliar
 DynMapAvlTree<typename GT::Node*, typename GT::Node*> table;
 while (true) // saca el componente y lo inserta en blk
 {
 typename GT::Node * p = pop_from_stack<GT>(stack);
 typename GT::Node * q = blk.insert_node(p->get_info());
 table.insert(q, p);
 }
 }
}

```

```

 table.insert(p, q);
 if (p == v)
 break;
 }
 if (blk.get_num_nodes() == 1)
 return false; // si blk sólo tiene un nodo no hay ciclo

 // terminan los de construir el bloque con los arcos
 for (typename GT::Node_Iterator j(blk); j.has_current(); j.next())
 {
 typename GT::Node * bsrc = j.get_current();
 typename GT::Node * gsrc = table[bsrc];

 // recorrer los arcos de gsrc
 for (Node_Arc_Iterator<GT, SA> k(gsrc); k.has_current(); k.next())
 {
 typename GT::Node * gtgt = k.get_tgt_node();
 typename GT::Node ** btgt_ptr = table.test(gtgt);
 if (btgt_ptr == NULL) // ¿arco del bloque?
 continue;

 typename GT::Arc * ga = k.get_current_arc();
 blk.insert_arc(bsrc, *btgt_ptr, ga->get_info());
 }
 }
 // buscar ciclo en blk.
 typename GT::Arc * a = blk.get_first_arc();
 Path<GT> aux_path(blk);

 Find_Path_Depth_First<GT>() (blk, blk.get_tgt_node(a),
 blk.get_src_node(a), aux_path);
 path.clear_path();
 for (typename Path<GT>::Iterator i(aux_path); i.has_current(); i.next())
 path.append(table[i.get_current_node()]);

 path.append(table[blk.get_tgt_node(a)]);

 return true;
}
return false;
}

```

Uses DynListStack 105c and Path 578a.

`Find_Path_Depth_First <GT> ()` no está diseñada para encontrar ciclos. Por ello, lo que hacemos es seleccionar un arco cualquiera  $s \rightarrow t$  y encontrar un camino  $t \rightsquigarrow s$ . De este modo, el ciclo lo conforma  $t \rightsquigarrow s \rightarrow t$ .

La rutina principal se llama `Compute_Cycle_In_Digraph()`, cuya estructura, con el añadido de cerrar el ciclo, es similar a `tarjan_connected_components()`. Es empleada para la búsqueda de ciclos negativos en un digrafo (ver § 7.9.3.9 (Pág. 703)).

### 7.7.6 Prueba de conectividad

La motivación inicial de esta sección bajo la cual desarrollamos todo este corpus fue la de conectividad de un digrafo (§ 7.7.1 (Pág. 638)). En este sentido, tanto el algoritmo de Kosaraju como el de Tarjan nos permiten con certitud determinar si un digrafo es o no fuertemente conexo. La idea es simple: si el cálculo de los componentes fuertemente conexos arroja un solo bloque, entonces el digrafo es enteramente conexo. Pero pagar estos costes, especialmente el consumo de memoria, sólo para un problema de existencia es innecesario, pues podemos adaptar el algoritmo de Tarjan para responder por la prueba de conectividad.

Recordemos que el algoritmo de Tarjan detecta un nuevo componente en el bloque *(Verificar si  $df(v) = low(v)$  y eventualmente llenar lista 648)*. Así, lo único que estructuralmente debemos hacer para verificar la conectividad es modificar este bloque de la siguiente manera:

652a *(Verificar si hay más de un bloque 652a)≡* (652b)  
 if ( $low > v$  ==  $df > v$ ) // ¿primer nodo visitado de bloque?  
 { // saque nodos de pila hasta encontrar a v  
 while ( $pop\_from\_stack > stack$  != v);  
 return stack.is\_empty();  
}

Notemos que si el digrafo es enteramente conexo, entonces la pila debe quedar vacía al culminar el while. Si eso no ocurre, entonces hay al menos dos componentes, lo que significa que el digrafo no es conexo.

Con el bloque *(Verificar si hay más de un bloque 652a)*, el recorrido en profundidad característico del algoritmo de Tarjan es el mismo:

652b *(Componentes fuertemente conexos 644)+≡* <650 653>  
 template <class GT, class SA> inline  
 bool \_\_ttc(GT & g, DynListStack<typename GT::Node\*> & stack,  
 typename GT::Node \* v, long & df\_count)  
{  
 push\_in\_stack(>stack, v);  
 NODE\_BITS(v).set\_bit(Aleph::Depth\_First, true);  
 df > (v) = low > (v) = df\_count++;  
 // recorrer en profundidad todos los nodos conectados a v  
 for (Node\_Arc\_Iterator<GT, SA> it(v); it.has\_current(); it.next())  
 {  
 typename GT::Node \* w = it.get\_tgt\_node();  
 if (not IS\_NODE\_VISITED(w, Aleph::Depth\_First))  
 {  
 if (not \_\_ttc <GT, SA> (g, stack, w, df\_count))  
 return false;  
 low > (v) = std::min(low > (v), low > (w));  
 }  
 else if (is\_node\_in\_stack(>w))  
 low > (v) = std::min(low > (v), df > (w));  
 }  
*(Verificar si hay más de un bloque 652a)*

```

 return true;
}

Uses DynListStack 105c.
```

Finalmente, la prueba de conectividad se instrumenta del siguiente modo:

```

653 <Componentes fuertemente conexos 644>+≡ ◁652b
template <class GT, class SA> inline bool tarjan_test_connectivity(GT & g)
{
 <Inicializar Tarjan 645a>
 for (typename GT::Node_Iterator it(g); curr_df < n; it.next())
 {
 typename GT::Node * v = it.get_current();
 if (not IS_NODE_VISITED(v, Aleph::Depth_First))
 if (not __ttc <GT, SA> (g, stack, v, curr_df))
 return false;
 }
 return true;
}
```

### 7.7.7 Digrafos acíclicos (DAG)

**Definición 7.6 (Digrafo acíclico -DAG-)** Un digrafo acíclico o DAG<sup>17</sup>, es un digrafo sin ciclos.

El empleo más popular de los digrafos acíclicos es para modelizar planificaciones u horarios. En este caso, los nodos representan “tareas” o “actividades” que instrumentan una obra y los arcos las precedencias posibles de las actividades. Consideremos la tarea general de vestirse, la cual consiste en ponerse las prendas en un orden específico que puede relacionarse según el digrafo de la figura 7.44. Tenemos nueve pasos, resumidos en el

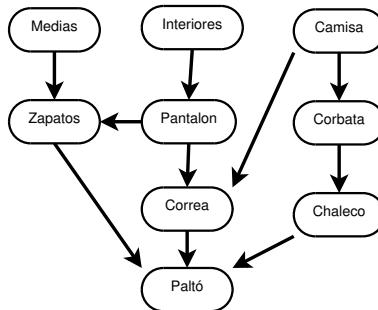


Figura 7.44: Grafo dirigido acíclico que define en los pasos para vestirse

digrafo bajo los nombres de las prendas, que no pueden efectuarse en un orden arbitrario, sino en el expresado por las relaciones del digrafo. Para vestirnos adecuadamente no podemos, por ejemplo, ponernos la correa antes que el pantalón.

En un digrafo acíclico siempre se distinguen, cuando menos, un nodo fuente y uno sumidero. En el vestirse tenemos a los interiores, medias y camisa como fuentes, mientras que a los zapatos y al paltó como sumideros.

<sup>17</sup>Del inglés “Directed Acyclic Graph”.

### 7.7.8 Planificación de tareas

El vestirse nos está tan arraigado que quizá no apreciemos bien la complejidad que tiene el hacerlo en el orden correcto<sup>18</sup>.

Según aumenta la cantidad de tareas y sus prelaciones, aumenta la dificultad. En obras grandes, es necesario, en muchas ocasiones crucial, encontrar el orden secuencial correcto en que deben ejecutarse todas las tareas o actividades. Tal orden es denominado “ordenamiento topológico”.

El vestirse puede ordenarse topológicamente, desde nodos fuentes hasta nodos sumideros. Un ejemplo es mostrado en la figura 7.45. Es decir, desde las actividades iniciales

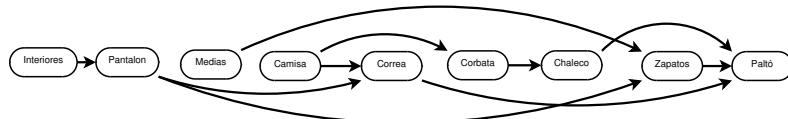


Figura 7.45: Secuencia topológica ordenada para vestirse

o primigenias, pasando por las intermedias según las dependencias, hasta las finales.

En la instrumentación de una obra hay situaciones en que es posible llevar a cabo actividades en paralelo, a veces simultáneamente. En la ocurrencia con el vestirse podríamos ponernos los interiores mientras otras dos personas amigas nos ponen la camisa y las medias. Si consideramos una duración constante para la puesta de una prenda de vestir, entonces, en lugar de emplear nueve pasos para vestirnos solos podríamos emplear cinco si nos ayudan dos personas más. Para aprehenderlo véase la figura 7.46.

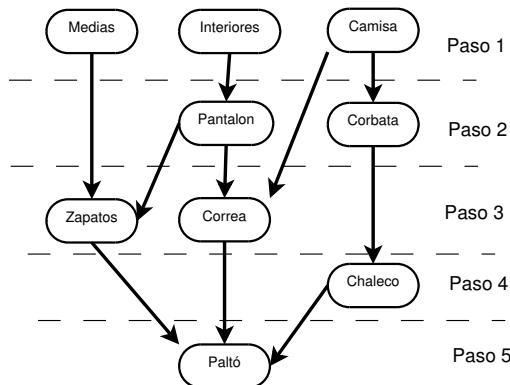


Figura 7.46: Ordenamiento topológico paralelo del vestirse. Pasos que pueden ejecutarse simultáneamente se encuentran en el mismo nivel

Cuando se obra en paralelo hay que sincronizar a los “obrantes” para que no emprendan tareas sin que previamente se hayan culminado actividades dependientes efectuadas por otros obrantes. Por ejemplo, la tarea que ponga las medias debe esperar a que el pantalón esté puesto antes de poner los zapatos.

El problema anterior puede objetizarse de manera general como la planificación de  $n$  tareas, cuyas dependencias están definida por un digrafo acíclico, en  $m$  obrantes.

<sup>18</sup> Considera un individuo que no conozca la cultura occidental y se le proponga vestirse. Considera también caminar por la selva amazónica con vestimentas occidentales.

A las tareas se les puede asociar una duración o coste. En este caso se trata de encontrar la mínima planificación, es decir, de encontrar  $m > 1$  secuencias de tareas, para darle a  $m$  obrantes de manera tal que la duración o coste total sea mínimo.

Los digrafos de tareas pueden simplificarse según circunstancias de la situación real modelizada, del medio de ejecución y del fin de la planificación. Podemos, por ejemplo, asumir que colocarse la camisa y la corbata son una sola tarea con una duración mayor. A esta acción se le denomina reducción porque “reduce” la cantidad de nodos del digrafo, simplifica la comprensión del modelo y, por lo general, reduce el tiempo de ejecución. Cadenas secuenciales, es decir, caminos simples de nodos con un solo arco de entrada y otro de salida, pueden comprimirse a un solo nodo consistente de su suma de tareas.

La reducción anterior se efectuó sobre una cadena secuencial, pero puede perfectamente hacerse sobre un subdigrafo entero. Como ejemplo culminante consideremos la organización del vestirse mostrada en la figura 7.47.

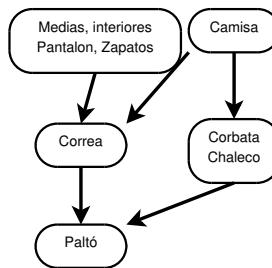


Figura 7.47: Grafo reducido dirigido acíclico de los pasos del vestirse

Los digrafos de planificación tienen diversas aplicaciones, entre las cuales cabe mencionar:

1. Planificación de instrucciones a encauzar por uno o varios procesadores: los compiladores generan instrucciones en código de máquina. Cada instrucción tiene una duración en ciclos y una precedencia sobre otras para que no ocurra ruptura del encauzamiento (pipelining). En este caso, el compilador construye un digrafo acíclico según la índole del procesador y optimiza la secuencia de instrucción.
2. Planificación de obras o proyectos: por ejemplo, para construir un complejo de edificios se puede realizar un digrafo de actividades y, en función de los obrantes (constructoras, maquinarias, cantidad de personas), encontrar una planificación de bajo coste o de mínima duración.

### 7.7.9 Ordenamiento topológico

Existe un clásico y simplísimo algoritmo para encontrar un ordenamiento topológico de un digrafo acíclico:

**Algoritmo 7.3 (Ordenamiento topológico)** La entrada del algoritmo es un digrafo  $G$ .

La salida es una lista de nodos  $l$ , inicialmente vacía, correspondiente al ordenamiento topológico del digrafo  $G$ .

Repita mientras  $G$  tenga nodos:

1. Escoja un nodo sumidero  $v$  (uno que no tenga arcos de salida).
2. Elimínelo de  $G$  junto con todos sus arcos de entrada.
3. Empile<sup>19</sup>  $v$  a la lista  $l$ .

Este algoritmo encuentra el ordenamiento de manera inversa: desde los sumideros hasta los fuentes. Cada iteración elimina un sumidero del digrafo, lo que con certitud deja, al eliminar sus arcos de entrada, al menos un nuevo sumidero dentro del grafo. El algoritmo progresá hasta que no queden más nodos.

Una manera directa y simple de implantar el algoritmo 7.3 es obteniendo el digrafo inverso y entonces ejecutar literalmente el algoritmo. El único inconveniente es que se duplica el espacio, lo que puede ser costoso para digrafos enormes.

El algoritmo 7.3 puede plantearse perfectamente en función de los fuentes; es decir, seleccionar nodos fuentes, eliminarlos junto a sus arcos de salida y continuar con los nuevos fuentes. Aún en este caso es necesario realizar una copia si se desea conservar el digrafo original.

La implantación del algoritmo 7.3 sin efectuar una copia del digrafo es más compleja, pues debemos tratar con dos problemas que el TAD List\_Digraph no nos resuelve directamente: (1) determinar el grado de entrada de un nodo y (2) procesar rápidamente los nodos, desde los sumideros hasta los fuentes.

### Ordenamiento topológico por profundidad

Según el sentido, podemos considerar dos clases de algoritmos. Si vamos desde los sumideros hacia los fuentes, entonces un recorrido en profundidad, que visite cada nodo en sufijo, comenzará por el primer sumidero que sea encontrado. Dado un nodo cualquiera  $curr$ , éste puede recorrerse recursivamente, en profundidad y en sufijo, de la siguiente forma:

```
656 ⟨Ordenamiento topológico 656⟩≡ 657a▷
 template <class GT, class SA> static inline
 void __topological_sort(GT & g, typename GT::Node * curr,
 DynDlist<typename GT::Node*> & list)
 {
 if (IS_NODE_VISITED(curr, Depth_First))
 return;

 NODE_BITS(curr).set_bit(Depth_First, 1); // marcarlo como visitado
 // visitar recursivamente en sufijo los nodos adyacentes a curr
 for (Node_Arc_Iterator<GT, SA> it(curr);
 it.has_current() and list.size() < g.get_num_nodes(); it.next())
 __topological_sort<GT, SA> (g, it.get_tgt_node(), list);

 list.insert(curr); // inserción sufija de nodo que devino sumidero
 }
```

Uses DynDlist 85a.

---

<sup>19</sup>Con la operación de tipo push().

Sin importar cual sea el primer nodo desde el cual se invoque esta rutina, ésta avanzará recursivamente hasta encontrar un nodo ya visitado o hasta un sumidero; en este último caso no se entra en el `for` y el nodo se inserta en `list`. Puesto que insertamos en `list` al principio, el primer sumidero que se observe será el último nodo en la lista, mientras que el último observado, o sea, un fuente, será el primero. Insertar en `list` como en una pila nos proporciona el orden topológico.

Una vez que invoca la última instrucción `list.insert(curr)`, todos los nodos alcanzables desde `curr` ya han sido insertados en `list`. Consecuentemente, el orden de inserción es correcto, pues `curr` va antes en su orden topológico.

Es importante observar que la condición de detención del `for` incluye el hecho de que ya todos los nodos hayan sido visitados. Esto se detecta cuando la cantidad de nodos de la lista sea la misma que la cardinalidad del grafo.

En caso de que `_topological_sort()` se inicie desde un nodo que no es fuente o que existan varios fuentes, quedarán nodos sin visitar, ergo, sin insertar en la lista. Esto se resuelve simplemente recorriendo todos los nodos del grafo e invocando a `_topological_sort()`:

657a *(Ordenamiento topológico 656) +≡* 656 657b ▷

```
template <class GT, class SA> inline
void topological_sort(GT & g, DynDlist<typename GT::Node*> & list)
{
 for (typename GT::Node_Iterator it(g);
 it.has_current() and list.size() < g.get_num_nodes(); it.next())
 {
 typename GT::Node * curr = it.get_current_node();
 if (not IS_NODE_VISITED(curr, Depth_First))
 _topological_sort<GT, SA>(g, curr, list);
 }
}
```

Uses DynDlist 85a.

`topological_sort()` puede iterar tantas veces como arcos tenga el digrafo; esto hace que el desempeño tienda a  $\mathcal{O}(E)$  lo cual, en el peor de los casos, puede ser  $\mathcal{O}(V^2)$ . Pero, por lo general, para un digrafo acíclico,  $E \approx 2V$ .

### Ordenamiento topológico por amplitud

Otro enfoque para ordenar topológicamente se equipara con un recorrido en amplitud, lo cual conlleva la ventaja de controlar mejor el consumo de espacio al limitarlo a una cola de nodos -con el recorrido en profundidad, la recursión puede alcanzar  $\mathcal{O}(V)$ , lo que podría, en algunos casos, acarrear peligro para la pila del sistema-:

657b *(Ordenamiento topológico 656) +≡* 657a 659 ▷

```
template <class GT, class SA> inline
void q_topological_sort(GT & g, DynDlist<typename GT::Node*> & list)
{
 // recorra todos los arcos para contar grados de entrada
 for (Arc_Iterator<GT,SA> it(g); it.has_current(); it.next())
 NODE_COUNTER(it.get_tgt_node())++;

 // revisar nodos con grado de entrada 0 y meterlos en cola
 DynListQueue<typename GT::Node*> q; // cola de fuentes
 for (typename GT::Node_Iterator it(g); it.has_current(); it.next())
```

```

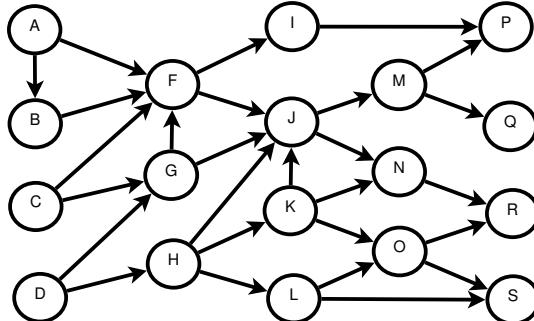
{
 typename GT::Node * p = it.get_current_node();
 if (NODE_COUNTER(p) == 0) // ¿es un nodo fuente?
 q.put(p); // sí ==> colocarlo en la cola
}

while (not q.is_empty())
{
 typename GT::Node * p = q.get(); // saque último fuente
 list.append(p); // insértelo en el orden topológico
 // decrementar grado de entrada de cada nodo adyacente a p
 for (Node_Arc_Iterator<GT, SA> it(p); it.has_current(); it.next())
 {
 typename GT::Node * tgt = it.get_tgt_node();
 if (-NODE_COUNTER(tgt) == 0) // ¿tgt deviene fuente?
 q.put(tgt); // sí ==> colóquelo en la cola
 }
}
}

```

Uses DynDlist 85a.

El desempeño de este algoritmo es  $2 \times \mathcal{O}(V)$  por los dos primeros lazos más  $\mathcal{O}(E)$  por el while. Lo que nos arroja  $\mathcal{O}(E)$  como desempeño general del algoritmo, rendimiento similar a la versión basada en la búsqueda en profundidad.



D → H → L → K → O → S → C → G → A → B → F → J → N → R → M → Q → I → P

(b) `topological_sort()`

D → H → L → K → O → S → C → G → A → B → F → J → N → R → M → Q → I → P

(c) `q_topological_sort()`

Figura 7.48: Un digrafo y sus ordenamientos topológicos

### Rangos topológicos

El rango topológico de un nodo es su nivel en el digrafo respecto a las tareas que lo preceden y las que los suceden. Un rango topológico a secas -sin especificar que se trata de un nodo- es un conjunto de tareas que pueden llevar a cabo independientemente una vez que han sido culminadas las actividades del rango predecesor. Por ejemplo, los rangos topológicos del digrafo de la figura 7.48 son:  $R_1 = \{A, C, D\}$ ,  $R_2 = \{B, G, H\}$ ,  $R_3 = \{F, K, L\}$ ,  $R_4 = \{I, J, O\}$ ,  $R_5 = \{M, N, S\}$  y  $R_6 = \{P, Q, R\}$ . Esto significa que si las tareas toman la misma duración, entonces primero habría que ejecutar  $R_1$ , luego,  $R_2$ , y así sucesivamente.

El cálculo de los rangos puede hacerse mediante cualquiera de los algoritmos de ordenamiento topológico. He aquí una variante del basado en amplitud que emplea dos colas:

```
659 <Ordenamiento topológico 656>+≡ ◁657b
 template <class GT, class SA> inline void
 q_topological_sort(GT & g, DynDlist<DynDlist<typename GT::Node*>> & ranks)
 {
 // recorra todos los nodos para contabilizar grado de entrada
 for (typename GT::Node_Iterator i(g); i.has_current(); i.next())
 for (Node_Arc_Iterator <GT, SA> j(i.get_current_node());
 j.has_current(); j.next())
 NODE_COUNTER(j.get_tgt_node())++;

 // revisar nodos con grado de entrada 0 y meterlos en cola
 DynListQueue<typename GT::Node*> q; // cola de fuentes
 for (typename GT::Node_Iterator it(g); it.has_current(); it.next())
 {
 typename GT::Node * p = it.get_current_node();
 if (NODE_COUNTER(p) == 0) // ¿es un nodo fuente?
 q.put(p); // sí ==> colocarlo en la cola
 }

 while (not q.is_empty())
 {
 DynDlist<typename GT::Node*> * rank = new DynDlist<typename GT::Node*>;
 DynListQueue<typename GT::Node*> aq;
 while (not q.is_empty()) // saca todos los nodos del nivel i
 {
 typename GT::Node * p = q.get(); // saque último fuente
 rank->append(p); // insértelo en el rango topológico
 // decrementar grado entrada de cada nodo adyacente a p
 for (Node_Arc_Iterator<GT, SA> it(p); it.has_current(); it.next())
 {
 typename GT::Node * tgt = it.get_tgt_node();
 if (-NODE_COUNTER(tgt) == 0) // ¿tgt deviene fuente?
 aq.put(tgt); // sí ==> colóquelo en cola auxiliar
 }
 }
 ranks.append(rank);
 q.swap(aq);
 }
 }

```

Uses DynDlist 85a.

El algoritmo es estructuralmente similar a `q_topological_sort()`. El parámetro `ranks` es una lista de listas de nodos. Cada lista es un rango y contiene los nodos que lo conforman. La lista está ordenada por el orden del rango, es decir, la primera lista contiene las tareas que se ejecutan de primero y así sucesivamente.

El algoritmo emplea una cola auxiliar `aq`. Cuando se decrementa el grado y se determina que hay que encolar, el nodo se mete en `qa`. Una vez que se han sacado todos los

nodos de grado cero, los cuales corresponden a los del rango actual, los nodos de  $q_a$  se copian (en  $\mathcal{O}(1)$  mediante `swap()`) hacia  $q$  y se repite el procedimiento.

En ocasiones, las tareas no tienen la misma duración, lo que en cierta forma rompe con la prelación. En estas situaciones, las tareas pueden dividirse en duraciones determinadas y así construir un digrafo equivalente cuyos nodos representan la misma duración. Sobre este digrafo extendido se calcula el rango.

## 7.8 Árboles abarcadores mínimos

Dado un grafo conexo  $G = \langle V, E \rangle$  con distancias (o pesos), un árbol abarcador mínimo es un árbol abarcador de  $G$  tal que la suma total de sus distancias es mínima.

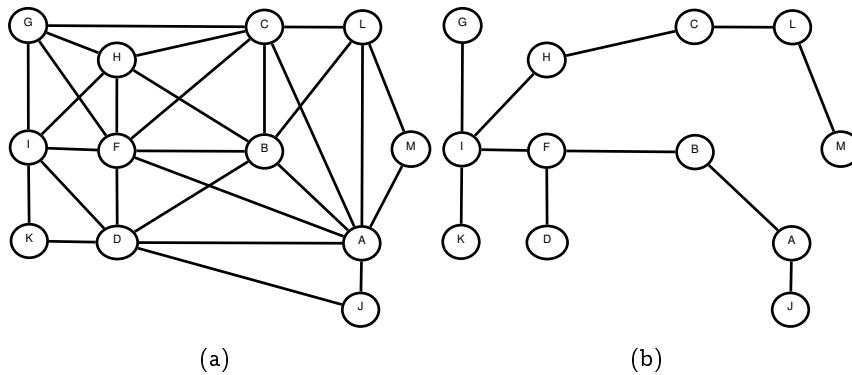


Figura 7.49: Un grafo y su árbol abarcador mínimo

Los árboles abarcadores tienen muy útiles aplicaciones. Supongamos, por ejemplo, la planificación de un sistema ferroviario nacional o de una red de transmisión eléctrica para el país. Ambos proyectos conllevan costes en términos de trabajo, duración y financiamiento. En estas situaciones se puede modelizar un grafo de todas las alternativas de factibilidad según los términos mencionados. El grafo mantendría las posibles vías en función de la geografía, o de la duración de construcción o del coste de financiamiento. En ambas situaciones, un árbol abarcador mínimo nos proporciona la red mínima que en los términos del modelo garantiza completa conectividad.

### 7.8.1 Manejo de los pesos del grafo

Si pretendemos elaborar algoritmos generales para calcular árboles abarcadores mínimos, es decir, que operen para cualquier clase de pesos, entonces debemos separar del algoritmo todo lo que atañe a los pesos. En tal sentido usaremos algunas clases que serán parámetros tipos de un algoritmo para calcular el árbol abarcador mínimo:

1. `Distance<GT>`: clase de acceso al peso de un arco, la cual debe exportar los siguientes atributos:
  - (a) `Distance<GT>::Distance_Type`: el tipo de dato que representa un peso en un arco.
  - (b) `typename Distance<GT>::Distance_Type operator() (typename GT::Arc *a)`: retorna el valor de peso contenido en el arco  $a$ .

- (c) typename Distance<GT>::Max\_Distance: constante estática correspondiente al valor máximo de distancia que un algoritmo consideraría como valor infinito.
- (d) typename Distance<GT>::Zero\_Distance: constante estática correspondiente al elemento neutro de la suma. En la inmensa mayoría de casos, este será el cero.
2. Compare<GT>: clase que efectúa la comparación entre dos pesos y que debe exportar:

```
bool operator () (const typename Distance<GT>::Distance_Type & op1,
 const typename Distance<GT>::Distance_Type & op2) const;
```

para implantar la relación “menor que”.

3. Plus: clase de suma entre distancias implantada mediante el operador:

```
typename Distance<GT>::Distance_Type
Plus::operator () (const typename Distance<GT>::Distance_Type & op1,
 const typename Distance<GT>::Distance_Type & op2) const;
```

Las clases Distance y Compare se conjugan bajo una clase especial de comparación de arcos cuya definición es como sigue:

661a *<Comparador de arcos 661a>*≡ 661b▷

```
template <class GT, class Distance, class Compare>
struct Distance_Compare
{
 bool operator () (typename GT::Arc * a1, typename GT::Arc * a2) const
 {
 return Compare () (Distance () (a1) , Distance () (a2));
 }
};
```

Defines:

*Distance\_Compare*, used in chunks 662a, 668, and 669a.

Una primera forma de conjugar estas clases es implantando el cálculo del coste de un grafo, es decir, la suma de los pesos de todos sus arcos:

661b *<Comparador de arcos 661a>+≡* ◁661a

```
template <class GT, class Distance, class Compare, class Plus, class SA>
typename Distance::Distance_Type total_cost(GT & g)
{
 typename Distance::Distance_Type sum = Distance::Zero_Distance;
 // recorrer todos los arcos y sumar su peso
 for (Arc_Iterator <GT, SA> it(g); it.has_current(); it.next())
 sum = Plus () (sum, Distance () (it.get_current_arc()));

 return sum;
}
```

### 7.8.2 Algoritmo de Kruskal

El algoritmo de Kruskal selecciona los arcos a insertar en  $T$  según su peso, desde el menor al mayor. Para eso los arcos se ordenan previamente, de manera tal que, iterando

sobre ellos (ver § 7.3.10.1 (Pág. 565)), éstos aparezcan ordenadamente. Este ordenamiento previo lo hacemos del siguiente modo:

662a *(ordenar arcos 662a)≡* (663c)  
*g.template sort\_arcs<Distance\_Compare<GT, Distance, Compare> >();*  
*Uses Distance\_Compare 661a.*

Es decir, se invoca el método de ordenamiento de arcos planteado en § 7.3.10.10 (Pág. 577).

El proceso para obtener el árbol abarcador mínimo, luego de tener los arcos ordenados se delinea, a grandes rasgos, de la siguiente manera:

662b *(calcular árbol abarcador mínimo según Kruskal 662b)≡* (663c)  
*// Recorrer arcos ordenados de g hasta que numero de arcos de*  
*// tree sea igual al numero de nodos de g*  
*for (Arc\_Iterator<GT, SA> arc\_itor(g);*  
*tree.get\_num\_arcs() < g.get\_num\_nodes() - 1; arc\_itor.next())*  
*{    // obtenga siguiente menor arco*  
*typename GT::Arc \* arc = arc\_itor.get\_current\_arc();*  
  
*⟨Verificar si nodo origen existe en tree 663a⟩*  
*⟨Verificar si nodo destino existe en tree 663b⟩*  
  
*typename GT::Arc \* arc\_in\_tree =*  
*tree.insert\_arc(tree\_src\_node, tree\_tgt\_node, arc->get\_info());*  
  
*if (Has\_Cycle<GT, SA>() (tree)) // ¿arc\_in\_tree causa ciclo?*  
*{        // hay ciclo ==> hay que remover el arco*  
*tree.remove\_arc(arc\_in\_tree); // eliminar arco y procesar*  
*continue;                       // siguiente arco*  
*}*  
*GT::map\_arcs(arc, arc\_in\_tree);*  
*}*

El ciclo mira los arcos del grafo desde el menor hasta el mayor. Cada arco que es mirado se inserta en tree y, si éste no causa un ciclo, entonces éste es parte del árbol abarcador. El proceso continúa hasta que la cantidad de arcos del árbol sea exactamente la cantidad de nodos del grafo menos uno, lo cual es la indicación de que el árbol ya abarca completamente a los nodos del grafo sin perder su condición de árbol; a partir de este momento cualquier otro arco que se inserte en tree causará un ciclo.

Una gran bondad del algoritmo de Kruskal, aprensible mediante inspección de sus bloques principales, es que puede modificarse con relativa facilidad para que opere concurrentemente. En primer lugar, como sucintamente explicamos con el quicksort en § 3.2.2.9 (Pág. 184), podemos acelerarlo substancialmente si lo modificamos para usar varios procesadores materiales. Más difícil, pero comprobadamente posible, es modificar el bloque *(calcular árbol abarcador mínimo según Kruskal 662b)* para que maneje los bloques internos *⟨Verificar si nodo origen existe en tree 663a⟩* y *⟨Verificar si nodo destino existe en tree 663b⟩* en procesadores diferentes.

Por requerimientos de interfaz, para insertar un arco en tree es necesario que sus nodos ya hayan sido previamente insertados. Este es el rol básico de los bloques *⟨Verificar si nodo origen existe en tree 663a⟩* y *⟨Verificar si nodo destino existe en tree 663b⟩*, del cual comentaremos el primero:

663a *⟨Verificar si nodo origen existe en tree 663a⟩≡* (662b)  
 typename GT::Node \* g\_src\_node = g.get\_src\_node(arc); // origen en g  
 typename GT::Node \* tree\_src\_node; // origen en tree  
 if (not IS\_NODE\_VISITED(g\_src\_node, Aleph::Kruskal)) // ¿está en tree?  
 { // No, crearlo en tree, atarlo al cookie y marcar como visitado  
 NODE\_BITS(g\_src\_node).set\_bit(Aleph::Kruskal, true);  
 tree\_src\_node = tree.insert\_node(g\_src\_node->get\_info());  
 GT::map\_nodes(g\_src\_node, tree\_src\_node);  
}  
else  
 tree\_src\_node = mapped\_node<GT>(g\_src\_node);  
Uses mapped\_node 560b.

Como se ve, el bloque tiene dos flujos: o el nodo ya está insertado y mapeado en tree (flujo del else), o hay que crearlo, marcarlo y mapearlo. El otro bloque es similar pero con el otro extremo del arco:

663b *⟨Verificar si nodo destino existe en tree 663b⟩≡* (662b)  
 typename GT::Node \* g\_tgt\_node = g.get\_tgt\_node(arc);  
 typename GT::Node \* tree\_tgt\_node; // destino en árbol abarcador  
 if (not IS\_NODE\_VISITED(g\_tgt\_node, Aleph::Kruskal))  
 { // No, crearlo en tree, atarlo al cookie y marcar como visitado  
 NODE\_BITS(g\_tgt\_node).set\_bit(Aleph::Kruskal, true);  
 tree\_tgt\_node = tree.insert\_node(g\_tgt\_node->get\_info());  
 GT::map\_nodes(g\_tgt\_node, tree\_tgt\_node);  
}  
else  
 tree\_tgt\_node = mapped\_node<GT>(g\_tgt\_node);  
Uses mapped\_node 560b.

Con todas las estructuras anteriores podemos especificar la rutina de interfaz:

663c *⟨Algoritmo de Kruskal 663c⟩≡*  
 template <class GT, class Distance, class Compare,  
 class SA = Default\_Show\_Arc<GT> > inline  
 void kruskal\_min\_spanning\_tree(GT & g, GT & tree)  
{  
 g.reset\_bit\_nodes(Aleph::Kruskal); // limpiar bits de marcado  
 clear\_graph(tree); // limpia grafo destino  
*⟨ordenar arcos 662a⟩*  
*⟨calcular árbol abarcador mínimo según Kruskal 662b⟩*  
}

Al final de la llamada, tree contiene un árbol abarcador cuyos cookies, tanto de nodos como de arcos, están mapeados al grafo g y viceversa. Las distancias de los arcos son de tipo Distance::Distance\_Type y accedidas por esta clase. Las comparaciones entre distancia las efectúa la clase Compare.

La implantación completa reside en el archivo Kruskal.H.

### 7.8.2.1 Análisis del algoritmo de Kruskal

El tiempo de ejecución estará dado por la suma de los bloques *⟨ordenar arcos 662a⟩* más *⟨calcular árbol abarcador mínimo según Kruskal 662b⟩*.

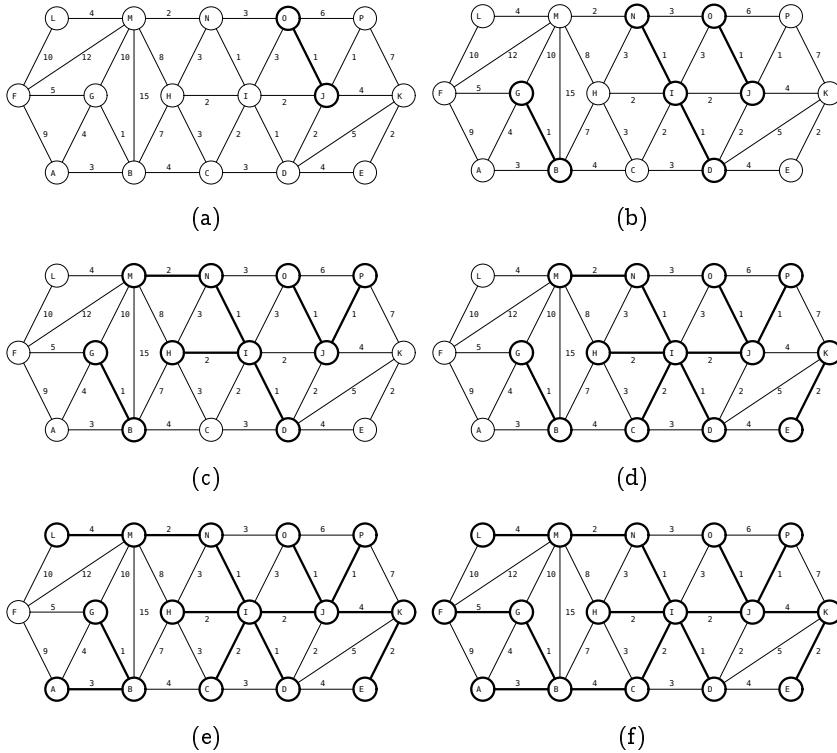


Figura 7.50: Progreso del algoritmo de Kruskal (cada figura corresponde a tres iteraciones)

Sabemos por análisis previos que la mejor manera de ordenar una lista es  $\mathcal{O}(n \lg(n))$ , por lo que el bloque `<ordenar arcos 662a>` exhibe  $\mathcal{O}(E \lg E)$ , esperado o determinista según el método de ordenamiento.

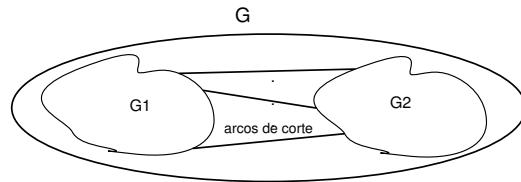
Para el bloque `<calcular árbol abarcador mínimo según Kruskal 662b>` asumimos, como peor caso, que la configuración de arcos es tal que éstos deben revisarse completamente. El `while` requeriría, pues,  $\mathcal{O}(E)$  iteraciones. Dentro del `while` se ejecutan constantemente dos operaciones: la inserción del arco en `tree` y la prueba de aciclicidad `has_cycle()`. La inserción de un arco es  $\mathcal{O}(1)$ , mientras que la invocación a `has_cycle()` (ver § 7.5.6 (Pág. 593)) es  $\mathcal{O}(V)$ , pues, por su condición de árbol o de grafo, parcialmente inconexo, en  $tree \leq V$ . El bloque `<calcular árbol abarcador mínimo según Kruskal 662b>` conlleva entonces  $\mathcal{O}(E) \times \mathcal{O}(V) = \mathcal{O}(EV)$  en el peor de los casos.

El algoritmo de Kruskal consume, para el peor caso,  $\mathcal{O}(E \lg E) + \mathcal{O}(EV) = \mathcal{O}(\max(E \lg E, EV))$ , lo cual lo hace atractivo para grafos esparcidos.

### 7.8.2.2 Correctitud del algoritmo de Kruskal

Para las pruebas de correctitud nos es conveniente introducir la idea de corte de un grafo, lo cual no es otra cosa que una partición de  $G$  en dos subgrafos disjuntos  $G_1$  y  $G_2$ . Los arcos eliminados de  $G$ , es decir, aquellos que conectan nodos de  $G_1$  con nodos de  $G_2$ , se denominan “arcos de corte”.

Robert Sedgewick [156] ha caracterizado dos propiedades sobre los árboles abarcadores, las cuales fungen de instrumentos de prueba. La primera de ellas ataña a una arborescencia de árboles abarcadores parciales y se describe bajo el lema siguiente.

Figura 7.51: Esquema genérico de corte de un grafo  $G$ 

**Lema 7.4 (Lema del corte - Sedgewick 2002 [156])** Sea  $G = \langle V, E \rangle$  un grafo y  $T = \langle V, E' \rangle | E' \subset E$  un árbol abarcador mínimo de  $G$ . Sea  $\langle G_1, G_2 \rangle$  un corte cualquiera de  $G$ . Entonces el arco de cruce mínimo entre los posibles de cruce entre  $G_1$  y  $G_2$  pertenece al árbol abarcador mínimo.

**Demostración (por contradicción)** Supongamos un corte cualquiera  $\langle G_1, G_2 \rangle$  y la existencia de un árbol abarcador mínimo  $T = \langle V, E' \rangle$  cuyo arco de cruce no es el mínimo.

Sean  $T_1$  y  $T_2$  los componentes de  $T$  según el corte. El asunto se pictoriza en la figura 7.52. Sean  $a$  el arco de cruce entre  $T_1$  y  $T_2$ , el cual, como ya hemos dicho, no es el mínimo

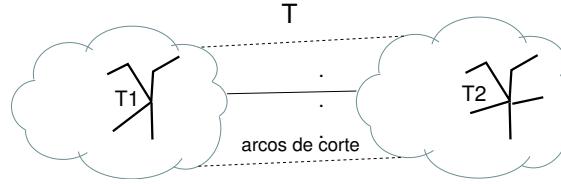


Figura 7.52: Esquema general del lema del corte

de los de cruce entre  $G_1$  y  $G_2$ . Puesto que  $T_1$  y  $T_2$  son acíclicos, podemos suprimir el arco de cruce  $a$  y substituirlo por el de cruce mínimo  $a_{\min}$  y así crear un árbol abarcador  $T'' = \langle V, E'' \rangle = \langle V, E' - \{a\} \cup \{a_{\min}\} \rangle$ . Puesto que  $\text{peso}(a_{\min}) < \text{peso}(a) \Rightarrow \sum_{e \in E''} \text{peso}(e) < \sum_{e \in E'} \text{peso}(e)$ , lo que contradice la premisa inicial de que  $T$  es el árbol abarcador mínimo de  $G$ . El lema es pues cierto  $\square$

El segundo lema ataña a la adición de arcos que causen ciclos y su consecuente supresión.

**Lema 7.5 (Lema del ciclo - Sedgewick 2002 [156])** Sea  $G$  un grafo conexo cualquiera y  $T$  su árbol abarcador mínimo. Sea  $G'$  un grafo construido por añadidura de un arco  $e$  a  $G$ . Entonces, añadir  $e$  a  $T$  y eliminar el arco de mayor peso en el ciclo resulta en un árbol abarcador  $T'$  que es mínimo.

**Demostración** Si  $e$  es el arco con mayor peso del ciclo, entonces éste no puede ser parte de  $T'$  porque sino  $T'$  no sería mínimo (existiría otro arco de menor peso).

De lo contrario, sea  $e_{\max}$  el arco con mayor peso. Eliminar  $e_{\max}$  de  $T$  lo corta en dos árboles disjuntos los cuales, según el lema del ciclo 7.4 están conectados en  $T'$  por el mínimo arco de cruce.  $T'$  es por tanto mínimo  $\square$

**Proposición 7.3 (Sedgewick - 2002 [156])** Sea  $G$  un grafo conexo cualquiera. Entonces el algoritmo de Kruskal encuentra un árbol abarcador mínimo.

### Demostración (por inducción sobre $n = |V|$ )

1.  $n \leq 2$ : la prueba es inmediata.
  2.  $n > 2$ : asumimos que la proposición es cierta para un grafo  $G_n$  de  $n$  nodos. Así, el algoritmo de Kruskal construye un árbol abarcador mínimo  $T_n$  de  $n - 1$  arcos.
- Ahora estudiaremos la validez del algoritmo para un grafo  $G_{n+1}$  de  $n + 1$  nodos.

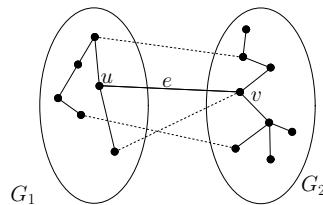


Figura 7.53: Esquema genérico de partición del grafo

El algoritmo de Kruskal mantiene una arborescencia de árboles abarcadores, los cuáles por la hipótesis inductiva son mínimos.

Ahora miremos la inserción del arco cuando la arborescencia tiene  $n - 1$  arcos. En este momento, podemos considerar una partición de  $G_{n+1}$  en  $G_1 \cup G_2$  y dos árboles abarcadores  $T_1$  y  $T_2$ . Así, al seleccionar  $e$  e insertar un arco  $e = u \rightarrow v$ , ocurre uno entre los siguientes casos:

- (a) Se crea un ciclo, en cuyo caso  $e$  se elimina. Puesto que los arcos fueron vistos ordenadamente,  $e$  es el arco con mayor peso dentro del ciclo causado. De este modo, al eliminarlo del árbol, según el lema del ciclo (7.5 (Pág. 665)),  $T_1$  y  $T_2$  se preservan y estos son árboles abarcadores mínimos.
- (b) No se crea un ciclo, en cuyo caso la adición de  $e$  une a los componentes inconexos  $T_1$  y  $T_2$  creando un solo componente  $T$ . Ahora bien, puesto que  $e$  es visto ordenadamente,  $e$  es el mínimo arco de cruce entre  $G_1$  y  $G_2$  y, por el lema del corte (7.4 (Pág. 664)),  $T$  es un árbol abarcador mínimo

La proposición es cierta para todo grafo ■

### 7.8.3 Algoritmo de Prim

El segundo algoritmo para construir un árbol abarcador se basa en una modificación del recorrido en amplitud, en la cual en lugar de emplear una cola FIFO, se usa una cola de prioridad que pondera los pesos de los arcos.

#### 7.8.3.1 Algoritmo genérico

**Algoritmo 7.4** (Algoritmo de Prim para cálculo de árbol abarcador mínimo)

La entrada es un un grafo conexo  $G$ .

La salida es un grafo conexo  $T$  correspondiente al árbol abarcador mínimo de  $G$ .

1. Sea  $q$  una cola de prioridad de arcos ordenada según su peso y  $T = \emptyset$

2. Seleccione un nodo  $v$  y meta en  $q$  todos sus arcos. Marque a  $v$  como visitado
3. while not  $q.is\_empty()$  y  $T$  no abarque a  $G$ 
  - (a)  $a = q.get();$
  - (b) if (los nodos de  $a$  están visitados)  $\Rightarrow$   
continue;
  - (c) Sea  $t$  el nodo no visitado de  $a$
  - (d) Insertar  $a$  en  $T$  y marque a  $y t$  como visitados
  - (e) Meter en  $q$  todos los arcos adyacentes a  $t$  que aún no hayan sido visitados.

Si la cola de prioridad se instrumenta mediante un heap, entonces este algoritmo se ejecuta en tiempo  $\mathcal{O}(E \lg(E))$  y consume espacio  $\mathcal{O}(E)$ .

Como se puede apreciar, el algoritmo de Prim es estructuralmente idéntico al recorrido en amplitud (§ 7.5.4 (Pág. 589)), con la diferencia de que en lugar de emplear una cola FIFO se emplea una de prioridad, la cual, a su vez, subyace sobre el heap.

### 7.8.3.2 Interfaz del algoritmo de Prim

El algoritmo de Prim se invoca mediante las siguiente interfaz:

*⟨Algoritmo de Prim 667⟩* ≡

```
template <class GT, class Distance,
 class Compare = Aleph::less<typename Distance::Distance_Type>,
 class SA = Default_Show_Arc<GT> > inline
void prim_min_spanning_tree(GT & g, typename GT::Node * first, GT & tree)
{
 ⟨Declaración de cola de prioridad 671a⟩
 ⟨Inicializar heap 671b⟩
 ⟨calcular árbol abarcador mínimo según Prim 671c⟩
}
```

Funcionalmente, esta interfaz es exacta a la del algoritmo de Kruskal.

### 7.8.3.3 Heap exclusivo de arcos mínimos

Existe una técnica, presentada por Sedgewick [156], que reduce substancialmente el tamaño del heap a no más de  $V$  entradas. Esto implica que el coste en tiempo se reduce a  $\mathcal{O}(\lg V)$  y el de espacio a  $\mathcal{O}(V)$ , en lugar de los respectivos  $\mathcal{O}(\lg E)$  y  $\mathcal{O}(E)$ .

**Definición 7.7 (Heap exclusivo de arcos mínimos)** Un heap exclusivo de arcos mínimos es un heap que guarda arcos de un grafo ordenados por su peso de manera tal que en ningún momento pueden existir dos arcos con el mismo nodo destino.

Para un grafo  $G = \langle V, E \rangle$ , es imposible que el heap contenga más de  $V - 1$  arcos, pues si no habría dos arcos con el mismo nodo destino.

Por su carácter de heap se garantiza que la extracción de un arco se corresponderá con el de peso mínimo entre todos los incluidos en el conjunto.

Un heap tal como el que acabamos de definir es implantable mediante el TAD BinHeap<Key> estudiado en § 4.7 (Pág. 282). Esta clase de heap, que llamaremos ArcHeap, se define en el archivo *(archheap.H 668a)*:

668a *(archheap.H 668a)≡*

```
template <class GT, class Distance, class Compare, class Access_Heap_Node>
struct ArcHeap
: public BinHeap<typename GT::Arc*, Distance_Compare<GT,Distance,Compare> >
{
 (Atributos públicos de ArcHeap 668b)
};
```

Defines:

*ArcHeap*, used in chunks 670b, 671a, and 679.

Uses *Distance\_Compare* 661a.

El archivo define la clase ArcHeap, la cual implementa un heap exclusivo de arcos mínimos pertenecientes a un grafo de tipo GT, con acceso a los pesos de los arcos mediante la clase Distance y comparación entre las distancias a través de la clase Compare. El sentido del parámetro tipo Access\_Arc será explicado prontamente.

Como se puede observar de la declaración, ésta hereda la implementación de BinHeap<Key>, pues el orden de salida es uno de los requerimientos principales cuando se extraiga algún arco. De este modo, el BinHeap<Key> manejará nodos del siguiente tipo:

668b *(Atributos públicos de ArcHeap 668b)≡*

(668a) 669a▷

```
typedef typename BinHeap <typename GT::Arc*,
 Distance_Compare<GT, Distance, Compare> >::Node Node;
```

Uses *Distance\_Compare* 661a.

En este heap la operación de inserción de un arco se resume bajo el siguiente algoritmo:

#### Algoritmo 7.5 (Inserción genérica en un heap exclusivo)

La entrada es un arco  $a$  que relaciona dos nodos  $s$  y  $t$ , origen y destino, respectivamente.  $s$  es un nodo que ya pertenece a un árbol abarcador y que ha sido, por tanto, visitado.  $t$  no ha sido visitado y no pertenece al árbol abarcador.

1. Busque en el heap un arco  $a'$  tal que su nodo destino sea  $t$ .

2. Si  $a'$  ha sido encontrado  $\Rightarrow$

- (a) Si  $a' > a \Rightarrow$  (según lo que arroje la clase Distance)

- i. Elimine  $a'$  del heap
- ii. Inserte  $a$  en el heap

de lo contrario  $\Rightarrow$

- iii. Inserte  $a$  en el heap

Para que esta operación se lleve a cabo en  $\mathcal{O}(\lg V)$  es necesario encontrar una manera rápida de recuperar  $a'$  que no comprometa el desempeño del heap. Podemos usar una tabla de nodos destinos de arcos contenidos en el heap, la cual podría instrumentarse mediante una tabla hash o un árbol abarcador.

En nuestra implementación, un poco en detrimento de la claridad, anotaremos en cada nodo del grafo, a través del cookie, un puntero hacia el nodo del heap dentro del ArcHeap.

Para acceder al cookie nos valemos de una clase de acceso, denominada Access\_Heap\_Node, cuyo operador () debe corresponderse con el siguiente prototipo:

```
template <class GT, class Distance, class Compare> typename
BinHeap<typename GT::Arc*,Distance_Compare<GT,Distance,Compare> >::Node
*& Access_Heap_Node::operator () (typename GT::Node * p);
```

El operador retorna un puntero al nodo del heap que contiene un arco incluido en el ArcHeap cuyo nodo destino es p, o NULL, si el heap no contiene ningún arco con p como nodo destino. Es responsabilidad del usuario de ArcHeap el que Access\_Heap\_Node::operator () retorne NULL la primera vez en que se inserte un arco con un nodo destino. Es muy importante que se retorne una referencia a un puntero y no una copia; de modo tal que se pueda cambiar el valor del apuntador.

Ahora podemos implantar la inserción según el algoritmo 7.5:

669a *(Atributos públicos de ArcHeap 668b) +≡* (668a) ▷ 668b 669b ▷

```
void put_arc(typename GT::Arc * arc, typename GT::Node * tgt)
{
 Node *& heap_node = Access_Heap_Node () (tgt);
 if (heap_node == NULL) // ¿existe arco insertado en el heap?
 {
 // No ==> crear nuevo nodo para el heap y asignarle arco
 heap_node = new Node;
 heap_node->get_key() = arc;
 this->insert(heap_node);

 return;
 }
 // dos arcos con igual destino ==> tomar menor; descartar mayor
 typename GT::Arc *& arc_in_heap = heap_node->get_key();

 // ¿arc_in_heap tiene distancia menor que arc?
 if (Distance_Compare <GT, Distance, Compare> () (arc_in_heap, arc))
 return; // antiguo arco permanece en el heap; nuevo se ignora

 // arc_in_heap será el arco recién insertado arc
 arc_in_heap = arc; // cambia el arco
 update(heap_node); // actualiza el heap con el nuevo peso de arc
}
```

Defines:

put\_arc, used in chunks 671, 680c, and 681b.

Uses Distance\_Compare 661a.

Para un grafo  $G = \langle V, E \rangle$ , el coste de una inserción de arcos es  $\mathcal{O}(\lg V)$ , pues la substitución de un arco acota el heap a un máximo de  $V - 1$  elementos.

La eliminación es tan compleja como la inserción, pues aparte del manejo de memoria hay que asegurar que el arco eliminado no se relacione con su nodo destino:

669b *(Atributos públicos de ArcHeap 668b) +≡* (668a) ▷ 669a

```
typename GT::Arc * get_min_arc()
{
 Node * heap_node = this->getMin();
 typename GT::Arc * arc = heap_node->get_key();
 // seleccionar nodo que retorne la clase de acceso
```

```

typename GT::Node * p = (typename GT::Node*) arc->src_node;
if (Access_Heap_Node () (p) != heap_node)
 p = (typename GT::Node*) arc->tgt_node;

Access_Heap_Node () (p) = NULL;
delete heap_node;

return arc;
}

```

Defines:

get\_min\_arc, used in chunks 671c and 681a.

Puesto que el heap ya está acotado a un máximo de  $V - 1$  arcos, la eliminación es  $\mathcal{O}(\lg V)$ .

#### 7.8.3.4 Inicialización del algoritmo de Prim

Lo primero que debemos hacer es definir la estructura que contendrá el cookie de un nodo, de modo tal que podamos usar el heap exclusivo diseñado en la subsección precedente:

670a

*(Definiciones de Prim 670a)≡*

```

template <class GT> struct Prim_Info
{
 typename GT::Node * tree_node; // imagen en el árbol abarcador
 void * heap_node; // puntero en el heap exclusivo
};

```

670b▷

El campo `tree_node` guarda la imagen del nodo dentro del árbol abarcador, mientras que `heap_node` la dirección dentro del heap exclusivo que guarda el mínimo arco al nodo en cuestión como destino. `heap_node` se declara como `void*` debido a que tenemos diferentes cruces de tipos involucrados. El constructor asegura que cuando se aparte la memoria los campos se inicien automáticamente en `NULL`.

Dado un puntero a nodo `p`, el acceso a estos campos se facilita mediante los siguientes macros `PRIMINFO()`, `TREENODE()` y `HEAPNODE()`.

Ahora, según los requerimientos del tipo `ArcHeap`, debemos especificar el acceso al campo `heap_node`:

670b

*(Definiciones de Prim 670a)+≡*

◁670a

```

template <class GT, class Distance, class Compare> struct Prim_Heap_Info
{
 typedef typename ArcHeap<GT, Distance, Compare, Prim_Heap_Info>::Node Node;
 Node *&& operator () (typename GT::Node * p)
 {
 return (Node*&&) HEAPNODE(p);
 }
};

```

Defines:

`Prim_Heap_Info`, used in chunk 671a.

Uses `ArcHeap` 668a.

La inicialización del cookie del nodo se efectúan mediante `Operate_On_Nodes()`, invocada al principio del algoritmo; similar se hace para liberar la memoria al final del algoritmo.

Lo que nos falta por definir en la inicialización del algoritmo de Prim concierne al heap exclusivo. En primer lugar, su declaración:

671a *Declaración de cola de prioridad 671a*≡ (667)  
`typedef Prim_Heap_Info<GT, Distance, Compare> Acc_Heap;  
 ArcHeap<GT, Distance, Compare, Acc_Heap> heap;`  
 Uses ArcHeap 668a and Prim\_Heap\_Info 670b.

En todo momento, la operación `heap.get_min_arc()` nos arroja el arco con el menor peso entre todos los incluidos dentro del heap. Esta es, hasta los momentos, la única diferencia con la cola tradicional empleada en el recorrido en amplitud estudiado en § 7.5.10 (Pág. 605).

En segundo lugar, la inserción de los arcos adyacentes al nodo de inicio:

671b *Inicializar heap 671b*≡ (667)  
`NODE_BITS(first).set_bit(Aleph::Prim, true); // visitado  
 TREENODE(first) = tree.insert_node(first->get_info());  
 // meter en heap arcos iniciales del primer nodo  
 for (Node_Arc_Iterator<GT,SA> it(first); it.has_current(); it.next())  
 {  
 typename GT::Arc * arc = it.get_current_arc();  
 ARC_BITS(arc).set_bit(Aleph::Prim, true);  
 heap.put_arc(arc, it.get_tgt_node());  
 }  
 Uses put_arc 669a.`

### 7.8.3.5 El algoritmo de Prim

Ahora estamos listos para construir iterativamente el árbol abarcador en el orden establecido por la cola, lo cual se hace de la siguiente manera:

671c *calcular árbol abarcador mínimo según Prim 671c*≡ (667)  
`while (tree.get_num_nodes() < g.get_num_nodes()) // mientras tree no abarque g  
{ // obtenga siguiente menor arco  
 typename GT::Arc * min_arc = heap.get_min_arc();  
 typename GT::Node * src = g.get_src_node(min_arc);  
 typename GT::Node * tgt = g.get_tgt_node(min_arc);  
 if (IS_NODE_VISITED(src, Aleph::Prim) and IS_NODE_VISITED(tgt, Aleph::Prim))  
 continue; // este arco cerraría un ciclo en el árbol  
  
 typename GT::Node * tgt_node = IS_NODE_VISITED(src, Aleph::Prim) ? tgt : src;  
 TREENODE(tgt_node) = tree.insert_node(tgt_node->get_info());  
 NODE_BITS(tgt_node).set_bit(Aleph::Prim, true);  
  
 // insertar en heap arcos no visitados de tgt_node  
 for (Node_Arc_Iterator<GT, SA> it(tgt_node); it.has_current();  
 it.next())  
 {  
 typename GT::Arc * arc = it.get_current_arc();  
 if (IS_ARC_VISITED(arc, Aleph::Prim))  
 continue;  
  
 ARC_BITS(arc).set_bit(Aleph::Prim, true); // marcar arco`

```

typename GT::Node * tgt = it.get_tgt_node();
if (IS_NODE_VISITED(tgt, Aleph::Dijkstra))
 continue; // nodo visitado ==> causará ciclo

 heap.put_arc(arc, tgt);
}

// insertar nuevo arco en tree y mapearlo
typename GT::Arc * tree_arc = tree.insert_arc(TREENODE(src), TREENODE(tgt),
 min_arc->get_info());
GT::map_arcs(min_arc, tree_arc);
}

```

Uses `get_min_arc` 669b and `put_arc` 669a.

Como veremos posteriormente, el árbol abarcador resultante de este proceso es mínimo.

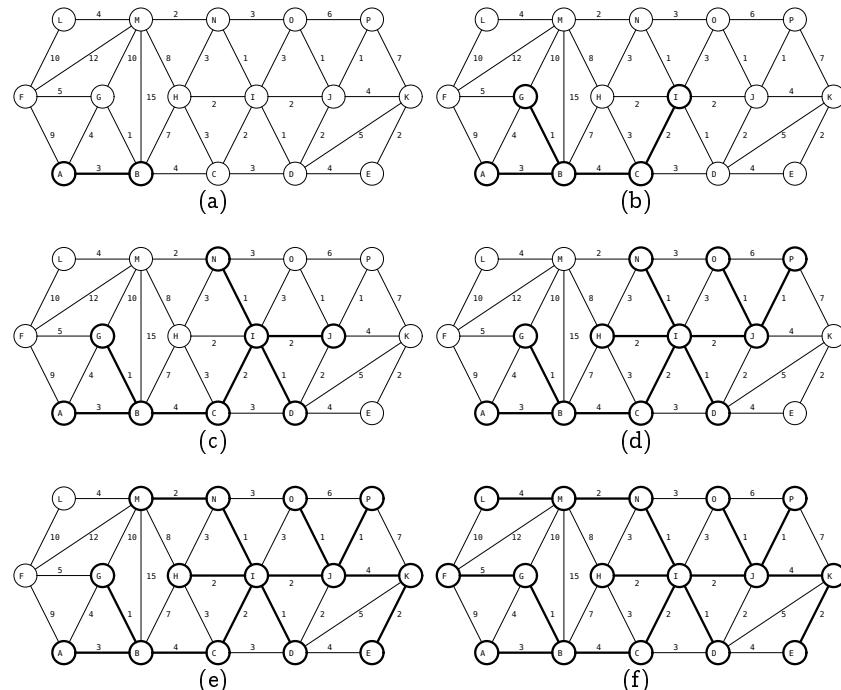


Figura 7.54: Progreso del algoritmo de Prim (cada figura corresponde a tres iteraciones)

Hay una diferencia crucial en la manera en que el algoritmo de Prim trata con la propiedad del ciclo (7.5 pág. 665) y que nos permite evadir una prueba de aciclicidad. Puesto que el árbol abarcador `tree` siempre es conexo, podemos, por simple inspección de visita de los nodos de un arco procesado, determinar si la inclusión del nuevo arco en `tree` causará un ciclo. Si ambos nodos ya han sido visitados, entonces la inserción del arco causará un ciclo; de lo contrario, el arco forma parte del árbol abarcador mínimo. Con el algoritmo de Kruskal no podemos hacer esta consideración porque `tree` es inconexo.

#### 7.8.3.6 Análisis del algoritmo de Prim

La inicialización para apartar los contenidos de los cookies y su liberación al final, requieren barrer todos los nodos del grafo. Ambos son, por tanto,  $\mathcal{O}(V)$ .

El bloque `<Iniciar heap 671b>` recorre los arcos del nodo origen. En el muy extremo peor caso en el que el nodo esté conectado al resto de los nodos `<Iniciar heap 671b>` es

$\mathcal{O}(V)$ .

Las tres fases anteriores son, en conjunto,  $\mathcal{O}(V)$ .

En cuanto al bloque *<calcular árbol abarcador mínimo según Prim 671c>*, éste es un poco más sutil de analizar porque la cantidad de iteraciones del while depende de lo que suceda con el for interno, el cual, a su vez, tampoco tiene una cantidad de repeticiones exacta. Si examinamos con cuidado el bloque, entonces podemos percatarnos de que debemos indagar dos cantidades:

1. El número de veces que se accede el heap exclusivo, el cual nos proporciona la complejidad del while combinado con el for.

Bajo el supuesto de que el grafo es conexo, es claro que todos los nodos deben ser mirados por el algoritmo, pues éste se inicia desde el primer arco del nodo origen y culmina hasta que todos los nodos estén abarcados, lo cual es la condición de parada del while. Cada nuevo nodo inspeccionado acarrea su inclusión en el árbol abarcador<sup>20</sup> y la consecuente inserción de todos los arcos no visitados adyacentes al nodo recién procesado. Cuando ocurre la inserción del último nodo, ya todos los arcos del grafo han sido visitados y, como éstos son marcados, se sacan del heap una sola vez.

Vemos pues que se efectúan a lo sumo  $E$  inserciones de arco en el heap. En cuanto a la cantidad de extracciones debemos notar que, puesto que tratamos con un heap exclusivo, éste saca arcos durante la inserción en un coste  $\mathcal{O}(1)$ . Ocurren entonces a lo sumo  $V$  extracciones de arcos, lo cual nos arroja una complejidad de  $\mathcal{O}(V + E) = \mathcal{O}(E)$  para la cantidad de repeticiones que realizan el while y el for combinados.

Si en lugar de un heap exclusivo tratásemos con un heap tradicional, entonces haríamos  $E$  extracciones, tendríamos un algoritmo más lento pero con la misma complejidad iterativa.

2. El tamaño máximo o promedio del heap, el cual nos proporciona el coste de operar con él. Puesto que usamos un heap exclusivo, es seguro, de lo que aprehendimos en la definición 7.7 (§ 7.8.3.3 (Pág. 667)) que su tamaño no excede de  $V - 1$  elementos, lo cual nos arroja un coste de operación de  $\mathcal{O}(\lg V)$  por cada la inserción o eliminación.

El coste de una operación sobre el heap exclusivo es, pues,  $\mathcal{O}(\lg V)$ .

De las cantidades recientemente calculadas podemos concluir que el coste del bloque *<calcular árbol abarcador mínimo según Prim 671c>* es  $\mathcal{O}(E) \times \mathcal{O}(\lg V) = \mathcal{O}(E \lg V)$ .

Los cuatro bloques involucrados contabilizan  $\mathcal{O}(V) + \mathcal{O}(E \lg V) = \mathcal{O}(E \lg V)$ , que es el desempeño definitivo en tiempo del algoritmo de Prim.

Es importantísimo destacar que si usásemos un heap tradicional para guardar los arcos, entonces el coste de una operación sobre el heap estaría acotado por  $\mathcal{O}(\lg E)$ , el cual acotaría al algoritmo de Prim a un desempeño de  $\mathcal{O}(E \lg E)$ , un coste substancialmente mayor al que nos proporciona el heap exclusivo. Análogamente, tendríamos un coste en espacio de  $\mathcal{O}(E)$  versus el  $\mathcal{O}(V)$ . El heap exclusivo, aunque complejo, no es un capricho, pues su uso se recompensa con creces, tanto en tiempo como en espacio.

El algoritmo de Prim es el método de preferencia para grafos densos.

---

<sup>20</sup>Recordemos que en este algoritmo, el árbol abarcador siempre es conexo, lo que asegura que no se creará un ciclo cuando se inserte en el árbol un nodo no visitado.

### 7.8.3.7 Correctitud del algoritmo de Prim

**Proposición 7.4** (Sedgewick - 2002 [156]) Sea  $G$  un grafo conexo cualquiera. Entonces el algoritmo de Prim encuentra un árbol abarcador mínimo.

**Demostración (por inducción sobre  $n = |V|$ )**

1.  $n < 2$ : En este caso, el heap retorna el menor arco del nodo inicial y construye un árbol parcial de dos nodos el cual es, con certitud, mínimo.
2.  $n > 2$ : ahora asumimos que la proposición cierta para un árbol abarcador mínimo parcial de  $n$  nodos de un grafo  $G_n$ .

A diferencia del algoritmo de Kruskal, el de Prim siempre maneja un solo árbol abarcador mínimo parcial (que por supuesto es conexo). Así, a lo largo de la ejecución, el corte de  $G_{n+1}$  puede interpretarse como  $G_i \cup \{v\}$ , donde  $i$  es la  $|T|$  (véase figura 7.55). Por la hipótesis inductiva, el algoritmo de Prim encuentra un árbol abarcador mínimo  $T_i$  por cada  $G_i \mid i < n$ .

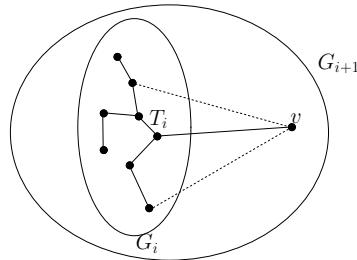


Figura 7.55: Esquema de corte del algoritmo de Prim

Cuando el árbol abarcador mínimo contiene  $n$  nodos, el corte llega a ser  $G_n \cup \{v\}$ . Al extraer del heap un arco, éste da el menor arco de entre los restantes, el cual, al insertarlo en  $T_n$ , nos da dos posibilidades:

- (a) Que el arco forme un ciclo, en cuyo caso, por el lema del ciclo (7.5 (Pág. 665)), éste es el mayor entre todos los arcos que conforman el ciclo. Al eliminarlo de  $T_{n+1}$  se obtiene árbol previo  $T_n$  que es mínimo.
- (b) Que el arco no forme un ciclo, en cuyo caso, planteando un corte  $G_n \cup \{b\}$ , el arco es el mínimo entre los posibles de cruce (fue extraído del heap) y, como lo sabemos del lema del corte (7.4 (Pág. 664)), el nuevo árbol  $T_{n+1}$  es mínimo.

La proposición es pues cierta para todos los nodos de grafo ■

## 7.9 Caminos mínimos

En grafos ponderados, el problema del camino mínimo consiste en encontrar un camino entre un par de nodos tal que la suma total de todos los pesos sea mínima. Como ejemplos podemos considerar:

1. Dadas ciudades y distancias entre carreteras, encontrar el camino de menor distancia entre un par de ciudades.

2. Dadas ciudades y duraciones en horas de vuelo en avión y en escalas, según ofertas y disponibilidades de aerolíneas, conseguir el itinerario que lleve de una ciudad a otra en el menor tiempo posible.

Si cambiamos duraciones por costes económicos, entonces podemos plantear el encontrar el itinerario económico más barato.

3. Dada una red de computadoras, se desea encontrar el camino más corto de nodos intermedios por donde puedan circular los paquetes en el menor tiempo posible.

Análogamente, si se trata de una red telefónica, entonces, a efectos de prestar buen servicio y, a la vez, de ahorrar costes, se desea encontrar el camino más corto entre conmutadores intermedios de tal manera que se mejoren parámetros como latencia de transmisión (fundamental para la conversación) o cantidad de llamadas simultáneas que pueden efectuarse en la red.

4. Un robot, en función de sus mecanismos de visión, debe planificar un trayecto desde un punto a otro. El terreno y sus obstáculos se modeliza mediante un grafo. En este caso, para ahorrar tiempo y energía se calcula el camino de mínima distancia entre los puntos.

Hay muchas más situaciones en las que puede aparecer este problema. Para que éste tenga sentido, el grafo debe ser conexo o que al menos exista un camino entre un par de nodos, cuestión última que podemos determinar mediante la rutina `test_for_path()` estudiada en § 7.5.7.1 (Pág. 596).

Los tres algoritmos de caminos mínimos que estudiaremos en esta sección operan para digrafos. Sólo el algoritmo de Dijkstra opera para grafos.

Normalmente, por razones de abstracción, es plausible tener pesos negativos en un grafo. En este sentido, en el estudio y cálculo de caminos mínimos hay un absurdo matemático y posiblemente físico, del cual debemos tener especial cuidado. Se trata de los ciclos negativos. Son absurdos porque su existencia en el grafo plantea un ciclo que infinitamente recorrido sería mínimo.

### 7.9.1 Algoritmo de Dijkstra

En 1959, Edsger W. Dijkstra, posiblemente uno de los científicos de las ciencias computacionales más grandiosos de todos los tiempos, hizo público un algoritmo para encontrar todos los caminos más cortos entre un nodo de inicio y el resto de los del grafo.

El algoritmo de Dijkstra puede emplearse para grafos o digrafos. Es bastante adecuado para grafos euclidianos.

Todo lo pertinente al cálculo de caminos mínimos según el algoritmo de Dijkstra subyace en el archivo `Dijkstra.H`.

Hay una gran reminiscencia entre el algoritmo de Dijkstra y el de Prim expresada por el hecho de que ambos se basan en una exploración de grafo según las prioridades. La diferencia esencial subyace en que el algoritmo de Dijkstra usa valores parciales tanto en los nodos como en los arcos.

Dado un nodo inicial  $v_0$ , cual representa el nodo origen de un camino mínimo, los valores de los que hablamos y a los cuales se refiere el algoritmo de Dijkstra, se clasifican de la siguiente manera:

- Para los nodos: dado un nodo cualquiera  $v_i$ ,  $\text{Acc}(v_i)$  representa la mínima distancia acumulada desde  $v_0$  hasta  $v_i$ .

Inicialmente,  $\text{Acc}(v_0) = 0$ .

- Para los arcos: dado un arco cualquiera  $e$  entre los adyacentes a un nodo  $v_i$ ,  $\text{Pot}(e)$  designa la “distancia potencial” acumulada a cubrir desde el nodo  $v_0$  hasta el nodo destino de  $e$ .

Si  $d(e)$  designa la distancia (o peso) del arco  $e$ , entonces, dado un nodo  $v_i$ :

$$\text{Pot}(e) = \text{Acc}(v_i) + d(e) \quad (7.7)$$

Esta operación se efectúa para cada arco de un nodo de tránsito durante la ejecución del algoritmo.

Inicialmente, para el nodo de inicio  $v_0$ ,  $\forall e$  adyacente a  $v_0$ ,  $\text{Pot}(e) = d(e)$ , pues la distancia acumulada desde  $v_0$  es nula ( $\text{Acc}(v_0) = 0$ ).

Similar al algoritmo de Prim, el algoritmo de Dijkstra utiliza un heap exclusivo (§ 7.8.3.3 (Pág. 667)) según los potenciales de arcos. Así, el algoritmo de Dijkstra es estructuralmente igual al de Prim; las operaciones sobre el heap no exceden de  $\mathcal{O}(\lg V)$  en tiempo y de  $\mathcal{O}(V)$  en espacio. Sin embargo, se requiere un poco más de cuidado por la información adicional en los nodos y los los arcos. El uso del heap exclusivo para el algoritmo de Dijkstra será presentado en § 7.9.1.2 (Pág. 679).

El prototipo de nuestra implementación del algoritmo de Dijkstra se define del siguiente modo:

```

676 <Prototipo del algoritmo de Dijkstra 676>≡ 682 ▷
 template <class GT, class Distance,
 class Compare = Aleph::less<typename Distance::Distance_Type>,
 class Plus = Aleph::plus<typename Distance::Distance_Type>,
 class SA = Default_Show_Arc<GT> > inline
void dijkstra_min_paths(GT & g, typename GT::Node * start_node, GT & tree)
{
 <algoritmo de Dijkstra 680b>
}

```

`g` es el grafo (o digrafo) sobre el cual se desea calcular un árbol abarcador de distancias mínimas. `start_node` es el nodo inicial, el cual puede considerarse raíz del árbol abarcador. Finalmente, `tree` es el árbol abarcador de caminos mínimos con inicio (o raíz) en `start_node`. Luego de la ejecución, `g` y `tree` están mapeados mediante sus cookies.

En cuanto a los tipos parametrizados, GT es el tipo del grafo y del árbol abarcador de distancias mínimas. Distance es una clase que accede a la distancia almacenada en el arco según la índole con que se defina el peso. Compare es una clase de comparación entre tipos typename Distance::Distance\_Type. Plus es una clase que efectúa la suma algebraica de distancias. Por último, SA es un filtro de arcos.

#### 7.9.1.1 Distancia acumulada en los nodos

La distancia acumulada puede guardarse mediante el cookie del nodo. Debemos apartar un bloque de memoria en dónde almacenar el acumulado  $\text{Acc}(v_i)$ . Guardaremos pues por

cada nodo, la información definida por el siguiente registro:

677a *(Información por nodo en Dijkstra 677a)≡* 679a▷

```
template <class GT, class Distance>
struct Dijkstra_Node_Info
{
 typename GT::Node * tree_node; // nodo árbol abarcador
 typename Distance::Distance_Type dist; // distancia acumulada
 void * heap_node;

 Dijkstra_Node_Info()
 : tree_node(NULL), dist(Distance::Zero_Distance), heap_node(NULL) {}

};
```

`tree_node` es, como su comentario lo indica, la imagen del nodo en el árbol abarcador de caminos mínimos. `dist` es el acumulado  $\text{Acc}(v_i)$  desde `start_node`. `heap_node` es un puntero a un nodo dentro del heap exclusivo ordenado por potenciales (ver § 7.8.3.3 (Pág. 667)).

La manipulación de los campos anteriores se realiza a través de los siguientes macros:

677b *(Definición de macros para algoritmo de Dijkstra 677b)≡* 679b▷

```
// conversión de cookie a Node_Info
#define DNI(p) ((Dijkstra_Node_Info<GT,Distance>*) NODE_COOKIE((p)))
// Acceso al nodo del árbol en el grafo
#define TREENODE(p) (DNI(p)->tree_node)
#define ACC(p) (DNI(p)->dist) // Acceso a la distancia acumulada
#define HEAPNODE(p) (DNI(p)->heap_node)
```

Antes de ejecutar el algoritmo, se requiere inicializar los nodos y, como veremos prontamente, los arcos. Esto lo ejecutamos del siguiente modo:

677c *(Inicialización de nodos y arcos 677c)≡* (680a)

```
Operate_On_Nodes<GT, Initialize_Dijkstra_Node<GT, Distance> > () (g);
```

La clase `Initialize_Dijkstra_Node` se especifica así:

677d *(Clase apartadora de información para nodo 677d)≡*

```
template <class GT, class Distance> struct Initialize_Dijkstra_Node
{
 void operator () (GT & g, typename GT::Node * p)
 {
 g.reset_bit(p, Aleph::Dijkstra);
 NODE_COOKIE(p) = new Dijkstra_Node_Info <GT, Distance>;
 }
};
```

Como vemos, la función es inicializar el bit de control, apartar la memoria para un objeto `Dijkstra_Node_Info` y asignarla al cookie.

Análogamente, después de culminar el algoritmo tenemos que liberar los objetos `Dijkstra_Node_Info`:

677e *(Desinicializar nodos y arcos 677e)≡* (680b 682)

```
Operate_On_Nodes <GT, Destroy_Dijkstra_Node <GT, Distance> > () (g);
```

La clase `Destroy_Dijkstra_Node` es un poco más delicada, pues aparte de liberar la memoria debe mapear el nodo del grafo con su correspondiente en el árbol abarcador contentivo

de los caminos mínimos:

678a *(Clase liberadora de información para nodo 678a)≡*

```
template <class GT, class Distance> struct Destroy_Dijkstra_Node
{
 void operator () (GT &, typename GT::Node * p)
 {
 Dijkstra_Node_Info<GT,Distance> * aux = DNI(p); //bloque a liberar
 typename GT::Node * tp = TREENODE(p); // imagen en árbol abarcador
 if (tp != NULL) // ¿está este nodo incluido en el árbol abarcador?
 {
 NODE_COOKIE(p) = NODE_COOKIE(tp) = NULL;
 GT::map_nodes (p, tp);
 }
 else
 NODE_COOKIE(p) = NULL;

 delete aux;
 }
};
```

### 7.9.1.2 Potencial en los arcos

La información para un arco es parecida a la del nodo y se define de manera similar:

678b *(Información por arco en Dijkstra 678b)≡*

```
template <class GT, class Distance>
struct Dijkstra_Arc_Info
{
 typename GT::Arc * tree_arc; // imagen en árbol
 typename Distance::Distance_Type pot; // potencial del arco
};
```

Defines:

`Dijkstra_Arc_Info`, used in chunks 678c and 679b.

El manejo de esta estructura mediante macros, su inicialización y liberación se realiza de manera similar a la de los nodos.

La informaciones adicionales asociadas a los nodos y arcos es apartada e inicializada al principio del algoritmo y liberada al final, de la misma manera que con el algoritmo de Prim.

En consonancia con la interfaz de ArcHeap debemos especificar la clase de acceso al campo `heap_node`. Pero antes de ello, debemos indicarle a la clase `ArcHeap` que debe acceder al potencial del arco y no al valor de la distancia retornado por `Distance::operator () (typename GT::Arc *a)`. Para eso, internamente diseñamos una clase “wrapper” (envoltaria) de la clase `Distance` proporcionada por el usuario que se encargue de retornar el potencial:

678c *(Definición de Distance en Dijkstra 678c)≡*

```
template <class GT, class Distance> struct Dijkstra_Pot : public Distance
{
 typename Distance::Distance_Type operator () (typename GT::Arc *a)
 {
 typedef Dijkstra_Arc_Info<GT, Distance> Arc_Info;
```

```

 Arc_Info * arc_info = (Arc_Info*) ARC_COOKIE(a);
 return arc_info->pot;
}
};

Uses Dijkstra_Arc_Info 678b.

```

Esta es la clase de distancia que le pasaremos a ArcHeap.

Ahora, definido el acceso al potencial de un arco, especificamos la información que tendrá el heap:

679a *<Información por nodo en Dijkstra 677a>* +≡ ◁677a

```

template <class GT, class Distance, class Compare>
struct Dijkstra_Heap_Info
{
 typedef typename ArcHeap<GT, Dijkstra_Pot<GT, Distance>,
 Compare, Dijkstra_Heap_Info>::Node Node;

 Node *& operator () (typename GT::Node * p)
 {
 return (Node*)& HEAPNODE(p);
 }
};

```

Uses ArcHeap 668a.

Esta estructura también merece algunos macros:

679b *<Definición de macros para algoritmo de Dijkstra 677b>* +≡ ◁677b

```

#define DAI(p) ((Dijkstra_Arc_Info<GT, Distance>*) ARC_COOKIE(p))
#define ARC_DIST(p) (Distance () (p))
#define TREEARC(p) (DAI(p)->tree_arc)
#define POT(p) (DAI(p)->pot)

```

Defines:

ARC\_DIST, used in chunks 680c and 681b.

POT, used in chunks 679–81.

TREEARC, used in chunk 681a.

Uses Dijkstra\_Arc\_Info 678b.

Antes de definir el heap debemos especificar la manera en que se comparan los potenciales:

679c *<Comparación entre potenciales 679c>* ≡

```

template <class GT, class Compare, class Distance> struct Compare_Arc_Data
{
 bool operator () (typename GT::Arc * a1, typename GT::Arc * a2) const
 {
 return Compare () (POT(a1), POT(a2));
 }
};

```

Uses POT 679b.

Esto nos permite declarar el heap;

679d *<Declaración de heap en Dijkstra 679d>* ≡ (680b 682)

```

typedef Dijkstra_Heap_Info<GT, Distance, Compare> Heap_Info;
ArcHeap<GT, Dijkstra_Pot<GT, Distance>, Compare, Heap_Info> heap;

```

Uses ArcHeap 668a.

### 7.9.1.3 El algoritmo de Dijkstra

El algoritmo tiene dos fases: una de inicialización otra de cálculo. La inicialización aparta la memoria adicional para la distancia acumulada y los potenciales:

680a *(Inicializar algoritmo de Dijkstra 680a)≡* (680b 682)

```
(Inicialización de nodos y arcos 677c)
NODE_BITS(start_node).set_bit(Aleph::Dijkstra, true);
ACC(start_node) = Distance::Zero_Distance;
TREENODE(start_node) = tree.insert_node(start_node->get_info());
NODE_COOKIE(TREENODE(start_node)) = start_node;
```

Al final del algoritmo debemos asegurarnos de que el heap sea liberado antes la información asociada a los nodos y arcos. Si ocurre al revés, entonces el heap referenciará cookies que ya fueron liberados. Por esa razón creamos un bloque interno donde se declara el heap. De ese modo, la estructura general del algoritmo de Dijkstra queda como sigue:

680b *(algoritmo de Dijkstra 680b)≡* (676)

```
(Inicializar algoritmo de Dijkstra 680a)
{ // bloque que asegura que heap se destruya antes que
 // los bloques almacenados en los cookies
(Declaración de heap en Dijkstra 679d)
(Meter arcos iniciales en heap 680c)
(Calcular árbol abarcador de Dijkstra 680d)
} // al final de este bloque se destruye el heap
(Desinicializar nodos y arcos 677e)
```

Al igual que el recorrido en amplitud y el algoritmo de Prim, debemos meter los arcos de inicio en el heap antes de iniciar el cálculo:

680c *(Meter arcos iniciales en heap 680c)≡* (680b 682)

```
for (Node_Arc_Iterator<GT, SA> it(start_node); it.has_current(); it.next())
{
 typename GT::Arc * arc = it.get_current_arc();
 POT(arc) = ARC_DIST(arc);
 ARC_BITS(arc).set_bit(Aleph::Dijkstra, true);
 heap.put_arc(arc, it.get_tgt_node());
}
```

Uses ARC\_DIST 679b, POT 679b, and put\_arc 669a.

Una vez que los potenciales iniciales están en el heap y el valor de distancia nula en start\_node, se inicia la iteración para calcular el árbol abarcador de Dijkstra, cuyas ramas son caminos mínimos desde la raíz start\_node:

680d *(Calcular árbol abarcador de Dijkstra 680d)≡* (680b)

```
// mientras tree no abarque a g
while (tree.get_num_nodes() < g.get_num_nodes())
{
 (Obtener y procesar próximo arco con menor potencial 681a)
 (Actualizar potenciales y meterlos en heap 681b)
}
```

El bloque repetitivo termina cuando tree abarca a g, lo cual se detecta cuando tree alcanza la misma cantidad de nodos. Al igual que el algoritmo de Prim, el de Dijkstra mantiene a tree conexo, lo cual permite detectar ciclos por simple marcado de los nodos visitados sin necesidad de una prueba de aciclicidad. Por modularidad, es muy conveniente

dividir el cálculo interno del lazo en dos partes estructuralmente idénticas a la del recorrido en amplitud y el algoritmo de Prim:

1.

```
681a ⟨Obtener y procesar próximo arco con menor potencial 681a⟩≡ (680d 683a)
 typename GT::Arc * garc = heap.get_min_arc();
 typename GT::Node * gsrc = g.get_src_node(garc);
 typename GT::Node * gtgt = g.get_tgt_node(garc);

 // ¿Están los dos nodos visitados?
 if (IS_NODE_VISITED(gsrc, Aleph::Dijkstra) and
 IS_NODE_VISITED(gtgt, Aleph::Dijkstra))
 continue; // insertar arco causaría ciclo

 typename GT::Node * new_node = // escoger no visitado
 IS_NODE_VISITED(gsrc, Aleph::Dijkstra) ? gtgt : gsrc;

 typename GT::Node * ttgt = tree.insert_node(new_node->get_info());
 TREENODE(new_node) = ttgt;
 NODE_BITS(new_node).set_bit(Aleph::Dijkstra, true);

 typename GT::Arc * tarc = // insertar nuevo arco en tree
 tree.insert_arc(TREENODE(gsrc), TREENODE(gtgt), garc->get_info());
 TREEARC(garc) = tarc;
Uses get_min_arc 669b and TREEARC 679b.
```

2.

```
681b ⟨Actualizar potenciales y meterlos en heap 681b⟩≡ (680d 683a)
 ACC(new_node) = POT(garc); // actualizar dist desde nodo inicial
 const typename Distance::Distance_Type & acc = ACC(new_node);

 // por cada arco calcular potencial e insertarlo en heap
 for (Node_Arc_Iterator<GT, SA> it(new_node); it.has_current(); it.next())
 {
 typename GT::Arc * arc = it.get_current_arc();
 if (IS_ARC_VISITED(arc, Aleph::Dijkstra))
 continue;

 ARC_BITS(arc).set_bit(Aleph::Dijkstra, true);
 typename GT::Node * tgt = it.get_tgt_node();
 if (IS_NODE_VISITED(tgt, Aleph::Dijkstra))
 continue; // causaría ciclo ==> no se mete en heap

 POT(arc) = Plus () (acc, ARC_DIST(arc)); // calcula potencial
 heap.put_arc(arc, tgt);
 }
Uses ARC_DIST 679b, POT 679b, and put_arc 669a.
```

Es importante remarcar la bondad que aporta verificar si el nodo destino del arco ya ha sido visitado. En efecto, esta verificación puede ahorrarnos una inserción al heap, cuyo coste es  $\mathcal{O}(\lg V)$ . Bajo esta misma línea de observación, notemos que puede ocurrir que el destino de arc ya haya sido visitado, pues ya puede existir otra vía insertada en el heap que alcance el nodo destino. Por esa razón es necesario que en el bloque *(Obtener y procesar próximo arco con menor potencial 681a)* se verifique visita de los dos nodos del arco.

El algoritmo de Dijkstra puede implantarse fácilmente con matrices de adyacencia. La estructura es similar e, inclusive, puede guardarse en una matriz que contenga todos los pares de caminos mínimos. Esta técnica se estudiará en § 7.9.2 (Pág. 685).

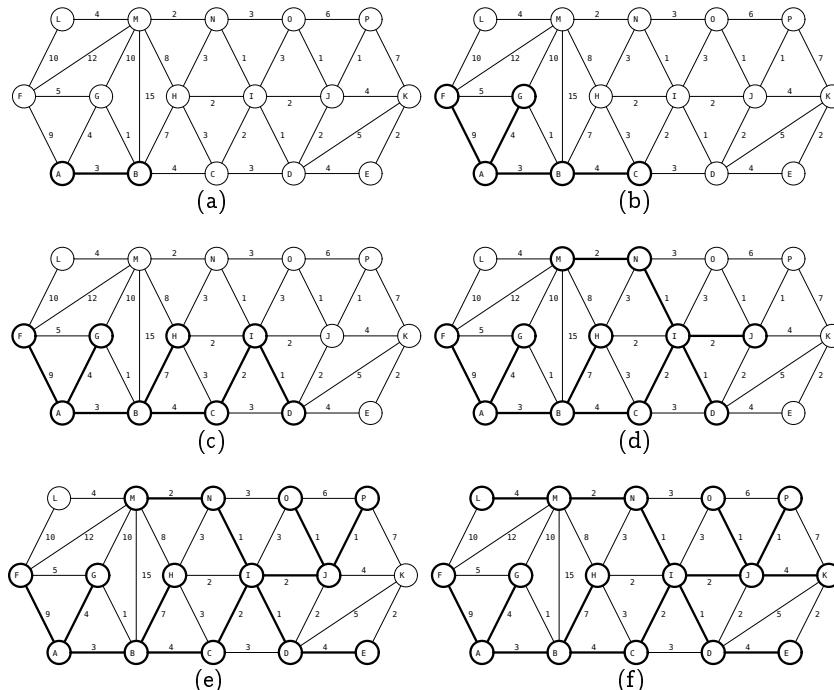


Figura 7.56: Progreso del algoritmo de Dijkstra con raíz en el nodo a (cada figura corresponde a tres iteraciones)

#### 7.9.1.4 Cálculo de un camino mínimo

El algoritmo anterior calcula un árbol abarcador con raíz en el nodo origen. Sus ramas representan caminos mínimos desde el origen a cualquier otro nodo. Si lo que se pretende es conocer un camino mínimo entre dos nodos, entonces podemos ahorrar tiempo de ejecución si detenemos el cálculo del árbol abarcador una vez que se alcance el nodo destino. Esto lo podemos plantear de la siguiente manera:

*(Prototipo del algoritmo de Dijkstra 676)* +≡

```
template <class GT, class Distance,
 class Compare = Aleph::less<typename Distance::Distance_Type>,
 class Plus = Aleph::plus<typename Distance::Distance_Type>,
 class SA = Default_Show_Arc<GT> > inline
void dijkstra_min_path(GT & g, typename GT::Node * start_node,
```

```

 typename GT::Node * end_node, Path<GT> & min_path)
{
 GT tree; // árbol abarcador temporal
 ⟨Inicializar algoritmo de Dijkstra 680a⟩
 {
 ⟨Declaración de heap en Dijkstra 679d⟩
 ⟨Meter arcos iniciales en heap 680c⟩
 ⟨Calcular árbol abarcador de Dijkstra parcial 683a⟩
 }
 ⟨Desinicializar nodos y arcos 677e⟩
 ⟨Buscar en tree camino entre start_node y end_node 683b⟩
}

```

Uses Path 578a.

Mención especial merece la posición del bloque penúltimo bloque ⟨Desinicializar nodos y arcos 677e⟩, la cual debe estar antes que ⟨Buscar en tree camino entre start\_node y end\_node 683b⟩. La razón de esto es que ⟨Desinicializar nodos y arcos 677e⟩ efectúa los mapeos de los nodos y arcos hacia el árbol abarcador.

La primera parte del algoritmo es muy similar al bloque ⟨algoritmo de Dijkstra 680b⟩. La única diferencia está dada por el bloque ⟨Calcular árbol abarcador de Dijkstra parcial 683a⟩ cuya estructura es parecida al bloque ⟨Calcular árbol abarcador de Dijkstra 680d⟩, salvo que para ahorrar tiempo de ejecución, el cálculo del árbol abarcador se detiene cuando se mira el nodo destino end\_node, o sea, cuando gtgt == end\_node:

683a ⟨Calcular árbol abarcador de Dijkstra parcial 683a⟩≡ (682)

```

while (tree.get_num_nodes() < g.get_num_nodes()) // mientras tree no abarque g
{
 ⟨Obtener y procesar próximo arco con menor potencial 681a⟩
 if (gtgt == end_node) // ¿está end_node en árbol abarcador?
 break; // sí ==> camino mínimo ya está en árbol abarcador

 ⟨Actualizar potenciales y meterlos en heap 681b⟩
}

```

Al término de este bloque, tree contiene un árbol abarcador, probablemente parcial, que contiene un camino mínimo desde start\_node hasta end\_node. Lo primero que hacemos, pues, es encontrar ese camino mínimo:

683b ⟨Buscar en tree camino entre start\_node y end\_node 683b⟩≡ (682) 683c▷

```

typename GT::Node * tstart_node = mapped_node<GT>(start_node);
typename GT::Node * tend_node = mapped_node<GT>(end_node);
Path<GT> tree_min_path(tree);
find_path_depth_first(tree, tstart_node, tend_node, tree_min_path);

```

Uses mapped\_node 560b and Path 578a.

tree\_min\_path contiene el camino mínimo en tree, pero lo que deseamos es el camino mínimo en g. Así pues, la siguiente fase consiste en copiar, siguiendo el mapeo, el camino tree\_min\_path al parámetro min\_path:

683c ⟨Buscar en tree camino entre start\_node y end\_node 683b⟩+≡ (682) ▷683b

```

min_path.clear_path();
min_path.init(start_node);
typename Path<GT>::Iterator it(tree_min_path);
for (it.next(); it.has_current(); it.next())

```

```

min_path.append(GRAPHNODE(it.get_current_node()));

Uses Path 578a.

```

### 7.9.1.5 Correctitud del algoritmo de Dijkstra

Un asunto de esencial conocimiento es que el algoritmo de Dijkstra no opera correctamente para arcos negativos. El algoritmo asume que la distancia total desde el nodo origen no

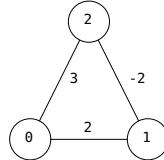


Figura 7.57: Ejemplo de arco negativo que hace fallar al algoritmo de Dijkstra

decrece; premisa fundamental para adjudicar pertenencia de un arco al árbol abarcador mínimo con raíz en el nodo origen. Consideremos, como contraejemplo, el grafo de la figura 7.57 y origen en el nodo 0. El algoritmo de Dijkstra selecciona como árbol abarcador los arcos  $0 \rightarrow 2$  y  $0 \rightarrow 1$ , lo que hace, falsamente, el arco  $0 \rightarrow 2$  como el camino mínimo desde 0 hasta 2 con coste 3, cuando el mínimo es  $0 \rightarrow 1 \rightarrow 2$  con coste 0<sup>21</sup>.

Aclarado lo anterior, podemos establecer la correctitud bajo la siguiente proposición:

**Proposición 7.5** Sea  $G$  un grafo conexo cualquiera. Entonces el algoritmo de Dijkstra sobre un nodo de origen  $v$  encuentra un árbol abarcador contentivo de todos los caminos mínimos desde  $v$ .

**Demostración (por inducción sobre  $n = |V|$ )** Suponga un grafo conexo  $G = \langle V, E \rangle$ , un nodo  $v \in V$  raíz de un árbol abarcador de Dijkstra y un árbol abarcador  $T = \langle V, E' \rangle$ .

1.  $n = 2$ : en este caso, el algoritmo saca un solo arco que conforma el árbol abarcador desde el nodo origen y que es el mínimo.
2. Ahora suponemos que la proposición es cierta para todo  $n$  y constatamos si lo es para  $n + 1$ .

Sea  $v \in V$  el nodo origen del árbol abarcador y  $w$  un nodo destino que en la iteración  $n$  no se encuentra en el árbol abarcador parcial  $T_n$ .

Por la hipótesis inductiva  $T_n$  es un árbol abarcador de todos los caminos mínimos desde  $v$ , es decir, cada camino  $v \rightsquigarrow u$  en  $T_n$  es de distancia mínima.

Ahora examinemos la iteración que añadiría  $w$  a  $T_n$  a través de un arco  $e = v \rightarrow w$ . Si se usa un heap tradicional, entonces pueden obtenerse arcos que causen ciclos y se descarten. Pero finalmente se obtendrá  $e = v \rightarrow w$ , cual es el arco de menor distancia entre un nodo  $x \in T_n$  y  $w$  de los que conectan a  $T_n$  con  $w$  sin causar un ciclo. Si se usa un heap exclusivo, entonces se obtiene directamente a  $e$ .

<sup>21</sup>Este simplísimo ejemplo es válido tanto para el heap exclusivo como para un heap tradicional. Hacemos esta aclaratoria porque muchos contraejemplos de falla del algoritmo de Dijkstra con pesos negativos son diseñados para un heap tradicional y no exclusivo. A pesar de la disminución de probabilidad de falla ante el uso de un heap exclusivo, el algoritmo de Dijkstra sigue siendo incorrecto con pesos negativos.

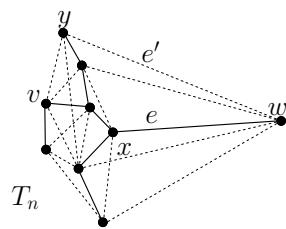


Figura 7.58: Esquema genérico para la última iteración del algoritmo de Dijkstra

Para probar que árbol abarcador resultante es el de todos los caminos mínimos desde  $v$ , debemos probar que el camino  $v \rightsquigarrow w$  es de distancia mínima. Hagámoslo por reducción al absurdo.

Supongamos que existe un camino  $v \rightsquigarrow y \rightarrow w$  tal que  $d(v \rightsquigarrow y \rightarrow w) < d(v \rightsquigarrow x \rightarrow w)$ . Por la hipótesis inductiva,  $d(v \rightsquigarrow y)$  es mínima. Ahora bien,  $d(x \rightarrow w) \leq d(y \rightarrow w)$ , pues el heap ordena por peso. Por tanto,  $d(v \rightsquigarrow x) + d(x \rightarrow w) \leq d(v \rightsquigarrow y) + d(y \rightarrow w)$ , pero esto contradice la suposición  $d(v \rightsquigarrow y \rightarrow w) < d(v \rightsquigarrow x \rightarrow w)$ . Por tanto,  $v \rightsquigarrow x \rightarrow w$  es un camino mínimo y el árbol abarcador es el de todos los caminos mínimos desde  $v$  ■

### 7.9.1.6 Análisis del algoritmo de Dijkstra

El análisis es prácticamente similar al del algoritmo de Prim, pues las estructuras de flujo de ambos algoritmos son las mismas.

La fase inicialización de nodos y arcos consume  $\mathcal{O}(V) + \mathcal{O}(E)$ , lo cual, puesto que el grafo es conexo, puede considerarse  $\mathcal{O}(E)$ . Lo mismo ocurre para la liberación de memoria al final del algoritmo.

Así pues, el análisis se centra en contabilizar las iteraciones del bloque *<Calcular árbol abarcador de Dijkstra 680d>*, cuyo análisis es idéntico al del algoritmo de Prim, pues la estructura es la misma y el algoritmo de Dijkstra también usa el heap exclusivo. El bloque *<Calcular árbol abarcador de Dijkstra 680d>* tiene una complejidad en tiempo de  $\mathcal{O}(E \lg V)$ , cual es el desempeño definitivo del algoritmo de Dijkstra.

### 7.9.2 Algoritmo de Floyd-Warshall

El algoritmo de Floyd-Warshall calcula todos los caminos mínimos entre pares de nodos. Es una variante del algoritmo de Warshall para calcular la clausura transitiva estudiado en § 7.6.3 (Pág. 635). Como tal, opera sobre matrices de adyacencia y posee la virtud de operar con pesos negativos. Puede adaptarse para detectar y determinar ciclos negativos.

El algoritmo fue primeramente reportado descubierto por Floyd [52], pero, puesto que estructuralmente es similar al de Warshall, suele llamársele de "Floyd-Warshall".

La especificación del algoritmo se encuentra en el archivo Floyd.H.

### 7.9.2.1 Interfaces

El algoritmo de Floyd-Warshall maneja las siguientes interfaces:

1.

686a *(Interfaces del algoritmo de Floyd-Warshall 686a)≡*

```
template <class GT, typename Distance,
 class Compare = Aleph::less<typename Distance::Distance_Type>,
 class Plus = Aleph::plus<typename Distance::Distance_Type>,
 class SA = Default_Show_Arc<GT> >
void floyd_all_shortest_paths
 (GT & g,
 Ady_Mat<GT, typename Distance::Distance_Type, SA> & dist,
 Ady_Mat<GT, long, SA> & path);
```

La rutina recibe cinco parámetros tipo:

- (a) GT: el tipo de grafo (o digrafo).
- (b) Distance: es la clase que maneja las distancias. Minimalmente, éste debe exportar los atributos siguientes explicados en § 7.8.1 (Pág. 660):
  - i. Distance\_Type
  - ii. Max\_Distance
  - iii. Zero\_Distance
  - iv. operator () (typename GT::Arc \*)
- (c) Compare: clase de comparación entre distancias.
- (d) Plus: clase de suma entre distancias.
- (e) SA: filtro de arcos.

Los parámetros de la rutina son:

- (a) g: el grafo o digrafo representado mediante una variante de List\_Graph.
- (b) dist: matriz de costes mínimos entre pares de nodos. Cada entrada  $dist_{i,j}$  contiene el coste mínimo entre los nodos de índices en la matriz de adyacencia i y j respectivamente.
- (c) path: matriz de recuperación de caminos. Cada entrada  $path_{i,j}$  contiene el nodo k que pasa por el camino mínimo entre i y j. Inspecciones sucesivas de  $path_{k,j}$  hasta que  $path_{k,j} = j$  permiten recuperar el camino mínimo entre el par  $< i, j >$ .

La rutina anterior se implantará en el siguiente bloque:

686b *(Rutinas del algoritmo de Floyd-Warshall 686b)≡*

```
template <class GT, typename Distance,
 class Compare = Aleph::less<typename Distance::Distance_Type>,
 class Plus = Aleph::plus<typename Distance::Distance_Type>,
 class SA = Default_Show_Arc<GT> >
void floyd_all_shortest_paths
 (GT & g,
 Ady_Mat<GT, typename Distance::Distance_Type, SA> & dist,
 Ady_Mat<GT, long, SA> & path)
{
 (Inicializar algoritmo de Floyd-Warshall 688a)
 (Algoritmo de Floyd 689b)
}
```

2.

687a *(Recuperación de caminos 687a)≡* 687b ▷  
 template <class Mat>  
 void find\_min\_path(Mat & p, const long & src\_index, const long & tgt\_index,  
 Path<typename Mat::List\_Graph\_Type> & path);  
 Uses Path 578a.

Dada una matriz de caminos mínimos calculada mediante `floyd_all_shortest_paths()`, la rutina construye un camino mínimo entre los pares de nodos con índices `src_index` y `tgt_index`.

3.

687b *(Recuperación de caminos 687a)+≡* △687a 692▷  
 template <class Mat>  
 void find\_min\_path(Mat & p, typename Mat::Node \* src\_node,  
 typename Mat::Node \* tgt\_node,  
 Path<typename Mat::List\_Graph\_Type> & path)  
 {  
 find\_min\_path(p, p(src\_node), p(tgt\_node, path));  
}  
 Uses Path 578a.

Similar a la anterior, salvo que los nodos se especifican directamente mediante punteros en la representación con listas enlazadas.

### 7.9.2.2 El algoritmo de Floyd-Warshall

Similar al de Warshall, el algoritmo de Floyd-Warshall emplea tres lazos que exploran, comparan y seleccionan los caminos de longitud  $1, 2, \dots, V$  hasta cubrir el número de nodos.

#### Inicialización

La idea básica consiste en considerar nodos intermedios de caminos más cortos bajo el mismo principio del de Warshall. A tal fin, el algoritmo utiliza una matriz  $\text{dist}_{i,j}$  de distancias  $n \times n$  cuyos valores iniciales obedecen a lo siguiente:

$$\text{dist}_{i,j}^0 = \begin{cases} C_{i,j} & \text{si existe arco entre } i \text{ y } j \\ \infty & \text{si no existe arco entre } i \text{ y } j \\ 0 & \text{si } i = j \end{cases}; \quad (7.8)$$

$C_{i,j}$  es el peso del arco entre el nodo  $i$  y  $j$ .

Dado un grafo, los valores iniciales se colocan mediante la siguiente clase:

687c *(Inicialización de matrices 687c)≡*  
 template <class AM, class Distance, class SA> struct Initialize\_Dist\_Floyd  
 {  
 typedef typename AM::List\_Graph\_Type::Arc\_Type Arc\_Type;  
 typedef typename AM::List\_Graph\_Type GT;

```

typedef typename GT::Node Node;
typedef typename GT::Arc Arc;
typedef typename Distance::Distance_Type Distance_Type;

 // inicializa cada entrada de la matriz
void operator () (AM & mat, Node * src, Node * tgt,
 const long & i, const long & j, Distance_Type & entry,
 void * p) const
{
 Ady_Mat <typename AM::List_Graph_Type, long, SA> & path =
 * (Ady_Mat <typename AM::List_Graph_Type, long> *) (p);

 if (i == j) // ¿es diagonal de la matriz?
 {
 // sí ==> la distancia es cero
 entry = Distance::Zero_Distance;
 path(i, j) = j;
 return;
 }
 GT & g = mat.get_list_graph();
 Arc * arc = search_arc(g, src, tgt);
 if (arc == NULL) // ¿existe un arco?
 {
 // sí ==> se coloca coste infinito
 entry = Distance::Max_Distance;
 return;
 }
 // existe arco ==> extraiga distancia
 entry = Distance () (arc); // la coloca en cost(i, j)
 path(i, j) = j; // coloca camino directo
}
};

};

Este operador es invocado en la fase del inicialización del algoritmo de Floyd-Warshall:

```

688a *(Inicializar algoritmo de Floyd-Warshall 688a)*≡ (686b) 688b▷

```

typedef typename Distance::Distance_Type Dist_Type;
typedef Ady_Mat <GT, Dist_Type> Dist_Mat;
dist.template operate_all_arcs_matrix
 <Initialize_Dist_Floyd <Dist_Mat, Distance, SA> > (&path);

```

Lo que deja a las matrices en los estados iniciales según (7.8).

El resto de la fase de inicialización es fijar algunas constantes de manera de facilitar el trabajo de optimización del compilador:

688b *(Inicializar algoritmo de Floyd-Warshall 688a)*+≡ (686b) ▷688a

```

const Dist_Type & max = Distance::Max_Distance;
const long & n = g.get_num_nodes();

```

### El algoritmo de Floyd-Warshall

Como ya señalamos, el algoritmo de Floyd-Warshall utiliza tres lazos anidados que exploran caminos posibles y deciden el mínimo. En una iteración  $k$ , cada entrada  $dist_{i,j}^k$  contiene el coste mínimo para ir del nodo  $i$  hacia el nodo  $j$  con un arco de longitud  $k$ . Pictóricamente, la  $k$ -ésima iteración se ilustra en la figura 7.59. Tenemos un coste  $dist_{i,j}^{k-1}$  previamente calculado para ir desde el nodo  $i$  hasta el  $j$ . Ahora la cuestión consiste en buscar sendos caminos parciales  $(i, k)$  y  $(k, j)$ , con distancias parciales  $dist_{i,k}^{k-1}$  y  $dist_{k,j}^{k-1}$

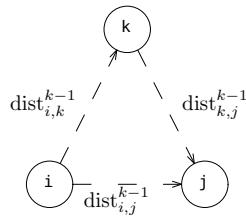


Figura 7.59: Esquema general de la  $k$ -ésima iteración del algoritmo de Floyd-Warshall

respectivamente, y evaluar si  $\text{dist}_{i,k}^{k-1} + \text{dist}_{k,j}^{k-1}$  es menos costoso que  $\text{dist}_{i,j}^{k-1}$ . Formalmente, la decisión se plantea del siguiente modo:

$$\text{dist}_{i,j}^k = \min \begin{cases} \text{dist}_{i,j}^{k-1} \\ \text{dist}_{i,k}^{k-1} + \text{dist}_{k,j}^{k-1} \end{cases} \quad (7.9)$$

La cual se modeliza de manera similar al algoritmo de Warshall:

689a *(Lazo del algoritmo de Floyd-Warshall 689a)≡*

```
for (int k = 0; k < n; ++k)
 for (int i = 0; i < n; ++i)
 for (int j = 0; j < n; ++j)
 if (dist(i, k) + dist(k, j) < dist(i, j))
 dist(i, j) = dist(i, k);
```

La diferencia esencial con el algoritmo de Warshall es que el if decide si existe un camino de menor coste que pase por un camino con nodo intermedio  $k$ .

Hay una mejora al algoritmo de Floyd-Warshall basada en la observación de que si la entrada  $\text{dist}_{i,k}^{k-1} = \infty$ , entonces, con certitud, no habrá un camino más corto que pase por el nodo intermedio  $k$ . En función de esta observación podemos plantear una variante que ahorre las iteraciones del lazo más interno:

689b *(Algoritmo de Floyd 689b)≡* (686b)

```
for (int k = 0; k < n; ++k)
 for (int i = 0; i < n; ++i)
 if (dist(i, k) < max) // ¿possible encontrar distancia menor?
 for (int j = 0; j < n; ++j)
 { // calcule nueva distancia pasando por k
 Dist_Type new_dist = Plus () (dist(i, k), dist(k, j));
 if (Compare () (new_dist, dist(i, j))) // d(i,j)<d(i,k)+d(k,j)
 {
 dist(i, j) = new_dist; // actualice menor distancia
 path(i, j) = path(i, k); // actualice nodo intermedio
 }
 }
```

Recordemos que `Plus::operator()()` implementa la suma, mientras que `Compare::operator()()` la comparación.

La actualización de la matriz  $\text{path}_{i,j}$  será explicada en § 7.9.2.5 (Pág. 692).

Para el grafo de la figura 7.60 el algoritmo calcula las siguientes matrices por iteración.

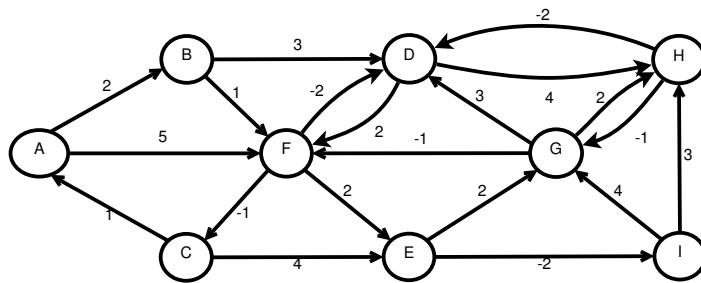


Figura 7.60: Un grafo con distancias negativas

|   | A        | B        | C        | D        | E        | F        | G        | H        | I        |
|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| A | 0        | 2        | $\infty$ | $\infty$ | $\infty$ | 5        | $\infty$ | $\infty$ | $\infty$ |
| B | $\infty$ | 0        | $\infty$ | 3        | $\infty$ | 1        | $\infty$ | $\infty$ | $\infty$ |
| C | 1        | $\infty$ | 0        | $\infty$ | 4        | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| D | $\infty$ | $\infty$ | 0        | $\infty$ | 0        | 2        | $\infty$ | 4        | $\infty$ |
| E | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 2        | $\infty$ | -2       |
| F | $\infty$ | $\infty$ | -1       | -2       | 2        | 0        | $\infty$ | $\infty$ | $\infty$ |
| G | $\infty$ | $\infty$ | $\infty$ | 3        | $\infty$ | -1       | 0        | 2        | $\infty$ |
| H | $\infty$ | $\infty$ | $\infty$ | -2       | $\infty$ | $\infty$ | -1       | 0        | $\infty$ |
| I | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 4        | 3        | 0        |          |

$P_0 =$

|   | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| A | A | B | A | A | A | F | A | A | A |
| B | A | B | A | D | A | F | A | A | A |
| C | A | A | C | A | E | A | A | A | A |
| D | A | A | A | D | A | F | A | H | A |
| E | A | A | A | A | E | A | G | A | I |
| F | A | A | C | D | E | F | A | A | A |
| G | A | A | A | D | A | F | G | H | A |
| H | A | A | A | D | A | A | G | H | A |
| I | A | A | A | A | A | A | G | H | I |

|   | A        | B        | C        | D        | E        | F        | G        | H        | I        |
|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| A | 0        | 2        | $\infty$ | $\infty$ | $\infty$ | 5        | $\infty$ | $\infty$ | $\infty$ |
| B | $\infty$ | 0        | $\infty$ | 3        | $\infty$ | 1        | $\infty$ | $\infty$ | $\infty$ |
| C | 1        | 3        | 0        | $\infty$ | 4        | 6        | $\infty$ | $\infty$ | $\infty$ |
| D | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 2        | $\infty$ | 4        | $\infty$ |
| E | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 2        | $\infty$ | -2       |
| F | $\infty$ | $\infty$ | -1       | -2       | 2        | 0        | $\infty$ | $\infty$ | $\infty$ |
| G | $\infty$ | $\infty$ | $\infty$ | 3        | $\infty$ | -1       | 0        | 2        | $\infty$ |
| H | $\infty$ | $\infty$ | $\infty$ | -2       | $\infty$ | $\infty$ | -1       | 0        | $\infty$ |
| I | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 4        | 3        | 0        |          |

$P_1 =$

|   | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| A | A | B | A | A | A | F | A | A | A |
| B | A | B | A | D | A | F | A | A | A |
| C | A | A | C | A | E | A | A | A | A |
| D | A | A | A | D | A | F | A | H | A |
| E | A | A | A | A | E | A | G | A | I |
| F | A | A | C | D | E | F | A | A | A |
| G | A | A | A | D | A | F | G | H | A |
| H | A | A | A | D | A | A | G | H | A |
| I | A | A | A | A | A | A | G | H | I |

|   | A        | B        | C        | D        | E        | F        | G        | H        | I        |
|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| A | 0        | 2        | $\infty$ | 5        | $\infty$ | 3        | $\infty$ | $\infty$ | $\infty$ |
| B | $\infty$ | 0        | $\infty$ | 3        | $\infty$ | 1        | $\infty$ | $\infty$ | $\infty$ |
| C | 1        | 3        | 0        | 6        | 4        | 4        | $\infty$ | $\infty$ | $\infty$ |
| D | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 2        | $\infty$ | 4        | $\infty$ |
| E | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 2        | $\infty$ | -2       |
| F | $\infty$ | $\infty$ | -1       | -2       | 2        | 0        | $\infty$ | $\infty$ | $\infty$ |
| G | $\infty$ | $\infty$ | $\infty$ | 3        | $\infty$ | -1       | 0        | 2        | $\infty$ |
| H | $\infty$ | $\infty$ | $\infty$ | -2       | $\infty$ | $\infty$ | -1       | 0        | $\infty$ |
| I | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 4        | 3        | 0        |          |

$P_2 =$

|   | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| A | A | B | A | B | A | B | A | A | A |
| B | A | B | A | D | A | F | A | A | A |
| C | A | A | C | A | E | A | A | A | A |
| D | A | A | A | D | A | F | A | H | A |
| E | A | A | A | A | E | A | G | A | I |
| F | A | A | C | D | E | F | A | A | A |
| G | A | A | A | D | A | F | G | H | A |
| H | A | A | A | D | A | A | G | H | A |
| I | A | A | A | A | A | A | G | H | I |

|   | A        | B        | C        | D        | E        | F        | G        | H        | I        |
|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| A | 0        | 2        | $\infty$ | 5        | $\infty$ | 3        | $\infty$ | 9        | $\infty$ |
| B | $\infty$ | 0        | $\infty$ | 3        | $\infty$ | 1        | $\infty$ | 7        | $\infty$ |
| C | 1        | 3        | 0        | 6        | 4        | 4        | $\infty$ | 10       | $\infty$ |
| D | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 2        | $\infty$ | 4        | $\infty$ |
| E | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 2        | $\infty$ | -2       |
| F | 0        | 2        | -1       | -2       | 2        | 0        | $\infty$ | 2        | $\infty$ |
| G | $\infty$ | $\infty$ | $\infty$ | 3        | $\infty$ | -1       | 0        | 2        | $\infty$ |
| H | $\infty$ | $\infty$ | $\infty$ | -2       | $\infty$ | 0        | -1       | 0        | $\infty$ |
| I | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 4        | 3        | 0        |          |

$P_3 =$

|   | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| A | A | B | A | B | A | B | A | A | A |
| B | A | B | A | D | A | F | A | D | A |
| C | A | A | C | A | E | A | A | A | A |
| D | A | A | A | D | A | F | A | H | A |
| E | A | A | A | A | E | A | G | A | I |
| F | C | C | C | D | E | F | A | D | A |
| G | A | A | A | D | A | F | G | H | A |
| H | A | A | A | D | A | A | G | H | A |
| I | A | A | A | A | A | A | G | H | I |

|   | A        | B        | C        | D        | E        | F        | G        | H        | I        |
|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| A | 0        | 2        | $\infty$ | 5        | $\infty$ | 3        | $\infty$ | 9        | $\infty$ |
| B | $\infty$ | 0        | $\infty$ | 3        | $\infty$ | 1        | $\infty$ | 7        | $\infty$ |
| C | 1        | 3        | 0        | 6        | 4        | 4        | $\infty$ | 10       | $\infty$ |
| D | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 2        | $\infty$ | 4        | $\infty$ |
| E | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0        | $\infty$ | 2        | $\infty$ | -2       |
| F | 0        | 2        | -1       | -2       | 2        | 0        | $\infty$ | 2        | $\infty$ |
| G | $\infty$ | $\infty$ | $\infty$ | 3        | $\infty$ | -1       | 0        | 2        | $\infty$ |
| H | $\infty$ | $\infty$ | $\infty$ | -2       | $\infty$ | 0        | -1       | 0        | $\infty$ |
| I | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 4        | 3        | 0        |          |

$P_4 =$

|   | A | B | C | D | E | F | G | H     | I |
|---|---|---|---|---|---|---|---|-------|---|
| A | A | B | A | B | A | B | A | B     | A |
| B | A | B | A | D | A | F | A | D     | A |
| C | A | A | C | A | E | A | A | A</td |   |

| $D_5 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                                                                                                                                                                                                                                        | $P_5 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                   |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|         | A 0 2 $\infty$ 5 $\infty$ 3 $\infty$ 9 $\infty$<br>B $\infty$ 0 $\infty$ 3 $\infty$ 1 $\infty$ 7 $\infty$<br>C 1 3 0 6 4 4 6 10 2<br>D $\infty$ $\infty$ $\infty$ 0 $\infty$ 2 $\infty$ 4 $\infty$<br>E $\infty$ $\infty$ $\infty$ 0 $\infty$ 2 $\infty$ -2<br>F 0 2 -1 -2 2 0 4 2 0<br>G $\infty$ $\infty$ 3 $\infty$ -1 0 2 $\infty$<br>H $\infty$ $\infty$ $\infty$ -2 $\infty$ 0 -1 0 $\infty$<br>I $\infty$ $\infty$ $\infty$ $\infty$ $\infty$ 4 3 0 |         | A A B A B A B A B A B A<br>B A B A D A F A D A D A<br>C A A C A E A E A E A E<br>D A A A D A F A H A H A<br>E A A A A E A G A I A<br>F C C D E F E D E E D E<br>G A A A D A F G H A A<br>H A A A D A D G H H A<br>I A A A A A G H H I |
| $D_6 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                                                                                                                                                                                                                                        | $P_6 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                   |
|         | A 0 2 2 1 5 3 7 5 3<br>B 1 0 0 -1 3 1 5 3 1<br>C 1 3 0 2 4 4 6 6 2<br>D 2 4 1 0 4 2 6 4 2<br>E $\infty$ $\infty$ $\infty$ 0 $\infty$ 2 $\infty$ -2<br>F 0 2 -1 -2 2 0 4 2 0<br>G -1 1 -2 -3 1 -1 0 1 -1<br>H 0 2 -1 -2 2 0 -1 0 0<br>I $\infty$ $\infty$ $\infty$ $\infty$ $\infty$ 4 3 0                                                                                                                                                                  |         | A A B B B B B B B B<br>B F B F F F F F F F<br>C A A C A E A E A E<br>D F F F D F F F F H F<br>E A A A A E A G A I A<br>F C C D E F E D E D E<br>G F F F F F F G F F F<br>H D D D D D D G H H D<br>I A A A A A G H H I                 |
| $D_7 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                                                                                                                                                                                                                                        | $P_7 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                   |
|         | A 0 2 2 1 5 3 7 5 3<br>B 1 0 0 -1 3 1 5 3 1<br>C 1 3 0 2 4 4 6 6 2<br>D 2 4 1 0 4 2 6 4 2<br>E 1 3 0 -1 0 1 2 3 -2<br>F 0 2 -1 -2 2 0 4 2 0<br>G -1 1 -2 -3 1 -1 0 1 -1<br>H -2 0 -3 -4 0 -2 -1 0 -2<br>I 3 5 2 1 5 3 4 3 0                                                                                                                                                                                                                                |         | A A B B B B B B B B<br>B F B F F F F F F F<br>C A A C A E A E A E<br>D F F F D F F F F H F<br>E G G G G E G G G G I<br>F C C C D E F E D E<br>G F F F F F F G F F F<br>H G G G G G G G G H G<br>I G G G G G G G G H I                 |
| $D_8 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                                                                                                                                                                                                                                        | $P_8 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                   |
|         | A 0 2 2 1 5 3 4 5 3<br>B 1 0 0 -1 3 1 2 3 1<br>C 1 3 0 2 4 4 5 6 2<br>D 2 4 1 0 4 2 3 4 2<br>E 1 3 0 -1 0 1 2 3 -2<br>F 0 2 -1 -2 2 0 1 2 0<br>G -1 1 -2 -3 1 -1 0 1 -1<br>H -2 0 -3 -4 0 -2 -1 0 -2<br>I 1 3 0 -1 3 1 2 3 0                                                                                                                                                                                                                               |         | A A B B B B B B B B<br>B F B F F F F F F F<br>C A A C A E A A A A E<br>D F F F D F F F H H F<br>E G G G G E G G G G I<br>F C C C D E F F D D E<br>G F F F F F F G F F F<br>H G G G G G G G G H G<br>I H H H H H H H H H I             |
| $D_9 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                                                                                                                                                                                                                                        | $P_9 =$ | $A \ B \ C \ D \ E \ F \ G \ H \ I$                                                                                                                                                                                                   |
|         | A 0 2 2 1 5 3 4 5 3<br>B 1 0 0 -1 3 1 2 3 1<br>C 1 3 0 1 4 3 4 5 2<br>D 2 4 1 0 4 2 3 4 2<br>E -1 1 -2 -3 0 -1 0 1 -2<br>F 0 2 -1 -2 2 0 1 2 0<br>G -1 1 -2 -3 1 -1 0 1 -1<br>H -2 0 -3 -4 0 -2 -1 0 -2<br>I 1 3 0 -1 3 1 2 3 0                                                                                                                                                                                                                            |         | A A B B B B B B B B<br>B F B F F F F F F F<br>C A A C E E E E E E<br>D F F F D F F H H H F<br>E I I I I E I I I I I<br>F C C C D E F D D E<br>G F F F F F F G F F F<br>H G G G G G G G G H G<br>I H H H H H H H H H I                 |

### 7.9.2.3 Análisis del algoritmo de Floyd-Warshall

Para estudiar el desempeño del bloque *(Inicializar algoritmo de Floyd-Warshall 688a)*, debemos considerar la estructura de la rutina `operate_all_arcs_list_graph()` descrita en § 7.6.1 (Pág. 633) y verificar que ésta ejecuta  $\mathcal{O}(V^2)$  iteraciones, pues en realidad se recorre toda la matriz. Por cada iteración, se invoca al operador () de la clase `Initialize_Dist_Floyd`, el cual ejecuta una búsqueda lineal `search_arc()` y que es  $\mathcal{O}(V)$ <sup>22</sup>. La inicialización consume entonces  $\mathcal{O}(V^2) \times \mathcal{O}(E) = \mathcal{O}(V^2 E)$ .

Claramente, el bloque *(Algoritmo de Floyd 689b)* es  $\mathcal{O}(V^3)$ .

El algoritmo presentado es entonces  $\mathcal{O}(V^2 E)$ , resultado que difiere del tradicional  $\mathcal{O}(V^3)$  asociado al algoritmo de Floyd-Warshall. La razón es simple, el análisis tradicional asume que el grafo ya está ubicado en una matriz de adyacencia.

<sup>22</sup>Recordemos que la implantación de `search_arc()` presentada en § 7.3.10.2 (Pág. 571) busca entre los arcos del nodo cuyo grado sea menor y no entre todos los arcos del grafo. De aquí, pues, que la búsqueda de un arco dado sus dos nodos esté acotada, para el peor caso, por  $\mathcal{O}(V)$ .

### 7.9.2.4 Correctitud del algoritmo de Floyd-Warshall

La correctitud del algoritmo de Floyd-Warshall debe ser nula meridiana si hemos comprendido que el algoritmo de Warshall explora todos los caminos posibles entre pares de nodos. Esto puede aceptarse como correcto si asumimos que el algoritmo de Warshall calcula la clausura transitiva de la relación binaria conformada por el grafo (ver § 7.6.3 (Pág. 635))<sup>23</sup>.

El algoritmo de Floyd-Warshall también explora todos los pares de caminos, lo cual se corrobora apreciando la similitud entre las ecuaciones (7.4) y (7.9). Cada iteración interna del algoritmo de Floyd-Warshall examina y compara un nuevo coste entre un par de nodos. Por cada par de nodos, el algoritmo de Floyd-Warshall explora todos los caminos posibles y selecciona el menor de ellos. Por tanto, el algoritmo de Floyd-Warshall descubre todos los caminos mínimos entre pares de nodos.

### 7.9.2.5 Recuperación de caminos

La matriz  $\text{dist}_{i,j}$  calculada por el algoritmo de Floyd-Warshall contiene los costes mínimos entre cada par de nodos  $i$  y  $j$ . Cada entrada  $\text{path}_{i,j}$  guarda un nodo  $k$ , sucesor de  $i$  que se encuentra en el camino mínimo hacia  $j$ . Por ejemplo, si queremos conocer el camino mínimo desde  $G$  hacia  $A$ , entonces miramos la entradas  $\text{path}_{G,A} = F$ ,  $\text{path}_{F,A} = C$  y  $\text{path}_{C,A} = A$ , lo cual conforma a  $G \rightarrow F \rightarrow C \rightarrow A$  como el camino mínimo.

El procedimiento `find_min_path()`, que recupera y coloca en `path` el camino más corto, entre los nodos cuyos índices origen y destino en la matriz de adyacencia `p` son `src_index` y `tgt_index`, se instrumenta del siguiente modo:

```
692 <Recuperación de caminos 687a>+≡ <687b
 template <class Mat> void
 find_min_path(Mat & p, const long & src_idx, const long & tgt_idx,
 Path<typename Mat::List_Graph_Type> & path)
 {
 typedef typename Mat::List_Graph_Type GT;
 GT & g = p.get_list_graph();
 typename GT::Node * src = p(src_idx);
 path.set_graph(g, src);
 for (int i = src_idx, j; j != tgt_idx; i = j)
 {
 j = p(i, tgt_idx);
 path.append(p(j));
 }
 }
```

Uses Path 578a.

Puesto que puede haber otros algoritmos que mantengan matrices de caminos similares, este algoritmo lo incluimos en el archivo `mat_path.H`.

### 7.9.3 Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford [18, 55], llamado así en honor a sus primeros descubridores conocidos<sup>24</sup>, independientes entre sí, encuentra todos los caminos mínimos desde un nodo

<sup>23</sup>En la sub-sección § 7.6.3 (Pág. 635) no se demostró que el algoritmo de Warshall es correcto.

<sup>24</sup>En realidad, parece que hay otros autores. Véanse las notas bibliográficas en § 7.15 (Pág. 829) para aclaratorias al respecto.

dado. El algoritmo opera con ciclos negativos y tiene la gran bondad de permitir su detección.

Es similar al de Dijkstra en el sentido de que construye un árbol abarcador de todos los caminos mínimos desde un nodo origen, al igual que requiere asociar a cada nodo la distancia acumulada desde el origen.

El algoritmo de Bellman-Ford reside en el archivo `Bellman_Ford.H`, en el cual se exportan las siguientes primitivas:

```
693a <Rutinas de algoritmo de Bellman-Ford 693a>≡ 704▷
template <class GT, class Distance,
 class Compare = Aleph::less<typename Distance::Distance_Type>,
 class Plus = Aleph::plus<typename Distance::Distance_Type>,
 class SA = Default_Show_Arc<GT> > inline
bool bellman_ford_min_spanning_tree(GT & g, typename GT::Node * start_node,
 GT & tree)
{
 <Inicialización de algoritmo de Bellman-Ford 693b>
 <Algoritmo genérico de Bellman-Ford 696a>
 <Detención de ciclos negativos 700>
 <Construcción árbol abarcador de algoritmo de Bellman-Ford 701>
}
template <class GT, class Distance,
 class Compare = Aleph::less<typename Distance::Distance_Type>,
 class Plus = Aleph::plus<typename Distance::Distance_Type>,
 class SA = Default_Show_Arc<GT> > inline
bool q_bellman_ford_min_spanning_tree(GT & g, typename GT::Node * start_node,
 GT & tree)
{
 <Inicialización de algoritmo de Bellman-Ford 693b>
 <Algoritmo de Bellman-Ford 696b>
 <Detención de ciclos negativos 700>
 <Construcción árbol abarcador de algoritmo de Bellman-Ford 701>
}
```

`bellman_ford_min_spanning_tree()` calcula el árbol abarcador de caminos mínimos desde el nodo origen `start_node` según el algoritmo clásico. El parámetro tipo `GT` representa el grafo, `Distance` la clase extractora de los pesos contenidos en los arcos, `Compare` la comparación entre pesos, `Plus` la suma entre pesos y `SA` el filtro sobre los arcos. Si existe el árbol abarcador de caminos mínimo, entonces la rutina retorna `true`, de lo contrario existe un ciclo negativo y se retorna `false`. `q_bellman_ford_min_spanning_tree()` es una versión mejorada que usa internamente una cola de nodos, lo que conlleva mayor consumo de espacio pero tiende substancialmente a acelerar el algoritmo.

### 7.9.3.1 Inicialización del algoritmo de Bellman-Ford

Para mantener el árbol abarcador, el algoritmo utiliza dos arreglos temporales, uno de nodos predecesores, parte de los caminos mínimos, y otro de arcos, todo según lo tratado en § 7.5.11 (Pág. 606):

```
693b <Inicialización de algoritmo de Bellman-Ford 693b>≡ (693a 705) 694b▷
DynArray<typename GT::Node*> pred(g.get_num_nodes());
DynArray<typename GT::Arc*> arcs(g.get_num_nodes());
```

Uses DynArray 34.

Por cada nodo se mantiene la siguiente información:

694a *(Información por nodo en Bellman-Ford 694a)≡*

```
template <class GT, class Distance> struct Bellman_Ford_Node_Info
{
 int idx;
 typename Distance::Distance_Type acum;
};
```

idx es el índice dentro del arreglo preds, utilizado para acceder en  $\mathcal{O}(1)$  al nodo predecesor en el camino mínimo parcial. acum es la distancia parcial acumulada desde el nodo origen start\_node.

El acceso a los campos de esta estructura se facilita mediante los macros NI para idx y ACU para acum respectivamente.

El valor inicial de ACU(start\_node) es cero, mientras que el del resto es infinito. En términos de código, el valor inicial de ACU(start\_node) es Distance::Zero\_Distance y Distance::Max\_Distance para el resto.

Los nodos y los arreglos se inician mediante el siguiente lazo:

694b *(Inicialización de algoritmo de Bellman-Ford 693b)≡ (693a 705) ▷693b 694c▷*

```
{
 typename GT::Node_Iterator it(g);
 for (int i = 0; it.has_current(); ++i, it.next())
 {
 pred[i] = NULL;
 arcs[i] = NULL;
 typename GT::Node * p = it.get_current_node();
 g.reset_bit(p, Aleph::Min); // colocar bit en cero
 Bellman_Ford_Node_Info <GT, Distance> * ptr =
 new Bellman_Ford_Node_Info <GT, Distance>;
 ptr->idx = i;
 ptr->acum = Distance::Max_Distance; // infinito
 NODE_BITS(p).set_bit(Min, false);
 NODE_BITS(p).set_bit(Breadth_First, false);
 NODE_COOKIE(p) = ptr;
 }
}
```

El bit Breadth\_First se utilizará para distinguir nodos que se hayan insertado en una cola. Usamos Breadth\_First porque el algoritmo de Bellman-Ford recorre el grafo en amplitud, lo que mostraremos en § 7.9.3.4 (Pág. 696). El bit Min será explicado en § 7.9.3.6 (Pág. 701).

El único nodo con valor acumulado de distancia conocido al principio es start\_node:

694c *(Inicialización de algoritmo de Bellman-Ford 693b)≡ (693a 705) ▷694b*

```
ACU(start_node) = Distance::Zero_Distance;
```

### 7.9.3.2 Manejo del árbol abarcador

A diferencia del algoritmo de Dijkstra, en el de Bellman-Ford, las ramas del árbol abarcador se modifican durante el cálculo. Por esa razón es conveniente la representación árbol explicada en § 7.5.11 (Pág. 606).

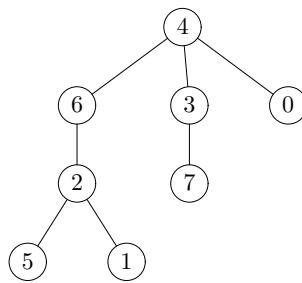


Figura 7.61: Un árbol ejemplo de representación mediante arreglos

Llamemos pred el arreglo de padres para representar el árbol. Supongamos un nodo p cuyo índice en pred es k. La entrada pred[i] indica que el nodo correspondiente al índice k tiene al valor pred[k] como nodo predecesor en el árbol abarcador de caminos mínimos desde el nodo start\_node. Por ejemplo, el árbol de la figura 7.61 se representa en el arreglo pred como:

| Índice en pred | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----------------|---|---|---|---|---|---|---|---|
| Nodo apuntado  | 4 | 2 | 6 | 4 | - | 2 | 4 | 3 |

Esta figura representa los índices de los nodos en pred, pero recordemos que en nuestra implementación se guardan punteros a nodos.

Si en el transcurso del cálculo se modifica una rama del árbol, entonces basta con modificar el parent del nodo en pred.

#### 7.9.3.3 El algoritmo genérico Bellman-Ford

El principio fundamental del algoritmo de Bellman-Ford reside en lo que se conoce como “relajación de arco”. Considerando que cada nodo almacena la mínima distancia acumulada desde origen, la relajación de un arco  $s \rightarrow t$  consiste en verificar si  $\text{Acc}(s) + w(s \rightarrow t) < \text{Acc}(t)$ . Si el predicado es cierto, entonces se actualiza  $\text{Acc}(t) = \text{Acc}(s) + w(s \rightarrow t)$ , de lo contrario,  $\text{Acc}(t)$  permanece invariable.

El proceso anterior se codifica del siguiente modo:

695  $\langle\text{Relajar arco } 695\rangle \equiv$  (696a)  
 const typename Distance::Distance\_Type & acum\_src = ACU(src);  
 if (acum\_src == Distance::Max\_Distance)  
 continue;  
  
 const typename Distance::Distance\_Type & dist = Distance () (arc);  
 typename Distance::Distance\_Type & acum\_tgt = ACU(tgt);  
 const typename Distance::Distance\_Type sum = Plus () (acum\_src, dist);  
 if (Compare () (sum, acum\_tgt)) //  $\&sum < acum\_tgt$ ?  
 {  
 const int & idx = IDX(tgt);  
 pred[idx] = src;  
 arcs[idx] = arc;  
 acum\_tgt = sum;  
}

En términos del arco  $u \xrightarrow{w} v$ , esta expresión es equivalente a:

- if  $\text{Acc}(v) > \text{Acc}(u) + w \implies$ 
  - $\text{pred}[v] = u$
  - $\text{Acc}(v) = \text{Acc}(u) + w$

La relajación consiste en tratar de disminuir el acumulado del nodo `tgt_node`. Tal disminución sólo ocurre si el predicado  $\text{Acc}(v) = \text{Acc}(u) + w$  es cierto, en cuyo caso, el arreglo `pred` es actualizado.

Cuidado particular debemos prestar con la representación de  $\infty$ . Si se escoge el mayor número representable, entonces la operación  $\text{Acc}(u) + w$  puede acarrear un desborde. De allí pues el `if acum_src == Distance::Max_Distance` que nos supervisa tal posibilidad. Aun así, es posible, aunque esperemos que improbable, tener desbordes si los valores de las distancias son muy grandes, cercanos al máximo.

*Grosso modo*, el cálculo del árbol abarcador consiste en revisar todos los arcos y relajarlos, lo cual se hace del siguiente modo:

696a *(Algoritmo genérico de Bellman-Ford 696a)*≡ (693a 705)

```
for (int i = 0, n = g.get_num_nodes() - 1; i < n; ++i)
 for (Arc_Iterator<GT, SA> it(g); it.has_current(); it.next())
 {
 typename GT::Arc * arc = it.get_current_arc();
 typename GT::Node * src = g.get_src_node(arc);
 typename GT::Node * tgt = g.get_tgt_node(arc);
 (Relajar arco 695)
 }
```

De este código podemos notar sin dificultad que se ejecuta en  $\mathcal{O}(V E)$  veces.

Si cada nodo asocia su propio arreglo `pred`, entonces la superposición de los arreglos conforma la matriz de caminos `path`; exactamente en el mismo estilo que en el algoritmo de Floyd-Warshall.

#### 7.9.3.4 Algoritmo mejorado de Bellman-Ford

El bloque *(Algoritmo genérico de Bellman-Ford 696a)* es simple y, veremos después, correcto y fácilmente adaptable a matrices de adyacencia. Podemos mejorarlo si observamos que cuando en una iteración no se modifiquen los acumulados de un par de nodos conectados por un arco, entonces, en la siguiente iteración, el arco tampoco se relajará. También debemos notar, como en efecto se puede observar en la figura 7.62, que en las primeras iteraciones puede haber nodos cuyos acumulados permanecen invariantes, tendencia que se potencia a medida que el grafo es más grande o más esparcido.

Por tanto, una mejora al algoritmo consiste en procesar sólo aquellos arcos cuyos acumulados de su nodo destino hayan sido modificados y, a la vez, asegurar que no se repita el procesamiento de un arco sin que previamente se hayan procesado el resto de los arcos. Esta conducta de flujo podemos modelizarla mediante la estructura de flujo destinada para ello, es decir, una cola que almacene aquellos nodos cuyo acumulado haya sido modificado. Tal cola se declara como sigue:

696b *(Algoritmo de Bellman-Ford 696b)*≡ (693a) 698b▷

```
DynListQueue<typename GT::Node*> q;
(Colocar centinela 698a)
put_in_queue <GT> (q, start_node);
```

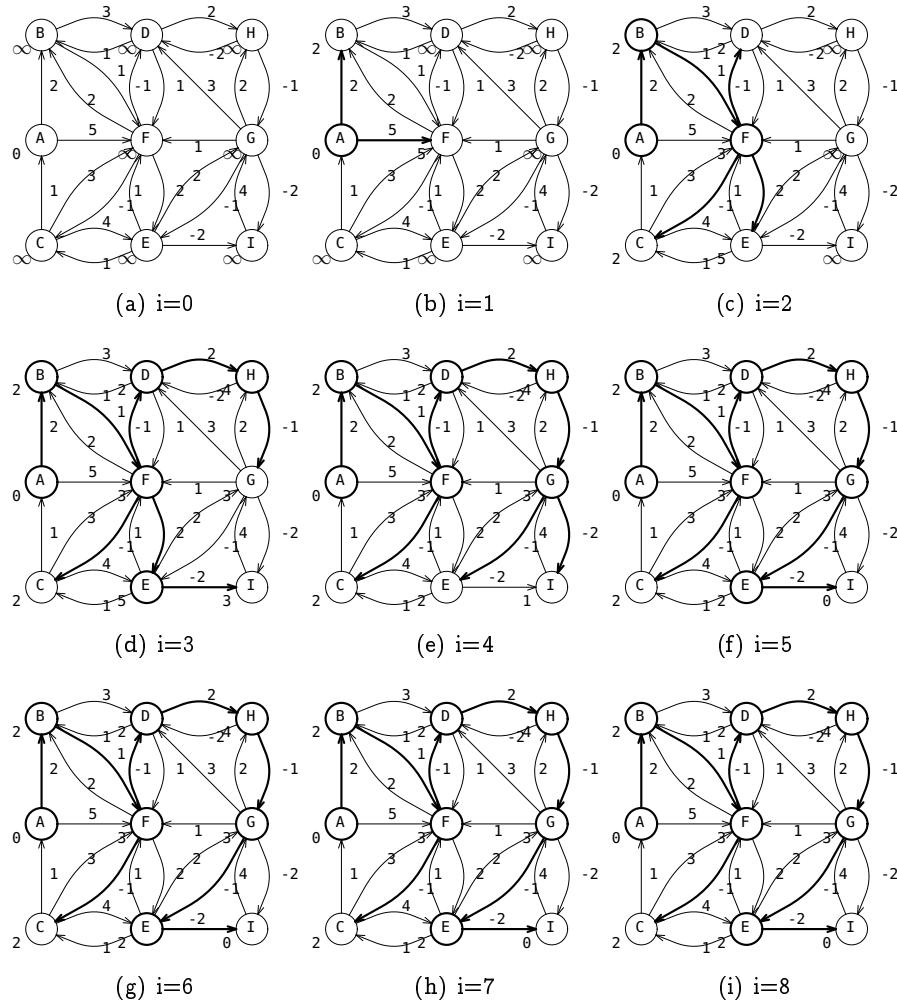


Figura 7.62: Progreso del algoritmo de Bellman-Ford con raíz en el nodo A. Observemos que el árbol abarcador definitivo permanece invariante a partir de la quinta iteración ( $i = 5$ )

La cola contiene el primer nodo cuyo acumulado ha sido modificado y es conocido: el nodo origen. Recordemos que el bit Breadth\_First se inicializó cuando se apartó la memoria para el atributo Bellman\_Ford\_Node\_Info (§ 7.9.3.1 (Pág. 693)).

La rutina `put_in_queue()` encola el nodo y le marca el bit `Breadth_First`, la cual es parte del siguiente trío de rutinas en torno a la cola:

*(Rutinas de cola del algoritmo de Bellman-Ford 697)*

```

697 template <class GT> inline static
 void put_in_queue(DynListQueue<typename GT::Node*> & q, typename GT::Node * p)
 {
 q.put(p);
 NODE_BITS(p).set_bit(Breadth_First, true);
 }

 template <class GT> inline static typename GT::Node *
get_from_queue(DynListQueue<typename GT::Node*> & q)
{
 typename GT::Node * p = q.get();
}

```

```

 NODE_BITS(p).set_bit(Breadth_First, false);
 return p;
}
template <class GT> inline static bool is_in_queue(typename GT::Node * p)
{
 return IS_NODE_VISITED(p, Breadth_First);
}

```

El propósito de manejar la cola con estas rutinas es asegurarse de no tener duplicados en la cola.

El número total de repeticiones que ejecuta el bloque *(Algoritmo genérico de Bellman-Ford 696a)* es exactamente  $V - 1 \times E$  veces, correspondiente a las  $V - 1$  relajaciones sobre los  $E$  arcos. Esta cantidad es importante porque nos acota la máxima cantidad de veces que debería de iterar la versión con cola, diseñada para una representación con listas de adyacencia tal como List\_Digraph. Así, debemos encontrar una manera de “simular” las  $V - 1$  iteraciones. Para hacerlo, Sedgewick [156] propone usar un valor centinela en la cola que nos indique cuando el *(Algoritmo genérico de Bellman-Ford 696a)* habría hecho una relajación de todos los arcos:

698a *(Colocar centinela 698a) ≡* (696b)  
`typename GT::Node __sentinel;
typename GT::Node * sentinel = &__sentinel;
put_in_queue <GT> (q, sentinel);`

De este modo, cada vez que saquemos de la cola sentinel tendremos la seguridad de haber realizado el mismo trabajo que una iteración externa del *(Algoritmo genérico de Bellman-Ford 696a)*.

Así las cosas, el algoritmo mejorado resultante se conforma como sigue:

698b *(Algoritmo de Bellman-Ford 696b) +≡* (693a) ↳ 696b  
`for (int i = 0, n = g.get_num_nodes() - 1; not q.is_empty(); )
{
 typename GT::Node * src = get_from_queue <GT> (q);
 if (src == sentinel) // ¿se saca el centinela?
 if (i++ == n)
 break;
 else
 put_in_queue <GT> (q, sentinel);

 // recorrer y relajar arcos del nodo src
 for (Node_Arc_Iterator<GT, SA> it(src); it.has_current(); it.next())
 {
 typename GT::Arc * arc = it.get_current_arc();
 const typename Distance::Distance_Type & acum_src = ACU(src);
 if (acum_src == Distance::Max_Distance)
 continue;

 typename GT::Node * tgt = it.get_tgt_node();
 const typename Distance::Distance_Type & w = Distance () (arc);
 const typename Distance::Distance_Type sum_src =
 Plus () (acum_src, w);
 typename Distance::Distance_Type & acum_tgt = ACU(tgt);
 if (Compare () (sum_src, acum_tgt)) // relajar arco`

```

 {
 const int & idx = IDX(tgt);
 pred[idx] = src;
 arcs[idx] = arc;
 acum_tgt = sum_src;
 if (not is_in_queue <GT> (tgt))
 put_in_queue <GT> (q, tgt);
 }
}
}

```

De este código es menester comentar:

1. El bit Breadth\_First, el cual es ocultado en las *(Rutinas de cola del algoritmo de Bellman-Ford 697)*, evita que un nodo destino (tgt), cuyo acumulado haya sido modificado en la iteración actual, sea doblemente introducido en la cola.

Por la vía de un arco que aún no haya sido procesado, es posible modificar el acumulado de un nodo que se encuentre en la cola, bien sea de la  $i$ -ésima iteración actual o de alguna  $i - 1$ -ésima iteración anterior.

2. El orden de la cola garantiza que, al menos en la primera iteración ( $i = 0$ ), el recorrido es exactamente uno de amplitud. Posteriormente, en una iteración  $i$ , en función de los nodos que hayan sido metidos en la cola (sólo aquellos cuyo acumulado haya sido modificado), se procesarán igual o menos arcos que en la iteración 0. De aquí se deduce por qué este algoritmo tiende a ser más rápido que el *(Algoritmo genérico de Bellman-Ford 696a)*.

El orden de la cola asegura que se procesen arcos conectados a nodos cuyos acumulados hayan sido modificados, lo cual posibilita una relajación que disminuya el acumulado. Por el contrario, con el *(Algoritmo genérico de Bellman-Ford 696a)* habrá iteraciones que conciernen a arcos cuyos acumulados origen y destino son infinitos y que por tanto no se modificarán a pesar de que el arco sea revisado. Por aquí tenemos otra idea acerca de por qué el algoritmo basado en la cola tiende a ser más eficiente que el genérico.

3. Un arco a relajado cuyo nodo destino ha sido metido en la cola no volverá a visitarse hasta que todos los arcos adyacentes a los nodos anteriores de la cola hayan sido visitados. La iteración 0 visita con certeza todos los arcos, pues los acumulados de los nodos tienen valor infinito, lo cual asegura su relajación. Un arco a relajado en la iteración  $i$  deberá esperar a la iteración  $i + 1$  para eventualmente volver a ser relajado.
4. Si el grafo no contiene ciclos negativos, entonces es seguro que la cola se vaciará, pues cuando el algoritmo estabilice los acumulados a sus valores definitivos, es decir, sus costes mínimos, no habrá más inserciones en la cola.
5. Si, por el contrario, el grafo contiene algún ciclo negativo, entonces los acumulados de los nodos parte del ciclo siempre serán relajados, por lo que el algoritmo no se detendrá por la cola vacía. En este caso, el contador  $i$  alcanzará el valor  $V - 1$ , lo que detendrá el algoritmo.

#### 7.9.3.5 Detección de ciclos negativos

Existen problemas sobre grafos que plantean transformar pesos de arcos de suerte que éstos devenguen negativos o la adición de arcos “ficticios” con pesos negativos. De hecho, desde

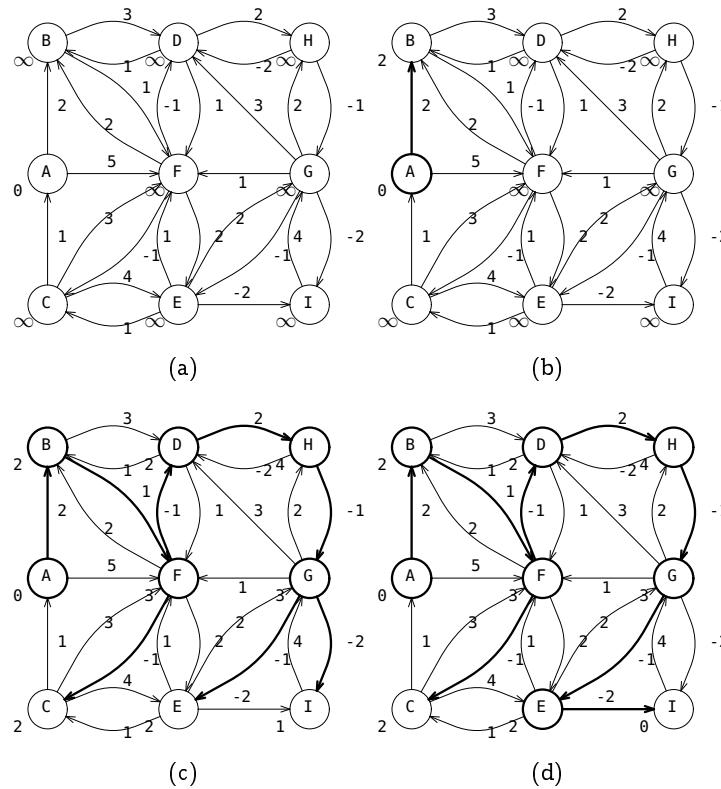


Figura 7.63: Progreso del algoritmo de Bellman-Ford mejorado con raíz en el nodo A. Cada figura corresponde a  $E$  arcos inspeccionados, lo que habida cuenta de la cantidad de figuras respecto a la figura 7.62, ofrece una idea de la aceleración del algoritmo mejorado

la perspectiva de la modelización, es posible imaginar arcos negativos; consiguientemente, cuando menos por detección de error, es plausible modelizar un grafo que contenga ciclos negativos.

¿Cómo saber si existe o no un ciclo? Para atacar esta pregunta es menester tener en cuenta que la máxima longitud en arcos de un camino es a lo sumo  $V - 1$ . Así, la idea del `for` en *(Algoritmo genérico de Bellman-Ford 696a)* es justamente ejecutar  $(V - 1) \times E$  relajaciones, las cuales, en distancia acumulada cubren los caminos en arcos más largos posibles. Luego de la  $V - 1$ -ésima iteración, cada valor  $\text{Acc}(v)$  contiene la mínima distancia desde el origen, por lo que no es posible relajar más arcos.

La consideración anterior es correcta sólo si no hay ciclos negativos, pues si los hay, entonces siempre será posible ejecutar relajaciones adicionales, infinitamente a menudo, que disminuyan el valor  $\text{Acc}(v)$  de cada nodo involucrado en el ciclo.

Consecuentemente, una manera directa de detectar algún ciclo negativo consiste en ejecutar una última iteración en la cual, de ocurrir una relajación, implica la presencia de un ciclo negativo.

De este modo, para detectar si hay un ciclo negativo basta con verificar si es posible realizar al menos una relajación de más:

700  $\langle$ Detención de ciclos negativos 700 $\rangle \equiv$  (693a 705)

```

bool negative_cycle = false;
// recorrer todos los arcos en búsqueda de un ciclo negativo
for (Arc_Iterator<GT, SA> it(g); it.has_current(); it.next())

```

```

{
 typename GT::Arc * arc = it.get_current_arc();
 typename GT::Node * src = g.get_src_node(arc);
 const typename Distance::Distance_Type & acum_src = ACU(src);
 if (acum_src == Distance::Max_Distance)
 continue;

 typename GT::Node * tgt = g.get_tgt_node(arc);
 const typename Distance::Distance_Type & dist = Distance () (arc);
 typename Distance::Distance_Type & acum_tgt = ACU(tgt);
 const typename Distance::Distance_Type sum = Plus ()(acum_src, dist);

 if (Compare () (sum, acum_tgt)) // ¿se relaja arco?
 {
 // si ==> ciclo negativo!
 negative_cycle = true;
 const int & idx = IDX(tgt);
 pred[idx] = src;
 arcs[idx] = arc;
 acum_tgt = sum;
 }
}

```

El bloque opera correctamente para los dos algoritmos. A efectos de que, como veremos posteriormente, el ciclo quede guardado en los arreglos `pred` y `arcs`, es importante culminar enteramente el `for`. Si sólo se trata de saber si existe o no un ciclo negativo, entonces podemos detenernos apenas lo detectemos.

#### 7.9.3.6 Construcción del árbol abarcador

Si no hay ciclos negativos, entonces construimos el árbol abarcador, lo que puede hacerse en una sola pasada recorriendo el arreglo `arcs` e insertando los nodos y arcos a la vez que los mapeamos. Durante esta pasada aprovechamos para liberar la memoria ocupada por los cookies. Para determinar si el cookie almacena un mapeo o un puntero a una estructura `Bellman_Ford_Node_Info` nos valemos del bit `Min`. Si un nodo está marcado con `Min`, entonces su cookie mapea al nodo del árbol abarcador, de lo contrario, el nodo no está en el árbol abarcador y el cookie contiene un puntero a un objeto `Bellman_Ford_Node_Info`:

701 ⟨Construcción árbol abarcador de algoritmo de Bellman-Ford 701⟩≡ (693a)

```

if (negative_cycle) // ¿hay ciclos negativos?
 // liberar memoria de los cookies de los nodos y terminar
for (int i = 0; i < g.get_num_nodes(); ++i)
{
 typename GT::Arc * garc = arcs[i];
 if (garc == NULL)
 continue;

 typename GT::Node * gsrc = g.get_src_node(garc);
 typename GT::Node * tsrc = NULL;
 if (IS_NODE_VISITED(gsrc, Min))
 tsrc = mapped_node<GT> (gsrc);
 else
 {

```

```

 NODE_BITS(gsrc).set_bit(Min, true); // marcar bit
 delete NI(gsrc);
 tsrc = tree.insert_node(gsrc->get_info());
 GT::map_nodes(gsrc, tsrc);
 }

 typename GT::Node * gtgt = g.get_tgt_node(garc);
 typename GT::Node * ttgt = NULL;
 if (IS_NODE_VISITED(gtgt, Min))
 ttgt = mapped_node<GT>(gtgt);
 else
 {
 NODE_BITS(gtgt).set_bit(Min, true); // marcar bit
 delete NI(gtgt);
 ttgt = tree.insert_node(gtgt->get_info());
 GT::map_nodes(gtgt, ttgt);
 }
 typename GT::Arc * tarc = tree.insert_arc(tsrc, ttgt, garc->get_info());
 GT::map_arcs(garc, tarc);
 ARC_BITS(garc).set_bit(Min, true);
}
return false; // no hay ciclos negativos
Uses mapped_node 560b.

```

Con el árbol abarcador `tree` podemos servirnos de la rutina `find_path_depth_first()` estudiada en § 7.5.7.2 (Pág. 597) y así encontrar un camino mínimo desde `start_node` hacia cualquier otro nodo.

#### 7.9.3.7 Análisis del algoritmo de Bellman-Ford

Analizar el algoritmo de Bellman-Ford requiere analizar sus cuatro bloques principales: *(Inicialización de algoritmo de Bellman-Ford 693b)*, *(Algoritmo genérico de Bellman-Ford 696a)*, *(Detección de ciclos negativos 700)* y *(Construcción árbol abarcador de algoritmo de Bellman-Ford 701)*.

*(Inicialización de algoritmo de Bellman-Ford 693b)* requiere recorrer los nodos y de los arcos, lo que arroja una complejidad de  $\mathcal{O}(V + E)$ .

*(Algoritmo genérico de Bellman-Ford 696a)* itera, tal como lo miramos en § 7.9.3.3 (Pág. 695),  $V \times E$  veces.

*(Detección de ciclos negativos 700)* itera a lo sumo  $\mathcal{O}(E)$ . Esto, aunado a *(Algoritmo genérico de Bellman-Ford 696a)* se aproxima a  $\mathcal{O}(V E)$ .

Finalmente, *(Construcción árbol abarcador de algoritmo de Bellman-Ford 701)* toma  $\mathcal{O}(V)$  veces.

Consecuentemente, el algoritmo toma  $\mathcal{O}(V + E) + \mathcal{O}(V E) + \mathcal{O}(V) = \mathcal{O}(V E)$ . La versión *(Algoritmo de Bellman-Ford 696b)* basada en la cola tiene la misma complejidad, aunado a un coste adicional de espacio de  $\mathcal{O}(V)$ ; pero en la práctica, *(Algoritmo de Bellman-Ford 696b)* tiende a ser más veloz que *(Algoritmo genérico de Bellman-Ford 696a)*.

#### 7.9.3.8 Correctitud del algoritmo de Bellman-Ford

**Proposición 7.6** El algoritmo de Bellman-Ford basado en el bloque *(Algoritmo genérico de Bellman-Ford 696a)* es correcto para una grafo sin ciclos negativos.

**Demostración (por inducción sobre la  $i$ -ésima iteración)** La hipótesis inductiva es que luego de la  $i$ -ésima iteración el valor  $\text{Acc}(v)$  es menor o igual al camino más corto desde un nodo origen  $v$  que está compuesto por  $i$  o menos arcos.

Sea  $u \in V$  el nodo origen (`start_node` en el algoritmo) y sea  $v \in V$  un nodo cualquiera. Entonces:

1.  $i = 0$ : Luego de la primera iteración,  $v$  puede englobarse bajo dos casos:
  - (a) Si existe un arco  $u \xrightarrow{w} v$ , entonces  $\text{Acc}(v) = w$ , el cual es el mínimo camino posible compuesto por un solo arco.
  - (b) Si no existe arco directo entre  $u$  y  $v$ , entonces  $\text{Acc}(v) = \infty$  y la cantidad de arcos involucrada es nula; lo que no viola la hipótesis inductiva.
2.  $i > 0$ : Ahora asumimos que la hipótesis inductiva es cierta para todo  $i$  y verificamos si aún lo es para  $i + 1$ . Podemos distinguir dos casos entre caminos desde  $u$  hacia  $v$ :
  - (a) El valor  $\text{Acc}(v)$  ya se corresponde con la distancia mínima desde  $u$  hacia  $v$ . En este caso ya se tiene un camino mínimo compuesto por  $i$  o menos arcos y ninguna relajación hacia  $v$  afectará el valor  $\text{Acc}(v)$ .
  - (b) En el caso contrario, existe un nodo  $w$  adyacente a  $v$  cuyo arco al relajarlo disminuye  $\text{Acc}(v)$ . Este camino puede interpretarse del siguiente modo:



Por la hipótesis inductiva,  $\text{Acc}(w)$  contiene el coste mínimo desde  $u$  hacia  $w$  posible con  $i$  o menos arcos. Del bloque *<Algoritmo genérico de Bellman-Ford 696a>* vemos que la iteración  $i + 1$  observará todos los arcos de  $w$  y seleccionará el arco que va hacia  $v$  actualizando su  $\text{Acc}(v)$  al mínimo ■

#### 7.9.3.9 Búsqueda de ciclos negativos

En ocasiones no sólo es necesario saber si existe o no un ciclo, sino determinarlo exactamente (nodos y arcos). ¿Cómo hacerlo?

Cuando ejecutamos el algoritmo de Bellman-Ford sobre un grafo con al menos un ciclo negativo, las relajaciones de los arcos componentes del ciclo siempre disminuyen los valores  $\text{Acc}$  de sus nodos. Cada vez que se relaja un arco, el padre de  $v$  se actualiza en el arreglo `pred[]`. Como esto sucede para todos los nodos, incluidos los del ciclo, el arreglo `pred[]` contiene el (o los) ciclo(s) en cuestión.

La reflexión anterior nos revela una alternativa para detectar un ciclo, la cual consiste en buscar “periódicamente” en el arreglo `pred[]` la existencia de un ciclo. Si lo encontramos, entonces detenemos el cálculo del árbol abarcador y extraemos el ciclo del arreglo.

Si `pred[]` contiene el ciclo, entonces, ¿cómo buscarlo?. Por simplicidad, sustentada en previos desarrollos, nos valdremos de la clase `Compute_Cycle_In_Digraph()` basada en el cálculo de componentes fuertemente conexos según el algoritmo de Tarjan en § 7.7.5 (Pág. 650). Pero esta clase requiere que el digrafo esté representado en una derivación de `List_Digraph` y no en los arreglos `pred[]` y `arcs[]`. Para calcular el ciclo debemos pues construir un objeto de tipo `List_Digraph` y sobre él encontrar el ciclo. El procedimiento puede resumirse así:

1. Construir un grafo auxiliar a partir de los arreglos `pred[]` y `arcs[]`. Llamemos `aux` a este grafo, el cual debe estar mapeado al grafo con ciclos.  
`aux` contiene con certitud un ciclo, pues está construido a partir de los arreglos `pred[]` y `arcs[]` resultantes de la detección de un ciclo por el algoritmo de Bellman-Ford.
2. Emplear la clase `Compute_Cycle_In_Digraph`, esbozada en § 7.7.5 (Pág. 651), sobre el grafo auxiliar para ubicar uno de sus ciclos (puede haber más). Sea `path` tal ciclo.
3. El ciclo sobre `g` se define por el mapeo de `path` en `g`.

Entendido este procedimiento, planteemos la siguiente rutina:

```
704 <Rutinas de algoritmo de Bellman-Ford 693a>+≡ ◁693a 705▷
 template <class GT>
 bool search_cycle(GT & g, DynArray<typename GT::Node *> & pred,
 DynArray<typename GT::Arc *> & arcs, Path<GT> & cycle)
 {
 const size_t & n = pred.size(); // cantidad de nodos de g
 DynMapAvlTree<typename GT::Node*, typename GT::Node*> nodes_table;
 GT aux;
 for (int i = 0; i < n; ++i) // insertar nodos en aux
 {
 typename GT::Node * p = pred.access(i);
 if (p == NULL or NODE_COOKIE(p) != NULL) // ¿insertado y mapeado?
 continue; // si ==> avance al siguiente

 typename GT::Node * q = aux.insert_node(p->get_info());
 GT::map_nodes(p, q);
 nodes_table.insert(q, p);
 }

 for (int i = 0; i < n; ++i) // insertar arcos en aux
 {
 typename GT::Arc * a = arcs.access(i);
 if (a == NULL or ARC_COOKIE(a) != NULL)
 continue;

 typename GT::Node * gsrc = g.get_src_node(a);
 typename GT::Node * gtgt = g.get_tgt_node(a);
 if (NODE_COOKIE(gsrc) == NULL or NODE_COOKIE(gtgt) == NULL)
 continue; // src o tgt no está en pred ==> no en ciclo

 typename GT::Node * aux_src = mapped_node<GT> (gsrc);
 typename GT::Node * aux_tgt = mapped_node<GT> (gtgt);
 aux.insert_arc(aux_src, aux_tgt);
 }
 Path<GT> path; // buscar ciclo en aux mediante algo de Tarjan
 if (not Compute_Cycle_In_Digraph <GT> () (aux, path))
 return false; // no existe ciclo

 cycle.set_graph(g); // mapear ciclo en aux al camino cycle en g
 }
```

```

 for (typename Path<GT>::Iterator it(path); it.has_current(); it.next())
 cycle.append(nodes_table[it.get_current_node()]);

 return true;
}

```

Uses DynArray 34, mapped\_node 560b, and Path 578a.

search\_cycle() busca en el subgrafo de  $g$  definido por los arreglos pred[] y arcs[], resultantes de la ejecución del algoritmo de Bellman-Ford, la presencia de un ciclo. Si el ciclo es encontrado, entonces la función retorna true y guarda en el camino cycle el ciclo. De lo contrario, retorna false.

Las interfaces que hemos desarrollado en torno al algoritmo de Bellman-Ford indican si existe o no un ciclo negativo, pero no lo determinan. Puesto que identificar ciclos negativos es una tarea plausible para otros algoritmos, el algoritmo de Bellman-Ford puede emplearse para calcularlos. En ese sentido diseñaremos la rutina siguiente:

705     ⟨Rutinas de algoritmo de Bellman-Ford 693a⟩+≡                          △704 706▷

```

template <class GT, class Distance,
 class Compare = Aleph::less<typename Distance::Distance_Type>,
 class Plus = Aleph::plus<typename Distance::Distance_Type>,
 class SA = Default_Show_Arc<GT> > inline
bool bellman_ford_negative_cycle(GT & g, typename GT::Node * start_node,
 Path<GT> & cycle)
{
 GT tree; // árbol sobre el cual opera algoritmo de Bellman-Ford
 ⟨Inicialización de algoritmo de Bellman-Ford 693b⟩
 ⟨Algoritmo genérico de Bellman-Ford 696a⟩
 ⟨Detección de ciclos negativos 700⟩
 if (not negative_cycle)
 return false;

 Search_Cycle <GT> () (g, pred, arcs, cycle);

 return true;
}

```

Uses Path 578a.

Como apreciamos, la rutina es esencialmente igual que q\_bellman\_ford\_min\_spanning\_tree(). El añadido es que no se calcula el árbol abarcador, sólo se determina si existen o no ciclos; de existir, entonces se retorna, en el parámetro cycle, el ciclo donde se detectó la negatividad.

Como ya indicamos al principio de la subsección, una variante de esta técnica estriba en verificar la existencia del ciclo antes de llegar a la última iteración. Para que esta alternativa sea competitiva con la presente recién presentada, es necesario que la búsqueda del ciclo se realice sobre el arreglo pred sin necesidad de construir un grafo auxiliar. También es importante seleccionar el criterio de periodicidad, el cual suele ser una iteración completa a partir de, por ejemplo, la  $V/2$ -ésima iteración.

#### 7.9.3.10 Cálculo de ciclos negativos en un digrafo

La rutina anterior sólo calcula ciclos en el componente fuertemente conexo al que pertenezca start\_node, lo cual no responde absolutamente si  $g$  tiene o no ciclos nega-

tivos. Si se pretende generalidad y asegurar si existen o no ciclos negativos en un digrafo cualquiera, entonces, aunque la técnica sea fundamentalmente la misma, debemos llevar a cabo un poco más trabajo.

Básicamente, la técnica que emplearemos para buscar ciclos negativos se resume en:

1. Obtenga  $C = \{\text{Componentes fuertemente conexos}\}$
  2.  $\forall c \in C \implies$ 
    - (a) Sea `there_is_is_cycle = Bellman_Ford_Negative_Cycle()` ejecutado sobre  $c$  a partir de un nodo cualquiera de  $c$ .
    - (b) Si `there_is_cycle`  $\implies$ 
      - i. Mapee el ciclo obtenido por `Bellman_Ford_Negative_Cycle()` hacia el digrafo original  $g$
      - ii. Terminar con valor de retorno `true`
  3. Terminar con valor de retorno `false`

Para el primer paso podemos perfectamente usar la rutina Tarjan\_Connected\_Components( $g$ ,  $list$ ), desarrollada en § 7.7.3.2 (Pág. 642), lo que nos permite definir:

706    *⟨Rutinas de algoritmo de Bellman-Ford 693a⟩* + ≡ □ 70

```

template <class GT, class Distance,
 class Compare = Aleph::less<typename Distance::Distance_Type>,
 class Plus = Aleph::plus<typename Distance::Distance_Type>,
 class SA = Default_Show_Arc<GT> > inline
bool bellman_ford_negative_cycle(GT & g, Path<GT> & cycle)
{
 DynDlist<DynDlist<typename GT::Node*>> blocks;

 Tarjan_Connected_Components <GT, SA> () (g, blocks);

 // recorrer todos los bloques
 for (typename DynDlist<DynDlist<typename GT::Node*>>::Iterator
 it(blocks); it.has_current(); it.next())
 {
 DynDlist<typename GT::Node*> & block = it.get_current();
 if (block.size() == 1)
 continue;

 typename GT::Node * src = block.get_first(); // nodo inicio
 if (bellman_ford_negative_cycle<GT,Distance,Compare,Plus,SA>
 (g, src, cycle))
 return true;
 }
 return false;
}

```

Uses DynList 85a and Path 578a.

Una técnica alterna consiste en añadir un nodo auxiliar  $s$ , conectarlo al resto de los nodos con valor de peso cero y ejecutar el algoritmo de Bellman-Ford desde  $s$ . El ciclo en cuestión se encontrará en el arreglo  $\text{pred}$  y será localizable mediante el algoritmo de Tarjan.

#### 7.9.4 Discusión sobre los algoritmos de caminos mínimos

El problema de camino mínimo es uno de los problemas resolubles más importantes de los grafos. Entre los algoritmos estudiados y otros existentes, ¿cuándo y cómo seleccionar uno? Iniciemos nuestra discusión presentando la siguiente tabla recapitulativa de tiempos de ejecución y espacio:

| Algoritmo    | Representación         | Desempeño              | Espacio            |
|--------------|------------------------|------------------------|--------------------|
| Dijkstra     | Listas de adyacencia   | $\mathcal{O}(E \lg V)$ | $\mathcal{O}(V)$   |
| Floyd        | Matrices de adyacencia | $\mathcal{O}(V^3)$     | $\mathcal{O}(V^2)$ |
| Bellman-Ford | Listas de adyacencia   | $\mathcal{O}(V E)$     | $\mathcal{O}(V)$   |

Hay varios criterios para atacar esa pregunta. Uno primero está determinado por el hecho de que el grafo sea o no dirigido. Si se trata de un grafo sin pesos negativos, entonces, debido a su mejor rendimiento, el algoritmo de Dijkstra es la escogencia de facto. Los grafos ponderados “naturales” no tienen pesos negativos; por “natural” entendemos que el grafo modeliza directamente una situación de la vida real y no es una consecuencia de alguna transformación matemática. Por ejemplo, en cualquier grafo euclíadiano, es decir, en uno que represente distancias euclidianas, o en un grafo temporal, o sea, uno que modelice duraciones, el algoritmo de Dijkstra es la mejor opción para calcular el camino mínimo entre un par de nodos.

La existencia de pesos negativos descarta de plano el algoritmo de Dijkstra. En esta situación, el algoritmo de Bellman-Ford es la preferencia, pues no sólo exhibe mejor tiempo, sino que detecta ciclos negativos; aunque esto último es más una consideración de validación que de realidad matemática, no se paga una coste significativo por la verificación.

Planteadas las reflexiones anteriores nos aparece entonces la siguiente pregunta: ¿cuándo usar el algoritmo de Floyd-Warshall? Una primera respuesta consiste en decir: cuando se requieran calcular todos los pares de caminos mínimos. A tales efectos, conviene comparar los otros algoritmos presentados para todos los pares de caminos mínimos:

| Algoritmo    | Desempeño esparcido      | Desempeño denso          | Espacio                             |
|--------------|--------------------------|--------------------------|-------------------------------------|
| Dijkstra     | $\mathcal{O}(V E \lg V)$ | $\mathcal{O}(V^3 \lg V)$ | $\mathcal{O}(V^2) + \mathcal{O}(V)$ |
| Floyd        | $\mathcal{O}(V^3)$       | $\mathcal{O}(V^3)$       | $\mathcal{O}(V^2)$                  |
| Bellman-Ford | $\mathcal{O}(V^2 E)$     | $\mathcal{O}(V^4)$       | $\mathcal{O}(V^2) + \mathcal{O}(V)$ |

Para grafos densos en los que  $E \approx V^2$  el algoritmo de Dijkstra es más costoso que el de Bellman-Ford, pero este último es todavía más costoso que el de Floyd-Warshall. Así las cosas, el algoritmo de Floyd-Warshall es la escogencia cuando requiramos calcular todos los pares de caminos mínimos y el grafo sea denso, mientras que el algoritmo de Bellman-Ford lo es cuando el grafo sea esparcido.

Pero la pregunta y reflexión previas no tienen sentido sin la pregunta: ¿cuándo se requieren calcular todos los pares de caminos mínimos? Grossó modo, existen dos situaciones: cuando se requiera disponer del mínimo entre cualquier par de nodos o cuando se requiera conocer el diámetro de una red.

En grafos, el diámetro se define como el camino simple más largo. La respuesta a esta interrogante la proporciona el más largo camino entre un par de nodos, dato que se corresponde con el máximo valor de la matriz de costes arrojada por el algoritmo de Floyd-Warshall o por el algoritmo de Bellman-Ford en su versión para todos los caminos mínimos.

## 7.10 Redes de flujo

Las cosas que nos son muy instituidas devienen parte de nuestro fondo y trasfondo. Es quizá por eso que tendemos a no distinguirlas y, mucho menos, a reconocer la trascendencia que algunas de ellas tienen sobre nuestra vida. Una de esas cosas, ubicua, fundamental para nuestro modo de vida, es el “tubo” o “tubería” con el fluido que éste lleva y la red de transporte, llamada “de tuberías” que esta “tecnología” conlleva.

Debe sernos reminiscente un parangón entre una red de tuberías y un grafo: las tuberías son arcos y los empalmes que juntan dos o más tuberías son nodos. Puesto que la red transporta un fluido, también debemos haber aprehendido que se trata de un grafo dirigido, pues el fluido circula en una sola dirección. A un digrafo que de alguna forma modelice una red de tuberías y alguna clase flujo circulante se le llama “red”.

### 7.10.1 Definiciones y propiedades fundamentales

**Definición 7.8 (Red capacitada)** Una red es un digrafo  $G = \langle V, E \rangle$  conexo conformado por la quíntupla  $\langle V, E, s, t, C \rangle$  donde:

1.  $s \in V$  es el nodo fuente u origen ( $g_{\text{in}}(s) = 0$ ).
2.  $t \in V$  es el nodo sumidero o terminal ( $g_{\text{out}}(t) = 0$ ).
3.  $C : E \rightarrow \mathbb{Z}^+$  es una función de capacidad que le asigna a cada arco  $e \in E$  una capacidad de flujo  $\text{cap}(e)$  circulante por el arco<sup>25</sup>.

Por lo general,  $C = \mathbb{Z}^+$ , es decir, las capacidades son enteros positivos. Sin embargo, pueden ser perfectamente racionales, irracionales o alguna otra clase de anillo en el sentido algebraico que represente una magnitud de flujo.

En algunos contextos, a esta clase de red se le llama “red capacitada”.

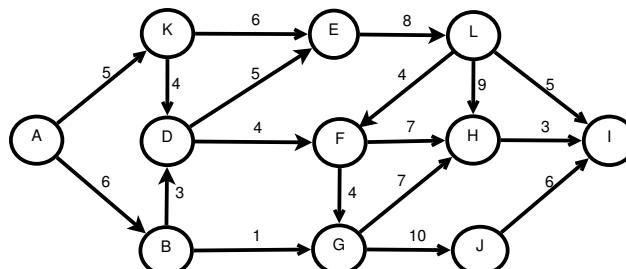


Figura 7.64: Una red

<sup>25</sup>Tubo o canal, etcétera, según sea el caso.

En términos más familiares con el devenir de nuestro discurso, una red es un digrafo con pesos donde se especifica un nodo fuente y uno sumidero. La figura 7.64 ilustra un ejemplo en el cual el nodo A es el fuente, I el sumidero y los pesos de los arcos las capacidades de cada tubería.

En una red es importante denotar objetivamente los conjuntos de arcos salientes y entrantes de un nodo. En ese sentido, dado un nodo  $v \in V$ ,  $IN(v)$  es el conjunto de arcos entrantes mientras que  $OUT(v)$  el de salientes.

En situaciones de la vida real pueden plantearse multiredes, es decir, un multidigrafo. Esto se corresponde con el hecho de tener dos o más tuberías entre un par de nodos, lo cual podría ser un poco problemático de manejar, sobre todo si usamos matrices; pero en términos de modelización, la añadidura de una tubería redundante se abstrae con un aumento de la capacidad del arco entre los nodos involucrados, mientras que un corte de tubería significa una disminución.

**Definición 7.9 (Flujo factible)** Sea  $N = < V, E, s, t, C >$  una red de flujo. Un flujo factible es una función  $f : E \rightarrow C$  que le asigna un valor  $f(e)$  a cada arco, denominado “flujo circulante” que satisface las siguientes condiciones:

1. Condición de capacidad:

$$\forall e \in E \implies 0 \leq f(e) \leq \text{cap}(e) \quad (7.10)$$

2. Condición de conservación del flujo:

$$\forall v \in V | v \neq s \wedge v \neq t \implies \sum_{e \in IN(v)} f(e) = \sum_{e \in OUT(v)} f(e) \quad (7.11)$$

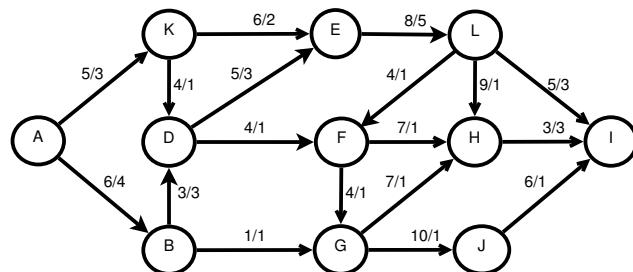


Figura 7.65: Un flujo factible sobre la red de la figura 7.64

La figura 7.65 muestra un ejemplo de un flujo factible sobre la red de la figura 7.64. La capacidades se ubican a la izquierda y el flujo circulante a la derecha. Según la índole del conjunto que represente los flujos, los valores en los arcos pueden separarse mediante comas, barras o simplemente espacios. A excepción del fuente y del sumidero todo nodo satisface la condición de conservación de flujo.

**Definición 7.10 (Valor de flujo)** El valor de flujo en una red  $N = < V, E, s, t, C >$  con flujo  $f$ , denotado como  $F(N, f)$  se define como:

$$F(N, f) = \sum_{e \in OUT(s)} f(e) = \sum_{e \in IN(t)} f(e) \quad (7.12)$$

En otras palabras, el valor de flujo es la cantidad de flujo que deja el fuente, la cual debe ser la misma que llega al sumidero.

En la figura 7.65 el valor de flujo es 7.

### 7.10.2 El TAD Net\_Graph

Las redes capacitadas modelizan situaciones reales de los mundos natural y tecnológico. Puesto que una red es un grafo dirigido, nos valdremos del TAD List\_Digraph. El TAD resultante lo llamaremos Net\_Graph.

Algebraicamente, en una red debemos manejar dos valores nominales: el cero o elemento neutro de la suma y el infinito, los cuales definimos de la siguiente manera:

710a *(Miembros de Net\_Graph 710a)≡* (711b) 711a▷  
    mutable Flow\_Type Infinity;

El añadido esencial de Net\_Graph reside en que los arcos manejen la idea de capacidad y flujo. Para eso definimos un nuevo tipo de arco, heredado de Graph\_Arc, para ser usado con Net\_Graph:

710b *(Arco de red 710b)≡*  
    template <typename Arc\_Info, typename F\_Type = double>  
    class Net\_Arc : public Graph\_Arc<Arc\_Info>  
    {  
        typedef F\_Type Flow\_Type;  
        Flow\_Type cap;  
        Flow\_Type flow;  
        *(Miembros de Net\_Arc 723)*  
    };

La clase Net\_Arc sólo se destina a modelizar e instrumentar algunas validaciones, no se destina a instanciarse por el usuario ni accederse mediante alguno de sus métodos.

A efectos de desempeño en tiempo, en detrimento de un poco de espacio pero en el interés de la validación, definimos la clase Net\_Node derivada de Graph\_Node:

710c *(Nodo de red 710c)≡*  
    template <typename Node\_Info, typename F\_Type = double>  
    class Net\_Node : public Graph\_Node<Node\_Info>  
    {  
        typedef F\_Type Flow\_Type;  
        size\_t in\_degree;  
        Flow\_Type out\_cap;  
        Flow\_Type in\_cap;  
        Flow\_Type out\_flow;  
        Flow\_Type in\_flow;  
  
        Net\_Node()  
         : in\_degree(0), out\_cap(0), in\_cap(0), out\_flow(0), in\_flow(0) {}  
    };

in\_degree almacena la cantidad de arcos entrantes (la de salientes ya se encuentra desde Graph\_Node); estas cantidades sirven para determinar en  $\mathcal{O}(1)$  nodos fuentes y sumideros. out\_cap es la capacidad de salida; o sea la suma de todas las capacidades de los arcos de salida, mientras que in\_cap es la suma de las de entrada. out\_flow es la suma total de flujo entrante mientras que in\_flow la del saliente. Estas magnitudes permiten verificar

la condición de conservación de flujo, según (7.11), mediante el predicado `out_flow == in_flow`. Los atributos a un nodo de red son observables mediante las siguientes primitivas:

711a *(Miembros de Net\_Graph 710a)*+≡ (711b) ◁710a 711c▷

```
Flow_Type get_in_cap(Node * node) const { return node->in_cap; }

Flow_Type get_out_cap(Node * node) const { return node->out_cap; }

size_t get_in_degree(Node * node) const { return node->in_degree; }

size_t get_out_degree(Node * node) const { return get_num_arcs(node); }

Flow_Type get_out_flow(Node * node) const { return node->out_flow; }

Flow_Type get_in_flow(Node * node) const { return node->in_flow; }
```

A parte la información de atributos `Node_Info` y `Arc_Info`, debemos haber notado que las clases `Net_Arc` y `Net_Node` manejan como parámetro plantilla un tipo aritmético llamado `F_Type`, exportado por ambas clases bajo el sinónimo `Flow_Type`. Por omisión, `Flow_Type` es `long`, pero bien pudiera ser `float` o cualquier otro tipo algebraico.

La red se define mediante el tipo `Net_Graph` cuya especificación general es como sigue:

711b *(Net\_Graph 711b)*+≡

```
template <class NodeT = Net_Node<Empty_Class, double>,
 class ArcT = Net_Arc<Empty_Class, double> >
class Net_Graph : public List_Digraph<NodeT, ArcT>
{
 typedef List_Digraph<NodeT, ArcT> Digraph;
 typedef ArcT Arc;
 typedef NodeT Node;
 typedef typename Arc::Flow_Type Flow_Type;
 typedef typename Node::Node_Type Node_Type;
 typedef typename Arc::Arc_Type Arc_Type;
 (Miembros de Net_Graph 710a)
};
```

Uses `List_Digraph 547a`.

Los sinónimos de tipos tienen los mismos sentidos que para los tipos de grafo `List_Graph` y `List_Digraph`.

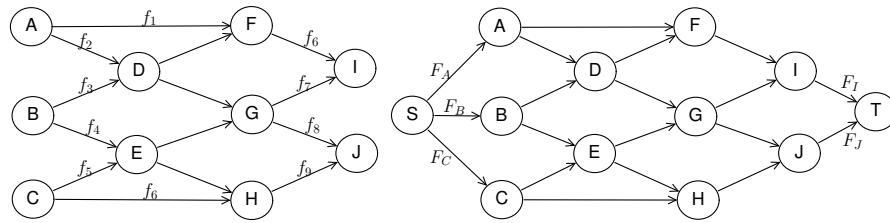
### 7.10.3 Manejos de varios fuentes o sumideros

Es claro que durante la construcción de una red habrá varios fuentes y sumideros. También es plausible, tanto a nivel del propio TAD como del físico, tener una red con varios fuentes y sumideros.

Para conocer rápidamente los nodos fuentes y sumideros se utilizan sendos conjuntos dinámicos cuya implantación se basa en el tipo `set<T>` de la biblioteca estándar C++, el cual, en *ALÉPH* está basado en árboles binarios de búsqueda aleatoriamente equilibrados del tipo treap estudiado en § 6.3 (Pág. 464):

711c *(Miembros de Net\_Graph 710a)*+≡ (711b) ◁711a 712a▷

```
Aleph::set<Node*> src_nodes;
Aleph::set<Node*> sink_nodes;
```



(a) Red con varios fuentes y sumideros  
(b) La misma red con un supra fuente y sumidero

Figura 7.66: Ejemplo de reducción de una red con varios nodos fuentes y sumideros a una con una supra-fuente y un supra-sumidero. La capacidad de los supra-arcos que emanan desde el fuente es  $F_A = \text{cap}(\text{OUT}(A))$ ,  $F_B = \text{cap}(\text{OUT}(B))$ ,  $F_C = \text{cap}(\text{OUT}(C))$ , y la de los que arriban al sumidero  $F_I = \text{cap}(\text{IN}(J))$ ,  $F_J = \text{cap}(\text{IN}(J))$

`src_nodes` almacena la lista de nodos fuente y `sink_nodes` la de sumideros. Cuando se crea un nodo, éste se inserta en ambos árboles. Cuando se inserta un arco se elimina su nodo origen de `sink_nodes` y su destino de `src_nodes`. De este modo, en cualquier momento conocemos los fuentes y sumideros de la red.

Para conocer los nodos fuentes o los sumideros se exportan primitivas de referencia a los conjuntos:

712a *(Miembros de Net\_Graph 710a)*+≡ (711b) ◁ 711c 712b ▷  
Aleph::set<Node\*> & get\_src\_nodes() { return src\_nodes; }  
Aleph::set<Node\*> & get\_sink\_nodes() { return sink\_nodes; }

Por razones “prácticas”, las referencias a estos conjuntos no son constantes, pero éstas no están destinadas a modificación.

A efectos de la simplicidad, tanto matemática como algorítmica, es preferible trabajar con una red que contenga exactamente un solo fuente y un sumidero. Así pues, si la red posee más de un fuente o sumidero, entonces ésta se reduce a colocar un nodo “suprafuente” único, que no es parte lógica de la red, conectado a los fuentes a través de arcos con capacidad igual a la suma de las capacidades de los arcos de salida de todos los fuentes. Del mismo modo, los sumideros se conectan a un solo “suprasumidero” con arcos de capacidad igual a la suma de todas las capacidades de los arcos de entrada de todos los sumideros. La figura 7.66 ilustra un ejemplo.

La añadidura de “supra-nodos” se realiza a través de la siguiente familia de primitivas públicas, de las cuales sólo mostramos la instrumentación de las que conciernen al fuente:

712b *(Miembros de Net\_Graph 710a)*+≡ (711b) ◁ 712a 713a ▷  
void make\_super\_source()  
{  
 DynDlist<Node\*> src\_list;  
 for (typename Aleph::set<Node\*>::iterator it = src\_nodes.begin();  
 it != src\_nodes.end(); ++it)  
 src\_list.append(\*it);  
  
 Node \* super\_source = insert\_node();  
 while (not src\_list.is\_empty())  
 {  
 Node \* p = src\_list.remove\_first();  
 insert\_arc(super\_source, p, get\_out\_cap(p));

```

 }
 with_super_source = true;
}
void unmake_super_source()
{
 remove_node(*src_nodes.begin());
 with_super_source = false;
}
void make_super_sink()

void unmake_super_sink()

void make_super_nodes()
{
 make_super_source();
 make_super_sink();
}
void unmake_super_nodes()
{
 unmake_super_source();
 unmake_super_sink();
}

```

Uses DynDlist 85a.

Estas primitivas, en particular las dos últimas, serían invocadas por un algoritmo de cálculo de flujo máximo al principio y final de sus cálculos.

Como puede apreciarse, las rutinas emplean las banderas lógicas `with_super_source` y `with_super_sink` para indicar si hay o no supra-nodos. Estas banderas son atributos de `Net_Graph`.

Una vez transformada la red mediante `make_super_nodes()`, se garantiza que se tiene un solo fuente y un solo sumidero. Así pues, vale la pena disponer de dos observadores:

713a *(Miembros de Net\_Graph 710a) +≡* (711b) ▷ 712b 713b ▷  
*Node \* get\_source() { return \*get\_src\_nodes().begin(); }*  
*Node \* get\_sink() { return \*get\_sink\_nodes().begin(); }*

#### 7.10.4 Operaciones topológicas sobre una red capacitada

`Net_Graph` requiere sobrecargar y extender algunas funciones de `List_Digraph`, específicamente las relacionadas con el manejo de nodos y arcos. Comencemos por la inserción de un nodo:

713b *(Miembros de Net\_Graph 710a) +≡* (711b) ▷ 713a 714a ▷  
*Node \* insert\_node(const Node\_Type & node\_info)*  
*{*  
 *Node \* p = Digraph::insert\_node(node\_info);*  
 *src\_nodes.insert(p);*  
 *sink\_nodes.insert(p);*  
 *return p;*  
*}*

La rutina apela a la maquinaria de `List_Digraph`, cuyo sinónimo dentro de `Net_Graph` es `Digraph`. La sobrecarga extiende el añadir el nuevo nodo a los conjuntos `src_nodes` y

`sink_nodes.`

Ahora veamos la extensión para la inserción de un arco:

714a *(Miembros de Net\_Graph 710a) +≡* (711b) ◁ 713b 714b ▷

```

virtual Arc * insert_arc(Node * src_node, Node * tgt_node,
 const typename Arc::Arc_Type & arc_info,
 const Flow_Type & cap, const Flow_Type & flow)
{
 // inserción en clase base
 Arc * arc = Digraph::insert_arc(src_node, tgt_node, arc_info);

 src_nodes.erase(tgt_node); // actualización de source/sink
 sink_nodes.erase(src_node);

 arc->cap = cap; // ajuste capacidad y flujo de arco
 arc->flow = flow;

 tgt_node->in_degree++; // actualizar info de control de nodo
 src_node->out_cap += cap;
 tgt_node->in_cap += cap;
 src_node->out_flow += flow;
 tgt_node->in_flow += flow;

 return arc;
}

```

`insert_arc()` sobrecargado inicializa el estado de los flujos en los nodos origen y destino; así como el grado de entrada del nodo destino. Una vez que existe un arco  $\text{src\_node} \rightarrow \text{tgt\_node}$ , `src_node` deja con certitud de ser un sumidero, mientras que `tgt_node` deja de ser fuente; de ahí las eliminaciones de los conjuntos `src_nodes` y `sink_nodes`.

A efectos de versatilidad, `Net_Graph` exporta versiones sobrecargadas de `insert_arc()` con distintas presentaciones para parámetros de capacidad y flujo, de las cuales, quizás la más interesante sea:

714b *(Miembros de Net\_Graph 710a) +≡* (711b) ◁ 714a 714c ▷

```

virtual Arc *
insert_arc(Node * src_node, Node * tgt_node, const Flow_Type & cap)
{
 return insert_arc(src_node, tgt_node, Arc_Type(), cap, 0);
}

```

La eliminación de un arco requiere actualizar el grado de entrada del nodo destino y los acumulados de flujos de los nodos, así como la añadidura eventual en los conjuntos `src_nodes` y `sink_nodes`. Por eso en `Net_Graph` también debemos sobrecargar:

714c *(Miembros de Net\_Graph 710a) +≡* (711b) ◁ 714b 715a ▷

```

virtual void remove_arc(Arc * arc)
{
 Node * src = get_src_node(arc);
 Node * tgt = get_tgt_node(arc);

 if (-(tgt->in_degree) == 0)
 src_nodes.insert(tgt); // tgt deviene un nodo fuente
}

```

```

src->out_cap -= arc->cap; // actualizar caps y flujos en src y tgt
src->out_flow -= arc->flow;
tgt->in_cap -= arc->cap;
tgt->in_flow -= arc->flow;

Digraph::remove_arc(arc); // eliminar en clase base

if (get_num_arcs(src) == 0)
 sink_nodes.insert(src); // src deviene un nodo sumidero
}

```

La eliminación de un nodo requiere eliminar de los conjuntos `src_nodes` y `sink_nodes` (en caso de que el nodo se encuentre en alguno de ellos), pero también actualizar los estados de los nodos conectados por los arcos del nodo eliminado. En la clase `List_Graph` esto se resuelve llamando sucesivamente a `remove_arc()`. Aquí hay que hacer lo mismo más la eliminación de los conjuntos `src_nodes` y `sink_nodes`. Surge entonces la pregunta ¿podemos reusar `List_Graph::remove_node()`? La respuesta es afirmativa, pues `remove_arc()` es un método virtual desde la clase base `List_Graph`. De este modo, instrumentar `remove_node()` es muy simple:

715a *(Miembros de Net\_Graph 710a)+≡* (711b) ↳ 714c 715b ▷

```

virtual void remove_node(Node * p)
{
 Digraph::remove_node(p); // eliminación en clase base
 src_nodes.erase(p);
 sink_nodes.erase(p);
}

```

La instrucción final `Digraph::remove_node(p)` invocará la eliminación de los arcos de un nodo (ver § 7.3.10.5 (Pág. 573)). Puesto que `remove_arc()` es virtual, `Digraph::remove_node(p)` invoca a `Net_Graph::remove_arc()`.

Muchos problemas de grafos se resuelven a través de redes de flujo. Es pues plausible y probable construir una red a partir de un grafo ya existente, resolver con ella el problema en cuestión y luego regresar al grafo original. Por esta razón es deseable disponer de un constructor copia de digrafo que nos construya una red mapeada:

715b *(Miembros de Net\_Graph 710a)+≡* (711b) ↳ 715a 715c ▷

```

Net_Graph(Digraph & digraph)
{
 Net_Graph::Net_Graph(); // inicializa atributos
 copy_graph(*this, digraph, true); // copia mapeada
}

```

Hay otro constructor `Net_Graph(Net_Graph & net)`, el cual no efectúa la copia mapeada.

Para conocer los atributos de red asociados a un arc se tienen los observadores siguientes:

715c *(Miembros de Net\_Graph 710a)+≡* (711b) ↳ 715b 715d ▷

```

const Flow_Type & get_flow(Arc * arc) const { return arc->flow; }
const Flow_Type & get_cap(Arc * arc) const { return arc->cap; }

```

A veces es deseable “re-iniciar” una red, esto es, poner su valor de flujo en cero:

715d *(Miembros de Net\_Graph 710a)+≡* (711b) ↳ 715c 716 ▷

```

void reset()
{
}

```

```
for (Arc_Iterator<Net_Graph> it(*this); it.has_current(); it.next())
 it.get_current()->flow = 0;

for (Node_Iterator<Net_Graph> it(*this); it.has_current(); it.next())
{
 Node * p = it.get_current();
 p->in_flow = p->out_flow = 0;
}
```

Probablemente, una de las primitivas fundamentales es conocer el valor del flujo de la red:

### 7.10.5 Cortes de red

Algunos problemas de grafos pueden plantearse en términos de una red e inversamente. Un primer vínculo entre estas dos visiones se notará en la próxima definición.

Definición 7.11 (Corte de una red) Sea una red  $N = \langle V, E, s, t, C \rangle$ . Sean

- $V_s = \{s\} \cup \{v \in V \mid v \neq t\}$
  - $V_t = \{t\} \cup \{v \in V \mid v \neq s\}$

Tales que  $V_s \cup V_t = V$   $\wedge V_s \cap V_t = \emptyset$ , es decir, una partición disjunta de  $V$ , denominada  $V_s$ , que contiene al nodo fuente pero no al destino y  $V_t$ , que contiene al nodo destino pero no al fuente.

Un corte de la red  $N$ , denotado  $\langle V_s, V_t \rangle$ , se define por el conjunto de arcos que conectan nodos de  $V_s$  hacia nodos de  $V_t$ .

Notemos que  $\langle V_s, V_t \rangle$  no contiene los arcos que van desde  $V_t$  hacia  $V_s$ . Este conjunto se denota como  $\langle \overline{V_s}, V_t \rangle$

La figura 7.67 ilustra un corte con  $V_s = \{A, B, D, F, G, K\}$  y  $V_t = \{E, H, I, J, L\}$  donde  $<V_s, V_t> = \{(K \rightarrow E), (D \rightarrow E), (F \rightarrow H), (G \rightarrow H), (G \rightarrow J)\}$  y  $<V_s, V_t> = \{(L \rightarrow F)\}$ .

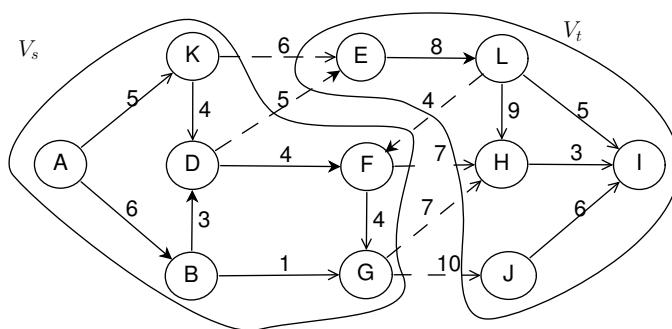


Figura 7.67: Un corte sobre la red de la figura 7.64

A menudo, la literatura de grafos denomina a  $< V_s, V_t >$  como un “corte s-t” por el hecho de que son los nodos fuente y sumidero los que “dominan” la partición.

El asunto esencial del parangón entre la red y el digrafo es el carácter dinámico de un flujo en la tubería. Si, por ejemplo, abrimos una llave y no surte agua, entonces intuimos que hay un corte en alguna parte. La relación de conectividad de un grafo puede estudiarse según esta intuición. Si tenemos un grafo y deseamos averiguar su conectividad, entonces podríamos injectarle un flujo; el fluido permeará a todos los nodos conectados.

**Definición 7.12 (Valor de flujo de un corte)** Sea una red  $N = \langle V, E, s, t, C \rangle$  con flujo  $f$  y un corte cualquiera  $\langle V_s, V_t \rangle$ . El valor de flujo del corte se define como:

$$F(\langle V_s, V_t \rangle, f) = \sum_{\forall e \in OUT(\langle V_s, V_t \rangle)} f(e) - \sum_{\forall e \in IN(\langle V_s, V_t \rangle)} f(e) \quad (7.13)$$

Donde  $OUT(\langle V_s, V_t \rangle)$  son los arcos del corte que van desde  $V_s$  hacia  $V_t$  y, recíprocamente,  $IN(\langle V_s, V_t \rangle)$  son los arcos del corte que van desde  $V_t$  hacia  $V_s$ .

**Proposición 7.7 (Equivalencia de flujo entre una red y un corte)** Sea una red  $N = \langle V, E, s, t, C \rangle$  con flujo  $f$  y un corte cualquiera  $\langle V_s, V_t \rangle$ . Entonces:

$$F(\langle V_s, V_t \rangle, f) = F(N, f) \quad (7.14)$$

Lo cual es equivalente a:

$$\sum_{\forall e \in OUT(s)} f(e) = \sum_{\forall e \in OUT(\langle V_s, V_t \rangle)} f(e) - \sum_{\forall e \in IN(\langle V_s, V_t \rangle)} f(e) \quad (7.15)$$

**Demostración (Por inducción sobre  $n = |V_s|$ )**

- $n = 1$ : en este caso  $V_s = \{s\}$  y  $V_t = V - \{s\}$ . Así,  $\langle V_s, V_t \rangle = OUT(s) \neq \emptyset$  implica directamente que  $F(\langle V_s, V_t \rangle, f) = \sum_{\forall e \in OUT(s)} f(e) = F(N, f)$ .
- $n > 1$ : ahora asumimos que la proposición es cierta para todo  $n$ .

Consideremos un nodo  $u \in \langle V_s, V_t \rangle$ . Según el sentido respecto a los conjuntos  $V_s$  y  $V_t$ , clasifiquemos los arcos de  $u$  en:

- $IN_s(u) = \{\text{arcos que llegan a } u \text{ desde } V_s\}$
- $IN_t(u) = \{\text{arcos que llegan a } u \text{ desde } V_t\}$
- $OUT_s(u) = \{\text{arcos que salen de } u \text{ hacia } V_s\}$
- $OUT_t(u) = \{\text{arcos que salen de } u \text{ hacia } V_t\}$

Los cuatro conjuntos junto con  $\langle V_s, V_t \rangle$  se ilustran genéricamente en la figura 7.68-a.

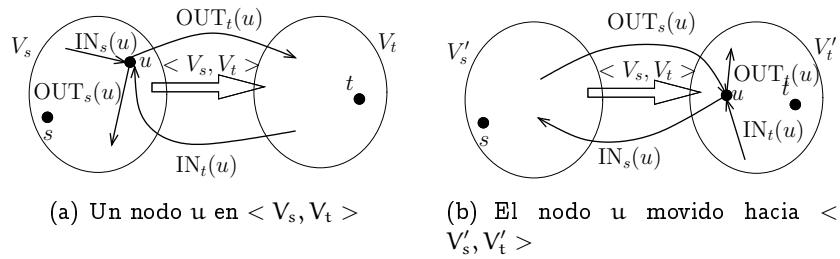
Ahora “movamos” el nodo  $u$  hacia el conjunto  $V_t$ , lo que acarrea un nuevo corte  $\langle V'_s, V'_t \rangle$  ilustrado en la figura 7.68-b.

Así, podemos expresar a  $F(\langle V_s, V_t \rangle, f)$  del siguiente modo:

$$F(\langle V_s, V_t \rangle, f) = F(\langle V'_s, V'_t \rangle, f) - IN_s(u) + OUT_s(u) + OUT_t(u) - IN_t(u) \quad (7.16)$$

Por la hipótesis inductiva  $F(\langle V'_s, V'_t \rangle, f) = F(N, f)$ , pues  $V'_s = V_s - \{u\} \Rightarrow |V'_s| < |V_s|$ . Por la condición de conservación del flujo  $IN_s(u) + IN_t(u) - OUT_s(u) - OUT_t(u) = 0$ . Hacemos las correspondientes sustituciones en (7.16) y obtenemos:

$$F(\langle V_s, V_t \rangle, f) = F(N, f) + 0 = F(N, f) \blacksquare$$

Figura 7.68: Consideración de un  $u \in <V_s, V_t>$ 

Como corolario de la proposición anterior podemos establecer que, dado un corte  $<V_s, V_t>$ , entonces:

$$F(<V_s, V_t>, f) = \sum_{\forall e \in OUT(s)} f(e) = \sum_{\forall e \in IN(t)} f(e) = F(N, f) \quad (7.17)$$

### 7.10.6 Flujo máximo/corte mínimo

La idea de red y flujo cobra mayor interés cuando nos preguntamos ¿cómo encontramos un flujo  $f$  de tal manera que éste sea máximo? Tal flujo máximo se denota como  $f^*$ .

Esencialmente, este es el problema instrumental de esta sección, y lo calificamos de instrumental porque el problema del flujo máximo es vehículo de resolución de una amplia gama de otros problemas. En este sentido nos conviene mostrar el paralelismo entre un flujo máximo y un corte de capacidad mínima, lo cual requiere la siguiente definición.

**Definición 7.13 (Capacidad de un corte)** Sea una red  $N = <V, E, s, t, C>$  y un corte cualquiera  $<V_s, V_t>$ . La capacidad del corte, denotada como  $cap(<V_s, V_t>)$ , se define como la suma de las capacidades de los arcos pertenecientes al corte. O sea:

$$cap(<V_s, V_t>) = \sum_{\forall e \in <V_s, V_t>} cap(e) \quad (7.18)$$

Encontrar el flujo máximo de una red está estrechamente relacionado con encontrar un corte de capacidad mínima por la simple razón de que un flujo máximo debe ser menor o igual que la mínima capacidad posible que se encuentre en un corte de red. En efecto, por la proposición 7.7 sabemos que:

$$\begin{aligned} F(N, f) &\leq \sum_{\forall e \in <V_s, V_t>} cap(e) - \\ &\sum_{\forall e \in <\overline{V_s}, \overline{V_t}>} f(e) = cap(<V_s, V_t>) - \sum_{\forall e \in <\overline{V_s}, \overline{V_t}>} f(e) \end{aligned}$$

pues  $\forall e \in E \implies f(e) \leq cap(e)$ . Por tanto, dado que los flujos son positivos:

$$F(N, f) \leq cap(<V_s, V_t>) \quad . \quad (7.19)$$

Consiguentemente, si  $f^*$  es un flujo máximo, entonces:

$$F(N, f^*) \leq cap(<V_s, V_t>) \quad (7.20)$$

Esto nos conduce a un muy interesante resultado.

**Proposición 7.8** Sea  $f$  un flujo sobre una red  $N = \langle V, E, s, t, C \rangle$  y  $\langle V_s, V_t \rangle$  un corte sobre  $N$ . Si  $F(N, f) = \text{cap}(\langle V_s, V_t \rangle)$  entonces:

1.  $f$  es un flujo máximo y
2.  $\langle V_s, V_t \rangle$  es un corte de capacidad mínima.

**Demostración** Sea  $f^*$  un flujo máximo sobre  $N$  y sea  $\langle V_s, V_t \rangle_{\min}$  el corte mínimo de  $N$ . Pictóricemos genéricamente el corte mínimo de la manera ilustrada en la figura 7.69.

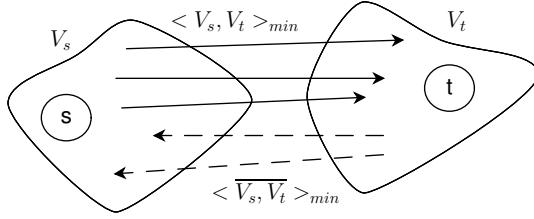


Figura 7.69: Esquema general de  $\langle V_s, V_t \rangle_{\min}$

1. Por definición sabemos que desde  $s$  parte un flujo  $f^*$  con valor de flujo  $F(N, f^*)$ , el cual es exactamente el mismo que llega a  $t$ . Claramente, para que  $f^*$  pueda pasar por  $\langle V_s, V_t \rangle$  se debe cumplir que  $F(N, f^*) \leq \text{cap}(\langle V_s, V_t \rangle)$  y, puesto que la proposición afirma que  $F(N, f^*) = \text{cap}(\langle V_s, V_t \rangle)$ , entonces  $f^*$  es el máximo posible que puede pasar por  $\langle V_s, V_t \rangle$ , lo que corrobora (7.20). La maximalidad de  $f^*$  está pues demostrada  $\square$
2. Supongamos que  $\text{cap}(\langle V_s, V_t \rangle) = K \neq F(N, f^*)$ , pero que  $\langle V_s, V_t \rangle$  no es mínimo. Si así fuese, entonces existiría otro corte con una capacidad  $K' < K$ , pero por esta capacidad no puede fluir  $f^*$ , pues  $F(N, f^*) > K'$ . La contradicción demuestra pues la minimalidad de  $\langle V_s, V_t \rangle$  ■

La proposición nos indica que si calculamos el flujo máximo, entonces la combinación cuyo conjunto de arcos  $\langle V_s, V_t \rangle$  nos dé con capacidad igual al flujo corresponde al corte mínimo.

**Corolario 7.1** Sea  $f^*$  un flujo máximo sobre una red  $N = \langle V, E, s, t, C \rangle$  con corte mínimo  $\langle V_s, V_t \rangle$ . Entonces:

1.

$$\forall e \in \langle V_s, V_t \rangle \implies f(e) = \text{cap}(e) \quad (7.21)$$

2.

$$\forall e \in \overline{\langle V_s, V_t \rangle} \implies f(e) = 0 \quad (7.22)$$

**Demostración**

1. Inmediato de la proposición 7.8 (Pág. 719)  $\square$
2. Puesto que  $\sum_{e \in \text{OUT}(s)} f(e) = \sum_{e \in \text{IN}(t)} f(e) = F(N, f^*)$ , entonces  $F(\langle \overline{V_s, V_t} \rangle, f^*) = 0$ , pues si no se le restaría a  $F(\langle V_s, V_t \rangle, f^*)$  y éste no sería máximo ■

El corolario nos afina más el cómputo de un corte mínimo a partir del conocimiento del flujo máximo. Todo arco que pertenezca a  $\langle V_s, V_t \rangle$  tiene valor de flujo igual a su capacidad. Análogamente, todo arco que pertenezca a  $\langle \overline{V_s}, \overline{V_t} \rangle$  tiene valor de flujo cero. Si a este criterio le aunamos como condición que la suma de las capacidades de los arcos de  $\langle V_s, V_t \rangle$  tiene que corresponderse con el valor de flujo máximo, entonces tenemos un método para calcular el corte mínimo, dado el flujo máximo.

### 7.10.7 Caminos de aumento

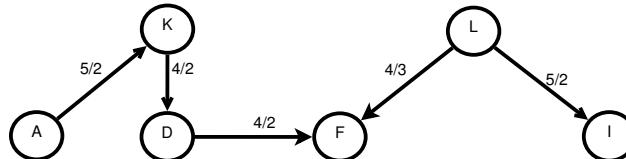
Los algoritmos de cálculo de flujo máximo se basan en encontrar caminos especiales llamados de aumento. Informalmente, un camino de aumento es una secuencia de arcos entre el fuente y el sumidero por la cual se puede aumentar el flujo. Sin embargo, no es un camino tradicional y para aprehender eso requerimos una definición intermedia.

**Definición 7.14 (Semicamino)** Un semicamino en una red  $N = \langle V, E, s, t, C \rangle$ , o cuasicamino, es una secuencia  $\langle s, e_1, v_1, e_2, \dots, e_{k-1}, t \rangle$  de nodos y arcos entre el nodo fuente y el sumidero.

Notemos que los arcos de un semicamino no necesariamente van dirigidos desde el fuente hacia el sumidero; puede haber arcos desde el destino hacia el sumidero.

Un arco desde el fuente hacia el sumidero se llama “arco de adelanto”. Simétricamente, uno desde el sumidero hacia el fuente “arco de retroceso”.

Por ejemplo, en el siguiente semicamino de la red mostrada en la figura 7.64:



Los arcos  $\{(A, K), (K, D), (D, F), (L, I)\}$  son de adelanto, mientras que  $\{(L, F)\}$  es de retroceso.

Con la idea de semicamino podemos definir la de camino de aumento.

**Definición 7.15 (Camino de aumento)** Sea una red  $N = \langle V, E, s, t, C \rangle$ . Un camino de aumento  $C_a$  es un semicamino en  $N$  tal que el flujo de sus arcos de adelanto puede incrementarse y el de sus arcos de retroceso decrementarse.

Si  $C_a$  es un camino de aumento, entonces, por definición, para cualquier arco  $e$  de adelanto  $f(e) \leq \text{cap}(e)$ , pues sino no habría forma de incrementar el flujo. En el mismo sentido, para cualquier arco  $e$  de retroceso  $f(e) > 0$ .

La cantidad en que puede incrementarse el flujo en un arco de adelanto, o decrementarse en uno de retroceso, se denomina “eslabón” (“slack”)<sup>26</sup>, se designa  $\Delta_e$  y se determina de la siguiente manera:

$$\Delta_e = \begin{cases} \text{cap}(e) - f(e) & \text{si } e \text{ es de adelanto} \\ f(e) & \text{si } e \text{ es de retroceso} \end{cases} \quad (7.23)$$

Dado un camino de aumento  $C_a$ , la máxima cantidad en que puede aumentarse el flujo de la red por ese camino está supeditada a la condición de conservación de flujo y se

<sup>26</sup>Este es el término castellano que el autor ha decidido emplear por el término inglés “slack”, el cual se traduce a menudo por “holgura”.

determina como:

$$\Delta_{C_a} = \min_{\forall e \in C_a} \Delta_e ; \quad (7.24)$$

es decir, el “mínimo eslabón” del camino de aumento.

Si encontramos un camino de aumento  $C_a$  sobre una red, entonces es seguro que sobre sus arcos podemos aumentar o disminuir el valor de flujo en el valor del mínimo eslabón. Esto indica el carácter algorítmico de los caminos de aumento en la determinación del flujo máximo.

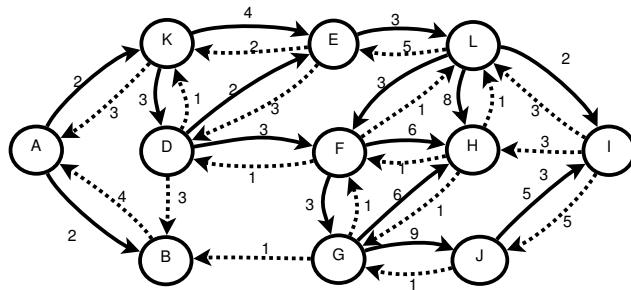


Figura 7.70: Red residual del grafo de la figura 7.65 (Pág. 709). Los arcos de flujo restante son continuos, mientras que los residuales son punteados.

**Definición 7.16 (Red residual)** Sea una red  $N = < V, E, s, t, C >$  con valor de flujo  $F(N, f)$ . El grafo residual de  $N$  denotado como  $\bar{N} = < V, E', s, t, C' >$ , tal que  $E'$  y  $C'$  se construyen del siguiente modo:

1.  $\forall e = (u, v) \in E \Rightarrow$ 
  - (a)  $\exists e' = (u, v) \in E' \mid \text{cap}(e') = \text{cap}(e) - f(e)$  (arco de flujo restante).
  - (b)  $\exists \bar{e} = (v, u) \in E' \mid \text{cap}(\bar{e}) = f(e)$  (arco residual).

La figura 7.70 ilustra la red residual del grafo con valor de flujo mostrado en la figura 7.65.

La red residual nos permite encontrar caminos de aumento mediante búsquedas de caminos, sea por profundidad o por amplitud, desde el fuente hacia el sumidero, en el mismo sentido de los algoritmos estudiados en § 7.5.7.2 (Pág. 597) y § 7.5.8 (Pág. 600). A ese tenor, la figura 7.71-a muestra un eventual camino encontrado por una hipotética búsqueda en profundidad y el correspondiente camino de aumento (7.71-b).

Una vez hallado un camino de aumento podemos incrementar el flujo de la red en el valor de su mínimo eslabón. Para la figura ejemplo 7.65 y el camino de aumento mostrado en la figura 7.71-b con mínimo eslabón  $\Delta_{C_a} = 1$ , podemos modificar el flujo de los arcos del camino de aumento en el valor de  $\Delta_{C_a}$ . Esto nos conduce al valor de flujo de la figura 7.72.

De la modificación efectuada, que deparó en la figura 7.72, nos conviene establecer las siguientes observaciones por separado:

1. El valor del flujo de la red se incrementó de 7 a 8.
2. Las modificaciones sobre el flujo sólo involucran arcos del camino de aumento.

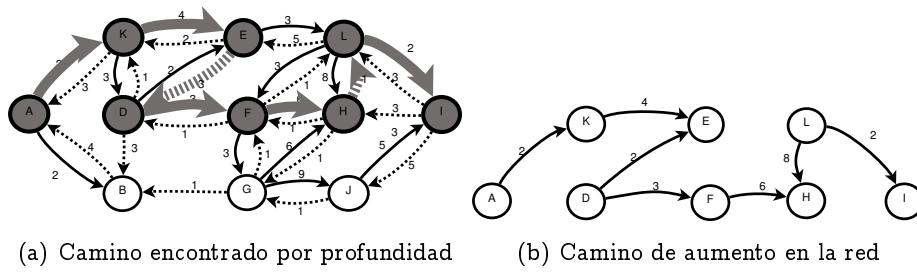
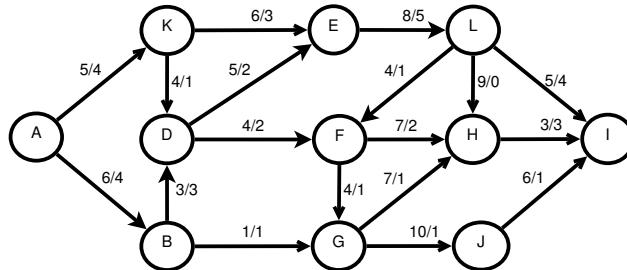


Figura 7.71: Un camino de aumento en la red residual de la figura 7.70.

Figura 7.72: Un flujo factible sobre la red de la figura 7.64 luego de incrementar el flujo en  $\Delta_{C_a} = 1$  correspondiente al mínimo eslabón del camino de aumento mostrado en la figura 7.71-b (pág 722)

3. A pesar de las modificaciones efectuadas, se mantiene la condición de conservación de flujo sobre los nodos que conforman el camino de aumento.

Estas observaciones se generalizan bajo la siguiente proposición.

**Proposición 7.9 (Ford-Fulkerson 1956 [54])** Sea una red  $N = \langle V, E, s, t, C \rangle$  con valor de flujo  $F(N, f)$ . Sea  $C_a$  un camino de aumento sobre la red  $N$ , calculado mediante una búsqueda de  $t$  en la red residual a partir de  $s$  y cuyo mínimo eslabón es  $\Delta_{C_a}$ . Entonces, el flujo de la red puede aumentarse en el valor  $\Delta_{C_a}$  del siguiente modo:

$$f(e) = \begin{cases} f(e) + \Delta_{C_a} & \text{si } e \text{ es un arco de adelanto} \\ f(e) - \Delta_{C_a} & \text{si } e \text{ es un arco de retroceso} \\ f(e) & \text{en cualquier otro caso} \end{cases} \quad (7.25)$$

**Demostración** Los arcos del grafo que son modificados son aquellos que pertenecen a  $C_a$ . Puesto que  $\Delta_{C_a}$  es el valor mínimo entre algún  $\text{cap}(e) - f(e)$  y  $f(e)$  (ver definición 7.15 pág. 720), el incremento sobre un arco de adelanto  $e$  no sobrepasa a  $\text{cap}(e)$ . Del mismo modo, el decremento sobre uno de retroceso  $e$  no es menor que 0. Por tanto, se preserva la condición de capacidad sobre todos los nodos involucrados en  $C_a$ .

Por definición, los extremos de  $C_a$  son  $s$  y  $t$ . Puesto que  $C_a$  fue calculado a partir de  $s$ , es seguro que el primer arco  $s \rightarrow v$  de  $C_a$  es de adelanto; por tanto, el flujo de la red se aumenta al incrementar el flujo en el arco de adelanto en el valor  $\Delta_{C_a}$ . Similamente, el último arco de  $C_a$ , con forma  $w \rightarrow t$ , que culmina en  $t$  también es de adelanto y su flujo se incrementa en el mismo valor  $\Delta_{C_a}$ . Así pues, el valor de flujo de la red aumenta en  $\Delta_{C_a}$ .

Para el resto de los nodos de  $C_a$  debemos verificar que se preserva la condición de conservación del flujo. Las posibles combinaciones de un nodo cualquiera  $v \in C_a, v \neq s, v \neq t$  son:

1.  $\xrightarrow{+\Delta_{C_a}} v \xrightarrow{+\Delta_{C_a}}$ : en cuyo caso el incremento sobre el flujo entrante es el mismo que para el flujo saliente.
2.  $\xleftarrow{-\Delta_{C_a}} v \xleftarrow{-\Delta_{C_a}}$ : simétrico al anterior, pero con decremento.
3.  $\xleftarrow{-\Delta_{C_a}} v \xrightarrow{+\Delta_{C_a}}$ : sin importar cuáles son los arcos incidentes o adyacentes a  $v$ , ni cuantos éstos son, el flujo se conserva, pues en un arco de salida se aumenta en  $\Delta_{C_a}$  pero en otro se decremente en la misma cantidad.
4.  $\xrightarrow{+\Delta_{C_a}} v \xleftarrow{-\Delta_{C_a}}$ : simétrico al caso anterior pero para arcos de entrada ■

#### 7.10.8 Cálculo de la red residual

Si los caminos de aumento se calculan por inspección de la red residual, entonces requerimos encontrar una manera de calcularla. Una primera consiste en hacer una copia mapeada, modificarle sus arcos a sus capacidades restantes y luego añadirle los arcos residuales. Sobre la red residual se puede actualizar el flujo al eslabón mínimo de un camino de aumento y repetir la búsqueda de caminos de aumento y actualizaciones de flujo hasta maximizarlo. Pero este enfoque conlleva el problema de duplicar el espacio en nodos y el doble de arcos. ¿Puede hacerse en tiempo equiparable y sin necesidad de duplicar?

La respuesta es afirmativa, es decir, podemos representar la red residual sobre la red original. Para eso añadimos un atributo al arco que indique si el arco es o no residual y exportamos interfaces que muestren los pesos de un arco según (7.25). Para saber si el arco es residual o no, guardamos la imagen residual del arco en un puntero y una indicación lógica:

723

*(Miembros de Net\_Arc 723)*

(710b)

```
Net_Arc * img_arc;
bool is_residual;
```

Si *this* es un arco residual, entonces *img\_arc* es un puntero al arco del cual *this* es residual. Análogamente, si *this* es un arco parte de la red, entonces *img\_arc* es la dirección de su arco residual.

El atributo *is\_residual* indica si *this* es o no residual.

Los algoritmos de cálculo de flujo máximo basados en búsquedas de caminos de aumento sobre la red residual se basan en la “capacidad restante” en el arco, la cual llamaremos como  $C_r(e) = cap(e) - f(e)$  para un arco normal. Para que la “ilusión” de “capacidad restante” sea la misma sobre un arco residual  $\bar{e}$ , debe satisfacerse que  $cap(\bar{e}) - f(\bar{e}) = cap(e) - f(e)$ . Así, los valores de un arco residual siempre se colocan en:

$$cap(\bar{e}) = cap(e) \quad (7.26)$$

$$f(\bar{e}) = cap(e) - f(e) \quad (7.27)$$

De este modo, para un arco cualquiera  $e$  y su residual  $\bar{e}$ ,  $cap_r(e) = cap(e) - f(e)$  proporciona la capacidad restante de  $e$ ; mientras que  $cap_r(\bar{e}) = cap(\bar{e}) - f(\bar{e}) = f(e)$  proporciona el flujo circulante por  $e$ .

Puesto que un arco residual se corresponde inicialmente con uno real, en su creación se especifica el arco real:

724a *(Miembros de Net\_Graph 710a) +≡* (711b) ↳ 716 724b ▷

```

void insert_residual_arc(Arc * arc)
{
 Arc * res_arc = Digraph::insert_arc(get_tgt_node(arc), get_src_node(arc));

 res_arc->is_residual = true;
 res_arc->img_arc = arc;
 res_arc->cap = arc->cap;
 res_arc->flow = arc->cap - arc->flow;

 arc->img_arc = res_arc;
}

```

El fin de la red residual es servir de medio para calcular caminos de aumento. Por esta razón no vale la pena preocuparse por las eventuales inconsistencias causadas por la presencia de los arcos residuales, el grado de salida de un nodo o el recorrer los arcos de una red, por ejemplos. El grafo residual se crea para calcular el flujo máximo y se destruye inmediatamente éste se haya calculado.

La eliminación de un arco residual debe realizarse particularmente; es decir, no podemos invocar a `Net_Graph::remove_arc()`, pues éste presumiría que el arco es parte lógica de la red, lo que no es específicamente el caso. Así, lo hacemos mediante otra primitiva particular:

724b *(Miembros de Net\_Graph 710a) +≡* (711b) ↳ 724a 724c ▷

```

void remove_residual_arc(Arc * arc)
{
 Digraph::remove_arc(arc);
}

```

Sabiendo insertar y eliminar arcos residuales, la construcción y destrucción de la red residual dentro de la propia red son directas:

724c *(Miembros de Net\_Graph 710a) +≡* (711b) ↳ 724b

```

void make_residual_net()
{
 size_t n = this->get_num_arcs(); // num arcos residuales a insertar
 for (Arc_Iterator<Net_Graph> it(*this); it.has_current() and n > 0; it.next())
 {
 Arc * arc = it.get_current_arc();
 if (not arc->is_residual)
 {
 insert_residual_arc(arc);
 -n;
 }
 }
}

void unmake_residual_net()
{
 for (Arc_Iterator<Net_Graph> it(*this); it.has_current();)
 {
 Arc * arc = it.get_current_arc();

```

```

if (arc->is_residual)
{
 it.next();
 remove_residual_arc(arc);
}
else
 it.next();
}
}

```

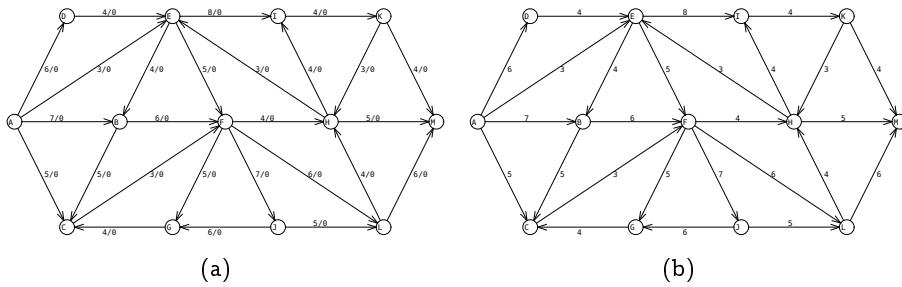


Figura 7.73: Una red ejemplo con flujo en 0 y su residual

### 7.10.9 Cálculo de caminos de aumento

Como ya hemos indicado, la búsqueda de un camino de aumento a través de la red residual se remite a una búsqueda de camino desde el fuente hasta el sumidero. Tal búsqueda puede hacerse por profundidad o amplitud a través de las primitivas de cálculo de caminos estudiadas en § 7.5.7.2 (Pág. 597) y § 7.5.8 (Pág. 600) respectivamente.

A efectos de utilizar exactamente las mismas primitivas debemos hacer que la clase `Node_Arc_Iterator` haga la búsqueda sobre la red residual y no sobre la original. En otras ocasiones será necesario que el iterador sólo muestre arcos residuales y en otras más todos los arcos.

Para tratar con esto definimos una clase filtro que filtre el arco según el valor del flujo restante, sea éste o no un arco residual:

725

*(Funciones para Net\_Graph 725)*≡

```

template <class N> class Res_F
{
 bool operator () (typename N::Node *, typename N::Arc * a) const
 {
 return (a->cap - a->flow) != 0;
 }
};

```

726 ▷

Con este filtro, `find_path_depth_first()` y `find_path_breadth_first()`, invocadas sobre el grafo con la red residual, encontrarán directa y transparentemente caminos de aumento entre el fuente y sumidero.

### 7.10.10 Incremento del flujo por un camino de aumento

Si tenemos un camino de aumento guardado en un objeto de tipo `Path<Net_Graph>` (según § 7.4 (Pág. 578)), entonces, según (7.25), podemos incrementar el flujo de la

red. Para eso definimos el siguiente procedimiento:

```
726 <Funciones para Net_Graph 725>+≡ <725 727a>
 template <class Net>
 typename Net::Flow_Type increase_flow(Net & net, Path<Net> & path)
 {
 typename Net::Flow_Type slack = net.Infinity; // eslabón mínimo
 // calcular el eslabón mínimo del camino de aumento
 for (typename Path<Net>::Iterator it(path); it.has_current_arc(); it.next())
 {
 typename Net::Arc * arc = it.get_current_arc();
 const typename Net::Flow_Type w = arc->cap - arc->flow;
 if (w < slack)
 slack = w;
 }
 // aumentar el flujo de la red por el camino de aumento
 for (typename Path<Net>::Iterator it(path); it.has_current_arc(); it.next())
 {
 typename Net::Arc * arc = it.get_current_arc();
 typename Net::Arc * img = (typename Net::Arc*) arc->img_arc;
 arc->flow += slack;
 img->flow -= slack;
 if (not arc->is_residual)
 {
 net.increase_out_flow(net.get_src_node(arc), slack);
 net.increase_in_flow(net.get_tgt_node(arc), slack);
 }
 else
 {
 net.decrease_in_flow(net.get_src_node(arc), slack);
 net.decrease_out_flow(net.get_tgt_node(arc), slack);
 }
 }
 return slack;
 }
 Uses Path 578a.
```

Cuando se incrementa el flujo de un arco residual se expresa que hay una disminución del flujo del arco real; análogamente, el incremento sobre un arco normal implica un aumento de la capacidad restante del arco residual. De ahí que sea válido aumentar el flujo en arc y disminuirlo en img independientemente de que arc sea o no residual.

### 7.10.11 El algoritmo de Ford-Fulkerson

Hasta el presente hemos descubierto conocimiento acerca de cómo incrementar el flujo de una red. ¿Cómo encontrar el flujo máximo? Fundamentalmente podemos plantear un algoritmo que progresivamente busque caminos de aumento e incremente el flujo hasta que éste devenga máximo. ¿Cuál sería la condición de parada?

Una primera respuesta nos la arroja el corolario 7.1 pag. 719, pero esta vía puede ser computacionalmente costosa, pues requiere manejar conjuntos combinatorialmente. Una vía más eficiente la proporciona la siguiente proposición.

**Proposición 7.10** Sea  $f$  un flujo de una red  $N$ . Entonces  $f$  es máximo si, y sólo si, no existe un camino de aumento en  $N$ .

**Demostración** La necesidad ( $\Rightarrow$ ) se demuestra por contradicción. Si  $f$  es máximo pero existe un camino de aumento, entonces, según (7.25), es posible incrementar el flujo, lo que contradice afirmar que  $f$  sea máximo  $\square$

Para mostrar la suficiencia ( $\Leftarrow$ ) supongamos que no existe un camino de aumento. La ausencia de caminos de aumento indica que el nodo sumidero es inalcanzable desde el fuente, pues el grafo residual estaría inconexo, es decir, todo camino en el grafo reducido que parte desde el fuente se corta al alcanzar arcos con flujo igual a su capacidad. No es posible por tanto aumentar más el flujo, lo cual indica que  $f$  es máximo  $\blacksquare$

Esta proposición nos ofrece el criterio de parada que estamos buscando y es la base del siguiente algoritmo genérico:

727a *<Funciones para Net\_Graph 725>+≡* △726 727b ▷  
 template <class Net, template <class, class> class Find\_Path>  
 typename Net::Flow\_Type  
 augmenting\_path\_maximum\_flow(Net & net, const bool & leave\_residual = false)  
 {  
   typename Net::Node \* source = net.get\_source();  
   typename Net::Node \* sink = net.get\_sink();  
   while (true) // mientras exista un camino de aumento  
   {  
     Path<Net> path(net);  
     if (not Find\_Path<Net, Res\_F<Net> >()(net, source, sink, path))  
       break;  
     increase\_flow <Net> (net, path);  
   }  
   return ret\_val;  
}

Uses Path 578a.

El parámetro `leave_residual` indica al algoritmo que la red residual no debe liberarse y su uso no se maneja en este código. Esto puede ser de utilidad para otros algoritmos, entre ellos, el cálculo del corte mínimo.

Para que este algoritmo opere correctamente, las búsquedas de caminos de aumento sólo deben considerar arcos normales cuya capacidad ya esté alcanzada, o arcos residuales que contengan algún flujo. Para eso instanciamos la búsqueda con el filtro `Res_F<Net>`.

Una instancia del algoritmo genérico anterior para búsquedas de caminos en profundidad sobre la red residual nos conduce al primer y más célebre algoritmo descubierto para flujo máximo, el algoritmo de Ford-Fulkerson:

727b *<Funciones para Net\_Graph 725>+≡* △727a 727c ▷  
 template <class Net> typename Net::Flow\_Type  
 ford\_fulkerson\_maximum\_flow(Net & net, const bool & leave\_residual = false)  
 {  
   return augmenting\_path\_maximum\_flow<Net, Find\_Path\_Depth\_First>  
     (net, leave\_residual);  
}

A efectos de parametrizar el algoritmo de flujo máximo, en otras aplicaciones que lo requieran, exportaremos este algoritmo bajo el nombre de una clase:

727c *<Funciones para Net\_Graph 725>+≡* △727b 731 ▷

```

template <class Net> class Ford_Fulkerson_Maximum_Flow
{
 typename Net::Flow_Type
operator () (Net & net, const bool & leave_residual = false) const
{
 return ford_fulkerson_maximum_flow(net, leave_residual);
}
};

```

En lo que sigue, todos los algoritmos de flujo máximo que desarrollaremos tendrán su clase invocante.

El algoritmo de Ford-Fulkerson se basa en encontrar caminos de aumento por búsqueda en profundidad e incrementar progresivamente el flujo en el eslabón mínimo. Los caminos se localizan en la red residual que se construye como extensión de la red original y las actualizaciones del flujo también se efectúan sobre la red.

Las figuras 7.74 y 7.75 (Págs. 728-729) ilustran todas las iteraciones del algoritmo de Ford-Fulkerson sobre la red mostrada en la figura 7.73.

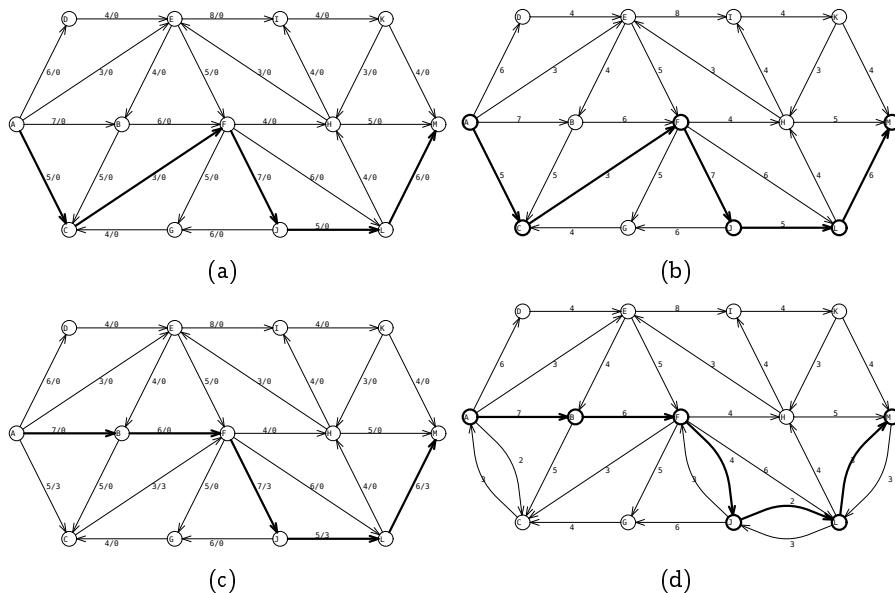


Figura 7.74: Evolución del algoritmo de Ford-Fulkerson sobre red de figura 7.73. Lado izquierdo muestra flujo de red y resalta camino de aumento; lado derecho red residual y con camino de aumento; arcos con valor 0 no se muestran

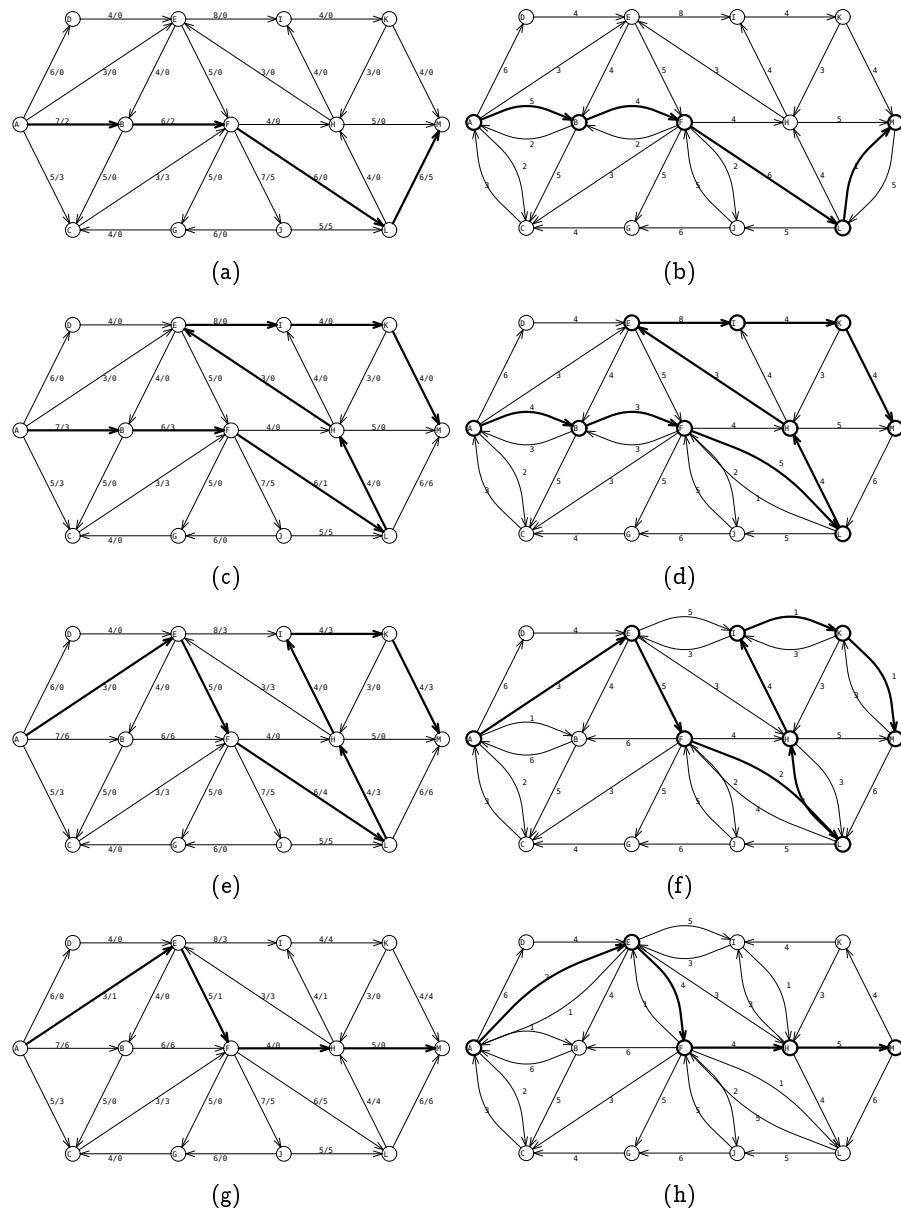


Figura 7.75: Continuación del algoritmo de Ford-Fulkerson sobre la red de la figura 7.73

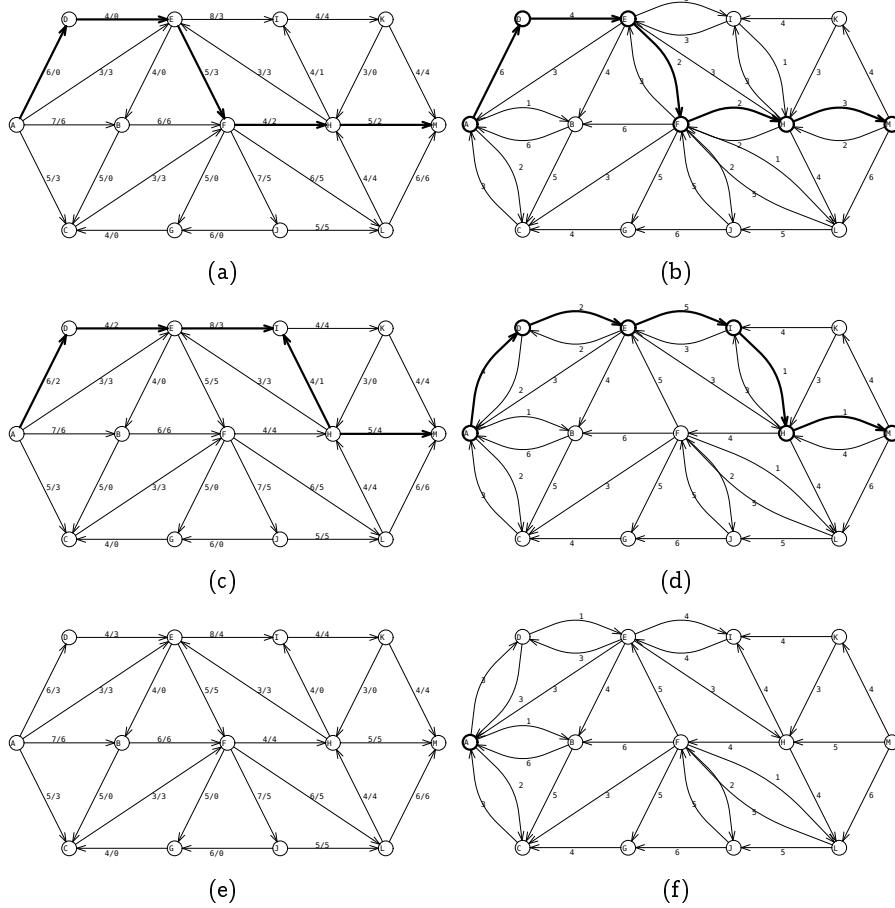


Figura 7.76: Culminación y estado final del algoritmo de Ford-Fulkerson con la red de la figura 7.73

#### 7.10.11.1 Análisis del algoritmo de Ford-Fulkerson

Puesto que el grafo residual se construye sobre la propia red mediante el método `make_residual_net()` (§ 7.10.10 (Pág. 725)), el consumo de espacio es  $\mathcal{O}(E)$ , cantidad que absorbe el espacio que pueda ocupar el camino de aumento `path`.

El coste en duración de la construcción de la red residual es  $\mathcal{O}(E)$ , pues se requieren revisar todos los arcos de la red.

La duración máxima o promedio de una búsqueda de camino de aumento por explotación en profundidad está acotada por  $\mathcal{O}(E)$  para el peor caso, lo cual es vigente a pesar de tener el doble de arcos. Si  $N$  fuese la cantidad de veces que repite el `while`, entonces la duración del algoritmo es  $\mathcal{O}(E) + N \times \mathcal{O}(E) = N \times \mathcal{O}(E)$ . Por tanto, la duración depende de que “tan bueno” sea el camino de aumento encontrado por `find_path_depth_first()`.

Cuanto mayor sea el mínimo eslabón de un camino de aumento, más se acercará el incremento del flujo al valor máximo. La bondad de un camino de aumento estriba entonces en que su eslabón mínimo sea alto. En este sentido, `find_path_depth_first()` no nos ofrece ningún criterio; la rutina encuentra el camino de aumento según las circunstancias topológicas, las cuales, a nuestros efectos, pueden considerarse aleatorias. Por tanto, asumiendo capacidades enteras podrían requerirse  $\mathcal{O}(f^*)$  repeticiones para llegar al máximo. En el dominio entero, esto sucede si `find_path_depth_first()` siempre

nos encuentra caminos con eslabón mínimo unitario. Para aprehender mejor el asunto, consideremos la red de la figura 7.77. Si la suerte decide que el primer camino de aumento descubierto por `find_path_depth_first()` sea  $s \rightarrow v \rightarrow w \rightarrow t$ , entonces `ford_fulkerson_maximum_flow()` puede efectuar  $N$  incrementos de flujo si, a partir del grafo residual de la figura 7.77-c, `find_path_depth_first()` encuentra progresivamente como camino de aumento a  $s \rightarrow w \rightarrow v \rightarrow t$ , luego  $s \rightarrow v \rightarrow w \rightarrow t$ , siempre con eslabón mínimo de 1, y así sucesivamente hasta finalmente maximizar el flujo.

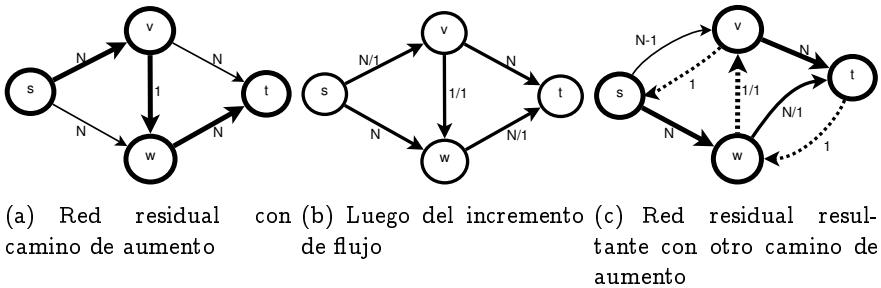


Figura 7.77: Ejemplo de red cuyo peor caso con una búsqueda en profundidad requiere  $\mathcal{O}(N)$  incrementos de flujo para el algoritmo de Ford-Fulkerson.

Una de las críticas al algoritmo de Ford-Fulkerson es que su terminación no estaría garantizada si las capacidades se representan con irracionales. La crítica es de índole teórica, pues si los irracionales se representan en punto flotante, entonces los sucesivos aumentos de flujo terminan alcanzando el valor de capacidad de un arco. Si se emplean fracciones como capacidades, entonces éstas pueden transformarse a enteros según su mínimo común múltiplo.

Sin embargo, aun si supeditamos las capacidades al dominio entero y una exploración en profundidad determinista, el algoritmo de Ford-Fulkerson puede exhibir, en el peor caso, una eficiencia de  $\mathcal{O}(f^* \times E)$ , lo cual, como se aprecia en el ejemplo de la figura 7.77, podría ser bastante severo.

### 7.10.12 El algoritmo de Edmonds-Karp

Un refinamiento al algoritmo anterior estriba en substituir la búsqueda en profundidad de un camino de aumento por una en amplitud. Si bien la búsqueda en amplitud tiende a ser un poco más onerosa en memoria que la de profundidad debido al encolamiento de caminos parciales, este sutil cambio asegura un rendimiento considerablemente más eficiente que el algoritmo de Ford-Fulkerson:  $\mathcal{O}(V E^2)$ , para el peor caso.

Habida cuenta de toda la maquinaria desplegada, el algoritmo de Edmonds-Karp es estructuralmente idéntico al de Ford-Fulkerson:

```
<Funciones para Net_Graph 725>+≡ ▷727c 736▷
 template <class Net> typename Net::Flow_Type
 edmonds_karp_maximum_flow(Net & net, const bool & leave_residual = false)
 {
 return augmenting_path_maximum_flow<Net, Find_Path_Breadth_First>
 (net, leave_residual);
 }
```

El sutilísimo cambio sobre una palabra -depth por breadth- es, como veremos prontamente, fundamental para acotar el desempeño del algoritmo.

Las figuras 7.78 y 7.79 (Págs. 732-733) ilustran todas las iteraciones del algoritmo de Edmonds-Karp sobre la red mostrada en la figura 7.73. A pesar de la baja escala y del azar, el algoritmo calcula el flujo máximo en una iteración menos que con el algoritmo de Ford-Fulkerson (ver figuras 7.74 y 7.75, págs. 728 y 729).

Para nuestras circunstancias hay una ganancia en desempeño fácilmente aprensible. A la postre, con este algoritmo, la rutina `increase_flow()` opera más rápido que con el de Ford-Fulkerson, pues la pasada para calcular el eslabón mínimo se hace sobre el camino más corto.

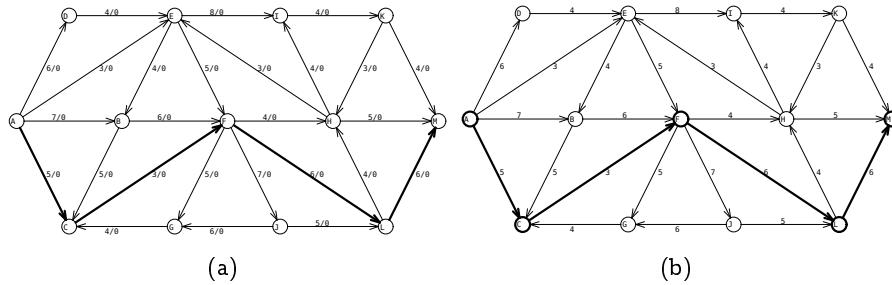


Figura 7.78: Evolución del algoritmo de Edmonds-Karp sobre la red de la figura 7.73

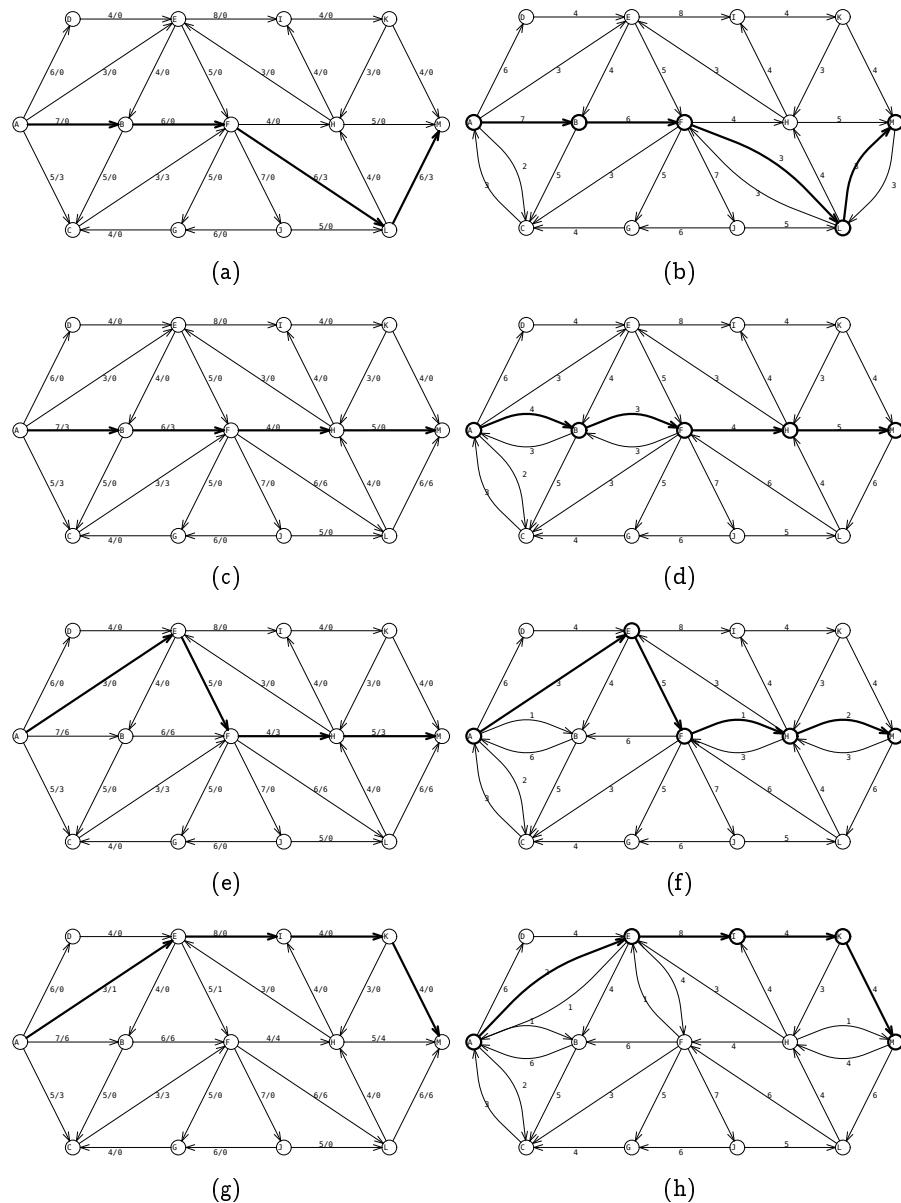


Figura 7.79: Continuación del algoritmo de Edmonds-Karp sobre la red de la figura 7.73.

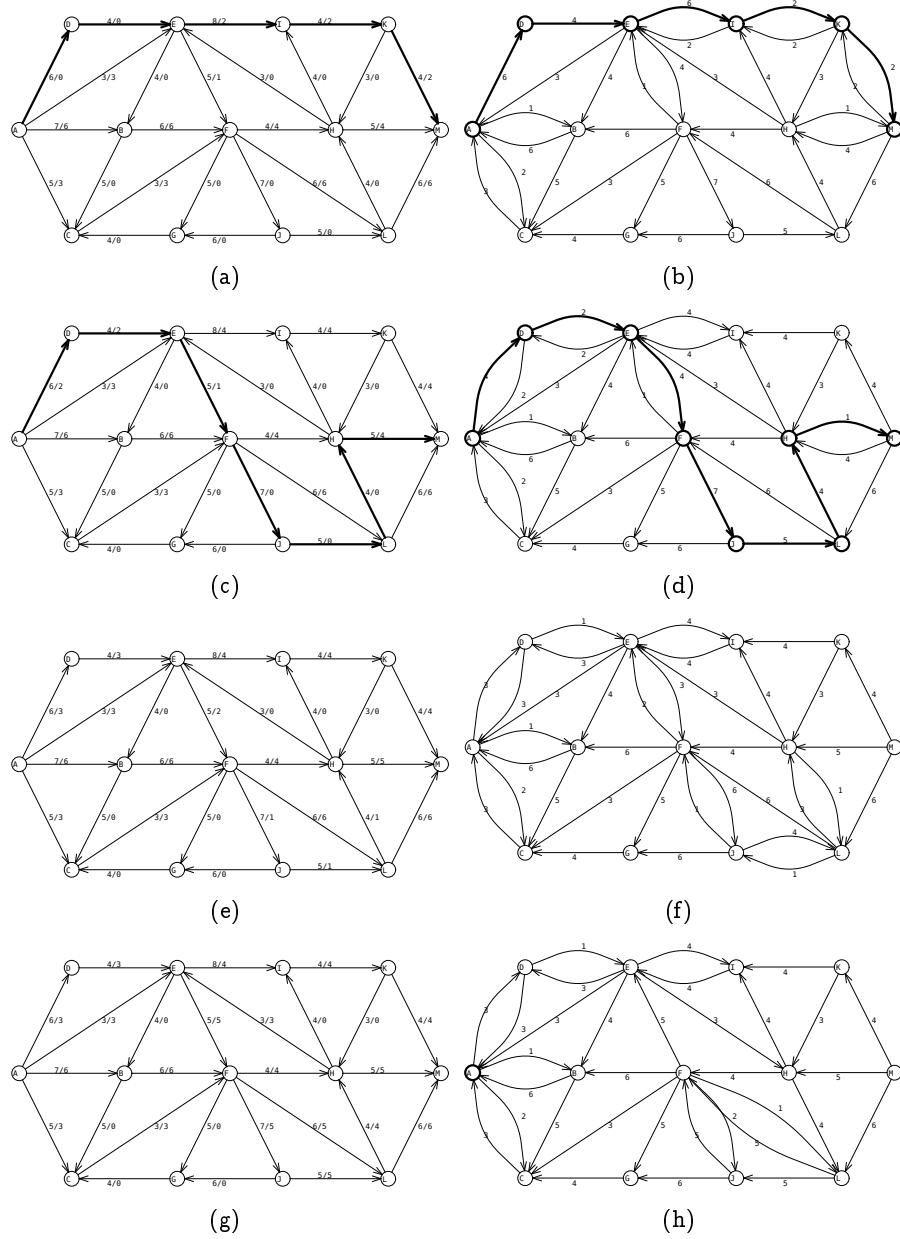


Figura 7.80: Estado final del algoritmo de Edmonds-Karp con la red de la figura 7.73.

#### 7.10.12.1 Análisis del algoritmo de Edmonds-Karp

Una búsqueda en amplitud requiere  $\mathcal{O}(E)$  para el peor caso (ver § 7.5.4 (Pág. 591)). De nuevo, el desempeño del algoritmo estará acotado por  $N \times \mathcal{O}(E)$ , donde  $N$  sería la máxima cantidad de incrementos de flujo.

Como la búsqueda es en amplitud, los caminos de aumento se descubren desde los más “cortos”, en cantidad de arcos, hasta los más largos. Definamos  $d(u, v)$  como la distancia en arcos de un camino entre dos nodos  $u$  y  $v$ .

**Lema 7.6** Sea  $C_{a_i}$  un camino de aumento calculado por `find_path_breadth_first()` durante la  $i$ -ésima iteración del algoritmo de Edmonds-Karp. Entonces, luego del  $i$ -ésimo incremento del flujo,  $d_{i+1}(s, t) \geq d_i(s, t)$ .

**Demostración** Supongamos que  $C_i = s \rightsquigarrow v$  el camino más corto en arcos desde  $s$  hasta  $v$ . Ahora supongamos la existencia de un camino de aumento  $C_{a_i}$ , calculado por `find_path_breadth_first()`, que contenga un camino más corto; es decir, que  $d_{i+1}(C_{a_i}) < d(C_i)$ . Debemos considerar dos situaciones generales:

1.  $v \notin C_{a_i} \Rightarrow C_i$  sigue siendo mínimo en arcos.
2.  $\exists w \in C_i | w \in C_{a_i} \Rightarrow$  podemos considerar dos “posibilidades”:
  - (a) Si  $C_{a_i} = s \rightsquigarrow v \rightsquigarrow w \rightsquigarrow t$ , entonces  $C_i$  sigue siendo mínimo y se satisfaría  $d_{i+1}(s, t) \geq d_i(s, t)$ . Pero,
  - (b) Si  $C_{a_i} = s \rightsquigarrow w \rightsquigarrow v$  sería una contradicción con el hecho de que  $C_i$  es el mínimo camino en arcos desde  $s$  hacia  $v$ . Esta única “posibilidad” es una contradicción. El lema es pues cierto  $\square$

**Definición 7.17 (Arco crítico de un camino de aumento)** Sea  $C_a = s \rightsquigarrow v_1 \rightsquigarrow \dots u \rightarrow v \rightsquigarrow v_m \rightsquigarrow t$  un camino de aumento cualquiera sobre una red residual. Se dice que un arco  $(u, v) \in C_a$  es crítico si su capacidad en el grafo residual es igual al mínimo eslabón de  $C_a$  y  $(u, v) \in E$ .

La propiedad fundamental de un arco crítico es que éste se elimina de la red residual al incrementarse el flujo si se trata de un arco de adelanto, o al decrementarse a 0 si es uno de retroceso. A partir de este hecho podemos establecer que si el camino de aumento  $C_{a_i}$  contiene un arco crítico  $(u, v)$ , entonces, para una iteración futura, un camino de aumento  $C_{a_{i+k}}$  que involucre a los nodos  $u$  y  $v$  es seguro que  $d(C_{a_{i+k}}) \geq d(C_{a_i}) + 2$ .

Para aprehender el hecho anterior, observemos la figura 7.81. La línea continua ilustra

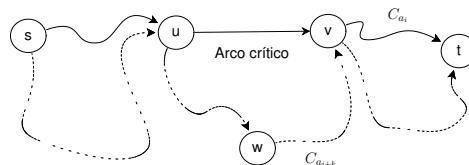


Figura 7.81: Argumento pictórico del aumento de arcos por un camino que involucre a un par  $(u, v)$

un hipotético camino de aumento con un arco crítico  $u, v$  a la  $i$ -ésima iteración del algoritmo de Edmonds-Karp, mientras que la línea punteada un camino de aumento en una iteración futura  $i + k$ . Al incrementarse el flujo el arco crítico desaparece. Consecuentemente, cualquier camino de aumento futuro que involucre a  $u$  y  $v$  tiene que contener cuando menos un nodo intermedio  $w$  y dos arcos de más; esto es seguro porque los caminos de aumento se buscan en amplitud, lo cual, como ya aclaramos, retorna caminos desde los más cortos hacia los más largos.

La máxima longitud posible de un camino de aumento es de  $V$  nodos. Consecuentemente, un arco puede ser crítico en a lo sumo  $V/2$  caminos de aumento. Por tanto, la máxima cantidad posible de caminos de aumento es  $E \times \frac{V}{2}$ .

De lo anterior concluimos que el algoritmo de Edmonds-Karp es  $E \times \frac{V}{2} \times \mathcal{O}(E) = \mathcal{O}(VE^2)$ . Si la red es espaciada, lo que no es infrecuente, entonces  $E \approx c V$  y el algoritmo de Edmonds-Karp sería  $\mathcal{O}(V^3)$

### 7.10.13 Algoritmos de empuje y preflujo

Un flujo es dinámico en el sentido de que el fluido siempre está corriendo por las tuberías; si no, entonces no podríamos hablar de flujo.

Imaginemos una red de tuberías, de agua, por ejemplo. Supongamos que deseamos maximizar la cantidad de agua que llega al sumidero.

Un enfoque que sugiere la intuición es inundar a la máxima capacidad posible las tuberías del nodo fuente. Para eso inyectamos un flujo cuya presión sobrepase la capacidad de los arcos salientes del fuente. Luego, desde los nodos adyacentes al fuente, se van inundando los arcos del segundo nivel. El proceso continua progresivamente hasta obtener flujo en el nodo sumidero. Debe sernos claro que el máximo valor de flujo tiene que ser menor o igual a este flujo inicial. Si todas las llaves están abiertas, entonces el flujo obtenido a través de este procedimiento es máximo.

Para que este proceso sea realista y el nodo fuente reciba exactamente el flujo máximo, los nodos destino de las tuberías queemanan del sumidero tendrían que disponer de una especie de aliviadero de modo que el flujo saliente del fuente pueda mantenerse constante mientras se van ajustando las llaves del resto de las tuberías.

Este patrón intuitivo es el fundamento de una serie de algoritmos llamados de “empuje de preflujo”.

**Definición 7.18 (Preflujo)** Sea una red  $N = \langle V, E, s, t, C \rangle$ . Un “preflujo” es un conjunto  $E' \subset E$  de arcos con valor de flujo positivo tal que  $\forall e = (u, v) \in E'$ :

1.  $f(e) \leq \text{cap}(e)$ , y
2.  $u \neq s, v \neq s \implies \text{OUT}(u) \leq \text{IN}(u), \text{OUT}(v) \leq \text{IN}(v)$ . En otras palabras, el flujo de salida debe ser menor o igual que el de entrada. Importante observar que bajo esta definición no se cumple -temporalmente- la condición de conservación del flujo; de ahí pues la necesidad abstracta y físicamente concreta de que cada nodo posea el aliviadero. Por la misma razón, a esta familia de algoritmos se le llama de “preflujo”, pues durante la ejecución del procedimiento intuitivo no hay flujo sobre algunos nodos, incluido el sumidero.

Un nodo distinto al fuente o al sumidero es llamado “interno”. Un nodo interno cuyo flujo de entrada es mayor que el de salida es denominado “activo”.

En función del TAD Net\_Graph, la determinación de un nodo activo es simple:

736

*(Funciones para Net\_Graph 725) +≡*

◀731 ▷737▶

```
template <class Net> static
bool is_node_active(typename Net::Node * p)
{
 return p->in_flow > p->out_flow;
}
```

Para un nodo activo  $u$ , el valor  $\text{IN}(u) - \text{OUT}(u)$  es llamado “exceso” y denotado como  $\text{excess}(u)$ .

Siguiendo con el procedimiento intuitivo, la idea fundamental de un algoritmo de preflujo es comenzar a empujar desde el fuente la mayor cantidad posible de flujo hasta que no se pueda empujar más sobre el sumidero. En este momento el flujo debe ser máximo, pero muy probablemente habrá nodos activos cuyo equilibrio hay que restaurar.

¿Cómo elegir nodos y arcos por donde empujar preflujo? Al inicio del algoritmo, la respuesta es directa: el fuente y todos sus arcos. Luego los nodos activos son elegibles pero, ¿por cuáles entre sus arcos empujar el preflujo? Esta pregunta avizora el asunto más complejo de esta clase de algoritmo, pues el cálculo del flujo máximo y la terminación depende de una escogencia adecuada del próximo nodo a procesar y por cuáles entre sus arcos de salida empujar preflujo.

#### 7.10.13.1 Altura de un nodo

Para la familia de algoritmos que prontamente definiremos, la escogencia del próximo nodo subyace sobre la idea de altura.

**Definición 7.19 (Función de altura de un nodo)** Sea una red  $N = < V, E, s, t, C >$  con red residual  $\bar{N} = < V, E', s, t, C' >$ . Una función de altura  $h : V \rightarrow \mathcal{N}$  es una función entre los nodos de la red tal que:

1.  $h(t) = 0$ .
2.  $\forall e = (u, v) \in E' \implies h(u) \leq h(v) + 1$ .

Recordemos que una función puede interpretarse como un conjunto de pares ordenados. En nuestro caso particular del tipo  $A_h = \{(u_1, h(u_1)), (u_2, h(u_2)), \dots, (u_n, h(u_n))\}$ .

Un arco  $e = (u, v)$  se califica de “elegible” si  $h(u) = h(v) + 1$ .

Para mantener el valor de la función de altura de un nodo usaremos su contador (véase § 7.3.5.2 (Pág. 558)):

737 *Funciones para Net\_Graph 725* +≡ △736 742 ▷  
 template <class Net> static  
 long & node\_height(typename Net::Node \* p) { return NODE\_COUNTER(p); }

La pertenencia o no de un nodo al conjunto  $A_h$  determina la función.

La idea de la función de altura es mantener una relación de proximidad entre el preflujo y los nodos fuente y sumidero. Cuando la altura de un nodo activo es menor que la del fuente, entonces es probable empujar preflujo hacia el sumidero. Simétricamente, si la altura es mayor, entonces probablemente haya que devolver preflujo hacia el fuente.

#### 7.10.13.2 Algoritmo genérico de preflujo

En virtud de su carácter intuitivo, los algoritmos basados en empuje de preflujo son más simples que los basados en búsquedas de caminos de aumento. En añadidura, prácticamente casi todo algoritmo de preflujo subyace sobre el siguiente algoritmo genérico.

**Algoritmo 7.6 (Algoritmo genérico de empuje de preflujo)** La entrada es una red  $N = < V, E, s, t, C >$ . La salida es la misma red con valores de flujo ajustado al máximo valor de flujo.

Además de la red residual  $\bar{N} = < V, E', s, t, C' >$ , el algoritmo usa una estructura de datos interna denominada  $A_h$ , cuyo fin es implantar la función de altura. Cada nodo  $u$  asocia su valor  $h(u)$  tal como se explicó en la subsección anterior.

1. Calcular la red residual  $\bar{N}$

2. Asignar a cada nodo  $u$  su valor  $h(u)$  acorde a la definición 7.19 (Pág. 737)
3.  $A_h = \emptyset$  (el conjunto de nodos activos)
4.  $\forall e = (s, u) \in s$  (*Preflujo inicial de arcos del nodo fuente 745a*):
  - (a)  $f(e) = \text{cap}(e)$  [Saturación]
  - (b) Si  $u \neq t \implies A_h = A_h \cup \{u\}$  [añadir  $u$  al conjunto de nodos activos]
5. Mientras  $A_h \neq \emptyset$  [Equilibrar nodos activos]
  - (a) Seleccione un nodo activo  $u \in A_h$ ,  $A_h = A_h - \{u\}$
  - (b)  $\forall e \in u, e \in E'$  [Empuje de preflujo de  $u$ ]:
    - i. Si  $e = (u, v)$  es un “arco elegible”, es decir, si  $h(u) = h(v) + 1 \implies \langle \text{Empujar preflujo por arco actual 745b} \rangle$ 
      - A.  $f_p = \min(\text{excess}(u), \text{cap}(e) - f(e))$  [Flujo a empujar sobre el arco]
      - B.  $f(e) = f_p$
      - C.  $\text{excess}(u) = \text{excess}(u) - f_p$
      - D. Si  $u \neq s$  y  $u \neq t \implies A_h = A_h \cup \{v\}$
    - (c) Si  $\text{excess}(u) > 0 \implies h(u) = h(u) + 1, A_h = A_h \cup \{u\}$  ( $u$  es aún un nodo activo con un incremento en la función de altura)

El algoritmo empuja todo el posible preflujo por los arcos del fuente. Paulatinamente, según el orden en que se presenten los nodos activos, el exceso se va propagando hasta alcanzar el sumidero. Cuando éste deviene saturado, el exceso retorna hacia el fuente drenándose, si es posible, en aquellos nodos que tengan capacidad. Finalmente, cuando el exceso alcanza el fuente, este se le resta a sus arcos hasta que el flujo devenga estable. En este estadio, el flujo es máximo y ya no existen nodos activos.

En la instrucción 5(b)i se determina si se empuja o no preflujo por cada arco saliente del nodo actualmente seleccionado. Si no se logra empujar todo el exceso, entonces el nodo es incluido de nuevo en  $A_h$ , pero con un valor  $h(u)$  incrementado en una unidad; esto es lo que permite que eventualmente otra iteración sobre el mismo nodo vea como elegible a un arco que en una iteración pasada no lo fue, y en lugar de avanzar el preflujo hacia el sumidero, se retroceda hacia el fuente, pudiendo ocurrir inclusive que se disminuyan los flujos emanantes del fuente.

Una manera de aproximarse a la bondad del algoritmo consiste en examinar lo que es un mal caso para un algoritmo cualquiera basado en la búsqueda de un camino de aumento. Para eso consideremos una red general con la forma de la figura 7.82. En esta clase de red, el nodo  $x$  tiene siete bifurcaciones que resultan con destino  $y$ . En este caso se requiere encontrar siete caminos de aumento para poder llenar los nodos  $x$  e  $y$ ; mientras que un algoritmo basado en empuje de preflujo probablemente los llenará apenas se empuje el exceso de  $x$ .

#### 7.10.13.3 Correctitud del algoritmo genérico

Antes de instrumentar el algoritmo genérico y algunos particulares debemos probar su correctitud, es decir, que calcula el flujo máximo. En lo que sigue plantearemos una serie

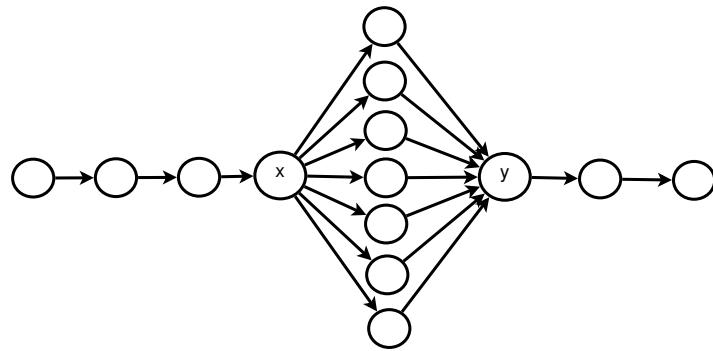


Figura 7.82: Esquema general de Un mal caso de maximización de flujo para un algoritmo basado en caminos de aumento

de proposiciones destinadas a probar la proposición de correctitud. Basaremos nuestro discurso en el magistral planteamiento de Sedgewick [156], el cual a su vez se basa en el de Goldberg y Tarjan [62].

El primer paso hacia la prueba de correctitud es demostrar que la función de altura es válida.

**Proposición 7.11 Validez de la función de altura Sedgewick 2002 [156]** Sea una red  $N = < V, E, s, t, C >$  con red residual  $\bar{N} = < V, E', s, t, C' >$ . Sea  $h : V \rightarrow \mathcal{N}$  una función de altura. Entonces el algoritmo genérico de empuje de preflujo 7.6 conserva la validez de la función de altura según la definición 7.19 (Pág. 737).

**Demostración** Para afirmar que  $h$  es válida hay que demostrar que a lo largo de la ejecución del algoritmo  $h(u) \leq h(v) + 1$  para todo arco de la red residual.

El paso 2 asigna valores “correctos” según la definición. Además,  $A_h$  es inicialmente vacío, por lo que la demostración se remite a comprobar que los nodos que se introduzcan a  $A_h$  satisfagan la definición.

Imaginemos un nodo activo  $u$ . Si no existe ningún arco elegible, entonces  $h(u) = h(u) + 1$  y  $u$  regresa al conjunto de los nodos activos con  $h(u)$  incrementado en uno por la instrucción 5c. Antes de esta instrucción había un conjunto válido  $A_h = \{(v_1, h(v_1)), (v_2, h(v_2)), \dots, (u, h(u)), \dots, (v_n, h(v_n))\}$ ; luego, el conjunto tiene forma  $A_h = \{(v_1, h(v_1)), (v_2, h(v_2)), \dots, (u, h(u) + 1), \dots, (v_n, h(v_n))\}$ , el cual sigue siendo válido según la definición.

Si  $u$  tiene uno o más arcos elegibles, entonces algunos se saturarán a plena capacidad, desaparecerán de la red residual y aparecerán otros inversos. Consideremos el último de los arcos elegibles  $u \rightarrow v$ , el cual determinará que  $u$  permanezca o no dentro del conjunto de nodos activos. Debemos contemplar dos posibilidades:

1. Que  $u \rightarrow v$  se sature a plena capacidad: en este caso,  $u \rightarrow v$  también desaparece de la red residual, se crea un arco inverso  $v \rightarrow u$  y  $u$  deja de ser un nodo activo, por lo que la función  $h$  es correcta.
2. Que  $u \rightarrow v$  no se sature a plena capacidad: en este caso,  $u \rightarrow v$  se mantiene en la red residual, aparece un nuevo arco  $v \rightarrow u$  pero  $h(u)$  se incrementa. La situación final se puede imaginar como en la figura 7.83. Cada arco  $w \rightarrow u$  satisface  $h(w) \leq h(u) + 1$ ,

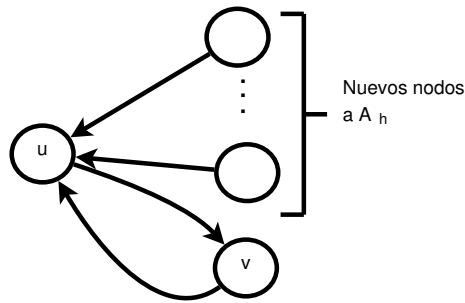


Figura 7.83: Esquema de demostración de la proposición 7.11 para el caso en que  $u \rightarrow v$  no se sature

pues  $u$  fue incrementado.

Para los arcos  $u \rightarrow v$  y  $v \rightarrow u$  tenemos que  $h(u) = h(v)$ , lo que satisface la validez de la función de altura ■

La acotación del valor de  $h(u)$  nos acota la cantidad de arcos que contiene el camino mínimo  $u \rightsquigarrow t$ , tal como establece la proposición siguiente.

**Proposición 7.12 (Sedgewick 2002 [156])** Sea una red  $\mathcal{N} = \langle V, E, s, t, C \rangle$  con red residual  $\bar{\mathcal{N}} = \langle V, E', s, t, C' \rangle$ . Sea  $h : V \rightarrow \mathcal{N}$  una función de altura. Entonces,  $\forall v \in V$   $h(v)$  es menor o igual que la longitud del camino más corto en arcos desde  $v$  hacia  $t$  en la red residual.

**Demostración** Sea  $d$  la longitud del camino más corto en arcos desde un nodo  $v$  hacia  $t$ . Sea  $C_{v,t} = v = v_1 \rightarrow v_2 \cdots \rightarrow v_d = t$  el camino más corto en la red residual. Entonces, según la definición 7.19 (Pág. 737):

$$\begin{aligned} h(v) &= h(v_1) \leq h(v_2) + 1 \\ &\leq h(v_3) + 1 \\ &\vdots \\ &\leq h(v_d) + d = h(t) + d = d \quad ■ \end{aligned}$$

El algoritmo siempre mantiene conectado al exceso, es decir, no hay nodos activos que estén inconexos en la red residual. Inicialmente, el exceso de flujo emana del fuente  $s$  y se dirige hacia el sumidero, el cual tiene  $h(t) = 0$  y nunca se añade a  $A_h$ . Los excesos son propagados hacia el sumidero en la instrucción 5(b)iD. Si no se alcanza el sumidero o aún quedan nodos activos, entonces el exceso es empujado hacia el fuente para decremetarlo. Este hecho, hasta el presente intuitivo, puede confirmarse objetivamente mediante la siguiente proposición.

**Proposición 7.13 (Sedgewick 2002 [156])** Sea una red  $\mathcal{N} = \langle V, E, s, t, C \rangle$  con red residual  $\bar{\mathcal{N}} = \langle V, E', s, t, C' \rangle$ . Sea  $A_h : V \rightarrow \mathcal{N}$  una función de altura. Entonces, durante la ejecución del lazo del algoritmo 7.6 (Pág. 737), para cada nodo activo:

1. Existe un camino en la red residual desde el nodo activo hasta el fuente.
2. No existe camino desde el fuente hacia el sumidero.

**Demostración (por inducción sobre los nodos activos)** Antes de entrar al lazo, el paso 4 introduce en  $A_h$  todos los nodos  $\{u_1, u_2, \dots, u_n\}$  adyacentes a  $s$ . Estos arcos son saturados a su plena capacidad, lo cual implica la aparición de arcos  $(u_1, s), (u_2, s), \dots, (u_n, s)$  en la red residual y la desaparición de los inversos  $(s, u_1), (s, u_2), \dots, (s, u_n)$ . Los nodos activos  $u_1, u_2, \dots, u_n$  tienen caminos hasta el sumidero y el fuente  $y$ , por la desaparición de los inversos,  $s$  queda desconectado. Por tanto, al entrar al lazo la proposición es cierta para los nodos activos iniciales adyacentes al fuente.

Ahora asumamos que la proposición es cierta para todo nodo dentro del conjunto  $A_h$  de nodos activos.

La única manera de que se añada un nuevo nodo  $v$  a  $A_h$ , es decir, de que aparezca un nuevo nodo activo, es que desde un actual activo  $u$  exista un arco elegible  $u \rightarrow v$  (instrucción 5(b)i). Hay dos casos a considerar:

1.  $v$  nunca ha sido activo: en este caso  $h(u) = h(v) + 1$ . Antes de empujar por  $u \rightarrow v$ , no hay camino  $v \rightsquigarrow s$ , pues los arcos salientes de  $v$  no contienen flujo.

Cuando se empuja el exceso por el arco  $u \rightarrow v$  aparece un arco residual saliente  $v \rightarrow u$ . Por la hipótesis inductiva existe un camino  $u \rightsquigarrow s$ , por tanto, el arco residual  $v \rightarrow u$  garantiza que también existe un camino desde  $v \rightsquigarrow s$ .

Del mismo modo, por la hipótesis inductiva no hay camino  $s \rightsquigarrow t$  y la añadidura del arco  $v \rightarrow u$  no altera esta afirmación.

2.  $v$  ya fue activo en un ciclo anterior: este caso ocurre cuando  $v$  se saca de  $A_h$  y no hay ningún arco elegible. Se incrementa el valor de  $h(v)$  y  $v$  regresa a  $A_h$  sin alterar la topología de la red residual ■

La proposición es cotejable para dos casos particulares de ejecución mostrados en la familia de figuras 7.85 (pag. 749), 7.87 (pag. 753) y 7.89 (pag. 758).

Ahora tenemos todas las herramientas necesarias para verificar la correctitud del algoritmo 7.6 (Pág. 737).

**Proposición 7.14 (Sedgewick 2002 [156])** El algoritmo 7.6 (Pág. 737) calcula el flujo máximo de una red.

**Demostración** El primer asunto a comprobar es que el algoritmo culmina, o sea, verificar que una vez que se entra al lazo (instrucción 5) éste termina. La terminación ocurre cuando  $A_h$  deviene vacío. Necesitamos entonces mirar la manera en que  $A_h$  crece y decrece y asegurarnos de que el lazo progrese; es decir, de que no caiga en un ciclo infinito.

$A_h$  crece por dos razones. La primera es por empuje de preflujo por un arco  $u \rightarrow v$  (instrucción 5(b)iD), en cuyo caso  $v$  deviene activo. La segunda es porque no se puede empujar todo el flujo de un nodo activo (instrucción 5c); en este caso,  $u$  regresa a  $A_h$  con un valor de  $h(u)$  incrementado.

$A_h$  sólo decrece en la instrucción 5a. Por tanto, la terminación del algoritmo depende de que la cantidad de añadiduras a  $A_h$  esté acotada. En este sentido nos será muy útil el siguiente lema.

**Lema 7.7** Durante la ejecución del algoritmo genérico 7.6 (Pág. 737),  $\forall u \in V \implies h(u) < 2V$ .

**Demostración** Sabemos, según la proposición 7.12 (Pág. 740), que  $h(u)$  es menor o igual que la longitud del camino más corto desde  $u$  hacia  $t$ . Como tratamos con una red residual, podemos tener hasta el doble de arcos y puede haber caminos que pasen por el fuente. Por tanto, el camino más largo desde un nodo  $u$  hacia  $t$  no puede exceder de  $2V$  arcos, pues sino contradiría la proposición 7.12 (Pág. 740)  $\square$

Con el conocimiento de este lema podemos afirmar que la cantidad de veces que la instrucción 5c añade nodos a  $A_h$  está acotada. Del mismo modo, para que la instrucción 5(b)iD añada el nodo activo  $v$ , el arco  $u \rightsquigarrow v$  tiene que ser elegible, es decir,  $h(u) = h(v) + 1$ . Independientemente de las combinaciones que se sucedan, el valor de  $h(u)$  está acotado, por lo que la máxima cantidad de veces que se añade un nodo activo  $v$  por adyacencia desde un nodo  $u$  también lo está.

Puesto que el lazo extrae nodos activos de  $A_h$  y la cantidad de añadiduras a  $A_h$  está acotada, concluimos que  $A_h$  deviene vacío, por lo que el lazo termina y con él el algoritmo.

Una vez que  $A_h$  deviene vacío la red contiene un flujo factible. En este estado sabemos, por la proposición 7.13 (Pág. 740), que no existe ningún camino desde el fuente hacia el sumidero, por lo que no existe ningún camino de aumento. Consecuentemente, según la proposición § 7.10 (Pág. 726), el flujo es máximo ■

El análisis de desempeño del algoritmo depende de la manera en que implante el conjunto de nodos activos  $A_h$ . Pero antes de abordar esta implantación debemos escoger una forma de asignar los valores iniciales de altura de cada nodo.

#### 7.10.13.4 Valores iniciales de la función de altura

Hay varias maneras de asignar valores iniciales a  $h(u)$  que satisfagan la definición y ejecución del algoritmo 7.6 (Pág. 737). Pero la inicialización puede ser esencial para el desempeño. Para indicar esto consideremos la siguiente inicialización:

$$h(u) = \begin{cases} V & u = s \\ 0 & u \neq s \end{cases}$$

Aunque esta configuración es válida, ésta tiende a causar extracciones vanas, es decir, extracciones de nodos sin arcos elegibles, pues es sólo luego de  $V$  iteraciones que uno de los nodos adyacentes al fuente alcanza el valor  $V + 1$  y deviene elegible.

¿Cuál es la mejor configuración?, es una pregunta que da pie a diversas investigaciones que consideren, entre otras cosas, la topología del grafo, el tipo de red que modeliza, sus puntos de corte, etcétera, que no abordaremos en este texto. Ahora bien, el algoritmo de Edmonds-Karp nos proporcionó la manera genérica más corta de ir desde el fuente hacia el sumidero: el recorrido en amplitud.

En lo que sigue consideramos como valor inicial de  $h(u)$  la longitud más corta desde  $u$  hasta el sumidero, la cual puede calcularse en  $\mathcal{O}(E)$  mediante un recorrido en amplitud iniciado desde el sumidero y directamente factible por la rutina `breadth_first_traversal()` estudiada en § 7.5.4 (Pág. 589).

A efectos de la simplicidad y de la eficiencia, `breadth_first_traversal()` sólo debe considerar arcos residuales. Para eso definimos el siguiente criterio de mirada de arco:

```

bool operator () (typename N::Node *, typename N::Arc * a) const
{
 return a->is_residual;
}
};

```

Con este simple filtro, la invocación a `breadth_first_traversal()` sobre el grafo residual e iniciada desde el sumidero opera como si se tratase de la red inversa (sin necesidad de construirla). Si asignamos  $h(t) = 0$ , entonces la siguiente función de visita sobre cada nodo asigna la longitud del camino más corto hacia el sumidero:

743a *(Funciones para Net\_Graph 725)*+≡ △742 □743b

```

template <class Net> static
bool initial_height(Net & net, typename Net::Node * p, typename Net::Arc * a)
{
 node_height<Net>(p) = node_height<Net>(net.get_src_node(a)) + 1;
 return false;
}

```

Si visitamos en amplitud al nodo  $p$  proveniendo desde el arco  $a$ , entonces  $\text{node\_height}_{\text{Net}}(\text{net.get\_src\_node}(a)) + 1$  (que se corresponde con  $h(p) + 1$ ) indica que hay un arco de más desde el sumidero.

El paso 2 del algoritmo genérico 7.6 (Pág. 737) se implanta del siguiente modo:

743b *(Funciones para Net\_Graph 725)*+≡ △743a □743c

```

template <class Net> static void init_height_in_nodes(Net & net)
{
 breadth_first_traversal <Net, Res_Arc<Net> >
 (net, net.get_sink(), &initial_height);
}

```

De este modo, la llamada a `init_height_in_nodes()` asignará a cada nodo su distancia hasta el sumidero.

### 7.10.13.5 Manejo del conjunto $A_h$

El manejo del conjunto  $A_h$  encaja perfectamente con los criterios del problema fundamental (§ 1.3 (Pág. 17)). Así que en primera instancia pudiéramos emplear cualquier estructura de datos entre las diversas estudiadas. Sin embargo, el conocimiento sobre el criterio de elegibilidad de arco nos ofrece maneras de emplear una estructura de datos que, cuando extraiga un nodo de  $A_h$  nos ofrezca una gran probabilidad de que éste contenga arcos elegibles. En este sentido, las implantaciones preferidas son las orientadas hacia el flujo, en particular, colas FIFO y de prioridad.

A efectos de obtener una implantación genérica empleamos las siguientes rutinas para insertar y eliminar del conjunto  $A_h$ :

743c *(Funciones para Net\_Graph 725)*+≡ △743b □744

```

template <class Q_Type> static
void put_in_active_queue(Q_Type & q, typename Q_Type::Item_Type & p)
{
 if (p->control_bits.get_bit(Aleph::Maximum_Flow))
 return;
 p->control_bits.set_bit(Aleph::Maximum_Flow, true);
 q.put(p);
}

```

```

 }

 template <class Q_Type> static
typename Q_Type::Item_Type get_from_active_queue(Q_Type & q)
{
 typename Q_Type::Item_Type p = q.get();
 p->control_bits.set_bit(Aleph::Maximum_Flow, false);
 return p;
}

```

`put_in_active_queue()` inserta el elemento `p` en una cola `q` de tipo genérico `Q_Type`, mientras que `get_from_active_queue` extrae un elementos de la cola `q`. Para detectar eficientemente y evitar inserciones duplicadas, marcamos el bit de control `Maximum_Flow` cuando un nodo esté dentro de `Q_Type`.

Cuando luego de empujar el prefujo de un nodo  $u$ , éste aún permanece activo, él regresa a  $A_h$  con un valor incrementado de  $h(u)$ . Una cuestión de interés es ¿cuán probable es que con el valor incrementado de  $h(u)$   $u$  contenga arcos elegibles?

#### 7.10.13.6 Implantación del algoritmo genérico

Al igual que con los algoritmos basados en búsqueda de caminos de aumento, los de prefujo también englobarse en uno genérico<sup>27</sup>:

```

744 <Funciones para Net_Graph 725>+≡ ▷743c 748▷
 template <class Net, class Q_Type> typename Net::Flow_Type
generic_preflow_vertex_push_maximum_flow
 (Net & net, const bool & leave_residual = false)
{
 init_height_in_nodes(net);
 typename Net::Node * source = net.get_source();
 typename Net::Node * sink = net.get_sink();

 Q_Type q; // instancia el conjunto de nodos activos
 <Preflujo inicial de arcos del nodo fuente 745a>

 while (not q.is_empty()) // mientras haya nodos activos
 {
 typename Net::Node * src = get_from_active_queue(q);
 typename Net::Flow_Type excess = src->in_flow - src->out_flow;

 for (Node_Arc_Iterator<Net, Res_F<Net> > it(src);
 it.has_current() and excess > 0; it.next())
 {
 typename Net::Arc * arc = it.get_current_arc();
 typename Net::Node * tgt = net.get_tgt_node(arc);
 if (node_height<Net>(src) != node_height<Net>(tgt) + 1)
 continue; // el nodo no es elegible

 <Empujar prefujo por arco actual 745b>
 }
 if (excess > 0) // ¿src aún sigue activo?

```

<sup>27</sup>Se omite, por simplicidad, la creación de los supra fuente y sumidero.

```

 { // si ==> incremente h(src) y re-inserte en q
 node_height<Net>(src)++;
 put_in_active_queue(q, src);
 }
}

const typename Net::Flow_Type ret_val = source->out_flow;

return ret_val;
}

```

La estructura es casi idéntica -y desde algunas perspectivas más comprensible- a la expresada en jerga matemática en § 7.6 (Pág. 737).

La instrucción 1 se corresponde con la invocación al método `net.make_residual_net()`, mientras que la 2 se corresponde con la llamada a la rutina `init_height_in_nodes(net)`.

El empuje de preflujo inicial por los arcos emanantes del fuente (instrucción 4) se efectúa así:

745a *(Preflujo inicial de arcos del nodo fuente 745a)≡* (744)

```

for (Node_Arc_Iterator<Net, Res_F<Net> > it(source); it.has_current(); it.next())
{
 typename Net::Arc * arc = it.get_current_arc();
 typename Net::Node * tgt = net.get_tgt_node(arc);
 arc->flow = tgt->in_flow = arc->cap - arc->flow; // inundar arco
 arc->img_arc->flow = 0;
 if (tgt != sink)
 put_in_active_queue(q, tgt); // tgt deviene activo
}
source->out_flow = source->out_cap;

```

Observando la condición de conservación del flujo es tentador para los programadores noveles empujar la capacidad mínima entre el total de salida del fuente y el total de entrada del sumidero. De ese modo podrían disminuirse las iteraciones. Pero esto no necesariamente funciona porque podría ocurrir que algunos de los nodos adyacentes al fuente no deviniese activo, lo cual posibilitaría que el algoritmo jamás lo viese. El primer empuje debe ser a su máxima capacidad.

El empuje de preflujo por el arco es fácilmente comprensible si se entiende el manejo de la red residual<sup>28</sup>:

745b *(Empujar preflujo por arco actual 745b)≡* (744)

```

const typename Net::Flow_Type flow_avail_in_arc = arc->cap - arc->flow;
typename Net::Flow_Type flow_to_push = std::min(flow_avail_in_arc, excess);
arc->flow += flow_to_push;
arc->img_arc->flow -= flow_to_push;
if (arc->is_residual)
{
 net.decrease_out_flow(tgt, flow_to_push);
 net.decrease_in_flow(src, flow_to_push);
}
else
{

```

---

<sup>28</sup>Véase también el método `increase_flow()` en § 7.10.10 (Pág. 725).

```

 net.increase_out_flow(src, flow_to_push);
 net.increase_in_flow(tgt, flow_to_push);
 }
excess -= flow_to_push;

if (is_node_active<Net>(tgt) and tgt != sink and tgt != source)
 put_in_active_queue(q, tgt);

```

#### 7.10.13.7 Análisis del algoritmo de preflujo

El análisis se centra en clasificar y acota a la cantidad de operaciones que ocurren dentro del while (`not q.is_empty()`) -instrucción 5 en algoritmo genérico-. Al respecto, clasifiquemos las operaciones en:

1. Empuje saturante: la ejecución del bloque *(Empujar preflujo por arco actual 745b)* -instrucción 5(b)i en el algoritmo genérico- que logra llenar el arco a su plena capacidad.  
Un empuje saturante implica la añadidura de un nodo al conjunto  $A_h$  y la desaparición del arco en la red residual.
2. Empuje no saturante: una ejecución del bloque *(Empujar preflujo por arco actual 745b)* que no logra llenar el arco a su plena capacidad.
3. Incremento o reetiquetado: ejecución del if (`excess > 0`) -instrucción 5c en el algoritmo genérico-.

La cantidad de incrementos es lo más fácil de contabilizar, según muestra el lema siguiente.

**Lema 7.8 (Cantidad de incrementos)** Sea una red  $N = \langle V, E, s, t, C \rangle$  con red residual  $\bar{N} = \langle V, E', s, t, C' \rangle$ . Entonces la cantidad de incrementos que ocurre en el algoritmo 7.6 (Pág. 737) es a lo sumo  $2V^2$ .

**Demostración** Los incrementos ocurren sobre  $V - 2$  nodos de la red, pues el fuente y el sumidero nunca son activos. Cada nodo se incrementa en uno hasta un máximo de  $2V - 1$ , según el lema 7.7 (Pág. 741). Por tanto, la cantidad de incrementos es  $(V - 1) \times (2V - 1) < 2V^2 \square$

Ahora contabilizamos la cantidad de empujes saturantes.

**Lema 7.9 (Cantidad de empujes saturantes)** Sea una red  $N = \langle V, E, s, t, C \rangle$  con red residual  $\bar{N} = \langle V, E', s, t, C' \rangle$ . Entonces la cantidad de empujes saturantes es menor que  $2VE$ .

**Demostración** Supongamos dos nodos cualesquiera  $u$  y  $v$  a los cuales deseamos contabilizar la cantidad de empujes saturantes. Para que ocurra tal empuje, debe existir en la red residual o un arco  $u \rightarrow v$  o uno  $v \rightarrow u$ .

Supongamos que ocurre un empuje saturante en el arco  $u \rightarrow v$ . Esto implica que  $h(u) = h(v) + 1$ . Esencial notar que para que ocurra otro empuje saturante en  $u \rightarrow v$  primero debe ocurrir un empuje en  $v \rightarrow u$ , pues para que  $u \rightarrow v$  devenga elegible debe cumplirse de nuevo que  $h(u) = h(v) + 1$ .  $h(v)$  debe incrementarse en al menos 2 unidades.

Sabemos, por el lema 7.7 (Pág. 741), que  $\forall x \in V \implies h(x) < 2V$ . Por tanto, la cantidad de veces que un nodo puede incrementarse es menor que  $2V$ . Hay, pues, a lo sumo  $2V$  empujes saturantes en  $u \rightarrow v$ . Multiplicando por la cantidad de arcos tenemos un total máximo de  $2VE$  empujes saturantes  $\square$

Nos falta contabilizar la cantidad de empujes no saturantes, la cual se establece por el siguiente lema.

**Lema 7.10** Sea una red  $N = \langle V, E, s, t, C \rangle$  con red residual  $\bar{N} = \langle V, E', s, t, C' \rangle$ . Entonces la cantidad de empujes saturantes es menor que  $4V^2(V + E)$ .

**Demostración** Definamos la siguiente función potencial sobre el conjunto  $A_h$ :

$$\Phi(A_h) = \sum_{u \in A_h} h(u) \quad (7.28)$$

Hay dos maneras en que puede incrementarse el valor de  $\Phi(A_h)$ :

1. Un empuje saturante: en esta operación se añade un nuevo nodo a  $A_h$ , pero no se aumenta ningún valor de  $h(u)$  para cualquier nodo  $u$ . Por el lema 7.7 (Pág. 741) sabemos que  $h(u) < 2V$  para cualquier  $u$ . Un empuje saturante aumenta  $\Phi(A_h)$  en a lo sumo  $2V$ .

Puesto que la cantidad de empujes saturantes está acotada por  $2VE$  (lema 7.9 (Pág. 746)), el incremento de los empujes saturantes sobre  $\Phi(A_h)$  está acotado por  $2V \times 2VE = 4V^2E$ .

2. Un incremento de altura: cada incremento de un nodo  $u$  aumenta  $\Phi(A_h)$  en uno. La incidencia de un nodo cualquiera  $u$  sobre  $\Phi(A_h)$  es, por el lema 7.7 (Pág. 741), menor que  $2V$ .

Por el lema 7.8 (Pág. 746) sabemos que la cantidad máxima de incrementos es menor que  $2V^2$ . Por tanto, la incidencia de los incrementos sobre  $\Phi(A_h)$  es menor que  $2V^2 \times 2V = 4V^2V$ .

¿Qué acerca de los empujes no-saturantes? Supongamos que ocurre un empuje no saturante sobre un arco  $u \rightarrow v$ . En este momento,  $u$  deviene inactivo y  $v$  probablemente activo. Así pues,  $\Phi(A_h) = \Phi(A_h) - h(u) + h(v)$ , pero, puesto que  $h(u) = h(v) + 1$ , entonces  $\Phi(A_h) = \Phi(A_h) - h(v) - 1 + h(v) = \Phi(A_h) - 1$ . O sea,  $\Phi(A_h)$  decrece al menos en una unidad.

Habiendo concluido que los empujes no saturantes no incrementan el valor de  $\Phi(A_h)$ , acotar la función potencial estará dado por:

$$\Phi(A_h) < \underbrace{4V^2E}_{\text{Incrementos}} + \underbrace{4V^2V}_{\text{Empujes saturantes}} = 4V^2(V + E) \blacksquare$$

Ahora tenemos todas las herramientas necesarias para concluir acerca del desempeño de algoritmo genérico.

**Proposición 7.15** Sea una red  $N = \langle V, E, s, t, C \rangle$  con red residual  $\bar{N} = \langle V, E', s, t, C' \rangle$ . Sea  $A_h : V \rightarrow \mathcal{N}$  una función de altura. Entonces la cantidad de operaciones que ejecuta el algoritmo genérico 7.6 (Pág. 737) es  $\mathcal{O}(V^2E)$ .

**Demostración** Inmediata de los lemas 7.8 (Pág. 746), 7.9 (Pág. 746) y 7.10 (Pág. 747)

■

#### 7.10.13.8 Empuje de preflujo FIFO

El orden en que se empuje el preflujo a través de los nodos puede incidir notablemente en el rendimiento de un algoritmo de preflujo. Si tratamos de que el preflujo alcance lo más rápidamente posible el sumidero, entonces un enfoque que extraiga de la cola los nodos en profundidad probablemente tiene, heurísticamente hablando, más posibilidades de arribar más rápido al sumidero. En este sentido, una manera de emular el procesamiento en profundidad consiste en encolar FIFO los nodos activos. Este enfoque es directamente instrumentable a partir del algoritmo genérico:

```
748 <Funciones para Net_Graph 725>+≡ <744 752a>
 template <class Net> typename Net::Flow_Type
 fifo_preflow_maximum_flow
 (Net & net, const bool & leave_residual = false)
 {
 return generic_preflow_vertex_push_maximum_flow
 <Net,DynListQueue<typename Net::Node*>>(net, leave_residual);
 }
}
```

Se trata del algoritmo genérico con parámetro tipo `Q_Type` instanciado a una cola de tipo `DynListQueue<T>` estudiado en § 2.6.6 (Pág. 130).

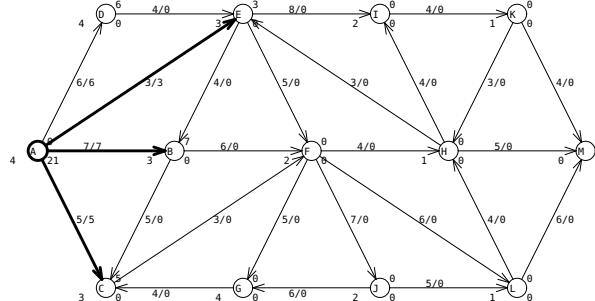


Figura 7.84: Fase inicial de un algoritmo de preflujo sobre la red de la figura 7.73 (Pág. 725). Los arcos salientes del fuente son inundados (y están resaltados). Las alturas de cada nodo están ubicadas al suroeste

Una ejecución completa de este algoritmo sobre el grafo de la figura 7.73 se muestra en la figura 7.85 (pags. 749-752). Nótese que el sumidero es alcanzado en la octava figura con un valor de flujo de 6.

La heurística intenta recorrer en profundidad a través de arcos elegibles, pero no se trata de un recorrido en profundidad tradicional. A destacarse el sentido heurístico, pues en la elegibilidad de los arcos inciden los valores de capacidades y el orden en que éstos se presenten durante las iteraciones.

Este algoritmo es  $\mathcal{O}(V^3)$ . Una vía de demostración consiste en definir como función potencial  $\Phi(A_h) = \begin{cases} 0 & , A_h = \emptyset \\ \max\{h(u) \mid u \in A_h\} & , A_h \neq \emptyset \end{cases}$  y examinar el valor de  $\Phi(A_h)$  según

fases de metidas en la cola. La primera fase la delimita el conjunto de nodos activos  $[u_1, u_2, \dots, u_m]$  justo después de ejecutar *(Preflujo inicial de arcos del nodo fuente 745a)*. Luego, las fases siguientes están delimitadas por el conjunto de nodos activos luego de que  $u_m$  sale de la cola. El análisis se hace sucesivamente.

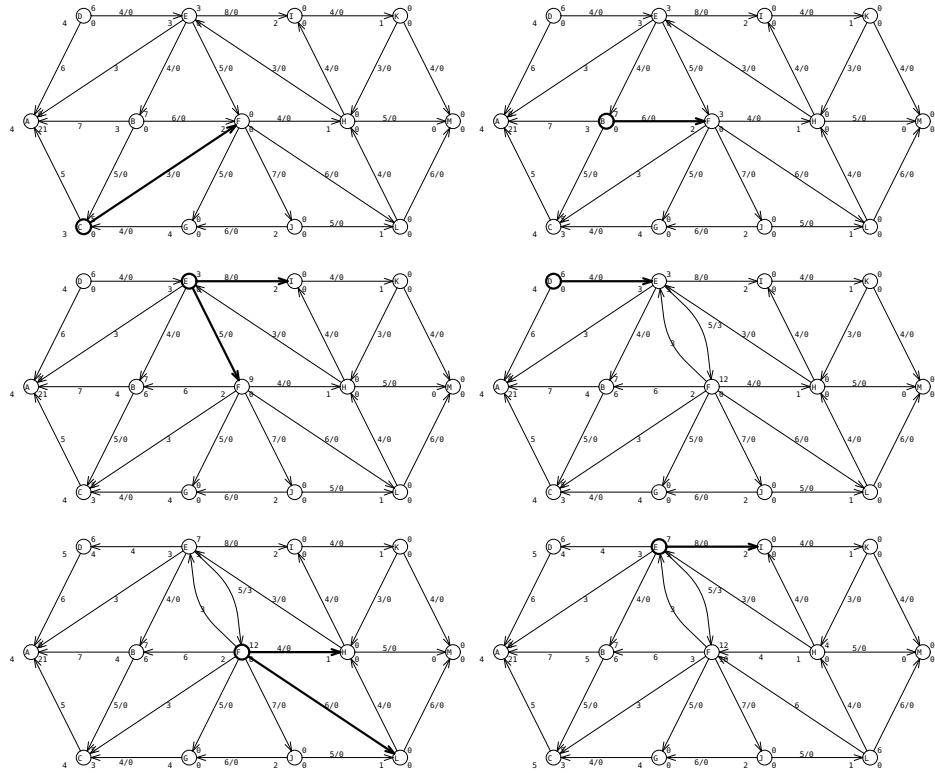
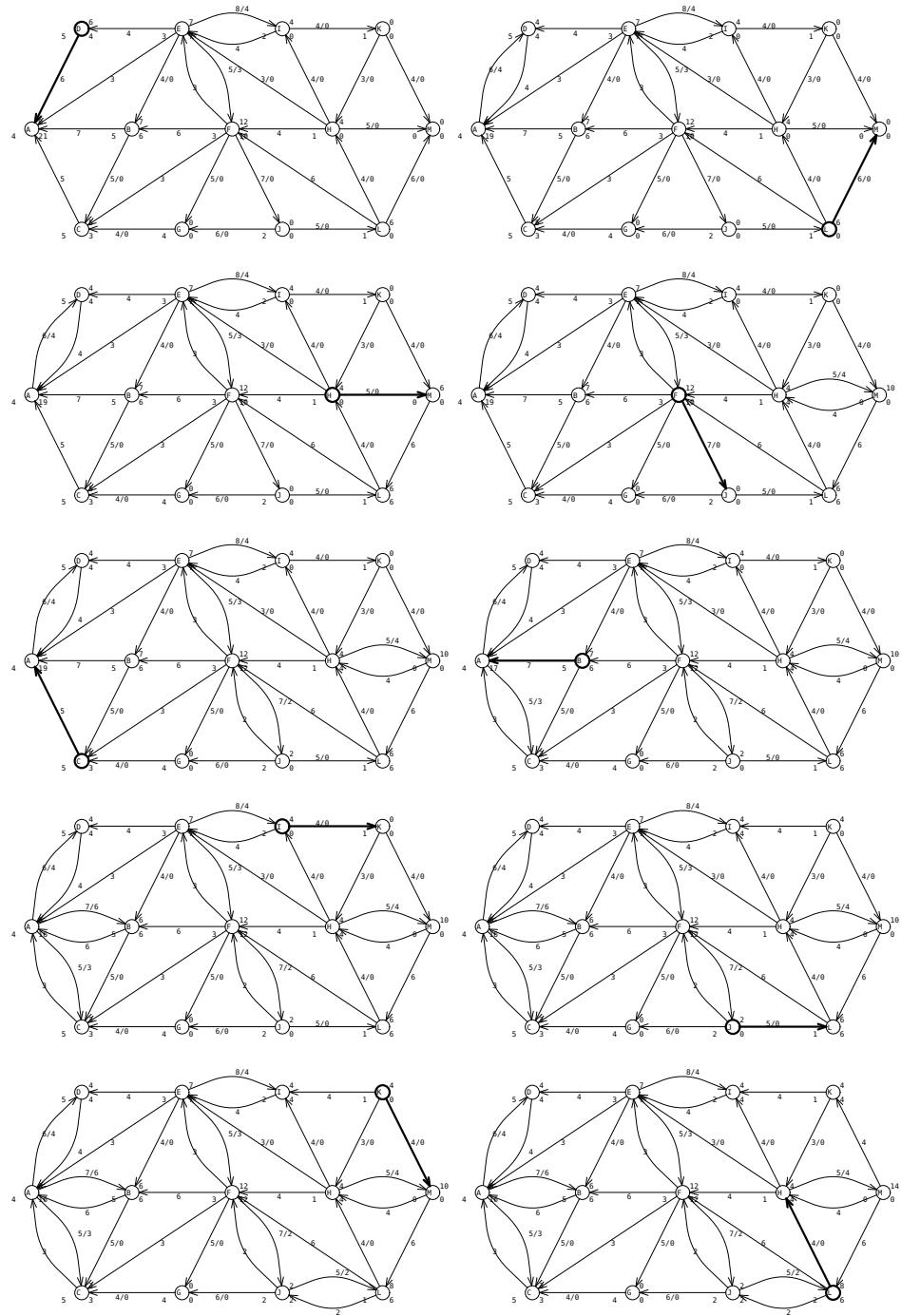
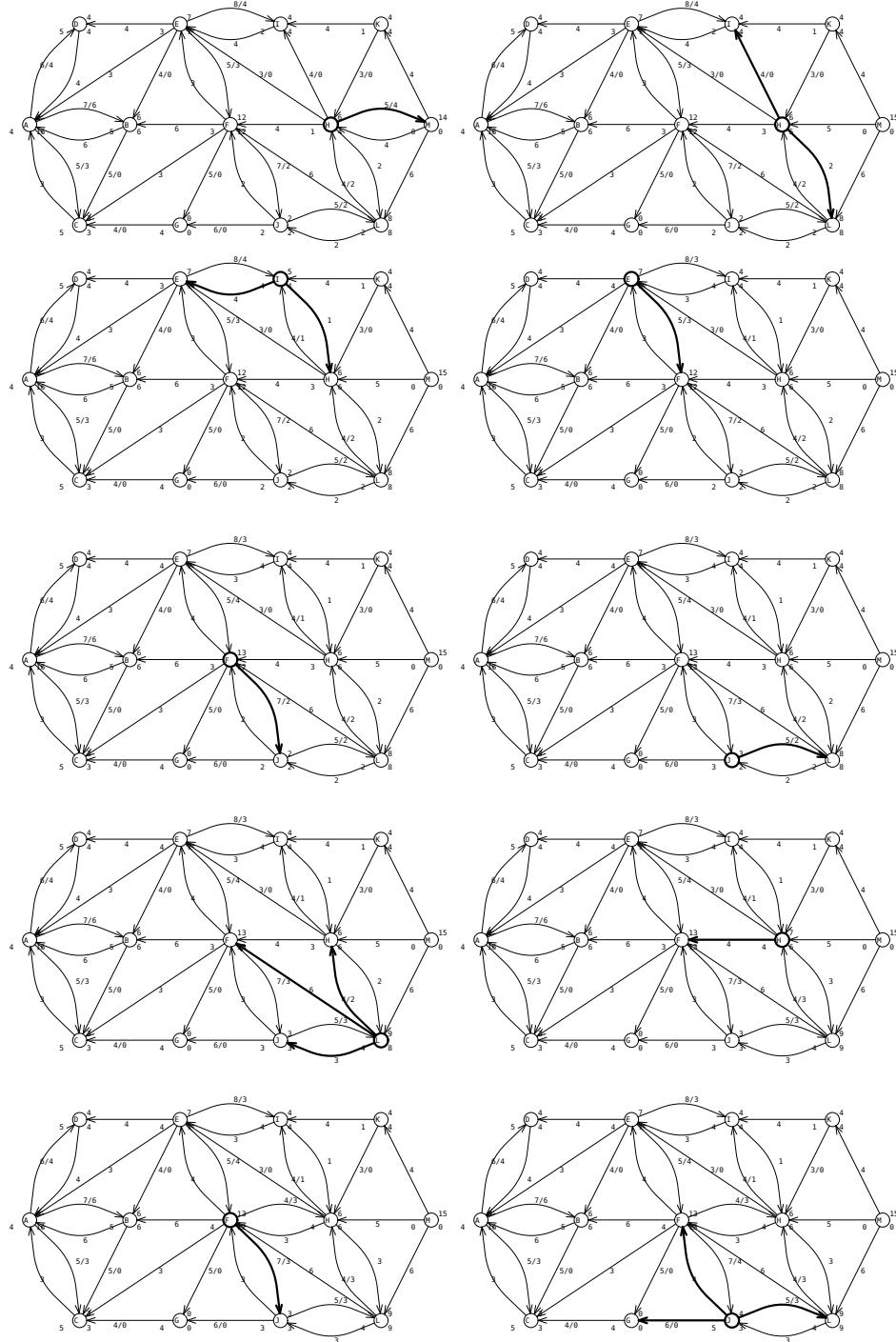


Figura 7.85: Algoritmo preflujo con cola sobre red de la figura 7.73 (Pág. 725). El nodo extraído junto con los arcos elegibles se resaltan. El flujo entrante se ubica al noreste del nodo, el saliente al sureste y la altura a suroeste





#### 7.10.13.9 Empuje de preflujo con mayor distancia

La segunda heurística más popular, la cual empíricamente resulta mejor que la cola FIFO, consiste en mirar los nodos activos en amplitud. La idea intuitiva es empujar todo el preflujo posible antes de avanzar un nivel, es decir, desde los nodos más distantes del sumidero hasta los más cercanos. La estructura de datos que nos ofrece esta disciplina es una cola de prioridad cuya extracción retorne el nodo activo con mayor altura, o sea, se le da prioridad al nodo activo más cercano a fuente -o más lejano al sumidero segú

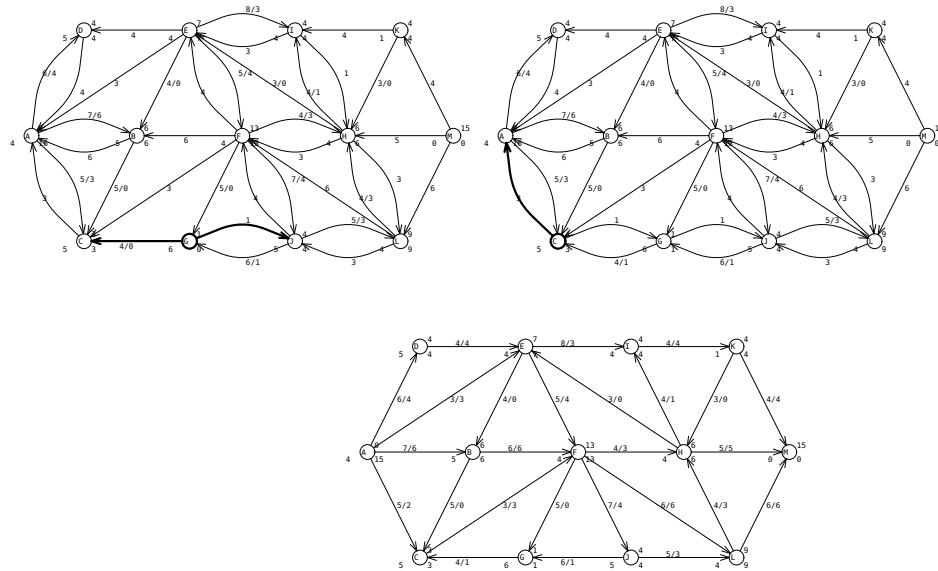


Figura 7.86: Estado final de ejecución algoritmo de preflujo sobre la figura 7.73 (Pág. 725) para cola FIFO

convenga ver-.

Las colas de prioridad están disponibles bajo cualquiera de los tipos asociados a los heaps. En nuestro caso emplearemos el tipo `DynBinHeap<Key>`, el cual es la versión dinámica del tipo `BinHeap<Key>` estudiado en § 4.7.6 (Pág. 293).

El criterio de comparación entre los elementos del heap se define de la siguiente manera:

752a *Funciones para Net\_Graph 725* +≡ ↳ 748 752b ▷  
`template <class Net> struct Compare_Height  
{  
 bool operator () (typename Net::Node * n1, typename Net::Node * n2) const  
 {  
 return node_height<Net>(n1) > node_height<Net>(n2);  
 }  
};`

La comparación está invertida a efectos de garantizar que siempre se extraiga el nodo con mayor prioridad.

Establecido el criterio de comparación, el empuje con la heurística por amplitud es directo:

752b *Funciones para Net\_Graph 725* +≡ ↳ 752a 757 ▷  
`template <class Net> typename Net::Flow_Type  
heap_preflow_maximum_flow(Net & net, const bool & leave_residual = false)  
{  
 return generic_preflow_vertex_push_maximum_flow  
 <Net, DynBinHeap<typename Net::Node*, Compare_Height<Net> >>  
 (net, leave_residual);  
}`

La ejecución completa de este algoritmo sobre el grafo de la figura 7.73 es mostrada en la figura 7.87 (págs. 753-756). El nodo sumidero se alcanza en la onceava iteración, con un valor de flujo de 4; tres iteraciones más que con la cola FIFO y un valor de flujo menos. En términos de la heurística podemos decir que puesto que se trata de ir en amplitud,

se demora más en alcanzar el sumidero. La misma observación aplica para el valor de flujo inferior; el flujo está concentrado hacia el fuente, mientras que con la cola FIFO está dispersado a través de los caminos que la suerte en profundidad haya determinado.

Es demostrable que este algoritmo se ejecuta en  $\mathcal{O}(V^3)$  pasos. Para eso se define la misma función potencial que para la cola FIFO, pero se mira la secuencia con que los nodos activos son examinados.

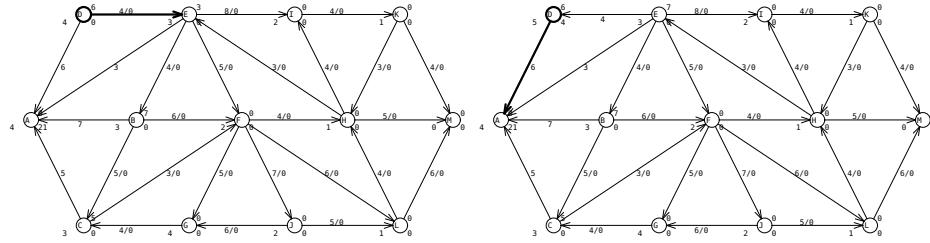
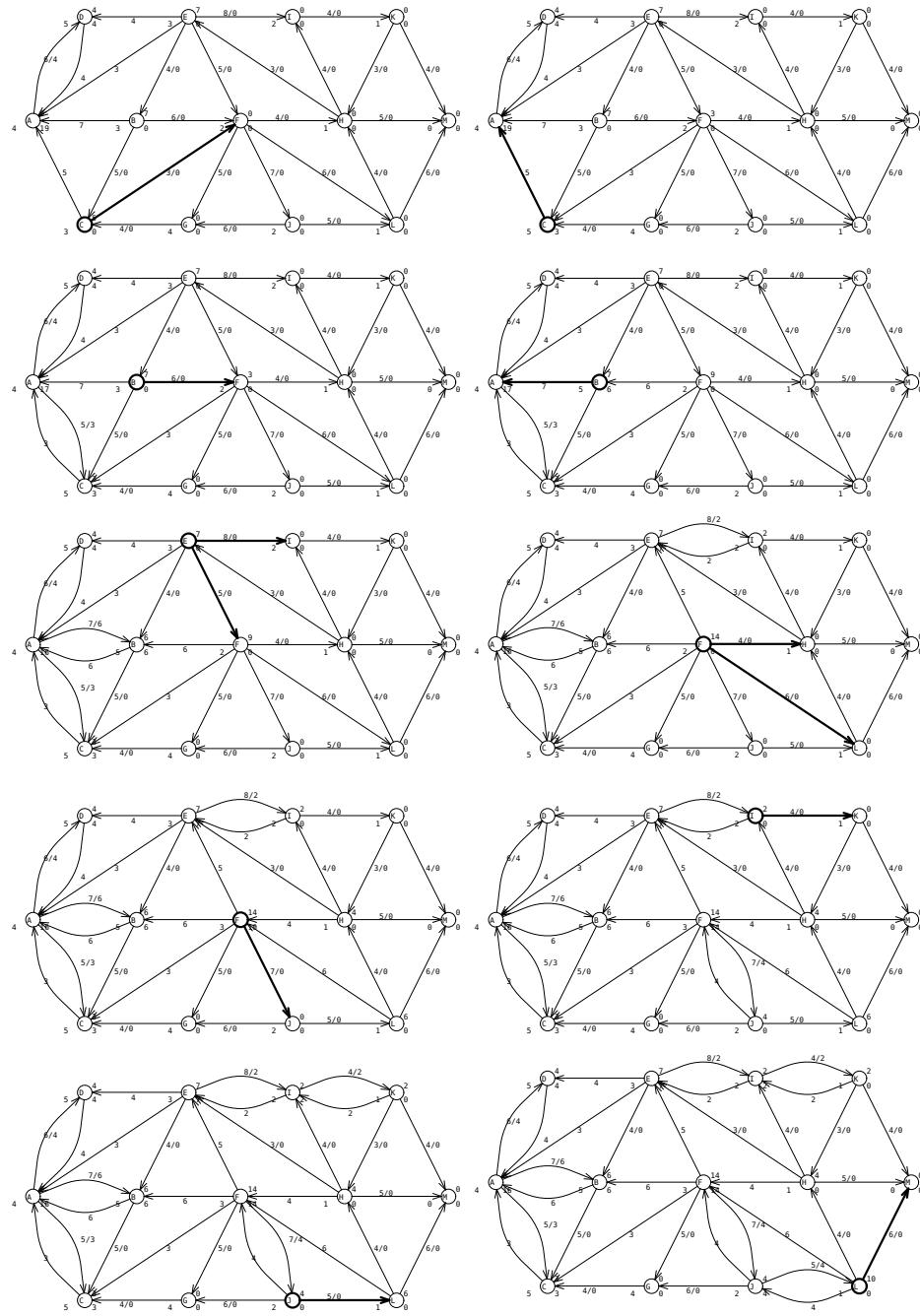
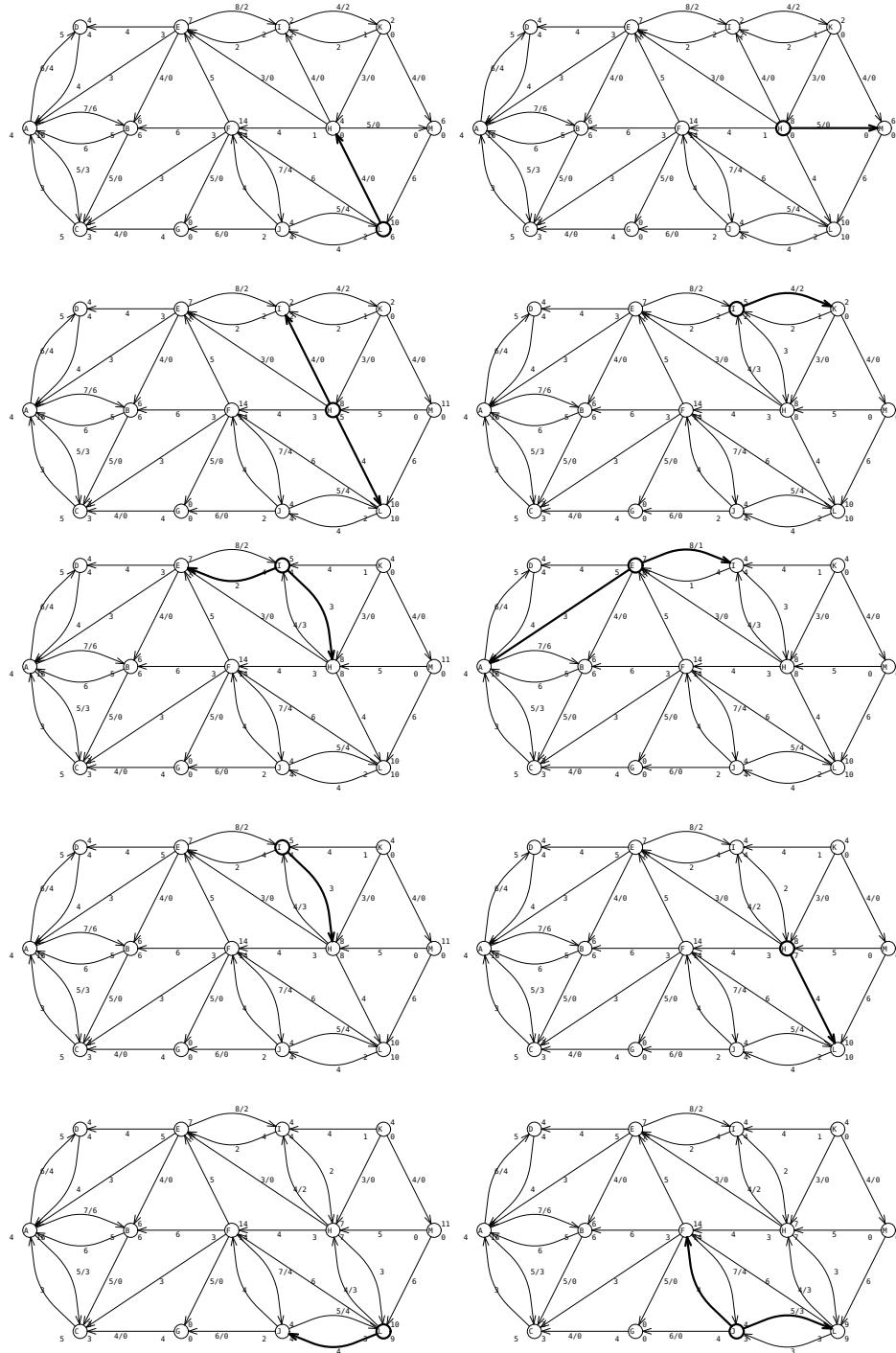


Figura 7.87: Algoritmo de preflujo con heap sobre la red de la figura 7.73 (Pág. 725)





#### 7.10.13.10 Empuje de preflujo con cola aleatoria

Es muy probable extraer nodos activos sin arcos elegibles. Por eso es deseable minimizar la cantidad de esta clase de extracciones vanas, lo cual depende de la suerte topológica del grafo, de la suerte con que se miren los arcos y de la suerte con que se presenten los valores de las capacidades.

Así, tanto parece incidir la suerte que es tentador introducirla en el algoritmo. A ese tenor, una estructura de datos para representar  $A_h$  se llama “cola aleatoria”, la cual

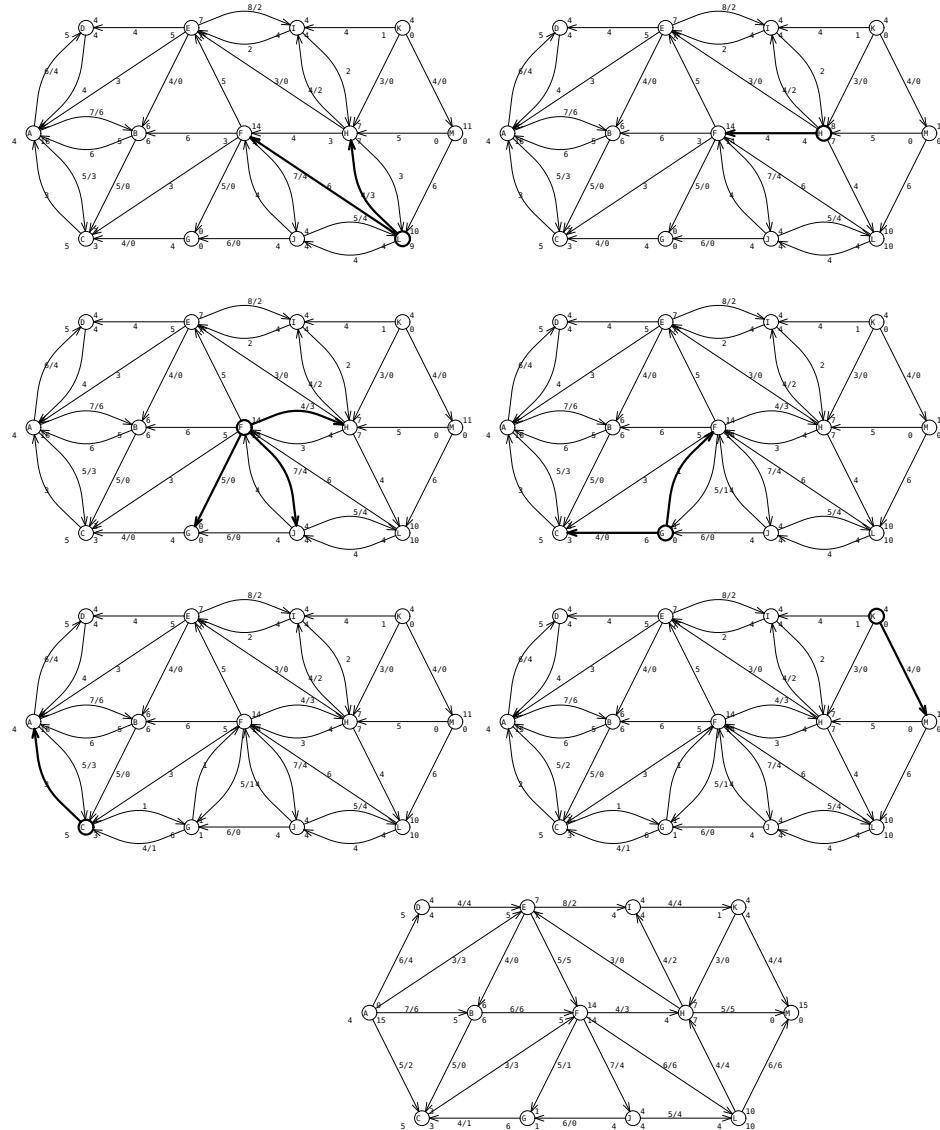


Figura 7.88: Estado final de ejecución algoritmo de preflujo sobre la figura 7.73 (Pág. 725) con cola de prioridad

| Heurística     | Cantidad de iteraciones                |
|----------------|----------------------------------------|
| Cola FIFO      | 44                                     |
| Cola prioridad | 43                                     |
| Cola aleatoria | min = 28 - promedio = 40,34 - max = 50 |

Table 7.1: Cantidad de iteraciones para las heurísticas de empuje de preflujo

puede implantarse muy fácilmente:

756  $\langle tpl\_random\_queue.H \rangle$

```
template <class T> class Random_Set
{
 DynArray<T> array;
 gsl_rng * r;
```

```

void put(const T & item) { array[array.size()] = item; }

T get()
{
 const size_t pos = gsl_rng_uniform_int(r, array.size()); // al azar
 T ret_val = array.access(pos);
 array.access(pos) = array.access(array.size() - 1);
 array.cut(array.size() - 1);
 return ret_val;
}
};

Uses DynArray 34.

```

O sea, una cola cuya operación `get()` selecciona al azar el elemento a eliminar. Con esta estructura podemos implantar la siguiente especialización:

757

*(Funciones para Net\_Graph 725)* +≡

△752b 761△

```

template <class Net> typename Net::Flow_Type
random_preflow_maximum_flow(Net & net, const bool & leave_residual = false)
{
 return generic_preflow_vertex_push_maximum_flow
 <Net, Random_Set<typename Net::Node*>> (net, leave_residual);
}

```

Una ejecución completa, buena, con esta cola, se muestra en la figura 7.89 (Págs. 758-760).

El promedio se calculó para 100 ejecuciones. Para el ejemplo, la cola aleatoria ofrece una ejecución promedio mejor a las otras dos heurísticas. En el ejemplo mostrado se tuvo la suerte de alcanzar el sumidero en cinco iteraciones.

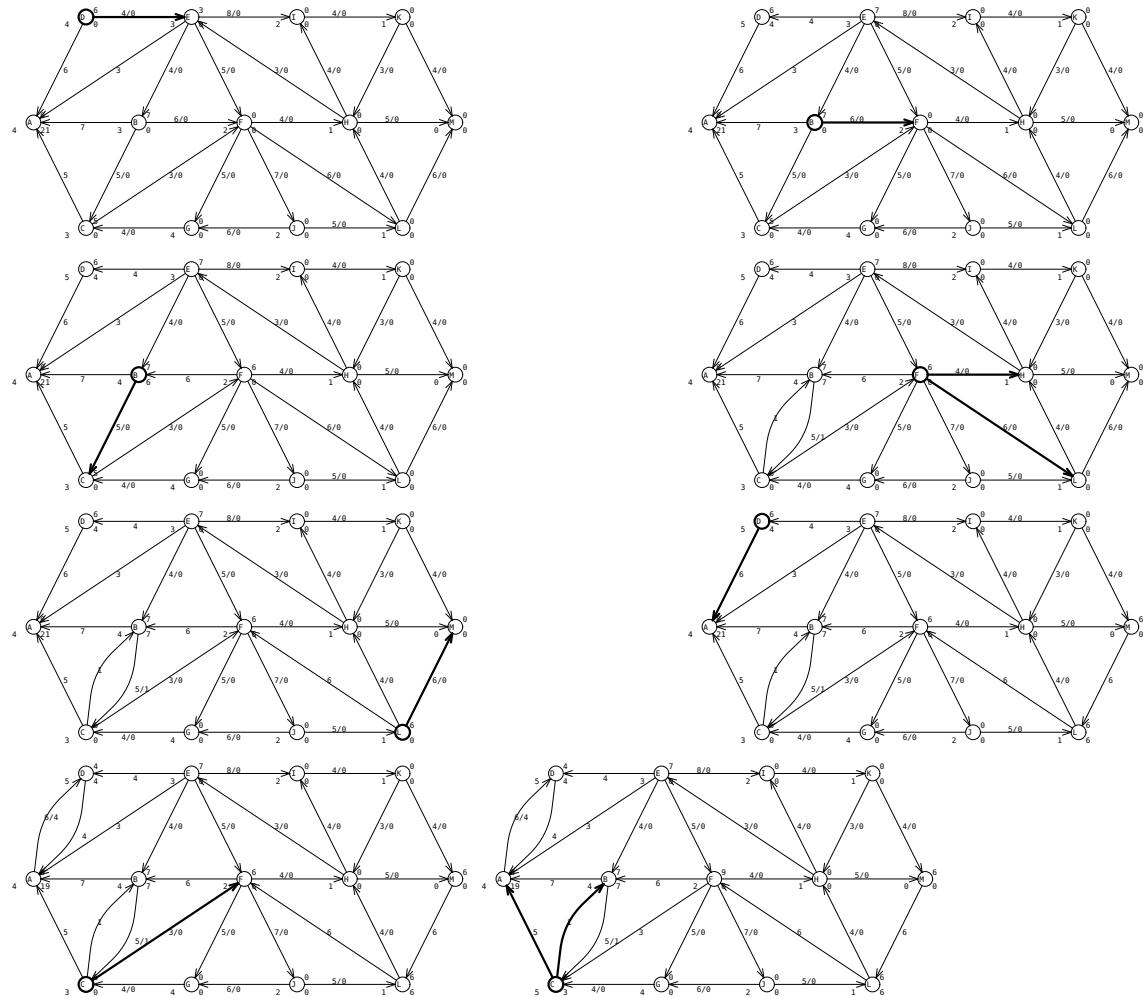
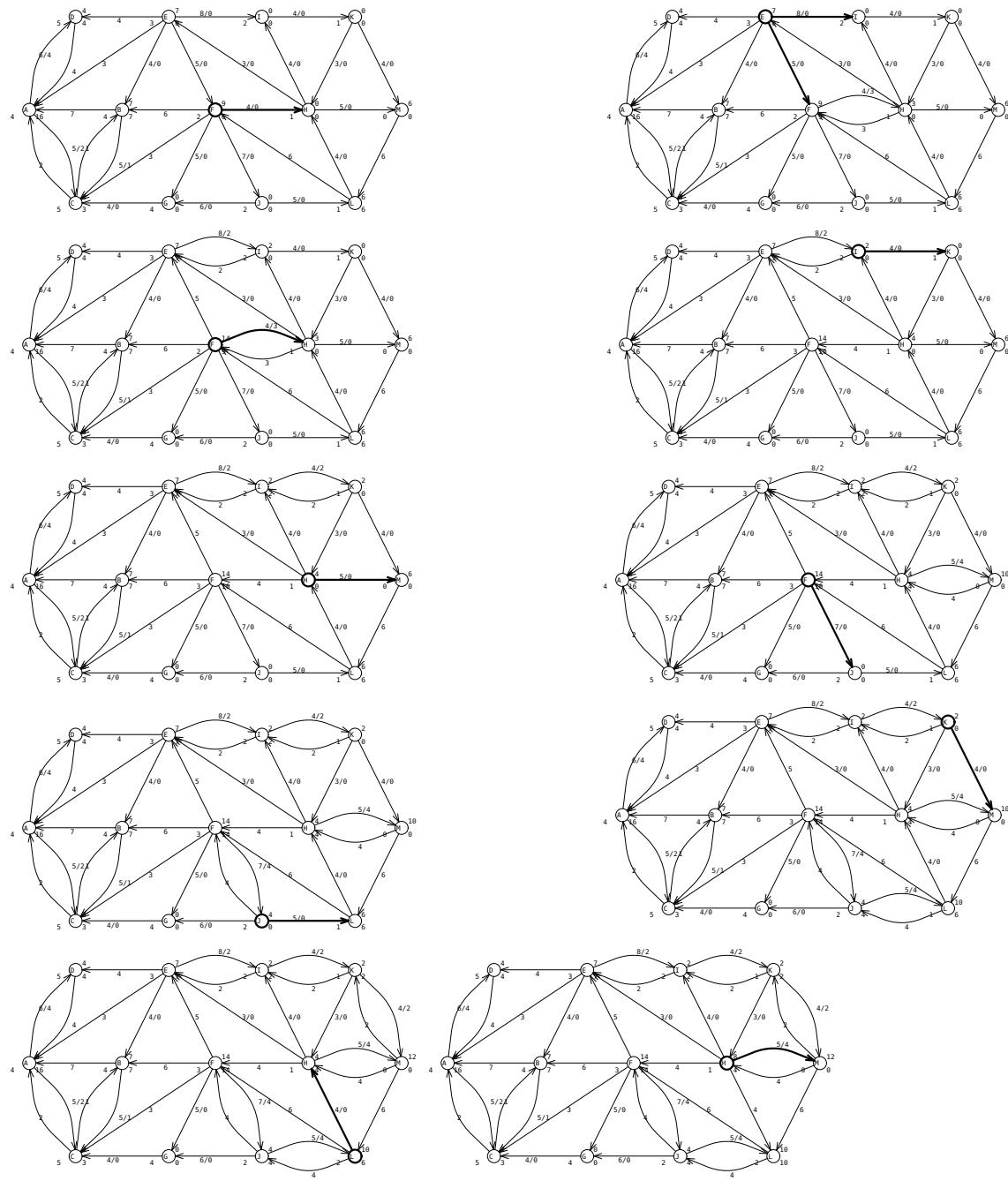


Figura 7.89: Una ejecución completa favorable (32 iteraciones) del algoritmo de preflujo con cola aleatoria sobre la red de la figura 7.73 (Pág. 725)



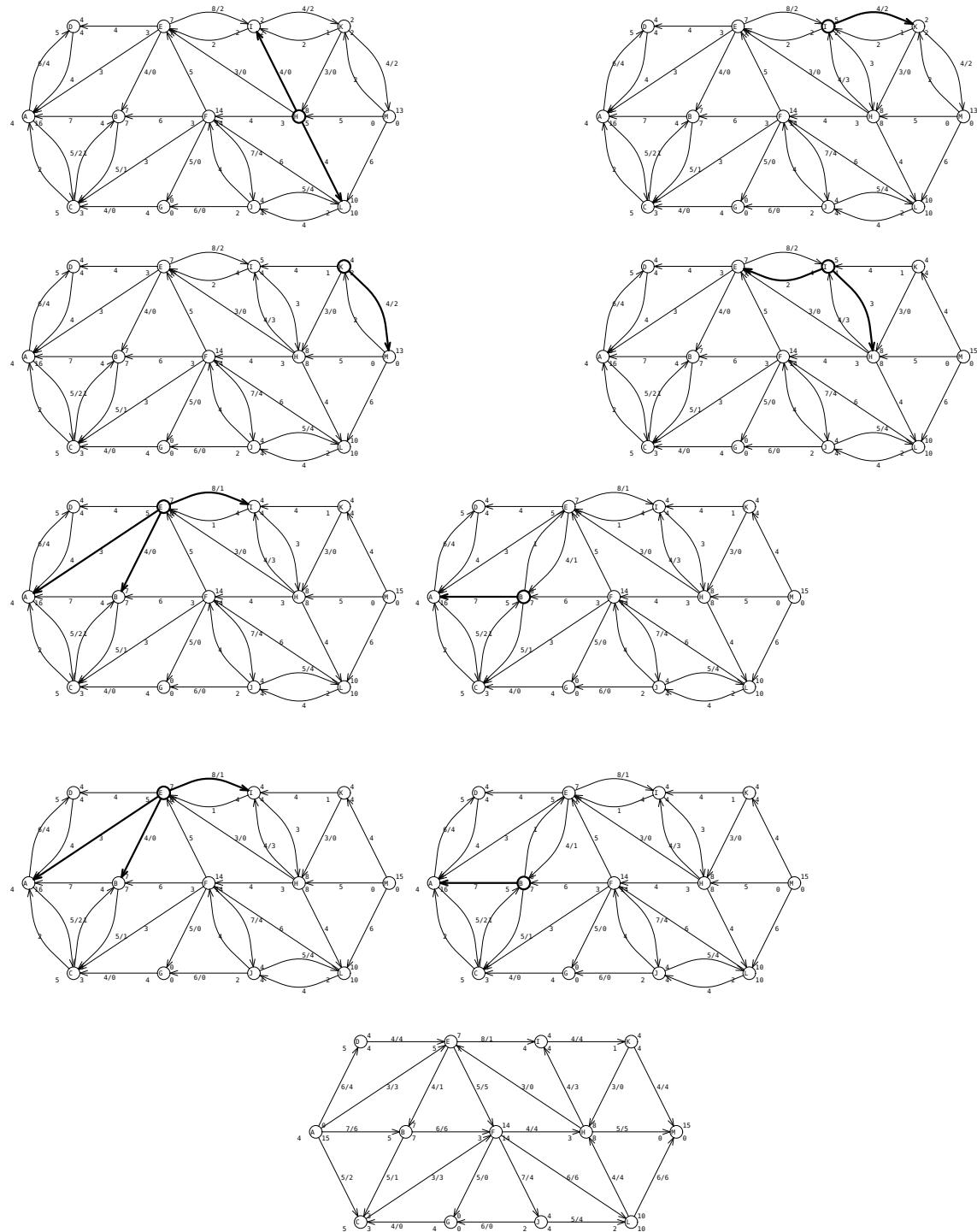


Figura 7.90: Estado final de ejecución algoritmo de preflujo sobre la figura 7.73 (Pág. 725) con cola aleatoria

#### 7.10.13.11 Algoritmo genérico de empuje de preflujo por arcos

Una alternativa al algoritmo genérico consiste en manejar una estructura de datos orientada al conjunto de arcos elegibles en lugar del de los nodos activos. La extracción de la cola sería la de un arco elegible según algún criterio combinado con la altura del nodo.

Debemos tener especial cuidado con el reetiquetado, pues cuando ocurre un incremento de altura de un nodo, arcos devienen elegibles, pero otros que eran elegibles devienen inelegibles. La estructura de datos debe manejar tanto los nodos activos como los arcos elegibles.

Hay varias maneras de mantener el conjunto de arcos elegibles. Optaremos por emplear dos conjuntos: el de los arcos elegibles, llamado `q` con tipo genérico `QA_Type`, y el de los nodos activos, llamado `p` con tipo genérico `QN_Type`. No nos preocuparemos, “en línea”, por sacar arcos que dejen de ser elegibles cuando ocurra un incremento. Dicho lo anterior, consideremos el siguiente algoritmo genérico:

761

```

⟨Funciones para Net_Graph 725⟩+≡ ◁757 764b▷
 template <class Net, class QN_Type, class QA_Type>
 typename Net::Flow_Type
generic_preflow_edge_maximum_flow(Net & net, const bool & leave_residual = false)
{
 QA_Type q; // conjunto de arcos elegibles
 QN_Type p; // conjunto de nodos activos
 typename Net::Node * source = net.get_source();
 typename Net::Node * sink = net.get_sink();
 init_height_in_nodes(net);

 ⟨Inundar arcos del fuente 762a⟩

 while (true)
 {
 while (not q.is_empty()) // mientras haya arcos elegibles
 {
 typename Net::Arc * arc = get_from_active_queue(q);
 typename Net::Node * src = net.get_src_node(arc);
 typename Net::Node * tgt = net.get_tgt_node(arc);
 if (node_height<Net>(src) != node_height<Net>(tgt) + 1)
 continue; // No ==> ignore arco

 typename Net::Flow_Type excess = src->in_flow - src->out_flow;
 if (excess == 0) // ¿fuente descargado en iteración anterior?
 {
 // sí ==> ignore arco
 remove_from_active_queue(p, src);
 continue; // avance al siguiente arco elegible
 }

 ⟨Empujar prefujo por arc 762b⟩

 if (is_node_active<Net>(tgt) and tgt != source and tgt != sink)
 ⟨Meter en q arcos elegibles de tgt 763⟩

 if (excess == 0) // ¿se descargó el fuente?
 {
 // sí ==> fuente deviene inactivo
 remove_from_active_queue(p, src);
 continue;
 }
 if (src != source and src != sink and // ¿empuje no saturante?

```

```

 not has_arcs_in_active_queue<Net>(src))
{ // si ==> incremente h(active) y re-inserte arc en q
 remove_from_active_queue(p, src);
 node_height<Net>(src)++;
 put_in_active_queue(p, src);

 <Meter en q arcos elegibles de src 764a>
}

if (p.is_empty()) // quedan nodos activos?
 break; // ya no hay nodos activos

typename Net::Node * src = get_from_active_queue(p);
node_height<Net>(src)++;
put_in_active_queue(p, src);

<Meter en q arcos elegibles de src 764a>
} // fin while (true)
return source->out_flow;
}

```

El algoritmo maneja dos lazos continuos (while), cada uno asociado a extraer y procesar un elemento de un conjunto. El primer conjunto q, de tipo QN\_Type, representa el de nodos activos, mientras que el segundo p, de tipo QA\_Type, el de arcos elegibles. Ambos lazos son parte de un supralazo que rompe cuando ya no hay más nodos activos.

Los conjuntos q y p son inicializados por el bloque inicial *<Inundar arcos del fuente 762a>*, cuyo fin algorítmico es el mismo que el *<Preflujo inicial de arcos del nodo fuente 745a>*: crear el máximo preflujo inicial:

762a *<Inundar arcos del fuente 762a>*≡ (761)

```

for (Node_Arc_Iterator<Net, Res_F<Net> > i(source); i.has_current(); i.next())
{
 typename Net::Arc * arc = i.get_current_arc();
 typename Net::Node * tgt = net.get_tgt_node(arc);
 arc->flow = tgt->in_flow = arc->cap - arc->flow; // inundar arco
 arc->img_arc->flow = 0;

 // meter arcos elegibles de tgt
 for (Node_Arc_Iterator<Net, Res_F<Net> > j(tgt); j.has_current(); j.next())
 {
 typename Net::Arc * a = j.get_current_arc();
 if (node_height<Net>(tgt) == node_height<Net>(net.get_tgt_node(a)) + 1)
 put_in_active_queue(q, a);
 }
 put_in_active_queue(p, tgt); // mete en cola de nodos activos
}
source->out_flow = source->out_cap;

```

El bloque *<Empujar preflujo por arc 762b>*, es similar a *<Empujar preflujo por arco actual 745b>*:

762b *<Empujar preflujo por arc 762b>*≡ (761)

```

const typename Net::Flow_Type push = std::min(excess, arc->cap - arc->flow);
arc->flow += push;
arc->img_arc->flow -= push;

```

```

if (arc->is_residual)
{
 net.decrease_out_flow(net.get_tgt_node(arc), push);
 net.decrease_in_flow(net.get_src_node(arc), push);
}
else
{
 net.increase_in_flow(net.get_tgt_node(arc), push);
 net.increase_out_flow(net.get_src_node(arc), push);
}
excess -= push;

```

Cuando se empuja el preflujo por arc, el nodo destino tgt probablemente devenga activo. En ese caso, hay que meter en la cola q los arcos salientes de tgt que sean elegibles y a tgt en p:

763 *<Meter en q arcos elegibles de tgt 763>*≡ (761)

```

{
 for (Node_Arc_Iterator<Net, Res_F<Net> > it(tgt); it.has_current(); it.next())
 {
 typename Net::Arc * a = it.get_current_arc();
 if (node_height<Net>(tgt) == node_height<Net>(net.get_tgt_node(a)) + 1)
 put_in_active_queue(q, a);
 }
 put_in_active_queue(p, tgt);
}

```

Después de *<Meter en q arcos elegibles de tgt 763>* puede suceder que src se haya descargado completamente. En ese caso, src deviene inactivo y hay que eliminarlo del conjunto de nodos activos. Esa es la función del if (*excess == 0*).

El bloque dentro del if (*src != source and ...*) es quizá la parte más difícil de comprender. Aquí hay que verificar si el nodo src devino inactivo, lo cual se conoce mediante el predicado *excess > 0*. Si es falso, entonces ya es con seguridad inactivo, pues el exceso fue drenado en el arco. De lo contrario debemos verificar si *<Empujar preflujo por arc 762b>* fue o no saturante, cuestión que determina la existencia de arcos de salida de src que estén dentro de la cola; este es el fin de la rutina *has\_arcs\_in\_active\_queue<Net, Q\_Type>(q, src)*, cuya implantación sólo se remite a inspeccionar el bit *Maximum\_Flow* en los arcos saliente de src.

En el incremento de altura de un nodo hay que asegurarse de observar varios asuntos:

- Puesto que src está activo, éste se encuentra dentro del conjunto p. Según la estructura de datos e interfaz de QN\_Type, no se puede llevar a cabo directamente el incremento *node\_height<Net>(src)++* si src está contenido dentro del conjunto, pues se afecta la disciplina de orden de la estructura de datos. Por esa razón, el incremento saca src de p, incrementa la altura y mete de nuevo en p.
- El incremento *node\_height<Net>(src)++* puede provocar la aparición y desaparición de arcos elegibles, tanto salientes desde como entrantes a src. Por los arcos salientes no hay problema en revisarlos, son directamente accesibles, pero los entrantes no lo son. En este caso, el haber escogido que la red residual esté dentro de la propia red cumple un papel esencial, pues por cada arco entrante a src existe un arco residual que nos

permite conocer en  $\mathcal{O}(1)$  el arco entrante a `src`.

764a *<Meter en q arcos elegibles de src 764a>* $\equiv$  (761)  
`for (Node_Arc_Iterator<Net> it(src); it.has_current(); it.next())`  
`{`  
 `typename Net::Arc * a = it.get_current_arc(); // sale de src`  
 `if ((node_height<Net>(src) == node_height<Net>(net.get_tgt_node(a)) + 1)`  
 `and (a->cap - a->flow > 0))`  
 `put_in_active_queue(q, a);`  
  
 `typename Net::Arc * i = a->img_arc; // arco entrante a src`  
 `if ((i->cap - i->flow > 0) and // i es elegible?`  
 `(node_height<Net>(net.get_src_node(i)) == node_height<Net>(src) + 1))`  
 `put_in_active_queue(q, i);`  
`}`

El iterador no utiliza el filtro `Res_F` definido en § 7.10.9 (Pág. 725), pues éste nos filtraría justamente arcos saturados y nos impediría ver el nodo origen de un arco entrante a `src`.

Los arcos contenidos en `q` que dejan de ser elegibles, simplemente se ignoran.

Podríamos diseñar una estructura de datos que permita el incremento directo y elimine los arcos no elegibles en  $\mathcal{O}(1)$ .

¿Para qué este enfoque según los arcos elegibles?, ¿en qué difiere del algoritmo genérico según nodos activos? Resulta que con este algoritmo es intuitivamente más sencillo pensar el orden de tratamiento de los nodos activos. Consideraremos algunas especializaciones.

### Selección de arcos por profundidad

Un primer criterio consiste en emular el recorrido en profundidad:

764b *<Funciones para Net\_Graph 725>* $\equiv$  <761 765a>  
`template <class Net> typename Net::Flow_Type`  
`depth_first_preflow_edge_maximum_flow`  
`(Net & net, const bool & leave_residual = false)`  
`{`  
 `typedef DynSetTreap<typename Net::Node*> Active;`  
 `typedef DynListStack<typename Net::Arc*> Stack;`  
 `return generic_preflow_edge_maximum_flow <Net, Active, Stack>`  
 `(net, leave_residual);`  
`}`

Uses `DynListStack 105c`.

Los nodos activos se guardan en un treap (§ 6.3 (Pág. 464)) cuya operación `get()` retorna el mayor valor, o sea, el nodo de mayor altura.

Un incremento cuesta  $\mathcal{O}(\lg V)$ , coste que puede llevarse a  $\mathcal{O}(1)$  si se elabora con cuidado una tabla hash, cuya función hash incluya la altura y la dirección del nodo. Podría basarse, por ejemplo, en el tipo `ODhashTable` definido en § 5.1.4.5 (Pág. 403).

Los arcos elegibles se encuentran en una pila, por lo que su procesamiento obedece al arquetipo en profundidad.

### Selección de arcos por amplitud

Otra versión, estructuralmente muy similar pero de procesamiento muy distinto, consiste en emular el recorrido en amplitud y se define como:

765a *(Funciones para Net\_Graph 725) +≡* △764b 765b ▽

```
template <class Net> typename Net::Flow_Type
breadth_first_preflow_edge_maximum_flow
(Net & net, const bool leave_residual = false)
{
 typedef DynSetTreap<typename Net::Node*> Active;
 typedef DynListQueue<typename Net::Arc*> Queue;
 return generic_preflow_edge_maximum_flow<Net, Active, Queue>
 (net, leave_residual);
}
```

La cual recorre los arcos elegibles en profundidad.

### Selección de arcos por prioridad

Podemos especificar una prioridad de elegibilidad entre dos arcos, por ejemplo:

765b *(Funciones para Net\_Graph 725) +≡* △765a 765c ▽

```
template <class Net> struct Compare_Arc
{
 bool operator () (typename Net::Arc * a1, typename Net::Arc * a2) const
 {
 typename Net::Node * src1 = (typename Net::Node *) a1->src_node;
 typename Net::Node * src2 = (typename Net::Node *) a2->src_node;
 if (src1->counter == src2->counter)
 return a1->cap - a1->flow > a2->cap - a2->flow;

 return src1->counter > src2->counter;
 }
};
```

El criterio mira las alturas de los nodos origen de cada arco. Si éstas son iguales, entonces se elige el arco que tenga mayor capacidad disponible, bajo la expectativa de que se empuje la mayor cantidad de flujo. Si, por el contrario, las alturas difieren, entonces se sigue el criterio de prioridad y se selecciona el arco con mayor altura.

Ahora podemos definir otra clase de algoritmo:

765c *(Funciones para Net\_Graph 725) +≡* △765b 766 ▽

```
template <class Net> typename Net::Flow_Type
priority_first_preflow_edge_maximum_flow
(Net & net, const bool & leave_residual = false)
{
 typedef DynSetTreap<typename Net::Node*> Active;
 typedef DynBinHeap<typename Net::Arc*, Compare_Arc<Net> > Prio_Queue;
 return generic_preflow_edge_maximum_flow <Net, Active, Prio_Queue>
 (net, leave_residual);
}
```

### Selección de arcos al azar

Finalmente, también podemos elegir al azar:

```
766 <Funciones para Net_Graph 725>+≡ ▷765c 767▷
 template <class Net> typename Net::Flow_Type
 random_first_preflow_edge_maximum_flow
 (Net & net, const bool & leave_residual = false)
 {
 typedef DynSetTreap<typename Net::Node*> Active;
 typedef Random_Set <typename Net::Arc*> Random_Queue;
 return generic_preflow_edge_maximum_flow <Net, Active, Random_Queue>
 (net, leave_residual);
 }
```

#### 7.10.13.12 Conclusión sobre algoritmos de preflujo

El estudio de heurísticas de orden de procesamiento de nodos activos es un campo de investigación fértil. A ese tenor es importante destacar algunas vías de exploración para el descubrimiento de mejores algoritmos o su adaptación a clases de problemas particulares.

La primera vía es la asignación inicial de la función de altura. En nuestro estudio sólo hemos considerado la distancia hasta el sumidero, pero podríamos pensar en otros. Por ejemplo, podríamos priorizar que se alcancen de primero aquellos nodos cuyos grados de salida sean bastantes grandes de modo que éstos tengan la mayor posibilidad de devenir inactivos en su primera observación; también podría considerarse la capacidad total de entrada, es decir, dar prioridad a aquellos nodos que en conjunto pueden enviar y recibir mayores flujos.

Otro camino de exploración lo conforma el diseño de estructura de datos altamente eficientes para manejar los conjuntos de arcos elegibles y nodos activos. Como vimos, el algoritmo genérico orientado hacia arcos (§ 7.10.13.11 (Pág. 761)) es más “maleable” en cuanto al orden de exploración de la red, pero el algoritmo desperdicia unos cuantos ciclos debido a extracciones de arcos no elegibles. Una estructura de datos que además de insertar en  $\mathcal{O}(1)$  también permita eliminar arcos no elegibles luego de un incremento aumentaría la eficiencia.

La tercera vía, por supuesto, consiste en encontrar mejores algoritmos. A ese tenor, una variante de empuje de preflujo se llama “por bloques”. La idea consiste en separar la red en bloques (subgrafos) por donde se lleve el preflujo. Por ejemplo, el grafo ejemplar de la figura 7.73 podríamos descomponerlo en los bloques conformados por  $\{D, E, I, K\}$ ,  $\{B, F, H\}$  y  $\{C, G, J, L\}$ . Esta idea, aparte de que puede hacer más rápido el cálculo, es “naturalmente” paralelizable, es decir, los bloques pueden procesarse en paralelo por varios procesadores según el hardware.

#### 7.10.14 Cálculo del corte mínimo

Tal vez es momento de conversar sobre aplicaciones concretas que nos hagan más aprensibles los conocimientos. Consideremos, por ejemplo, el flujo automotor de una ciudad. Es bien posible obtener un modelo cuyo fin sea maximizar el flujo de automóviles durante horas críticas entre un extremo fuente y otro sumidero. En ese sentido, el flujo máximo sería muy útil para diseñar una política de dirección de tránsito: programación de los semáforos, de desvíos, de vigilancia de vías críticas, etcétera, tendiente a descongestionar

el tráfico. Con mucho embargo, en esta “vida postmoderna” se encuentran numerosísimos ejemplos en que a pesar de conocer el flujo máximo y de emprender correctamente políticas para preservarlo, el flujo automotriz es alienante para la ciudadanía. En estos casos “podría” plantearse aumentar la capacidad vial. ¿Cuáles son los lugares de la ciudad en donde debe aumentarse la capacidad? La respuesta comienza por el corte mínimo. En efecto, el corte mínimo representa un conjunto de arcos que si se redujesen menoscabaría el flujo de la red. Se trata de vías críticas, de lo que a menudo llamamos “cuello de botella”.

Hay muchas más circunstancias de la postmoderna “vida real” en las cuales el problema del flujo máximo y su dual corte mínimo pueden ser de crítica utilidad, desde una red de agua citadina, a una red ferroviaria o de transporte en períodos de fuerte demanda o a una red de transmisión eléctrica.

Regresemos al asunto formal del corte mínimo tratado en § 7.10.6 (Pág. 718). En aquel entonces descubrimos conocimiento, particularmente el establecido por el corolario 7.1 (pag. 719), que nos permite afinar un algoritmo de cálculo de corte mínimo. En el mismo sentido, la proposición 7.8 (pag. 719) nos ofrece certitud de que por la vía del flujo máximo podemos encontrar el corte mínimo.

A efectos de aprehender la importancia algorítmica de aquellos conocimientos, permitámosnos mirar un poco la complejidad de encontrar a fuerza bruta el corte mínimo. Básicamente tendríamos que determinar las combinaciones posibles de los conjuntos  $V_s$  y  $V_t$  junto con sus arcos de cruce. Un algoritmo basado en este principio requeriría, aproximadamente, examinar  $\sum_{i=0}^{V-2} \binom{V-2}{i}$  combinaciones de  $V_s$  y  $V_t$  posibles. Por cada uno de ellos, habría que recorrer los  $E$  arcos para determinar el corte. Se requerirían entonces  $\sum_{i=0}^{V-2} \binom{V-2}{i} \times E$  iteraciones; cantidad combinatorialmente explosiva e intratable a partir de escalas medianas. Aprehendemos pues que el estimado sobre los arcos de cruce que nos proporciona el corolario 7.1 (pág. 719) nos reduce muy significativamente la dificultad del problema; basta con construir el conjunto de arcos saturados o plenos, es decir, aquellos cuyo flujo sea igual a su capacidad, y cualquier combinación entre estos arcos cuya suma sea igual al máximo valor de flujo con arcos de retroceso con flujo cero es un corte mínimo. Empero aun con este conocimiento, el problema sigue siendo algo complejo.

¿Puede simplificarse el cálculo del corte? La proposición 7.10 (pág. 726), nos asegura que si no existe un camino de aumento, entonces el flujo es máximo. Pero, ¿qué subyace tras este conocimiento que nos permita calcular rápida y certamente el corte mínimo? La respuesta puede plantearse como corolario de la proposición 7.10 (pag. 726): puesto que no existe camino de aumento, entonces es imposible llegar al sumidero. Por tanto,  $V_s$  es el conjunto de nodos alcanzables desde  $s$  y  $V_t$  es  $V - V_s$ . Un simple recorrido en profundidad sobre la red residual resultante luego de maximizar el flujo, nos puede descubrir el conjunto  $V_s$ .

A partir del hecho anterior podemos en una primera instancia diseñar una función de visita para el recorrido en profundidad de la red residual que nos añada al conjunto `vs_ptr` los nodos pertenecientes a  $V_s$

767

*<Funciones para Net\_Graph 725>+≡*

▷766 768c▷

```
template <class Net> static bool
add_vertex_to_vs(Net &, typename Net::Node * node, typename Net::Arc*)
{
 ((Aleph::set<typename Net::Node*>*)vs_ptr)->insert(node);
 return false; // indica que debe seguir explorando
}
```

`vs_ptr` es un puntero global al conjunto  $V_s$ .

De este modo, podemos plantear el cálculo de  $V_s$  así:

768a  $\langle \text{Calcular } V_s \text{ 768a} \rangle \equiv$  (769)  
`vs_ptr = &vs;  
 depth_first_traversal <Net, Res_F<Net> >  
 (net, source, &add_vertex_to_vs);`

Luego del recorrido, `vs` contiene todo el conjunto  $V_s$ . El conjunto  $V_t$  podría calcularse por inspección en  $V_s$ , pero una vía más expedita consiste el mirar el bit `Depth_First` que marcó la llamada a `depth_first_traversal()`; si nodo no está marcado, entonces éste pertenece a  $V_t$ . El cálculo de  $V_t$  se plantea del siguiente modo:

768b  $\langle \text{Calcular } V_t \text{ 768b} \rangle \equiv$  (769)  
`const size_t size_vt = net.get_num_nodes() - vs.size();  
 for (Node_Iterator<Net> it(net); it.has_current() and  
 vt.size() < size_vt; it.next())  
 {  
 typename Net::Node * p = it.get_current();  
 if (not IS_NODE_VISITED(p, Aleph::Depth_First))  
 vt.insert(p);  
 }`

Luego de conocidos `vs` y `vt` lo que falta es conocer los respectivos arcos de cruce. Para eso recorremos todos los arcos que no sean residuales y seleccionamos aquellos que satisfagan las condiciones del corolario 7.1 (pag. 719). Esto requiere filtrar los arcos residuales:

768c  $\langle \text{Funciones para Net_Graph 725} \rangle + \equiv$  <767 769>  
`template <class N> class No_Res_Arc  
{  
 bool operator () (N&, typename N::Arc * a) const  
 {  
 return not a->is_residual;  
 }  
};`

Con este filtro inspeccionamos los arcos de la red residual y procesamos aquellos con  $f(e) = 0$  y  $f(e) = \text{cap}(e)$ :

768d  $\langle \text{Calcular arcos de cruce del corte mínimo 768d} \rangle \equiv$  (769)  
`for (Arc_Iterator<Net, No_Res_Arc<Net> > it(net); it.has_current(); it.next())  
{  
 typename Net::Arc * arc = it.get_current();  
 if (arc->flow == 0) // ¿candidato a arco de retroceso?  
 if (vt.count(net.get_src_node(arc)) > 0 and // ¿de vt hacia vs?  
 vs.count(net.get_tgt_node(arc)) > 0)  
 {  
 cutt.insert(arc);  
 continue;  
 }  
 if (arc->flow == arc->cap) // ¿candidato a arco de cruce?  
 if (vs.count(net.get_src_node(arc)) > 0 and // ¿de vs hacia vt?  
 vt.count(net.get_tgt_node(arc)) > 0)  
 cuts.insert(arc);  
}`

Los bloques desarrollados se juntan secuencialmente para hacer el cálculo del corte mínimo:

```
769 <Funciones para Net_Graph 725>+≡ ◁768c 770a▷
 template <class Net, template <class> class Maxflow>
 typename Net::Flow_Type min_cut(Net &
 net,
 Aleph::set<typename Net::Node*> & vs,
 Aleph::set<typename Net::Node*> & vt,
 DynDlist<typename Net::Arc *> & cuts,
 DynDlist<typename Net::Arc *> & cutt,
 const bool leave_residual = false)
 {
 Maxflow <Net> () (net, true); // calcula flujo máximo y red residual

 typename Net::Node * source = net.get_source();

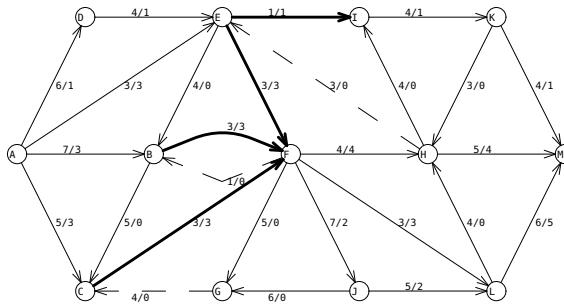
 <Calcular V_s 768a>
 <Calcular V_t 768b>

 <Calcular arcos de cruce del corte mínimo 768d>

 if (not leave_residual)
 {
 net.unmake_residual_net();
 net.unmake_super_nodes();
 }
 return source->out_flow;
 }
```

Uses DynDlist 85a.

`min_cut()` parametriza el tipo de red a la cual se le desea calcular el flujo máximo y el corte mínimo y el algoritmo de cálculo de flujo máximo que se desea emplear. Los parámetros de la función son la red `net`, el conjunto  $V_s$  `vs`, el conjunto  $V_t$  `vt`, el conjunto de arcos  $\langle V_s, V_t \rangle$  `cuts` y el conjunto de arcos  $\langle \overline{V_s}, \overline{V_t} \rangle$  `cutt`.



### 7.10.15 Aumento o disminución de flujo de una red

En las redes reales no siempre se requiere maximizar el flujo. Dentro de las circunstancias es posible, como requerimiento, un ajuste del flujo a un valor dado.

Consideremos un flujo de petróleo por una red de oleoductos que comprende desde los pozos hasta los tanques llenaderos. Nos interesa esta clase de flujo porque no se debe detener<sup>29</sup>. En este caso, según la demanda en los llenaderos, puede ser necesario mantener el flujo a un valor específico. ¿Cómo incrementar o decrementar el flujo en un valor dado?

En esta situación cobra mucho sentido la idea de camino de aumento, pues su eslabón mínimo determina una cota de incremento (o decremento) del flujo por el camino en cuestión. Sabemos, por la proposición 7.9 (Pág. 722), que si tenemos un camino de aumento  $C_a$  con mínimo eslabón es  $\Delta_{C_a}$ , entonces el flujo puede aumentarse en un valor  $x \leq \Delta_{C_a}$ . Para incrementar o decrementar el flujo en un valor  $x$  debemos encontrar un camino con eslabón mínimo  $\Delta_{C_a} \geq x$ .

Encontrar un camino de aumento condicionado a que su eslabón mínimo sea mayor o igual a un  $x$  es sencillo si empleamos un filtro para el iterador sobre los arcos que ignore los arcos cuyo flujo restante es menor que  $x$ .

Para eso definimos la siguiente clase filtro del iterador de arcos:

```
770a <Funciones para Net_Graph 725>+≡ ◁769 770b▷
 template <class N> class Flow_Filter
 {
 static typename N::Flow_Type min;
 typename N::Arc * operator () (typename N::Node *, typename N::Arc * arc)
 {
 return arc->cap - arc->flow >= min ? arc : NULL;
 }
 };

```

La clase `Flow_Filter<N>` filtra arcos cuyo flujo restante es menor que el valor de `min`<sup>30</sup>. De este modo podemos programar la búsqueda de un camino de aumento, con umbral `min`, de la siguiente manera:

```
770b <Funciones para Net_Graph 725>+≡ ◁770a 771a▷
 template <class Net, template <class, class> class Find_Path>
 bool find_aumenting_path(Net & net, Path<Net> & path,
 const typename Net::Flow_Type & min_slack)
 {
 Flow_Filter<Net>::min = min_slack;
 typename Net::Node * src = net.get_source();
 typename Net::Node * sink = net.get_sink();

 if (not net.residual_net)
 net.make_residual_net();

 return Find_Path<Net, Flow_Filter<Net> > () (net, src, sink, path);
 }
```

Uses Path 578a.

`find_aumenting_path()` busca un camino de aumento con eslabón mínimo suficiente para aumentar el flujo en el valor `min_slack`. El tipo de búsqueda (en profundidad o en

<sup>29</sup>El petróleo, cuando se encuentra inmóvil, tiende a solidificarse.

<sup>30</sup>Un problema del diseño de esta clase es que no es reentrante. Durante el tiempo en que un algoritmo está empleando `Flow_Filter`, la variable `min` no puede ser utilizada. Futuras versiones de *ALÉPH* manejarán este parámetro de forma reentrante.

amplitud) es un parámetro tipo de la rutina. Si el camino existe, entonces la rutina retorna true; de lo contrario, retorna false.

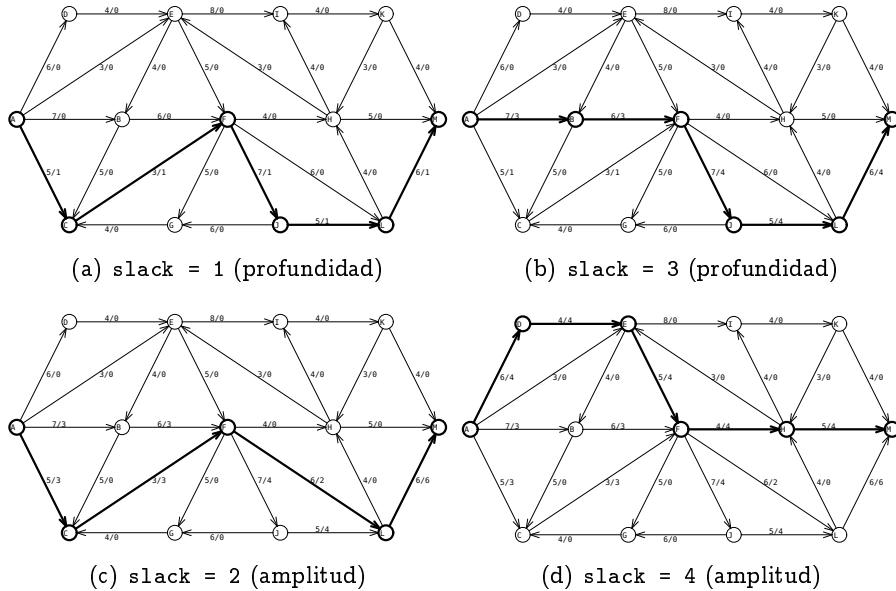


Figura 7.92: Incrementos sucesivos del flujo mediante caminos de aumento encontrados por `find_aumenting_path()`

¿Qué sucede si se desea encontrar un camino de “des-aumento”?, o sea, uno por el cual, en lugar de aumentar el flujo en un valor dado, se disminuya. La técnica es similar, con la sutil diferencia de que la búsqueda debe hacerse desde el sumidero hacia el fuente:

771a

<*Funciones para Net\_Graph* 725>+≡

△ 770b 771b ▷

```
template <class Net, template <class, class> class Find_Path>
bool find_decrementing_path(Net & net, Path<Net> & path,
 const typename Net::Flow_Type & min_slack)
{
 Flow_Filter<Net>::min = min_slack;
 typename Net::Node * src = net.get_source();
 typename Net::Node * sink = net.get_sink();

 if (not net.residual_net)
 net.make_residual_net();

 return Find_Path<Net, Flow_Filter<Net> > () (net, sink, src, path);
}
```

Uses Path 578a.

Para incrementar el flujo se emplea una variante de la rutina `increase_flow()`, desarrollada en § 7.10.10 (Pág. 725), con un parámetro adicional `val`. Si se desea, por ejemplo, aumentar el flujo en un valor `val` a través de un camino de aumento, entonces se invoca a `increase_flow(net, path, val)`, la cual incrementa el flujo en `val` y no en el eslabón mínimo de `path`. Análogamente, otra rutina, llamada `decrease_flow()`, hace el trabajo inverso, o sea, decrementar el flujo en un valor `val`.

771b

<*Funciones para Net\_Graph* 725>+≡

△ 771a ▷

```
template <class Net>
```

```

void increase_flow(Net & net, Path<Net> & path,
 const typename Net::Flow_Type & val)
{
 // aumentar flujo de red por el camino de aumento
 for (typename Path<Net>::Iterator it(path); it.has_current_arc(); it.next())
 {
 typename Net::Arc * arc = it.get_current_arc();
 typename Net::Arc * img = (typename Net::Arc *) arc->img_arc;
 if (arc->cap - arc->flow < val)
 // ERROR!

 arc->flow += val;
 img->flow -= val;
 if (not arc->is_residual)
 {
 net.increase_out_flow(net.get_src_node(arc), val);
 net.increase_in_flow(net.get_tgt_node(arc), val);
 }
 else
 {
 net.decrease_in_flow(net.get_src_node(arc), val);
 net.decrease_out_flow(net.get_tgt_node(arc), val);
 }
 }
}

template <class Net>
void decrease_flow(Net & net, Path<Net> & path,
 const typename Net::Flow_Type & val)
{
 // decrementar flujo de red por camino de decremiento
 for (typename Path<Net>::Iterator it(path); it.has_current_arc(); it.next())
 {
 typename Net::Arc * arc = it.get_current_arc();
 typename Net::Arc * img = (typename Net::Arc *) arc->img_arc;

 if (arc->cap - arc->flow < val)
 // ERROR !

 arc->flow -= val;
 img->flow += val;
 if (not arc->is_residual)
 {
 net.decrease_out_flow(get_src_node(arc), val);
 net.decrease_in_flow(net.get_tgt_node(arc), val);
 }
 else
 {
 net.increase_out_flow(get_tgt_node(arc), val);
 net.increase_in_flow(net.get_src_node(arc), val);
 }
 }
}

```

}

Uses Path 578a.

Es importante mencionar que la rutina no verifica que se trate estrictamente de un camino de aumento. Esto ofrece la posibilidad de incrementar el flujo por un camino que no necesariamente parte desde el fuente y llegue al sumidero. Del mismo modo, el camino puede ser un ciclo, lo que cobrará interés más adelante cuando estudiemos circulaciones (§ 7.11.5 (Pág. 783)) y flujos máximos de coste mínimo (§ 7.12 (Pág. 792)).

## 7.11 Reducciones al problema del flujo máximo

En la sección pasada estudiamos una serie de algoritmos y estructura de datos tendientes a resolver el problema del flujo máximo. Resulta que este problema que, diríamos, no nos fue muy simple de resolver, es fundamento de modelización e instrumento de solución de una vasta cantidad de problemas más. En esta sección mencionaremos algunas situaciones del mundo tecnológico en que el flujo máximo es aplicable, enunciaremos variantes de problemas reducibles al flujo máximo y estudiaremos con algún grado de detalle algunos de ellos.

La literatura menciona como primera variante el manejo de varios fuentes y sumideros. Este problema fue estudiado y resuelto en § 7.10.3 (Pág. 711). ¿En cuáles contextos puede aparecer? En los contextos culturales, por decirlo de alguna manera, es muy probable que la red tenga varios fuentes y sumideros, así que por esta vía nos resultan prácticas las primitivas `make_super_nodes()` y `unmake_super_nodes()` presentadas en § 7.10.3 (Pág. 711).

### 7.11.1 Flujo máximo en redes no dirigidas

La idea de un líquido que fluye por una red de tuberías no asume el sentido. De hecho, por un tubo puede circular fluido en cualquier sentido. Así pues, está dentro de las expectativas del mundo tecnológico encontrar situaciones en las cuales se deseé maximizar el flujo y conocer sus tuberías críticas; o sea, el corte mínimo. En situaciones de este tipo tendríamos arcos por los cuales es permisible el flujo en ambos sentidos.

Una red no dirigida  $G = \langle V, E, C \rangle$ , con pesos interpretables como capacidades, tiene un equivalente dirigido directo  $\langle V, E', C \rangle$ : por cada arco  $(u, v) = e \in E$  de capacidad  $\text{cap}(e)$  existen dos arcos  $e_u = (u, v) \in E'$  y  $e_v = (v, u) \in E'$  ambos con capacidad  $\text{cap}(e_u) = \text{cap}(e_v) = \text{cap}(e)$ .

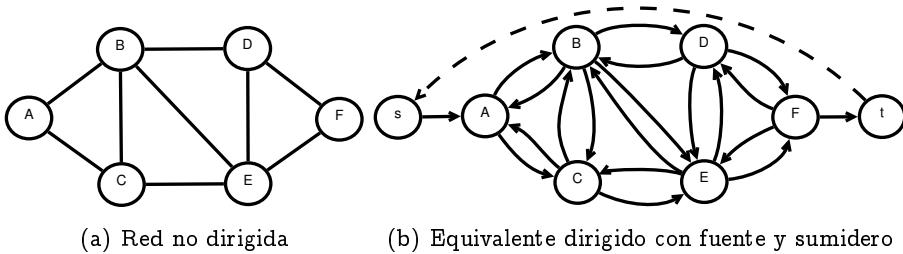


Figura 7.93: Equivalencia entre una red no dirigida y una red

Un efecto “aparente” de la transformación anterior es que la red no tiene fuente ni sumidero. En una red de tuberías, este podría parecer ser el caso. Por ejemplo, consideremos un sistema de refrigeración en el cual es menester maximizar el flujo de líquido refrigerante, un sistema de aire acondicionado o de calefacción por citar ocurrencias particulares. En toda clase de sistemas de flujo se requiere bombear el fluido desde al menos un tubo inicial. Para que el fluido sea continuo se requiere cerrar el lazo de bombeo. Aparecen pues el fuente y el sumidero. La figura 7.93 ilustra una red no dirigida y su equivalente en red capacitada con un fuente  $s$  desde donde se enviaría un flujo inicial, y un sumidero por donde se recogería el flujo para bombearlo de nuevo. Los valores de capacidades de los arcos desde el fuente y hacia el sumidero deben corresponderse con la capacidad de flujo disponible, la cual puede ser mayor que el valor de flujo máximo que puede circular por la red.

La validez matemática del equivalente es fácil de demostrar. Cualquier flujo circulante por la red no dirigida puede perfectamente circular por la red equivalente; en ese caso uno de los dos arcos en la red dirigida tiene valor de flujo cero. Devolverse de la red al grafo no es problemático. Supongamos cualquier par de arcos  $u \rightarrow v$  y  $v \rightarrow u$  con valores de flujo  $f(u \rightarrow v)$  y  $f(v \rightarrow u)$  respectivamente. Si  $f(u \rightarrow v) > f(v \rightarrow u)$ , entonces  $f(u \leftrightarrow v) = f(u \rightarrow v) - f(v \rightarrow u)$  y el fluido va en el sentido  $u \rightarrow v$ ; de lo contrario,  $f(u \leftrightarrow v) = f(v \rightarrow u) - f(u \rightarrow v)$  y el fluido va en el sentido  $v \rightarrow u$ .

Puesto que estas reducciones son factibles mediante las interfaces de los TAD List\_Graph y Net\_Graph, éstas son delegadas en ejercicios.

### 7.11.2 Capacidades en nodos

Una situación tecnológicamente esperable, pero también útil para algunas situaciones de modelización, es el considerar capacidades en los nodos. En este caso, un valor de capacidad de nodo indica un flujo máximo que éste puede recibir. La figura 7.94-a muestra un ejemplo.

Una red con capacidades en los nodos puede transformarse a un red tradicional mediante el método siguiente.

#### Algoritmo 7.7 Conversión de una red con capacidades en los nodos a una red tradicional

La entrada es una red  $\vec{G} = \langle V, E, C, s, t \rangle$  con capacidades en los nodos.

La salida es una red tradicional  $N^{\vec{G}} = \langle V', E', C', s, t \rangle$ .

1.  $\forall u \in V$  particione en dos nodos  $u'$  y  $u''$  |  $u', u'' \in V'$ .

Por cada partición, añada a  $E'$  un arco  $u' \rightarrow u'' \in E'$  con capacidad  $\text{cap}(u' \rightarrow u'') = \text{cap}(u)$ .

2.  $\forall e = (u \rightarrow v) \in E$  se corresponde con  $e' = (u'' \rightarrow v') \in E'$ .

La figura 7.94-b muestra el equivalente según el método a la red de la figura 7.94-a.

La biblioteca contiene una implementación de este esquema llamada, Net\_Cap\_Graph, contenida en el archivo tpl\_netcapgraph.H.

### 7.11.3 Flujo factible

En algunas circunstancias se requieren provisiones de flujo; es decir, algunos nodos de la red requieren recibir una cantidad específica de flujo. Es posible también que otros nodos

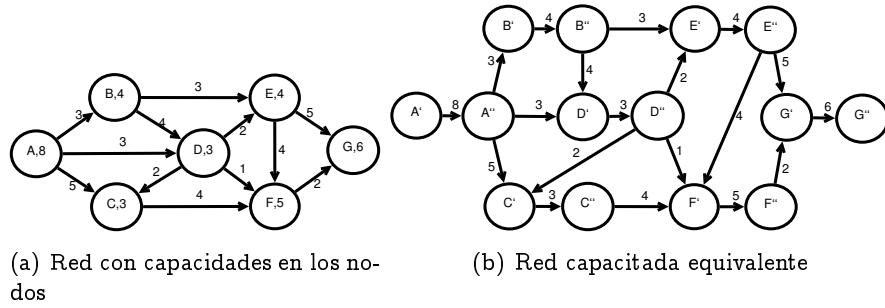


Figura 7.94: Ejemplo de una red con capacidad en los nodos y su equivalente a un red tradicional

provean flujo. Supongamos por ejemplo una red de transporte vial y la realización de evento excepcional, tal como unos juegos deportivos, que cambie las demandas. Como la red vial sigue siendo la misma, la topología de la red no cambia, pero el interés de asistencia y dispersión geográfica de los lugares de las competiciones transforma las demandas. En situaciones de este tipo suele aumentarse la capacidad de transporte trayendo, por ejemplo, buses adicionales.

Situaciones como la que acabamos de describir pueden modelizarse poniendo sobre los nodos valores de demanda y de provisión. Un valor positivo indica que el nodo puede suprir en flujo el valor en cuestión; un valor negativo significa una demanda de flujo, es decir, en el nodo en cuestión se requiere recibir el flujo demandado. A esta clase de red se le llama “red capacitada con provisión de flujo”.

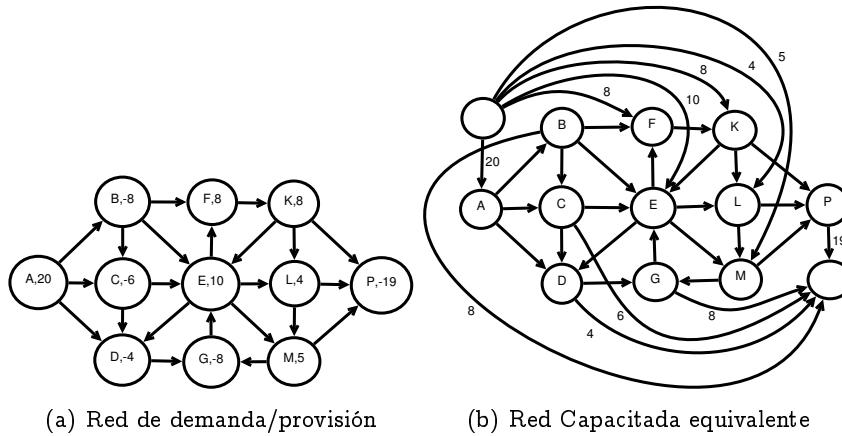


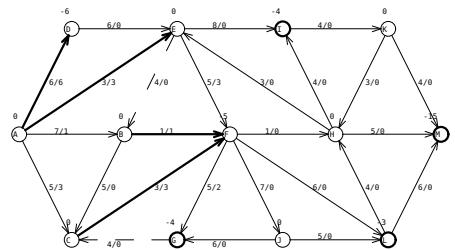
Figura 7.95: Ejemplo de equivalencia entre red de demanda/provisión y una red capacitada. Se omiten las capacidades de los arcos en la red original y en la equivalente; esta última muestra capacidades equivalentes a provisiones

Modelos de esta clase no son para nada excepcionales. En estas circunstancias, la pregunta de interés es averiguar si se puede o no satisfacer las demandas en función de las posibilidades de provisión. Una respuesta negativa, mostrativa, indicará que hay que encontrar más provisión, mientras que una afirmativa nos dirá que con la provisión e infraestructura dadas se pueden satisfacer las demandas. Si la respuesta es afirmativa,

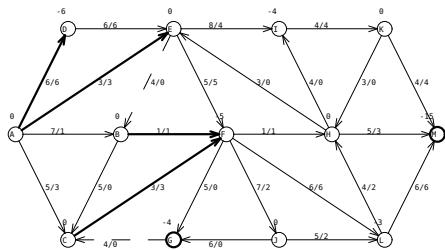
entonces decimos que existe un flujo factible, pues todos los nodos de demanda pueden obtener su flujo requerido; de lo contrario decimos que el flujo no es factible.

¿Cómo saber si existe un flujo factible? Sin sorpresa, la respuesta nos la da el cálculo del flujo máximo sobre una red equivalente. En principio podemos calcular el flujo máximo y luego verificar si éste satisface las provisiones dadas. Sin embargo, si encontramos nodos con fallas de provisiones, particularmente en demandas, no podemos concluir que no existe otro flujo menor que sea factible. La solución es transformar la red a una equivalente y sobre ella calcular el flujo máximo.

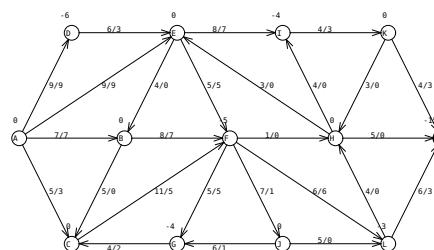
Una red capacitada con provisión de flujo tiene un equivalente en una red capacitada. La equivalencia requiere añadir un suprafuente y un suprasumidero. Desde el suprafuente emanan arcos hacia los nodos proveedores con capacidad igual al valor de provisión. Hacia el suprasumidero inciden arcos desde los nodos demandantes con capacidad igual al flujo demandado. En la red equivalente no es necesario colocar valores de provisión, pues éstos están expresados en las capacidades de los arcos entre suprafuente y el suprasumidero respectivamente.



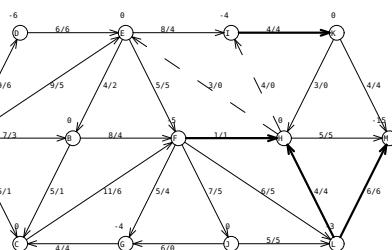
(a) Flujo máximo sobre red extendida



(b) Flujo máximo



(c) Flujo máximo sobre red extendida



(d) Flujo máximo

Figura 7.96: Dos ejemplos de flujo factible sobre demandas. La figura (a) muestra un flujo máximo con corte mínimo sobre la red extendida que no satisface las provisiones; luego en la figura (b), el flujo máximo la red que muestra un flujo máximo insatisfactorio. Del análisis del corte mínimo se incrementan capacidades en los arcos del corte, lo cual conduce a un flujo factible, tanto en la red extendida mostrada en la figura (c) como en la red simple mostrada en la figura (d).

Una red con provisión de flujo es factible si el flujo máximo de la red equivalente logra llenar todos los arcos añadidos, es decir, los que emanan desde el suprafuente y lo que llegan al suprasumidero. En algunos casos es posible saber que el flujo no es factible si la capacidad saliente del suprafuente es distinta a la entrante por el suprasumidero. Esto es debido a la condición de conservación del flujo y es el caso para la red ejemplo de la figura 7.95-a, la cual, para la red equivalente de la figura 7.95-b tiene una capacidad saliente del suprafuente de 55 mientras que la entrante al suprasumidero es de 45. Este conocimiento

es visible desde la red original en el sentido de que para que el flujo sea factible, el total de provisión tiene que ser mayor o igual a la demanda, pero lo contrario no necesariamente es cierto; por eso, si la provisión es mayor o igual a la demanda, entonces es necesario calcular el flujo máximo sobre la red equivalente para responder si el flujo es o no factible.

En una situación de flujo no factible, el corte mínimo nos identifica otros puntos por donde puede aumentarse la provisión o, en su imposibilidad, reducirse o redireccionarse la demanda.

A tenor del flujo factible, en la biblioteca se instrumenta el `Net_Sup_Dem_Node`, especificado en el archivo `tpl_net_sup_dem.H`. Sin embargo, a veces es preferible aplicar la técnica en el contexto del problema en lugar de emplear este TAD.

#### 7.11.4 Máximo emparejamiento bipartido

Consideremos  $n$  personas y  $n$  tipos de trabajos. Para cada persona se tiene una lista de posibles trabajos o, análogamente, por cada trabajo se tiene una lista de personas que pueden hacerlo. ¿Existe una asignación de personas a trabajos que cubra a todos los trabajos y personas? Si existe, ¿cuál es?; si no existe, ¿cuál es la asignación que cubre la mayor cantidad de personas o trabajos?

Este problema, tradicionalmente conocido como el “de asignación”, constituye la variante más simple de una familia de problemas más general que trataremos más adelante. Por los momentos, es menester introducir algunas definiciones formales.

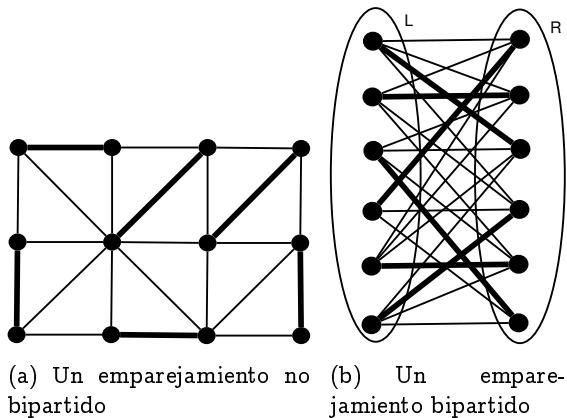


Figura 7.97: Ejemplos de emparejamiento. Los arcos del emparejamiento son resaltados

**Definición 7.20 (Emparejamiento)** Dado un grafo  $G = \langle V, E \rangle$ , un emparejamiento es un subconjunto de arcos  $M \subset E$  tal que no existen dos arcos en  $M$  que incidan sobre un nodo común.

Un emparejamiento es de cardinalidad máxima si  $|M|$  es máximo, es decir, si incluye la mayor cantidad de arcos.

Un emparejamiento es bipartido si el grafo es bipartido.

Un emparejamiento es perfecto si incluye a todos los nodos del grafo.

Cuando tenemos un grafo con pesos, un emparejamiento es máximo si la suma de los pesos de los arcos de  $M$  es máxima. Simétricamente, es mínimo si la suma es mínima.

Asignar  $n$  personas a  $n$  trabajos puede resolverse mediante un grafo bipartido que modelice las relaciones entre las personas y los posibles trabajos tal como el mostrado en la figura 7.97-b. El emparejamiento de cardinalidad máxima nos proporciona la asignación que más cubre personas y trabajos.

Puesto que atacar “cándidamente” el problema de la asignación, o sea, explorar todas las combinaciones posibles de emparejamiento, deviene intratable, quizás esta sea la primera situación en la que podamos apreciar el potencial de la teoría de redes de flujo para la resolución de problemas sobre grafos.

Un grafo bipartido sobre el cual deseamos encontrar un emparejamiento de cardinalidad máxima puede modelizarse mediante una red de flujo. *Grosso modo*, el procedimiento es el siguiente.

#### Algoritmo 7.8 (Transformación de un grafo bipartido a una red de flujo)

Entrada: un grafo bipartido  $G = \langle V, E \rangle$  |  $V = L \cup R$  y  $L \cap R = \emptyset$  ( $L$  y  $R$  son los conjuntos de la bipartición).

Salida: una red capacitada  $\overrightarrow{G} = \langle V', E', C, s, t \rangle$  equivalente sobre la cual puede calcularse un emparejamiento de cardinalidad máxima.

1.  $V' = V \cup \{s\} \cup \{t\}$ .
2.  $\forall e = (u, v) \in E \implies E' = E' \cup \{(u, v)\} \mid \text{cap}(e) = 1$ .
3.  $\forall u \in L \implies E' = E' \cup \{(s, u)\} \mid \text{cap}((s, u)) = 1$ .
4.  $\forall u \in R \implies E' = E' \cup \{(u, t)\} \mid \text{cap}((u, t)) = 1$ .

La figura 7.98 ilustra el resultado final de ejecución de este algoritmo sobre el grafo de la figura 7.97-b.

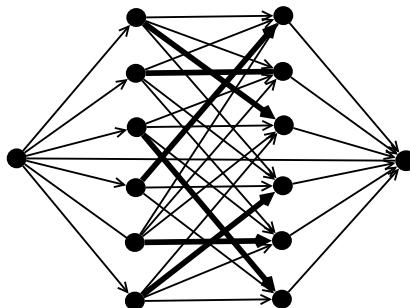


Figura 7.98: Grafo dirigido equivalente al de la figura 7.97-b

#### 7.11.4.1 Flujo máximo y emparejamiento bipartido

La transformación anterior nos ofrece una red capacitada cuyo flujo máximo nos muestra en sus arcos saturados un emparejamiento de cardinalidad máxima.

Todos los arcos de la transformación tienen capacidad 1. Todo nodo en  $L$  tiene capacidad de entrada 1 y capacidad de salida igual al número de arcos que se conectan con  $R$ . Simétricamente, todo nodo en  $R$  tiene capacidad de entrada igual al número de arcos de entrada y capacidad de salida 1. Claramente, el máximo flujo de esta red es

igual  $|L|$  ó a  $|R|$ , o sea, al número de arcos de salida del fuente o el de entrada del sumidero. Un camino de aumento cualquiera, cuyo eslabón mínimo siempre es 1, descubre una asignación  $(u, v) \mid u \in L, v \in R$ . Cuando el camino de aumento se incrementa, los nodos  $u$  y  $v$  quedan inconexos en la red residual. El nodo  $u$  es inaccesible desde el fuente  $s$  y  $t$  es inaccesible desde  $v$ . Consecuentemente, cualquier próximo camino de aumento no contendrá a  $u$  ni a  $v$ . Una vez que no existen más caminos de aumento, el flujo es máximo (proposición § 7.10 (Pág. 726)). Puesto que el flujo es máximo y los arcos tienen capacidades unitarias, la cantidad de arcos con flujo unitario que van desde  $L$  hacia  $R$  es máxima. El emparejamiento de cardinalidad máxima está dado por aquellos arcos entre  $L$  y  $R$  cuyo flujo sea unitario.

#### 7.11.4.2 Análisis del emparejamiento bipartido por flujo máximo

La duración del cálculo del emparejamiento bipartido es la suma de duraciones del algoritmo 7.8, luego, la maximización del flujo y, finalmente, el reconocimiento de los arcos que pertenecen al emparejamiento.

Aplicar el algoritmo 7.8 sobre un grafo bipartido para obtener la red equivalente sobre la cual maximizar el flujo toma  $\mathcal{O}(E)$  pasos; la misma complejidad se requiere para reconocer los arcos del emparejamiento luego de maximizado el flujo.

En cuanto a la duración de la maximización de flujo debemos observar que las duraciones de los algoritmos estudiados fueron deducidas para los peores casos, considerando topologías arbitrarias, lo cual no es la situación de un grafo bipartido, menos aún la de la red equivalente.

Puesto que la red transformada sólo tiene capacidades unitarias, cualquier camino de aumento y su incremento corta un arco de salida del fuente y uno del sumidero; aquí el algoritmo de Ford-Fulkerson no tiene el problema de la integridad de las capacidades. Por tanto, la máxima cantidad de caminos de aumento se presenta cuando la cantidad de arcos desde el fuente y hacia el sumidero es máxima. Tal cantidad es  $V/2$ . Maximizar el flujo de la red transformada toma  $\frac{V}{2}\mathcal{O}(E) = \mathcal{O}(VE)$  pasos de ejecución.

#### 7.11.4.3 Cálculo de la bipartición

Si tenemos un grafo  $G$  bipartido, un paso esencial para la instrumentación de nuestro algoritmo es calcular la bipartición. Como este problema puede verse por separado, merece un tratamiento general que lo desarrollaremos en el archivo de la biblioteca `tpl_bipartite.H`, el cual contiene rutinas utilitarias concernientes a grafos bipartidos, específicamente la prueba de bipartición `test_bipartite()` y su cálculo `compute_bipartite()`.

Hay varias maneras de verificar el carácter bipartido de un grafo. Si un grafo es bipartido, entonces éste no tiene ningún ciclo de longitud impar. Para demostrarlo consideremos un ciclo de longitud impar. Algo así como  $u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5 \rightarrow u_1$ . Si seguimos el ciclo y alternativamente asignamos las particiones podemos hacer  $L = \{u_1, u_3, u_5\}$  e  $R = \{u_2, u_4\}$ , pero el arco  $u_5 \rightarrow u_1$  rompe la bipartición. Igual ocurre si comenzamos desde cualquier otro nodo del ciclo. Si generalizamos este contraejemplo para cualquier longitud impar nos encontramos con la misma contradicción.

Otra forma menos clara pero computacionalmente más eficiente, es verificar si el grafo es bicoloreable, es decir, si puede pintarse con sólo dos colores sin que el mismo color

colinde. El siguiente algoritmo utiliza el coloreado para construir la bipartición:

```

780 <Rutinas para grafos bipartidos 780>≡ 782b▷
 template <class GT, class SA = Default_Show_Arc<GT> >
 void compute_bipartite(GT & g,
 DynDlist<typename GT::Node *> & l,
 DynDlist<typename GT::Node *> & r)
 {
 DynDlist<typename GT::Node *> red, blue;
 typename GT::Node * p = g.get_first_node();
 color<GT>(p) = Bp_Red;
 red.put(p); l.put(p);

 while (true)
 if (not red.is_empty()) // ¿Hay rojo con arcos no mirados?
 {
 typename GT::Node * p = red.get();
 for (Node_Arc_Iterator<GT, SA> it(p); it.has_current(); it.next())
 {
 typename GT::Arc * a = it.get_current();
 if (color<GT>(a) == Bp_Red)
 throw std::domain_error("Graph is not bipartite");
 else if (color<GT>(a) == Bp_Blue)
 continue;

 color<GT>(a) = Bp_Red;
 typename GT::Node * q = it.get_tgt_node();
 if (color<GT>(q) == Bp_Red)
 throw std::domain_error("Graph is not bipartite");
 else if (color<GT>(q) == Bp_Blue)
 continue;

 color<GT>(q) = Bp_Blue;
 blue.put(q); r.put(q);
 }
 }
 else if (not blue.is_empty()) // ¿Hay azul con arcos no mirados?
 {
 typename GT::Node * p = blue.get();
 for (Node_Arc_Iterator<GT, SA> it(p); it.has_current(); it.next())
 {
 typename GT::Arc * a = it.get_current();
 if (color<GT>(a) == Bp_Blue)
 throw std::domain_error("Graph is not bipartite");
 else if (color<GT>(a) == Bp_Red)
 continue;

 color<GT>(a) = Bp_Blue;

 typename GT::Node * q = it.get_tgt_node();
 if (color<GT>(q) == Bp_Blue)
 throw std::domain_error("Graph is not bipartite");
 }
 }
 }

```

```

 else if (color<GT>(q) == Bp_Red)
 continue;

 color<GT>(q) = Bp_Red;
 red.put(q); l.put(q);
 }
}
else
 break;
}

```

Uses DynDlist 85a.

La rutina dispara la excepción std::domain\_error si el grafo no es bipartido. Si lo es, entonces se retornan en las listas l y r los nodos que conforman la partición.

#### 7.11.4.4 Construcción de la red bipartita

Con los conjuntos de la bipartición ya tenemos la capacidad para calcular una red equivalente mapeada al grafo. Llaremos g el grafo bipartido y net la red auxiliar que deseamos transformar, la cual se especifica de la siguiente manera:

781a *(Construir red equivalente 781a)*≡ (782b) 781b▷  
 typedef Net\_Graph<Net\_Node<Empty\_Class>, Net\_Arc<Empty\_Class> > AN;  
 AN net;

El cálculo de net se hace entonces del siguiente modo:

781b *(Construir red equivalente 781a)+*≡ (782b) ▷781a  
 // recorrer nodos de g e insertar imagen en net  
 for (Node\_Iterator<GT> it(g); it.has\_current(); it.next())  
 {  
 typename GT::Node \* p = it.get\_current();  
 NODE\_COOKIE(p) = net.insert\_node();  
 NODE\_COOKIE((typename GT::Node \*)NODE\_COOKIE(p)) = p;  
 }

 typename AN::Node \* source = net.insert\_node();
 // recorrer nodos de l, atarlos a fuente e insertar sus arcos
 for (typename DynDlist<typename GT::Node\*>::Iterator i(l);
 i.has\_current(); i.next())
 {
 typename GT::Node \* p = i.get\_current();
 typename AN::Node \* src = mapped\_node<GT, AN> (p);
 net.insert\_arc(source, src, 1);

 for (Node\_Arc\_Iterator<GT, SA> j(p); j.has\_current(); j.next())
 {
 typename GT::Arc \* arc = j.get\_current\_arc();
 typename AN::Node \* tgt = mapped\_node <GT, AN> (g.get\_tgt\_node(arc));
 typename AN::Arc \* a = net.insert\_arc(src, tgt, 1);
 ARC\_COOKIE(arc) = a;
 ARC\_COOKIE(a) = arc;
 }
 }
}

```

typename AN::Node * sink = net.insert_node();

// recorrer nodos de r y atarlos al sumidero
for (typename DynDlist<typename GT::Node*>::Iterator it(r);
 it.has_current(); it.next())
{
 typename GT::Node * p = it.get_current();
 net.insert_arc(mapped_node<GT, AN> (p), sink, 1);
}

```

Uses DynDlist 85a and mapped\_node 560b.

#### 7.11.4.5 Extracción del emparejamiento de la red maximizada

Si matching es la lista de arcos que conforman el emparejamiento, su determinación se efectúa mediante inspección de la red auxiliar net en búsqueda de los arcos saturados; su imagen en el grafo g pertenece al emparejamiento.

782a *(Extraer de net los arcos saturados 782a)*≡ (782b)

```

for (Arc_Iterator<AN> it(net); it.has_current(); it.next())
{
 typename AN::Arc * a = it.get_current();
 if (a->flow == 0)
 continue;

 typename GT::Arc * arc = mapped_arc <GT> (a);
 if (arc == NULL)
 continue;

 matching.append(arc);
}

```

#### 7.11.4.6 Implantación final del emparejamiento de cardinalidad máxima

Con los bloques anteriores se conforma el algoritmo final:

782b *(Rutinas para grafos bipartidos 780)*+≡ <780

```

template <class GT,
 template <class> class Max_Flow = Ford_Fulkerson_Maximum_Flow,
 class SA = Default_Show_Arc<GT> >
void compute_maximum_cardinality_bipartite_matching
(GT & g, DynDlist<typename GT::Arc *> & matching)
{
 DynDlist<typename GT::Node *> l, r;

 compute_bipartite(g, l, r);

 (Construir red equivalente 781a)

 Max_Flow <AN> () (net);

 (Extraer de net los arcos saturados 782a)
}

```

Uses DynDlist 85a.

### 7.11.5 Circulaciones

¿Puede concebirse una red sin fuentes ni sumideros? Consideremos una red tradicional  $N = \langle V, E, C, s, t \rangle$  y una ligera modificación sobre ella consistente en añadir un arco con capacidad igual a  $\max(\text{cap}(\text{OUT}(s)), \text{cap}(\text{IN}(t)))$ . La idea general es pictORIZADA en la figura 7.99 y evidencia que, al menos en este caso genérico, es concebible una red sin fuentes ni sumideros reducible a una con un fuente y un sumidero.

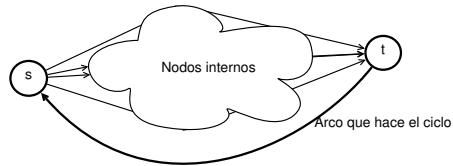


Figura 7.99: Diagrama general de una circulación

A un flujo por una red sin fuentes ni sumideros se le llama “circulación”, pues alegoriza una situación en la cual ni entra ni sale flujo de la red; simplemente, posibilitado por alguna extraña inercia<sup>31</sup>, o a efectos del modelo, el fluido circula establemente por la red sin necesidad de especificar una entrada y salida de flujo.

Notemos que una circulación acarrea al menos un ciclo en un grafo dirigido, situación que explícitamente no hemos estudiado hasta el presente. Los algoritmos de maximización de flujo presentados operan sobre redes que pueden contener ciclos internos, pero con un sólo fuente y un sumidero. ¿Es en general una red sin fuentes ni sumideros reducible a una red tradicional? La respuesta es afirmativa, cualquier red capacitada válida, sin fuente ni sumidero, puede reducirse a una red con un fuente y un sumidero y sobre esta última aplicar los algoritmos estudiados.

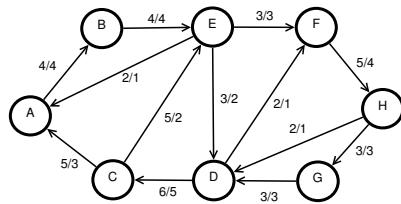


Figura 7.100: Una circulación

Un ciclo puede reducirse a una subred con un fuente y un sumidero, simplemente cortar el ciclo y asumir sus extremos como fuentes y sumideros. La red se descompone desde los ciclos más simples, que no tienen nodos comunes, hacia los más complicados, que tienen nodos comunes con otros ciclos. El grado de salida de cada nodo arroja una primera aproximación, pero hay otra métrica, que presentaremos prontamente, que ofrece mejor información acerca de la conectividad del grafo y que se llama “conectividad en arcos”. El algoritmo de conversión general es delegado en ejercicio.

<sup>31</sup>Físicamente es concebible mantener una circulación por la mera inercia, pero en el mundo real es improbable.

Un aspecto interesante de una circulación es que su valor de flujo puede representarse como un conjunto de flujos circulantes por sus ciclos. Por ejemplo, la circulación de la figura 7.100 puede expresarse mediante los siguientes ciclos:

| Ciclo                                                                                                               | Valor de flujo |
|---------------------------------------------------------------------------------------------------------------------|----------------|
| $A \rightarrow B \rightarrow E \rightarrow A$                                                                       | 1              |
| $A \rightarrow B \rightarrow E \rightarrow F \rightarrow H \rightarrow G \rightarrow D \rightarrow C \rightarrow A$ | 1              |
| $A \rightarrow B \rightarrow E \rightarrow F \rightarrow H \rightarrow D \rightarrow C \rightarrow A$               | 1              |
| $A \rightarrow B \rightarrow E \rightarrow D \rightarrow C \rightarrow A$                                           | 1              |
| $C \rightarrow E \rightarrow D \rightarrow C$                                                                       | 2              |
| $D \rightarrow F \rightarrow H \rightarrow D$                                                                       | 1              |
| $D \rightarrow F \rightarrow H \rightarrow G \rightarrow D$                                                         | 1              |

La interpretación anterior de un flujo evidencia que, para una red sin fuentes ni sumideros, el flujo máximo se reduce a encontrar una circulación que maximice el flujo en cada arco. Podemos sin problema aumentar el flujo de la red descubriendole ciclos, calculándoles su capacidad restante mínima y aumentando el flujo en este valor a lo largo de todo el ciclo.

La tabla de ciclos anterior nos evidencia el sentido de “circulación”. En efecto, cada ciclo es una circulación. En el caso de la tabla encontramos 7 circulaciones. Notemos que si tuviésemos una red con un fuente y un sumidero, atar un arco desde el sumidero hacia el fuente nos da tantas circulaciones como caminos distintos existan desde el fuente hacia el sumidero.

El valor de flujo de la red puede expresarse como una suma de ciclos y flujos. Pero la misma red y su valor de flujo también pueden representarse como una composición de ciclos y flujo, tal como evidencia la proposición siguiente.

**Proposición 7.16 (Descomposición de flujo)** Cualquier circulación puede representarse como un flujo entre un conjunto de a lo más  $|E|$  ciclos distintos.

**Demostración** Consideremos el siguiente algoritmo de descomposición:

**Algoritmo 7.9 (Descomposición de una circulación en ciclos)**

1. Sea  $E^\circ = \emptyset$  el conjunto de ciclos que descomponen la circulación.  $E^\circ$  contiene elementos de la forma  $(e^\circ, \Delta)$ , donde  $e^\circ$  es un ciclo y  $\Delta$  el valor de flujo que circula por el ciclo.
2. while existan arcos sobre  $N$ 
  - (a) Encuentre un ciclo  $e^\circ$ .
  - (b) Determine el eslabón mínimo del ciclo  $\Delta$ .
  - (c)  $E^\circ = E^\circ \cup \{(e^\circ, \Delta)\}$
  - (d)  $\forall e \in e^\circ \implies f(e) = f(e) - \Delta$ .
  - (e) Elimine de  $N$  (o del ciclo) todos los arcos con flujo cero.

Un ejemplo de ejecución de este algoritmo es evidenciado en la figura 7.101.

Hay tres observaciones cruciales de este algoritmo que nos ayudan a comprender la demostración:

1. El conjunto resultante  $E^\circ$  permite reconstruir enteramente la red.
2. Puesto que cada iteración elimina un ciclo, los ciclos contenidos en  $E^\circ$  son distintos.

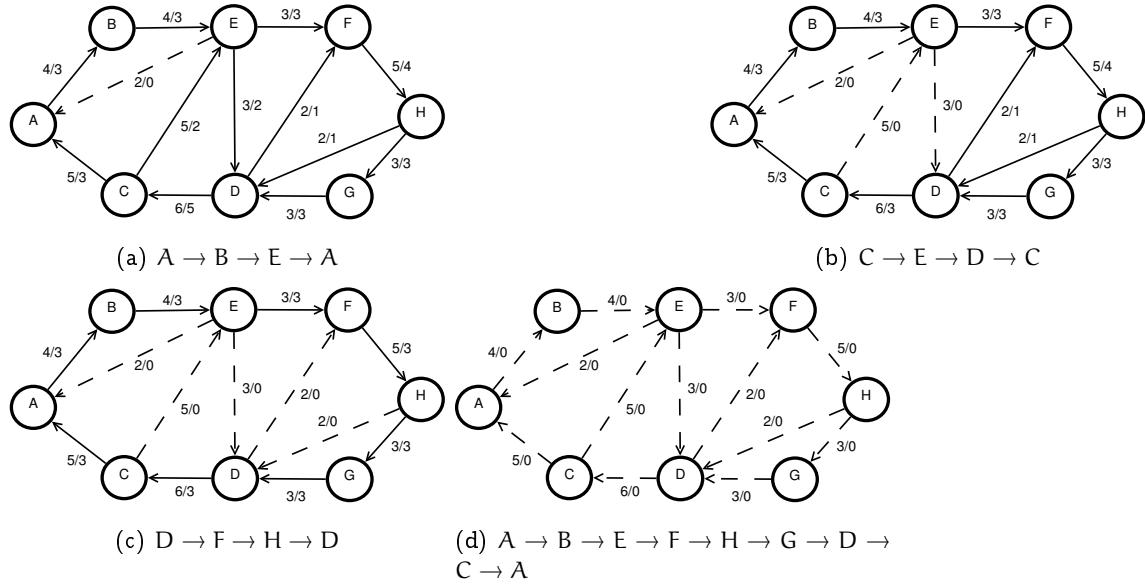


Figura 7.101: Ejemplo del teorema de descomposición de flujo sobre la circulación de la figura 7.100

3. El algoritmo termina en a lo sumo  $|E|$  iteraciones, que es la máxima cantidad posible de ciclos.

Cuenta habida de estas observaciones la proposición es cierta ■

### 7.11.6 Conectividad de grafos

Regresemos a la idea de conectividad de un grafo. En ese sentido, ¿cuán conexo puede considerarse un grafo? Dos métricas dan cuenta de la pregunta y se llaman “conectividad en arcos” y “conectividad en nodos” respectivamente. La conectividad en arcos es la mínima cantidad de arcos que deben eliminarse de un grafo para que éste quede dividido en dos componentes inconexos. Similarmente, la conectividad en nodos es la mínima cantidad de nodos a remover para que el grafo devenga desconectado.

Nosotros ya habíamos estudiado ideas afines a la conectividad. En § 7.5.14 (Pág. 613) estudiamos los puntos de articulación o de corte, los cuales expresan de alguna manera conectividades en nodos unitarias y más, porque aquí desarrollamos un algoritmo que en  $\mathcal{O}(E)$  nos calcula todos los nodos de corte. Además, para muchas situaciones es suficiente calcular los nodos de corte, que es barato en tiempo y espacio ( $\mathcal{O}(E)$  y  $\mathcal{O}(1)$  respectivamente), en lugar de calcular conectividades que, como veremos en esta subsección, son más costosas.

Por otra parte, durante nuestro estudio de árboles abarcadores mínimos presentamos la definición de corte (7.8.2.2 pag. 664). La conectividad en arcos no es otra cosa que calcular un corte de capacidad mínima a la excepción de que debemos hacerlo en término de un grafo y no de un digrafo. Haría falta pues un paralelo entre las redes de flujos y los grafos, el cual fue mostrado en § 7.11.1 (Pág. 773).

### 7.11.6.1 El teorema de Menger

A menudo, a un valor de conectividad en arcos  $k$  se le llama  $k$ -conexión y se denota formalmente como  $K_e(G)$ .

Decimos que un digrafo es “ $k$ -conectado” desde  $s$  hacia  $t$  si, para todo conjunto  $C$  de  $k - 1$  nodos, exceptuando a  $s$  y  $t$ , existe un camino desde  $s$  hacia  $t$ . Dicho de otro modo, no es posible desconectar a  $s$  de  $t$  sin eliminar  $k$  nodos.

Dos caminos entre  $s$  y  $t$  se denominan independientes si éstos no tienen nodos en común (a la excepción de  $s$  y  $t$ ).

Estamos listos para enunciar uno de los teoremas más interesantes de la teoría de grafos, descubierto antes de la computación moderna, el teorema de Menger.

**Proposición 7.17 (Karl Menger 1928 [123])** Si un digrafo es  $k$ -conectado desde un nodo  $s$  hacia otro  $t$ , entonces existen  $k$  caminos independientes desde  $s$  hacia  $t$ .

**Demostración** Dado un digrafo cualquiera, consideremos una red capacitada con los mismo nodos y arcos puestos a capacidad unitaria. Luego eliminemos todos los arcos entrantes a  $s$  y todos los salientes de  $t$ .

Ahora maximicemos la red. Por la proposición 7.8 (Pág. 719) sabemos que la cardinalidad del corte mínimo es  $k$ , pues todas las capacidades son unitarias y el digrafo es  $k$  conectado.

Sabemos, por el teorema 7.16 (Pág. 784) de descomposición de flujo, que podemos representar el flujo desde  $s$  hacia  $t$  como un conjunto de caminos disjuntos<sup>32</sup> cuya suma de flujos (por cada camino) es el valor de flujo de la red. Puesto que la cardinalidad del corte mínimo es  $k$ , existen  $k$  caminos distintos desde  $s$  hacia  $t$ , uno por cada arco del corte mínimo ■

El teorema de Menger nos aporta directamente el conocimiento para diseñar un algoritmo que calcule  $K_e(G)$  y, más aún, para calcular el corte crítico o mínimo, es decir, el conjunto específico de arcos que hay que suprimir para que el grafo devenga partido en dos o más subgrafos.

### 7.11.6.2 Cálculo de $K_e(G)$

Llaremos  $\overrightarrow{G}$  a la red transformada equivalente a una red no dirigida. Para encontrar  $K_e(G)$  podemos aplicar el siguiente algoritmo.

#### Algoritmo 7.10 (Cálculo de $K_e(G)$ )

Entrada: Un grafo  $G = \langle V, E \rangle$ .

Salida:  $K_e(G)$ .

1. Calcule  $\overrightarrow{G}$ .
2. Sea  $s$  el nodo en  $\overrightarrow{G}$  correspondiente al nodo de menor grado en  $G$ .
3.  $K_e(G) = \infty$
4.  $\forall v \in V \mid v \neq s$

<sup>32</sup> Átese un lazo  $t \rightarrow s$  que conforme el ciclo  $s \rightsquigarrow t \rightarrow s$  y se tiene la circulación.

- (a)  $t = v$
- (b) Sea la red  $N = \langle V, E', C, s, t \rangle$  tal que  $C = \{1 \mid e \in E' \cup \text{cap}(e) = 1\}$
- (c) Maximizar  $N$  y calcular el corte de capacidad mínima  $\langle V_s, V_t \rangle$ .
- (d) if flujo de  $N < K_e(G) \Rightarrow$ 
  - i.  $K_e(G) = |\langle V_s, V_t \rangle|$
  - ii. Aquí eventualmente se puede guardar  $\langle V_s, V_t \rangle$ .

#### 7.11.6.3 Implementación del algoritmo de cálculo de $K_e(G)$

La implantación del cálculo de la conectividad es relativamente simple, habida cuenta de que ya disponemos de toda la maquinaria necesaria. Comencemos entonces con el cálculo de  $\overrightarrow{G}$ :

787

*Calcular  $\overrightarrow{G}$  787*  $\equiv$  (788c)

```

typedef Net_Graph<Net_Node<Empty_Class>, Net_Arc<Empty_Class> > Net;
Net net;
typename Net::Node * source = NULL;
long min_degree = numeric_limits<long>::max();
for (Node_Iterator<GT> it(g); it.has_current(); it.next())
{
 typename GT::Node * p = it.get_current();
 NODE_COOKIE(p) = net.insert_node();
 if (g.get_num_arcs(p) < min_degree)
 {
 source = mapped_node<GT, Net> (p);
 min_degree = g.get_num_arcs(p);
 }
}
if (min_degree <= 1)
 return min_degree;

for (Arc_Iterator<GT, SA> it(g); it.has_current(); it.next())
{
 typename GT::Arc * a = it.get_current();
 typename Net::Node * src = mapped_node<GT, Net> (g.get_src_node(a));
 typename Net::Node * tgt = mapped_node<GT, Net> (g.get_tgt_node(a));
 if (src != source)
 net.insert_arc(tgt, src, 1);

 if (tgt != source)
 net.insert_arc(src, tgt, 1);
}

```

Uses mapped\_node 560b.

A efectos de ahorrar espacio y no tener que emplear una tabla, cada nodo de  $\overrightarrow{G}$  se guarda en el cookie del nodo en  $G$ .

Aprovechamos la pasada sobre los nodos de  $G$  para determinar el nodo de menor grado. Al final del primer for, source es la imagen en  $\overrightarrow{G}$  del nodo de menor grado en  $G$ . Si el grado es inferior o igual a la unidad, entonces no vale la pena proseguir, pues la conectividad no puede ser mayor.

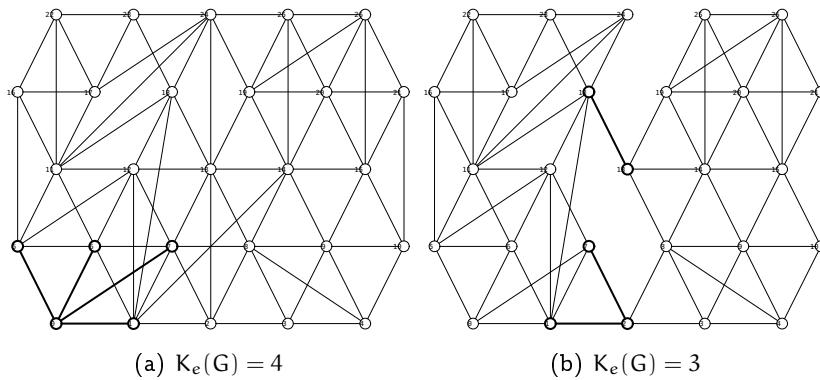


Figura 7.102: Dos grafos de distintas conectividades con sus cortes

El bloque  $\langle$ Calcular  $\vec{G}$  787 $\rangle$  nos cubre los pasos 1 y 2 del algoritmo 7.10.

El paso 4 se instrumenta con un `for` que recorra todos los nodos de  $\vec{G}$  (net en la implantación) y verifique de no procesar el fuente. Sea `sink` el nodo actual sobre el cual se encuentra el `for` mencionado.

El paso 4b consiste en suprimir los arcos que salen de `sink` (de este modo deviene sumidero), pero debemos guardar estos arcos suprimidos para restaurarlos en la siguiente iteración. Esto se implanta de la siguiente forma:

788a  $\langle$ Eliminar arcos salientes de sink 788a $\rangle \equiv$  (788c 789)

```
DynDlist<typename Net::Arc*> from_sink_arcs;
```

```
for (Node_Arc_Iterator<Net> it(sink); it.has_current(); it.next())
 from_sink_arcs.append(it.get_current());
```

```
for (typename DynDlist<typename Net::Arc*>::Iterator
 it(from_sink_arcs); it.has_current(); it.next())
 net.disconnect_arc(it.get_current());
```

Uses DynDlist 85a.

Empleamos una lista de arcos a eliminar `from_sink_arcs`. Efectuamos dos pasadas, la primera para guardar los arcos, la segunda para eliminarlos. Lo hacemos de esta manera porque es delicado eliminar un arco dentro de un iterador de arcos.

Los arcos salientes de `sink`, guardados en `from_sink_arcs`, deben ser restaurados mediante:

788b  $\langle$ Restaurar arcos salientes de sink 788b $\rangle \equiv$  (788c 789)

```
while (not from_sink_arcs.is_empty())
 net.connect_arc(from_sink_arcs.get());
```

Con todos los bloques desarrollados, el algoritmo 7.10 se implanta así:

788c  $\langle$ Conectividad en arcos 788c $\rangle \equiv$  789 ▷

```
template <class GT,
 template <class> class Max_Flow = Random_Preflow_Maximum_Flow,
 class SA = Default_Show_Arc<GT> >
long edge_connectivity(GT & g)
{
 \langle Calcular \vec{G} 787 \rangle
```

```
 long min_k = min_degree;
```

```

for (Node_Iterator<Net> it(net); it.has_current(); it.next())
{
 typename Net::Node * sink = it.get_current();
 if (sink == source)
 continue;

 ⟨Eliminar arcos salientes de sink 788a⟩

 const typename Net::Flow_Type flow = Max_Flow <Net> () (net);
 if (flow < min_k)
 min_k = flow;

 ⟨Restaurar arcos salientes de sink 788b⟩

 net.reset(); // colocar flujo en cero
}
return min_k;
}

```

La conectividad en arcos es una medida muy importante de “densidad” del grafo. En muchos contextos,  $K_e(G)$  expresa la robustez del grafo en sentido de su conectividad. En algunas situaciones puede ser importante calcular el corte mínimo de  $G$ . Para eso la biblioteca provee la rutina `compute_min_cut()`, cuya estructura es la misma que la de `edge_connectivity()`, con el añadido de que ésta calcula el corte mínimo de  $\vec{G}$  y, mediante mapeo a  $G$ , obtiene el valor del corte:

789

```

⟨Conejividad en arcos 788c⟩+≡ ▷788c
template <class GT,
 template <class> class Max_Flow = Heap_Preflow_Maximum_Flow,
 class SA = Default_Show_Arc<GT> >
long compute_min_cut(GT & g,
 Aleph::set<typename GT::Node*> & l,
 Aleph::set<typename GT::Node*> & r,
 DynDlist<typename GT::Arc *> & cut)
{
 ⟨Calcular y mapear \vec{G} 790⟩

 Aleph::set<typename Net::Node*> tmp_vs, tmp_vt;
 DynDlist<typename Net::Arc*> tmp_cuts, tmp_cutt;
 long min_k = numeric_limits<long>::max();
 for (Node_Iterator<Net> it(net); it.has_current(); it.next())
 {
 typename Net::Node * sink = it.get_current();
 if (sink == source)
 continue;

 ⟨Eliminar arcos salientes de sink 788a⟩

 Aleph::set<typename Net::Node*> vs, vt;
 DynDlist<typename Net::Arc *> cuts, cutt;
 const typename Net::Flow_Type flow =
 Min_Cut <Net, Max_Flow> () (net, vs, vt, cuts, cutt);
 }
}

```

```

if (flow < min_k)
{
 min_k = flow;
 tmp_vs.swap(vs);
 tmp_vt.swap(vt);
 tmp_cuts.swap(cuts);
 tmp_cutt.swap(cutt);
}
net.reset(); // colocar flujo en cero

<Restaurar arcos salientes de sink 788b>
}

<Determinar corte (l, r y cut) con tmp_vs, tmp_vt y tmp_cuts 791>

return min_k;

```

Uses DynList 85a.

Constatamos que la estructura es idéntica a `edge_connectivity()`. Hay una sutileza diferencial no mostrada en el bloque: si la conectividad es menor o igual que la unidad, no se calcula el corte; consecuentemente, el invocante debe consultar el valor de retorno para indagar si éste se calculó o no. La razón de esta decisión es doble. En primer lugar, si la conectividad es uno, entonces es bastante probable que el grafo sea esparcido, con más de un nodo de grado unitario. En este caso, el algoritmo calcularía flujos vanos y debería de considerar en las iteraciones la desconexión del valor actual de `sink`. En segundo lugar, en § 7.5.14 (Pág. 613) consagramos un amplia subsección para tratar con los puntos de corte, que es el caso particular para  $K_e(G) = 1$ .

Eventualmente, con los valores de  $l$ ,  $r$  y  $cut$  podemos emplear técnicas similares a las desarrolladas en § 7.5.15 (Pág. 622) para calcular los bloques.

```

790 <Calcular y mapear \vec{G} 790>≡ (789)
 typedef Net_Graph<Net_Node<Empty_Class>, Net_Arc<Empty_Class>> Net;
 Net net;
 typename Net::Node * source = NULL;
 long min_degree = numeric_limits<long>::max();
 for (Node_Iterator<GT> it(g); it.has_current(); it.next())
 {
 typename GT::Node * p = it.get_current();
 typename Net::Node * q = net.insert_node();
 GT::map_nodes (p, q);

 if (g.get_num_arcs(p) < min_degree)
 {
 source = mapped_node<GT, Net> (p);
 min_degree = g.get_num_arcs(p);
 }
 }
 if (min_degree <= 1)
 return min_degree;

```

```

for (Arc_Iterator<GT, SA> it(g); it.has_current(); it.next())
{
 typename GT::Arc * a = it.get_current();
 typename Net::Node * src = mapped_node<GT, Net>(g.get_src_node(a));
 typename Net::Node * tgt = mapped_node<GT, Net>(g.get_tgt_node(a));
 if (src != source)
 {
 typename Net::Arc * arc = net.insert_arc(tgt, src, 1);
 ARC_COOKIE(arc) = a;
 }
 if (tgt != source)
 {
 typename Net::Arc * arc = net.insert_arc(src, tgt, 1);
 ARC_COOKIE(arc) = a;
 }
}

```

Uses `mapped_node` 560b.

791  $\langle$  Determinar corte ( $l, r$  y  $cut$ ) con  $tmp\_vs$ ,  $tmp\_vt$  y  $tmp\_cuts$  791  $\rangle \equiv$  (789)

```

for (typename Aleph::set<typename Net::Node*>::iterator
 it = tmp_vs.begin(); it != tmp_vs.end(); it++)
 l.insert(mapped_node <Net, GT> (*it));

for (typename Aleph::set<typename Net::Node*>::iterator
 it = tmp_vt.begin(); it != tmp_vt.end(); it++)
 r.insert(mapped_node <Net, GT> (*it));

for (typename DynDlist<typename Net::Arc*>::Iterator it(tmp_cuts);
 it.has_current(); it.next())
{
 typename Net::Arc* arc = it.get_current();
 cut.append(mapped_arc<Net, GT> (arc));
}

```

Uses `DynDlist` 85a and `mapped_node` 560b.

#### 7.11.6.4 Análisis del algoritmo 7.10

En los términos en que lo hemos planteado, el rendimiento del algoritmo 7.10 se corresponde con el de la rutina `edge_connectivity()`. De allí se nota claramente que se ejecutan  $|V| - 1$  maximizaciones sobre  $\vec{G}$ . Por tanto, el desempeño será  $V - 1 \times$  (desempeño de `Max_Flow <Net> () (net)`). Si empleamos un algoritmo de empuje por pre-flujo, que es el algoritmo por omisión, entonces podemos encontrar  $K_e(G)$  en  $(V - 1) \times \mathcal{O}(V^2 E) = \mathcal{O}(V^3 E)$  pasos.

$\langle$  Calcular  $\vec{G}$  787  $\rangle$  toma  $\mathcal{O}(E)$  pasos, los cuales son asintóticamente absorbidos por el `for`. Del mismo modo,  $\langle$  Eliminar arcos salientes de sink 788a  $\rangle$  y  $\langle$  Restaurar arcos salientes de sink 788b  $\rangle$  toman el grado de salida sink, el cual lo absorbe asintóticamente `Max_Flow <Net> () (net)`.

### 7.11.7 Cálculo de $K_v(e)$

Obviamente, la conectividad en nodos está estrechamente relacionada con su equivalente en arcos. Es directamente deducible que  $K_v(G) \leq K_e(G)$ , pues los arcos que determinan  $K_e(G)$  no sólo deben relacionar a los nodos que desconectan a  $G$ , sino que es posible que uno o más arcos de los involucrados en el corte que arroje  $K_e(G)$  comparta nodos involucrados en  $K_v(G)$ . Podemos calcular, pues,  $K_v(G)$  a partir del corte mínimo. Para ello, mediante `compute_min_cut()`, calculamos el corte mínimo del grafo y luego miramos el conjunto de nodos que contiene el corte. De este conjunto, en relación a los arcos del corte, calculamos  $K_v(G)$ .

Otra alternativa consiste en desarrollar un algoritmo similar al 7.10. Pero para eso debemos interpretar (o plantear) el teorema de Menger (7.17) en función de nodos y no de arcos. En este sentido, una primitiva de cálculo de  $K_v(G)$ , llamada `vertex_connectivity()`, del mismo estilo que `edge_connectivity` pero para los nodos, es exportada en `ALCPH`.

## 7.12 Flujos de coste mínimo

En ocasiones, mantener un flujo por una red acarrea costes. Consideraremos algunas situaciones:

1. En una red que modelice algún fluido, mantener un valor de flujo conlleva un coste energético o económico a veces proporcional a la longitud de la tubería.
2. En una red de tráfico aéreo mantener un flujo dado por alguna ruta también involucra costes de combustible, cantidad de aviones, etc.
3. Una red eléctrica requiere costes de mantenimiento según su ubicación geográfica.

Estos ejemplos nos permiten vislumbrar que hay situaciones en las cuales no sólo cuenta maximizar la cantidad de flujo, sino satisfacer algún criterio de sustentabilidad según sea la índole de los costes.

Los costes, no necesariamente son de índole económica y los flujos no necesariamente representan líquidos. A tenor de ejemplo, una aplicación clásica se relaciona con el transporte de insumos o mercancías. En este caso hablamos de una red de transporte. La capacidad de flujo representa a la de transporte asociada a una duración. Aparece pues el problema de maximizar la cantidad de insumos a transportar en el menor tiempo posible.

Para tratar problemas que encajen en la maximización de flujo al mínimo coste, plantearemos el siguiente modelo.

**Definición 7.21 (Modelo de red capacitada con costes)** Una red capacitada de flujo con costes es una red capacitada  $N = < V, E, s, t, C, C' >$ .  $C' : E \rightarrow C$ , la cual le asocia a cada arco un atributo adicional llamado “coste” que representa el coste por unidad de flujo.

El coste asociado a un arco se denota como  $c(a)$  y representa lo que cuesta fluir una unidad.

El coste de un flujo por un arco  $a$  se define como:

$$c_f(a) = f(a) c(a) \quad (7.29)$$

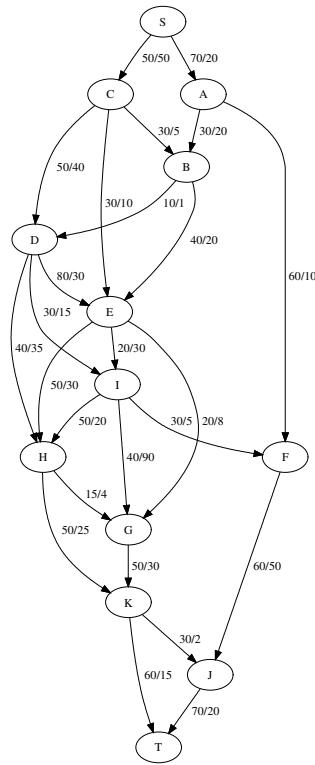


Figura 7.103: Ejemplo de una red capacitada con costes. Primer valor de cada arco corresponde a las capacidades, mientras que el segundo al coste de llevar una unidad de flujo

El costo de un flujo de una red se define como:

$$c_f(N) = \sum_{\forall e \in E} c_f(e) = \sum_{\forall e \in E} f(e) c(e) \quad (7.30)$$

Con esta definición ya estamos preparados para enunciar un problema general que nos servirá de fundamento de solución a una buena variedad de otros problemas.

**Definición 7.22 (Flujo máximo de coste mínimo)** Sea  $N = < V, E, s, t, C, C' >$  una red capacitada con costes. Un flujo máximo se dice de “coste mínimo” si no existe otro flujo máximo con coste menor.

En una red capacitada puede haber varias configuraciones de flujos que conduzcan a un flujo máximo. Por ejemplo, para la red ilustrada en la figura 7.103 tenemos dos (o más) configuraciones que maximizan el flujo y que son mostradas en las figuras 7.104-a y 7.103-b, respectivamente. Ambos flujos son máximos, pero el primero tiene un coste de 14860 mientras que el segundo de 16400.

Sabemos que una red capacitada puede tener varias configuraciones que maximicen el flujo. Así las cosas, puede aparecerse como enfoque de solución el encontrar todas las configuraciones máximas y entonces seleccionar la de menor coste, pero entonces caemos en el terreno de lo intratable.

Puesto que ambas técnicas de maximización de flujo se sirven de la red residual, es de interés definir una red residual en el ámbito de coste.

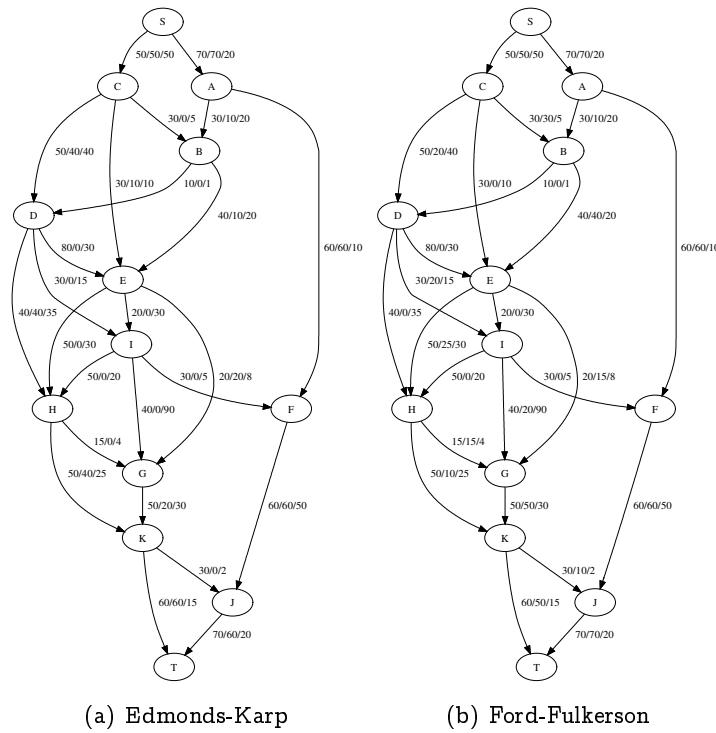


Figura 7.104: Dos configuraciones de flujo máximo para la red de la figura 7.103

**Definición 7.23 (Red residual en una red capacitada con costes)** Sea  $N = \langle V, E, s, t, C, C' \rangle$  una red capacitada con costes. Respecto al flujo, su red residual es idéntica a la de la definición 7.16 (Pág. 721) con la que ya hemos trabajado, pero respecto al coste, los arcos residuales tienen coste negativos.

El coste negativo para un arco residual cobra más sentido cuando se piensa en un arco de retroceso. Al decrementar el flujo por un arco de retroceso aumenta el valor de flujo y con ello el coste total. Así, aumentar flujo por un arco residual hacia el sumidero (un arco de retroceso) equivale a disminuir el coste total, pues en realidad se disminuye el flujo.

La figuras 7.105-a y 7.105-b muestran las redes residuales de los flujos máximos mostrados en las figuras 7.104-a y 7.104-b respectivamente. En ambas figuras se pueden notar la presencia de ciclos negativos según el coste del flujo. Vemos pues que entre distintos flujos posibles, máximos inclusive, se pueden tener ciclos de coste negativo, lo cual sugiere, cuando menos por absurdo, que el flujo en cuestión no puede ser de mínimo coste. Cabe entonces replantear la idea de flujo factible.

**Definición 7.24 (Flujo factible sobre una red capacitada con costes)** Sea  $N = \langle V, E, s, t, C, C' \rangle$  una red capacitada con costes. Un flujo máximo es “factible” si, y sólo si, éste no contiene ciclos de coste negativo.

La factibilidad estriba en que es absurdo tener ciclos negativos. Pero este carácter de absurdo tiene capacidad reveladora, tal como veremos en la siguiente proposición fundamental.

**Proposición 7.18** Un flujo máximo es de coste mínimo si, y sólo si, su red residual no contiene ciclos negativos.

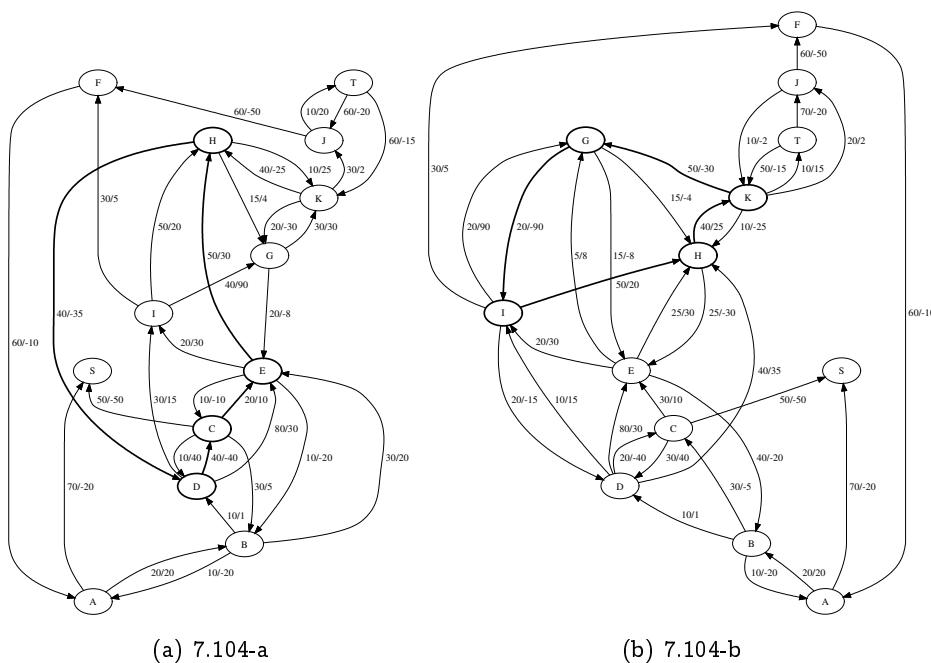


Figura 7.105: Redes residuales de para los flujos máximos de las figuras 7.104-a y 7.104-b. Para cada figura se resalta ciclos negativos según el coste

**Demostración** Para la aprensión es menester recordar la idea de circulación y el teorema 7.16 (Pág. 784) de descomposición de flujo en circulaciones. En este sentido consideremos un ciclo (negativo o no) como una circulación en la red. Ahora planteemos los dos caminos mostrativos:

- $\Rightarrow$  (por contradicción): supongamos que tenemos un flujo máximo de coste mínimo cuya red residual contiene un ciclo de coste negativo. Sea  $u$  el arco del ciclo con menor capacidad restante con valor  $c_m$ . Ahora aumentemos el flujo de la red a través del ciclo en  $c_m$  sumándoselo a todos los arcos de coste positivo (de adelanto) y restándoselo a todos los arcos de coste negativo (de retroceso). Puesto que estos cambios no afectan la propiedad de conservación del flujo, el flujo circulante es el mismo y, por tanto, aún máximo, pero su coste disminuye en  $c_m$ , lo que contradice la suposición de que el coste es mínimo. Por tanto, si un flujo máximo es de mínimo coste, entonces éste no puede tener ciclos negativos  $\square$
  - $\Leftarrow$  (por contradicción): supongamos que conocemos un flujo máximo  $f'$ , sin ciclos negativos, cuyo coste no es mínimo. Del mismo modo, supongamos que conocemos un flujo  $f_m$  máximo de coste mínimo. En esta hipotética situación seríamos capaces de conocer los valores de los costes y mirar la diferencia de coste  $f_m - f'$ .

Según el teorema 7.16 (Pág. 784) de descomposición de flujo, por la vía de aplicación del algoritmo 7.9 (Pág. 784) podemos operar sobre  $f'$  mediante búsquedas sucesivas de ciclos, aumentos y reducciones, y así deparar en la misma configuración de flujo  $f_m$ . Sin embargo, puesto que  $f'$  no tiene ciclos negativos, esta serie de operaciones no podría bajar el coste total. Nos encontraríamos entonces con un flujo  $f''$  idéntico en valor de flujo a  $f_m$  pero con coste (el de  $f''$ ) distinto, lo cual es una contradicción. Por consiguiente, si un flujo no tiene ciclos negativos, entonces éste es máximo y de coste

mínimo ■

El conocimiento que nos arroja este teorema es la base de una gama de algoritmos que calculan el flujo máximo cuyo coste es mínimo. Fundamentalmente, esta clase de algoritmos se estructuran como sigue.

#### Algoritmo 7.11 (Algoritmo genérico de eliminación de ciclos negativos)

La entrada del algoritmo es una red capacitada de costes  $N$ .

La salida es la misma red con el flujo máximo cuyo coste es mínimo.

1. Calcule un flujo inicial sobre  $N$  (puede ser un flujo máximo)
2. while  $N$  contenga ciclos de coste negativo
  - (a) Sea  $C$  un ciclo de coste negativo sobre  $N$ . Sea  $c_m$  la mínima capacidad restante del ciclo.
  - (b) Aumente el flujo de  $N$  en el valor de  $c_m$ .

#### 7.12.1 El TAD Net\_Max\_Flow\_Min\_Cost

El TAD Net\_Max\_Flow\_Min\_Cost instrumenta la maquinaria necesaria para operar redes de flujo con costes asociados. Su especificación reside en el archivo `tpl_maxflow_mincost.H`.

Net\_Max\_Flow\_Min\_Cost se fundamenta sobre Net\_Graph, con el añadido de que sus arcos albergan información para el coste:

796a *(Arco con coste 796a)≡*

```
template <typename Arc_Info, typename F_Type = long>
class Net_Cost_Arc : public Net_Arc<Arc_Info, F_Type>
{
 typedef F_Type Flow_Type;
 Flow_Type cost;

 Flow_Type flow_cost() const { return this->flow*cost; }
};
```

Como vemos, sólo se añade el coste por unidad de flujo. Los nodos son los mismos que para Net\_Graph.

Con lo anterior ya podemos definir el TAD Net\_Max\_Flow\_Min\_Cost :

796b *(Clase Net\_Max\_Flow\_Min\_Cost 796b)≡*

```
template <class NodeT, class ArcT>
class Net_Max_Flow_Min_Cost : public Net_Graph<NodeT, ArcT>
{
 <Métodos públicos de Net_Max_Flow_Min_Cost 796c>
};
```

El uso es similar al de los grafos y redes: se especifican los nodos a través del tipo Net\_Node, los arcos con Net\_Cost\_Arc y la red con Net\_Max\_Flow\_Min\_Cost .

Un primer método de Net\_Max\_Flow\_Min\_Cost consiste en ofrecer la capacidad de consultar o modificar el coste asociado a un arco:

796c *(Métodos públicos de Net\_Max\_Flow\_Min\_Cost 796c)≡* (796b) 797a▷

```
Flow_Type & get_cost(Arc * a) { return a->cost; }
```

así como de consultar el coste del flujo:

797a *(Métodos públicos de Net\_Max\_Flow\_Min\_Cost 796c) +≡ (796b) ▷796c 797b▷*  
 const Flow\_Type & flow\_cost(Arc \* a) const { return a->flow\_cost(); }

Puesto que en este modelo se requiere del coste, tenemos la necesidad de sobrecargar la inserción de arco:

797b *(Métodos públicos de Net\_Max\_Flow\_Min\_Cost 796c) +≡ (796b) ▷797a 797c▷*  
 virtual Arc \* insert\_arc(Node \* src\_node, Node \* tgt\_node,  
                           const Flow\_Type & cap, const Flow\_Type & cost)  
 {  
     Arc \* a = Net::insert\_arc(src\_node, tgt\_node, Arc\_Type(), cap, 0);  
     a->cost = cost;  
     return a;  
 }

Otra alternativa es emplear cualquier primitiva de inserción de Net\_Graph y luego especificar el coste, el cual, según el constructor de Net\_Cost\_Arc, también toma valor omisión cero.

El coste del flujo circulante por la red se calcula según (7.30) (pág 793) y se instrumenta así:

797c *(Métodos públicos de Net\_Max\_Flow\_Min\_Cost 796c) +≡ (796b) ▷797b*  
 Flow\_Type compute\_flow\_cost()  
 {  
     Flow\_Type total = 0;  
     for (Arc\_Iterator<Net\_MFMC> it(\*this); it.has\_current(); it.next())  
     {  
         Arc \* a = it.get\_current();  
         if (not a->is\_residual)  
             total += a->flow\_cost();  
     }  
     return total;  
 }

Definido el TAD Net\_Max\_Flow\_Min\_Cost , nos abocaremos a desarrollar otras piezas que requerimos para instrumentar el algoritmo 7.11.

### 7.12.1.1 Acceso al coste de un arco

Para detectar ciclos negativos nos servimos del algoritmo de Bellman-Ford presentado en § 7.9.3 (Pág. 692). Este algoritmo requiere una clase de acceso a la distancia (véase § 7.8.1 (Pág. 660)). En nuestro caso debemos modelizar el coste en función del valor de flujo circulante por el arco y considerar si éste es o no residual. Esto lo hacemos siguiendo las reglas de definición de acceso de pesos a los arcos:

797d *(Funciones para Net\_Max\_Flow\_Min\_Cost 797d) ≡* 798▷  
 template <class Net> class Access\_Cost  
 {  
     typename Net::Flow\_Type operator () (typename Net::Arc \* a) const  
     {  
         return a->is\_residual ? -((typename Net::Arc \*) a->img\_arc)->cost : a->cost;  
     }  
 };

### 7.12.2 Algoritmos de máximo flujo con coste mínimo mediante eliminación de ciclos

Ahora disponemos de todo lo necesario para instrumentar variantes del algoritmo 7.11. Fundamentalmente presentaremos dos algoritmos, cuya diferencia estriba en la manera en que se calcula un valor de flujo inicial.

#### 7.12.2.1 Eliminación de ciclos con flujo máximo inicial

Nuestra primera variante emplea un flujo máximo inicial. Puesto que disponemos de una completa familia de algoritmos para maximizar flujo, el criterio de maximización será un parámetro de la primitiva:

```
798 <Funciones para Net_Max_Flow_Min_Cost 797d>+≡ ◁797d 800▷
 template <class Net, template <class> class Max_Flow_Algo>
 void max_flow_min_cost_by_cycle_canceling
 (Net & net, bool leave_residual = false)
 { // tipos sinónimos para hacer más simple la lectura
 typedef Aleph::less<typename Access_Cost<Net>::Distance_Type> Cmp;
 typedef Aleph::plus<typename Access_Cost<Net>::Distance_Type> Plus;
 typedef Res_F<Net> Filter;

 Max_Flow_Algo <Net> () (net, true); // obtiene flujo máximo inicial

 Path<Net> cycle(net); // aquí se guardan ciclos negativos
 // eliminar ciclos negativos mientras éstos existan
 while (Bellman_Ford_Negative_Cycle
 <Net, Access_Cost<Net>, Cmp, Plus, Filter>()(net, cycle))
 if (increase_flow <Net> (net, cycle) == 0)
 break;
 }
```

Uses Path 578a.

La red maximizada a coste mínimo calculada mediante este programa con el algoritmo de Ford-Fulkerson es mostrada en la figura 7.106. El coste mínimo del flujo máximo es 13620.

En este momento puede decirse que nos encontramos en un punto culminante de este texto. Como vemos, la rutina `max_flow_min_cost_by_cycle_canceling()` instrumenta el algoritmo 7.11 de una manera prácticamente idéntica a su especificación, “simplicidad” posible gracias a que nos apoyamos sobre toda una maquinaria de estructura de datos y algoritmos. En este caso podemos mencionar el TAD `List_Graph`, así como sus clases derivadas. Del mismo modo, este algoritmo subyace sobre otros algoritmos, de los cuales cabe mencionar la búsqueda de caminos de aumento, maximización de flujo, búsqueda de ciclos negativos, caminos más cortos y componentes fuertemente conexos, entre otros, los cuales a su vez subyacen en más estructuras de datos y algoritmos.

#### 7.12.2.2 Eliminación de ciclos mediante supra arco negativo

Otro algoritmo basado eliminación de ciclos negativos parte desde un valor de flujo nulo. Luego se aumenta el flujo en un valor dado, se eliminan ciclos negativos y se vuelve a aumentar el flujo. El procedimiento se aplica sucesivamente hasta que ya no haya ciclos

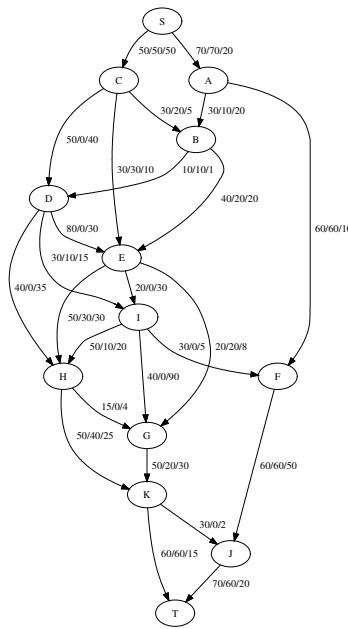
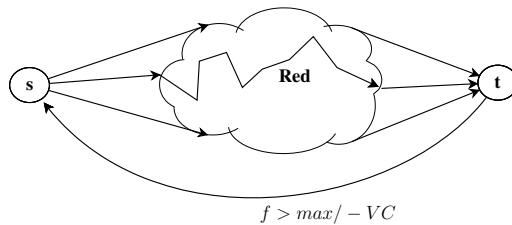


Figura 7.106: Flujo máximo a coste mínimo de la red de la figura 7.103 (Pág. 793) producido por la eliminación de ciclos con flujo máximo inicial

negativos; momento en el cual, por el conocimiento que nos aportó la proposición 7.18 (Pág. 794) ( $\Leftarrow$ ), sabemos que el flujo es máximo.

Ahora bien, ¿por cuál valor de flujo comenzar? Si bien existen varias y antiguas técnicas [59, 138, 107], una de las cuales se caracteriza por su simplicidad y velocidad -relativa a la clase de algoritmo-, y que consiste en añadir un lazo entre el fuente y el sumidero con flujo mayor al máximo y coste negativo cuyo valor absoluto sea mayor al coste total de cualquier camino entre el fuente y el sumidero. A este lazo lo llamamos “supra arco”. Este “truco” nos permitirá evadir el cálculo inicial del flujo máximo e ir iterativamente calculando el mínimo coste a la vez que maximizamos la red. La idea es crear un ciclo negativo inicial, cual puede pictorizarse del siguiente modo:



Los ciclos que contienen al supra arco fungen de caminos de aumento, pues incluyen un camino entre el fuente y el sumidero. Un intento de eliminación de este ciclo mueve la mayor cantidad posible de flujo del supra arco hacia el camino de aumento entre el fuente y el sumidero. Según sea el valor del eslabón mínimo, el ciclo puede o no eliminarse. Así, esta clase de ciclos aumenta el flujo.

Los ciclos que no contienen al supra arco están dominados, en el sentido de ser negativos por arcos de retroceso (o residuales). Su eliminación reduce el coste total del flujo.

El enfoque de supra arco tiene la ventaja de ser más sencillo de implementar que el

anterior, pues no requiere explícitamente pensar en algorítmica de flujo máximo, caminos de aumento y otras cosas.

El primer paso hacia el diseño de este algoritmo es la inserción del supra arco:

```
800 <Funciones para Net_Max_Flow_Min_Cost 797d>+≡ <798 801>
 template <class Net, template <class> class Cost> static
 typename Net::Arc * create_dummy_arc(Net & net)
 {
 const typename Net::Flow_Type max_cost =
 net.get_num_nodes() * search_max_arc_cost <Net, Cost> (net);

 net.make_residual_net();

 typename Net::Node * src = net.get_source();
 typename Net::Node * tgt = net.get_sink();
 const typename Net::Flow_Type max_flow =
 std::min(net.get_out_cap(src), net.get_in_cap(tgt));
 typename Net::Net::Digraph * digraph = &net;
 typename Net::Arc * a = digraph->insert_arc(src, tgt);
 net.get_cookie() = a;

 typename Net::Arc * img = digraph->insert_arc(tgt, src);

 digraph->disconnect_arc(a); // no supra-arco, dejarlo por coste

 a->is_residual = false;
 a->img_arc = img;
 a->cap = max_flow;
 a->cost = max_cost;
 a->flow = 0;

 img->is_residual = true;
 img->img_arc = a;
 img->cap = max_flow;
 img->cost = max_cost;
 img->flow = 0;

 return img;
 }
```

Para que la clase de observación de distancia Access\_Cost considere al supra arco negativo, éste debe ser residual. De allí que creamos dos arcos, uno residual, que es el supra arco, y otro normal, que es desligado de la red mediante disconnect\_arc(); de este modo, el arco paralelo al supra arco no es visto por ningún iterador de arcos.

create\_dummy\_arc() apela a dos rutinas. La primera, compute\_max\_possible\_flow(), retorna el máximo valor de flujo que puede alcanzar la red en función de las capacidades de salida del fuente y de entrada del sumidero. La segunda rutina, search\_max\_arc\_cost(), retorna el mayor valor del coste C entre todos los arcos de la red. El coste máximo de algún flujo de red está acotado por  $V \cdot C$ , cual es mayor que la máxima longitud en arcos de un ciclo (circulación).

Simétrico a create\_dummy\_arc(), hay otra primitiva llamada destroy\_dummy\_arc(),

la cual elimina el supra arco y la red residual.

Ahora ya estamos listos para desarrollar el algoritmo:

```
801 <Funciones para Net_Max_Flow_Min_Cost 797d>+≡ ◁800
 template <class Net>
 void max_flow_min_cost_by_cycle_canceling(Net & net)
 {
 typedef Aleph::less<typename Access_Cost<Net>::Distance_Type> Cmp;
 typedef Aleph::plus<typename Access_Cost<Net>::Distance_Type> Plus;
 typedef Res_F<Net> Res_Filter;

 create_dummy_arc <Net, Access_Cost> (net);

 Path<Net> cycle(net); // aquí se guardan ciclos negativos
 while (Bellman_Ford_Negative_Cycle
 <Net, Access_Cost<Net>, Cmp, Plus, Res_Filter> () (net, cycle))
 if (increase_flow <Net> (net, cycle) == 0)
 break;

 destroy_dummy_arc(net);
 }
```

Uses Path 578a.

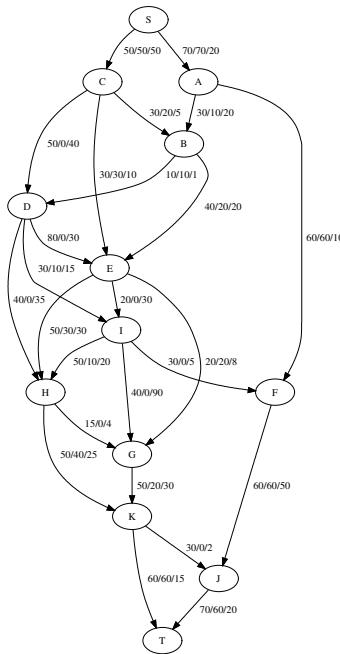


Figura 7.107: Flujo máximo a coste mínimo de la red de la figura 7.103 producido por la eliminación de ciclos sin flujo inicial y supra arco

El flujo y coste del supra arco sea calculan un “poco” más altos que el flujo máximo y el coste máximo posible. La primera iteración arroja con certeza un ciclo que contiene al supra arco. Todo ciclo que involucra al supra arco conforma un camino de aumento, pues éste incluye al fuente y al sumidero. Por consiguiente, toda cancelación de ciclos negativos por el supra arco se corresponde con un camino de aumento. Según la suerte, el algoritmo aumentará flujo por un ciclo que contenga al supra arco o meramente eliminará ciclos negativos sin aumentar el flujo. Una vez que el flujo sea máximo el algoritmo no

encuentra más ciclos que contengan al supra arco y sólo se remite a buscar ciclos negativos y cancelarlos.

### 7.12.3 Análisis de los algoritmos basados en eliminación de ciclos negativos

Para analizar la clase de algoritmos que acabamos de estudiar debemos apelar a los análisis previos de flujos máximos y el algoritmo de Bellman-Ford. La siguiente proposición elucida al respecto.

**Proposición 7.19** Sea  $N = \langle V, E, s, t, C, C' \rangle$  una red capacitada con costes. Sea  $\mathcal{F}$  el valor de flujo máximo. Sea  $C$  el máximo coste de un flujo máximo. Entonces, si el digrafo es esparcido, la duración de ejecución del algoritmo 7.11 (Pág. 796) está acotada por  $\mathcal{O}(V^3 C \mathcal{F})$ .

**Demostración** El peor escenario que podemos encontrar es que cada arco tenga capacidad  $\mathcal{F}$  y coste  $C$ . En esta situación puede alcanzarse un coste máximo de red de  $\mathcal{O}(E C M)$ .

La búsqueda de ciclos en digrafos se basa en el algoritmo de Tarjan (§ 7.7.3.2 (Pág. 642)), el cual emplea una búsqueda en profundidad que cuesta  $\mathcal{O}(E)$ .

Desde el análisis del algoritmo de Ford-Fulkerson (§ 7.10.11.1 (Pág. 730)) sabemos que pueden requerirse  $\mathcal{O}(\mathcal{F})$  repeticiones para maximizar el flujo, caso extremo cuando se aumentan circulaciones en una unidad. Por tanto, maximizar el flujo puede requerir  $\mathcal{O}(E\mathcal{F})$ .

Pero un análisis análogo al del párrafo anterior nos arroja la posibilidad de que puedan requerirse  $\mathcal{O}(E C \mathcal{F})$  disminuciones de coste para disminuir el coste al mínimo.

Cada una de estas disminuciones requiere invocar al algoritmo de Bellman-Ford, el cual es, según lo analizado en § 7.9.3.7 (Pág. 702),  $\mathcal{O}(V E)$ .

Consecuentemente, el algoritmo puede tomar  $\mathcal{O}(E C \mathcal{F}) \times \mathcal{O}(V E) = \mathcal{O}(V E^2 C \mathcal{F})$ .

Si el grafo es esparcido, entonces  $E \approx cV$ , donde  $c$  es constante, lo cual nos arroja una complejidad de  $\mathcal{O}(V^3 C \mathcal{F})$  ■

En rigor a la práctica, este análisis es muy pesimista. En promedio, la duración es mucho mejor. Sin embargo, por cada búsqueda de ciclo, esta clase de algoritmo no aprovecha el cálculo realizado y conocimiento obtenido por búsquedas anteriores de ciclos. Por añadidura, el algoritmo de Bellman-Ford retorna ciclos negativos “en bruto”. no necesariamente los que más puedan acelerar el aumento del flujo o la reducción del coste.

### 7.12.4 Problemas que se reducen a enunciados de flujo máximo a coste mínimo

A efectos de mejorar nuestra compresión, consideremos una hipotética situación macroeconómica que vincule productos agropecuarios, regiones o países productores y ciudades que demanden los productos.

En el sentido ya explicado, supongamos como productos a la carne, pollo, pescado y arroz y como productores a Falcón, Sucre, Portuguesa, Zulia, Argentina, Ecuador, Colombia y Brasil. Las capacidades de producción de cada zona se pictorian en el grafo de la figura 7.108-a., el cual puede interpretarse como un esquema de provisión.

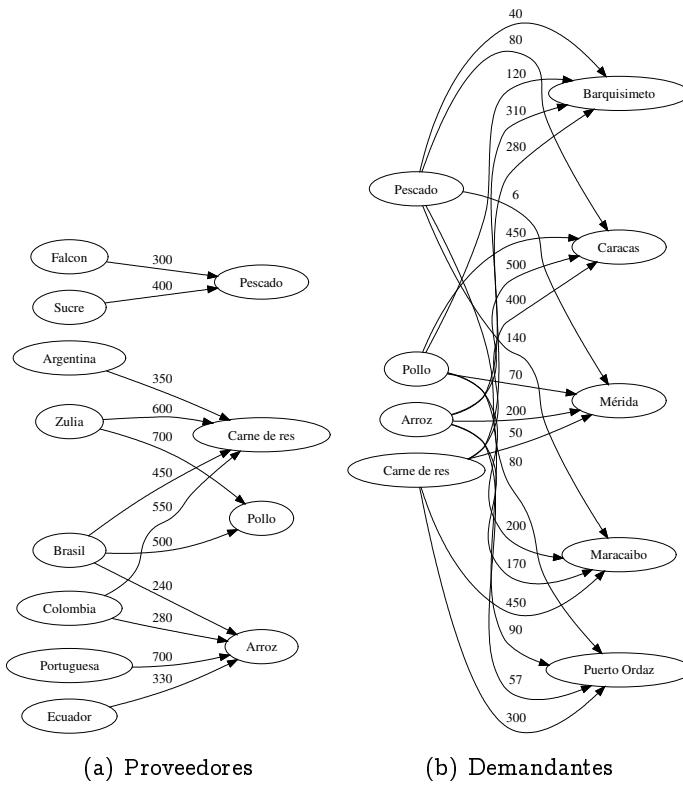


Figura 7.108: Grafos de proveedores y demandas de alimentos

Del mismo modo, las regiones ejercen una demanda de los productos. En nuestro hipotético ejemplo, el grafo de la figura 7.108-b ilustra las necesidades de los productos por cada región.

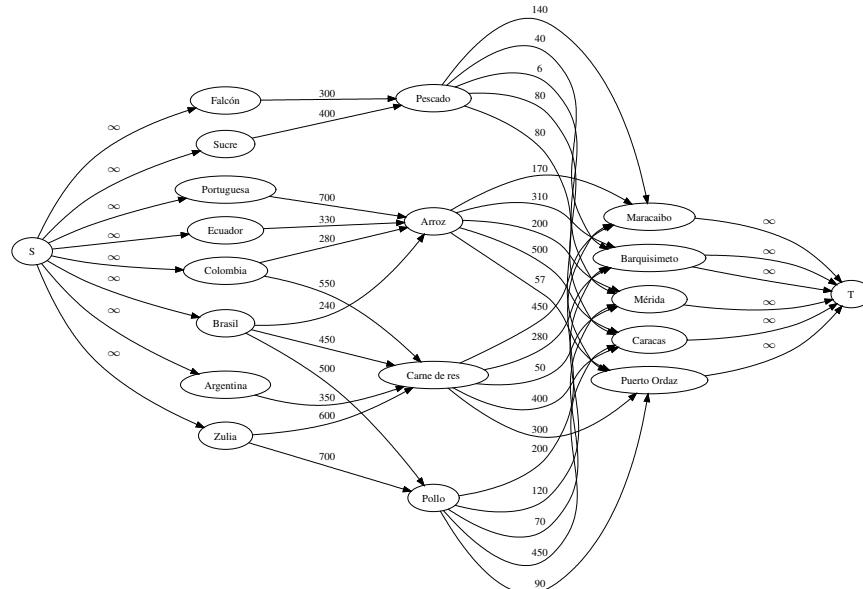


Figura 7.109: Red equivalente para calcular factibilidad provisión/demanda

Para estudiar la factibilidad de satisfacción de necesidades según las capacidades puede usarse una variante de la técnica explicada en § 7.11.2 (Pág. 774), la cual se pictoriza en la figura 7.109. La idea consiste en “fusionar” los nodos productos y así conectar productores con demandantes. En este caso, la capacidad de un arco entre producto y ciudad representa la demanda -sin necesidad de partir el nodo ciudad como se explica en § 7.11.2 (Pág. 774). Luego, se enlazan los productores con un suprafuente mediante arcos de capacidad infinita y los demandantes con un suprasumidero con arcos de capacidad infinita (en caso de que se desee validar la factibilidad). Luego de maximizar el flujo, si cada arco desde producto hacia demandante tiene flujo igual a su capacidad, entonces el flujo es factible, es decir, la demanda puede ser satisfecha.

Saber si el flujo es factible o no es esencial antes de pensar en los costes asociados al producto, pues la estructura de la red define la factibilidad. La suma de demandas versus la suma de provisiones no necesariamente arroja la factibilidad, pues ésta depende de la posibilidad de llevar el producto hacia el demandante, lo cual, en términos de una red, depende de las capacidades entre los arcos que conectan al productor con el demandante y de la topología.

Hasta el presente sólo hemos estudiado redes de flujo con un solo tipo de “fluído” circulante. En este momento es en sumo importante notar que al estar en juego cuatro productos distintos, el planteamiento de este problema podría interpretarse con cuatro fluidos distintos, correspondientes al pescado, arroz, pollo y carne. Si bien eso es correcto, en este caso es equiparable a un solo fluido correspondiente al valor de la demanda. Otra manera de interpretar esto es descomponer la red de la figura 7.109 en subredes por productos. En este caso, cualesquiera que sean las soluciones obtenidas, éstas son idénticas a las que nos arroja una sola red.

Mencionaremos muy brevemente las redes con varios fluidos en § 7.14.5 (Pág. 825).

#### 7.12.4.1 Transporte

En términos generales, en el problema del transporte se tienen  $m$  proveedores y  $n$  demandantes. Cada proveedor tiene una cierta capacidad de provisión y cada demandante tiene una necesidad o demanda de provisión. Por cada asociación entre un proveedor  $i$  y un demandante  $j$  existe un coste de transporte  $c_{i,j}$ . Como ejemplo consideremos la siguiente instancia:

|             |           | Demandantes: | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ |     |
|-------------|-----------|--------------|-------|-------|-------|-------|-------|-------|-----|
|             |           | Demanda:     | 30    | 40    | 10    | 25    | 15    | 20    | 140 |
| Proveedores | Provisión |              |       |       |       |       |       |       |     |
|             | $P_1$     | 30           | 78    | 30    | 90    | 16    | 17    | 20    |     |
|             | $P_2$     | 50           | 15    | 20    | 76    | 10    | 20    | 20    |     |
|             | $P_3$     | 25           | 100   | 120   | 85    | 75    | 40    | 50    |     |
|             | $P_4$     | 80           | 25    | 50    | 30    | 75    | 20    | 30    |     |
|             | $P_5$     | 70           | 85    | 35    | 40    | 76    | 40    | 15    |     |
|             |           | 250          |       |       |       |       |       |       |     |

Este problema puede modelizarse y resolverse mediante una red capacitada. El nodo fuente conecta a los proveedores con capacidades iguales a las provisiones y a costes nulos. Los nodos demandantes se conectan al sumidero con capacidades iguales a sus demandas y a costes nulos. Finalmente, cada coste de transporte se representa por un arco desde el proveedor hacia el demandante con capacidad infinita -o un valor superior o igual que la provisión- y con coste igual al de transporte desde el proveedor hacia el demandante.

La figura 7.110 ilustra el modelo en red capacitada de nuestra instancia ejemplo.

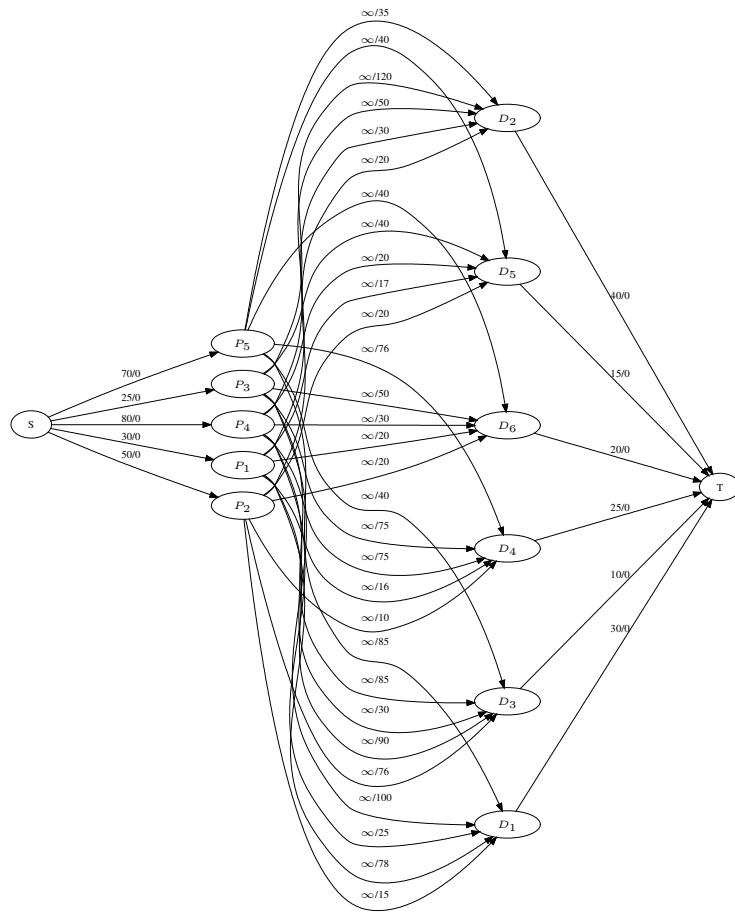


Figura 7.110: Red que modeliza el problema del transporte según la tabla anterior

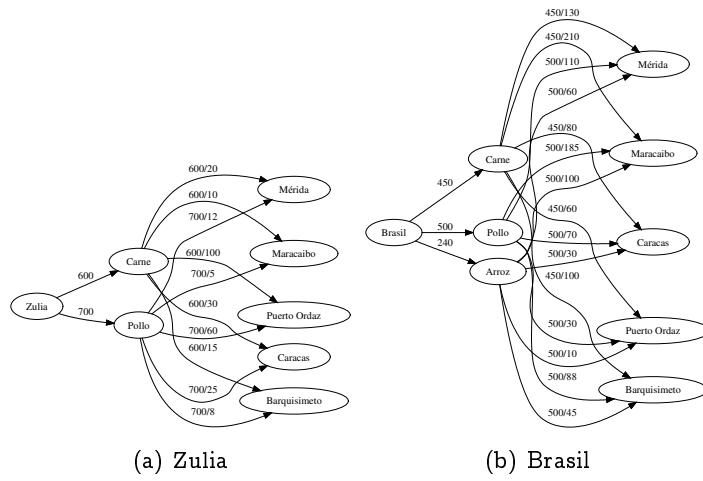


Figura 7.111: Relaciones de coste de transporte para algunos proveedores y demandantes

Según las relaciones de provisión y demanda expresadas en la figura 7.109 (Pág. 803), hay varias maneras de plantear el transporte. Por ejemplo, podríamos mirar a los proveedores y obtener, para el caso de Zulia y Brasil, las relaciones de coste de transporte que se muestran en la figura 7.111 (Pág. 805). Es importante destacar que estos grafos están

orientados hacia el beneficio de los productores y no del de los demandantes.

Para que un modelo del tipo mostrado en la figura 7.111 (Pág. 805) calcule la distribución en cuestión, es conveniente calcular cada rubro por separado y acotar la demanda de cada ciudad. La carne, en nuestros ejemplos, arrojaría grafos como los de la figura 7.112 (Pág. 806).

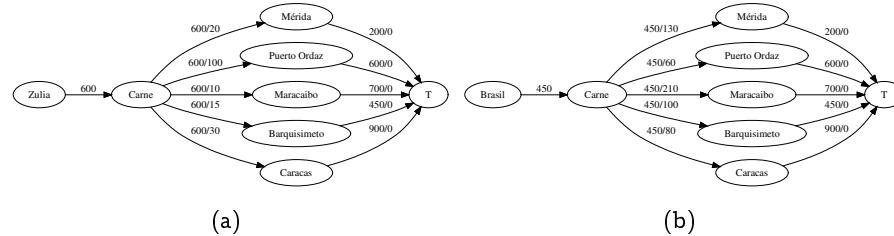


Figura 7.112: Grafos de provisión de carne para Zulia y Brasil. Capacidades desde la ciudad hacia el sumidero representan las demandas de la ciudad

Si miramos el grafo más sencillo, el de Zulia, nos percatamos que este solo productor no basta para cubrir la demanda. En la vida real se apela a solicitar al proveedor un aumento de producción, a buscar otros productores o a ambos esquemas. En términos de una red de transporte, esto equivale a añadir todos los productores posibles y obtener grafos como los de la figura 7.113 (Pág. 806).

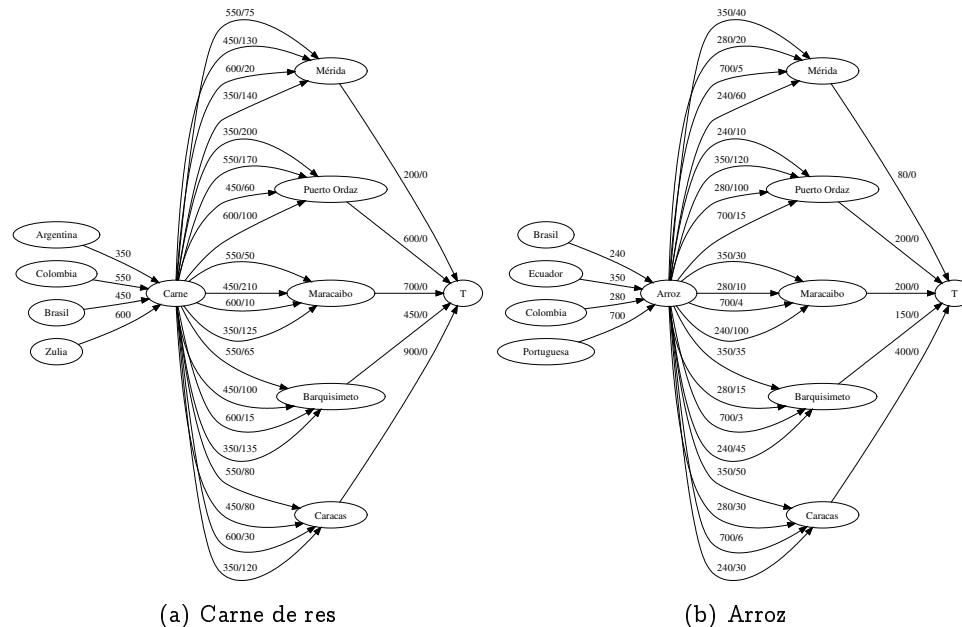


Figura 7.113: Costes de transporte para algunos productos según las relaciones de la red 7.108

Si establecidos todos los proveedores y capacidades posibles no tenemos factibilidad, es decir, que la cantidad provista por los proveedores no satisface la demanda, cual es el caso de la carne, entonces podemos considerar priorizar sumideros, o sea, ciudades. Para eso podemos definir, por ejemplo, un digrafo de prioridades entre las ciudades tal como el de la figura 7.114; por simplicidad, las demandas de cada ciudad son anotadas en el nodo.

En este caso calculamos la demanda total, cual es la suma de demandas de cada ciudad. Análogamente debemos calcular la provisión total, cual es la suma de provisiones. En el

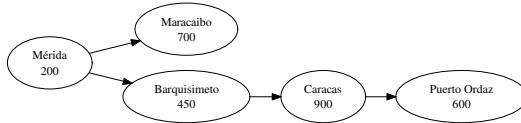


Figura 7.114: Grafo de prioridades de distribución entre las ciudades para la demanda de carne

ejemplo tenemos como demanda total 2900 y provisión total 1950. Ahora comenzamos a suprimir los nodos ciudades y a restar la demanda total desde el último rango topológico hasta el primero y detenemos el procedimiento cuando la demanda sea menor que la provisión. En el caso de la carne (figura 7.113 (Pág. 806)), comenzamos por Puerto Ordaz -último rango topológico-, lo que nos deja una demanda de 2300. Luego restamos Caracas -penúltimo rango topológico-, lo que nos deja una demanda de 1400, la cual es menor que la provisión de 1950. Según esto nos queda la subred de la figura 7.115 (Pág. 807)-a, la cual nos permite calcular la asignación a coste mínimo. En esta subred es esencial restringir la capacidad de una ciudad hacia el sumidero al valor de la demanda, pues de lo contrario, el algoritmo puede llevar más flujo del demandado.

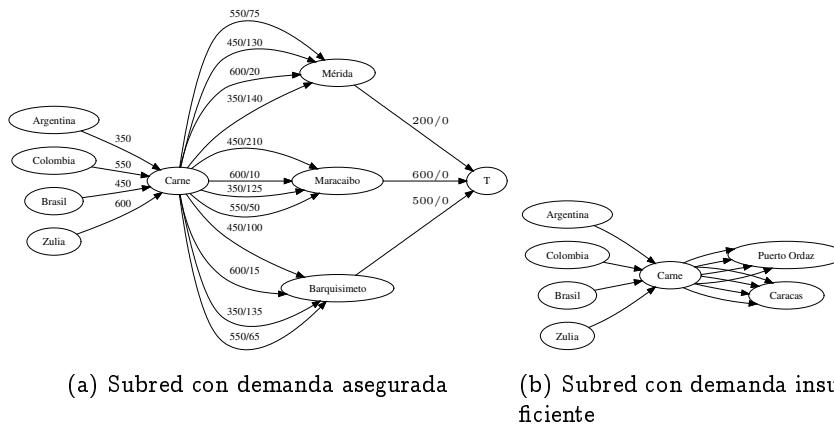


Figura 7.115: Subredes de cálculo según prioridades

Seguidamente tenemos dos alternativas generales para distribuir la provisión restante. La primera es construir una subred con los nodos del primer rango topológico según la prioridad; la segunda es hacer una red con todos los nodos restantes, de manera de dar más minimalidad. Este último criterio nos arroja una subred estructuralmente similar a la de la figura 7.115-b. En cualquiera de las dos alternativas se distribuye toda la provisión restante al mínimo coste.

#### 7.12.4.2 Trasbordo

En el problema del trasbordo se tienen  $m$  proveedores,  $n$  demandantes y  $k$  depósitos o puntos de trasbordo. Similar al problema del transporte, cada proveedor tiene una capacidad de provisión, así como cada demandante una de demanda. La diferencia con el problema de transporte reside en que puede haber nodos depósitos entre los proveedores y demandantes que representan puntos de "trasbordo". Aparte de los costes de envío desde

los proveedores hacia los depósitos y desde los depósitos hacia los demandante, también se cobra por almacenamiento. Extendiendo el ejemplo de la subsección precedente, consideremos la tabla 7.2.

|                |           | Demandantes | D <sub>1</sub> | D <sub>2</sub> | D <sub>3</sub> | D <sub>4</sub> | Total |
|----------------|-----------|-------------|----------------|----------------|----------------|----------------|-------|
| Proveedores    | Provisión | Demanda     | 60             | 70             | 40             | 55             | 270   |
|                |           |             | 78             | 30             | 90             | —              |       |
| P <sub>1</sub> | 60        |             | 15             | 20             | 76             | 10             |       |
| P <sub>2</sub> | 50        |             | 100            | 120            | 85             | —              |       |
| P <sub>3</sub> | 55        |             |                |                |                |                |       |
| Total          | 165       |             |                |                |                |                |       |

| Depósitos      | Coste depósito |    |    |    |    |  | Costes envío |
|----------------|----------------|----|----|----|----|--|--------------|
| W <sub>1</sub> | 30             | 35 | 55 | —  | 35 |  | hacia el     |
| W <sub>2</sub> | 40             | —  | 40 | 20 | 10 |  | demandante   |

Table 7.2: Un ejemplo del problema de trasbordo

El problema es una extensión del de transporte, con los depósitos como nodos intermedios. Notemos que algunas entradas en la tabla no tienen valores, lo cual significa que no existe conexión directa entre el proveedor o depósito y el demandante. Esto puede representar diversas situaciones, desde imposibilidades geográficas hasta embargos.

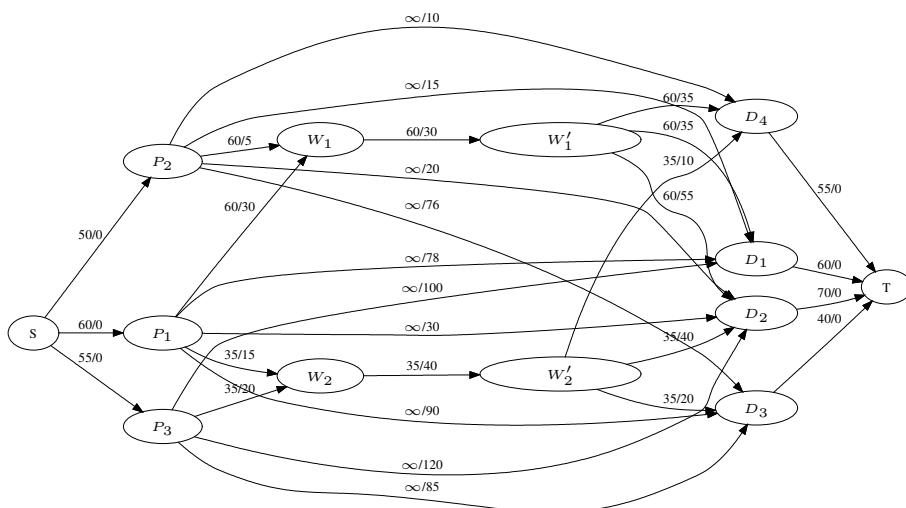


Figura 7.116: Red que modeliza el trasbordo según la tabla anterior

La solución consiste en simplemente dividir los nodos “depósitos” en dos. Un depósito  $w$  se divide en dos nodos  $w' \rightarrow w''$ . Hacia  $w'$  inciden arcos desde los proveedores que modelizan las capacidades de envío y sus costes. Desde  $w'$  emana un sólo arco correspondiente al coste de depósito por unidad. Desde  $w''$ emanan arcos a los demandantes según la capacidad de transporte y coste de envío. La figura 7.116 (Pág. 808) ilustra la representación en red capacitada con costes de la tabla anterior.

El coste por depósito puede omitirse si no aplica, en cuyo caso no es necesario dividir en dos a los nodos depósitos.

Como se ve, el problema de trasbordo es un problema de transporte, con el añadido de que existen nodos intermedios (los trasbordos). En sí, su valor es de abstracción en el sentido de que puede ser más fácil interpretar una situación en términos del trasbordo que en el del transporte o redes capacitadas.

Muchas situaciones mundanas son modelizables en términos de trasbordo. Las rutas

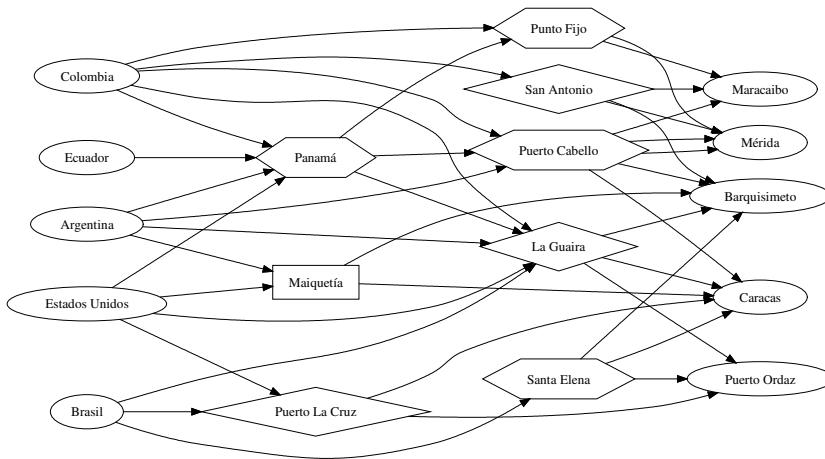


Figura 7.117: Una red hipotética de proveedores, puntos de trasbordo y ciudades demandantes. Los trasbordos terrestres son reconocidos con un rombo, los aéreos con un rectángulo y los marítimos con un hexágono

aéreas, marítimas, importaciones, comercios, etc., constituyen casos paradigmáticos, por ejemplo, la red mostrada en la figura 7.117.

#### 7.12.4.3 Asignación de trabajos

De manera genérica, en el problema de asignación se tienen  $n$  trabajos o proyectos y  $m$  trabajadores. Cada trabajador puede hacer algunos (o todos) trabajos a un coste fijado por él. La meta es maximizar la cantidad de trabajos al mínimo coste. Un ejemplo puede esquematizarse así:

|            | Trabajo | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ |
|------------|---------|-------|-------|-------|-------|-------|-------|
| Trabajador |         |       |       |       |       |       |       |
| $T_1$      | 20      |       | 30    | 10    |       | 20    |       |
| $T_2$      |         | 10    | 25    | 25    |       | 15    |       |
| $T_3$      | 15      | 15    | 30    |       | 40    |       |       |
| $T_4$      |         | 30    | 10    | 20    | 35    |       |       |

El problema se modeliza y resuelve mediante una red capacitada con costes en la cual las capacidades entre los trabajadores y trabajos son unitarias y los costes los correspondientes declarados. El nodo fuente se conecta con los trabajadores a capacidades mayores o iguales que la cantidad de trabajos y a costes nulos. Los trabajos se conectan a un sumidero con capacidades mayores o iguales que la cantidad de trabajadores y costes nulos. Un ejemplo para la tabla anterior es mostrado en la figura 7.118.

La maximización del flujo arroja la asignación que cubre más trabajos. Consecuentemente, la minimización del coste proporciona la máxima asignación a mínimo coste.

Si es aceptable que a un trabajador efectúe más de un trabajo, entonces la capacidad del arco desde el fuente hacia el trabajador se incrementa por la cantidad de trabajos asignables.

#### 7.12.4.4 Camino mínimo

Para calcular el camino mínimo entre un par de nodos  $u$  y  $v$  de un digrafo, conectamos a  $u$  un fuente  $s$  con capacidad mayor o igual al máximo coste entre un camino cualquiera en

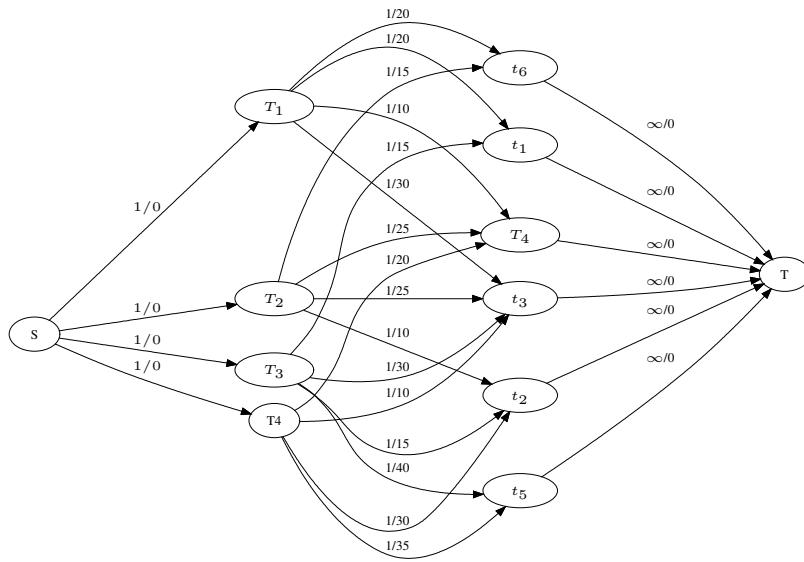


Figura 7.118: Red capacitada del problema de asignación según la tabla anterior

el digrafo. Análogamente, conectamos  $v$  a un sumidero  $t$ . El resto de la red la conforman los arcos del digrafo con capacidad unitaria y coste igual al peso del arco.

Luego, al maximizar el flujo de la red anterior a coste mínimo, los arcos con flujo igual a su capacidad, que son unitarios, conforman el camino mínimo entre  $u$  y  $v$ .

En el caso de un grafo  $G$  el algoritmo es similar, pero previamente calculamos el equivalente dirigido  $\overrightarrow{G}$ .

## 7.13 Programación lineal

Básicamente, un “programa lineal” se compone de los siguientes elementos estándar:

1. Variables de decisión (o simplemente variables)  $x_1, x_2, \dots, x_n$ : representan puntos ajustables cuyos valores son desconocidos al inicio del problema, y desde los cuales se controla la situación de interés.

La meta es encontrar los valores de las variables que maximizan o minimizan una “función objetivo”.

2. Función objetivo: es una expresión matemática que involucra a las variables de decisión para expresar un fin o meta, cuya forma general es:

$$Z = c_1x_1 + c_2x_2 + \dots + c_nx_n$$

3. Restricciones: son expresiones matemáticas que involucran a las variables de decisión

y que expresan límites a las soluciones posibles y expresadas en forma:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &\leq b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &\leq b_2 \\ &\vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &\leq b_m \end{aligned}$$

$$x_1 \geq 0, x_2 \geq 0, \dots, x_n \geq 0$$

Esta forma general de expresar un programa lineal se clasifica de “forma estándar”.

Apelemos a un ejemplo clásico. Supongamos, por ejemplo, una empresa productora de bicicletas con tres plantas y dos tipos de bicicleta: de montaña y de ruta, a precios de venta de 2000 y 1400 respectivamente. Los cuadros de las bicicletas se fabrican en la planta 1, los componentes en la planta 2 y el ensamblaje se efectúa en la planta 3. La planta que fabrica los cuadros puede producir al mes 70 cuadros de montaña y 30 de ruta, pero por restricciones en su línea de soldadura puede fabricar a lo sumo 80 cuadros al mes. Análogamente, la planta 2 puede producir al mes 40 componentes para bicicleta de montaña y 90 de ruta, pero por restricciones de su proceso de cromado, el máximo de componentes que puede fabricar al mes es de 100. Finalmente, la planta 3 (la de ensamblaje) puede ensamblar por mes 50 bicicletas montañeras y 60 de ruta; sin embargo, también por restricciones de mano de obra, la planta no puede ensamblar más de 80 bicicletas al mes.

| Planta               | Bici montaña | Bici Ruta | Capacidad |
|----------------------|--------------|-----------|-----------|
| 1 (cuadros)          | 70           | 30        | 80        |
| 2 (componentes)      | 40           | 90        | 100       |
| 3 (ensamblaje)       | 50           | 60        | 80        |
| Precio por bicicleta | 2000         | 1400      |           |

Table 7.3: Tabla resumen de costes y restricciones para el ejemplo de fabricación de bicicletas

Para restringir que el afán lucrativo lleve a la empresa a cometer injusticias, el Gobierno impone a la empresa la condición de que por cada bicicleta de ruta que produzca la empresa tiene que producir al menos 0,8 de montaña.

Así las cosas, se desea conocer las cantidades de bicicletas de cada tipo que se deben fabricar al mes para maximizar la ganancia. Si  $x_1$  representa la cantidad de bicicletas de montaña y  $x_2$  las de ruta, entonces, asumiendo que las bicicletas serían vendidas en su totalidad, se plantea la siguiente función objetivo:

$$\text{maximizar } Z = 2000x_1 + 1400x_2 = 10x_1 + 7x_2 \quad (7.31)$$

Condicionado a las siguientes restricciones:

$$\begin{aligned} 70x_1 + 30x_2 &\leq 80 \Rightarrow 7x_1 + 3x_2 \leq 8 \\ 40x_1 + 90x_2 &\leq 100 \Rightarrow 4x_1 + 9x_2 \leq 10 \\ 50x_1 + 60x_2 &\leq 80 \Rightarrow 5x_1 + 6x_2 \leq 8 \\ 0,8x_1 > x_2 &\Rightarrow 0,8x_1 - x_2 \geq 0 \end{aligned}$$

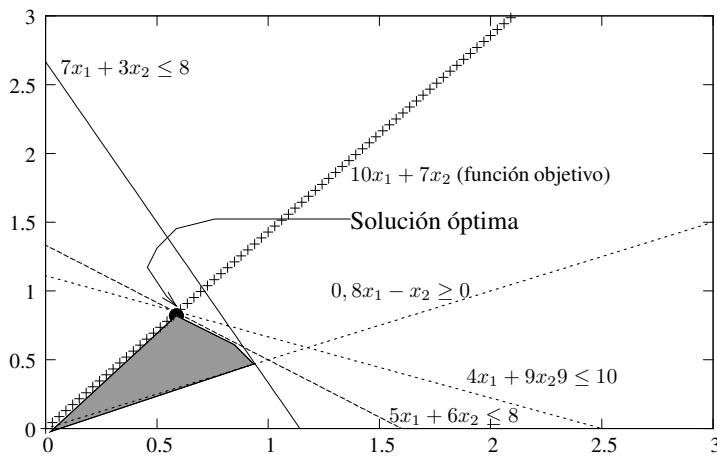


Figura 7.119: Interpretación geométrica del programa lineal y su solución óptima

Se le dice programa lineal porque todas las desigualdades y ecuaciones involucradas son lineales. En efecto, tal como se ilustra en la figura 7.119 (Pág. 812), las rectas correspondientes a las restricciones delinean un polígono (sombreado en la figura) en donde se satisfacen las restricciones. Así, vemos que el máximo de la función objetivo supeditado a las restricciones se cumple en el punto de intersección entre el polígono y la recta de la función objetivo.

Programas lineales de este tipo pueden resolverse mediante un célebre algoritmo llamado simplex [35], descubierto por vez primera por George Dantzig en 1947, pilar de la investigación de operaciones y que revisaremos en § 7.13.4 (Pág. 815). En geometría, un “simplex” es un polígono o politopo  $n$ -dimensional de  $n+1$  vértices. Así, el método hace alusión a que la solución está delimitada por el politopo cuyos vértices corresponden con las intersecciones entre las rectas de las desigualdades correspondientes a las restricciones. La factibilidad de solución depende de que la función objetivo pase o no por el politopo.

### 7.13.1 Forma estándar de un programa lineal

Un programa lineal en forma estándar especifica  $n$  variables de decisión  $x_1, x_2, \dots, x_n$ ,  $n$  números reales  $c_1, c_2, \dots, c_n$  correspondientes a los coeficientes en la función objetivo,  $m$  reales  $b_1, b_2, \dots, b_m$  asociados a las cotas de las restricciones y  $n \times m$  reales  $a_{i,j}$  para  $i = 1, 2, \dots, m$  y  $j = 1, 2, \dots, n$  correspondientes a los coeficientes de las restricciones. Así, genérica y computacionalmente, el fin de un programa lineal es encontrar  $n$  reales  $x_1, x_2, \dots, x_n$  tal que:

$$Z = \sum_{j=1}^n c_j x_j \text{ sea máxima} \quad (7.32)$$

Sujeto a:

$$\sum_{j=1}^n a_{i,j} x_j \leq b_i \text{ para } i = 1, 2, \dots, m \quad (7.33)$$

$$x_j \geq 0 \text{ para } j = 1, 2, \dots, n \quad (7.34)$$

La idea de la forma estándar es una homogeneidad necesaria para una solución computacional. Hay diversas maneras de expresar un programa lineal, las cuales varían según la circunstancia y que pueden contravenir la forma estándar. En lo que sigue, clasificaremos las posibles contravenciones y las maneras en que éstas se solventan para lograr la forma estándar.

#### 7.13.1.1 Minimización de la función objetivo

En este caso, la función objetivo se transforma a la maximización de la negación. Es decir, si la función objetivo es minimizar  $Z$ , entonces maximizar  $Z' = -Z$  es equivalente.

#### 7.13.1.2 Variables sin restricciones de negatividad

Si el modelo permite que una variable  $x_i$  adquiera valores negativos, entonces cada ocurrencia de  $x_i$  se sustituye por  $x'_i - x''_i$  con restricciones  $x'_i \geq 0$  y  $x''_i \geq 0$ . Si la función objetivo contiene un término  $c_i x_i$ , entonces éste se sustituye por  $c_i(x'_i - x''_i)$ . Del mismo modo, todo término  $a_{j,i} x_i$  que aparezca en una restricción  $i$  se reemplaza por  $a_{j,i}x'_i - a_{j,i}x''_i$ .

#### 7.13.1.3 Restricciones de igualdad

A veces se tienen restricciones de igualdad, por ejemplo,  $x_1 + x_2 = 5$ .

Esta clase de restricción  $f(x_1, x_2, \dots, x_n) = b$  se divide en dos restricciones  $f(x_1, x_2, \dots, x_n) \leq b$  y  $f(x_1, x_2, \dots, x_n) \geq b$ .

#### 7.13.1.4 Restricciones mayor o igual

La forma estándar de un restricción es  $\sum_{j=1}^n a_{i,j}x_j \leq b_i$ . Empero, a veces se presentan como mayor o igual, por ejemplo,  $2x_1 + x_2 \geq 2$ .

Una restricción del tipo  $\sum_{j=1}^n a_{i,j}x_j \geq b_i$  se transforma a dos restricciones  $\sum_{j=1}^n a_{i,j}x_j \leq b_i$  respectivamente.

### 7.13.2 Un ejemplo de estandarización

Supongamos el siguiente programa lineal:

$$\text{Minimizar } 2x_1 - x_3$$

Sujeto a:

$$\begin{aligned} 2x_1 + x_2 - 3x_3 &\leq 3 \\ 2x_2 + x_3 &= 4 \\ x_1 + x_2 - 2x_3 &\geq -2 \\ x_1, x_2 &\geq 0 \end{aligned} \tag{7.35}$$

Aplicando las transformaciones referidas el programa resultante es:

$$\text{Maximizar } -2x_1 + (x'_3 - x''_3)$$

Sujeto a:

$$\begin{aligned}
 2x_1 + x_2 + 3x'_3 - 3x''_3 &\leq 3 \\
 2x_2 + x'_3 - x''_3 &\leq 4 \\
 2x_2 + x'_3 - x''_3 &\geq 4 \\
 -x_1 - x_2 + 2x'_3 - 2x''_3 &\leq 2 \\
 x_1, x_2, x'_3, x''_3 &\geq 0
 \end{aligned} \tag{7.36}$$

Por simplicidad, en un programa lineal en su forma estándar sólo se especifica la función objetivo y sus restricciones menor o igual; las restricciones de no negatividad no se especifican.

### 7.13.3 Forma “holgada” de un programa lineal

La forma estándar puede abreviarse como la función objetivo  $Z$  y desigualdad matricial así:

$$Ax \leq b \tag{7.37}$$

Donde  $A$  es la matriz de coeficientes de las restricciones,  $x$  el vector de variables de decisión y  $b$  el vector de cotas de las restricciones. Las restricciones de no negatividad se asumen para cada elemento  $x_i$  y no se ponen por omisión.

Ahora bien, la mayoría de los métodos de solución de programas lineales requieren como entrada un sistema lineal de forma:

$$Ax = b \tag{7.38}$$

Es decir, a la excepción de la no negatividad para cada  $x_i$ , todas las restricciones son de igualdad.

Supongamos un restricción:

$$\sum_{j=1}^n a_{i,j}x_j = b_i \tag{7.39}$$

Ahora introduzcamos una nueva variable  $s$  y planteamos la desigualdad bajo dos restricciones:

$$s = b_i - \sum_{j=1}^n a_{i,j}x_j \tag{7.40}$$

$$s \geq 0 \tag{7.41}$$

La variable  $s$  se califica de “holgada” (“slack”), porque expresa la diferencia entre los dos lados de la desigualdad (7.39).

Para convertir un programa lineal en forma estándar a uno en forma holgada, se usan  $m$  variables adicionales de holgura del tipo:

$$x_{j+k} = b_j - \sum_{j=1}^n a_{i,j}x_j \tag{7.42}$$

$$x_{j+k} \geq 0 \text{ para } k = 1, 2, \dots, m \tag{7.43}$$

Por ejemplo, el programa en forma estándar:

$$\begin{aligned} Z &= x_1 - 2x_2 + x_3 \\ x_1 + 2x_2 - x_3 &\leq 4 \\ 2x_1 - 3x_2 - 2x_3 &\leq 3 \\ x_1 + x_2 + 2x_3 &\leq 2 \end{aligned}$$

Se transforma en su forma holgada en:

$$\begin{aligned} Z &= x_1 - 2x_2 + x_3 \\ x_4 &= 4 - x_1 - 2x_2 + x_3 \\ x_5 &= 3 - 2x_1 + 3x_2 + 2x_3 \\ x_6 &= 2 - x_1 - x_2 - 2x_3 \end{aligned}$$

#### 7.13.4 El método simplex

Una buena manera de entender el principio del simplex consiste en plantearlo como la resolución de un sistema de ecuaciones. Consideremos pues el siguiente problema:

$$\begin{aligned} \text{Maximizar: } Z &= 25x_3 + 40x_2 + 30x_1 \\ \text{Sujeto a: } 2x_3 + 4x_2 + 3x_1 &\leq 500 \\ 5x_3 + 2x_2 + 5x_1 &\leq 650 \\ 2x_3 + 6x_2 + x_1 &\leq 820 \end{aligned}$$

El cual al llevarlo a su forma holgada nos queda:

$$\text{Maximizar } z = 25x_3 + 40x_2 + 30x_1 \quad (7.44)$$

$$\text{Sujeto a: } x_4 = -2x_3 - 4x_2 - 3x_1 + 500 \quad (7.45)$$

$$x_5 = -5x_3 - 2x_2 - 5x_1 + 650 \quad (7.46)$$

$$x_6 = -2x_3 - 6x_2 - x_1 + 820 \quad (7.47)$$

A un sistema de ecuaciones de este tipo suele llamársele “diccionario”. En este estado, el valor de la función objetivo es 0 correspondiente al vértice  $(0, 0, 0)$ , el cual es una solución válida respecto al conjunto de restricciones, pero no máxima. Comencemos por aumentar el valor de  $Z$  despejando a  $x_2$  de algunas de las restricciones. Para aumentar las posibilidades de circunscribirse a las restricciones debemos escoger la más severa. Para eso evaluamos el impacto que tendría el aumento de valor de  $x_2$ , que tiene el coeficiente más alto en  $Z$  (40), en cada una de las restricciones:

$$x_4 \geq 0 \Rightarrow -4x_2 + 500 \geq 0 \Rightarrow x_2 < 125$$

$$x_5 \geq 0 \Rightarrow -2x_2 + 650 \geq 0 \Rightarrow x_2 < 325$$

$$x_6 \geq 0 \Rightarrow -6x_2 + 820 \Rightarrow x_2 < \frac{410}{3} \approx 136.67$$

Así, despejamos  $x_2$  de (7.45), pues esta restricción es la que más nos acota su valor:

$$x_2 = -\frac{x_4}{4} - \frac{x_3}{2} - \frac{3x_1}{4} + 125 \quad (7.48)$$

Ahora reemplazamos esta ecuación en el diccionario precedente y obtenemos:

$$\text{Maximizar: } Z = -10x_4 + 5x_3 + 5000 \quad (7.49)$$

$$\text{Sujeto a: } x_2 = -\frac{x_4}{4} - \frac{x_3}{2} - \frac{3x_1}{4} + 125 \quad (7.50)$$

$$x_5 = \frac{x_4}{2} - 4x_3 - \frac{7x_1}{2} + 400 \quad (7.51)$$

$$x_6 = \frac{3x_4}{2} + x_3 + \frac{7x_1}{2} + 70 \quad (7.52)$$

Aquí el sistema se encuentra en el punto  $(0, 125, 0)$ , el cual da una ganancia en  $Z$  de 5000. Examinando  $Z$  en este nuevo diccionario vemos que la única posibilidad de aumentarla es por la variable  $x_3$ , pues la otra restante es negativa. Así, seleccionamos a  $x_3$  como la variable a aumentar y la despejamos de la restricción más severa, la cual es (7.51). Esto nos da el siguiente diccionario:

$$\text{Maximizar: } Z = -\frac{5x_5}{4} - \frac{75x_4}{8} - \frac{35x_1}{8} + 5500 \quad (7.53)$$

$$\text{Sujeto a: } x_2 = \frac{x_5}{8} - \frac{5x_4}{16} - \frac{5x_1}{16} + 75 \quad (7.54)$$

$$x_3 = -\frac{x_5}{4} + \frac{x_4}{8} - \frac{7x_1}{8} + 100 \quad (7.55)$$

$$x_6 = -\frac{x_5}{4} + \frac{13x_4}{8} + \frac{21x_1}{8} + 170 \quad (7.56)$$

Al ser todos los coeficientes de  $Z$  negativos podemos concluir que no es posible aumentarla más. Consiguientemente,  $Z$  es máxima, con valor 5500, correspondiente al vértice  $(0, 75, 100)$ .

El proceso que acabamos de exemplificar puede generalizarse a una cantidad genérica de variables y automatizarse. Para eso, en primera instancia, definimos la clase Simplex<T>, la cual reside en el archivo *<Simplex.H 816a>*, y que a grandes rasgos se define como sigue:

816a

*<Simplex.H 816a>*≡

```
template <typename T> class Simplex
{
public:
 enum State { Not_Solved, Solving, Unbounded, Solved, Unfeasible };
 <Miembros privados de Simplex 818a>
 <Miembros públicos de Simplex 816b>
};
```

La clase Simplex es un “solucionador” de programas lineales especificados en forma estándar, cuya cantidad de variables debe indicarse en tiempo de construcción:

816b

*<Miembros públicos de Simplex 816b>*

(816a) 816c▷

```
Simplex(int n)
```

n es la cantidad de variables de decisión del sistema, las cuales se enumeran desde el 0 hasta  $n - 1$ . El valor de n puede consultarse mediante la primitiva `get_num_vars()`.

#### 7.13.4.1 Definición de la función objetivo

Cuando se instancia un sistema, los coeficientes de la función objetivo son nulos y no hay restricciones. Así, su especificación se remite a indicar sus coeficientes mediante:

816c

*<Miembros públicos de Simplex 816b>*+=

(816a) <816b 817a>

```
void put_objective_function_coef(int i, const T & coef)
```

Donde  $i$  es el número de la variable y  $\text{coef}$  su coeficiente en la función objetivo.

#### 7.13.4.2 Definición de las restricciones

Hay varios esquemas para definir las restricciones. El primero, cual parece ser el más usual, consiste en indicar al sistema la existencia de una restricción:

817a  $\langle \text{Miembros públicos de Simplex 816b} \rangle + \equiv \quad (816a) \triangleleft 816c \ 817b \triangleright$   
 $T * \text{put_restriction}(T * \text{coefs} = \text{NULL})$

Una llamada a `put_restriction()` crea en el sistema una nueva restricción en forma estándar con coeficientes nulos. La rutina retorna un puntero a un arreglo de coeficientes de la restricción enumerados entre  $0.. \text{num\_var}$ . La obtención de este puntero da referencia directa a cada uno de los coeficientes, de manera tal que se puedan especificar o modificar.

Es posible obtener el mismo puntero mediante:

817b  $\langle \text{Miembros públicos de Simplex 816b} \rangle + \equiv \quad (816a) \triangleleft 817a \ 821a \triangleright$   
 $T * \text{get_restriction}(\text{int rest_num})$

`get_restriction(i)` retorna un puntero al  $i$ -ésimo arreglo restricción.

También es posible pasar directamente los valores de los coeficientes en un arreglo mediante una llamada a `put_restriction(ptr)`, donde `ptr` es el puntero al arreglo de coeficientes.

#### 7.13.4.3 La estructura de datos

Una clave crucial en la automatización del simplex reside en el empleo de una adecuada estructura de datos. No es difícil percibirse de que el método es bastante reminiscente a resolver un sistema lineal de ecuaciones. En ese sentido, sus métodos de resolución son aplicables para el simplex.

Consideremos como ejemplo el siguiente sistema en forma estándar:

$$Z = 40x_0 + 50x_1 + 60x_2 + 30x_3$$

Sujeto a:

$$\begin{aligned} 2x_0 + x_1 + 2x_2 + 2x_3 &\leq 205 \\ x_0 + x_1 + 3x_2 + x_3 &\leq 205 \\ x_0 + 3x_1 + 4x_2 &\leq 255 \\ 3x_0 + 2x_1 + 2x_2 + 2x_3 &\leq 250 \end{aligned}$$

El problema lineal se representan mediante una matriz  $m \times (m+1)$ , donde  $n$  (`num_var`) es la cantidad de variables de decisión y  $m$  (`num_rest`) la de restricciones. La primera fila contiene a  $Z - f(x_0, x_1, \dots, x_{n-1})$  y el resto las restricciones en forma holgada. Así, para nuestro ejemplo, la matriz resultante se define como sigue:

$$\left( \begin{array}{ccccccccc} -40.00 & -50.00 & -60.00 & -30.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 2.00 & 1.00 & 2.00 & 2.00 & 1.00 & 0.00 & 0.00 & 0.00 & 205.00 \\ 1.00 & 1.00 & 3.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 205.00 \\ 1.00 & 3.00 & 4.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 255.00 \\ 3.00 & 2.00 & 2.00 & 2.00 & 0.00 & 0.00 & 0.00 & 1.00 & 250.00 \end{array} \right)$$

Según la cantidad de variables que tenga el sistema, es muy importante, a efectos de economía de memoria, que la matriz sea esparcida. En ese sentido, la clase Simplex<T> emplea una clase llamada DynMatrix<T>, la cual está fundamentada en DynArray<T> § 2.1.4 (Pág. 34), lo que posibilita el ahorro de celdas de memoria para elementos nulos.

#### 7.13.4.4 Selección del pivote

Una vez construida la matriz inicial se procede iterativamente a seleccionar el elemento pivote y a pivotear hasta que todos los coeficientes de la función objetivo -los de la primera fila- devengan positivos.

El primer paso consiste en determinar la columna, la cual es la del menor coeficiente negativo de la función objetivo, o sea, el coeficiente por el cual hay mayor posibilidad de aumentar a Z:

818a *(Miembros privados de Simplex 818a)≡* (816a) 818b▷

```
int compute_pivot_col() const
{
 T minimum = numeric_limits<T>::max();
 int p = -1;
 for (int i = 0, M = num_var + num_rest; i < M; i++)
 {
 const T & c = m->read(0, i);
 if (c < minimum)
 {
 p = i;
 minimum = c;
 }
 }
 return minimum >= 0 ? -1 : p;
}
```

Si no hay coeficientes negativos, entonces el sistema ya se encuentra en una solución óptima y se retorna -1. De lo contrario se retorna el índice de la columna del pivote, el cual llamamos p.

Para determinar la fila del pivote calculamos el menor radio positivo  $m(i, p)/m(i, n+m)$ , lo cual nos proporciona la restricción más severa:

818b *(Miembros privados de Simplex 818a)+≡* (816a) ◁818a 819▷

```
int compute_pivot_row(int p) const
{
 int q = -1;
 T min_ratio = numeric_limits<T>::max();
 for (int i = q + 1, M = num_var + num_rest; i <= num_rest; i++)
 {
 const T val = m->read(i, M);
 if (val < 0)
 continue;

 const T den = m->read(i, p);
 if (den <= 0)
 continue;
```

```

 const T ratio = val / den;
 if (ratio < min_ratio)
 {
 q = i;
 min_ratio = ratio;
 }
}
return q;
}

```

Si no se encuentra ningún radio positivo, entonces se trata de un sistema “ilimitado”, es decir, que no tiene un politopo definido (lo que sería un error de modelizado) y se retorna -1. De lo contrario se retorna el índice de la fila del pivote llamado q.

Para “pegar” las dos rutinas anteriores empleamos un método explícito que determine el elemento pivote:

819 *(Miembros privados de Simplex 818a)*+≡ (816a) ↳ 818b 820▷

```

State select_pivot(int & p, int & q)
{
 const int col = compute_pivot_col();
 if (col == -1)
 return state = Solved;

 const int row = compute_pivot_row(col);
 if (row == -1)
 return state = Unbounded;

 p = row;
 q = col;

 return state = Solving;
}

```

Originalmente, antes de la primera llamada a `select_pivot(p, q)`, el sistema se encuentra en el estado `Not_Solved`.

Así para el programa y la matriz ejemplos, la rutina anterior identifica el siguiente pivote (encerrado en círculo):

$$\left( \begin{array}{cccccccc} -40.00 & -50.00 & -60.00 & -30.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 2.00 & 1.00 & 2.00 & 2.00 & 1.00 & 0.00 & 0.00 & 0.00 & 205.00 \\ 1.00 & 1.00 & 3.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 205.00 \\ 1.00 & 3.00 & 4.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 255.00 \\ 3.00 & 2.00 & 2.00 & 2.00 & 0.00 & 0.00 & 0.00 & 1.00 & 250.00 \end{array} \right)$$

El cual se corresponde con la columna 2 cuyo mínimo coeficiente de la función objetivo es -60 y a la fila 3 correspondiente al mínimo radio  $\min(205/2, 205/3, 255/4, 250/2) = 255/4$ .

#### 7.13.4.5 Pivoteo

El pivoteo consiste en calcular los nuevos coeficientes de la matriz con el siguiente criterio:

1. Para la fila del pivote p: cada elemento de esta fila se divide entre  $m(p, q)$ .

Este paso es equivalente a despejar la variable  $x_q$  de la restricción  $p$ .

2. Para el resto de las filas: sea  $m'(p)$  la fila del pivote transformada según el paso anterior. Cada fila  $m(i)$  se calcula como  $m(i) = m(i) - m(i, p) \times m'(p)$ .

Vistos por filas, estos pasos equivalen a evaluar  $Z$  y las restricciones con la adición de la variable  $x_q$ .

El proceso anterior se puede codificar como sigue:

```
820 <Miembros privados de Simplex 818a>+≡ (816a) ◁819 821b▷
void to_pivot(size_t p, size_t q)
{
 const int M = num_var + num_rest; // cantidad de columnas
 const T pivot = m->read(p, q);
 for (int j = 0; j <= M; j++) // fila del pivote
 if (j != q)
 m->write(p, j, m->read(p, j) / pivot);

 m->write(p, q, 1);

 for (int i = 0; i <= num_rest; i++) // resto de las filas
 for (int j = 0; j <= M; j++)
 if (i != p and j != q)
 m->write(i, j, m->read(i, j) - m->read(i, q)*m->read(p, j));

 for (int i = 0; i <= num_rest; i++) // col de pivote en 0 salvo q
 if (i != p)
 m->write(i, q, 0);
}
```

El pivoteo es la misma operación sobre la cual se fundamenta un método muy popular de resolución de sistemas lineales de ecuaciones llamado de Gauss-Jordan.

Para nuestra matriz ejemplo, el valor inicial de la función objetivo es 0, el cual se corresponde con el vértice del simplex  $(0, 0, 0, 0)$ . Al hacer el primer pivoteo se incrementa el valor de la función objetivo y hace, a excepción de la fila pivote, todos los elementos de su columna cero. Así, para  $p = 2$  y  $q = 3$ , el pivoteo produce:

$$\left( \begin{array}{ccccccc|c}
-25.00 & -5.00 & 0.00 & -30.00 & 0.00 & 0.00 & 15.00 & 0.00 & 3825.00 \\
1.50 & -0.50 & 0.00 & 2.00 & 1.00 & 0.00 & -0.50 & 0.00 & 77.50 \\
0.25 & -1.25 & 0.00 & 1.00 & 0.00 & 1.00 & -0.75 & 0.00 & 13.75 \\
0.25 & 0.75 & 1.00 & 0.00 & 0.00 & 0.00 & 0.25 & 0.00 & 63.75 \\
2.50 & 0.50 & 0.00 & 2.00 & 0.00 & 0.00 & -0.50 & 1.00 & 122.50
\end{array} \right)$$

La operación elimina la columna 2 y lleva el valor de la función objetivo a 3825, correspondiente al vértice del simplex  $(0, 0, 63.75, 0)$ . Continuando con el cálculo aparece como nuevo pivote  $p = 3$  (del  $-30$ ) y  $q = 2$  (de  $\min(77.5/2, 13.75, 122.5/2)$ ), lo que nos arroja:

$$\left( \begin{array}{ccccccc|c}
-17.50 & -42.50 & 0.00 & 0.00 & 0.00 & 30.00 & -7.50 & 0.00 & 4237.50 \\
1.00 & 2.00 & 0.00 & 0.00 & 1.00 & -2.00 & 1.00 & 0.00 & 50.00 \\
0.25 & -1.25 & 0.00 & 1.00 & 0.00 & 1.00 & -0.75 & 0.00 & 13.75 \\
0.25 & 0.75 & 1.00 & 0.00 & 0.00 & 0.00 & 0.25 & 0.00 & 63.75 \\
2.00 & 3.00 & 0.00 & 0.00 & 0.00 & -2.00 & 1.00 & 1.00 & 95.00
\end{array} \right)$$

Nos encontramos en el vértice  $(0, 0, 63.75, 13, 75)$  con valor de función objetivo 4237.5. Pivoteamos con  $m(1,1)$ , lo cual nos produce:

$$\left( \begin{array}{ccccccccc} 3.75 & 0.00 & 0.00 & 0.00 & 21.25 & -12.50 & 13.75 & 0.00 & 5300.00 \\ 0.50 & 1.00 & 0.00 & 0.00 & 0.50 & -1.00 & 0.50 & 0.00 & 25.00 \\ 0.88 & 0.00 & 0.00 & 1.00 & 0.62 & -0.25 & -0.12 & 0.00 & 45.00 \\ -0.12 & 0.00 & 1.00 & 0.00 & -0.38 & 0.75 & -0.12 & 0.00 & 45.00 \\ 0.50 & 0.00 & 0.00 & 0.00 & -1.50 & 1.00 & -0.50 & 1.00 & 20.00 \end{array} \right)$$

Es decir, el punto  $(0, 25, 45, 45)$  con valor de función objetivo de 5300. Nos falta un último pivoteo de  $m(4,4)$ , cuyo resultado final es:

$$\left( \begin{array}{ccccccccc} 10.00 & 0.00 & 0.00 & 0.00 & 2.50 & 0.00 & 7.50 & 12.50 & 5550.00 \\ 1.00 & 1.00 & 0.00 & 0.00 & -1.00 & 0.00 & 0.00 & 1.00 & 45.00 \\ 1.00 & 0.00 & 0.00 & 1.00 & 0.25 & 0.00 & -0.25 & 0.25 & 50.00 \\ -0.50 & 0.00 & 1.00 & 0.00 & 0.75 & 0.00 & 0.25 & -0.75 & 30.00 \\ 0.50 & 0.00 & 0.00 & 0.00 & -1.50 & 1.00 & -0.50 & 1.00 & 20.00 \end{array} \right)$$

O sea, el vértice  $(0, 45, 30, 50)$  para un valor máximo de 5550.

El proceso anterior se automatiza mediante el siguiente método:

821a  $\langle$  Miembros públicos de Simplex 816b  $\rangle + \equiv$  (816a)  $\triangleleft$  817b 822  $\triangleright$

```
State solve()
{
 for (int i, j; true;) {
 const Simplex<T>::State state = select_pivot(i, j);
 if (state == Simplex<T>::Unbounded or state == Simplex<T>::Solved)
 return state;

 to_pivot(i, j);
 }
}
```

Como podemos notar del algoritmo, el valor final de una variable  $i$  se determina mirando su columna -la cual debe contener ceros y un uno- y examinando el valor de  $m(i, num\_rest)$ , donde  $j$  es el índice del valor uno en la columna. Esto se puede efectuar con la siguiente rutina:

821b  $\langle$  Miembros privados de Simplex 818a  $\rangle + \equiv$  (816a)  $\triangleleft$  820

```
T find_value(const size_t j) const
{
 T ret_val = 0.0;
 for (int i = 1, count = 0; i < num_rest; i++)
 {
 const T & value = m->read(i, j);
 if (value == 0.0)
 continue;

 if (value == 1.0)
 if (count++ == 0)
 ret_val = m->read(i, num_var + num_rest);
 else
 return 0.0;
```

```

 else
 return ret_val;
 }
 return ret_val;
}

```

Así, una vez resuelto el sistema podemos cargar la solución mediante:

822  $\langle \text{Miembros públicos de Simplex 816b} \rangle + \equiv$  (816a)  $\triangleleft 821a$

```

void load_solution()
{
 for (int j = 0; j < num_var; j++)
 solution[j] = find_value(j);
}

```

### 7.13.5 Conclusión sobre la programación lineal

En esta sección hemos presentado un panorama muy general y superficial sobre la programación lineal y uno de sus métodos de resolución más populares: el simplex.

El algoritmo simplex es extremadamente difícil de analizar. Matemáticamente es un algoritmo exponencial y se han encontrado casos que degeneran la ejecución a este tipo de rendimiento. A pesar de ello, en la práctica el promedio de ejecución del simplex es espectacular, aunque aún no se comprende bien la razón.

A pesar de la aparición de nuevos métodos de optimización (véase sección de referencias § 7.15 (Pág. 830)) por su simplicidad de implementación y rendimiento, el simplex sigue siendo una de las escogencias principales como solucionador de programas lineales.

La importancia de la programación lineal subyace sobre el hecho de que existe una íntima conexión entre el simplex, así como otros métodos de optimización, y las redes de flujo. Consecuentemente, esta conexión es extensiva a los grafos en general. A los aspectos de este vínculo nos abocaremos brevemente en la siguiente sección.

## 7.14 Redes de flujo y programación lineal

Una red de flujo puede expresarse como un programa lineal, resoluble mediante cualquiera de los métodos de la programación lineal.

### 7.14.1 Conversión de una red capacitada de costes a un programa lineal

Consideremos una representación matricial para una red capacitada con costes de  $n$  nodos. Cada nodo se señala por un índice matricial y el arco por su entrada en la matriz. Así, el problema de flujo máximo a coste mínimo puede plantearse, genéricamente, como uno de optimización, del siguiente modo:

$$\text{Minimizar } Z = \sum_{i=1}^n \sum_{j=1}^n c_{i,j} x_{i,j} \quad (7.57)$$

Sujeto a los conjuntos de restricciones:

$$\sum_{j=1}^n x_{i,j} - \sum_{k=1}^n x_{k,i} = b_i, \quad i = 1, 2, \dots, n \quad (7.58)$$

$$x_{i,j} \geq 0, \quad i, j = 1, 2, \dots, n \quad (7.59)$$

Por cada arco  $i \rightarrow j$ , su valor de flujo representa una variable de decisión  $x_{i,j}$ . La suma  $Z$ , correspondiente al valor de flujo de la red, representa la función objetivo. Por cada nodo  $i$ , el valor  $b_i$ , correspondiente a la diferencia entre el flujo de entrada y el de salida, representa una restricción. A excepción del fuente y del sumidero<sup>33</sup>,  $b_i = 0$ , mientras que para el fuente y sumidero  $b_s$  y  $b_t$  respectivamente. Finalmente, puesto que el flujo no puede ser negativo, cada variable  $x_{i,j} \geq 0$ .

Vemos entonces un mapeo directo desde una red de costes a un programa lineal. Tal mapeo es de enorme importancia por dos razones vinculadas entre si:

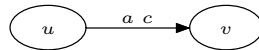
1. Cualquier problema de flujo máximo, con o sin coste mínimo, puede resolverse mediante el simplex u otro método de la programación lineal.
2. Algunos problemas de programación lineal pueden resolverse mediante una red capacitada.

La algorítmica para resolver problemas con redes capacitadas es con creces más eficiente que la de la programación lineal, pero no todo programa lineal puede expresarse como una red capacitada.

### 7.14.2 Redes generalizadas

Hasta el presente, para todos los modelos de redes capacitadas hemos asumido la condición de conservación de flujo definida en 7.9-b pág. 709. Redes que no satisfagan aquella condición son llamadas, bajo ciertas restricciones, "redes generalizadas".

En las redes generalizadas los arcos poseen un factor de ganancia o pérdida de flujo. En el modelo pictórico, un factor de ganancia (o de pérdida)  $a$  entre un arco  $u \rightarrow v$  se expresa multiplicándolo por la capacidad  $c$  del arco:



Si  $a = 1$ , entonces se trata de un nodo normal. Si  $0 < a < 1$ , entonces se trata de un factor de pérdida, por ejemplo, un flujo de corriente eléctrica que pasa por una resistencia o un flujo de agua que sufre alguna pérdida por evaporación<sup>34</sup>.

Si  $a > 1$ , entonces, se trata de una ganancia de flujo, lo que en una primera mirada parece físicamente improbable. Existen empero casos no tan excepcionales, ejemplos, redes por donde circulan "capitales económicos", redes de "agentes biológicos" (i.e. bacterias), aguas a la intemperie (que aumentan por lluvias), etc.

Un factor cero no tiene sentido, pues equivale a la ausencia de arco.

Aunque matemáticamente posible, un factor negativo tampoco tiene mucho sentido físico, pues modeliza una especie de retroceso de flujo, el cual probablemente sea modelizable desde la perspectiva topológica de la red.

<sup>33</sup>Recuérdese que la red puede reducirse a un solo fuente y un solo sumidero.

<sup>34</sup>En algunos casos no se trata en si de un factor de pérdida, sino de una reducción a la capacidad, la cual modelizarse mediante un nodo intermedio  $w$ :



En términos de un programa lineal, para un nodo cualquiera  $v$ , “la conservación” del flujo puede expresarse como:

$$\sum_{e \in \text{IN}(v)} f(e) - \sum_{e \in \text{OUT}(v)} a_e f(e) = b_e \quad (7.60)$$

Notemos que la diferencia de (7.60) con la conservación del flujo expresada en (7.11) (def. 7.9-b pag. 709) es un factor multiplicativo  $a_e$  en cada arco  $e$ .

Redes generalizadas pueden resolverse a través de un algoritmo llamado “simplex de red”.

### 7.14.3 Capacidades acotadas

En este tipo de red el flujo de los arcos está acotado por los extremos: uno mínimo y uno máximo, este último correspondiente a la capacidad.

En el modelo pictórico, cada arco contiene dos valores: el flujo mínimo requerido y el máximo permitido, los cuales suelen expresarse entre corchetes como en la figura 7.120-a.

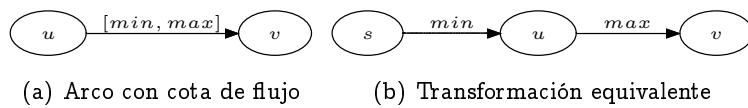
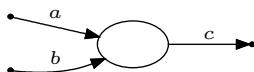


Figura 7.120: Arco con cotas de flujo y su transformación

La cota inferior puede interpretarse como una demanda de flujo en el nodo  $u$ , mientras que la superior es la capacidad del arco. Así, según lo visto en § 7.11.3 (Pág. 774), un arco de este tipo es equivalente al de la figura 7.120-b.

### 7.14.4 Redes con restricciones laterales

En algunas ocasiones se tienen restricciones adicionales que involucran varios flujos. Supongamos por ejemplo tres arcos  $a, b, c$ :



y como restricción sobre el flujo de ellos a  $2f(a) - f(b) + 3f(c) \leq 12$ . Una restricción de este tipo se califica de “lateral”. Consecuentemente, redes con esta clase restricciones se tildan de “con restricciones laterales”.

Las restricciones laterales no necesariamente tienen que estar relacionadas a los arcos de un nodo. Podría tratarse de una restricción entre arcos que no estén directamente relacionados, por ejemplo, separados por al menos  $m$  arcos de distancia.

Claramente, las restricciones laterales no encajan en el modelo de red capacitada, pero sí como un programa lineal, lo que podría sugerir que la red se resuelva con un método de la programación lineal. El problema con este enfoque es que se pierde la ganancia en desempeño que un algoritmo de red tiene sobre cualquiera de la programación lineal.

Una manera particular de tratar con restricciones laterales, posible en muchos casos, consiste en “monitorear” la validez de la restricción durante la ejecución de la solución. El problema de esta técnica es que no es generalizable.

Para el caso general, para tratar con restricciones laterales la red se partitiona en bloques (subgrafos) separados por las índoles de sus restricciones mediante un método

llamado “relajación lagrangiana”<sup>35</sup>, el cual no será tratado en este texto. En este caso se consideran dos tipos de bloques: los que contienen restricciones y los que son redes capacitadas. De este modo, los bloques con restricciones laterales se resuelven con métodos de programación lineal, mientras que los de red se solventan con redes de flujo. Las soluciones parciales se concatenan iterativamente hasta que converjan.

#### 7.14.5 Redes multiflujo

Hasta el presente, todos los modelos de red capacitada asumen que las unidades de flujo, o sea, el fluido, son las mismas; entiéndase, por los arcos (tuberías) circula un sólo tipo de fluido<sup>36</sup>.

Hay situaciones en las que por los mismos arcos pueden circular distintas clases de fluido, emanados desde varios fuentes independientes, al menos, tantos como distintos tipos de fluido se tengan. No hay restricción sobre la cantidad de sumideros, y cualquiera de éstos puede recibir cualesquiera de los distintos tipos de flujo.

Una situación tradicional de esta clase de red se presenta con el modelado de tráfico automotor o peatonal. Se pueden considerar, por ejemplo, varias zonas geográficas  $z_1, z_2, \dots, z_n$  como proveedoras de flujo automotor. A una cierta hora pico -las 12pm, por ejemplo-, los automóviles se desplazan desde y hacia las zonas en cuestión a través de vías compartidas. Como puede inferirse, en esta situación, los fluidos en algún momento se encuentran.

Otros ejemplos los constituyen los problemas del transporte (§ 7.12.4.1 (Pág. 804)) y del trasbordo (§ 7.12.4.2 (Pág. 807)), en los cuales se poseen las mismas líneas y almacenes, pero éstos se emplean para transportar y almacenar diversos tipos de bienes.

Hasta el presente sólo se conoce una manera polinomial de resolver este problema: transformarlo a un programa lineal y desde allí solucionarlo.

En ocasiones, a una red multi-flujo pueden encontrársele pedazos (bloques) de un sólo flujo. Así, de la misma manera que para las redes generalizadas, la red puede dividirse en bloques según los distintos tipos de flujo, resolverse y luego pegarse hasta encontrar la solución<sup>37</sup>.

#### 7.14.6 Redes de procesamiento

Una red de procesamiento es una red capacitada especial que sirve para modelizar y estudiar sistemas de producción. Tal red contiene dos clases de nodos. Los normales, en los cuales se satisface la conservación de flujo o son fuentes o sumideros, y los de “procesamiento”, en el sentido de que modelizan un procesamiento que no satisface la conservación del flujo. La figura 7.121 ejemplifica una hipotética red de procesamiento para fabricar botellas de vidrio en la cual se muestran dos nodos de procesamiento.

El nodo “Horno” representa un eventual proceso en el cual se mezcla silicio con oxígeno para obtener cuarzo. Este proceso, por así decirlo, debe satisfacer funciones de “fundido”,

<sup>35</sup>Para detalles sobre este método pueden consultarse [10, 16, 21].

<sup>36</sup>Si hay distintos tubos para cada clase de fluido, entonces se pueden separar tantas redes como tipos de fluidos se tengan. Este fue el caso para el ejemplo del transporte presentado en § 7.12.4.1 (Pág. 804).

<sup>37</sup>Igual que para las redes generalizadas, técnicas para resolver redes multi-flujo pueden consultarse [10, 16, 21, 101].

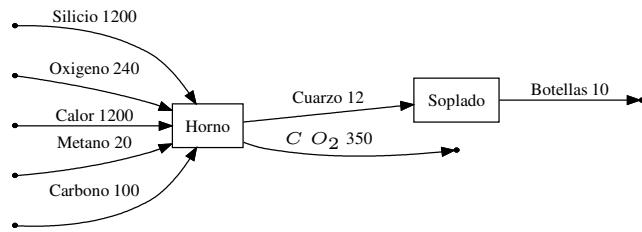
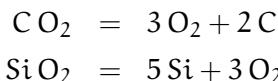


Figura 7.121: Un ejemplo de red de procesamiento

que modelicen los productos resultantes, por ejemplo, el horno podría modelizarse así:



Mientras que el soplado así:

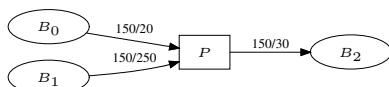
$$12\text{C} = 10 \text{ Botellas} \implies \text{Botellas} = \frac{6}{5}\text{C}$$

Por lo general, una red de procesamiento se representa mediante una red multiflujo con restricciones laterales.

El poder de una red de procesamiento es su versatilidad para la modelización de una muy amplia gama de sistemas ingenieriles, en particular, muchos procesos concernientes a la fabricación de bienes.

#### 7.14.6.1 Cadenas de producción

Supongamos la producción de un cierto bien o la prestación de un cierto servicio. Para elaborar una cierta cantidad del bien se requieren de otros bienes y servicios, los cuales, a su vez, también requieren de otros bienes y servicios. Los bienes y servicios son elaborados por agentes llamados productores. Un productor es nodo de procesamiento del tipo:



A este subgrafo lo llamamos “grafo productor”. En este caso, en un lapso de tiempo fijo para todos los productores, el productor P elabora 150 unidades del bien B<sub>2</sub> que se vende a un precio de 30 por unidad. Para fabricar B<sub>2</sub>, el productor P requiere los bienes B<sub>0</sub> y B<sub>1</sub> a las cantidades y costes señalados en el grafo.

Es posible tener varios productores del mismo bien; podría haber inclusive productores que ante el mismo bien de salida difiriesen de otros en algunos bienes de entrada, lo cual podría deberse, entre otras cosas, a procesos distintos de fabricación u organización.

Si estudiamos los bienes de entrada, entonces podemos identificar sus productores y elaborar un grafo productor para cada uno de ellos. Podemos continuar este proceso para los nuevos bienes que aparezcan hasta que finalmente nos encontraremos con bienes primarios, es decir, aquellos a los cuales no se les puede identificar un productor; bien sea porque son materia prima, bienes importados u otra situación equivalente.

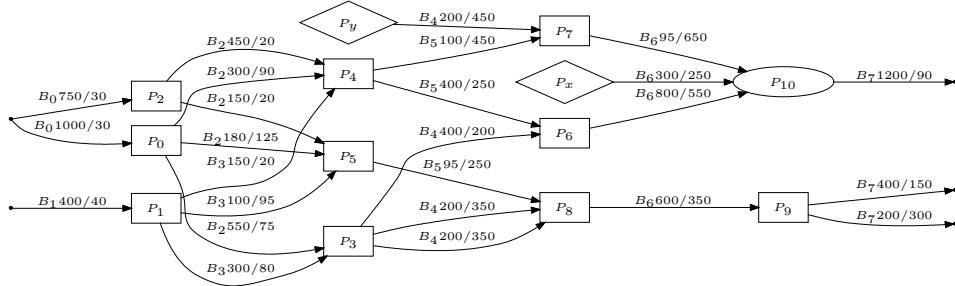


Figura 7.122: Una cadena productiva

A un grafo que represente relaciones entre los productores según uno o más bienes finales le llamamos “cadena productiva”. En este grafo, que relaciona productores, los arcos representan bienes y sus pesos sus cantidades y costes. La figura 7.122 muestra un ejemplo hipotético, cuyos extremos izquierdos representan los bienes primarios y derechos los bienes finales. Según la índole del sistema productivo pueden representar materia prima (minerales básicos, agua, etc.). En nuestro ejemplo, los bienes primarios son  $B_0$  y  $B_1$  respectivamente. Los extremos finales representan el bien final objeto de estudio, el cual llamamos  $B_7$ . Los nodos del grafo representan productores de productos intermedios que entre los bienes primarios  $B_0$  y  $B_1$  y el bien final  $B_7$ <sup>38</sup>.

Para aprehender la importancia de este ejemplo y, con él, la de las redes de procesamiento, planteemos algunos escenarios en los cuales las cadenas productivas ayuden a vislumbrar respuestas:

- Impacto que tendría sobre la producción una sequía: si asumimos por simplicidad que la sequía afecta proporcionalmente a todo el territorio, entonces, examinando cada uno de los grafos productores con arcos de entrada acuíferos podemos restar la disminución. Luego, calculamos el impacto que sobre los bienes de salida tenga la disminución debida a la sequía. Aplicando el criterio y propagándolo a través de las cadenas podemos examinar las consecuencias que una sequía tiene sobre la producción de cada bien.
- Dado un bien específico, determinar cuál es el bien intermedio crítico: para este problema se construye toda la cadena productiva detrás del bien de interés. Luego se calcula el corte mínimo, el cual arroja los arcos cuya capacidad limita la producción del bien de interés. Los productores vinculados a estos arcos son aquellos que hay que garantizar el aumento para que en última instancia se pueda aumentar la producción.
- Se desea conocer cuáles rubros y productores hay que apoyar o proteger para aumentar la producción de algún bien específico. Aquí, aparte de determinar los eslabones críticos, podemos plantear nuevos mapas productivos. Por ejemplo, podríamos simular la construcción de fábricas nuevas de bienes críticos y evaluar su efecto en la producción.

<sup>38</sup>En algunos mundos vinculados a la economía, los eslabones de las cadenas productivas se clasifican en aguas altas, medias y bajas, en paráfrasis a un río con un aserradero y la cadena productiva de la madera. Los bosques madereros se encuentran en las aguas altas del río. Los árboles se cortan y los troncos se echan al río cuya corriente los arrastra hasta el aserradero que se encuentra en las aguas media (o intermedias). El aserradero recibe los troncos y los transforma en productos intermedios (tablas, bloques, etc.) que se distribuyen a diversas industrias tales como los astilleros, mueblerías, constructoras de casas, etc.. Esta metáfora, aún en uso, ilustra la importancia que tiene el cuidado del bosque para la sustentabilidad de la cadena productiva.

Del mismo modo podríamos colocar fábricas del bien de interés y observar, según criterios estocásticos o deterministas (el algoritmo de flujo, por ejemplo) qué sucede con el estado de la producción.

Estos escenarios, entre muchos otros, reflejan la importancia de las redes de procesamiento; como consolidación de la redes de flujo, así como su versatilidad en representación y modelizado.

#### 7.14.7 Conclusión sobre el problema del flujo máximo a coste mínimo

El problema paradigmático de esta sección tiene dos fuentes de estudio. La primera es la de investigación de operaciones; la segunda, la de la algorítmica.

De la investigación de operaciones tenemos el problema genérico de optimización formulado en § 7.13 (Pág. 810) y sus métodos de resolución, entre ellos el simplex (§ 7.13.4 (Pág. 815)). Esta vertiente ha incidido el pensamiento matricial y geométrico.

Durante los años 50 y 60 del siglo XX, la investigación de operaciones tomó la teoría de grafos como un vehículo de modelización. A partir de entonces, se han descubierto conexiones íntimas entre ambos mundos.

La gran ventaja de la investigación de operaciones es su enorme experiencia en modelos genéricos de problemas y su mapeo a situaciones de la vida real; los problemas de transporte (§ 7.12.4.1 (Pág. 804)), de trasbordo (§ 7.12.4.2 (Pág. 807)) y de asignación de trabajos (§ 7.12.4.3 (Pág. 809)), formulados desde la investigación de operaciones, bastante realistas en el mundo productivo, son aplicables a muchas otras situaciones. Ambos problemas fueron planteados en términos de matrices y de grafos. Más general como problema es la programación lineal (§ 7.13 (Pág. 810)), pero de ésta sólo un conjunto es representable -y resoluble- con grafos.

Por su amplia experiencia, un auténtico "investigador de operaciones" es un sujeto a consultar cuando se esté modelizando un problema. Pero el computista también es otro sujeto al que el investigador de operaciones debe consultar a la hora de hacer el modelo. No es común encontrar una persona que conozca y comprenda cabalmente los dos mundos.

Pero el modelo orientado a grafos, a través de la red capacitada, tiene soberbias ventajas sobre la programación lineal. La primera de ellas es la rapidez algorítmica, la cual depara en la obtención más célere de resultados respecto a los enfoques de la programación lineal. La subsección § 7.14.1 (Pág. 822) muestra la manera general de interpretar una red capacitada como un programa lineal. Cada vez que se pueda identificar esta correspondencia es preferible el modelo de red al programa lineal por la sencilla razón de que los órdenes de ejecución de las redes de flujo están alrededor de un orden de magnitud superior a los de la programación lineal. La segunda ventaja es que el carácter gráfico de las redes capacitadas hace más fácil y aprehensible, para la mayoría de las personas que no tengan fuerte formación matemática y abstracta, la modelización e interpretación de situaciones del mundo real. Después de todo, dicen que en este mundo posmoderno está dominado por lo visual. Finalmente, en las redes capacitadas hay una gama más amplia de combinaciones de algoritmos, lo que cual proporciona una holgura importante a la hora de mejorar los tiempos de ejecución.

Por otra parte, también hemos visto la aplicabilidad de las redes capacitadas a otros problemas más pertinentes al mundo de los grafos. El problema del camino mínimo entre un par de nodos, efectivamente tratado en § 7.9 (Pág. 674), también puede resolverse

mediante redes capacitadas (§ 7.12.4.4 (Pág. 809)). Algo parecido ocurre con los puntos de corte estudiados en § 7.5.14 (Pág. 613) y el corte mínimo a mínimo coste, no estudiado explícitamente pero deducible a partir de § 7.11.6.3 (Pág. 789).

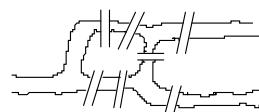
En cierta forma, el modelo de red capacitada es a los grafos, y quizá a la programación lineal, lo que la transformada de Laplace es al cálculo diferencial: otra perspectiva de interpretar un problema, para el cual, en ciertas situaciones, es más simple encontrar soluciones.

## 7.15 Notas bibliográficas

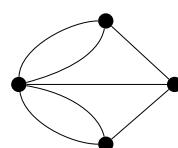
Un grafo, como los estudiamos en este texto, conforma una representación, gráficamente orientada, de una realidad concreta, o sea, de algo presente, de la vida real, tal como un mapa, o de una representación de una abstracción, tal como una relación matemática. Así, los grafos son inherentes al conocimiento humano. De alguna forma u otra están presentes desde tiempos inmemoriales; prácticamente desde los albores del ser humano. Puesto que la idea de grafo es parte del acervo humano desde tiempos milenarios, no parece justo atribuirle alguna autoría particular o individual.

Se han descubierto mapas babilonios que se remontan al 2300 a.C. Así que probablemente problemas como el de la ruta más corta entre dos sitios han sido planteados en distintos tiempos, en lugares diferentes, por diversas culturas.

Ahora bien, sin pretensión de demérito, la historia “oficial” de los grafos se remonta al primer y célebre artículo científico escrito por el matemático suizo Leonard Euler y su estudio sobre un problema citadino conocido como los “Puentes de Königsberg”. Por esta ciudad pasa el río Pregel, del cual existen dos islas enmarcadas en el perímetro de la ciudad. Desde las riberas hacia las islas y entre ellas se cuentan siete puentes distribuidos de la siguiente manera:



Los ciudadanos se preguntaban si había alguna manera de recorrer todos los puentes exactamente una vez; es decir, sin pasar dos veces por el mismo puente. Euler modelizó el asunto mediante el siguiente grafo:



Euler describió la manera general de resolver el problema, cual es hoy conocida bajo el rótulo de “ciclo euleriano”. Desde entonces, a Euler lo consideran el precursor de la teoría de grafos o, más justamente, de la topología.

Nativo de Königsberg fue Gustav Robert Kirchhoff, quien, a tenor de las corrientes y voltajes en redes eléctricas, planteó que:

1. La suma de las corrientes asociadas a los arcos incidentes de un nodo es cero, y
2. La suma de los voltajes asociados a cualquier ciclo es cero.

He aquí una reminiscencia con la condición de conservación de flujo (7.11) (pag. 709) y el teorema 7.16 (pag. 784) de la descomposición de flujo presentado en § 7.11.5 (Pág. 783).

Los recorridos arquetípicos sobre grafos, de profundidad y de amplitud, fueron reportados por primera vez por Robert. E. Tarjan [166].

El problema del árbol abarcador mínimo se conoce desde los años 20 del siglo XX. Boruvka reportó una primera solución en 1926 [23] y, prácticamente, el hoy conocido algoritmo de Prim fue descubierto también, y varias décadas antes, por Jarnik [84].

Como ya hemos indicado, las redes flujo fueron precedidas por los trabajos de Kirchhoff. En este tema resulta interesante considerar el prisma cultural. Encontrar caminos de aumento no sólo es necesario para maximizar flujo, sino también para suplir una demanda de flujo de petróleo en una intrincada red de oleoductos. El teorema del corte mínimo estudiado en § 7.10.6 (Pág. 718), cual dio pie a los sucesivos algoritmos de maximización basados en camino de aumento, fue presentado independientemente, en el mismo año por Ford y Fulkerson [54] y por Elias, Feinstein y Shannon [41]. Posteriormente, Edmonds y Karp [40] demostraron una mejora substancial a los algoritmos primando a la búsqueda en amplitud para encontrar caminos de aumento. Por otro lado, los algoritmos de preflujo fueron precedidos por el trabajo de Karzanov [92]. El preflujo tiene más sentido para maximizar un flujo en una red donde la viscosidad del fluido es muy baja, lo cual es el caso del agua y los sistemas de riego.

Sobre las redes de flujo hay libros importantes, yendo desde clásicos de optimización combinatoria como el Lawler [107] o el Papadimitriou y Steiglitz [138] hasta otros más recientes como el Korte y Vygen [101], el Rao [148] y el Nocedal Wright [133]. También hay textos especializados en las redes de flujo; los más notables son el Ahuja, Magnanti y Orlin [10] y el Bazaraa, Jarvis y Sherali [16].

Puede decirse que la programación lineal, y su célebre método simplex, figuran entre los descubrimientos más importantes del siglo XX. Actualmente no existe organización seria que no abarate sus costes de producción, gestión, servicio, etc. mediante el empleo y resolución de modelos de programación lineal. En ese sentido, el algoritmo simplex, vislumbrado a finales de los años 30 del siglo XX, final y formalmente publicado en 1949 [35], no sólo aún es uno de los principales algoritmos de la programación lineal, sino que se encuentra entre los diez algoritmos más importantes del siglo XX [174].

El 1979, el matemático ruso Leonid Khachiyan publicó un nuevo método de solución cuyo tiempo es polinomial [95]. Su trabajo es un hito matemático que dio inicio a una gama de algoritmos denominados de “de puntos internos” y descubiertos por Narendra Karmarkar en 1984 [90].

En programación lineal se encuentran muy buenos y modernos textos, algunos de ellos previamente citados [133, 101, 148, 16] y otros más especializados, entre los cuales cabe notar el Luenberger y Ye [111].

Sobre grafos en general es menester mencionar el clásico teórico, aunque ya adolescente de falta de descubrimientos importantes, de Harary [70]. A la fecha de esta publicación, a criterio de este autor el mejor texto teórico sobre grafos es el texto de Jungnickel [89]. Un excelente texto entre lo teórico y lo instrumental lo constituye el libro de Gross y Yellen [187]. Para textos instrumentales (algorítmicos), un excelente libro es el de Alan Gibbons [59], y más recientemente el Sedgewick [156].

Algunos de los grafos mostrados en este texto fueron dibujados mediante el muy

buen visualizador Graphviz [42]. Para otros grafos se empleó un programa de dibujado desarrollado por este autor, denominado en la biblioteca graphpic, mediante el cual se dibujaron muchas de las figuras de este capítulo. En el desarrollo de graphpic, el libro de Kozo Sugiyama [165] fue muy útil. Otro texto al respecto es el Battista, Eades, Tamassia y Tollis [13].

## 7.16 Ejercicios

1. Asumiendo que no hay arcos paralelos, ¿cuánta es la cantidad máxima de arcos que puede tener un grafo?
2. Asumiendo que no hay arcos paralelos, ¿cuánta es la cantidad máxima de arcos que puede tener un digrafo?
3. Para el grafo de la figura 7.123:
  - (a) Determine y dibuje un árbol de recorrido en profundidad a partir del nodo A.
  - (b) Determine y dibuje un árbol de recorrido en amplitud a partir del nodo A.
  - (c) Determine y dibuje un árbol abarcador a partir del nodo F.
  - (d) Encuentre el (o los )camino(s) más largo(s).
  - (e) Encuentre el grafo complemento.
  - (f) Encuentre todos los cliques o subgrafos completos.

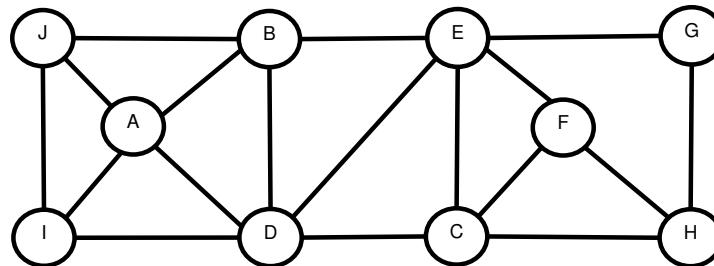


Figura 7.123: Un grafo de ejercicio

4. Plantee desventajas de los diagramas de sagitales para expresar relaciones binarias. Para cada planteamiento, establezca la diferencia con un grafo.
5. Diseñe una rutina que realice el recorrido en profundidad sin utilizar recursión.
6. Diseñe la primitiva `test_for_cycle()` presentada en § 7.5.5 (Pág. 592) y basada en una exploración en amplitud.
7. Diseñe la primitiva `test_for_cycle(g, node, len)` la cual retorna `true` si existe un ciclo en el nodo `node` cuya longitud sea exactamente `len` arcos.
8. ¿Cómo puede especificarse un camino en un grafo o digrafo a través de sus arcos cuando éstos no estén etiquetados?

9. Plantee un algoritmo a fuerza bruta que verifique si dos grafos son o no isomorfos.
10. Plantee esquemas para representar una matriz de adyacencia correspondiente a un multigrafo.
11. Escriba el algoritmo de recorrido en profundidad sin recursión.
12. Modifique el bloque *< inicializar arreglo de arreglos 543 >* para que sólo aparte los elementos por arriba de la diagonal.
13. Modifique el bloque *< inicializar arreglo de arreglos 543 >* para usar el tipo BitArray.
14. Diseñe un TAD Ady\_Bit que represente una matriz de adyacencia de bits. Use un arreglo dinámico DynArray<T> de arreglos de bits BitArray. Minimice el consumo de memoria.
15. Dado un grafo  $\langle V, A \rangle$  y sean  $s_v$  y  $s_a$  los tamaños de los tipos almacenados en los nodos y arcos respectivamente. Sea  $s_p$  el tamaño de un apuntador. ¿Cuál es la relación entre la cantidad de arcos respecto a la de nodos de modo tal que sea menos o más costoso en espacio la representación con listas de adyacencia que con la de matrices?
16. ¿En cuáles casos la verificación de existencia de arco con matrices de adyacencia es  $\mathcal{O}(\lg(n))$ ?
17. Determine los puntos de articulación para del grafo de la figura 7.124.

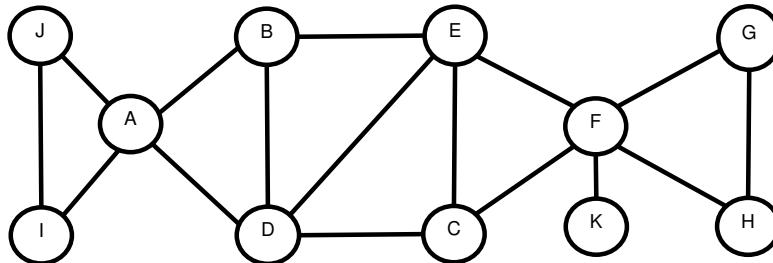


Figura 7.124: Un grafo de ejercicio para el cálculo de puntos de articulación

18. La prueba de correctitud del algoritmo de detección de puntos de corte presentada en § 7.5.14.3 (Pág. 619) sólo demuestra que los criterios de detección presentados en § 7.5.14.1 (Pág. 616) son correctos, es decir, que el algoritmo efectivamente encuentra puntos de corte. Sin embargo, la prueba no demuestra que el algoritmo encuentra todos los puntos de corte de un grafo.  
Así pues, demuestre que el algoritmo desarrollado en § 7.5.14.2 (Pág. 616) encuentra todos los puntos de corte.
19. Como se definió en § 7.5.14 (Pág. 613), a un grafo sin puntos de articulación se le califica de biconexo.  
Desarrolle un algoritmo, basado en una versión simplificada de `compute_cut_nodes()`, que determine si un grafo es o no biconexo.

20. En función del TAD List\_Graph, construya un algoritmo que calcule el complemento de un grafo.
21. Construya una versión del método sort\_arcs() que no use un arreglo dinámico temporal, sino que ordene directamente la lista de arcos.
22. En la rutina copy\_graph() (§ 7.3.10.9 (Pág. 577)), ¿cuáles son los inconvenientes de implantar el mapeo con una tabla hash?
23. Implante el recorrido en profundidad para que explore todos los arcos.
24. Implante la búsqueda en profundidad para la representación con matrices de adyacencia.
25. Implante el recorrido en amplitud para que explore todos los arcos.
26. Implante la búsqueda en amplitud para la representación con matrices de adyacencia.
27. Suponga un tipo abstracto de dato derivado de List\_Graph denominado Laberinto. Este tipo modela un laberinto cuyos arcos representan los pasillos y nodos las intersecciones (o encrucijadas) entre los pasillos.

(a) Considere la siguiente interfaz

```
Path * escapar(Laberinto & g, Laberinto::Node * s,
 Laberinto::Node * t)
```

Cuya finalidad es encontrar un camino desde s hacia t. El nodo s representa un lugar en el laberinto y el t una salida.

Escriba un algoritmo que encuentre un camino con la mínima cantidad de pasillos. (3 ptos)

(b) Considere la siguiente interfaz

```
Path * escapar(Laberinto & g, Laberinto::Node * s,
 Laberinto::Node * t, Laberinto::Node * p)
```

La cual encuentra un camino desde el nodo s hacia la salida t pasando por el cruce p.

Escriba un algoritmo que encuentre el camino pasando por la mínima cantidad de pasillos. (3 ptos)

28. Escriba un algoritmo que determine el grado de un grafo.
29. Implante un cálculo de árbol abarcador basado en un recorrido en amplitud. Explique las diferencias del árbol resultante respecto al algoritmo basado en búsqueda en profundidad desarrollado en § 7.5.9 (Pág. 603).
30. Implante una rutina que convierta un árbol abarcador almacenado en un objeto de tipo List\_Graph a un árbol de clase Tree\_Node . Estudie minuciosamente cómo hacer la conversión de los tipos asociados a los nodos y arcos.
31. Diseñe un algoritmo que calcule los puntos de corte y durante el mismo recorrido coloree los componentes conexos asociados a los puntos de corte. (+)

32. Suponga que posee el grafo de corte (obtenido a través de `map_cut_graph()`) y los bloques (obtenidos con llamadas a `map_subgraph()` con los diferentes colores dados por `compute_cut_nodes()`). Diseñe un algoritmo que calcule el grafo original sin apelar a los cookies.
33. Desarrolle una versión del TAD `Bit_Mat_Graph<GT>` que maneje estrictamente bits y que se fundamente en el tipo `DynArray<T>`, de manera tal que muchas entradas de valor cero no ocupen memoria.
34. Dado un grafo  $G$  y una lista  $l$  de nodos, diseñe un algoritmo con prototipo;

```
template <class GT>
void cut(GT & g, const DynDlist<typename GT::Node*> & l, GT & g1, GT & g2);
```

el cual retorna un corte del grafo tal que  $g_1$  contiene los nodos de la lista  $l$  y  $g_2$  los restantes.

35. Escriba el algoritmo de Dijkstra para matrices de adyacencia. Asegúrese de que, como en el algoritmo de Floyd, se calculen todos los pares de caminos más cortos.
36. Escriba el algoritmo de Bellman-Ford para matrices de adyacencia.
37. Modifique el algoritmo de Floyd-Warshall para detectar ciclos negativos. (+)
38. Suponga un grafo con ciclos negativos, escriba un algoritmo que identifique los ciclos y construya una lista caminos correspondiente a los ciclos en cuestión. (+)
- Ayuda: vea § 7.9.3.5 (Pág. 699).
39. En base a lo discutido al final de § 7.9.3.9 (Pág. 705), diseñe un algoritmo, basado en el de Bellman-Ford, que encuentre un ciclo negativo durante la fase de relación. Diseñe el algoritmo para que busque el ciclo sobre el arreglo `pred` sin necesidad de construir un grafo auxiliar. (+)
40. Con base en lo discutido al final de § 7.9.3.10 (Pág. 707), diseñe un algoritmo que añada un nodo auxiliar conectado con peso cero al resto de los nodos y que el ejecute el algoritmo de Bellman-Ford para encontrar un ciclo negativo. (+)
41. Estudie minuciosamente la detección de ciclos del algoritmo de Bellman-Ford, en particular `bellman_ford_negative_cycle()`, presentada § 7.9.3.10 (Pág. 705) y explique cómo podría este algoritmo encontrar varios ciclos negativos. (++)
42. El problema de encontrar el mínimo ciclo negativo de un digrafo con pesos es conocido como intratable. Demuestre la susodicha intratabilidad. (++)

43. Como es bien sabido, el algoritmo de Dijkstra no opera para arcos con distancias negativas. Una técnica para aplicar el algoritmo de Dijkstra a un grafo con distancias negativas consistiría en buscar la mínima distancia del grafo y, entonces, sumarle el negado a todos los arcos del grafo. De este modo, el grafo sólo contendría distancias positivas.

Demuestre que esta técnica no es válida y que los caminos mínimos sobre el grafo transformado pueden ser distintos a los del grafo original. (+)

44. Considere el grafo de la figura 7.125:

- (a) Calcule el camino mínimo desde el nodo a hasta el nodo k.
- (b) Calcule el árbol abarcador mínimo condicionado a que éste contenga el camino mínimo entre x e y.
- (c) Calcule el camino desde a hacia k condicionado a que éste contenga el camino mínimo entre x e y.

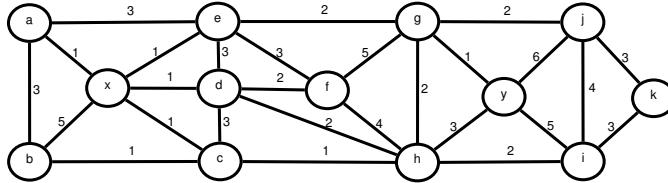


Figura 7.125: Un grafo de ejercicio para caminos mínimos

45. Segundo lo estudiado sobre el heap exclusivo en § 7.8.3.3 (Pág. 667), diseñe una estrategia general de recorrido en amplitud cuyo consumo en espacio no exceda de  $\mathcal{O}(V)$ . (+)

46. Escriba el ordenamiento topológico basado en el digrafo inverso.

47. Desarrolle un algoritmo de cálculo de componentes fuertemente conexos basado en el algoritmo de Warshall.

48. Demuestre que el algoritmo Kosaraju bosquejado en § 7.7.3.1 (Pág. 640) es correcto.

49. Instrumente completamente el algoritmo de Kosaraju estudiado en § 7.7.3.1 (Pág. 640).

50. Considere un plano de compuesto por  $W \times H$  puntos;  $W$  es el ancho y  $H$  es la altura.

Un punto en el plano se especifica por un par de valores  $(x, y)$ . El punto  $(0, 0)$  está situado en la esquina inferior izquierda y el  $(W-1, H-1)$  en la esquina superior derecha.

Los puntos pueden manejarse a través de la clase Point con los atributos x e y respectivamente.

- (a) Considere el siguiente prototipo de función:

```
BinNode<Point> * points(int n)
```

La cual retorna un árbol binario de búsqueda que contiene  $n$  puntos en el plano generados al azar. Las claves del árbol binario son de tipo Point

Usando la función random() de la pregunta anterior, instrumente la función.

- (b) Considere la siguiente función:

```
template <class GT>
GT * min_spanning_tree(DynDlist<Point> * points)
```

La cual retorna un grafo que contiene el árbol abarcador mínimo entre todos los puntos.

Cada nodo del árbol abarcador representa un punto. Cada arco del árbol abarcador representa la distancia entre los puntos.

Diseñe e Implemente un algoritmo que instrumente la función anterior.

(c) La misma que la anterior pero empleando matrices de adyacencia.

51. Dado un digrafo  $G = \langle V, E \rangle$ , el digrafo  $G' = \langle V, E' \rangle$  se define con los mismos componentes fuertemente conexos (en nodos) y las mismas relaciones de conexión entre los componentes, pero  $E' \subseteq E$  mínimo.
  - (a) Explique cómo calcular el digrafo  $G'$ .
  - (b) Describa un algoritmo eficiente para calcular  $G'$ .
52. Modifique los algoritmos de cálculo del flujo máximo de red para que calculen el eslabón mínimo en la misma pasada que el cálculo del camino de aumento.
53. Demuestre (7.17) (pag. 718).
54. Haga el análisis del algoritmo de empuje de preflujo con cola FIFO estudiado en 7.10.13.8.
55. Haga el análisis del algoritmo de empuje de preflujo con mayor distancia estudiado en 7.10.13.9.
56. Modifique todas las funciones diseñadas para calcular el flujo máximo según empuje de preflujo para que tengan como parámetro la operación de inicialización de la altura de los nodos (véase § 7.10.13.11 (Pág. 761)).
57. Diseñe una rutina de conversión de un grafo con capacidades a una red del tipo `Net_Graph`.
58. Diseñe un rutina de conversión de una red del tipo `Net_Graph` a un grafo `List_Graph`.
59. Diseñe e instrumente un algoritmo general que convierta una circulación con una cantidad arbitraria de ciclos a una red con un fuente y un sumidero.
60. Diseñe un algoritmo genérico que convierta una red de circulación a una red con un solo fuente y un sumidero.
61. Considere el algoritmo genérico de empuje de preflujo por arcos estudiado en § 7.10.13.11 (Pág. 760).
  - (a) Escriba una especialización del algoritmo que maneje el conjunto de nodos activos mediante una tabla hash.
  - (b) Diseñe una estructura de datos que permita en  $\mathcal{O}(1)$  incrementar directamente el flujo y a la vez detectar los arcos de la cola q que devienen inelegibles. (++)
62. Proponga y realice un estudio de rendimiento para todas las combinaciones posibles de algoritmos estudiados de maximización de flujo. (+)

63. En función del conocimiento que aporta el teorema de descomposición de flujo 7.16, instrumente el algoritmo 7.9 empleado para la demostración.
64. Enuncie y demuestre el teorema de Menger para grafos no dirigidos.
65. En una organización se tienen  $n$  trabajadores  $\{t_1, t_2, \dots, t_n\}$  y  $m$  proyectos  $\{p_1, p_2, \dots, p_m\}$ . Cada trabajador está interesado en participar en algunos proyectos  $\{p_x, p_y, \dots\}$ . Cada proyecto tiene una cota de participantes de  $N(p_i)$ ; es decir, en el proyecto  $p_i$  a lo sumo pueden participar  $N(p_i)$  trabajadores.
- Modelice el problema para maximizar la cantidad de trabajadores que participarían en los proyectos.
  - Suponga que un trabajador  $t_i$  cobra una cantidad  $S(t_i, p_j)$  por participar en el proyecto  $p_j$ . ¿Cómo se encuentra la participación de trabajadores máxima al mínimo coste?
  - Suponga que un trabajador  $t_i$  tiene una eficiencia  $E(t_i, p_j)$  en el proyecto  $p_j$ . ¿Cómo se encuentra la participación de trabajadores en la mayor cantidad de proyectos con la máxima eficiencia?
66. Dado un corte de un grafo calculado mediante `compute_min_cut()` (§ 7.11.6.3 (Pág. 789)), calcule los bloques que desconectaría la supresión del corte mínimo.
67. Diseñe un algoritmo que calcule  $K_v(G)$  conocido el corte mínimo de  $G$ .
68. Modifique el TAD `Net_Graph` para que incluya factores de ganancia en los arcos de la manera explicada en § 7.14.2 (Pág. 823). Revise todas las rutinas de la biblioteca relacionadas con redes de flujo, identifique las que manejen incrementos o decrementos de flujo y modifíquelas para que consideren este factor.
69. Exprese una red genérica multifujo en términos de un programa lineal.
70. Considere la siguiente red capacitada de la figura 7.126 en la cual el valor de flujo de la red es cero.

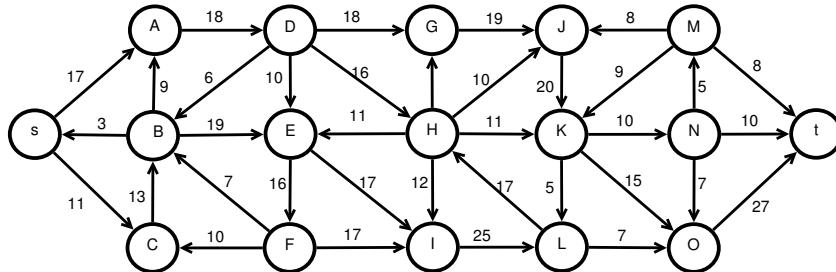


Figura 7.126: Una red de ejercicio

- Encuentre un camino de aumento con eslabón mínimo de 11.
- Maximice el valor de flujo de la red (iniciado éste en cero).
- En función de la respuesta anterior, encuentre un camino de aumento que permita disminuir el valor de flujo de la red en 5.

- (d) Calcule el corte de capacidad mínima. Resalte los arcos que son parte del corte mínimo.
- (e) Calcule la conectividad en arcos de la red entre los nodos s y t.
- (f) Asumiendo un valor de flujo cero, inyecte el mayor flujo posible por el camino de aumento s – C – B – A – D – H – K – N – M – t y luego maximice la red.
71. En cierta Universidad se desean organizar cursos intensivos durante los diversos períodos de vacaciones.

Hay  $m$  cursos intensivos de un conjunto  $C = \{C_1, C_2, \dots, C_m\}$  cada uno de  $H_1, H_2, \dots, H_m$  horas que se pueden dictar al año. Por simplicidad, se asume que cada curso intensivo tiene un programa tan bien detallado que cualquier profesor, conociendo el estado del curso, está en la capacidad de dictar cualquier parte del programa.

Hay  $n$  profesores  $P_1, P_2, \dots, P_n$ . Para cada profesor  $P_i$  se asocia la siguiente información:

- (a)  $M_i$ : cantidad máxima de horas de curso intensivo que puede dictar al año.
- (b)  $\{C_1, C_5, \dots\} \subset C$ : lista de cursos intensivos que puede dictar.
- (c) Categoría a la que pertenece: instructor, asistente, agregado, asociado y titular.

El objetivo del problema es diseñar un algoritmo, con complejidad polinomial, que tome la información dada y determine la mejor asignación de profesores a cursos intensivos con la restricción, imperativa, de que los profesores de más altas categorías tienen preferencias sobre los de menores.

El algoritmo debe retornar una asignación correcta o indicar que no existe.

Explique cómo se modelizaría una red de flujo que instrumente el algoritmo en cuestión.

# Bibliografía

- [1] Electric fence. <http://www.pf-lug.de/projekte/haya/efence.php>.
- [2] Flawfinder. <http://www.dwheeler.com/flawfinder/>.
- [3] [http://en.wikipedia.org/wiki/birthday\\_problem](http://en.wikipedia.org/wiki/birthday_problem).
- [4] <http://www.efgh.com/math/birthday.htm>.
- [5] <http://www.gnu.org/software/gperf/>.
- [6] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.
- [7] Aho, Hopcroft, and Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, 1983.
- [8] Aho, Hopcroft, and Ullman. *Estructuras de datos y algoritmos*. Addison-Wesley Iberoamericana, 1988. Traducci?n de Am?rico Vargas Villaz?n y Jorge Lozano Moreno.
- [9] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, USA, 1974.
- [10] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
- [11] Paul Bachmann. *Die Analytische Zahlentheorie. Zahlentheorie, pt. 2*. B. G. Teubner, Liepzig, 1894.
- [12] J. W. Backus and W. P. Heising. FORTRAN. *IEEE Transactions on Electronic Computers*, EC-13(4):382–385, 1964.
- [13] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing; Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.
- [14] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, November 1972.
- [15] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.
- [16] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows*. Wiley, fourth edition, 2009.

- [17] <http://invisible-island.net/bcpp>.
- [18] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [19] J. L. Bentley. *Programming Pearls*. Addison-Wesley, 1986.
- [20] Daniel J. Bernstein. Página web de daniel j. bernstein.
- [21] Dimitri P. Bertsekas. *Network Optimization: Continuous and Discrete Models*. Athena Scientific, Belmont, Massachusetts, 1998.
- [22] <http://www.ctan.org>.
- [23] O. Boruvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 3:37–58, 1926. In Czech.
- [24] Fredrick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison Wesley, Reading, Mass., second edition, 1995.
- [25] J. Büchi. Weak second-order logic and finite automata. *Z. Math. Logik Grundlagen Math.*, 5:66–92, 1960.
- [26] Byte, editor. *Special issue on Smalltalk*, volume 6, August 1981.
- [27] Peter Calingaert. *Assemblers, compilers, and program translation*. Computer Science Press, 1979.
- [28] Carter and Wegman. Universal classes of hash functions. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1977.
- [29] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, January 1980.
- [30] <http://www.cmake.org>.
- [31] F. J. Corbato, J. H. Saltzer, and C. T. Clingen. Multics – the first seven years. In *Spring Joint Computer Conference*, pages 571–583. AFIPS Press, May 1972.
- [32] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 1989.
- [33] C. A. Crane. *Linear lists and priority queues as balanced binary trees*. PhD thesis, Computer Science Department, Stanford University, 1972.
- [34] O. J. Dahl and K. Nygaard. SIMULA – an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–682, September 1966.
- [35] G. B. DANTZIG. Programming in a linear structure. *Econometrica*, 17, 1949.
- [36] <http://www.gnome.org/projects/dia>.
- [37] E. W. Dijkstra. In honour of fibonacci. In F. L. Bauer and M. Broy, editors, *Program Construction*, volume 69 of *Lecture Notes in Computer Science*, pages 49–50. Springer Verlag, 1978.

- [38] <http://www.doxygen.org>.
- [39] Arnold I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, December 1956. First paper in open literature on hashing. First use of hashing by taking the modulus of division by a prime number. Mentions chaining for collision handling, but not open addressing. See [47] for the latter.
- [40] Edmonds and Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *JACM: Journal of the ACM*, 19, 1972.
- [41] P. Elias, A. Feinstein, and C. E. Shannon. Note on flow through a network. *IRE Trans. on Information Theory*, pages 117–119, 1956.
- [42] Ellson, Gansner, Koutsofios, North, and Woodhull. Graphviz – open source graph drawing tools. In *GDRAWING: Conference on Graph Drawing (GD)*, 2001.
- [43] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools, 2003.
- [44] <http://www.gnu.org/software/emacs/>.
- [45] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code, August 09 2001.
- [46] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 7th Symposium on Operating System Design and Implementation*, pages 1–16, 2000.
- [47] A. P. Ershov. *Doklady Adak. Nauk SSSR*, 118:427–430, 1958. Rediscovery and first publication of linear open addressing.
- [48] William Feller. *An Introduction to Probability Theory and Its Applications*, Vol. 1. Wiley, 1970.
- [49] William Feller. *An Introduction to Probability Theory and Its Applications*, Vol. 2. Wiley, 1970.
- [50] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. Technical Report STAN-CS-87-1176, Department of Computer Science, Stanford University, 1987, August.
- [51] Philippe Flajolet, P. J. Grabner, P. Kirschenhofer, and H. Prodinger. On ramanujan's Q-function. Technical Report RR-1760, Inria, Institut National de Recherche en Informatique et en Automatique.
- [52] R. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.
- [53] R. W. Floyd. Algorithm 245 : Treesort 3. *Comm. ACM*, 7:701, 1964.

- [54] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian J. of Mathematics*, 8:399–404, 1956.
- [55] Lester R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
- [56] Ramsés Fuenmayor. *Interpretando Organizaciones... Una Teoría Sistémico-Interpretativa de Organizaciones*. Universidad de Los Andes - Consejo de publicaciones - Consejo de Estudios de Postgrado, 2001.
- [57] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [58] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, February 2002.
- [59] Alan Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, 1985.
- [60] <http://www.gnu.org>.
- [61] <http://www.gnuplot.info>.
- [62] Goldberg and Tarjan. A new approach to the maximum-flow problem. *JACM: Journal of the ACM*, 35, 1988.
- [63] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.
- [64] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Pub., 1994.
- [65] Object Management Group. *OMG Unified Modeling Language 2.0 Proposal, Revised submission to OMG RFPs ad/00-09-01 and ad/00-09-02, Version 0.671*. OMG, <http://www.omg.com/uml/>, 2002.
- [66] <http://www.gnu.org/software/gsl/>.
- [67] Guibas and Sedgewick. A dichromatic framework for balanced trees. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, 1978.
- [68] Takao Gunji and Eiichi Goto. Studies on hashing — 1. a comparison of hashing algorithms with key deletion. *Journal of the Information Processing Society of Japan*, 3(1):1–12, ???? 1980.
- [69] David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison Wesley, 1995.
- [70] Frank Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.

- [71] C. A. R. Hoare. Algorithms 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.
- [72] Per Holager, Jean-Marc Jézéquel, and Bertrand Meyer. Letters: Placing the blame for Ariane 5. *Computer*, 30(5):6, 8, May 1997.
- [73] Holzmann. The power of 10: Rules for developing safety-critical code. *COMPUTER: IEEE Computer*, 39, 2006.
- [74] G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
- [75] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [76] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [77] Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.
- [78] J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation. *Commun. ACM*, 1(6):372–378, June 1973.
- [79] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable length alphabetic codes. *SIAM J. Applied Math.*, 21:514–532, 1971.
- [80] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. I.R.E.*, 40:1098–1101, 1951.
- [81] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.
- [82] <http://xorg.freedesktop.org>.
- [83] D. H. H. Ingalls. Design principles behind smalltalk. *BYTE*, 6(8):286–298, August 1981.
- [84] V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930. in Czech.
- [85] Jean-Marc Jézéquel and Bertrand Meyer. Object technology: Design by contract: The lessons of Ariane. *Computer*, 30(1):129–130, January 1997. See also letters [72].
- [86] Andrew L. Johnson and Brad C. Johnson. Literate programming using noweb. *Linux Journal*, pages 64–69, october 1997.
- [87] S. C. Johnson. LINT : A C program checker. In *UNIX Programmer Manual*. BELL Labs., 7<sup>th</sup> edition, 1979.
- [88] Nicolai M. Josuttis. *The C++ Standard Library: a tutorial and reference*. pub-AW, pub-AW:adr, 1999.

- [89] Dieter Jungnickel. *Graphs, Networks and Algorithms*. Springer, 2004.
- [90] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.
- [91] Karp and Rabin. Efficient randomized pattern-matching algorithms. *IBM JRD: IBM Journal of Research and Development*, 31, 1987.
- [92] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl.*, 15:434–437, 1974.
- [93] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Reading, MA, 1999.
- [94] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.
- [95] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.
- [96] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
- [97] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, third edition, 1997.
- [98] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, third edition, 1997.
- [99] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.
- [100] Donald E. Knuth. *Selected Papers on Analysis of Algorithms*. CSLI Publications, Stanford, CA, USA, 2000.
- [101] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer-Verlag, fourth edition, 2008.
- [102] Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms*. CRC Press, 1999. ISBN 0-8493-3988-X.
- [103] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–28, January 1984.
- [104] Yedidyah Langsam, Moshe Augenstein, and Aaron M. Tenenbaum. *Data structures using C and C++*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1996.
- [105] Larson. Dynamic hash tables. *CACM: Communications of the ACM*, 31, 1988.
- [106] <http://www.ctan.org>.
- [107] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt-Rinehart-Winston, New York, 1976.

- [108] Harry R. Lewis and Larry Denenberg. *Data Structures and Their Algorithms*. Harper Collins Publishers, New York, 1991.
- [109] Liskov and Zilles. Programming with abstract data types. *Sigplan Notices*, 9, 1974.
- [110] Witold Litwin. Linear hashing: A new algorithm for files and tables addressing. In *ICOD*, pages 260–276, 1980.
- [111] David G. Luenberger and Yinyu Ye. *Linear and Nonlinear Programming*. Springer-Verlag, third edition, 2008.
- [112] J. Lukasiewicz. O znaczeniu i potrzebach logiki matematycznej (on the importance and needs of mathematic logic). *Nauka Polska*, 10:604–620, 1929.
- [113] Alasdair MacIntyre, editor. *After Virtue*. University of Notre Dame Press, 1984.
- [114] Steve Maguire. *Writing solid code: Microsoft's techniques for developing bug-free C programs*. MICROSOFT, 1993.
- [115] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by V-automata. In *Conference record of the 14th ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–12, 1987.
- [116] Zohar Manna and Richard Waldinger. The origin of the binary-search paradigm. Technical Report Report, Stanford University, Stanford, CA, 1987.
- [117] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, March 1998.
- [118] Makoto Matsumoto and Yoshiharu Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2(3):179–194, July 1992.
- [119] Makoto Matsumoto and Yoshiharu Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation*, 4(3):254–266, July 1994.
- [120] Makoto Matsumoto and Takuji Nishimura. The dynamic creation of distributed random number generators. In *PPSC*, 1999.
- [121] <http://maxima.sourceforge.net>.
- [122] Steve McConnell. *Code Complete*. Microsoft Press, Redmond, WA., 1993.
- [123] K. Menger. Untersuchungen über allgemeine Metrik. *Math. Ann.*, 100:75–163, 1928.
- [124] Bertrand Meyer. Design by contract, components and debugging. *JOOP*, 11(8):75–79, 1999.
- [125] Scott Meyer. *More effective C++: 35 new ways to improve your programs and designs*. Addison-Wesley, 1996. ISBN 0-201-63371-X.
- [126] Scott Meyers. *Effective C++*. Addison Wesley, 1992.
- [127] Scott Meyers. *More Effective C++*. Addison Wesley, 1996.

- [128] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1978.
- [129] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.
- [130] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.
- [131] Allen Newell and J. C. Shaw. Programming the logic theory machine. In *Proceedings of the 1957 Western Joint Computer Conference*, pages 230–240. IRE, 1957.
- [132] Allen Newell, J. C. Shaw, and H. A. Simon. Empirical explorations with the logic theory machine. In *Western Joint Computer Conference*, volume 15, pages 218–239, 1957.
- [133] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer series in operations research. Springer-Verlag, pub-SV:adr, second edition, 2006.
- [134] A Managerial View of the Multics System Development. F. J. corbato and C. T. clin-gen. In *Conference on Research Directions in Software Technology*, Providence, Rhode Island, October 1977.
- [135] A. Oram and G. Wilson, editors. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly, 2007.
- [136] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.
- [137] Ozan Yigit oz@nexus.yorku.ca. Página web del proyecto sdbm.
- [138] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Prentice-Hall, 1982.
- [139] Ian Parberry. *Problems on algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
- [140] D. L. Parnas. A technique for software module specification with examples. *Communications of the Association of Computing Machinery*, 15(5):330–336, May 1972.
- [141] A. J. Perlis and C. Thornton. Symbol manipulation by threaded lists. *Communications of the ACM*, 3(4):195–204, April 1960.
- [142] Y. Perret. Functioncheck. <http://sourceforge.net/projects/fnccheck>.
- [143] W. W. Peterson. Addressing for random access storage. *IBM Journal of Research and Development*, 1(2), April 1957.
- [144] P.J.Maker and The Free Software Foundation. The GNU NANA homepage.
- [145] A. Pnueli. The temporal logic of programs. In *focs'77*, pages 46–57, 1977.
- [146] [www.http://r-project.org](http://r-project.org).

- [147] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.
- [148] Singiresu S. Rao. *Engineering Optimization*. JOHN WILEY & SONS, INC., fourth edition, 2009.
- [149] David S. Rosenblum. Correction to “A practical approach to programming with assertions”. *IEEE Transactions on Software Engineering*, 21(3):265, March 1995.
- [150] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
- [151] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating systems. *USENIX Computing Systems*, 1(4):305–370, 1988.
- [152] James E. Rumbaugh. OMT: The object model. *JOOP*, 7(8):21–27, 1995.
- [153] Theo C. Ruys and Gerard J. Holzmann. Advanced SPIN tutorial. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 304–305. Springer, 2004.
- [154] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), 1984.
- [155] Robert Sedgewick. *Algorithms in C: Parts 1–4: Fundamentals, data structures, sorting, searching*. Addison-Wesley, Reading, MA, USA, 1998.
- [156] Robert Sedgewick. *Algorithms in C: Part 5: Graph algorithms*. Addison-Wesley, pub-AW:adr, 2002.
- [157] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Publishing Company, 1996.
- [158] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, October/November 1996.
- [159] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1998.
- [160] D.D. Sleator and Robert E Tarjan. Self-adjusting Binary Search Trees. *Journal of the ACM*, (32):195–204, 1985.
- [161] Richard Stallman and Roland H. Pesch. *Debugging with GDB: the GNU source-level debugger*. Free Software Foundation, Inc., pub-FSF:adr, 4.09 for GDB version 4.9 edition, 1993. Previous edition published under title: The GDB manual. August 1993.
- [162] C. J. Stephenson. A method for constructing binary search trees by making insertions at the root. *International Journal of Computer and Information Sciences*, 9(1):15–29, February 1980.

- [163] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley, Reading Mass., 1994.
- [164] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 2000.
- [165] Kozo Sugiyama. *Graph Drawing and Applications for Software Engineering and Knowledge Engineers*, volume 11 of *Software Engineering and Knowledge Engineering*. World Sceintific, 2002.
- [166] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972.
- [167] Robert E. Tarjan. Amortized computational complexity. *Sialgdis*, 6:306–318, 1985.
- [168] Larry Tesler. The Smalltalk environment. *Byte*, 6(8):90–147, August 1981.
- [169] <http://tug.org/tetex/>.
- [170] <http://www.kde.org>.
- [171] Peter van der Linden. *Expert C Programming - Deep C Secrets*. SunSoft Press -A Prentice Hall Tittle-, 1994. ISBN 0-13-177429-8.
- [172] G. van Rossum and F. L. Drake, Jr., editors. *An Introduction to Python*. Network Theory Ltd, September 2003.
- [173] T. van Vleck. Three questions about each bug you find. 1989.
- [174] Varios. Guest editor's introduction: The top 10 algorithms. *Computing in Science and Engineering (CSE)*, 2(1):22–23, January 2000.
- [175] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference*. ACM, December 2000.
- [176] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978.
- [177] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1990.
- [178] Stephen Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, January 1962.
- [179] Gray Watson. Dmalloc - debug malloc library. <http://dmalloc.com/>.
- [180] M. A. Weiss. *Data Structures and Algorithm Analysis in C*. Addison \_Wesley, 1997. “D S and A A” [not in C], Benjamin \_ Cummings, '92.
- [181] J. W. J. Williams. Algorithm 232 : HEAPSORT. *Comm. ACM*, 7:347–348, 1964.
- [182] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1976.

- [183] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 1993.
- [184] Derick Wood. *Data Structures, Algorithms, and Performance*. Addison-Wesley, Reading, 1993.
- [185] <http://www.xfig.org>.
- [186] Junfeng Yang, Can Sar, and Dawson Engler. Xplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating System Design and Implementation*, 2006.
- [187] Jay Yellen and Jonathan L. Gross. *Graph Theory & Its Applications*. CRC Press, 1998. ISBN: 0-849-33982-0.
- [188] P. Zafiropulo, C. West, H. Rudin, D. Cowan, and D. Brand. Towards analysing and synthesizing protocols. *IEEE Transactions on Communication*, Comm-28(4):651–661, 1980.
- [189] Andreas Zeller and Dorothea Lütkehaus. DDD - A free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, 31(1):22–27, 1996.

# Índice de términos

- árbol
  - altura de un, 275
  - completo
    - representación con arreglo secuencial, 282
    - definición, 226
  - árbol abarcador de un grafo, 538
  - árbol abarcador de amplitud, 603
  - árbol binario
    - definición, 233
    - nodo completo, 280
    - nodo incompleto, 280
    - nodo lleno, 280
    - recorrido infijo, 235
    - recorrido por niveles, 235
    - recorrido prefijo, 235
    - recorrido sufijo, 235
  - árbol binario de búsqueda
    - búsqueda, 315
    - definición, 314
    - eliminación, 324
    - inserción, 320
    - inserción en raíz, 326
    - supresión, (324, 326)
  - árbol binario de búsqueda extendido
    - inserción en raíz, 340
  - árboles
    - ancestros de un nodo, 226
    - binarios de búsqueda, (313, 332)
      - análisis, (329, 332)
      - búsqueda, (315, 319)
      - búsqueda de padre, 317
      - búsqueda de rango, 317
      - búsqueda del predecesor, 317
      - búsqueda del sucesor, 317
      - crítica, (329, 332)
      - definición, 314
      - inserción, (320, 321)
    - máximo, 316
    - mínimo, 316
    - supresión, (324, 326)
  - generación de un nodo, 227
  - grado de un nodo, 227
  - representación en memoria, 231
- árboles binarios
  - cálculo de la altura, 249
  - cálculo de la cardinalidad, 249
  - comparación, 250
  - destrucción, 250
  - equivalencia, 251
  - extensiones
    - cálculo de la posición infija, 338
    - inserción por clave, 338
    - selección, 337
  - hilados, (255, 260)
  - join, 327
  - join exclusivo, 323
  - partición, 321
    - por posición, 340
  - partición por clave, 339
  - recorridos no recursivos, 246
  - rotación, 344
  - split, 321
  - unión, 327
  - unión exclusiva, 323
- árboles binarios con rangos, 336
- árboles binarios de búsqueda, (313, 332)
  - análisis, (329, 332)
  - crítica, (329, 332)
  - extendidos, 336
- árboles binarios extendidos, (335, 344)
  - eliminación por clave, 342
  - eliminación por posición, 343
  - inserción
    - por posición, 341
  - rotación, 345

- unión exclusiva, 342
- árboles estáticos óptimos, (358, 363)
- remove\_all\_and\_delete(), 82
- estructura de datos
  - orientada hacia el concepto, 21
  - orientada hacia el flujo, 21
  - orientadas hacia los datos, 21
- atributos, 385
- cubetas, 384
- inserción, 386
- 80-20 regla, 212
- ABB**
  - búsqueda, 315
  - búsquedas especiales, 317
  - concatenación, 327
  - concatenación exclusiva, 323
  - eliminación, 324
  - inserción, 320
  - inserción en raíz, 326
  - join, 327
  - join exclusivo, 323
  - partición, 321
  - predecesor, 317
  - split, 321
  - sucesor, 317
  - unión, 327
  - unión exclusivo, 323
- ABBA**
  - unión exclusiva, 458
- ABBE**
  - cálculo de la posición infija, 338
  - desempeño, 343
  - eliminación por clave, 342
  - eliminación por posición, 343
  - inserción en raíz, 340
  - inserción por clave, 338
  - inserción por posición, 341
  - partición, 339
  - partición por posición, 340
  - selección, 337
  - split, 339
  - unión exclusiva, 342
- acceso fuera de bloque, 208
- aciclicidad, 591
- actuador, 9
- alcanzabilidad, 636
- aleatorización, 180
- álgebra de  $\mathcal{O}$ , 159–160
- algoritmo
  - de cálculo de  $K_e(G)$ , 784
  - de comparación de árbol binario, 250
  - de descomposición de flujo, 782
  - de destrucción de árbol binario, 250
  - de Kruskal, 659–664
  - de Prim, 664–672
  - recursivo de altura de un árbol binario, 249
  - recursivo de cardinalidad de un árbol binario, 249
- algoritmo de Ford-Fulkerson, 724
- análisis, 728
- algoritmo de Edmonds-Karp, 729
- análisis, 732
- algoritmo de cálculo de  $K_e(G)$ , 784
- algoritmo de cálculo de conectividad de arcos, 784
- algoritmo de conversión de red con capacidades en nodos a una red tradicional, 772
- Algoritmo de Rabin-Karp, 431
- Algoritmo de Warshall para la clausura transitiva, 633–635
- algoritmo simplex, 813–820
- algoritmos aleatorios, 180
- altura
  - de nodo en red de flujo, 735
  - de un árbol, 228
  - de un nodo, 227
- altura de un árbol, 275
- altura de un árbol binario, 249
- altura negra en un árbol rojo-negro, 490
- análisis amortizado, 187
- análisis contable, 191–192
- análisis de algoritmos, 147
- análisis de los árboles binarios de búsqueda, (329, 332)
- análisis del algoritmo de Ford-Fulkerson, 728
- análisis del algoritmo de Edmonds-Karp, 732
- análisis del algoritmo genérico de preflujo, 744

- análisis del manejo de colisiones por direccionamiento abierto y sondeo lineal, 399
- análisis del manejo de colisiones por encadenamiento cerrado, 392
- análisis del manejo de colisiones por encadenamiento separado, 387
- análisis del mergesort, 171
- análisis del sondeo ideal, 393
- análisis potencial, 189–191
- análisis amortizado
- análisis contable, 191–192
  - análisis potencial, 189–191
  - selección de créditos, 192–193
  - selección de función potencial, 192–193
- análisis de los árboles aleatorizados, 460–464
- análisis de los árboles splay, 513–518
- análisis de los treaps, 470–471
- análisis del quicksort, 176
- analizador externo, 200
- ancestros de un nodo, 226
- aplicaciones de las colas, 123
- apuntadores y arreglo, 29
- árbol
- rojo-negro, 489
  - de Fibonacci, 484
- árbol AVL crítico, 486
- árbol abarcador de profundidad, 601
- árbol abarcador mínimo, 658–672
- árbol binario de búsqueda aleatorio, 455
- árboles
- almacenamiento, 311
  - AVL
    - análisis, 483–489
    - eliminación, (478, 483)
    - inserción, (473, 478)  - equilibrio perfecto, 452
  - nodo binario
    - TAD BinNode<Key>, 238
- rojo-negro, 489–506
- análisis, 502–506
  - eliminación, (496, 502)
  - inserción, (492, 496)
- splay, 506–518
- árboles AVL, 471–489
- análisis, 483–489
- definición, 471
- eliminación, (478, 483)
- inserción, (473, 478)
- árboles aleatorizados, 455–464
- análisis, 460–464
- concatenación, 458
- eliminación, 457
- inserción, 456
- supresión, 457
- treap, 468
- árboles binarios
- similaridad, 251
- árboles binarios aleatorizados
- unión exclusiva, 458
- árboles binarios extendido
- split, 339
- árboles de Fibonacci, 484–487
- altura, 485
  - cardinalidad, 485
- árboles equilibrados, 452–454
- árboles splay, 506–518
- análisis, 513–518
- árboles treaps
- análisis, 470–471
- árboles binarios de búsqueda, 332
- árboles completos
- representación en memoria, 280
- arborescencia, 226
- recorridos, 262, 270
  - TAD Tree\_Node , 269
- arco
- relajación, 693
- arco incidente, 536
- arco paralelo, 537
- arcos de un grafo
- implantación, 564
- aritmética de polinomios
- uso de listas enlazadas, 91
- aritmética de punteros, 29
- arreglo
- en memoria dinámica, 33
  - búsqueda binaria, 30
  - búsqueda por clave, 30
  - cálculo de dirección, 29
  - conceptos generales, 28
  - de bits, 53–58

- dinámico, 34–53
- eliminación por clave, 31
- en memoria estática, 32
- en pila, 33
- inserción por clave, 31
- multidimensional, 58
- para representar árboles, 232
- asertos, 204
- asignación, 775
  - flujo máximo coste mínimo, 807
- LhashTable<Key>, 385
- atributos de control de nodos y arcos de un grafo, 553
- autómata, 202
- AVL
  - árboles, 471–489
- búsqueda
  - de arcos en grafo, 552
  - de ciclos negativos, 701
  - de nodos en grafo, 547
  - secuencial, 15
- búsqueda binaria, 30
- búsqueda de arcos en un grafo
  - implementación, 563
- búsqueda de extremos, 156
- búsqueda de máximo, 156
- búsqueda de mínimo, 156
- búsqueda de nodos en un grafo
  - implementación, 563
- búsqueda en tabla hash lineal, 418
- búsqueda secuencia, 154–155
- Bellman-Ford
  - caminos mínimos, 690–705
- BinHeap<Key>
  - actualización de un elemento, 299
  - eliminación, 298
  - eliminación de cualquier elemento, 299
  - eliminación de todos los elementos, 300
  - Inserción, 297
  - intercambio entre nodos, 296
- BinNode<Key>
  - conversión a Tree\_Node , 273
- BinTree<Key>, 319
- BitArray, 53–58
- bits de control, 553
- bucle, 536
- búsqueda
  - sobre arreglos, 184
- búsqueda aleatoria
  - en arreglo, 184
  - en listas, 185
- búsqueda binaria, 30, 165–166
- búsqueda de caminos por amplitud, 598
- búsqueda de caminos por profundidad, 594
- búsqueda de ciclos en un grafo, 590
- cálculo de nodos pertenecientes a un nivel, 254
- cálculo del número de Dewey, 272
- códigos de Huffman, 345–358
  - codificación de texto, 353
  - decodificación, 349
  - ingreso de frecuencias, 352
  - optimación, 355
- cache, 215
- cadenas de producción
  - flujo máximo coste mínimo, 824
- calculo de red residual en una red de flujo, 721
- camino
  - búsqueda en profundidad, 595
  - entre nodos de un árbol, 226
  - longitud en un árbol, 226
  - prueba de existencia, 594, 633
  - simple, 536
- camino de Catalan, 308
- camino de un grafo, 536
- Camino más corto, 211
- camino más corto, 210
- caminos de aumento, 718
  - cálculo de, 723
- caminos mínimos, 672–706
  - algoritmo de Bellman-Ford, 690–705
  - algoritmo de Dijkstra, 673–683
  - algoritmo de Floyd-Warshall, 683–690
  - discusión, 705
- cancelación de ciclos negativos
  - algoritmo genérico de, 794
- capacidad, 706
- capacidad de un corte, 716
- cardinalidad
  - de un árbol, 228
  - identidad, 278

- del nivel, 277  
 cardinalidad de un árbol binario, 249  
 casting  
     Dlink a estructura contenida, 80  
     Slink a estructura contenida, 68  
 Catalan  
     número de, 307–311  
 ciclos  
     búsqueda de negativos, 701  
     detección, 590  
     prueba de aciclicidad, 591  
     uso de algoritmo de Bellman-Ford para su detección, 701  
 ciclos negativos  
     algoritmo de Bellman-Ford, 697  
     búsqueda, 701  
 circulaciones, 781  
     algoritmo de descomposición de flujo, 782  
     descomposición de una red en, 782  
 clase  
     gestora, 49  
     intermediaria, 49  
     proxy, 49  
 clausura transitiva, 595, 633  
 clique de un grafo, 537  
 código  
     de Lukasiewicz, 302, 306  
 código de un árbol binario, 302  
 colas  
     aplicaciones, 123  
     mediante arreglos, 124  
     mediante listas enlazadas, 128  
     mediante listas enlazadas dinámicas, 130  
     representaciones en memoria, 123  
 colas de prioridad, 288  
     actualización de un elemento, 289  
 comparación de árboles binarios, 250  
 Compare y Dlink, 152  
 complemento de un grafo, 537  
 componentes conexos de los puntos de corte, 620  
 componentes fuertemente conexos de un grafo, 637–651  
 Componentes inconexos de un grafo, 608  
 comprensión de datos, 345–358  
 concatenación de árboles aleatorizados, 458  
 condición roja, 490  
 condición negra, 490  
 conectividad, 783  
     entre digrafos, 636  
     entre grafos, 587  
 conectividad de arcos  
     algoritmo, 784  
     cálculo, 784  
     implantación del algoritmo de cálculo, 785  
 conectividad entre grafos, 633  
 conjunto, 20  
 conjunto prefijo, 303  
 conjuntos  
     representación de árboles, 228  
 contracción de tabla hash lineal, 413  
 conversión  
     Dlink a estructura contenida, 80  
     Slink a estructura contenida, 68  
 conversión de un árbol abarcador a un Tree\_Node , 606  
 correctitud  
     búsqueda, 194  
     detección, 194  
     planteamiento del problema, 193  
 correspondencia entre árboles binarios y m-rios, 260  
 correspondencia entre BinNode<Key> y Tree\_Node , 273  
 correspondencia entre Tree\_Node y BinNode<Key>, 273  
 corte  
     capacidad de, 716  
     valor de flujo, 715  
 corte de una red, 714  
 corte mínimo, 716  
     cálculo de, 764  
 cortes de red, 714  
 coste en espacio de algoritmos dividir y binar, 177  
 coste mínimo  
     del flujo máximo, 791  
 costumbres de correctitud  
     decálogo de Holtzmann, 196

- Preguntas actitudinarias de Van Cleck, 199  
crítica de algoritmos, 147  
crítica de los árboles binarios de búsqueda, (329, 332)  
crítica del mergesort, 171  
crítica del quicksort, 176  
`LhashTable<Key>`, 384  
cumpleaños, 381
- DAG, 651  
`ddd`, 210  
deadlock, 202  
DEBUG, 204  
decálogo de Holtzmann, 196  
deducción, 22  
`delete`, 33  
depuradores, 210  
descomposición de flujo, 782  
desempeño de los árboles binarios extendidos, 343  
`destroyRec`, 250  
destrucción  
    de `Tree_Node`, 270  
destrucción de un árbol binario, 250  
detección de ciclos negativos, 697  
Deway  
    búsqueda en `Tree_Node`, 271  
    notación para árboles, 231  
`df`, 611  
diámetro de un grafo, 706  
Diagrama UML  
    Listas doblemente enlazadas, 84  
dicolas, 123  
digrafo, 536  
    componentes fuertemente conexos, 637–651  
digrafos, 635–658  
    conectividad, 636  
    inversión, 636  
    ordenamiento topológico, 653–658  
    planificación, 652  
digrafos acíclicos, 651  
Dijkstra  
    caminos mínimos, 673–683  
    Triple partición del quicksort, 184  
dipolos, 123
- dispersión de cadenas de caracteres, 429  
dispersión por división, 425  
dispersión por multiplicación, 426  
dispersión universal, 430  
dividir combinar, 168  
    coste en espacio, 177  
`dmalloc`, 209  
`DynArray<T>`  
    el problema fundamental, 155  
`DynSetTree`, 332
- edad de un nodo, 227  
el problema del estacionamiento, 379  
eliminación  
    en ABB, 324  
    árbol binario de búsqueda, 324  
eliminación de ciclos negativos, 796  
eliminación de la recursión, 116  
eliminación de la recursión cola, 116  
eliminación de la recursión por emulación de los registros de activación, 117  
eliminación en tabla hash lineal, 418  
eliminación por clave en árboles binarios extendidos, 342  
eliminación por posición en árboles binarios extendidos, 343  
eliminación total de la recursión, 121  
eliminación en árbol árbol rojo-negro, (496, 502)  
eliminación en árbol aleatorizado, 457  
eliminación en treap, 468  
emparejamiento, 775  
    bipartido, 775  
    de cardinalidad máxima, 775  
    máximo, 775  
    perfecto, 775  
emparejamiento bipartido  
    flujo máximo, 776  
Engler, 201  
enlace doble, 72  
enlace simple, 65  
enseñanza  
    etimología, i  
enumeración de árboles, 302  
equilibrio perfecto de Wirth, 452  
equivalencia  
    de árboles binarios, 251

- equivalencia entre red y corte, 715  
 error de compilación, 200  
 errores de memoria  
   acceso fuera de bloque, 208  
   fuga, 207  
 espacio global de estado, 201  
 estabilidad  
   en ordenamiento, 172  
   mergesort, 172  
 estacionamiento, 379  
 Estructura de datos  
   problema fundamental, 17–20  
 etimología  
   amortizar, 192  
   análisis, 145  
   balanza, 451  
   crédito, 192  
   crítica, 145  
   equilibrio, 451  
   genérico, 17  
   genealogía, 223  
   general, 17  
   grafo, 533  
   método, 8  
   objeto, 5  
   polimorfo, 13  
   recurrencia, 2  
   recursión, 2  
   síntesis, 145  
   sujeto, 5  
 evaluación de expresiones infijas, 106  
 evaluación de expresión sufija, 107  
 exceso de flujo, 734  
 expansión de tabla hash lineal, 413  
 extensiones a los árboles binarios, 335  
   cálculo de la posición infija, 338  
 extensiones a los árboles binarios símbolos  
   desempeño, 343  
 extensiones a los árbol binario símbolos  
   inserción por clave, 338  
 extensiones a los ABB  
   selección, 337  
 Fibonacci, 116  
 fin, 22  
 flawfinder, 201  
 Floyd-Warshall  
   caminos mínimos, 683–690  
 flujo  
   teorema de descomposición, 782  
   valor de, 707  
 flujo factible, 707  
 flujo máximo  
   algoritmo genérico de empuje de pre-flujo, 735  
   algoritmo genérico de preflujo, 735  
   valores iniciales de altura, 740  
 algoritmo por empuje de preflujo genérico  
   por arcos  
   selección aleatoria, 763  
   selección por prioridad, 763  
   selección por profundidad, 762  
 algoritmo por empuje de preflujo genérico  
   por arcos, 758  
   selección por amplitud, 763  
 algoritmos de empuje y preflujo, 734  
 algoritmos de preflujo  
   altura de nodo, 735  
   altura de nodo, 735  
   análisis del algoritmo genérico de pre-flujo, 744  
   cálculo del corte mínimo, 764  
   con capacidades en los nodos, 772  
   de coste mínimo, 791  
   emparejamiento bipartido, 776  
   empuje de preflujo con cola aleatoria, 753  
   empuje de preflujo con cola de priori-dad, 749  
   empuje de preflujo con cola FIFO, 746  
   empuje de preflujo con mayor distancia, 749  
   implantación del algoritmo genérico de preflujo, 742  
   nodo activo, 734  
   provisión de flujo, 772  
   reducción desde una red no dirigida, 771  
 flujo máximo coste mínimo  
   asignación, 807  
   cadenas de producción, 824  
   el problema de asignación, 807  
   problema del transporte, 802  
   problema del trasbordo, 805

- flujo máximo/corte mínimo, 716, 764  
flujo realizable, 772  
flujos de coste mínimo, 790–827  
    conversión a programa lineal, 820  
forma estándar de un programa lineal, 810  
forma holgada de un programa lineal, 812  
Fortran, 59  
fuente, 637, 706  
fugas de memoria, 207  
función de Fibonacci, 116  
función hash  
    holgura de dispersión, 423  
    método de división, 425  
    método de multiplicación, 426  
funciones hash, 423  
gdb, 210  
genérico, 17  
generación de un nodo, 227  
general, 17  
grado  
    de un grafo, 536  
grafo  
    árbol abarcador, 538  
    bipartido, 538  
    camino, 536  
    ciclo, 536  
    clique, 537  
    complemento de, 537  
    completamente conexo, 537  
    completo, 537  
    con pesos, 535  
    conexo, 536  
    dirigido, 536  
    inconexo, 537  
    nodos  
        iterador de arcos, 548  
        iterador de nodos, 548  
    subgrafo, 537  
    totalmente conexo, 537  
grafo bipartido  
    cálculo de, 777  
grafo bipartido a red de flujo, 776  
grafos  
    coloración, 539  
    conectividad, 587  
    corte, 714  
    definición formal, 535  
    diámetro, 706  
    emparejamiento, 775  
Graph\_Node, 545  
heap  
    BinHeap<Key>, 293  
    aplicaciones, 291  
    colas de prioridad, 288  
        actualización de un elemento, 289  
    con árboles, 293  
    con listas enlazadas, 293  
    eliminación, 285  
    inserción, 284  
    heap de arcos mínimos, 665  
    heap exclusivo, 665  
    heaps, 282  
        con arreglos, 283  
    heapsort, 289  
    Heisenberg, 208  
    heisenbug, 208  
    herencia, 11  
        tipos, 12  
    herencia múltiple, 12  
    hilado de árboles binarios, (255, 260  
    holgura de dispersión de una función hash, 423  
Holtzmann, 196  
Huffman  
    códigos, 345–358  
    codificación de texto, 353  
    decodificación, 349  
    ingreso de frecuencias, 352  
    optimación, 355  
indentación  
    representación de árboles, 230  
inducción, 22  
LhashTable<Key>, 386  
inserción en árbol binario de búsqueda, 320  
inserción en raíz  
    en árbol binario de búsqueda, 326  
    en ABB, 326  
    en ABRE, 340  
    en binario de búsqueda extendido, 340  
inserción en tabla hash con resolución de colisiones por encadenamiento separado, 340

- rado, 386
- inserción en tabla hash lineal, 418
- inserción por clave en árbol binario extendido, 338
- inserción por posición de árboles binarios extendidos, 341
- inserción
  - método de ordenamiento, 162, 165
- inserción en árbol aleatorizado, 456
- inserción en árbol árbol rojo-negro, (492, 496)
- inserción en treap, 466
- iterador filtro
  - de arcos de nodo, 582
  - de arcos de un grafo, 582, 583
- interfaz de la función hash, 423
- invariantes, 204
- Iterador
  - de arcos de un grafo
    - implantación, 563
  - de arcos de un nodo de grafo, 568
  - de nodos de un grafo
    - implantación, 563
  - sobre Dlink, 80
  - sobre Dnode<T>, 84
- iterador
  - de arcos de un nodo de grafo, 548
  - de nodos sobre un grafo, 548
  - lista simplemente enlazada, 70
- Iterador de arcos sobre un nodo
  - implantación, 568
- iteradores, 59–61
- iteradores filtros, 580–583
- its4, 201
- $K_e(G)$ 
  - algoritmo, 784
  - cálculo, 784
  - implementación del algoritmo de cálculo de  $K_e(G)$ , 785
- Korajasu
  - algoritmo de cálculo de componentes fuertemente conexos, 638
- Kruskal, 659
  - análisis, 661
  - correctitud, 662
- lazo, 536
- lema
  - de la unicidad de un treap, 465
- lint, 201
- listas
  - doblemente enlazadas
    - Dlink, 72
    - Dnode<T>, 83
    - DynDlist<T>, 84
    - enlace doble, 72
    - nodo doble, 83
  - doblemente enlazadas dinámicas, 84
  - simplemente enlazadas
    - DynSlist<T>, 71
    - enlace simple, 65
    - iterador, 70
    - nodo simple, 68
    - Slink, 65, 68
    - Slist<T>, 69
  - listas de adyacencia, 542
    - en List\_Graph, 564
  - listas enlazadas, 61
    - para representar árboles, 231
- List\_Graph, 543
  - asignación, 560
  - construcción, 560
  - destrucción, 560
  - implementación de la copia de grafos, 574
  - implementación de la eliminación de arcos, 571
  - implementación de la eliminación de nodos, 572
  - implementación de la inserción de arcos, 570
  - implementación de la inserción de nodos, 569
  - implementación de la limpieza de un grafo, 573
  - implementación del mapeo de nodos y arcos de un grafo, 573
- livelock, 202
- localidad de referencia, 212, 214
  - espacial, 212, 214
  - temporal, 212, 214
- longitud
  - del camino en árboles, 226

- longitud del camino  
definición, 277  
externo  
fórmula en función del camino interno, 279  
identidades, 278  
interno  
límites, 280  
longitud del camino interna ponderada, 358  
Lukasiewicz, 302  
máximo emparejamiento bipartido, 775–781  
método de división como función hash, 425  
método de multiplicación como función hash, 426  
métodos virtual, 13  
mínimo coste de flujo máximo, 791  
macros para grafos, 559  
map, 20  
mapeo, 20  
arcos, 558  
nodos, 558  
mapeo de componentes conexos, 624  
mapping, 20  
matching, 775  
matemáticas sobre árboles, (275, 311)  
matrices, 58  
matriz  
cálculo de dirección, 59  
en Fortran, 59  
Matriz de adyacencia, 540  
matriz de adyacencia, 626  
memory leaks, 207  
mergesort  
análisis, 171  
crítica, 171  
meta-compilación, 201  
metacompilación, 201  
mezcla  
método de ordenamiento, 169–173  
mezcla de arreglos, 170  
modificador, 9  
multiconjunto, 20  
multigrafo, 537  
multiherencia, 12  
multilistas, 131  
multimap, 20  
multipop, 102  
multiset, 20  
número de Catalan, 307–311  
Número de Dewey  
cálculo, 272  
números de Fibonacci, 115  
nana, 204–207  
Net\_Graph, 708  
new, 29, 33  
nivel de un nodo en un árbol, 227  
Node\_Arc\_Iterator, 568  
nodo  
edad de, 227  
generación de, 227  
grado de un nodo en árbol, 227  
nodo incidente, 536  
nodo activo, 734  
nodo adyacente, 536  
Nodo de un grafo, 545  
nodo externo  
cantidad en un árbol, 277  
definición, 277  
nodo simple, 68  
notación  
de Dewey, 231  
notación  $\mathcal{O}$ , 157–168  
 $\mathcal{O}$   
definición, 157  
 $\circ$ , 158  
 $\Omega$ , 158  
 $\omega$ , 158  
 $\Theta$ , 159  
notación infija, 106  
notación prefija, 106  
notación sufija, 106  
número df, 611  
 $\mathcal{O}$ , 157–168  
Álgebra de  $\mathcal{O}$ , 159–160  
análisis de algoritmos mediante  $\mathcal{O}$ , 160–165  
errores, 166–167  
 $\circ$ , 158  
observador, 9  
ocultamiento de información, 23  
 $\Omega$ , 158

- $\omega$ , 158
- operador
  - delete, 33
  - new, 29, 33
- orden
  - de un árbol, 227
- ordenamiento
  - estabilidad, 172
  - inserción, 162, 165
  - mezcla, 169–173
  - selección, 150–153
- ordenamiento de arcos sobre un grafo, 575
- Ordenamiento de listas enlazadas, 152
- ordenamiento topológico, 653–658
- orientación
  - hacia el concepto, 21
  - hacia el flujo, 21
  - hacia los datos, 21
- paradoja
  - cumpleaños, 381
- Pareto Vilfredo, 212
- partición de árboles binarios, 321
- partición de ABB, 321
- partición de ABBE, 339
- partición del quicksort, 174
- partición por clave de árboles binarios extendidos, 339
- partición por posición de árboles binarios, 340
- paso de ejecución, 148
- perfilaje, 214
- perl, 215
- pila
  - representaciones en memoria, 99
- pilas
  - con arreglos, 101
  - con listas enlazadas, 103
  - con listas enlazadas dinámicas, 105
  - llamadas a procedimientos, 112
  - recursión, 112
- pintado de componentes conexos, 622
- pivote
  - selección en el quicksort, 179
- plantillas, 15
- plegado de clave en función hash, 424
- polimorfismo, 13
- de herencia, 14
- de sobrecarga, 13
- polinomios
  - implantación con listas enlazadas, 91
  - suma, 95
- Preguntas de Van Cleck, 199
- Prim, 664
  - análisis, 670
  - correctitud, 672
- principio
  - de Pareto, 212
  - del 80-20, 212
- principio fin-a-fin, 22
- prioridades implícitas en treaps, 471
- problema del transporte, 802
- problema del trasbordo, 805
- problema del vendedor viajero, 212
- problema fundamental de estructuras de datos, 17
- profiling, 214
- programación lineal, 808–820
  - conversión desde una red capacitada, 820
  - forma estándar, 810
  - forma holgada, 812
  - forma slack, 812
  - simplex, 813
- promela/spin, 203
- propiedades matemáticas de los árboles, (275, 311)
- proxy, 49
- prueba de aciclicidad, 591
- prueba de conectividad, 587, 636
- prueba de existencia de camino, 594, 633
- punteros y arreglo, 29
- punto de articulación, 611
- punto de corte, 611
  - pintado de componentes conexos, 622
- puntos de corte
  - componentes conexos, 620
  - mapeo de componentes conexos, 624
- python, 215
- quicksort, 173–187
  - análisis, 176
  - claves repetidas, 183
  - con listas enlazadas, 181

- crítica, 176  
selección del pivote, 179  
sin recursión, 180  
triple partición de Dijkstra, 184
- quicksort concurrente, 184  
quicksort paralelo, 184
- Rabin-Karp  
    búsqueda de subcadenas, 431
- rápido  
    método de ordenamiento, 187
- realloc, 34
- recorrido  
    infijo  
        sobre árbol binario, 235  
    por niveles  
        sobre árbol binario, 235, 251  
    prefijo  
        sobre arborescencia, 262  
        sobre árbol binario, 235  
    sufijo  
        sobre árbol binario, 235  
        sobre arborescencia, 262
- recorrido en amplitud, 587
- recorrido en profundidad, 583
- recorrido enorden, *see* recorrido infijo
- recorrido postorden, *see* recorrido sufijo
- recorrido preorden, *see* recorrido prefijo
- recorridos  
    sobre arborescencias, 262, 270
- recorridos pseudo-hilados sobre árboles binarios, 258
- recorridos sobre árboles binarios, 234
- recorridos sobre grafos, 579–626
- recursión  
    consejos, 115  
    eliminación, 116  
        emulación de los registros de activación, 117  
    eliminación total, 121  
    pilas, 112
- recursión cola  
    eliminación, 116
- red, 706  
    corte de, 714  
    residual, 721
- red residual  
    en red capacitada con costes, 792
- red capacitada  
    cálculo de camino de aumento, 723
- red capacitada con costes, 790
- red con capacidades en los nodos, 772
- red residual, 719
- redes  
    camino de aumento  
        incremento de flujo, 723  
    capacidad, 706  
    fuente, 706  
    prefljo, 734  
    sumidero, 706  
    valor de flujo, 715
- redes capacitadas  
    algoritmo de Ford-Fulkerson, 724  
    algoritmo de Edmonds-Karp, 729  
    caminos de aumento, 718  
    con capacidades en los nodos, 772  
    con grafos no dirigidos, 771  
    corte mínimo, 716  
    cortes, 714  
    definiciones, 706  
    flujo máximo, 716  
    incremento de flujo por camino de aumento, 723  
    manejo de varios fuentes o sumideros, 709  
    operaciones topológicas, 711  
    red residual, 721
- redes capacitadas con provisión de flujo, 772
- redes capacitadas generalizadas, 821
- Redes con restricciones laterales, 822
- redes de flujo, 706–771  
    circulaciones, 781
- redes de procesamiento, 823
- Redes multi-flujo, 823
- reducciones al flujo mínimo, 771–790
- registro de activación, 114
- regla  
    de Pareto, 212  
    del 80-20, 212
- relación  
    simétrica, 535
- relajación de arco, 693
- remove\_all\_and\_delete(), 82

- representación de árboles  
 conjuntos anidados, 228  
 indentación, 230  
 secuencia parentizada, 229
- representación de árboles con arreglos, 232
- representación de árboles con listas enlazadas, 231
- Representación de un árbol abarcador en un arreglo, 604
- representaciones en memoria de una cola, 123
- rotación en árbol binarios, 344
- rotación en árboles binarios extendidos, 345
- ruby, 215
- secuencia parentizada para representar árboles, 229
- Sedgewick, 665
- selección  
 método de ordenamiento, 150–153  
 por posición en árbol binario, 337
- selección  
 por posición en arreglo, 186  
 sobre arreglos, 186
- set, 20
- sift\_down, 285
- sift\_up, 284
- similaridad de árboles binarios, 251
- simplex, 813–820
- sistema de tipos, 200
- Skiiena, 210
- Skiiena, Steven, 210
- slack  
 en camino de aumento, 718  
 programa lineal, 812
- Slink, 65
- sondeo  
 lineal, 397
- sondeo ideal  
 análisis, 393
- spin/promela, 203
- subgrafo de un grafo, 537
- suma de polinomios, 95
- sumidero, 637, 706
- supertraza, 203, 434
- supresión en árbol binario de búsqueda, (324, 326)
- supresión en árbol aleatorizado, 457
- supresión en treap, 468
- tabla hash  
 holgura de dispersión, 423  
 interfaz de la función hash, 423  
 plegado de clave en función hash, 424
- tablas hash  
 dispersión de cadenas de caracteres, 429  
 dispersión universal, 430  
 funciones, 423  
 funciones hash, 423  
 manejo de colisiones, 381  
 direcccionamiento abierto, 393  
 encadenamiento, 383  
 encadenamiento separado, 383  
 manejo de colisiones por direcccionamiento abierto y sondeo lineal  
 análisis, 399
- re-dimensionamiento, 409
- reajuste de dimensión, 409
- supertraza, 434
- tablas hash lineales, 412–422  
 búsqueda, 418  
 eliminación, 418  
 expansión, 413, 417  
 inserción, 418
- tablas hash perfectas, 431
- TAD  
 Ady\_Mat, 627–631  
 Bit\_Mat\_Graph<GT>, 631–633  
 DynMapTree, 332–335  
 List\_Graph, 543  
 ArrayStack<T>, 101–103  
 ArrayQueue<T>, 124–128  
 Avl\_Tree<Key>, 472–483  
 BinNode<Key>, (238, 242)  
 BinNodeXt<Key>, (336, 345)  
 BinTree<Key>, (319, 329)  
 Dlink, 72–83  
 Dnode<T>, 83–84  
 DynArray<T>, 34–53  
 DynDlist<T>, 84–91  
 DynListQueue<T>, 130–131  
 DynListStack<T>, 105–106  
 DynSlist<T>, 71–72  
 ListQueue<T>, 128–130

- ListStack<T>, 103–105  
Rand\_Tree<Key>, 455–460  
Rb\_Tree<Key>, 491–502  
Slink, 65–68  
Slist<T>, 69–71  
Snode<T>, 68–69  
Splay\_Tree<Key>, 508–513  
Treap<Key>, 465–470  
Tree\_Node , (264, 275)  
Tarjan  
    algoritmo para cálculo de componentes fuertemente conexos, 640  
templates, 15  
teorema  
    de descomposición de flujo, 782  
teorema del flujo máximo/corte mínimo, 717  
test de aciclicidad, 591  
 $\Theta$ , 159  
thrashing, 444  
tipos de estructura de datos, 21  
tipos de errores, 194  
Tom Van Cleck, 199  
Treap  
    definición, 464  
treaps, 464–471  
    análisis, 470–471  
    eliminación, 468  
    inserción, 466  
    prioridades implícitas, 471  
Tree\_Node  
    búsqueda por número de Dewey, 271  
    conversión a BinNode<Key>, 273  
    destrucción, 270  
    modificadores, 267  
    observadores, 266  
    observadores de arborescencias, 269  
    recorridos, 270  
UML, 10  
unión de árboles binarios, 327  
unión de ABB, 327  
unión exclusiva de árboles binarios, 323  
unión exclusiva de árboles binarios extendidos, 342  
unión exclusiva de ABB, 323  
unidad de ejecución, 148  
    uniión exclusiva de árboles binarios aleatorizados, 458  
    valgrind, 209  
    valor de flujo, 707, 715  
    vendedor viajero, 212  
    virtual, 13  
    Warshall, 595  
    algoritmo para la clausura transitiva, 633  
Wirth  
    equilibrio perfecto, 452



# Índice de identificadores

Activation\_Record: [118b](#), [118e](#)  
ARC\_DIST: [679b](#), [680c](#), [681b](#)  
ArcHeap: [668a](#), [670b](#), [671a](#), [679a](#), [679d](#)  
Arc\_Node: [567a](#), [568a](#), [568b](#), [569](#), [570c](#), [572](#), [573](#), [574](#)  
ArrayQueue: [125a](#), [126c](#), [251c](#)  
ArrayStack: [101a](#), [101c](#), [110a](#), [110b](#), [118e](#), [246](#), [247](#), [248a](#), [248b](#), [509b](#)  
AvlNode: [472](#), [473a](#)  
binary\_search: [30](#), [31a](#), [32](#)  
BinNode: [240](#), [319b](#), [347c](#), [348a](#), [348c](#), [349d](#), [350b](#), [351a](#), [351b](#), [352c](#), [509a](#)  
BinNodeVtl: [240](#)  
BitArray: [54a](#), [55c](#), [56b](#), [56c](#), [57a](#), [311](#), [312a](#), [312b](#), [313b](#), [349a](#), [349d](#), [351b](#), [351c](#), [353b](#),  
[354](#), [633a](#)  
Bit\_Fields: [556a](#), [557a](#), [557b](#), [557c](#), [557f](#)  
Cache\_Entry: [437c](#), [438a](#), [440a](#), [440b](#), [440c](#), [441a](#), [441b](#), [441c](#), [443a](#), [443b](#), [443c](#), [443d](#),  
[444a](#), [444b](#), [445](#)  
cache\_size: [439b](#), [440b](#), [444b](#)  
call\_hash\_fct: [415b](#), [418a](#), [418b](#)  
chunk\_list: [440a](#), [440b](#), [444b](#)  
Compare\_Dnode: [153a](#), [153b](#), [165a](#)  
contract: [417c](#), [418c](#)  
copy\_list\_graph: [634a](#), [634b](#)  
Cross\_Arc: [623](#), [624](#), [625b](#)  
DECLARE\_BINNODE: [240](#), [295c](#), [472](#)  
DECLARE\_BINNODE\_SENTINEL: [336](#), [456b](#), [465](#), [491](#)  
Dijkstra\_Arc\_Info: [678b](#), [678c](#), [679b](#)  
Distance\_Compare: [661a](#), [662a](#), [668a](#), [668b](#), [669a](#)  
dlink\_lru: [437e](#), [438a](#), [438b](#)  
DLINK\_TO\_TYPE:  
Dnode: [83a](#), [83c](#), [84](#), [85a](#), [85d](#), [86b](#), [87b](#), [88a](#), [88c](#), [88d](#), [89a](#), [89b](#), [89c](#), [89d](#), [90a](#), [90b](#), [91b](#),  
[153a](#), [153b](#), [154b](#), [155a](#), [165a](#), [185](#), [384b](#), [385c](#), [440a](#)  
do\_mru: [441a](#), [441c](#), [443b](#)  
doubleRotateLeft: [477a](#)  
doubleRotateRight: [477a](#)  
DynArray: [34](#), [42](#), [43c](#), [45a](#), [49b](#), [49c](#), [50a](#), [54b](#), [155b](#), [253](#), [315a](#), [361a](#), [361d](#), [362c](#), [363](#),  
[607](#), [630a](#), [633c](#), [641](#), [693b](#), [704](#), [756](#)  
DynDlist: [85a](#), [85c](#), [85e](#), [87a](#), [87c](#), [87d](#), [88a](#), [88d](#), [89c](#), [90b](#), [90c](#), [91a](#), [91b](#), [91c](#), [94a](#), [95d](#),  
[98a](#), [98b](#), [155a](#), [207](#), [254b](#), [255](#), [579b](#), [580c](#), [581b](#), [610b](#), [611b](#), [615b](#), [617b](#), [617c](#), [626a](#),

628, 644, 645b, 646c, 647a, 647b, 648, 656, 657a, 657b, 659, 706, 712b, 769, 780, 781b, 782b, 788a, 789, 791  
DynListStack: [105c](#), 106c, 188, 645b, 647b, 649a, 649b, 650, 652b, 764b  
expand: [417a](#), 418b, 444b  
Figure: [8a](#), 8b, 9a, 12, 15a  
Filter\_Iterator: [582a](#), 583b, 584a, 584b, 585a  
find\_succ\_and\_swap: 497  
fix\_black\_condition: 497, [499](#)  
FixedStack: [101a](#), 180c, 473b, 492b  
fix\_red\_condition: 493, [494](#)  
get\_lru\_entry: [441c](#), 443a  
get\_min\_arc: [669b](#), 671c, 681a  
inconnected\_components: 610b, [611b](#)  
insert\_entry\_to\_lru\_list: 440b, [440c](#), 443d, 444b  
insert\_in\_binary\_search\_tree: [321a](#), 321b, 327  
insert\_root: [326b](#), 327  
insert\_sorted: [164a](#), 164b  
inside\_list: [439b](#), 443a, 445  
is\_red\_black\_tree: 493  
join\_exclusive: [324](#), 325, 513  
join\_exclusive\_xt: [342a](#), 342b, 343  
List\_Digraph: [547a](#), 562d, 711b  
List\_Graph: [546a](#), 547a, 550a, 555, 561b, 562a, 562b, 562c, 562d, 564d, 565a, 565b, 566b, 566d, 571b, 571c, 572, 573, 574, 576a, 577b  
ListQueue: [128h](#), 129a, 130a, 131  
lru\_list: [438c](#), 440c, 441a, 441b, 441c, 444b  
mapped\_node: [560b](#), 605, 607, 612, 626b, 628, 638b, 646c, 663a, 663b, 683b, 701, 704, 781b, 787, 790, 791  
merge: 169, [171](#)  
mergesort: 169, [173a](#), 577b  
Null\_Value: 630a, 630b, 630d, 631a  
partition: 173b, [175b](#), 177, 180c, 182b, 184, 186a  
Path: [578a](#), 579c, 597b, 598a, 598b, 599, 600a, 600c, 600d, 601, 602, 650, 682, 683b, 683c, 687a, 687b, 692, 704, 705, 706, 726, 727a, 770b, 771a, 771b, 798, 801  
Path\_Desc: [579a](#), 579b, 579d, 580a, 580c, 581b  
POT: [679b](#), 679c, 680c, 681b  
Prim\_Heap\_Info: [670b](#), 671a  
push\_in\_splay\_stack: [509c](#), 510a, 511b, 512b  
put\_arc: [669a](#), 671b, 671c, 680c, 681b  
random\_insert: [456f](#), 457b  
random\_join\_exclusive: [458a](#), 458c  
random\_remove: [458c](#), 459a  
RbNode: [491](#), 492a  
remove\_entry\_from\_hash\_table: [441c](#), 444a  
remove\_entry\_from\_lru\_list: [440c](#), 443c  
remove\_from\_search\_binary\_tree: [325](#), 326a, 327

rotateLeft: 477a  
rotateRight: 477a  
search\_and\_push\_in\_splay\_stack: 510a, 512a, 512b, 513  
search\_and\_stack\_rb: 492c, 493, 497  
searchInBinTree: 316a, 320b  
search\_min: 151c, 152, 156b  
sequential\_search: 15b, 31b, 154a, 154b, 155a, 155b  
Slink: 65a, 65b, 65c, 66a, 66b, 68a, 68b, 69d, 69e, 72  
Snode: 68b, 69c, 69d, 69e, 69f, 69g, 71c, 103h, 104b, 116a, 116b, 129a  
splay: 511b, 512a, 512b, 513  
split\_key\_rec: 322, 323, 326b  
split\_key\_rec\_xt: 339, 340a  
split\_pos\_rec: 340b, 341  
Termino: 93c, 94a, 94b, 94c, 94d, 94e, 94f, 95d, 96b, 96c, 97b, 98a, 98b  
TREEARC: 679b, 681a  
zig\_type: 511a, 511b