Algoritmos y Estructuras de Datos

Bottazzi, Cristian. cristian.bottazzi@gmail.com,
Costarelli, Santiago. santi.costarelli@gmail.com,
D'Elía, Jorge. jdelia@intec.unl.edu.ar,
Dalcin, Lisandro. dalcinl@gmail.com,
Galizzi, Diego. dgalizzi@gmail.com,
Giménez, Juan Marcelo. jmarcelogimenez@gmail.com,
Olivera, José. joseolivera123@gmail.com,
Novara, Pablo. zaskar_84@yahoo.com.ar,
Paz, Rodrigo. rodrigo.r.paz@gmail.com,
Prigioni, Juan. jdprigioni@gmail.com,
Pucheta, Martín. martinpucheta@gmail.com,
Rojas Fredini, Pablo Sebastián. floyd.the.rabbit@gmail.com,
Romeo, Lautaro. lau337.r@gmail.com ,
Storti, Mario. mario.storti@gmail.com,

www: http://www.cimec.org.ar/aed
Facultad de Ingeniería y Ciencias Hídricas
Universidad Nacional del Litoral http://fich.unl.edu.ar
Centro de Investigación de Métodos Computacionales
http://www.cimec.org.ar

Indice

Sobre este libro:

Este libro corresponde al curso *Algoritmos y Estructura de Datos* que se dicta en la currícula de *Ingeniería Informática* de la *Facultad de Ingeniería y Ciencias Hídricas* (http://www.fich.unl.edu.ar) de la *Universidad Nacional del Litoral* (http://www.unl.edu.ar).

Página web del curso: La página web del curso es http://www.cimec.org.ar/aed. En esa página funciona un wiki, listas de correo y un repositorio de archivos donde se puede bajar la mayor parte del código que figura en el libro. Este libro se puede bajar en formato PDF de esa página también.

Utilitarios usados: Todo este libro ha sido escrito con utilitarios de software libre, de acuerdo a los lineamientos de la *Free Software Foundation/GNU Project* (http://www.gnu.org). La mayoría de los utilitarios corresponden a un sistema **Fedora release 27 (Twenty Seven) Kernel 4.14.16-300.fc27.x86_64**.

- El libro ha sido escrito en LaTeX y convertido a PDF con **pdflatex**. El libro está completamente inter-referenciado usando las utilidades propias de LaTeX y el paquete **hyperref**.

- Muchos de los ejemplos con un matiz matemáticos han sido parcialmente implementados en *Octave* (http://www.octave.org). También muchos de los gráficos.

- Los ejemplos en C++ han sidos desarrollados y probados con el compilador **gcc version 7.3.1 20180130 (Red Hat 7.3.1-2) (GCC)** (http://gcc.gnu.org) y con la ayuda de **GNU Make 4.2.1** http://www.gnu.org/software/make/make.html.

- Las figuras han sido generadas con *Inkscape 0.92.2 (5c3e80d, 2017-08-06)* (http://inkscape.sourceforge.net/http://inkscape.sourceforge.net/).
- El libro ha sido escrito en forma colaborativa por los autores usando Git 2.14.3 (http://git-scm.com/).

Capítulo 1

Diseño y análisis de algoritmos

1.1. Conceptos básicos de algoritmos

No existe una regla precisa para escribir un programa que resuelva un dado problema práctico. Al menos por ahora escribir programas es en gran medida un arte. Sin embargo con el tiempo se han desarrollado un variedad de conceptos que ayudan a desarrollar estrategias para resolver problemas y comparar *a priori* la eficiencia de las mismas.

Por ejemplo supongamos que queremos resolver el *"Problema del Agente Viajero"* (TSP, por *"Traveling Salesman Problem"*) el cual consiste en encontrar el orden en que se debe recorrer un cierto número de ciudades (esto es, una serie de puntos en el plano) en forma de tener un recorrido mínimo. Este problema surge en una variedad de aplicaciones prácticas, por ejemplo encontrar caminos mínimos para recorridos de distribución de productos o resolver el problema de *"la vuelta del caballo en el tablero de ajedrez"*, es decir, encontrar un camino para el caballo que recorra toda las casillas del tablero pasando una sola vez por cada casilla. Existe una estrategia (trivial) que consiste en evaluar todos los caminos posibles. Pero esta estrategia de *"búsqueda exhaustiva"* tiene un gran defecto, el costo computacional crece de tal manera con el número de ciudades que deja de ser aplicable a partir de una cantidad relativamente pequeña. Otra estrategia *"heurística"* se basa en buscar un camino que, si bien no es el óptimo (el de menor recorrido sobre todos los posibles) puede ser relativamente bueno en la mayoría de los casos prácticos. Por ejemplo, empezar en una ciudad e ir a la más cercana

que no haya sido aún visitada hasta recorrerlas todas.

Una forma abstracta de plantear una estrategia es en la forma de un *"algoritmo"*, es decir una secuencia de instrucciones cada una de las cuales representa una tarea bien definida y puede ser llevada a cabo en una cantidad finita de tiempo y con un número finito de recursos computacionales. Un requerimiento fundamental es que el algoritmo debe terminar en un número finito de pasos, de esta manera él mismo puede ser usado como una instrucción en otro algoritmo más complejo.

Entonces, comparando diferentes algoritmos para el TSP entre sí, podemos plantear las siguientes preguntas

- ¿Da el algoritmo la solución óptima?
- Si el algoritmo es iterativo, ¿converge?
- ¿Cómo crece el esfuerzo computacional a medida que el número de ciudades crece?

1.1.1. Ejemplo: Sincronización de acceso a objetos en cálculo distribuido

Consideremos un sistema de procesamiento con varios procesadores que acceden a un área de memoria compartida. En memoria hay una serie de objetos O_0, O_1, ... O_{n-1}, con $n = 10$ y una serie de tareas a realizar T_0, T_1, ... T_{m-1} con $m = 12$. Cada tarea debe modificar un cierto subconjunto de los objetos, según la siguiente tabla

- T_0 modifica O_0, O_1 y O_3.
- T_1 modifica O_4 y O_5.
- T_2 modifica O_4.
- T_3 modifica O_2 y O_6
- T_4 modifica O_1 y O_4.
- T_5 modifica O_4 y O_7.
- T_6 modifica O_0, O_2, O_3 y O_6.
- T_7 modifica O_1, O_7, O_8.
- T_8 modifica O_5, O_7 y O_9.
- T_9 modifica O_3.
- T_{10} modifica O_6, O_8 y O_9.
- T_{11} modifica O_9.

Las tareas pueden realizarse en cualquier orden, pero dos tareas *no pueden ejecutarse al mismo tiempo si acceden al mismo objeto*, ya que los cambios hechos por una de ellas puede interferir con los cambios hechos por la otra. Debe entonces desarrollarse un sistema que sincronice entre sí la ejecución de las diferentes tareas.

Una forma trivial de sincronización es ejecutar cada una de las tareas en forma secuencial. Primero la tarea T_0 luego la T_1 y así siguiendo hasta la T_{11}, de esta forma nos aseguramos que no hay conflictos en el acceso a los objetos. Sin embargo, esta solución puede estar muy lejos de ser óptima en cuanto al tiempo de ejecución ya que por ejemplo T_0 y T_1 pueden ejecutarse al mismo tiempo en diferentes procesadores ya que modifican diferentes objetos. Si asumimos, para simplificar, que todas las tareas llevan el mismo tiempo de ejecución τ, entonces la versión trivial consume una cantidad de tiempo $m\tau$, mientras que ejecutando las tareas T_0 y T_1 al mismo tiempo reducimos el tiempo de ejecución a $(m-1)\tau$.

El algoritmo a desarrollar debe *"particionar"* las tareas en una serie de p *"etapas"* E_0, ... E_p. Las etapas son simplemente subconjuntos de las tareas y la partición debe satisfacer las siguientes restricciones

- Cada tarea debe estar en una y sólo una etapa. (De lo contrario la tarea no se realizaría o se realizaría más de una vez, lo cual es redundante. En el lenguaje de la teoría de conjuntos, estamos diciendo que debemos particionar el conjunto de etapas en un cierto número de subconjuntos *"disjuntos"*.)

- Las tareas a ejecutarse en una dada etapa no deben acceder al mismo objeto.

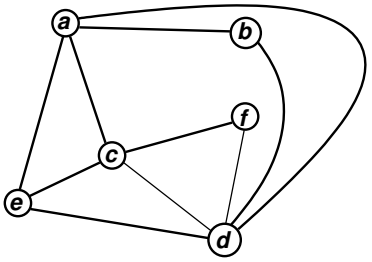
Una partición *"admisible"* es aquella que satisface todas estas condiciones. El objetivo es *determinar aquella partición admisible que tiene el mínimo número de etapas*.

Este problema es muy común, ya que se plantea siempre que hay un número de tareas a hacer y conflictos entre esas tareas, por ejemplo sincronizar una serie de tareas con maquinaria a realizar en una industria, evitando conflictos en el uso del instrumental o maquinaria, es decir no agendar dos tareas para realizar simultáneamente si van a usar el microscopio electrónico.

1.1.2. Introducción básica a grafos

El problema se puede plantear usando una estructura matemática conocida como *"grafo"*. La base del grafo es un conjunto finito V de puntos llamados *"vértices"*. La estructura del grafo está dada por las conexiones entre los vértices. Si dos vértices están conectados se dibuja una línea que va desde un vértice al otro. Estas conexiones se llaman *"aristas"* (*"edges"*) del grafo. Los vértices pueden identificarse con un número de 0 a $m-1$ donde m es el número total de vértices. También es usual representarlos gráficamente con un letra $a, b, c, ...$ encerrada en un círculo o usar cualquier etiqueta única relativa al problema.

Desde el punto de vista de la teoría de conjuntos un grafo es un subconjunto del conjunto G de pares de vértices. Un par de vértices está en el grafo si existe una arista que los conecta. También puede representarse como una matriz A simétrica de tamaño $m \times m$ con 0's y 1's. Si hay una arista entre el vértice i y el j entonces el elemento A_{ij} es uno, y sino es cero. Además, si existe una arista entre dos vértices i y j entonces decimos que i es *"adyacente"* a j.



	a	b	c	d	e	f
a	0	1	1	1	1	0
b	1	0	0	1	0	0
c	1	0	0	1	1	1
d	1	1	1	0	1	1
e	1	0	1	1	0	0
f	0	0	1	1	0	0

Figura 1.1: Representación gráfica y matricial del grafo G

En la figura 1.1 vemos un grafo con 6 vértices etiquetados de a a f, representado gráficamente y como una matriz de 0's y 1's. El mismo grafo representado como pares de elementos es

$$G = \{\{a, b\}, \{a, c\}, \{a, d\}, \{a, e\}, \{b, d\}, \{c, d\}, \{c, e\}, \{c, f\}, \{d, e\}, \{d, f\}, \} \tag{1.1}$$

Para este ejemplo usaremos *"grafos no orientados"*, es decir que si el vértice i está conectado con el j entonces el j está conectado con el i. También existen *"grafos orientados"* donde las aristas se representan por flechas.

Se puede también agregar un peso (un número real) a los vértices o aristas del grafo. Este peso puede representar, por ejemplo, un costo computacional.

1.1.3. Planteo del problema mediante grafos

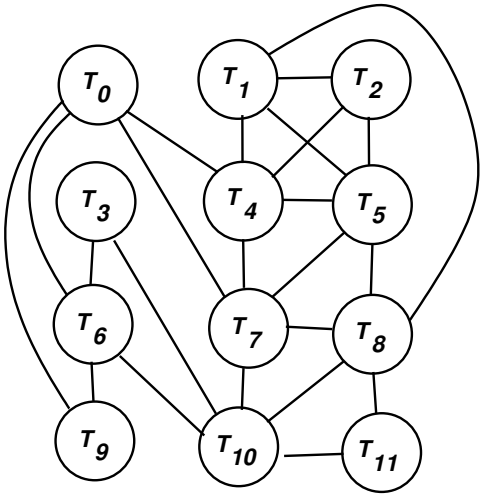


Figura 1.2: Representación del problema mediante un grafo.

Podemos plantear el problema dibujando un grafo donde los vértices corresponden a las tareas y dibujaremos una arista entre dos tareas si son incompatibles entre sí (modifican el mismo objeto). En este caso el grafo resulta ser como muestra la figura 1.2.

La buena noticia es que nuestro problema de particionar el grafo ha sido muy estudiado en la teoría de grafos y se llama el problema de *"colorear"* el grafo, es decir se representan gráficamente las etapas asignándole colores a los vértices del grafo. La mala noticia es que se ha encontrado que obtener el coloreado óptimo (es decir el coloreado admisible con la menor cantidad de colores posibles) resulta ser un problema extremadamente costoso en cuanto a tiempo de cálculo. (Se dice que es *"NP"*. Explicaremos esto en la sección §1.3.12.)

El término *"colorear grafos"* viene de un problema que también se puede poner en términos de colorear grafos y es el de colorear países en un mapa. Consideremos un mapa como el de la figura 1.3. Debemos asignar a cada país un color, de manera que países limítrofes (esto es, que com-

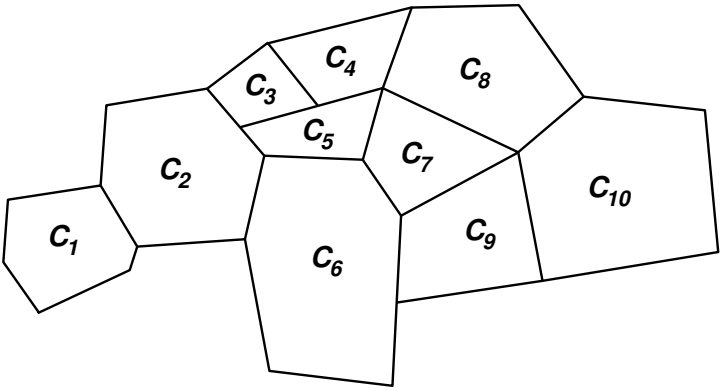


Figura 1.3: Coloración de mapas.

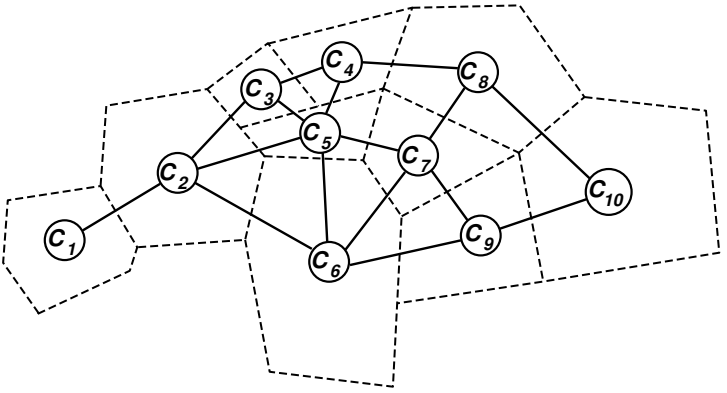


Figura 1.4: Grafo correspondiente al mapa de la figura 1.3.

parten una porción de frontera de medida no nula) tengan diferentes colores y, por supuesto, debemos tratar de usar el mínimo número de colores posibles. El problema puede ponerse en términos de grafos, poniendo vértices en los países (C_j, $j = 1..10$) y uniendo con aristas aquellos países que son limítrofes (ver figura 1.4).

1.1.4. Algoritmo de búsqueda exhaustiva

Consideremos primero un algoritmo de *"búsqueda exhaustiva"* es decir, probar si el grafo se puede colorear con 1 solo color (esto sólo es posible si no hay ninguna arista en el grafo). Si esto es posible el problema está resuel-

to (no puede haber coloraciones con menos de un color). Si no es posible entonces generamos todas las coloraciones con 2 colores, para cada una de ellas verificamos si satisface las restricciones o no, es decir si es admisible. Si lo es, el problema está resuelto: encontramos una coloración admisible con dos colores y ya verificamos que con 1 solo color no es posible. Si no encontramos ninguna coloración admisible de 2 colores entonces probamos con las de 3 colores y así sucesivamente. Si encontramos una coloración de n_c colores entonces será óptima, ya que previamente verificamos para cada número de colores entre 1 y $n_c - 1$ que no había ninguna coloración admisible.

Ahora tratando de resolver las respuestas planteadas en la sección §1.1, vemos que el algoritmo propuesto si da la solución óptima. Por otra parte podemos ver fácilmente que sí termina en un número finito de pasos ya que a lo sumo puede haber $n_c = m$ colores, es decir la coloración que consiste en asignar a cada vértice un color diferente es siempre admisible.

1.1.5. Generación de las coloraciones

En realidad todavía falta resolver un punto del algoritmo y es cómo generar todas las coloraciones posibles de n_c colores. Además esta parte del algoritmo debe ser ejecutable en un número finito de pasos así que trataremos de evaluar cuantas coloraciones $N(n_c, m)$ hay para m vértices con n_c colores. Notemos primero que el procedimiento para generar las coloraciones es independiente de la estructura del grafo (es decir de las aristas), sólo depende de cuantos vértices hay en el grafo y del número de colores que pueden tener las coloraciones.

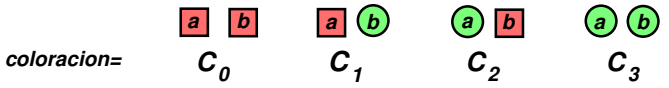


Figura 1.5: Posibles coloraciones de dos vértices con dos colores

Para $n_c = 1$ es trivial, hay una sola coloración donde todos los vértices tienen el mismo color, es decir $N(n_c = 1, m) = 1$ para cualquier m.

Consideremos ahora las coloraciones de $n_c = 2$ colores, digamos rojo y verde. Si hay un sólo vértice en el grafo, entonces hay sólo dos coloraciones posibles: que el vértice sea rojo o verde. Si hay dos vértices, entonces podemos tener 4 coloraciones rojo-rojo, rojo-verde, verde-rojo y verde-verde, es decir $N(2, 2) = 4$ (ver figura 1.5. *Nota: Para que los gráficos con co-*

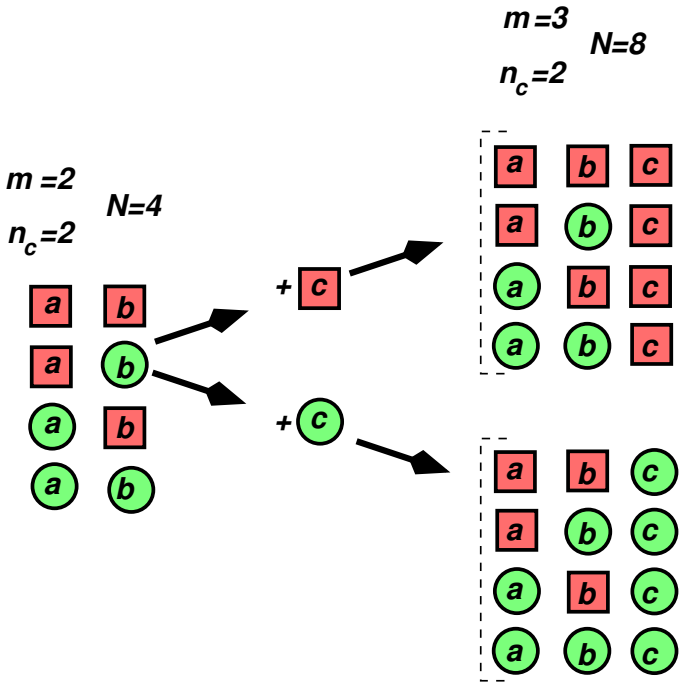


Figura 1.6: Las coloraciones de 3 vértices y dos colores se pueden obtener de las de 2 vértices.

lores sean entendibles en impresión blanco y negro hemos agregado una pequeña letra arriba del vértice indicando el color). Las coloraciones de 3 vértices a, b, c y dos colores las podemos generar a partir de las de 2 vértices, combinando cada una de las 4 coloraciones para los vértices a y b con un posible color para c (ver figura 1.6, de manera que tenemos

$$N(2,3) = 2\,N(2,2) \tag{1.2}$$

Recursivamente, para cualquier $n_c, m \geq 1$, tenemos que

$$\begin{aligned} N(n_c, m) &= n_c\, N(n_c, m-1) \\ &= n_c^2\, N(n_c, m-2) \\ &\vdots \\ &= n_c^{m-1}\, N(n_c, 1) \end{aligned} \tag{1.3}$$

Pero el número de coloraciones para un sólo vértice con n_c colores es n_c,

de manera que

$$N(n_c, m) = n_c^m \tag{1.4}$$

Esto cierra con la última pregunta, ya que vemos que el número de pasos para cada uno de los colores es finito, y hay a lo sumo m colores de manera que el número total de posibles coloraciones a verificar es finito. Notar de paso que esta forma de contar las coloraciones es también *"constructiva"*, da un procedimiento para generar todas las coloraciones, si uno estuviera decidido a implementar la estrategia de búsqueda exhaustiva.

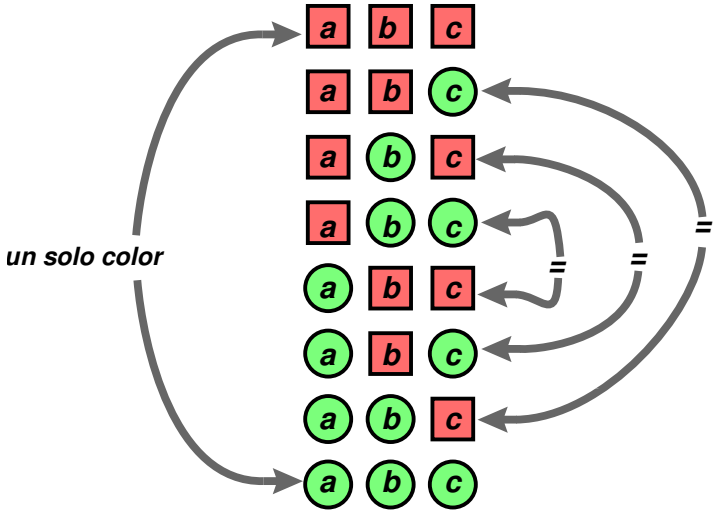


Figura 1.7: Las 8 posibles coloraciones de un grafo de 3 vértices con dos colores

Notemos que en realidad el conteo de coloraciones (1.4) incluye todas las coloraciones de n_c *o menos* colores. Por ejemplo si $m = 3$ y $n_c = 2$ entonces las $n_c^m = 2^3 = 8$ coloraciones son las que se pueden ver en la figura 1.7. En realidad hay dos (la primera y la última) que tienen un sólo color y las 6 restantes sólo hay 3 esencialmente diferentes, ya que son equivalentes de a pares, por ejemplo la segunda y la séptima son equivalentes entre sí de manera que una de ellas es admisible si y solo si la otra lo es. De las $3^3 = 27$ coloraciones posibles para un grafo de 3 vértices con 3 colores (o menos) en realidad 3 corresponden a un sólo color, 18 a dos colores y 6 con 3 colores (ver figura 1.8). Las 6 coloraciones de un sólo color son variantes de la única posible coloración de un color. Las 18 de dos colores son variantes de las 3 únicas coloraciones de dos colores y las 6 de 3 colores

son variantes de la única coloración posible de 3 colores. O sea que de las 27 coloraciones posibles en realidad sólo debemos evaluar 5.

1.1.6. Crecimiento del tiempo de ejecución

No consideremos, por ahora, la eliminación de coloraciones redundantes (si quisiéramos eliminarlas deberíamos generar un algoritmo para generar sólo las esencialmente diferentes) y consideremos que para aplicar la estrategia exhaustiva al problema de coloración de un grafo debemos evaluar $N = m^m$ coloraciones. Esto corresponde al peor caso de que el grafo necesite el número máximo de $n_c = m$ colores.

Para verificar si una coloración dada es admisible debemos realizar un cierto número de operaciones. Por ejemplo si almacenamos el grafo en forma matricial, podemos ir recorriendo las aristas del grafo y verificar que los dos vértices conectados tengan colores diferentes. Si el color es el mismo pasamos a la siguiente arista. Si recorremos todas las aristas y la coloración es admisible, entonces hemos encontrado la solución óptima al problema. En el peor de los casos el número de operaciones necesario para verificar una dada coloración es igual al número de aristas y a lo sumo el número de aristas es $m \cdot m$ (el número de elementos en la forma matricial del grafo) de manera que para verificar todas las coloraciones necesitamos verificar

$$N_{\text{be}} = m^2\, m^m = m^{m+2} \tag{1.5}$$

aristas. Asumiendo que el tiempo de verificar una arista es constante, este es el orden del número de operaciones a realizar.

El crecimiento de la función m^m con el número de vértices es tan rápido que hasta puede generar asombro. Consideremos el tiempo que tarda una computadora personal típica en evaluar todas las posibilidades para $m = 20$ vértices. Tomando un procesador de 2.4 GHz (un procesador típico al momento de escribir este apunte) y asumiendo que podemos escribir un programa tan eficiente que puede evaluar una arista por cada ciclo del procesador (en la práctica esto es imposible y al menos necesitaremos unas decenas de ciclos para evaluar una coloración) el tiempo en años necesario para evaluar todas las coloraciones es de

$$T = \frac{20^{22}}{2.4\times 10^9\,.\,3600\,.\,24\,.\,365} = 5.54\times 10^{11} \text{ años} \tag{1.6}$$

Esto es unas 40 veces la edad del universo (estimada en 15.000.000.000 de años).

Algo que debe quedar en claro es que el problema no esta en la velocidad de las computadoras, sino en la estrategia de búsqueda exhaustiva. Incluso haciendo uso de las más sofisticadas técnicas de procesamiento actuales los tiempos no bajarían lo suficiente. Por ejemplo usando uno de los *"clusters"* de procesadores más grandes existentes actualmente (con más de mil procesadores, ver http://www.top500.org) sólo podríamos bajar el tiempo de cálculo al orden de los millones de años.

Otra forma de ver el problema es preguntarse cuál es el máximo número de vértices que se puede resolver en un determinado tiempo, digamos una hora de cálculo. La respuesta es que ya con $m = 15$ se tienen tiempos de más de 5 horas.

En la sección siguiente veremos que si bien la eliminación de las coloraciones redundantes puede reducir significativamente el número de coloraciones a evaluar, el crecimiento de la función sigue siendo similar y no permite pasar de unas cuantas decenas de vértices.

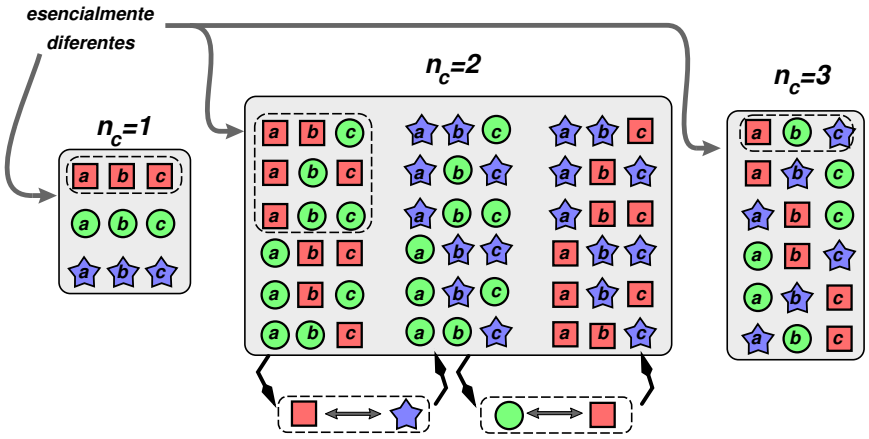


Figura 1.8: Todas las 27 coloraciones de 3 o menos colores son en realidad combinaciones de 5 esencialmente diferentes.

1.1.7. Búsqueda exhaustiva mejorada

Para reducir el número de coloraciones a evaluar podemos tratar de evaluar sólo las coloraciones esencialmente diferentes. No entraremos en el detalle de cómo generar las coloraciones esencialmente diferentes, pero sí las contaremos para evaluar si el número baja lo suficiente como para hacer viable esta estrategia.

Llamaremos entonces $N_d(n_c, m)$ al número de coloraciones esencialmente diferentes para m vértices y n_c colores. Observando la figura 1.7 vemos que el número total de coloraciones para 3 vértices con 2 o menos colores es

$$\begin{array}{ccccc} 2^3 & = & 2 & + & 6 \\ \left\{\begin{array}{c}\text{Número de}\\ \text{coloraciones}\\ \text{con } n_c = 2 \textit{ o}\\ \textit{menos}\end{array}\right\} & = & \left\{\begin{array}{c}\text{Número de}\\ \text{coloraciones}\\ \text{con}\\ \textit{exactamente}\\ n_c = 1\end{array}\right\} & + & \left\{\begin{array}{c}\text{Número de}\\ \text{coloraciones}\\ \text{con}\\ \textit{exactamente}\\ n_c = 2\end{array}\right\} \end{array} \tag{1.7}$$

A su vez el número de coloraciones con $n_c = 1$, que es 2, es igual al número de coloraciones esencialmente diferentes $N_d(1, 3) = 1$ que es, por las posibilidades de elegir un color de entre 2 que es 2. También el número de coloraciones con exactamente 2 colores es igual al número de coloraciones esencialmente diferentes $N_d(2, 3)$ que es 3, por el número de posibles maneras de elegir 2 colores de dos, que es 2 (rojo-verde, verde-rojo). En general, puede verse que la cantidad posible de elegir k colores de n_c es

$$\left\{\begin{array}{c}\text{Número de formas}\\ \text{de elegir } k \text{ colores}\\ \text{de } n_c\end{array}\right\} = \frac{n_c!}{(n_c - k)!} \tag{1.8}$$

de manera que tenemos

$$\begin{array}{rcccccccc} 2^3 & = & 2 & . & 1 & + & 2 & . & 3 \\ 2^3 & = & \dfrac{2!}{1!} & . & N_d(1,3) & + & \dfrac{2!}{0!} & . & N_d(2,3) \end{array} \tag{1.9}$$

Supongamos que queremos calcular las coloraciones esencialmente diferentes para $m = 5$ colores, entonces plantemos las relaciones

$$\begin{array}{rcccccccccc} 1^5 & = & \dfrac{1!}{0!}N_d(1,5) & & & & & & & & \\ 2^5 & = & \dfrac{2!}{1!}N_d(1,5) & + & \dfrac{2!}{0!}N_d(2,5) & & & & & & \\ 3^5 & = & \dfrac{3!}{2!}N_d(1,5) & + & \dfrac{3!}{1!}N_d(2,5) & + & \dfrac{3!}{0!}N_d(3,5) & & & & \\ 4^5 & = & \dfrac{4!}{3!}N_d(1,5) & + & \dfrac{4!}{2!}N_d(2,5) & + & \dfrac{4!}{1!}N_d(3,5) & + & \dfrac{4!}{0!}N_d(4,5) \end{array} \tag{1.10}$$

o sea

$$\begin{array}{rcl} 1 & = & N_d(1,v) \\ 32 & = & 2\,N_d(1,5) + 2\,N_d(2,5) \\ 243 & = & 3\,N_d(1,5) + 6\,N_d(2,5) + 6\,N_d(3,5) \\ 1024 & = & 4\,N_d(1,5) + 12\,N_d(2,5) + 24\,N_d(3,5) + 24\,N_d(4,5) \end{array} \tag{1.11}$$

Notemos que de la segunda ecuación puede despejarse fácilmente $N_d(2,5)$ que resulta ser 15. De la tercera se puede despejar $N_d(3,5)$ ya que conocemos $N_d(1,5)$ y $N_d(2,5)$ y resulta ser $N_d(3,5) = 25$ y así siguiendo resulta ser

$$\begin{array}{l} N_d(1,5) = 1 \\ N_d(2,5) = 15 \\ N_d(3,5) = 25 \\ N_d(4,5) = 10 \\ N_d(5,5) = 1 \end{array} \tag{1.12}$$

de manera que el número total de coloraciones esencialmente diferentes es

$$\begin{array}{c} N_d(1,5) + N_d(2,5) + N_d(3,5) + N_d(4,5) + N_d(5,5) = \\ 1 + 15 + 25 + 10 + 1 = 52 \end{array} \tag{1.13}$$

Es muy fácil escribir un programa (en C++, por ejemplo) para encontrar el número total de coloraciones esencialmente diferentes para un dado número de vértices, obteniéndose una tabla como la 1.1

m	coloraciones	coloraciones diferentes
1	1	1
2	4	2
3	27	5
4	256	15
5	3125	52
6	46656	203
7	823543	877

Tabla 1.1: Número de coloraciones para un grafo con m vértices. Se indica el número de coloraciones total y también el de áquellas que son esencialmente diferentes entre sí.

A primera vista se observa que eliminando las coloraciones redundantes se obtiene una gran reducción en el número de coloraciones a evaluar. Tomando una serie de valores crecientes de m y calculando el número de coloraciones diferentes como en la tabla 1.1 se puede ver que éste crece como

$$N_d(m) = \sum_{n_c=1}^{m} N_d(n_c, m) \approx m^{m/2} \tag{1.14}$$

El número de aristas a verificar, contando m^2 aristas por coloración es de

$$N_{\text{bem}} \approx m^{m/2}\, m^2 = m^{m/2+2} \tag{1.15}$$

Sin embargo, si bien esto significa una gran mejora con respecto a m^m, el tiempo para colorear un grafo de 20 vértices se reduce del tiempo calculado en (1.6) a sólo 99 días. Está claro que todavía resulta ser excesivo para un uso práctico.

Una implementación en C++ del algoritmo de búsqueda exhaustiva puede encontrarse en el código que se distribuye con este apunte en **aedsrc/colgraf.cpp**. La coloración óptima del grafo se encuentra después de hacer 1.429.561 evaluaciones en 0.4 secs. Notar que el número de evaluaciones baja notablemente con respecto a la estimación $m^{m+2} \approx 9 \times 10^{12}$ ya que se encuentra una coloración admisible para $n_c = 4$ con lo cual no es necesario llegar hasta $n_c = m$. De todas formas incluso si tomáramos como cota inferior para evaluar los tiempos de ejecución el caso en que debierámos evaluar al menos todas las coloraciones de 2 colores, entonces tendríamos al menos un tiempo de ejecución que crece como 2^m evaluaciones. Incluso con este *"piso"* para el número de evaluaciones, el tiempo de cálculo sería de una hora para $m = 33$.

1.1.8. Algoritmo heurístico ávido

Una estrategia diferente a la de búsqueda exhaustiva es la de buscar una solución que, si bien no es la óptima (es decir, la mejor de todas), sea aceptablemente buena, y se pueda obtener en un tiempo razonable. Si se quiere, ésta es una estrategia que uno utiliza todo el tiempo: si tenemos que comprar una licuadora y nos proponemos comprar la más barata, no recorremos absolutamente todos los bazares y supermercados de todo el planeta y revisamos todos las marcas posibles, sino que, dentro del tiempo que aceptamos dedicar a esta búsqueda, verificamos el costo de los artículos dentro de algunos comercios y marcas que consideramos los más representativos.

Un algoritmo que produce una solución razonablemente buena haciendo hipótesis *"razonables"* se llama *"heurístico"*. Del diccionario: *heurístico: una regla o conjunto de reglas para incrementar la posibilidad de resolver un dado problema*.

Un posible algoritmo heurístico para colorear grafos es el siguiente algoritmo *"ávido"*. Primero tratamos de colorear tantos vértices como podamos con el primer color, luego con el segundo color y así siguiendo hasta colorearlos todos. La operación de colorear con un dado color puede resumirse como sigue

- Seleccionar algún vértice no coloreado y asignarle el nuevo color.
- Recorrer la lista de vértices no colorados. Para cada vértice no coloreado determinar si está conectado (esto es, posee algún vértice en común) con un vértice del nuevo color.

Esta aproximación es llamada ávida ya que asigna colores tan rápido como lo puede hacer, sin tener en cuenta las posibles consecuencias negativas de tal acción. Si estuviéramos escribiendo un programa para jugar al ajedrez, entonces una estrategia ávida, sería evaluar todas las posibles jugadas y elegir la que da la mejor ventaja material. En realidad no se puede catalogar a los algoritmos como ávidos en forma absoluta, sino que se debe hacer en forma comparativa: hay algoritmos más ávidos que otros. En general cuanto más ávido es un algoritmo más simple es y más rápido es en cuanto a avanzar para resolver el problema, pero por otra parte explora en menor medida el espacio de búsqueda y por lo tanto puede dar una solución peor que otro menos ávida. Volviendo al ejemplo del ajedrez, un programa que, además de evaluar la ganancia material de la jugada a realizar, evalúe las posibles consecuencias de la siguiente jugada del oponente requerirá mayor tiempo pero a largo plazo producirá mejores resultados.

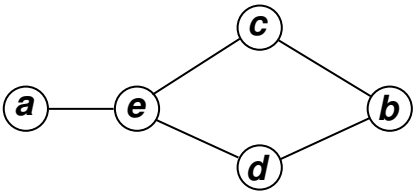


Figura 1.9: Ejemplo de un grafo simple para aplicar el algoritmo ávido.

Si consideramos la coloración de un grafo como el mostrado en la figura 1.9 entonces empezando por el color rojo le asignaríamos el rojo al vértice

a. Posteriormente recorreríamos los vértices no coloreados, que a esta altura son $\{b, c, d, e\}$ y les asignamos el color rojo si esto es posible. Podemos asignárselo a b pero no a c y d, ya que están conectados a b ni tampoco a e ya que está conectado a a. Pasamos al siguiente color, digamos verde. La lista de vértices no coloreados es, a esta altura $\{c, d, e\}$. Le asignamos verde a c y luego también a d, pero no podemos asignárselo a e ya que está conectado a c y d. El proceso finaliza asignándole el siguiente color (digamos azul) al último vértice sin colorear e. El grafo coloreado con esta estrategia se muestra en la figura 1.10.

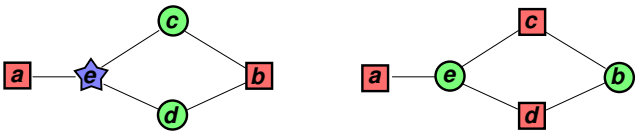


Figura 1.10: Izquierda: El grafo de la figura 1.9 coloreado con la estrategia ávida. Derecha: Coloración óptima.

El algoritmo encuentra una solución con tres colores, sin embargo se puede encontrar una solución con dos colores, como puede verse en la misma figura. Esta última es óptima ya que una mejor debería tener sólo un color, pero esto es imposible ya que entonces no podría haber ninguna arista en el grafo. Este ejemplo ilustra perfectamente que si bien el algoritmo ávido da una solución razonable, ésta puede no ser la óptima.

Notemos también que la coloración producida por el algoritmo ávido depende del orden en el que se recorren los vértices. En el caso previo, si recorriéramos los nodos en el orden $\{a, e, c, d, b\}$, obtendríamos la coloración óptima.

En el caso del grafo original mostrado en la figura 1.2, el grafo coloreado resultante con este algoritmo resulta tener 4 colores y se muestra en la figura 1.11.

Notemos que cada vez que se agrega un nuevo color, por lo menos a un nodo se le asignará ese color, de manera que el algoritmo usa a lo sumo m colores. Esto demuestra también que el algoritmo en su totalidad termina en un número finito de pasos.

Una implementación en C++ del algoritmo ávido puede encontrarse en el código que se distribuye con este apunte en **aedsrc/colgraf.cpp**.

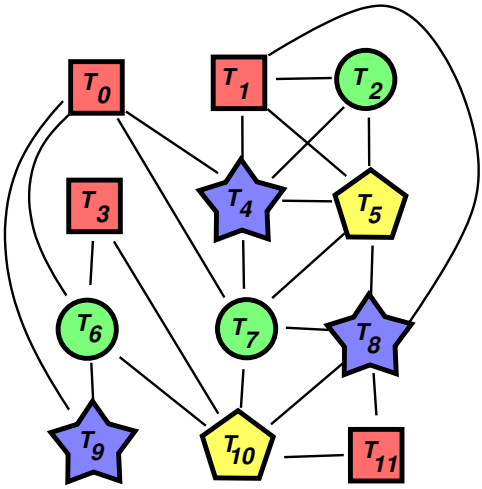


Figura 1.11: Grafo coloreado.

1.1.9. Descripción del algoritmo heurístico en seudo-código

Una vez que tenemos una versión abstracta (matemática) del modelo o algoritmo podemos empezar a implementarlo para llegar a un programa real que resuelve el problema. Este proceso puede llevarse a cabo en varias etapas empezando por una descripción muy general en forma de sentencias vagas, llamado *"seudo-código"*, como *"elegir un vértice no coloreado"*. A veces es común incluir este seudo-código en forma de comentarios seguidos por puntos suspensivos que indican que falta completar esa parte del programa.

Lo ideal es que estas sentencias sean suficientemente claras como para no dejar dudas de cual es la tarea a realizar, pero también lo suficientemente generales como para no tener que entrar en detalles y poder diseñar rápidamente una versión básica del código. Luego en un paso de *"refinamiento"* posterior estas sentencias en seudo-código son refinadas en tareas más pequeñas, las cuales pueden ser descriptas parte en líneas de seudo-código ellas mismas y parte en sentencias válidas del lenguaje, hasta que finalmente terminamos con un código que puede ser compilado y linkeditado en un programa.

Tomemos como ejemplo el algoritmo heurístico descripto previamente en §1.1.8. La rutina `greedyc` mostrada en el código 1.1 (archivo `greedy.cpp`) toma como argumentos un grafo `G`, el conjunto de vértices no coloreados

hasta el momento **no_col** y determina un conjunto de nodos **nuevo_color** los cuales pueden ser coloreados con el nuevo color. Los vértices son identificados por un entero de 0 a **nv-1**. La rutina además mantiene una tabla **tabla_color** donde finalmente quedarán los colores de cada vértice. El argumento de entrada **color** indica el color (un entero empezando desde 0 e incrementándose de a 1 en cada llamada a **greedyc**) con el cual se está coloreando en esta llamada a **greedyc**. Notar que esta rutina será llamada posteriormente dentro de un lazo sobre los colores hasta colorear todos los vértices. Los conjuntos son representados con el template **set<...>** de la librerías STL (por *"Standard Template Library"*, hoy parte del C++ estándar). Falta todavía implementar la clase **graph**. El lazo de las líneas 9-17 es un lazo típico para recorrer un **set<>**.

```
1.  void greedyc(graph &G, set<int> &no_col,
2.           set<int> &nuevo_color,
3.           vector<int> &tabla_color,int color) {
4.    // Asigna a 'nuevo_color' un conjunto de vertices
5.    // de 'G' a los cuales puede darse el mismo nuevo color
6.    // sin entrar en conflicto con los ya coloreados
7.    nuevo_color.clear();
8.    set<int>::iterator q;
9.    for (q=no_col.begin(); q!=no_col.end(); q++) {
10.     if (/* '*q' no es adyacente a
11.            ningun vertice en 'nuevo_color' . . . */) {
12.       // marcar a '*q' como coloreado
13.       // . . .
14.       // agregar '*q' a 'nuevo_color'
15.       // . . .
16.     }
17.   }
18. }
```

Código 1.1: *Rutina para la coloración de un grafo. Determina el conjunto de vértices que pueden ser coloreados con un nuevo color sin entrar en conflicto. Versión inicial. [Archivo: greedy.cpp]*

Haciendo un breve repaso de los *iterators* de STL, el iterator **q**, declarado en la línea 8 actúa como un puntero a **int** , de hecho ***q** es una referencia a un **int**. Al empezar el lazo apunta al primer elemento del **no_col** y en cada iteración del lazo pasa a otro elemento, hasta que cuando se han acabado

todos toma el valor `no_col.end()`, con lo cual finaliza al lazo. Dentro del lazo faltan implementar 3 porciones de código. La condición del `if` de las líneas 10-11, el código para marcar al vértice como coloreado en la línea 13 y para agregarlo a `nuevo_color` en la línea 15.

Vamos ahora a refinar el algoritmo anterior, expandiendo ahora más aún la expresión condicional del `if`. Para verificar si `*q` es adyacente a algún vértice de `nuevo_color` debemos recorrer todos los nodos de `nuevo_color` y verificar si hay alguna arista entre los mismos y `*q`. Para esto hacemos un lazo, definiendo una variable `adyacente` (ver código 1.2, archivo `greedy2.cpp`). Al llegar al comienzo del condicional en la línea 10 la variable `adyacente` tiene el valor apropiado. Notar que si se detecta que uno de los vértices de `nuevo_color` es adyacente a `*q` , entonces no es necesario seguir con el lazo, por eso el `break` de la línea 15. Además hemos implementado las líneas 13 y líneas 15 del código 1.1, resultando en las líneas líneas 20 y líneas 22 del código 1.2. La línea 20 simplemente registra el nuevo color asignado a la tabla `tabla_color` y la línea 22 inserta el vértice que se termina de colorear `*q` al conjunto `nuevo_color`. Notar que deberíamos eliminar los vértices que coloreamos de `no_col` pero esto no lo podemos hacer dentro del lazo de las líneas 9-24 del código 1.2, ya que dentro del mismo se itera sobre `no_col`. Modificar `no_col` convertiría en inválido el iterador `q` y por lo tanto después no se podría aplicar el operador `++` a `q` en la línea 9.

```
1.  void greedyc(graph &G, set<int> &no_col,
2.           set<int> &nuevo_color,
3.           vector<int> &tabla_color,int color) {
4.    // Asigna a 'nuevo_color' un conjunto de vertices
5.    // de 'G' a los cuales puede darse el mismo nuevo color
6.    // sin entrar en conflicto con los ya coloreados
7.    nuevo_color.clear();
8.    set<int>::iterator q,w;
9.    for (q=no_col.begin(); q!=no_col.end(); q++) {
10.     int adyacente=0;
11.     for (w=nuevo_color.begin();
12.         w!=nuevo_color.end(); w++) {
13.       if (/* '*w' es adyacente a '*q' . . . */) {
14.         adyacente = 1;
15.         break;
16.       }
17.     }
18.     if (!adyacente) {
19.       // marcar a '*q' como coloreado
20.       tabla_color[*q] = color;
```

```
21.       // agregar '*q' a 'nuevo_color'
22.       nuevo_color.insert(*q);
23.     }
24.   }
25. }
```

Código 1.2: *Rutina para la coloración de un grafo. Versión refinada. [Archivo: greedy2.cpp]*

Para refinar el condicional de la línea 13 necesitamos definir la clase **grafo**, para ello utilizaremos una representación muy simple, útil para grafos pequeños basada en mantener una tabla de unos y ceros como fue descripto en §1.1.2. Como el grafo no es orientado, la matriz es simétrica ($A_{jk} = A_{kj}$) de manera que sólo usaremos la parte triangular inferior de la misma. La matriz será almacenada por filas en un arreglo **vector<int> g** de manera que el elemento A_{jk} estará en la posición $g[m * j + k]$. La clase grafo se puede observar en el código 1.3. Los elementos de la matriz se acceden a través de una función miembro **edge(j,k)** que retorna una referencia al elemento correspondiente de la matriz. Notar que si $j < k$, entonces se retorna en realidad el elemento simétrico A_{kj} en la parte triangular inferior. Como **edge()** retorna una referencia al elemento correspondiente, puede usarse tanto para insertar aristas en el grafo

```
G.edge(j,k) = 1;
```

como para consultar si un dado par de vértices es adyacente o no

```
if (!G.edge(j,k)) {
  // no estan conectados
  // ...
}
```

```
1. class graph {
2. private:
3.   const int nv;
4.   vector<int> g;
5. public:
6.   // Constructor a partir del numero de vertices
7.   graph(int nv_a) : nv(nv_a) { g.resize(nv*nv,0); }
8.   // Este metodo permite acceder a una arista tanto para
```

```
9.   // agregar la arista ('g.edge(i,j)=1') como para
10.  // consultar un valor particular de la
11.  // arista. ('adyacente = g.edge(i,j)')
12.  int &edge(int j,int k) {
13.    if (k<=j) return g[nv*j+k];
14.    else return g[nv*k+j];
15.  }
16. };
```

Código 1.3: *Clase básica de grafo. [Archivo: graph.cpp]*

La versión final de la rutina **greedyc** puede observarse en el código 1.4.

```
1.  void greedyc(graph &G, set<int> &no_col,
2.           set<int> &nuevo_color,
3.           vector<int> &tabla_color,int color) {
4.   // Asigna a 'nuevo_color' un conjunto de vertices
5.   // de 'G' a los cuales puede darse el mismo nuevo color
6.   // sin entrar en conflicto con los ya coloreados
7.   nuevo_color.clear();
8.   set<int>::iterator q,w;
9.   for (q=no_col.begin(); q!=no_col.end(); q++) {
10.    int adyacente=0;
11.    for (w=nuevo_color.begin();
12.        w!=nuevo_color.end(); w++) {
13.      if (G.edge(*q,*w)) {
14.        adyacente = 1;
15.        break;
16.      }
17.    }
18.    if (!adyacente) {
19.      // marcar a '*q' como coloreado
20.      tabla_color[*q] = color;
21.      // agregar '*q' a 'nuevo_color'
22.      nuevo_color.insert(*q);
23.    }
24.  }
25. }
```

Código 1.4: *Versión final de la rutina [Archivo: greedy3.cpp]*

Ahora falta definir el código exterior que iterará los colores, llamando a **greedyc**. Un primer esbozo puede observarse en el código 1.5. La rutina

greedy toma como argumentos de entrada el grafo a colorear **G**, el número de vértices **nv** y devuelve la coloración en **tabla_color**. Internamente inicializa el conjunto de vértices no coloreados insertando todos los vértices del grafo en la línea 8. A continuación entra en un lazo infinito, del cual sólo saldrá cuando todos los vértices estén coloreados, y por lo tanto **no_col** sea vacío, lo cual todavía debemos implementar en la línea 19. (Notemos que es válido utilizar un lazo infinito ya que hemos garantizado que el algoritmo se ejecuta a lo sumo un número finito de veces, más precisamente a lo sumo m veces.) Dentro del lazo, se determina el conjunto de vértices al cual se asignará el nuevo color llamando a **greedyc(...)** (línea 13). Luego debemos sacar los vértices asignados al nuevo color de **no_col** y, después de verificar la condición de fin del algoritmo, incrementar el número de color.

```
1.  void greedy(graph &G, int nv,
2.          vector<int> &tabla_color) {
3.    int color=0;
4.    set<int> nuevo_color, no_col;
5.    set<int>::iterator q;
6.    // Inicialmente ponemos todos los vertices en
7.    // 'no_col'
8.    for (int k=0; k<nv; k++) no_col.insert(k);
9.    while (1) {
10.     // Determina a cuales vertices podemos asignar el
11.     // nuevo color
12.     greedyc(G,no_col,nuevo_color,
13.           tabla_color,color);
14.     // Saca los vertices que se acaban de colorear
15.     // ('nuevo_color') de 'no_col'
16.     // . . .
17.     // Detecta el fin del algoritmo cuando ya no hay
18.     // mas vertices para colorear.
19.     // . . .
20.     color++;
21.   }
22. }
```

Código 1.5: *Algoritmo de coloración. Se van agregando nuevos colores llamando a* greedyc *[Archivo: greedy4.cpp]*

En el código 1.6 vemos la versión refinada definitiva. La eliminación de los elementos de **nuevo_color** de **no_col** se realiza recorriéndolos y usando

la función **erase** de **set<>**. La detección de si **no_col** esta vacío o no se realiza usando la función **size()**. Esta retorna el número de elementos en el conjunto, de manera que si retorna cero, entonces el conjunto esta vacío.

```
1.  void greedy(graph &G, int nv,
2.          vector<int> &tabla_color) {
3.    int color=0;
4.    set<int> nuevo_color, no_col;
5.    set<int>::iterator q;
6.    // Inicialmente ponemos todos los vertices en 'no_col'
7.    for (int k=0; k<nv; k++) no_col.insert(k);
8.    while (1) {
9.      // Determina a cuales vertices podemos asignar
10.     // el nuevo color
11.     greedyc(G,no_col,nuevo_color,tabla_color,color);
12.     // Saca los vertices que se acaban de colorear
13.     // ('nuevo_color') de 'no_col'
14.     for (q=nuevo_color.begin();
15.        q!=nuevo_color.end(); q++)
16.       no_col.erase(*q);
17.     // Detecta el fin del algoritmo cuando ya no hay
18.     // mas vertices para colorear.
19.     if (!no_col.size()) return;
20.     color++;
21.   }
22. }
```

Código 1.6: *Algoritmo de coloración. Versión final. [Archivo: greedy5.cpp]*

El código **greedyf.cpp** contiene una versión completa del código. En el programa principal se definen dos grafos pequeños (correspondientes a las figuras 1.9 y 1.2) para probar el algoritmo y también incluye la posibilidad de generar grafos aleatorios con un número de aristas prefijado. Como para darse una idea de las posibilidades prácticas de este algoritmo, es capaz de colorear un grafo aleatorio de 5000 vértices y 6.25×10^6 aristas (la mitad del total posible $m(m-1)/2$ en 7 segs, en un procesador de características similares a las mencionadas en §1.1.6.

1.1.10. Crecimiento del tiempo de ejecución para el algoritmo ávido

Si bien el cálculo de tiempos de ejecución será desarrollado más adelante, en la sección §1.3, podemos rápidamente tener una idea de como se comporta en función del número de vértices. Consideremos el número de operaciones que realiza la rutina **greedyc** (ver código 1.4). El lazo de las líneas 9-17 se ejecuta a los sumo m veces, ya que **no_col** puede tener a lo sumo m elementos. Dentro del lazo hay una serie de llamados a funciones (los métodos **begin()**, **end()** e **insert()** de la clase **set<>** y el operador incremento de la clase **set<>::iterator**). Por ahora no sabemos como están implementadas estas operaciones, pero de la documentación de las STL se puede deducir que estas operaciones se hacen en tiempo constante, es decir que no crecen con m (En realidad no es así. Crecen como $\log m$, pero por simplicidad asumiremos tiempo constante, y las conclusiones serían básicamente las mismas si incluyéramos este crecimiento). Por lo tanto, fuera del lazo de las líneas 12-17, todas las otras instrucciones consumen un número de operaciones constante. El lazo de las líneas 12-17 se ejecuta a lo sumo m veces y dentro de el sólo se realizan un número constante de operaciones (la llamada a **G.edge(...)** en nuestra implementación es simplemente un acceso a un miembro de **vector<...>** el cual, también de acuerdo a la documentación de STL se hace en tiempo constante). De manera que el tiempo de ejecución de **greedyc** es a lo sumo m^2 operaciones. Por otra parte, en el código 1.6 tenemos el lazo de las líneas 8-21 el cual se ejecuta un número máximo de m veces. En cada ejecución del lazo, la llamada a **greedyc** consume a lo sumo m^2 operaciones. En el resto del bloque tenemos todas líneas que consumen un número constante de operaciones, menos el lazo de las líneas 15-16, el cual se ejecuta a lo sumo m veces y consume en cada iteración a lo sumo un número constante de operaciones. De manera que todo el bloque de las líneas 8-21 consume a lo sumo m^3 operaciones. Lo cual significa una dramática reducción con respecto a las estimaciones para los algoritmos de búsqueda exhaustiva (1.5) y búsqueda exhaustiva mejorada (1.15).

1.1.11. Conclusión del ejemplo

En toda esta sección §1.1.1 hemos visto un ejemplo en el cual resolvemos un problema planteando un modelo matemático abstracto (en este caso las estructuras *grafo* y *conjunto*. Inicialmente el algoritmo es expresado infor-

malmente en términos de operaciones abstractas sobre estas estructuras. Posteriormente se genera un primer esbozo del algoritmo con una mezcla de sentencias escritas en C++ (u otro lenguaje) y seudo-código, el cual es refinado en una serie de etapas hasta llegar a un programa que se puede compilar, linkeditar y ejecutar.

1.2. Tipos abstractos de datos

Una vez que se ha elegido el algoritmo, la implementación puede hacerse usando las estructuras más simples, comunes en casi todos los lenguajes de programación: escalares, arreglos y matrices. Sin embargo algunos problemas se pueden plantear en forma más simple o eficiente en términos de estructuras informáticas más complejas, como listas, pilas, colas, árboles, grafos, conjuntos. Por ejemplo, el TSP se plantea naturalmente en términos de un grafo donde los vértices son las ciudades y las aristas los caminos que van de una ciudad a otra. Estas estructuras están incorporadas en muchos lenguajes de programación o bien pueden obtenerse de librerías. El uso de estas estructuras tiene una serie de ventajas

- Se ahorra tiempo de programación ya que no es necesario codificar.
- Estas implementaciones suelen ser eficientes y robustas.
- Se separan dos capas de código bien diferentes, por una parte el algoritmo que escribe el programador, y por otro las rutinas de acceso a las diferentes estructuras.
- Existen estimaciones bastante uniformes de los tiempos de ejecución de las diferentes operaciones.
- Las funciones asociadas a cada estructura son relativamente independientes del lenguaje o la implementación en particular. Así, una vez que se plantea un algoritmo en términos de operaciones sobre una tal estructura es fácil implementarlo en una variedad de lenguajes con una performance similar.

Un *"Tipo Abstracto de Datos"* (TAD) es la descripción matemática de un objeto abstracto, definido por las operaciones que actúan sobre el mismo. Cuando usamos una estructura compleja como un conjunto, lista o pila podemos separar tres niveles de abstracción diferente, ejemplificados en la

figura 1.12, a saber las *"operaciones abstractas"* sobre el TAD, la *"interfaz"* concreta de una implementación y finalmente la *"implementación"* de esa interfaz.

Tomemos por ejemplo el TAD CONJUNTO utilizado en el ejemplo de la sección §1.1.1 . Las siguientes son las operaciones abstractas que podemos querer realizar sobre un conjunto

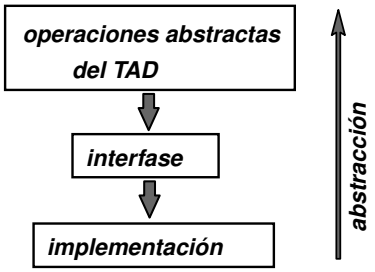


Figura 1.12: Descripción de lo diferentes niveles de abstracción en la definición de un TAD

1.2.1. Operaciones abstractas y características del TAD CONJUNTO

- Contiene elementos, los cuales deben ser diferentes entre sí.
- No existe un orden particular entre los elementos del conjunto.
- Se pueden insertar o eliminar elementos del mismo.
- Dado un elemento se puede preguntar si está dentro del conjunto o no.
- Se pueden hacer las operaciones binarias bien conocidas entre conjuntos a saber, unión, intersección y diferencia.

1.2.2. Interfaz del TAD CONJUNTO

La *"interfaz"* es el conjunto de operaciones (con una sintaxis definida) que producen las operaciones del TAD. Por supuesto depende del lenguaje a utilizar, si bien algunas veces es también común que una librería pueda ser usada desde diferentes lenguajes y trate de mantener la interfaz entre esos diferentes lenguajes.

Por ejemplo la implementación del TAD CONJUNTO en la librería STL es (en forma muy simplificada) la siguiente,

```
1.  template<class T>
2.  class set {
3.  public:
4.    class iterator { /* . . . */ };
5.    void insert(T x);
6.    void erase(iterator p);
7.    void erase(T x);
8.    iterator find(T x);
9.    iterator begin();
10.   iterator end();
11. };
```

Código 1.7: *Interfaz de la clase set<> [Archivo: stl-set.cpp]*

Esta interfaz tiene la desventaja de que no provee directamente las operaciones mencionadas previamente pero encaja perfectamente dentro de la interfaz general de los otros *"contenedores"* de STL. Recordemos que en STL los elementos de los contenedores, como en este caso **set**, se acceden a través de *"iteradores"* (*"iterators"*). En la siguiente descripción **s** es un conjunto, **x** un elemento y **p** un iterador

- **s.insert(x)** inserta un elemento en el conjunto. Si el elemento ya estaba en el conjunto **s** queda inalterado.
- **p=s.find(x)** devuelve el iterador para el elemento **x**. Si **x** no está en **s** entonces devuelve un iterador especial **end()**. En consecuencia, la expresión lógica para saber si un elemento está en **s** es

  ```
  if(s.find(x)==s.end()) {
    // `x' no esta en `s'
    // ...
  }
  ```

- **s.erase(p)** elimina el elemento que está en el iterador **p** en **s**. **s.erase(x)** elimina el elemento **x** (si está en el conjunto).
- La unión de dos conjuntos, por ejemplo $C = A \cup B$ podemos lograrla insertando los elementos de A y B en C:

```
set A,B,C;
// Pone elementos en A y B
// ...
C.insert(A.begin(),A.end());
C.insert(B.begin(),B.end());
```

Normalmente en C/C++ la interfaz está definida en los headers de las respectivas clases.

- Todas las operaciones binarias con conjuntos se pueden realizar con algoritmos genéricos definidos en el header **algorithm**, usando el adaptador **inserter**:

```
template<class Container, class Iter)
insert_iterator<Container>
inserter(Container& C, Iter i);
```

Típicamente:

- $C = A \cup B$

```
set_union(a.begin(),a.end(),b.begin(),b.end(),
          inserter(c,c.begin()));
```

- $C = A - B$

```
set_difference(a.begin(),a.end(),b.begin(),b.end(),
               inserter(c,c.begin()));
```

- $C = A \cap B$

```
set_intersection(a.begin(),a.end(),b.begin(),b.end(),
                 inserter(c,c.begin()));
```

1.2.3. Implementación del TAD CONJUNTO

Finalmente la *"implementación"* de estas funciones, es decir el código específico que implementa cada una de las funciones declaradas en la interfaz.

Como regla general podemos decir que un programador que quiere usar una interfaz abstracta como el TAD CONJUNTO, debería tratar de elaborar

primero un algoritmo abstracto basándose en las operaciones abstractas sobre el mismo. Luego, al momento de escribir su código debe usar la interfaz específica para traducir su algoritmo abstracto en un código compilable. En el caso del TAD CONJUNTO veremos más adelante que internamente éste puede estar implementado de varias formas, a saber con listas o árboles, por ejemplo. En general, el código que escribe no debería depender *nunca* de los detalles de la implementación particular que esta usando.

1.3. Tiempo de ejecución de un programa

La eficiencia de un código va en forma inversa con la cantidad de recursos que consume, principalmente tiempo de CPU y memoria. A veces en programación la eficiencia se contrapone con la sencillez y legibilidad de un código. Sin embargo en ciertas aplicaciones la eficiencia es un factor importante que no podemos dejar de tener en cuenta. Por ejemplo, si escribimos un programa para buscar un nombre en una agenda personal de 200 registros, entonces probablemente la eficiencia no es la mayor preocupación. Pero si escribimos un algoritmo para un motor de búsqueda en un número de entradas $> 10^9$, como es común en las aplicaciones para buscadores en Internet hoy en día, entonces la eficiencia probablemente pase a ser un concepto fundamental. Para tal volumen de datos, pasar de un algoritmo $O(n \log n)$ a uno $O(n^{1.3})$ puede ser fatal.

Más importante que saber escribir programas eficientemente es saber *cuándo* y *donde* preocuparse por la eficiencia. Antes que nada, un programa está compuesto en general por varios componentes o módulos. No tiene sentido preocuparse por la eficiencia de un dado módulo si este representa un 5 % del tiempo total de cálculo. En un tal módulo tal vez sea mejor preocuparse por la robustez y sencillez de programación que por la eficiencia.

El tiempo de ejecución de un programa (para fijar ideas, pensemos por ejemplo en un programa que ordena de menor a mayor una serie de números enteros) depende de una variedad de factores, entre los cuales

- *La eficiencia del compilador y las opciones de optimización que pasamos al mismo en tiempo de compilación.*
- *El tipo de instrucciones y la velocidad del procesador donde se ejecutan las instrucciones compiladas.*
- *Los datos del programa.* En el ejemplo, la cantidad de números y su distribución estadística: ¿son todos iguales?, ¿están ya ordenados o casi ordenados?

- *La "complejidad algorítmica" del algoritmo subyacente*. En el ejemplo de ordenamiento, el lector ya sabrá que hay algoritmos para los cuales el número de instrucciones crece como n^2, donde n es la longitud de la lista a ordenar, mientras que algoritmos como el de *"ordenamiento rápido"* (*"quicksort"*) crece como $n \log n$. En el problema de coloración estudiado en §1.1.1 el algoritmo de búsqueda exhaustiva crece como m^m, donde m es el número de vértices del grafo contra m^3 para el algoritmo ávido descripto en §1.1.8.

En este libro nos concentraremos en los dos últimos puntos de esta lista.

```
1. int search(int l,int *a,int n) {
2.   int j;
3.   for (j=0; j<n; j++)
4.     if (a[j]==l) break;
5.   return j;
6. }
```

Código 1.8: *Rutina simple para buscar un elemento* `l` *en un arreglo* `a[]` *de longitud* n *[Archivo: search.cpp]*

En muchos casos, el tiempo de ejecución depende no tanto del conjunto de datos específicos, sino de alguna *"medida"* del tamaño de los datos. Por ejemplo sumar un arreglo de n números no depende de los números en sí mismos sino de la longitud n del arreglo. Denotando por $T(n)$ el tiempo de ejecución

$$T(n) = cn \tag{1.16}$$

donde c es una constante que representa el tiempo necesario para sumar un elemento. En otros muchos casos, si bien el tiempo de ejecución sí depende de los datos específicos, *en promedio* sólo depende del tamaño de los datos. Por ejemplo, si buscamos la ubicación de un elemento l en un arreglo a, simplemente recorriendo el arreglo desde el comienzo hasta el fin (ver código 1.8) hasta encontrar el elemento, entonces el tiempo de ejecución dependerá fuertemente de la ubicación del elemento dentro del arreglo. El tiempo de ejecución es proporcional a la posición j del elemento dentro del vector tal que $a_j = l$, tomando $j = n$ si el elemento no está. El mejor caso es cuando el elemento está al principio del arreglo, mientras que el peor es cuando está al final o cuando no está. Pero *"en promedio"* (asumiendo que la

distribución de elementos es aleatoria) el elemento buscado estará en la zona media del arreglo, de manera que una ecuación como la (1.16) será válida (en promedio). Cuando sea necesario llamaremos $T_{\text{prom}}(n)$ al promedio de los tiempos de ejecución de un dado algoritmo sobre un *"ensamble"* de posibles entradas y por $T_{\text{peor}}(n)$ el peor de todos sobre el ensamble. Entonces, para el caso de buscar la numeración de un arreglo tenemos

$$\begin{aligned} T(n) &= cj \\ T_{\text{peor}}(n) &= cn \\ T_{\text{prom}}(n) &= c\frac{n}{2} \end{aligned} \tag{1.17}$$

En estas expresiones c puede tomarse como el tiempo necesario para ejecutar una vez el cuerpo del lazo en la rutina `search(...)`. Notar que esta constante c, si la medimos en segundos, puede depender fuertemente de los ítems considerados en los dos primeros puntos de la lista anterior. Por eso, preferimos dejar la constante sin especificar en forma absoluta, es decir que de alguna forma estamos evaluando el tiempo de ejecución en términos de *"unidades de trabajo"*, donde una unidad de trabajo c es el tiempo necesario para ejecutar una vez el lazo.

En general, determinar analíticamente el tiempo de ejecución de un algoritmo puede ser una tarea intelectual ardua. Muchas veces, encontrar el $T_{\text{peor}}(n)$ es una tarea relativamente más fácil. Determinar el $T_{\text{prom}}(n)$ puede a veces ser más fácil y otras veces más difícil.

1.3.1. Notación asintótica

Para poder obtener una rápida comparación entre diferentes algoritmos usaremos la *"notación asintótica"* $O(...)$. Por ejemplo, decimos que el tiempo de ejecución de un programa es $T(n) = O(n^2)$ (se lee *"$T(n)$ es orden n^2"*) si existen constantes $c, n_0 > 0$ tales que para

$$T(n) \leq c\, n^2, \quad \text{para} \quad n \geq n_0 \tag{1.18}$$

La idea es que no nos interesa como se comporta la función $T(n)$ para valores de n pequeños sino sólo la tendencia para $n \to \infty$.

Ejemplo 1.1: Sea $T(n) = (n+1)^2$, entonces si graficamos $T(n)$ en función de n (ver figura 1.13) junto con la función $2n^2$ vemos que la relación $T(n) \leq 2n^2$ es cierta para valores de $3 \leq n \leq 10$. Para ver que esta

relación es válida para *todos* los valores de n tales que $n \geq 3$, entonces debemos recurrir a un poco de álgebra. Tenemos que, como

$$n \geq 3, \tag{1.19}$$

entonces

$$\begin{aligned} n - 1 &\geq 2, \\ (n-1)^2 &\geq 4, \\ n^2 - 2n + 1 &\geq 4, \\ n^2 &\geq 3 + 2n, \\ 3 + 2n + n^2 &\leq 2n^2. \end{aligned} \tag{1.20}$$

Pero

$$3 + 2n + n^2 = (n+1)^2 + 2, \tag{1.21}$$

y por lo tanto

$$(n+1)^2 \leq (n+1)^2 + 2 \leq 2n^2, \tag{1.22}$$

que es la relación buscada. Por lo tanto $T(n) = O(n^2)$ con $c = 2$ y $n_0 = 3$.

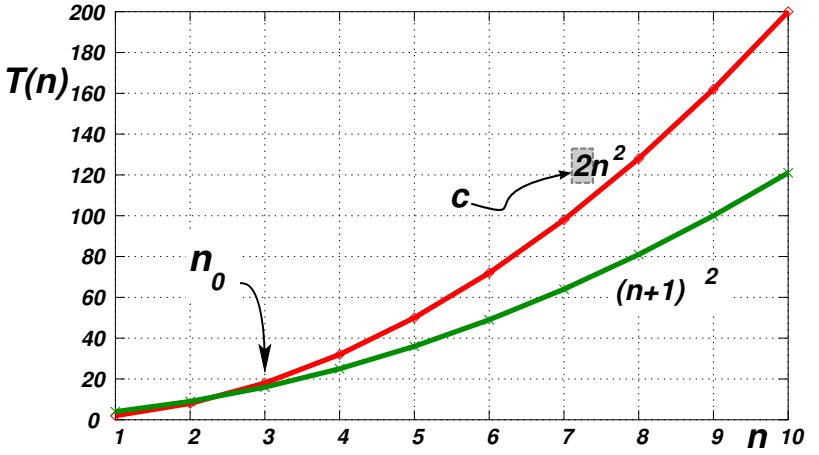


Figura 1.13: $T(n) = (n+1)^2$ es $O(n^2)$

La notación $O(...)$ puede usarse con otras funciones, es decir $O(n^3)$, $O(2^n)$, $O(\log n)$. En general decimos que $T(n) = O(f(n))$ *(se lee "$T(n)$ es orden $f(n)$") si existen constantes $c, n_0 > 0$ tales que*

$$T(n) \leq c\, f(n), \quad \text{para} \quad n \geq n_0 \tag{1.23}$$

Se suele llamar a $f(n)$ la *"tasa de crecimiento"* de $T(n)$ (también *"velocidad de crecimiento"* o *"complejidad algorítmica"*). De la definición de $O(..)$ pueden hacerse las siguientes observaciones.

1.3.2. Invariancia ante constantes multiplicativas

Podemos ver que la definición de la tasa de crecimiento es invariante ante constantes multiplicativas, es decir

$$T(n) = O(c\, f(n)) \quad \Longrightarrow \quad T(n) = O(f(n)) \tag{1.24}$$

Por ejemplo, si $T(n) = O(2n^3)$ entonces también es $O(n^3)$.
Demostración: Esto puede verse facilmente ya que si $T(n) = O(2n^3)$, entonces existen c y n_0 tales que $T(n) \leq c\, 2n^3$ para $n \geq n_0$, pero entonces también podemos decir que $T(n) \leq c' n^3$ para $n \geq n_0$, con $c' = 2c$.

1.3.3. Invariancia de la tasa de crecimiento ante valores en un conjunto finito de puntos

Es decir si

$$T_1(n) = \begin{cases} 100 & ;\ \text{para } n < 10, \\ (n+1)^2 & ;\ \text{para } n \geq 10, \end{cases} \tag{1.25}$$

entonces vemos que $T_1(n)$ coincide con la función $T(n) = (n+1)^2$ estudiada en el ejemplo 1.1. Por lo tanto, como sólo difieren en un número finito de puntos (los valores de $n < 10$) las dos son equivalentes y por lo tanto $T_1(n) = O(n^2)$ también.
Demostración: Esto puede verse ya que si $T(n) < 2n^2$ para $n \geq 3$ (como se vio en el ejemplo citado), entonces $T_1(n) < 2n^2$ para $n > n_0' = 10$.

1.3.4. Transitividad

La propiedad $O(\ )$ es transitiva, es decir si $T(n) = O(f(n))$ y $f(n) = O(g(n))$ entonces $T(n) = O(g(n))$.
Demostración: si $T(n) \leq cf(n)$ para $n \geq n_0$ y $f(n) \leq c'\, g(n)$ para $n \geq n_0'$, entonces $T(n) \leq c''\, g(n)$ para $n \geq n_0''$, donde $c'' = cc'$ y $n_0'' = \max(n_0, n_0')$.
En cierta forma, $O(...)$ representa una relación de orden entre las funciones (como "$<$" entre los números reales).

1.3.5. Regla de la suma

Si $f(n) = O(g(n))$, y a, b son constantes positivas, entonces $a\,f(n) + b\,g(n) = O(g(n))$. Es decir, si en una expresión tenemos una serie de términos, sólo queda el *"mayor"* de ellos (en el sentido de $O(...)$). Así por ejemplo, si $T(n) = 2n^3 + 3n^5$, entonces puede verse fácilmente que $n^3 = O(n^5)$ por lo tanto, $T(n) = O(n^5)$.
Demostración: Si $f(n) \leq cg(n)$ para $n \geq n_0$ entonces $a\,f(n) + b\,g(n) \leq c'g(n)$ para $n \geq n_0$ con $c' = ac + b$.
Nota: Pero la regla de la suma debe aplicarse un número constante de veces, si no, por ejemplo, consideremos una expresión como

$$T(n) = n^2 = n + n + ... + n. \tag{1.26}$$

Aplicando repetidamente la regla de la suma, podríamos llegar a la conclusión que $T(n) = O(n)$, lo cual es ciertamente falso.

1.3.6. Regla del producto

Si $T_1(n) = O(f_1(n))$ y $T_2(n) = O(f_2(n))$ entonces $T_1(n)T_2(n) = O(f_1(n)f_2(n))$.
Demostración: Si $T_1(n) \leq c_1 f_1(n)$ para $n \geq n_{01}$ y $T_2(n) \leq c_2 f_2(n)$ para $n \geq n_{02}$ entonces $T_1(n)T_2(n) \leq f_1(n)f_2(n)$ para $n > n_0 = \max(n_{01}, n_{02})$ con $c = c_1c_2$.

1.3.7. Funciones típicas utilizadas en la notación asintótica

Cualquier función puede ser utilizada como tasa de crecimiento, pero las más usuales son, en orden de crecimiento

$$1 < \log n < \sqrt{n} < n < n^2 < ... < n^p < 2^n < 3^n < \ldots < n! < n^n \tag{1.27}$$

- La función logaritmo crece menos que cualquier potencia n^α con $\alpha > 1$.
- Los logaritmos en diferente base son equivalentes entre sí por la bien conocida relación

  $$\log_b n = \log_b a\, \log_a n, \tag{1.28}$$

  de manera que en muchas expresiones con logaritmos no es importante la base utilizada. En computación científica es muy común que aparezcan expresiones con logaritmos en base 2.

- En (1.27) "$<$" representa $O(...)$. Es decir, $1 = O(\log n)$, $\log n = O(\sqrt{n})$, ...
- 1 representa las funciones constantes.
- Las potencias n^α (con $\alpha > 0$) se comparan entre sí según sus exponentes, es decir $n^\alpha = O(n^\beta)$ si $\alpha \leq \beta$. Por ejemplo, $n^2 = O(n^3)$, $n^{1/2} = O(n^{2/3})$.
- Las exponenciales a^n (con $a > 1$) se comparan según su base, es decir que $a^n = O(b^n)$ si $a \leq b$. Por ejemplo $2^n = O(3^n)$. En los problemas de computación científica es muy común que aparezcan expresiones con base $a = 2$.

Ejemplo 1.2: Notar que los valores de c y n_0 no son absolutos. En el caso del ejemplo 1.1 podríamos demostrar, con un poco más de dificultad, que $T(n) > 1.1n^2$ para $n \geq 21$.

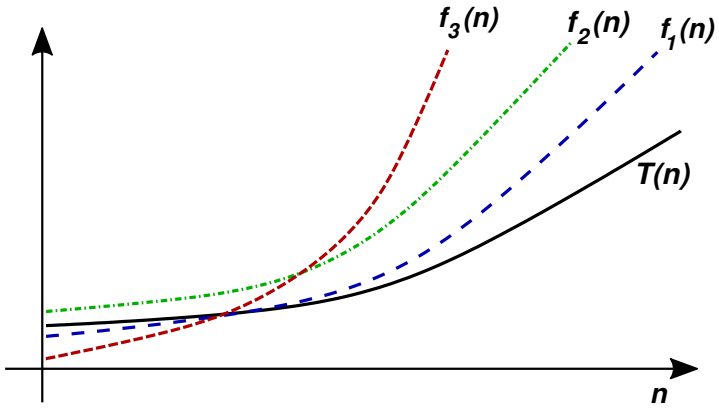


Figura 1.14: Decir que $T(n) = O(f_1(n))$ es *"más fuerte"* que $T(n) = O(f_2(n))$ o $T(n) = O(f_3(n))$

Ejemplo 1.3: Para $T(n) = (n+1)^2$ entonces también podemos decir que $T(n) = O(n^3)$ ya que, como vimos antes $T(n) = O(n^2)$ y $n^2 = O(n^3)$ de manera que por la transitividad (ver §1.3.4) $T(n) = O(n^3)$, pero decir que $T(n) = O(n^2)$ es una aseveración *"más fuerte"* del tiempo de ejecución ya que $n^2 < n^3$. En general (ver figura 1.14) si podemos tomar varias funciones $f_1(n)$, $f_2(n)$, $f_3(n)$ entonces debemos tomar la *"menor"* de todas. Así, si bien podemos decir que $T(n) = O(n^\alpha)$ para cualquier $\alpha \geq 2$ la más fuerte de todas es para $T(n) = O(n^2)$. Por otra parte puede verse que $T(n) \neq$

$O(n^\alpha)$ para $\alpha < 2$. Razonemos por el absurdo. Si, por ejemplo, fuera cierto que $T(n) = O(n)$ entonces deberían existir $c, n_0 > 0$ tales que $T(n) = (n+1)^2 \leq cn$ para $n \geq n_0$. Pero entonces

$$\frac{(n+1)^2}{n} \leq c, \quad \text{para } n \geq n_0 \tag{1.29}$$

lo cual es ciertamente falso ya que

$$\frac{(n+1)^2}{n} \to \infty, \quad \text{para } n \to \infty. \tag{1.30}$$

Esta demostración puede extenderse fácilmente para cualquier $\alpha < 2$.

1.3.8. Equivalencia

Si dos funciones f y g satisfacen que $f(n) = O(g(n))$ y $g(n) = O(f(n))$ entonces decimos que *"sus tasas de crecimiento son equivalentes"* lo cual denotamos por

$$f \sim g \tag{1.31}$$

Así, en el ejemplo anterior 1.3 se puede demostrar ver que $n^2 = O((n+1)^2)$ por lo que

$$T(n) = (n+1)^2 \sim n^2 \tag{1.32}$$

1.3.9. La función factorial

Una función que aparece muy comúnmente en problemas combinatorios es la función factorial $n!$ (ver §1.1.7). Para poder comparar esta función con otras para grandes valores de n es conveniente usar la *"aproximación de Stirling"*

$$n! \sim \sqrt{2\pi}\, n^{n+1/2}\, \mathrm{e}^{-n} \tag{1.33}$$

Gracias a esta aproximación es fácil ver que

$$n! = O(n^n) \tag{1.34}$$

y

$$a^n = O(n!), \tag{1.35}$$

lo cual justifica la ubicación del factorial en la tabla (1.3.7).

Demostración: La primera relación (1.34) se desprende fácilmente de aplicar la aproximación de Stirling y del hecho que $n^{1/2}\mathrm{e}^{-n} \to 0$ para $n \to \infty$.

La segunda relación (1.35) se deduce de

$$n^{n+1/2}\,\mathrm{e}^{-n} = a^n \left(\frac{n}{a\mathrm{e}}\right)^n n^{1/2} \tag{1.36}$$

Entonces para $n \geq n_0 = \max(a\mathrm{e}, 1)$

$$\left(\frac{n}{a\mathrm{e}}\right)^n \geq 1 \tag{1.37}$$

y

$$n^{1/2} \geq n_0^{1/2} \tag{1.38}$$

con lo cual

$$n^{n+1/2}\,\mathrm{e}^{-n} \geq n_0^{1/2}\,a^n \tag{1.39}$$

y por lo tanto

$$a^n \leq n_0^{-1/2} n^{n+1/2}\,\mathrm{e}^{-n} \tag{1.40}$$

entonces

$$a^n = O(n^{n+1/2}\,\mathrm{e}^{-n}) = O(n!) \tag{1.41}$$

con $c = n_0^{-1/2}$.

Ejemplo 1.4: Una de las ventajas de la notación asintótica es la gran simplificación que se obtiene en las expresiones para los tiempos de ejecución de los programas. Por ejemplo, si

$$T(n) = (3n^3 + 2n^2 + 6)\,n^5 + 2^n + 16n! \tag{1.42}$$

entonces vemos que, aplicando la regla de la suma, la expresión entre paréntesis puede estimarse como

$$(3n^3 + 2n^2 + 6) = O(n^3) \tag{1.43}$$

Aplicando la regla del producto, todo el primer término se puede estimar como

$$(3n^3 + 2n^2 + 6)\,n^5 = O(n^8) \tag{1.44}$$

Finalmente, usando la tabla (1.3.7) vemos que el término que gobierna es el último de manera que

$$T(n) = O(n!) \tag{1.45}$$

Muchas veces los diferentes términos que aparecen en la expresión para el tiempo de ejecución corresponde a diferentes partes del programa, de manera que, como ganancia adicional, la notación asintótica nos indica cuáles son las partes del programa que requieren más tiempo de cálculo y, por lo tanto, deben ser eventualmente optimizadas.

1.3.10. Determinación experimental de la tasa de crecimiento

A veces es difícil determinar en forma analítica la tasa de crecimiento del tiempo de ejecución de un algoritmo. En tales casos puede ser útil determinarla aunque sea en forma *"experimental"* es decir corriendo el programa para una serie de valores de n, tomar los tiempos de ejecución y a partir de estos datos obtener la tasa de crecimiento. Por ejemplo, para el algoritmo heurístico de la sección §1.1.8, si no pudiéramos encontrar el orden de convergencia, entonces ejecutamos el programa con una serie de valores de n obteniendo los valores de la tabla 1.2.

n	$T(n)$ [segundos]
300	0.2
600	1.2
1000	4.8
1500	14.5
3000	104.0

Tabla 1.2: Tiempo de ejecución del algoritmo ávido de la sección §1.1.8.

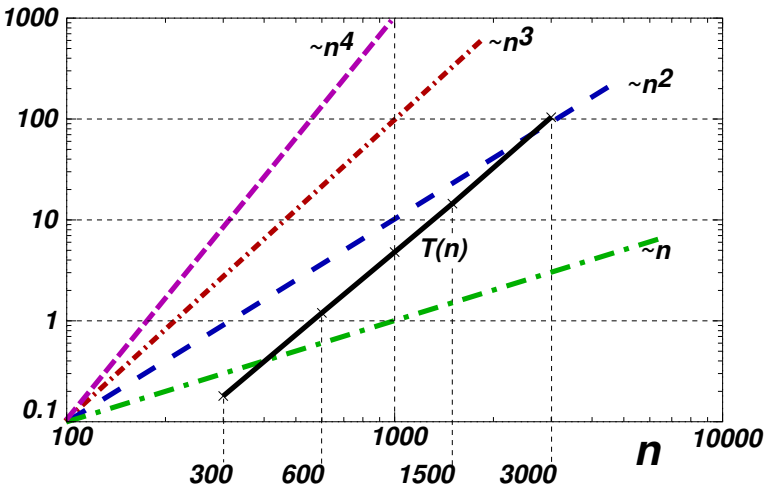


Figura 1.15: Determinación experimental del tiempo de ejecución del algoritmo ávido de coloración de la sección §1.1.8

Graficando los valores en ejes logarítmicos (es decir, graficar $\log T(n)$ en función de $\log n$) obtenemos un gráfico como el de la figura 1.15. La uti-

lidad de tales ejes es que las funciones de tipo potencia $\propto n^\alpha$ resultan ser rectas cuya pendiente es proporcional a α. Además curvas que difieren en una constante multiplicativa resultan ser simplemente desplazadas según la dirección vertical. Pero entonces si un programa tiene un comportamiento $T(n) = O(n^\alpha)$ pero no conocemos α, basta con graficar su tiempo de ejecución en ejes logarítmicos junto con varias funciones n^α y buscar cuál de ellas es paralela a la de $T(n)$. En el ejemplo de la figura vemos que $T(n)$ resulta ser perfectamente paralela a n^3 confirmando nuestra estimación de la sección §1.1.10.

En casos donde el tiempo de ejecución es exponencial, es decir $\sim a^n$ pueden ser preferibles ejes semi-logarítmicos, es decir, graficar $\log T(n)$ en función de n (y no en función de $\log n$ como en los logarítmicos) ya que en estos gráficos las exponenciales son rectas, con pendiente $\log a$.

Si no tenemos idea de que tipo de tasa de crecimiento puede tener un programa, podemos proceder en forma incremental. Primero probar con un gráfico logarítmico, si la curva resulta ser una recta, entonces es una potencia y determinando la pendiente de la recta determinamos completamente la tasa de crecimiento. Si la función no aparece como una recta y tiende a acelerarse cada vez más, de manera que crece más que cualquier potencia, entonces podemos probar con las exponenciales, determinando eventualmente la pendiente correspondiente. Finalmente, si crece todavía más que las exponenciales, entonces puede ser del tipo $n!$ o n^n. En este caso pueden intentarse una serie de procedimientos para refinar la estimación, pero debemos notar que en muchos casos basta con notar que la tasa de crecimiento es mayor que cualquier exponencial para calificar al algoritmo.

1.3.11. Otros recursos computacionales

Las técnicas desarrolladas en esta sección pueden aplicarse a cualquier otro tipo de recurso computacional, no sólo el tiempo de ejecución. Otro recurso muy importante es la memoria total requerida. Veremos, por ejemplo, en el capítulo sobre algoritmos de ordenamiento que el algoritmo de ordenamiento por *"intercalamiento"* (*merge-sort*) tiene un tiempo de ejecución $O(n \log n)$ en el caso promedio, pero llega a ser $O(n^2)$ en el peor caso. Pero existe una versión modificada que es siempre $O(n \log n)$, sin embargo la versión modificada requiere una memoria adicional que es a lo sumo $O(\sqrt{n})$.

1.3.12. Tiempos de ejecución no-polinomiales

Se dice que un algoritmo tiene tiempo *"polinomial"* (*"P"*, para abreviar), si es $T(n) = O(n^\alpha)$ para algún α. Por contraposición, aquellos algoritmos que tienen tiempo de ejecución mayor que cualquier polinomio (funciones exponenciales a^n, $n!$, n^n) se les llama *"no polinomiales"*. Por ejemplo, en el caso del problema de coloración de grafos discutido en la sección §1.1.1, el algoritmo exhaustivo descripto en §1.1.4 resulta ser no polinomial mientras que el descripto en la sección §1.1.8 es P. Esta nomenclatura ha sido originada por la gran diferencia entre la velocidad de crecimiento entre los algoritmos polinomiales y no polinomiales. En muchos casos, determinar que un algoritmo es no polinomial es la razón para descartar el algoritmo y buscar otro tipo de solución.

1.3.13. Problemas P y NP

Si bien para ciertos problemas (como el de colorear grafos) no se conocen algoritmos con tiempo de ejecución polinomial, es muy difícil demostrar que realmente es así, es decir que para ese problema no existe ningún algoritmo de complejidad polinomial.

Para ser más precisos hay que introducir una serie de conceptos nuevos. Por empezar, cuando se habla de tiempos de ejecución se refiere a instrucciones realizadas en una *"máquina de Turing"*, que es una abstracción de la computadora más simple posible, con un juego de instrucciones reducido. Una *"máquina de Turing no determinística"* es una máquina de Turing que en cada paso puede *invocar* un cierto número de instrucciones, y no una sola instruccion como es el caso de la máquina de Turing determinística. En cierta forma, es como si una máquina de Turing no-determinística pudiera invocar otras series de máquinas de Turing, de manera que en vez de tener un *"camino de cómputo"*, como es usual en una computadora secuencial, tenemos un *"árbol de cómputo"*. Un problema es *"NP"* si tiene un tiempo de ejecución polinomial en una máquina de Turing no determinística (*NP* significa aquí *non-deterministic polynomial*, y no *non-polynomial*). La pregunta del millón de dólares (literalmente!, ver http://www.claymath.org) es si existen problemas en NP para los cuales no exista un algoritmo polinomial.

Una forma de simplificar las cosas es demostrar que un problema se *"reduce"* a otro. Por ejemplo, el problema de hallar la mediana de un conjunto de N números (ver §5.4.2) (es decir el número tal que existen $N/2$ números menores o iguales que él y otros tantos $N/2$ mayores o iguales) se reduce

a poner los objetos en un vector y ordenarlo, ya que una vez ordenado basta con tomar el elemento de la posición media. De manera que si tenemos un cierto algoritmo con una complejidad algorítmica para el ordenamiento, automáticamente tenemos una cota superior para el problema de la mediana. Se dice que un problema es *"NP-completo"* (NPC) si cualquier problema de NP se puede reducir a ese problema. Esto quiere decir, que los problemas de NPC son los candidatos a tener la más alta complejidad algorítimica de NP. Se ha demostrado que varios problemas pertenecen a NPC, entre ellos el *Problema del Agente Viajero* (1.1). Si se puede demostrar que algún problema de NPC tiene complejidad algorítmica no-polinomial (y por lo tanto todos los problemas de NPC) entonces $P \neq NP$. Por otra parte, si se encuentra algún algoritmo de tiempo polinomial para un problema de NPC entonces todos los problemas de NPC (y por lo tanto de NP) serán P, es decir $P = NP$. De aquí la famosa forma de poner la pregunta del millón que es: *¿"Es P=NP"?*.

1.3.14. Varios parámetros en el problema

No siempre hay un sólo parámetro n que determina el tamaño del problema. Volviendo al ejemplo de la coloración de grafos, en realidad el tiempo de ejecución depende no sólo del número de vértices, sino también del número de aristas n_e, es decir $T(m, n_e)$. Algunos algoritmos pueden ser más apropiados cuando el número de aristas es relativamente bajo (grafos *"ralos"*) pero ser peor cuando el número de aristas es alto (grafos *"densos"*). En estos casos podemos aplicar las técnicas de esta sección considerando uno de los parámetros fijos y (digamos n_e) y considerar la tasa de crecimiento en función del otro parámetro m y viceversa. Otras veces es conveniente definir un parámetro adimensional como la *"tasa de ralitud"* (*"sparsity ratio"*)

$$s = \frac{n_e}{m(m-1)/2} \tag{1.46}$$

y considerar el crecimiento en función de m a tasa de ralitud constante. (Notar que $s = 1$ para un grafo completamente *conectado*, $s = 0$ para un grafo completamente *desconectado*).

Por supuesto el análisis de los tiempos de ejecución se hace mucho más complejo cuantos más parámetros se consideran.

1.4. Conteo de operaciones para el cálculo del tiempo de ejecución

Comenzaremos por asumir que no hay llamadas recursivas en el programa. (Ni cadenas recursivas, es decir, **`sub1()`** llama a **`sub()`** y **`sub2()`** llama a **`sub1()`**). Entonces, debe haber rutinas que no llaman a otras rutinas. Comenzaremos por calcular el tiempo de ejecución de éstas. La regla básica para calcular el tiempo de ejecución de un programa es ir *desde los lazos o construcciones más internas hacia las más externas*. Se comienza asignando un costo computacional a las sentencias básicas. A partir de esto se puede calcular el costo de un bloque, sumando los tiempos de cada sentencia. Lo mismo se aplica para funciones y otras construcciones sintácticas que iremos analizando a continuación.

1.4.1. Bloques if

Para evaluar el tiempo de un bloque **`if`**

```
if(<cond>) {
  <body>
}
```

podemos o bien considerar el peor caso, asumiendo que <*body*> se ejecuta siempre

$$T_{\text{peor}} = T_{\text{cond}} + T_{\text{body}} \tag{1.47}$$

o, en el caso promedio, calcular la probabilidad P de que <*cond*> de verdadero. En ese caso

$$T_{\text{prom}} = T_{\text{cond}} + PT_{\text{body}} \tag{1.48}$$

Notar que T_{cond} no está afectado por P ya que la condición se evalúa siempre. En el caso de que tenga un bloque **`else`**, entonces

```
if(<cond>) {
  <body-true>
} else {
  <body-false>
}
```

podemos considerar,

$$\begin{aligned} T_{\text{peor}} &= T_{\text{cond}} + \max\left(T_{\text{body-true}}, T_{\text{body-false}}\right) \\ &\leq T_{\text{cond}} + T_{\text{body-true}} + T_{\text{body-false}} \\ T_{\text{prom}} &= T_{\text{cond}} + P\,T_{\text{body-true}} + (1-P)\,T_{\text{body-false}} \end{aligned} \tag{1.49}$$

Las dos cotas para T_{peor} son válidas, la que usa "$\max$" es más precisa.

1.4.2. Lazos

El caso más simple es cuando el lazo se ejecuta un número fijo de veces, y el cuerpo del lazo tiene un tiempo de ejecución constante,

```
for (i=0; i<N; i++) {
   <body>
}
```

donde $T_{\text{body}} =$ constante. Entonces

$$T = T_{\text{ini}} + N(\,T_{\text{body}} + T_{\text{inc}} + T_{\text{stop}}) \tag{1.50}$$

donde

- T_{ini} es el tiempo de ejecución de la parte de *"inicialización"* del lazo, en este caso **`i=0`**,
- T_{inc} es el tiempo de ejecución de la parte de *"incremento"* del contador del lazo, en este caso, **`i++`** y
- T_{stop} es el tiempo de ejecución de la parte de *"detención"* del contador del lazo, en este caso, **`i<N`**.

En el caso más general, cuando T_{body} no es constante, debemos evaluar explícitamente la suma de todas las contribuciones,

$$T = T_{\text{ini}} + \sum_{i=0}^{N-1} (T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}}). \tag{1.51}$$

Algunas veces es difícil calcular una expresión analítica para tales sumas. Si podemos determinar una cierta tasa de crecimiento para todos los términos

$$T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}} = O(f(n)), \quad \text{para todo } i \tag{1.52}$$

entonces,

$$T \leq N \max_{i=1}^{N-1}(T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}} = O(Nf(n)) \tag{1.53}$$

Más difícil aún es el caso en que el número de veces que se ejecuta el lazo no se conoce a priori, por ejemplo un lazo **while** como el siguiente

```
while (<cond>) {
  <body>
}
```

En este caso debemos determinar también el número de veces que se ejecutará el lazo.

Ejemplo 1.5: Calcularemos el tiempo de ejecución del algoritmo de ordenamiento por el *"método de la burbuja"* (*"bubble-sort"*). Si bien a esta altura no es necesario saber exactamente como funciona el método, daremos una breve descripción del mismo. La función **bubble_sort(...)** toma como argumento un vector de enteros y los ordena de menor a mayor. En la ejecución del lazo de las líneas 6–16 para **j=0** el menor elemento de todos es insertado en **a[0]** mediante una serie de intercambios. A partir de ahí **a[0]** no es tocado más. Para **j=1** el mismo procedimiento es aplicado al rango de índices que va desde **j=1** hasta **j=n-1**, donde **n** es el número de elementos en el vector, de manera que después de la ejecución del lazo para **j=1** el segundo elemento menor es insertado en **a[1]** y así siguiendo hasta que todos los elementos terminan en la posición que les corresponde en el elemento ordenado.

```
1.  void bubble_sort(vector<int> &a) {
2.    int n = a.size();
3.    // Lazo externo. En cada ejecucion de este lazo
4.    // el elemento j-esimo menor elemento llega a la
5.    // posicion 'a[j]'
6.    for (int j=0; j<n-1; j++) {
7.      // Lazo interno. Los elementos consecutivos se
8.      // van comparando y eventualmente son intercambiados.
9.      for (int k=n-1; k>j; k--) {
10.       if (a[k-1] > a[k]) {
11.         int tmp = a[k-1];
12.         a[k-1] = a[k];
13.         a[k]=tmp;
14.       }
15.     }
16.   }
```

```
17. }
```

Código 1.9: *Algoritmo de clasificación por el método de la burbuja. [Archivo: bubble.cpp]*

En el lazo interno (líneas 9–15) se realizan una serie de intercambios de manera de llevar el menor elemento del rango `k=j` a `k=n-1` a su posición. En cada ejecución del lazo se ejecuta en forma condiconal el cuerpo del bloque `if` (líneas 11–13). Primero debemos encontrar el número de operaciones que se realiza en el cuerpo del bloque `if`, luego sumarlos para obtener el tiempo del lazo interno y finalmente sumarlo para obtener el del lazo externo.

El bloque de las líneas 11–13 requiere un número finito de operaciones (asignaciones de enteros, operaciones de adición/sustracción con enteros, referenciación de elementos de vectores). Llamaremos c_0 al número total de operaciones. Por supuesto lo importante aquí es que c_0 no depende de la longitud del vector n. Con respecto al condicional (líneas 10–14) tenemos que, si llamamos c_1 al número de operaciones necesarias para evaluar la condición `a[k-1]>a[k]`, entonces el tiempo es $c_0 + c_1$ cuando la condición da verdadera y c_1 cuando da falsa. Como de todas formas ambas expresiones son constantes ($O(1)$), podemos tomar el criterio de estimar el peor caso, es decir que siempre se ejecute, de manera que el tiempo de ejecución de las líneas (líneas 10–14) es $c_0 + c_1$, constante. Pasando al lazo interno (líneas 9–15) éste se ejecuta desde $k = n-1$ hasta $k = j+1$, o sea un total de $(n-1) - (j+1) + 1 = n - j - 1$ veces. Tanto el cuerpo del lazo, como el incremento y condición de detención (línea 9) consumen un número constante de operaciones. Llamando c_2 a este número de operaciones tenemos que

$$T_{\text{lineas } 9-15} = c_3 + (n - j - 1)\, c_2 \tag{1.54}$$

donde c_3 es el número de operaciones en la inicialización del lazo (`k=n-1`). Para el lazo externo (líneas 6–16) tenemos

$$\begin{aligned} T_{\text{ini}} &= c_4, \\ T_{\text{stop}} &= c_5, \\ T_{\text{inc}} &= c_6, \\ T_{\text{body},j} &= c_3 + (n - j - 1)\, c_2. \end{aligned} \tag{1.55}$$

Lamentablemente el cuerpo del lazo no es constante de manera que no po-

demos aplicar (1.50), sino que debemos escribir explícitamente una suma

$$T(\text{líneas 6–16}) = c_4 + \sum_{j=0}^{n-2}(c_5 + c_6 + c_3 + (n-j-1)c_2) \tag{1.56}$$

Los términos c_3, c_5 y c_6 dentro de la suma son constantes, de manera que podemos poner

$$T(\text{líneas 6–16}) = c_4 + (n-1)(c_3 + c_5 + c_6) + c_2 \sum_{j=0}^{n-2}(n-j-1) \tag{1.57}$$

Ahora debemos hallar una expresión para la sumatoria. Notemos que

$$\sum_{j=0}^{n-2}(n-j-1) = (n-1) + (n-2) + ... + 1 = \left(\sum_{j=1}^{n} j\right) - n. \tag{1.58}$$

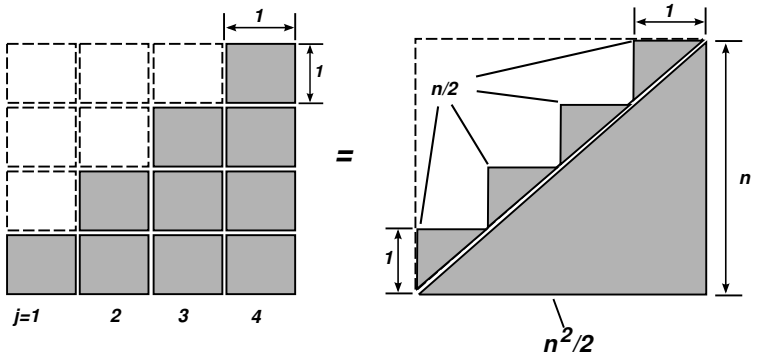


Figura 1.16: Cálculo gráfico de la suma en (1.59)

Consideremos entonces la expresión

$$\sum_{j=1}^{n} j \tag{1.59}$$

Gráficamente, podemos representar esta suma por una serie de columnas de cuadrados de lado unitario. En la primera columna tenemos un sólo cuadrado, en la segunda 2, hasta que en la última tenemos n. Para $n = 4$ tenemos una situación como en la figura 1.16. La suma (1.59) representa el

área sombreada de la figura, pero ésta puede calcularse usando la construcción de la izquierda, donde se ve que el área es igual a la mitad inferior del cuadrado, de área $n^2/2$ más la suma de n áreas de triángulos de área $1/2$, por lo tanto

$$\sum_{j=1}^{n} j = \frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2} \tag{1.60}$$

Podemos verificar la validez de esta fórmula para valores particulares de n, por ejemplo para $n = 4$ da 10, lo cual coincide con la suma deseada (1+2+3+4).

Volviendo a (1.57), vemos entonces que, usando (1.58) y (1.60)

$$\begin{aligned} T(\text{líneas 6–16}) &= c_4 + (n-1)(c_5 + c_6) + c_2 \left(\frac{n(n+1)}{2} - n \right) \\ &= c_4 + (n-1)(c_5 + c_6) + c_2 \frac{n(n-1)}{2} \end{aligned} \tag{1.61}$$

Finalmente, notemos que el tiempo total de la función **`bubble_sort()`** es igual al del lazo externo más un número constante de operaciones (la llamada a **`vector<...>::size()`** es de tiempo constante, de acuerdo a la documentación de STL). Llamando c_7 a las operaciones adicionales, incluyendo la llamada a la función, tenemos

$$T(\textbf{bubble\_sort}) = c_4 + c_7 + (n-1)(c_5 + c_6) + c_2 \frac{n(n-1)}{2} \tag{1.62}$$

Ahora vamos a simplificar esta expresión utilizando los conceptos de notación asintótica. Primero notemos que

$$T(\textbf{bubble\_sort}) \leq (c_4 + c_7) + n(c_5 + c_6) + c_2 \frac{n^2}{2} \tag{1.63}$$

y que los tres términos involucrados son $O(1)$, $O(n)$ y $O(n^2)$ respectivamente. De manera que, aplicando la regla de la suma, tenemos que

$$T(\textbf{bubble\_sort}) = O(n^2) \tag{1.64}$$

Si bien, estos cálculos pueden parecer tediosos y engorrosos al principio, poco a poco el programador se va acostumbrando a hacerlos mentalmente y con experiencia, se puede hallar la tasa de crecimiento para funciones como **`bubble_sort()`** simplemente por inspección.

1.4.3. Suma de potencias

Sumas como (1.60) ocurren frecuentemente en los algoritmos con lazos anidados. Una forma alternativa de entender esta expresión es aproximar la suma por una integral (pensando a j como una variable continua)

$$\sum_{j=1}^{n} j \approx \int_0^n j \, \mathrm{d}j = \frac{n^2}{2} = O(n^2). \tag{1.65}$$

De esta forma se puede llegar a expresiones asintóticas para potencias más elevadas

$$\sum_{j=1}^{n} j^2 \approx \int_0^n j^2 \, \mathrm{d}j = \frac{n^3}{3} = O(n^3) \tag{1.66}$$

y, en general

$$\sum_{j=1}^{n} j^p \approx \int_0^n j^p \, \mathrm{d}j = \frac{n^{p+1}}{p+1} = O(n^{p+1}) \tag{1.67}$$

1.4.4. Llamadas a rutinas

Una vez calculados los tiempos de ejecución de rutinas que no llaman a otras rutinas (llamemos al conjunto de tales rutinas S_0), podemos calcular el tiempo de ejecución de aquellas rutinas que sólo llaman a las rutinas de S_0 (llamemos a este conjunto S_1), asignando a las líneas con llamadas a rutinas de S_0 de acuerdo don el tiempo de ejecución previamente calculado, como si fuera una instrucción más del lenguaje.

Por ejemplo, si un programa tiene un árbol de llamadas como el de la figura 1.17, entonces podemos empezar por calcular los tiempos de ejecución de las rutinas `sub4()` y `sub5()`, luego los de `sub1()` y `sub2()`, luego el de `sub3()` y finalmente el de `main()`.

1.4.5. Llamadas recursivas

Si hay llamadas recursivas, entonces el principio anterior no puede aplicarse. Una forma de evaluar el tiempo de ejecución en tales casos es llegar a una expresión recursiva para el tiempo de ejecución mismo.

```
1. int bsearch2(vector<int> &a,int k,int j1, int j2) {
2.   if (j1==j2-1) {
3.     if (a[j1]==k) return j1;
```

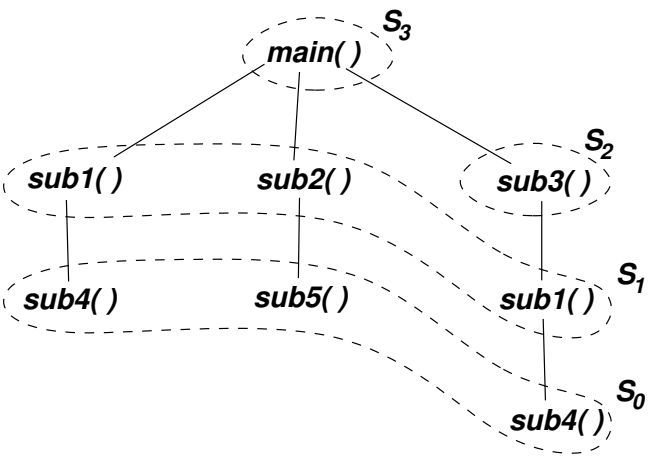


Figura 1.17: Ejemplo de árbol de llamadas de un programa y como calcular los tiempos de ejecución

```
4.     else return j2;
5.   } else {
6.     int p = (j1+j2)/2;
7.     if (k<a[p]) return bsearch2(a,k,j1,p);
8.     else return bsearch2(a,k,p,j2);
9.   }
10. }
11.
12. int bsearch(vector<int> &a,int k) {
13.   int n = a.size();
14.   if (k<a[0]) return 0;
15.   else return bsearch2(a,k,0,n);
16. }
```

Código 1.10: *Algoritmo de búsqueda binaria. [Archivo: bsearch.cpp]*

Ejemplo 1.6: Consideremos el algoritmo de *"búsqueda binaria"* (*"binary search"*) que permite encontrar un valor dentro de un vector ordenado. En el código 1.10 vemos una implementación típica. La rutina **bsearch()** toma como argumentos un **vector<int>** (que se asume que esta ordenado de menor a mayor y sin valores repetidos) y un entero k y retorna la primera posición j dentro del arreglo tal que $k \leq a_j$. Si k es mayor que todos los elementos de a, entonces debe retornar n, como si en la posición n,

(que está fuera del arreglo) hubiera un ∞. La rutina utiliza una rutina auxiliar **bsearch2(a,k,j1,j2)** la cual busca el elemento en un rango $[j_1, j_2)$ (que significa $j_1 \leq j < j_2$). Este rango debe ser un rango válido en a, es decir $j_1 < j_2$, $0 \leq j_1 < n$, $1 \leq j_2 \leq n$ y $a[j_1] \leq k < a[j_2]$. Notar que j_2 puede tomar la posición *"ficticia"* n, pero j_1 no.

La rutina **bsearch()** determina primero un rango válido inicial. Si $k \geq a_0$, entonces $[0, n)$ es un rango válido y llama a **bsearch()** mientras que si no la posición $j = 0$ es la solución al problema.

La rutina **bsearch2** opera recursivamente calculando un punto medio p y llamando nuevamente a **bsearch2()**, ya sea con el intervalo $[j_1, p)$ o $[p, j_2)$. En cada paso el tamaño del rango se reduce en un factor cercano a 2, de manera que en un cierto número de pasos el tamaño del intervalo se reduce a 1, en cuyo caso termina la recursión.

Consideremos ahora el tiempo de ejecución de la función **bsearch2()** como función del número de elementos $m = j_2 - j_1$ en el intervalo. Si la condición de la línea 2 da verdadero entonces $m = 1$ y el tiempo es una constante c. Caso contrario, se realiza un número constante de operaciones d más una llamada a **bsearch2()** (en la línea 7 ó la 8) con un rango de longitud menor. Por simplicidad asumiremos que m es una potencia de 2, de manera que puede verse que el nuevo intervalo es de longitud $m/2$. Resumiendo

$$T(m) = \begin{cases} c & ;\ \text{si } m = 1; \\ d + T(m/2) & ;\ \text{si } m > 1; \end{cases} \tag{1.68}$$

Ahora, aplicando recursivamente esta expresión, tenemos que

$$\begin{aligned} T(2) &= d + T(1) = d + c \\ T(4) &= d + T(2) = 2d + c \\ T(8) &= d + T(4) = 3d + c \\ &\vdots \\ T(2^p) &= d + T(2^{p-1}) = pd + c \end{aligned} \tag{1.69}$$

como $p = \log_2 m$, vemos que

$$T(m) = d \log_2 m + c = O(\log_2 m) \tag{1.70}$$

Notar que el algoritmo más simple, consistente en recorrer el vector hasta el primer elemento mayor que k sería $O(n)$ en el peor caso y $O(n/2)$ en promedio (siempre que el elemento este en el vector), ya que en promedio

encontrará al elemento en la parte media del mismo. El reducir el tiempo de ejecución de $O(n)$ a $O(\log n)$ es, por supuesto, la gran ventaja del algoritmo de búsqueda binaria.

Capítulo 2

Tipos de datos abstractos fundamentales

En este capítulo se estudiarán varios tipos de datos básicos. Para cada uno de estos TAD se discutirán en el siguiente orden

1. Sus operaciones abstractas.
2. Una interfaz básica en C++ y ejemplos de uso con esta interfaz.
3. Una o más implementaciones de esa interfaz, discutiendo las ventajas y desventajas de cada una, tiempos de ejecución ...
4. Una interfaz más avanzada, compatible con las STL, usando templates, sobrecarga de operadores y clases anidadas y ejemplos de uso de esta interfaz.
5. Una implementación de esta interfaz.

La razón de estudiar primero la interfaz básica (sin templates, sobrecarga de operadores ni clases anidadas) es que estos ítems pueden ser demasiado complejos de entender, en cuanto a sintaxis, para un programador principiante, y en este libro el énfasis esta puesto en el uso de la interfaz, el concepto de TAD y la comprensión de los tiempos de ejecución de los diferentes algoritmos, y no en sutilezas sintácticas del C++. De todas formas, en las fases 4 y 5 se muestra una interfaz compatible con la STL, ejemplos de uso e implementación, ya que pretendemos que este libro sirva también para aprender a usar correcta y eficientemente las STL.

2.1. El TAD Lista

Las listas constituyen una de las estructuras lineales más flexibles, porque pueden crecer y acortarse según se requiera, insertando o suprimiendo elementos tanto en los extremos como en cualquier otra posición de la lista. Por supuesto esto también puede hacerse con vectores, pero en las implementaciones más comunes estas operaciones son $O(n)$ para los vectores, mientras que son $O(1)$ para las listas. El poder de las listas es tal que la familia de lenguajes derivados del Lisp, que hoy en día cuenta con el Lisp, Common Lisp y Scheme, entre otros, el lenguaje mismo está basado en la lista (*"Lisp"* viene de *"list processing"*).

2.1.1. Descripción matemática de las listas

Desde el punto de vista abstracto, una lista es una secuencia de cero o más elementos de un tipo determinado, que en general llamaremos `elem_t`, por ejemplo `int` o `double`. A menudo representamos una lista en forma impresa como una sucesión de elementos entre paréntesis, separados por comas

$$L = (a_0, a_1, \ldots, a_{n-1}) \tag{2.1}$$

donde $n \geq 0$ es el número de elementos de la lista y cada a_i es de tipo `elem_t`. Si $n \geq 1$ entonces decimos que a_0 es el primer elemento y a_{n-1} es el último elemento de la lista. Si $n = 0$ decimos que la lista *"está vacía"*. Se dice que n es la *"longitud"* de la lista.

Una propiedad importante de la lista es que sus elementos están ordenados en forma lineal, es decir, para cada elemento a_i existe un sucesor a_{i+1} (si $i < n-1$) y un predecesor a_{i-1} (si $i > 0$). Este orden es parte de la lista, es decir, dos listas son iguales si tienen los mismos elementos *y en el mismo orden*. Por ejemplo, las siguientes listas son distintas

$$(1, 3, 7, 4) \neq (3, 7, 4, 1) \tag{2.2}$$

Mientras que en el caso de conjuntos, éstos serían iguales. Otra diferencia con los conjuntos es que puede haber elementos repetidos en una lista, mientras que en un conjunto no.

Decimos que el elemento a_i *"está en la posición i"*. También introducimos la noción de una posición ficticia n que *está fuera de la lista*. A las posiciones en el rango $0 \leq i \leq n-1$ las llamamos *"dereferenciables"* ya que pertenecen a un objeto real, y por lo tanto podemos obtener una referencia a ese objeto.

Notar que, a medida que se vayan insertando o eliminando elementos de la lista la posición ficticia n va variando, de manera que convendrá tener un método de la clase **`end()`** que retorne esta posición.

2.1.2. Operaciones abstractas sobre listas

Consideremos un operación típica sobre las listas que consiste en eliminar todos los elementos duplicados de la misma. El algoritmo más simple consiste en un doble lazo, en el cual el lazo externo sobre i va desde el comienzo hasta el último elemento de la lista. Para cada elemento i el lazo interno recorre desde $i+1$ hasta el último elemento, eliminado los elementos iguales a i. Notar que no hace falta revisar los elementos anteriores a i (es decir, los elementos j con $j < i$), ya que, por construcción, todos los elementos de 0 hasta i son distintos.

Este problema sugiere las siguientes operaciones abstractas

- Dada una posición i, *"insertar"* el elemento x en esa posición, por ejemplo

$$\begin{aligned} &L = (1, 3, 7) \\ &\text{inserta } 5 \text{ en la posición } 2 \\ &\rightarrow L = (1, 3, 5, 7) \end{aligned} \tag{2.3}$$

  Notar que el elemento 7, que estaba en la posición 2, se desplaza hacia el fondo, y termina en la posición 3. Notar que es válido insertar en cualquier posición dereferenciable, o en la posición ficticia **`end()`**

- Dada una posición i, *"suprimir"* el elemento que se encuentra en la misma. Por ejemplo,

$$\begin{aligned} &L = (1, 3, 5, 7) \\ &\text{suprime elemento en la posición } 2 \\ &\rightarrow L = (1, 3, 7) \end{aligned} \tag{2.4}$$

  Notar que esta operación es, en cierta forma, la inversa de insertar. Si, como en el ejemplo anterior, insertamos un elemento en la posición i y después suprimimos en esa misma posición, entonces la lista queda inalterada. Notar que *sólo* es válido suprimir en las posiciones dereferenciables.

Si representáramos las posiciones como enteros, entonces avanzar la posición podría efectuarse con la sencilla operación de enteros $i \leftarrow i + 1$,

pero es deseable pensar en las posiciones como entidades abstractas, no necesariamente enteros y por lo tanto para las cuales no necesariamente es válido hacer operaciones de enteros. Esto lo sugiere la experiencia previa de cursos básicos de programación donde se ha visto que las listas se representan por celdas encadenadas por punteros. En estos casos, puede ser deseable representar a las posiciones como punteros a las celdas. De manera que asumiremos que las posiciones son objetos abstractos. Consideramos entonces las operaciones abstractas:

- Acceder al elemento en la posición p, tanto para modificar el valor ($a_p \leftarrow x$) como para acceder al valor ($x \leftarrow a_p$).
- Avanzar una posición, es decir dada una posición p correspondiente al elemento a_i, retornar la posición q correspondiente al elemento a_{i+1}. (Como mencionamos previamente, no es necesariamente $q = p + 1$, o más aún, pueden no estar definidas estas operaciones aritméticas sobre las posiciones p y q.)
- Retornar la primera posición de la lista, es decir la correspondiente al elemento a_0.
- Retornar la posición ficticia al final de la lista, es decir la correspondiente a n.

2.1.3. Una interfaz simple para listas

Definiremos ahora una interfaz apropiada en C++. Primero observemos que, como las posiciones no serán necesariamente enteros, enmascararemos el concepto de posición en una clase **iterator_t**. El nombre está tomado de las STL, agregándole el sufijo **_t** para hacer énfasis en que es un tipo. En adelante hablaremos indiferentemente de posiciones o iterators. La interfaz puede observarse en el código 2.1.

Primero declaramos la clase **iterator_t** de la cual no damos mayores detalles. Luego la clase **list**, de la cual sólo mostramos algunos de sus métodos públicos.

```
1. class iterator_t { /* . . . */ };
2.
3. class list {
4. private:
5.   // . . .
```

```
6.  public:
7.    // . . .
8.    iterator_t insert(iterator_t p,elem_t x);
9.    iterator_t erase(iterator_t p);
10.   elem_t & retrieve(iterator_t p);
11.   iterator_t next(iterator_t p);
12.   iterator_t begin();
13.   iterator_t end();
14. }
```

Código 2.1: *Interfaz básica para listas. [Archivo: listbas.cpp]*

- **`insert`**: inserta el elemento **`x`** en la posición **`p`**, devolviendo una posición **`q`** al elemento insertado. Todas las posiciones de **`p`** en adelante (incluyendo **`p`**) pasan a ser inválidas, por eso la función devuelve a **`q`**, la nueva posición insertada, ya que la anterior **`p`** es inválida. Es válido insertar en cualquier posición dereferenciable o no dereferenciable, es decir que es válido insertar también en la posición ficticia.

- **`erase`**: elimina el elemento en la posición **`p`**, devolviendo una posición **`q`** al elemento que previamente estaba en la posición siguiente a **`p`**. Todas las posiciones de **`p`** en adelante (incluyendo **`p`**) pasan a ser inválidas. Sólo es válido suprimir en las posiciones dereferenciables de la lista.

- **`retrieve`**: *"recupera"* el elemento en la posición **`p`**, devolviendo una *referencia* al mismo, de manera que es válido hacer tanto **`x = L.retrieve(p)`** como **`L.retrieve(p)=x`**. Se puede aplicar a cualquier posición **`p`** dereferenciable y no modifica a la lista. Notar que retorna una *referencia* al elemento correspondiente de manera que este puede ser cambiado, es decir, puede ser usado como un *"valor asignable"* (*"left hand side value"*).

- **`next`**: dada una posición dereferenciable **`p`**, devuelve la posición del siguiente elemento. Si **`p`** es la última posición dereferenciable, entonces devuelve la posición ficticia. No modifica la lista.

- **`begin`**: devuelve la posición del primer elemento de la lista.

- **`end`**: devuelve la posición ficticia (no dereferenciable), después del final de la lista.

Algunas observaciones con respecto a estas funciones son

- **Posiciones inválidas:** Un elemento a tener en cuenta es que las funciones que modifican la lista como `insert` and `erase`, convierten en inválidas algunas de las posiciones de la lista, normalmente desde el punto de inserción/supresión en adelante, incluyendo `end()`. Por ejemplo,

```
1. iterator_t p,q,r;
2. list L;
3. elem_t x,y,z;
4. //...
5. // p es una posicion dereferenciable
6. q = L.next(p);
7. r = L.end();
8. L.erase(p);
9. x = *p;        // incorrecto
10. y = *q;       // incorrecto
11. L.insert(r,z); // incorrecto
```

  ya que `p,q,r` ya no son válidos (están después de la posición borrada `p`. La forma correcta de escribir el código anterior es

```
1. iterator_t p,q,r;
2. list L;
3. elem_t x,y,z;
4. //...
5. // p es una posicion dereferenciable
6. p = L.erase(p);
7. x = *p;        // correcto
8. q = L.next(p);
9. y = *q;        // correcto
10. r = L.end();
11. L.insert(r,z); // correcto
```

- **Las posiciones sólo se acceden a través de funciones de la clase:** Las únicas operaciones válidas con posiciones son

  ▷ **Asignar:**

```
p = L.begin();
q = L.end();
```

  ▷ **Avanzar:**

```
q = L.next(p);
```

▷ **Acceder al elemento:**

```
x = L.retrieve(p);
L.retrieve(q) = y;
```

▷ **Copiar:**

```
q = p;
```

▷ **Comparar:** Notar que sólo se puede comparar por igualdad o desigualdad, no por operadores de comparación, como < ó >.

```
q == p
r != L.end();
```

2.1.4. Funciones que retornan referencias

A veces es útil escribir funciones que dan acceso a ciertos componentes internos de estructuras complejas, permitiendo cambiar su valor. Supongamos que queremos escribir una función `int min(int *v,int n)` que retorna el mínimo de los valores de un vector de enteros `v` de longitud `n`, pero además queremos dar la posibilidad al usuario de la función de *cambiar el valor interno* correspondiente al mínimo. Una posibilidad es retornar por un argumento adicional el índice `j` correspondiente el mínimo. Posteriormente para modificar el valor podemos hacer `v[j]=`<*nuevo-valor*>. El siguiente fragmento de código modifica el valor del mínimo haciendo que valga el doble de su valor anterior.

```
1. int min(int *v,int n,int *jmin);
2. ...
3.
4. int jmin;
5. int m = min(v,n,&jmin);
6. v[jmin] = 2*v[jmin];
```

Sin embargo, si `min` operara sobre estructuras más complejas sería deseable que retornara directamente un objeto modificable, es decir que pudiéramos hacer

```
1. min(v,n) = 2*min(v,n);
```

```
1. int *min(int *v,int n) {
```

```
2.   int x = v[0];
3.   int jmin = 0;
4.   for (int k=1; k<n; k++) {
5.     if (v[k]<x) {
6.       jmin = k;
7.       x = v[jmin];
8.     }
9.   }
10.  return &v[jmin];
11. }
12.
13. void print(int *v,int n) {
14.   cout << "Vector: (";
15.   for (int j=0; j<n; j++) cout << v[j] << " ";
16.   cout << "), valor minimo: " << *min(v,n) << endl;
17. }
18.
19. int main() {
20.   int v[ ] = {6,5,1,4,2,3};
21.   int n = 6;
22.
23.   print(v,n);
24.   for (int j=0; j<6; j++) {
25.     *min(v,n) = 2* (*min(v,n));
26.     print(v,n);
27.   }
28. }
```

Código 2.2: *Ejemplo de función que retorna un puntero a un elemento interno, de manera de poder modificarlo. [Archivo: ptrexa.cpp]*

Esto es posible de hacer en C, si modificamos `min` de manera que *retorne un puntero* al elemento mínimo. El código 2.2 muestra una posible implementación. A continuación se muestra la salida del prograna.

```
1. [mstorti@spider aedsrc]$ ptrexa
2. Vector: (6 5 1 4 2 3 ), valor minimo: 1
3. Vector: (6 5 2 4 2 3 ), valor minimo: 2
4. Vector: (6 5 4 4 2 3 ), valor minimo: 2
5. Vector: (6 5 4 4 4 3 ), valor minimo: 3
6. Vector: (6 5 4 4 4 6 ), valor minimo: 4
7. Vector: (6 5 8 4 4 6 ), valor minimo: 4
8. Vector: (6 5 8 8 4 6 ), valor minimo: 4
9. [mstorti@spider aedsrc]$
```

```
1.  int &min(int *v,int n) {
2.    int x = v[0];
3.    int jmin = 0;
4.    for (int k=1; k<n; k++) {
5.      if (v[k]<x) {
6.        jmin = k;
7.        x = v[jmin];
8.      }
9.    }
10.   return v[jmin];
11. }
12.
13. void print(int *v,int n) {
14.   cout << "Vector: (";
15.   for (int j=0; j<n; j++) cout << v[j] << " ";
16.   cout << "), valor minimo: " << min(v,n) << endl;
17. }
18.
19. int main() {
20.   int v[ ] = {6,5,1,4,2,3};
21.   int n = 6;
22.
23.   print(v,n);
24.   for (int j=0; j<6; j++) {
25.     min(v,n) = 2*min(v,n);
26.     print(v,n);
27.   }
28. }
```

Código 2.3: *Ejemplo de función que retorna una referencia a un elemento interno, de manera de poder modificarlo. [Archivo: refexa.cpp]*

C++ permite retornar directamente referencias (`int &`, por ejemplo) a los elementos, de manera que no hace falta despues dereferenciarlos como en la línea 25. El mismo program usando referencias puede verse en el código 2.3. El método `val=retrieve(p)` en la interfaz presentada para listas es un ejemplo. Esta técnica es usada frecuentemente en las STL.

2.1.5. Ejemplos de uso de la interfaz básica

```
1.  void purge(list &L) {
```

```
2.   iterator_t p,q;
3.   p = L.begin();
4.   while (p!=L.end()) {
5.     q = L.next(p);
6.     while (q!=L.end()) {
7.       if (L.retrieve(p)==L.retrieve(q)) {
8.         q = L.erase(q);
9.       } else {
10.        q = L.next(q);
11.      }
12.    }
13.    p = L.next(p);
14.  }
15. }
16.
17. int main() {
18.   list L;
19.   const int M=10;
20.   for (int j=0; j<2*M; j++)
21.     L.insert(L.end(),rand()%M);
22.   cout << "Lista antes de purgar: " << endl;
23.   print(L);
24.   cout << "Purga lista. . . " << endl;
25.   purge(L);
26.   cout << "Lista despues de purgar: " << endl;
27.   print(L);
28. }
```

Código 2.4: *Eliminar elementos repetidos de una lista [Archivo: purge.cpp]*

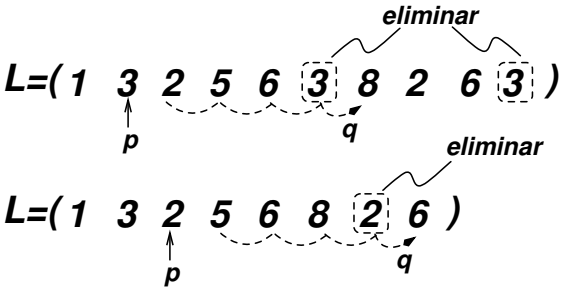


Figura 2.1: Proceso de eliminación de elementos en purge

Ejemplo 2.1: Eliminar elementos duplicados de una lista. Volviendo al problema descripto en §2.1.1 de eliminar los elementos repetidos de una lista, consideremos el código código 2.4. Como describimos previamente, el algoritmo tiene dos lazos anidados. En el lazo exterior una posición **p** recorre todas las posiciones de la lista. En el lazo interior otra posición **q** recorre las posiciones más alla de **p** eliminando los elementos iguales a los que están en **p**.

El código es muy cuidadoso en cuanto a usar siempre posiciones válidas. Por ejemplo la operación **L.retrieve(q)** de la línea 7 está garantizado que no fallará. Asumiendo que **p** es una posición válida dereferenciable a la altura de la línea 5, entonces en esa línea **q** recibe una posición válida, dereferenciable o no. Si la posición asignada a **q** es **end()**, entonces la condición del **while** fallará y **retrieve** no se ejecutará. A su vez, **p** es dereferenciable en la línea 5 ya que en la línea 3 se le asignó una posición válida (**L.begin()** es siempre válida, y también es dereferenciable a menos que la lista este vacía). Pero al llegar a la línea 5 ha pasado el test de la línea precedente, de manera que seguramente es dereferenciable. Luego, **p** sólo es modificada en la línea 13. Notar que a esa altura no es trivial decir si **p** es válida o no, ya que eventualmente pueden haberse hecho operaciones de eliminación en la línea 8. Sin embargo, todas estas operaciones se realizan sobre posiciones que están más allá de **p**, de manera que, efectivamente **p** llega a la línea 13 siendo una posición válida (de hecho dereferenciable). Al avanzar **p** una posición en línea 13 puede llegar a tomar el valor **end()**, pero en ese caso el lazo terminará, ya que fallará la condición de la línea 4.

En el **main()** (líneas 17–28) se realiza una verificación del funcionamiento de **purge()**. Primero se declara una variable **L** de tipo **list** (asumiendo que el tipo elemento es entero, es decir **elem_t=int**). Por supuesto esto involucra todas las tareas de inicialización necesarias, que estarán dentro del constructor de la clase, lo cual se discutirá más adelante en las diferentes implementaciones de **list**. Para un cierto valor **M** se agregan **2*M** elementos generados aleatoriamente entre **0** y **M-1**. (La función **irand(int n)** retorna elementos en forma aleatoria entre 0 y $n-1$). La lista se imprime por consola con la función **print()**, es purgada y luego vuelta a imprimir. Por brevedad, no entraremos aquí en el código de **print()** e **irand()**. Una salida típica es la siguiente

```
1. [mstorti@minerva aedsrc]$ purge
2. Lista antes de purgar:
3. 8 3 7 7 9 1 3 7 2 5 4 6 3 5 9 9 6 7 1 6
4. Purga lista...
```

```
5. Lista despues de purgar:
6. 8 3 7 9 1 2 5 4 6
7. [mstorti@minerva aedsrc]$
```

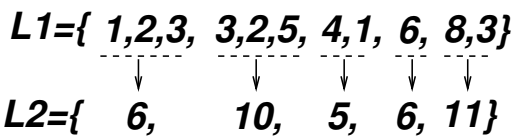


Figura 2.2: Agrupar subsecuencias sumando.

Ejemplo 2.2: *Consigna:* Dadas dos listas de enteros positivos `L1` y `L2` escribir una función `bool check_sum(list &L1, list &L2);` que retorna verdadero si los elementos de `L1` pueden agruparse (sumando secuencias de elementos contiguos) de manera de obtener los elementos de `L2` *sin alterar el orden de los elementos*. Por ejemplo, en el caso de la figura 2.2 `check_sum(L1,L2)` debe retornar *verdadero* ya que los agrupamientos mostrados reducen la lista `L1` a la `L2`.

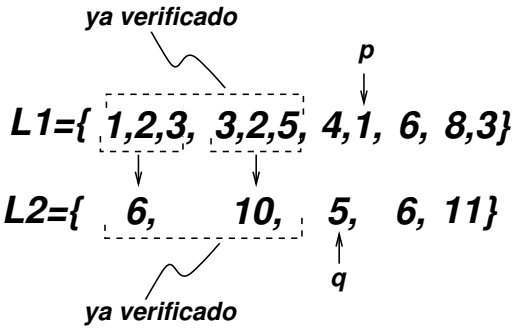


Figura 2.3: Estado parcial en el algoritmo `check-sum`

Solución Proponemos el siguiente algoritmo. Tenemos dos posiciones `p,q` en `L1` y `L2`, respectivamente, y un acumulador `suma` que contiene la suma de un cierto número de elementos de `L1`. En un lazo infinito vamos avanzando los punteros `p,q` de manera que los elementos que están *antes* de ellos verifican las siguientes condiciones (ver figura 2.3):

- Cada elemento de `L2` antes de `p` se corresponde con una serie de elementos de `L1`, sin dejar huecos, salvo eventualmente un resto, al final de `L1`.

- La suma del resto de los elementos de `L1` coincide con el valor de `suma`.

Inicialmente `p,q` están en los comienzos de las listas y `suma=0`, que ciertamente cumple con las condiciones anteriores. Después de una serie de pasos, se puede llegar a un estado como el mostrado en la figura 2.3. Tenemos `(p->1,q->5,suma=4)`.

Para avanzar las posiciones comparamos el valor actual de `suma` con `L.retrieve(q)` y seguimos las siguientes reglas,

1. **Avanza** q**:** Si `suma==L2.retrieve(q)` entonces ya hemos detectado un grupo de `L1` que coincide con un elemento de `L2`. Ponemos `suma` en 0, y avanzamos `q`.
2. **Avanza** p**:** Si `suma<L2.retrieve(q)` entonces podemos avanzar `p` acumulando `L2.retrieve(p)` en `suma`.
3. **Falla:** Si `suma>L2.retrieve(q)` entonces las listas no son compatibles, hay que retornar falso.

Mientras tanto, en todo momento antes de avanzar una posición hay que verificar de mantener la validez de las mismas. El lazo termina cuando alguna de las listas se termina. El programa debe retornar verdadero si al salir del lazo ambas posiciones están al final de sus respectivas listas y `suma==0`.

Partiendo del estado de la figura 2.3, tenemos los siguientes pasos

- `(p->1,q->5,suma=4)`: avanza `p`, `suma=5`
- `(p->6,q->5,suma=5)`: avanza `q`, `suma=0`
- `(p->6,q->6,suma=0)`: avanza `p`, `suma=6`
- `(p->8,q->6,suma=6)`: avanza `q`, `suma=0`
- `(p->8,q->11,suma=0)`: avanza `p`, `suma=8`
- `(p->3,q->11,suma=8)`: avanza `p`, `suma=11`
- `(p->end(),q->11,suma=11)`: avanza `q`, `suma=0`
- `(p->end(),q->end(),suma=0)`: Sale del lazo.

```
1. bool check_sum(list &L1, list &L2) {
2.   iterator_t p,q;
3.   p = L1.begin();
4.   q = L2.begin();
5.   int suma = 0;
6.   while (true) {
7.     if (q==L2.end()) break;
8.     else if (suma==L2.retrieve(q)) {
```

```
9.      suma=0;
10.     q = L2.next(q);
11.   }
12.   else if (p==L1.end()) break;
13.   else if (suma<L2.retrieve(q)) {
14.     suma += L1.retrieve(p);
15.     p = L1.next(p);
16.   }
17.   else return false;
18. }
19. return suma==0 && p==L1.end() && q==L2.end();
20. }
```

Código 2.5: *Verifica que* L2 *proviene de* L1 *sumando elementos consecutivos. [Archivo: check-sum.cpp]*

El código correspondiente se muestra en el código 2.5. Después de inicializar **p,q,suma** se entra en el lazo de las líneas 6–18 donde se avanza **p,q**. Los tres casos listados más arriba coinciden con tres de las entradas en el **if**. Notar que antes de recuperar el elemento en **q** en la línea 8 hemos verificado que la posición es dereferenciable porque pasó el test de la línea precedente. **q** es avanzado en la línea 10 con lo cual uno podría pensar que el **retrieve** que se hace en la línea 13 podría fallar si en ese avance **q** llego a **end()**. Pero esto no puede ser así, ya que si se ejecuta la línea 10, entonces el condicional de la línea 13 sólo puede hacerse en otra ejecución del lazo del **while** ya que ambas líneas están en ramas diferentes del mismo **if**.

2.1.6. Implementación de listas por arreglos

A continuación veremos algunas posibles implementaciones de listas. Probablemente la representación de listas más simple de entender es mediante arreglos. En esta representación los valores son almacenados en celdas contiguas de un arreglo, como se muestra en la figura 2.4. Las posiciones se representan simplemente mediante enteros (recordemos que, en general, esto no es así para otras implementaciones). El principal problema de esta representación es que, para insertar un elemento en una posición intermedia de la lista requiere mover todos los elementos que le suceden una posición hacia el final (ver 2.5). Igualmente, para borrar un elemento hay que desplazar todos los elementos que suceden una posición hacia el comienzo para "rellenar" el hueco dejado por el elemento eliminado.

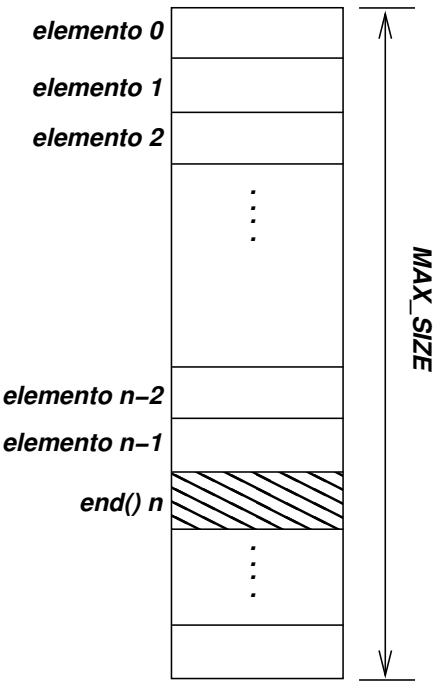


Figura 2.4: Representación de listas mediante arreglos

```
1.  typedef int iterator_t;
2.
3.  class list {
4.  private:
5.    static int MAX_SIZE;
6.    elem_t *elems;
7.    int size;
8.  public:
9.    list();
10.   ~list();
11.   iterator_t insert(iterator_t p,elem_t j);
12.   iterator_t erase(iterator_t p);
13.   iterator_t erase(iterator_t p,iterator_t q);
14.   void clear();
15.   iterator_t begin();
16.   iterator_t end();
17.   iterator_t next(iterator_t p);
18.   iterator_t prev(iterator_t p);
```

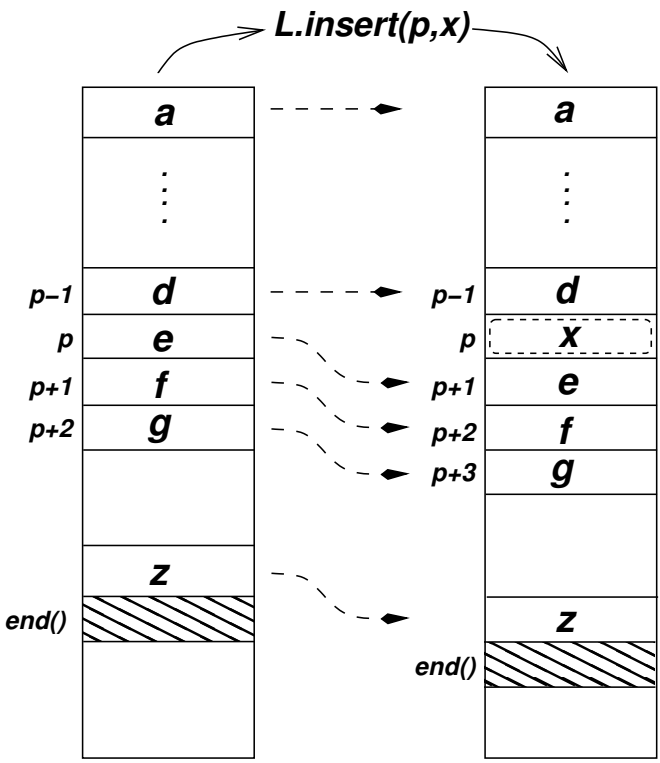


Figura 2.5: Inserción de un elemento en la representación de listas por arreglos.

```
19.     elem_t & retrieve(iterator_t p);
20.   };
```

Código 2.6: *Declaraciones para listas implementadas por arreglos. [Archivo: lista.h]*

```
1.  #include <iostream>
2.  #include <aedsrc/lista.h>
3.  #include <cstdlib>
4.
5.  using namespace std;
6.  using namespace aed;
7.
```

```
8.  int list::MAX_SIZE=100;
9.
10. list::list() : elems(new elem_t[MAX_SIZE]),
11.                size(0) { }
12.
13. list::~list() { delete[] elems; }
14.
15. elem_t &list::retrieve(iterator_t p) {
16.   if (p<0 || p>=size) {
17.     cout << "p: mala posicion.\n";
18.     abort();
19.   }
20.   return elems[p];
21. }
22.
23.
24. iterator_t list::begin() { return 0; }
25.
26. iterator_t list::end() { return size; }
27.
28. iterator_t list::next(iterator_t p) {
29.   if (p<0 || p>=size) {
30.     cout << "p: mala posicion.\n";
31.     abort();
32.   }
33.   return p+1;
34. }
35.
36. iterator_t list::prev(iterator_t p) {
37.   if (p<=0 || p>size) {
38.     cout << "p: mala posicion.\n";
39.     abort();
40.   }
41.   return p-1;
42. }
43.
44. iterator_t list::insert(iterator_t p,elem_t k) {
45.   if (size>=MAX_SIZE) {
46.     cout << "La lista esta llena.\n";
47.     abort();
48.   }
49.   if (p<0 || p>size) {
50.     cout << "Insertando en posicion invalida.\n";
51.     abort();
52.   }
53.   for (int j=size; j>p; j--) elems[j] = elems[j-1];
54.   elems[p] = k;
```

```
55.   size++;
56.   return p;
57. }
58.
59. iterator_t list::erase(iterator_t p) {
60.   if (p<0 || p>=size) {
61.     cout << "p: posicion invalida.\n";
62.     abort();
63.   }
64.   for (int j=p; j<size-1; j++) elems[j] = elems[j+1];
65.   size--;
66.   return p;
67. }
68.
69. iterator_t list::erase(iterator_t p,iterator_t q) {
70.   if (p<0 || p>=size) {
71.     cout << "p: posicion invalida.\n";
72.     abort();
73.   }
74.   if (q<0 || q>size) {
75.     cout << "q: posicion invalida.\n";
76.     abort();
77.   }
78.   if (p>q) {
79.     cout << "p debe estar antes de q\n";
80.     abort();
81.   }
82.   if (p==q) return p;
83.   int shift = q-p;
84.   for (int j=p; j<size-shift; j++)
85.     elems[j] = elems[j+shift];
86.   size -= shift;
87.   return p;
88. }
89.
90. void list::clear() { erase(begin(),end()); }
```

Código 2.7: *Implementación de funciones para listas implementadas por arreglos. [Archivo: lista.cpp]*

En esta implementación, el header de la clase puede ser como se muestra en el código 2.6. Los detalles de la implementación de los diferentes métodos está en el código 2.7. El tipo `iterator_t` es igual al tipo entero (`int`) y por lo tanto, en vez de declarar una clase, hacemos la equivalencia

via un **typedef**. Los únicos campos datos en la clase **list** son un puntero a enteros **elems** y un entero **size** que mantiene la longitud de la lista. Por simplicidad haremos que el vector subyacente **elems** tenga siempre el mismo tamaño. Esta cantidad está guardada en la variable estática de la clase **MAX_SIZE**. Recordemos que cuando una variable es declarada estática dentro de una clase, podemos pensar que en realidad es una constante dentro de la clase, es decir que no hay una copia de ella en cada instancia de la clase (es decir, en cada objeto). Estos miembros de la clase, que son datos, están en la parte privada, ya que forman parte de la implementación de la clase, y no de la interfaz, de manera que un usuario de la clase no debe tener acceso a ellos.

Los métodos públicos de la clase están en las líneas líneas 9–19. Además de los descriptos en la sección §2.1.3 (código 2.1) hemos agregado el constructor y el destructor y algunos métodos que son variantes de **erase()** como el **erase(p,q)** de un rango (línea 13) y **clear()** que equivale a **erase(begin(),end())**, es decir que borra todos los elementos de la lista. También hemos introducido **prev()** que es similar a **next()** pero retorna el antecesor, no el sucesor y un método básico de impresión **print()**.

En la implementación (código 2.7), vemos que el constructor inicializa las variables **size** y aloca el vector **elems** con **new[]**. El destructor desaloca el espacio utilizado con **delete[]**. Notemos que la inicialización se realiza en la *"lista de inicialización"* del constructor. Los métodos **retrieve**, **next** y **prev** son triviales, simplemente retornan el elemento correspondiente del vector o incrementan apropiadamente la posición, usando aritmética de enteros. Notar que se verifica primero que la posición sea válida para la operación correspondiente. En **retrieve()**, después de verificar que la posición es válida para insertar (notar que en el test de la línea 49 da error si **p==size**), el elemento es retornado usando la indexación normal de arreglos a través de **[]**. El método **insert()**, después de verificar la validez de la posición (notar que **p==size** no da error en este caso), corre todos los elementos después de **p** en la línea 53, inserta el elemento, e incrementa el contador **size**. **erase()** es similar.

erase(p,q) verifica primero la validez del rango a eliminar. Ambas posiciones deben ser válidas, incluyendo **end()** y **p** debe preceder a **q** (también pueden ser iguales, en cuyo caso **erase(p,q)** no hace nada). **shift** es el número de elementos a eliminar, y por lo tanto también el desplazamiento que debe aplicarse a cada elemento posterior a **q**. Notar que uno podría pensar en implementar **erase(p,q)** en forma *"genérica"*

```
1. iterator_t list::erase(iterator_t p,iterator_t q) {
2.   while (p!=q) p = erase(p); // Ooops! q puede no ser válido...
3.   return p;
4. }
```

Este código es genérico, ya que en principio sería válido para cualquier implementación de listas que siga la interfaz código 2.1. Sin embargo, hay un error en esta versión: después de ejecutar el primer **`erase(p)`** la posición **`q`** deja de ser válida. Además, en esta implementación con arreglos hay razones de eficiencia para no hacerlo en forma genérica (esto se verá en detalle luego). **`clear()`** simplemente asigna a **`size`** 0, de esta forma es $O(1)$. La alternativa *"genérica"* (**`erase(begin(),end())`**), sería $O(n)$.

Otro ejemplo de código genérico es **`purge`**. Por supuesto, la mayoría de las funciones sobre listas que no pertenecen a la clase, son en principio genéricas, ya que sólo acceden a la clase a través de la interfaz pública y, por lo tanto, pueden usar cualquier otra implementación. Sin embargo, el término genérico se aplica preferentemente a operaciones bien definidas, de utilidad general, como **`purge()`** o **`sort()`** (que ordena los elementos de menor a mayor). Estas funciones, a veces son candidatas a pertenecer a la clase. **`print()`** es genérica y podríamos copiar su código tal cual e insertarlo en cualquier otra implementación de listas. Pero todavía sería mejor evitar esta duplicación de código, usando la noción de polimorfismo y herencia de clases.

2.1.6.1. Eficiencia de la implementación por arreglos

La implementación de listas por arreglos tiene varias desventajas. Una es la rigidez del almacenamiento. Si en algún momento se insertan más de **`MAX_SIZE`** elementos, se produce un error y el programa se detiene. Esto puede remediarse realocando el arreglo **`elems`**, copiando los elementos en el nuevo arreglo y liberando el anterior. Sin embargo, estas operaciones pueden tener un impacto en el tiempo de ejecución si la realocación se hace muy seguido.

Pero el principal inconveniente de esta implementación se refiere a los tiempos de ejecución de **`insert(p,x)`** y **`erase(p)`**. Ambos requieren un número de instrucciones que es proporcional al número de ejecuciones de los lazos correspondientes, es decir, proporcional al número de elementos que deben moverse. El mejor caso es cuando se inserta un elemento en **`end()`** o se elimina el último elemento de la lista. En este caso el lazo no se ejecuta ninguna vez. El peor caso es cuando se inserta o elimina en la

posición **begin()**. En ese caso ambas operaciones son $O(n)$, donde n es el número de elementos en la lista. El caso promedio, depende de la probabilidad P_j de que al elemento a eliminar esté en la **j**

$$T_{\text{prom}}(n) = \sum_{j=0}^{n-1} P_j T(j) \tag{2.5}$$

Si asumimos que la probabilidad del elemento a eliminar es la misma para todos los elementos ($P_j = 1/n$), y tenemos en cuenta que $T(j) = n - j - 1$ entonces el tiempo promedio está dado por

$$\begin{aligned} T_{\text{prom}}(n) &= \frac{1}{n} \sum_{j=0}^{n-1} n - j - 1 \\ &= \frac{1}{n} \left((n-1) + (n-2) + \cdots + 1 + 0 \right) \\ &= \frac{1}{n} \frac{(n-1)n}{2} = \frac{n-1}{2} \approx \frac{n}{2} = O(n) \end{aligned} \tag{2.6}$$

Como se espera que **insert(p,x)** y **erase(p)** sean dos de las rutinas más usadas sobre listas, es deseable encontrar otras representaciones que disminuyan estos tiempos, idealmente a $O(1)$.

Los tiempos de ejecución de las otras rutinas es $O(1)$ (salvo **erase(p,q)** y **print()**).

2.1.7. Implementación mediante celdas enlazadas por punteros

```
1.  class cell;
2.  typedef cell *iterator_t;
3.
4.  class list {
5.  private:
6.    cell *first, *last;
7.  public:
8.    list();
9.    ~list();
10.   iterator_t insert(iterator_t p,elem_t j);
11.   iterator_t erase(iterator_t p);
12.   iterator_t erase(iterator_t p,iterator_t q);
13.   void clear();
14.   iterator_t begin();
```

```
15.     iterator_t end();
16.     void print();
17.     void printd();
18.     iterator_t next(iterator_t p);
19.     iterator_t prev(iterator_t p);
20.     elem_t & retrieve(iterator_t p);
21.     int size();
22.   };
23.
24.   class cell {
25.     friend class list;
26.     elem_t elem;
27.     cell *next;
28.     cell() : next(NULL) {}
29.   };
```

Código 2.8: *Implementación de listas mediante celdas enlazadas por punteros. Declaraciones. [Archivo: listp.h]*

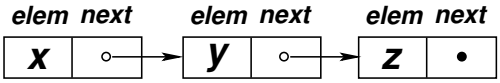


Figura 2.6: Celdas enlazadas por punteros

La implementación mediante celdas enlazadas por punteros es probablemente la más conocida y más usada. Una posible interfaz puede verse en código 2.8. La lista está compuesta de una serie de celdas de tipo `cell` que constan de un campo `elem` de tipo `elem_t` y un campo `next` de tipo `cell *`. Las celdas se van encadenando unas a otras por el campo `next` (ver figura 2.6). Teniendo un puntero a una de las celdas, es fácil seguir la cadena de enlaces y recorrer todas las celdas siguientes. El fin de la lista se detecta manteniendo en el campo `next` de la última celda un puntero nulo (`NULL`). Está garantizado que `NULL` es un puntero inválido (`new` o `malloc()` no retornarán nunca `NULL` a menos que fallen). (En el gráfico representamos al `NULL` por un pequeño círculo lleno.)

Notemos que es imposible recorrer la celda en el sentido contrario. Por ejemplo, si tenemos un puntero a la celda que contiene a `z`, entonces no es posible saber cual es la celda cuyo campo `next` apunta a ella, es decir, la celda que contiene `y`. Las celdas normalmente son alocada con `new` y

liberadas con **delete**, es decir están en el área de almacenamiento dinámico del programa (el *"free store"* o *"heap"*). Esto hace que se deba tener especial cuidado en liberar la memoria alocada, de lo contrario se producen pérdidas de memoria (*"memory leaks"*).

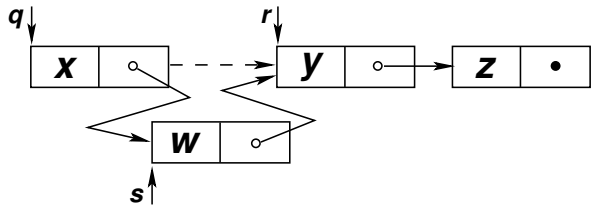


Figura 2.7: Operaciones de enlace necesarias para insertar un nuevo elemento en la lista.

2.1.7.1. El tipo posición

Debemos definir ahora que tipo será, en esta implementación, una posición, es decir el tipo **iterator_t**. La elección natural parece ser **cell ***, ya que si tenemos un puntero a la celda tenemos acceso al contenido. Es decir, parece natural elegir como posición, el puntero a la celda que contiene el dato. Sin embargo, consideremos el proceso de insertar un elemento en la lista (ver figura 2.7). Originalmente tenemos los elementos **L=(...,x,y,z,..)** y queremos insertar un elemento **w** en la posición de **y** es decir **L=(...,x,w,y,z,..)**. Sean **q** y **r** los punteros a las celdas que contienen a **x** e **y**. Las operaciones a realizar son

```
1.  s = new cell;
2.  s->elem = w;
3.  s->next = r;
4.  q->next = s;
```

Notar que para realizar las operaciones necesitamos el puntero **q** a la celda anterior. Es decir, si definimos como posición el puntero a la celda que contiene el dato, entonces la posición correspondiente a **y** es **r**, y para insertar a **w** en la posición de **y** debemos hacer **L.insert(r,w)**. Pero entonces **insert(...)** no podrá hacer las operaciones indicadas arriba ya que no hay forma de conseguir el puntero a la posición anterior **q**. La solución es *definir como posición el puntero a la celda anterior a la que contiene el dato'*. De esta forma, la posición que corresponde a **y** es **q** (antes de insertar **w**)

y está claro que podemos realizar las operaciones de enlace. Decimos que las posiciones están *"adelantadas"* con respecto a los elementos. Notar que después de la inserción la posición de `y` pasa a ser el puntero a la nueva celda `s`, esto ilustra el concepto de que después de insertar un elemento las posiciones posteriores a la de inserción (inclusive) son inválidas.

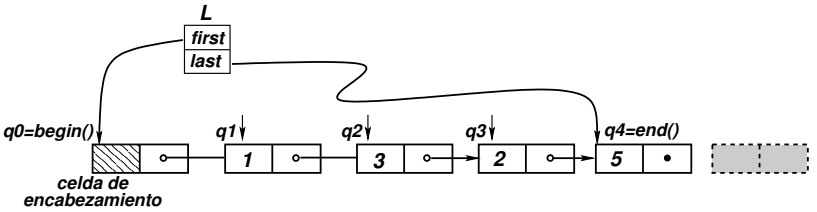


Figura 2.8: Lista enlazada por punteros.

2.1.7.2. Celda de encabezamiento

La lista en sí puede estar representada por un campo `cell *first` que es un puntero a la primera celda. Recordemos que una vez que tenemos un puntero a la primera celda podemos recorrer todas las siguientes. Pero el hecho de introducir un adelanto en las posiciones trae aparejado un problema. ¿Cuál es la posición del primer elemento de la lista? ¿A qué apunta `first` cuando la celda esta vacía? Estos problemas se resuelven si introducimos una *"celda de encabezamiento"*, es decir una celda que no contiene dato y tal que `first` apunta a ella. Entonces, por ejemplo, si nuestra lista contiene a los elementos `L=(1,3,2,5)`, la representación por punteros sería como se muestra en la figura 2.8. Si `q0-q4` son punteros a las 5 celdas de la lista (incluyendo la de encabezamiento), entonces el elemento `1` está en la posición `q0` de manera que, por ejemplo

- `L.retrieve(q0)` retornará 1.
- `L.retrieve(q1)` retornará 3.
- `L.retrieve(q2)` retornará 2.
- `L.retrieve(q3)` retornará 5.
- `L.retrieve(q4)` dará error ya que corresponde a la posición de la celda ficticia (representada en línea de trazos en la figura).

```
1. list::list() : first(new cell), last(first) {
2.   first->next = NULL;
```

```
3.  }
4.
5.  list::~list() { clear(); delete first; }
6.
7.  elem_t &list::retrieve(iterator_t p) {
8.    return p->next->elem;
9.  }
10.
11. iterator_t list::next(iterator_t p) {
12.   return p->next;
13. }
14.
15. iterator_t list::prev(iterator_t p) {
16.   iterator_t q = first;
17.   while (q->next != p) q = q->next;
18.   return q;
19. }
20.
21. iterator_t
22. list::insert(iterator_t p,elem_t k) {
23.   iterator_t q = p->next;
24.   iterator_t c = new cell;
25.   p->next = c;
26.   c->next = q;
27.   c->elem = k;
28.   if (q==NULL) last = c;
29.   return p;
30. }
31.
32. iterator_t list::begin() { return first; }
33.
34. iterator_t list::end() { return last; }
35.
36. iterator_t list::erase(iterator_t p) {
37.   if (p->next==last) last = p;
38.   iterator_t q = p->next;
39.   p->next = q->next;
40.   delete q;
41.   return p;
42. }
43.
44. iterator_t list::erase(iterator_t p,iterator_t q) {
45.   if (p==q) return p;
46.   iterator_t s, r = p->next;
47.   p->next = q->next;
48.   if (!p->next) last = p;
```

```
49.   while (r!=q->next) {
50.     s = r->next;
51.     delete r;
52.     r = s;
53.   }
54.   return p;
55. }
56.
57. void list::clear() { erase(begin(),end()); }
58.
59. void list::print() {
60.   iterator_t p = begin();
61.   while (p!=end()) {
62.     cout << retrieve(p) << " ";
63.     p = next(p);
64.   }
65.   cout << endl;
66. }
67.
68. void list::printd() {
69.   cout << "h(" << first << ")" << endl;
70.   iterator_t c = first->next;
71.   int j=0;
72.   while (c!=NULL) {
73.     cout << j++ << "(" << c << ") :" << c->elem << endl;
74.     c = c->next;
75.   }
76. }
77.
78. int list::size() {
79.   int sz = 0;
80.   iterator_t p = begin();
81.   while (p!=end()) {
82.     sz++;
83.     p = next(p);
84.   }
85.   return sz;
86. }
```

Código 2.9: *Implementación de listas mediante celdas enlazadas por punteros. Implementación de los métodos de la clase. [Archivo: listp.cpp]*

2.1.7.3. Las posiciones `begin()` y `end()`

`begin()` es la posición correspondiente al primer elemento, por lo tanto un puntero a la celda anterior, es decir la celda de encabezamiento. Por otra parte, **`end()`** es una posición ficticia *después* del último elemento (marcada con línea de trazos en la figura). Su posición es un puntero a la celda anterior, es decir la celda que contiene el último elemento (**`q4`** en la figura).

Notar que **`begin()`** no cambia nunca durante la vida de la lista ya que inserciones o supresiones, incluso en el comienzo de la lista no modifican la celda de encabezamiento. Por otra parte **`end()`** sí cambia cuando hay una inserción o supresión al final de la lista. Esto significa que al momento de implementar **`end()`** debemos comenzar desde **`begin()`** y recorrer toda los enlaces hasta llegar a la última celda. Por ejemplo:

```
1.  iterator_t list::end() {
2.    cell *q = first;
3.    while (q->next) q = q->next;
4.    return q;
5.  }
```

Pero el tiempo de ejecución de esta operación es $O(n)$, y **`end()`** es usada frecuentemente en los lazos para detectar el fin de la lista (ver por ejemplo los códigos 2.4 y 2.5), con lo cual *debe* tener costo $O(1)$.

La solución es mantener en la declaración de la lista un puntero a la última celda, actualizándolo convenientemente cuando es cambiado (esto sólo puede ocurrir en **`insert()`** y **`erase()`**).

2.1.7.4. Detalles de implementación

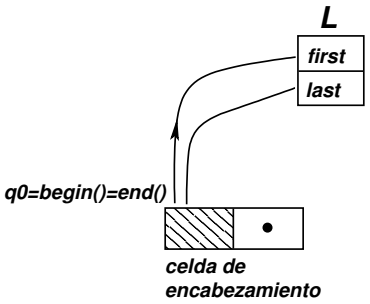


Figura 2.9: Una lista vacía.

- El constructor aloca la celda de encabezamiento y asigna su puntero a **first**. Inicialmente la celda está vacía y por lo tanto **last=first** (**begin()=end()**). También debe inicializar el terminador (de la única celda) a **NULL**.
- El destructor llama a **clear()** (que está implementada en términos de **erase(p,q)**, la veremos después) y finalmente libera la celda de encabezamiento.
- **retrieve(p)** retorna el elemento en el campo **elem**, teniendo en cuenta previamente el adelanto en las posiciones.
- **next()** simplemente avanza un posición usando el campo **next**. Notar que no hay colisión entre la función **next()** y el campo **next** ya que *pertenecen a clases diferentes* (el campo **next** pertenece a la clase **cell**).
- **begin()** y **end()** retornan los campos **first** y **last**, respectivamente.

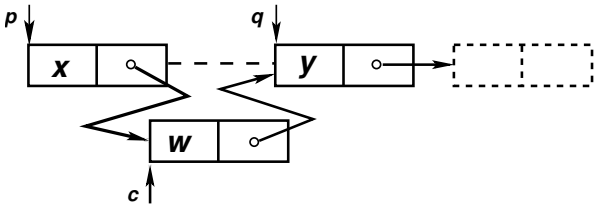


Figura 2.10: Operaciones de punteros para `insert`

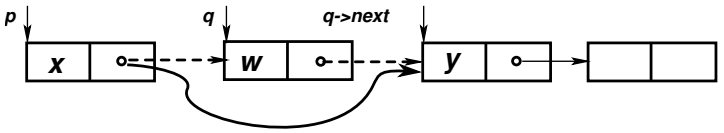


Figura 2.11: Operaciones de punteros para `erase`

- **insert()** y **erase()** realizan los enlaces ya mencionados en §2.1.7.1 (ver figura 2.10) y actualizan, de ser necesario, **last**.
- **erase(p,q)** es implementado haciendo una operación de enlaze de punteros descripta en la figura 2.12. Luego es necesario liberar todas las celdas que están en el rango a eliminar. Recordar que **erase(p,q)** debe eliminar las celdas desde **p** hasta **q**, excluyendo a **q**. Como w

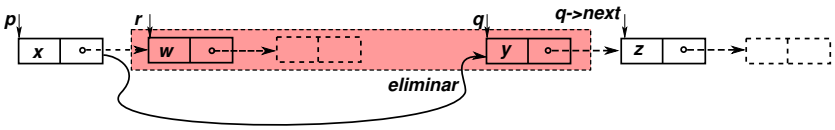


Figura 2.12: Operaciones de punteros para `erase(p,q)`

está en la posición **p** y z en la posición **q**, esto quiere decir que hay que eliminar las celdas que contienen a los elementos w a y.

- **prev()** es implementada mediante un lazo, se recorre la lista desde el comienzo hasta encontrar la celda anterior a la de la posición **p**. Esto es $O(n)$ y por lo tanto debe ser *evitada en lo posible*. (Si **prev()** debe ser usada frecuentemente, entonces puede considerarse en usar una lista *doblemente enlazada*).

2.1.8. Implementación mediante celdas enlazadas por cursores

En la implementación de celdas enlazadas por punteros, los datos son guardados en celdas que son alocadas dinámicamente en el área de almacenamiento dinámico del programa. Una implementación similar consiste en usar celdas enlazadas por cursores, es decir celdas indexadas por punteros dentro de un gran arreglo de celdas. Este arreglo de celdas puede ser una variable global o un objeto estático de la clase, de manera que muchas listas pueden convivir en el mismo espacio de celdas. Puede verse que esta implementación es equivalente a la de punteros (más adelante daremos una tabla que permite *"traducir"* las operaciones con punteros a operaciones con celdas y viceversa) y tiene ventajas y desventajas con respecto a aquella. Entre las ventajas tenemos que,

- La gestión de celdas puede llegar a ser más eficiente que la del sistema (en tiempo y memoria).

- Cuando se alocan dinámicamente con **new** y **delete** muchos objetos pequeños (como las celdas) el área de la memoria ocupada queda muy fragmentada. Esto impide la alocación de objetos grandes , incluso después de eliminar gran parte de estos objetos pequeños. Usando cursores, todas las celdas viven en un gran espacio de celdas, de manera que no se mezclan con el resto de los objetos del programa.

- En lenguajes donde no existe la alocación dinámica de memoria, el uso de cursores reemplaza al de punteros.

Esto puede ser de interés sobre todo para el manejo de grandes cantidades de celdas relativamente pequeñas. Además, el uso de cursores es interesante en si mismo, independientemente de las ventajas o desventajas, ya que permite entender mejor el funcionamiento de los punteros.

Entre las desventajas que tienen los cursores podemos citar que,

- Hay que reservar de entrada un gran espacio de celdas. Si este espacio es pequeño, corremos riesgo de que el espacio se llene y el programa aborte por la imposibilidad de alocar nuevas celdas dentro del espacio. Si es muy grande, estaremos alocando memoria que en la práctica no será usada. (Esto se puede resolver parcialmente, realocando el espacio de celdas, de manera que pueda crecer o reducirse.)
- Listas de elementos del mismo tipo comparten el mismo espacio de celdas, pero si son de diferentes tipos se debe generar un espacio celdas por cada tipo. Esto puede agravar más aún las desventajas mencionadas en el punto anterior.

```
1.  class list;
2.  typedef int iterator_t;
3.
4.  class cell {
5.    friend class list;
6.    elem_t elem;
7.    iterator_t next;
8.    cell();
9.  };
10.
11. class list {
12. private:
13.   friend class cell;
14.   static iterator_t NULL_CELL;
15.   static int CELL_SPACE_SIZE;
16.   static cell *cell_space;
17.   static iterator_t top_free_cell;
18.   iterator_t new_cell();
19.   void delete_cell(iterator_t c);
20.   iterator_t first, last;
21.   void cell_space_init();
```

Código 2.10: *Implementación de listas por cursores. Declaraciones. [Archivo: listc.h]*

Un posible juego de declaraciones puede observarse en el código 2.10. Las celdas son como en el caso de los punteros, pero ahora el campo **next** es de tipo entero, así como las posiciones (**iterator_t**). Las celdas viven en el arreglo **cell_space**, que es un arreglo estándar de elementos de tipo **cell**. Este arreglo podría declararse global (es decir fuera de la clase), pero es más prolijo incluirlo en la clase. Sin embargo, para evitar que cada lista tenga su espacio de celdas, lo declaramos **static**, de esta forma actúa como si fuera global, pero dentro de la clase. También declaramos **static** el tamaño del arreglo **CELL_SPACE_SIZE**. Así como con punteros existe el puntero inválido **NULL**, declaramos un cursor inválido **NULL_CELL**. Como nuestras celdas están indexadas dentro de un arreglo estándar de C, los índices pueden ir entre 0 y **CELL_SPACE_SIZE-1**, de manera que podemos elegir **NULL_CELL** como -1.

2.1.8.1. Cómo conviven varias celdas en un mismo espacio

Las listas consistirán entonces en una serie de celdas dentro del arreglo, con una celda de encabezamiento y terminadas por una celda cuyo campo **next** posee el cursor inválido **NULL_CELL**. Por ejemplo, en la figura 2.13 vemos una situación típica, el espacio de celdas **cell_space** tiene **CELL_SPACE_SIZE**=12 celdas. En ese espacio conviven 2 listas **L1=**(6,9,5,3) y **L2=**(2,5). La lista **L1** ocupa 5 celdas incluyendo la de encabezamiento que en este caso es la celda 2. Como el dato en las celdas de encabezamiento es irrelevante ponemos un "*". El campo **next** de la celda de encabezamiento apunta a la primera celda que en este caso es la 5. La celda 5 contiene el primer elemento (que es un 6) en el campo **elem** y el campo **next** apunta a la siguiente celda (que es la 11). Los enlaces para las celdas de la lista **L1** se muestran con flechas a la derecha del arreglo. La última celda (la 8) contiene en el campo **next** el cursor inválido **NULL_CELL**, representado en el dibujo por un pequeño círculo negro. La lista **L2** contiene 3 celdas, incluyendo la de encabezamiento, a saber las celdas 9, 1 y 10. Las 4 celdas restantes (7,0,4 y 3) están libres.

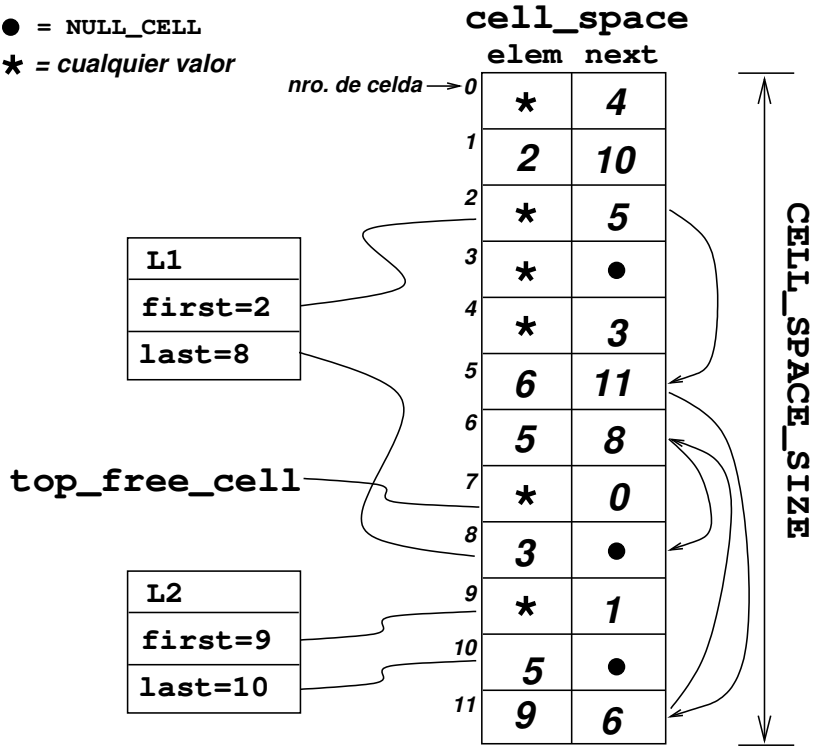


Figura 2.13: Lista enlazada por cursores.

2.1.8.2. Gestión de celdas

Debemos generar un sistema de gestión de celdas, similar a como los operadores **new** y **delete** operan sobre el *heap*. Primero debemos mantener una lista de cuales celdas están alocadas y cuales no. Para esto construimos una lista de celdas libres enlazadas mediante el campo **next** ya que para las celdas libres este campo no es utilizado de todas formas. El cursor **top_free_cell** (que también es estático) apunta a la primera celda libre. El campo **next** de la i-ésima celda libre apunta a la $i + 1$-ésima celda libre, mientras que la última celda libre contiene un cursor inválido **NULL_CELL** en el campo **next**. Esta disposición es completamente equivalente a la de las listas normales, sólo que no hace falta un cursor a la última celda ni necesita el concepto de posición, sino que sólo se accede a través de uno de los extremos de la lista, apuntado por **top_free_cell**. (En realidad se trata de una *"pila"*, pero esto lo veremos en una sección posterior).

Las rutinas **c=new_cell()** y **delete_cell(c)** definen una interfaz abstracta (dentro de la clase **list**) para la gestión de celdas, la primera devuelve una nueva celda libre y la segunda libera una celda utilizada, en forma equivalente a los operadores **new** y **delete** para las celdas enlazadas por punteros.

Antes de hacer cualquier operación sobre una lista, debemos asegurarnos que el espacio de celdas esté correctamente inicializado. Esto se hace dentro de la función **cell_space_init()**, la cual aloca el espacio celdas e inserta todas las celdas en la lista de celdas libres. Esto se realiza en el lazo de las líneas 16–17 en el cual se enlaza la celda **i** con la **i+1**, mientras que en la última celda **CELL_SPACE_SIZE-1** se inserta el cursor inválido.

Para que **cell_space_init()** sea llamado automáticamente antes de cualquier operación con las listas, lo incluimos en el constructor de la clase lista (línea 9). Podemos verificar si el espacio ya fue inicializado por el valor del puntero **cell_space** ya que si el espacio no fue inicializado entonces este puntero es nulo (ver línea 3). El **if** de la línea 9 hace esta verificación.

2.1.8.3. Analogía entre punteros y cursores

	Punteros	**Cursores**
Area de almacenamiento	*heap*	`cell_space`
Tipo usado para las direcciones de las celdas (`iterator_t`)	`cell* c`	`int c`
Dereferenciación de direcciones (dirección → celda)	`*c`	`cell_space[c]`
Dato de una celda dada su dirección `c`	`c->elem`	`cell_space[c].elem`
Enlace de una celda (campo `next`) dada su dirección `c`	`c->next`	`cell_space[c].next`
Alocar una celda	`c = new cell;`	`c = new_cell();`
Liberar una celda	`delete cell;`	`delete_cell(c);`
Dirección inválida	`NULL`	`NULL_CELL`

Tabla 2.1: Tabla de equivalencia entre punteros y cursores.

Hemos mencionado que hay un gran parecido entre la implementación con punteros y cursores. De hecho casi todos los métodos de la clase se

pueden implementar para cursores simplemente aplicando a los métodos de la implementación por punteros (código 2.9) las transformaciones listadas en la tabla 2.1). En la tabla se usa el nombre genérico de *"direcciones"* a los punteros o cursores.

```
1.  cell::cell() : next(list::NULL_CELL) {}
2.
3.  cell *list::cell_space = NULL;
4.  int list::CELL_SPACE_SIZE = 100;
5.  iterator_t list::NULL_CELL = -1;
6.  iterator_t list::top_free_cell = list::NULL_CELL;
7.
8.  list::list() {
9.    if (!cell_space) cell_space_init();
10.   first = last = new_cell();
11.   cell_space[first].next = NULL_CELL;
12. }
13.
14. void list::cell_space_init() {
15.   cell_space = new cell[CELL_SPACE_SIZE];
16.   for (int j=0; j<CELL_SPACE_SIZE-1; j++)
17.     cell_space[j].next = j+1;
18.   cell_space[CELL_SPACE_SIZE-1].next = NULL_CELL;
19.   top_free_cell = 0;
20. }
21.
22. iterator_t list::new_cell() {
23.   iterator_t top = top_free_cell;
24.   if (top==NULL_CELL) {
25.     cout << "No hay mas celdas \n";
26.     abort();
27.   }
28.   top_free_cell = cell_space[top_free_cell].next;
29.   return top;
30. }
31.
32. void list::delete_cell(iterator_t c) {
33.   cell_space[c].next = top_free_cell;
34.   top_free_cell = c;
35. }
36.
37. list::~list() { clear(); }
38.
39. elem_t &list::retrieve(iterator_t p) {
```

```
40.   iterator_t q= cell_space[p].next;
41.   return cell_space[q].elem;
42. }
43.
44. iterator_t list::next(iterator_t p) {
45.   return cell_space[p].next;
46. }
47.
48. iterator_t list::prev(iterator_t p) {
49.   iterator_t q = first;
50.   while (cell_space[q].next != p)
51.     q = cell_space[q].next;
52.   return q;
53. }
54.
55. iterator_t list::insert(iterator_t p,elem_t k) {
56.   iterator_t q = cell_space[p].next;
57.   iterator_t c = new_cell();
58.   cell_space[p].next = c;
59.   cell_space[c].next = q;
60.   cell_space[c].elem = k;
61.   if (q==NULL_CELL) last = c;
62.   return p;
63. }
64.
65. iterator_t list::begin() { return first; }
66.
67. iterator_t list::end() { return last; }
68.
69. iterator_t list::erase(iterator_t p) {
70.   if (cell_space[p].next == last) last = p;
71.   iterator_t q = cell_space[p].next;
72.   cell_space[p].next = cell_space[q].next;
73.   delete_cell(q);
74.   return p;
75. }
76.
77. iterator_t list::erase(iterator_t p,iterator_t q) {
78.   if (p==q) return p;
79.   iterator_t s, r = cell_space[p].next;
80.   cell_space[p].next = cell_space[q].next;
81.   if (cell_space[p].next == NULL_CELL) last = p;
82.   while (r!=cell_space[q].next) {
83.     s = cell_space[r].next;
```

```
84.     delete_cell(r);
85.     r = s;
86.   }
87.   return p;
88. }
89.
90. void list::clear() { erase(begin(),end()); }
91.
92. void list::print() {
93.   iterator_t p = begin();
94.   while (p!=end()) {
95.     cout << retrieve(p) << " ";
96.     p = next(p);
97.   }
98.   cout << endl;
99. }
100.
101. void list::printd() {
102.   cout << "h(" << first << ")" << endl;
103.   iterator_t c = cell_space[first].next;
104.   int j=0;
105.   while (c!=NULL_CELL) {
106.     cout << j++ << "(" << c << ") :" << cell_space[c].elem << endl;
107.     c = next(c);
108.   }
109. }
```

Código 2.11: *Implementación de listas por cursores. Implementación de los métodos. [Archivo: listc.cpp]*

Por ejemplo, el método `next()` en la implementación por punteros simplemente retorna `p->next`. Según la tabla, esto se traduce en retornar `cell_space[c].next` que es lo que precisamente hace el método correspondiente en la implementación por cursores.

2.1.9. Tiempos de ejecución de los métodos en las diferentes implementaciones.

Consideremos ahora los tiempos de ejecución de las operaciones del TAD lista en sus diferentes implementaciones. Los tiempos de implementación de la implementación por punteros y cursores son los mismos, ya que

Método	Arreglos	Punteros / cursores
`insert(p,x)`, `erase(p)`	$O(n)$ $[T = c(n-j)]$	$O(1)$
`erase(p,q)`	$O(n)$ $[T = c(n-k)]$	$O(n)$ $[T = c(k-j)]$
`clear()`	$O(1)$	$O(n)$
`begin()`, `end()`, `next()`, `retrieve()`	$O(1)$	$O(1)$
`prev(p)`	$O(1)$	$O(n)$ $[T = cj]$

Tabla 2.2: Tiempos de ejecución de los métodos del TAD lista en las diferentes implementaciones. j, k son las posiciones enteras correspondientes a `p,q`

sólo difieren en cómo acceden a las celdas, pero de todas formas las operaciones involucradas son $O(1)$, en ambos casos, de manera que la comparación es entre punteros/cursores y arreglos. Las operaciones **`begin()`**, **`end()`**, **`next()`**, **`retrieve()`** son $O(1)$ en ambos casos. La diferencia más importante es, como ya hemos mencionado, en las operaciones **`insert(p,x)`** y **`erase(p)`**. En la implementación por arreglos se debe mover todos los elementos que están después de la posición **`p`** (los lazos de las líneas 64 y 53, código 2.7), o sea que es $O(n-j)$ donde j es la posición (como número entero) de la posición abstracta **`p`**, mientras que para punteros/cursores es $O(1)$. **`erase(p,q)`** debe hacer el **`delete`** (o **`delete_cell()`** de todas las celdas en el rango **`[p,q)`** de manera que requiere $O(k-j)$ operaciones, donde k, j son las posiciones enteras correspondientes a **`p,q`**. Por otra parte, en la implementación por arreglos sólo debe moverse los elementos en el rango **`[q,end())`** (esto es $n-k$ elementos) $k-j$ posiciones hacia el comienzo. Pero el mover cada elemento en un arreglo es tiempo constante, independientemente de cuantas posiciones se mueve, de manera que la operación es $O(n-k)$. En el límite, la función **`clear()`** es $O(n)$ para punteros/cursores y $O(1)$ para arreglos. Por otra parte, la función **`prev(p)`** es $O(j)$ para punteros/cursores, ya que involucra ir al comienzo de la lista y recorrer todas las celdas hasta encontrar la **`p`** (los lazos de las líneas línea 17 en el código 2.9 y la línea 50 en el código 2.11). Esto involucra un tiempo $O(j)$, mientras que en el caso de la implementación por arreglos debe retornar **`p-1`** ya que las posiciones son enteras, y por lo tanto es $O(1)$. Los tiempos de ejecución están listados en la Tabla 2.2. (*Nota: Para* `insert` *por arreglos se indica en*

la tabla $O(n)$, que corresponde al peor caso que es cuando p está al principio de la lista y entre corchetes se indica $T = c(n - j)$ que es el número de operaciones dependiendo de j. La constante c es el tiempo promedio para hacer una de las operaciones. Lo mismo ocurre para otras funciones.)

Comparando globalmente las implementaciones, vemos que la implementación por arreglos es más competitiva en **`prev()`**, **`clear()`**. Pero **`prev()`** decididamente es una operación para la cual no están diseñadas las listas simplemente enlazadas y normalmente **`clear()`** debería ser menos usada que **`insert()`** o **`erase()`**, por lo cual raramente se usan arreglos para representar listas. Por otra parte la diferencia para **`erase(p,q)`** puede ser a favor de punteros/cursores (cuando se borran pequeños intervalos en la mitad de la lista) o a favor de los arreglos (cuando se borran grandes regiones cerca del final).

2.1.10. Interfaz STL

2.1.10.1. Ventajas de la interfaz STL

Uno de los principales inconvenientes de la interfaz definida hasta ahora (ver código 2.1) es que asocia el tipo lista a una lista de un tipo de elemento dado. Es decir, normalmente un juego de declaraciones como el citado debe ser precedido de una serie de asignaciones de tipo, como por ejemplo

```
1. typedef elem_t int;
```

si se quieren manipular listas de enteros. Por otra parte, si se desean manipular listas de dos tipos diferentes, por ejemplo enteros y dobles, entonces se debe duplicar el código (las declaraciones y las implementaciones) definiendo un juego de tipos para cada tipo de dato, por ejemplo **`list_int`** y **`iterator_int_t`** para enteros y **`list_double`** y **`iterator_double_t`**. Pero esto, por supuesto, no es deseable ya que lleva a una duplicación de código completamente innecesaria.

La forma correcta de evitar esto en C++ es mediante el uso de *"templates"*. Los templates permiten definir clases o funciones parametrizadas por un tipo (u otros objetos también). Por ejemplo, podemos definir la clase **`list<class T>`** de manera que luego el usuario puede declarar simplemente

```
1. list<int> lista_1;
2. list<double> lista_2;
```

Esta es la forma en que los diferentes contenedores están declarados en las STL.

En esta sección veremos como generalizar nuestras clases con el uso de templates, es más modificaremos nuestra interfaz de manera que sea totalmente *"compatible con las STL"*, es decir que si un código funciona llamando a nuestros contenedores, también funciona con el contenedor correspondiente de las STL. Una diferencia puramente sintáctica con respecto a las STL es el uso de la sobrecarga de operadores para las funciones **`next()`** y **`prev()`**. Notemos que, si las posiciones fueran enteros, como en el caso de los arreglos, entonces podríamos hacer los reemplazos

```
p = next(p);   →  p++
p = prev(p);   →  p--
```

Las STL extienden el alcance de los operadores **`++`** y **`--`** para los objetos de tipo posición usando sobrecarga de operadores y lo mismo haremos en nuestra implementación. También el operador **`*p`** que permite dereferenciar punteros será usado para recuperar el dato asociado con la posición, como lo hace la función **`retrieve(p)`** en la interfaz básica código 2.1.

Finalmente, otra diferencia sintáctica entre las STL y nuestra versión básica es la clase de posiciones. Si usamos templates, deberemos tener un template separado para la clase de iteradores, por ejemplo **`iterator<int>`**. Pero cuando tengamos otros contenedores como conjuntos (**`set`**) y correspondencias (**`map`**), cada uno tendrá su clase de posiciones y para evitar la colisión, podemos agregarle el tipo de contenedor al nombre de la clase, por ejemplo, **`list_iterator<int>`**, **`set_iterator<int>`** o **`map_iterator<int>`**. Esto puede evitarse haciendo que la clase **`iterator`** sea una *"clase anidada"* (*"nested class"*) de su contenedor, es decir que la declaración de la clase **`iterator`** está dentro de la clase del contenedor correspondiente. De esta manera, el contenedor actúa como un **`namespace`** para la clase del contenedor y así pasa a llamarse **`list<int>::iterator`**, **`set<int>::iterator`**...

2.1.10.2. Ejemplo de uso

```
1. bool check_sum(list<int> &L1, list<int> &L2) {
2.   list<int>::iterator p,q;
3.   p = L1.begin();
4.   q = L2.begin();
```

```
5.   int suma = 0;
6.   while (true) {
7.     if (q==L2.end()) break;
8.     else if (suma==*q) { suma=0; q++; }
9.     else if (p==L1.end()) break;
10.    else if (suma<*q) suma += *p++;
11.    else return false;
12.  }
13.  return suma==0 && p==L1.end() && q==L2.end();
14. }
```

Código 2.12: *El procedimiento* `check-sum`*, con la sintaxis de la librería STL. [Archivo: check-sum-stl.cpp]*

Con la nueva sintaxis (idéntica a la de los contenedores de STL) el ejemplo del procedimiento **`bool check-sum(L1,L2)`** descrito en la sección§2.1.5 puede escribirse como en el código 2.12.

2.1.10.2.1. Uso de templates y clases anidadas El uso de templates permite definir contenedores en base al tipo de elemento en forma genérica. En el ejemplo usamos listas de enteros mediante la expresión **`list<int>`**. Como las posiciones están declaradas *dentro* de la clase del contenedor deben declararse con el *scope* (*"alcance"*) correspondiente, es decir **`list<int>::iterator`**.

2.1.10.2.2. Operadores de incremento prefijo y postfijo: Recordemos que los operadores de incremento *"prefijo"* (**`++p`**) y *"postfijo"* (**`p++`**) tienen el mismo *"efecto colateral"* (*"side effect"*) que es incrementar la variable **`p`** pero tienen diferente *"valor de retorno"* a saber el valor incrementado en el caso del postfijo y el valor no incrementado para el prefijo. Es decir

- **`q = p++;`** es equivalente a **`q = p; p = p.next();`**, mientras que
- **`q = ++p;`** es equivalente a **`p = p.next(); q = p;`** .

Por ejemplo en la línea 10, incrementamos la variable **`suma`** con el elemento de la posición **`p`** *antes* de incrementar **`p`**.

2.1.10.3. Detalles de implementación

```
1.  #ifndef AED_LIST_H
2.  #define AED_LIST_H
3.
4.  #include <cstddef>
5.  #include <iostream>
6.
7.  namespace aed {
8.
9.    template<class T>
10.   class list {
11.   public:
12.     class iterator;
13.   private:
14.     class cell {
15.       friend class list;
16.       friend class iterator;
17.       T t;
18.       cell *next;
19.       cell() : next(NULL) {}
20.     };
21.     cell *first, *last;
22.   public:
23.     class iterator {
24.     private:
25.       friend class list;
26.       cell* ptr;
27.     public:
28.       T & operator*() { return ptr->next->t; }
29.       T *operator->() { return &ptr->next->t; }
30.       bool operator!=(iterator q) { return ptr!=q.ptr; }
31.       bool operator==(iterator q) { return ptr==q.ptr; }
32.       iterator(cell *p=NULL) : ptr(p) {}
33.       // Prefix:
34.       iterator operator++() {
35.         ptr = ptr->next;
36.         return *this;
37.       }
38.       // Postfix:
39.       iterator operator++(int) {
40.         iterator q = *this;
41.         ptr = ptr->next;
42.         return q;
43.       }
44.     };
```

```
45.
46.     list() {
47.       first = new cell;
48.       last = first;
49.     }
50.     ~list() { clear(); delete first; }
51.     iterator insert(iterator p,T t) {
52.       cell *q = p.ptr->next;
53.       cell *c = new cell;
54.       p.ptr->next = c;
55.       c->next = q;
56.       c->t = t;
57.       if (q==NULL) last = c;
58.       return p;
59.     }
60.     iterator erase(iterator p) {
61.       cell *q = p.ptr->next;
62.       if (q==last) last = p.ptr;
63.       p.ptr->next = q->next;
64.       delete q;
65.       return p;
66.     }
67.     iterator erase(iterator p,iterator q) {
68.       cell *s, *r = p.ptr->next;
69.       p.ptr->next = q.ptr->next;
70.       if (!p.ptr->next) last = p.ptr;
71.       while (r!=q.ptr->next) {
72.         s = r->next;
73.         delete r;
74.         r = s;
75.       }
76.       return p;
77.     }
78.     void clear() { erase(begin(),end()); }
79.     iterator begin() { return iterator(first); }
80.     iterator end() { return iterator(last); }
81.     void print() {
82.       iterator p = begin();
83.       while (p!=end()) std::cout << *p++ << " ";
84.       std::cout << std::endl;
85.     }
86.     void printd() {
87.       std::cout << "h(" << first << ")" << std::endl;
88.       cell *c = first->next;
89.       int j=0;
90.       while (c!=NULL) {
91.         std::cout << j++ << "(" << c << ") :" << c->t << std::endl;
92.         c = c->next;
```

```
93.       }
94.     }
95.     int size() {
96.       int sz = 0;
97.       iterator p = begin();
98.       while (p++!=end()) sz++;
99.       return sz;
100.    }
101.  };
102.
103. }
104. #endif
```

Código 2.13: *Implementación de listas por punteros con sintaxis compatible STL. Declaraciones. [Archivo: list.h]*

En el código 2.13 podemos ver un posible juego de declaraciones para las listas implementadas por punteros con sintaxis compatible STL. Como es la clase que usaremos en los ejemplos hemos incluido todos los detalles de implementación (en los ejemplos anteriores hemos omitido algunos detalles para facilitar la lectura).

- El encabezado `#ifndef AED_LIST_H...` es para evitar la doble inclusión de los headers.
- Hemos incluido un `namespace aed` para evitar la colisión con otras clases con nombres similares que pudieran provenir de otros paquetes. Por lo tanto las clases deben ser en realidad referenciadas como `aed::list<int>` y `aed::list<int>::iterator`. Otra posibilidad es incluir una declaración `using namespace aed;`.
- La clase list va precedida del calificador `template<class T>` que indica que la clase `T` es un tipo genérico a ser definido en el momento de *instanciar* la clase. Por supuesto, también puede ser un tipo básico como `int` o `double`. Por ejemplo, al declarar `list<int>` el tipo genérico `T` pasa a tomar el valor concreto `int`.
- La clase `cell<T>` ha sido incluida también como clase anidada dentro de la clase `list<T>`. Esto permite (al igual que con `iterator`) tener una clase `cell` para cada tipo de contenedor (lista, pila, cola...) sin necesidad de agregarle un prefijo o sufijo (como en `list_cell`, `stack_cell`, etc...).

- La clase `cell<T>` declara `friend` a las clases `list<T>` e `iterator<T>`. Recordemos que el hecho de declarar la clase en forma anidada dentro de otra no tiene ninguna implicancia en cuanto a la privacidad de sus miembros. Si queremos que `cell<T>` acceda a los miembros privados de `list<T>` y `iterator<T>` entonces debemos declarar a las clases como `friend.`

- Las clases `list<T>`, `cell<T>` e `iterator<T>` son un ejemplo de *"clases fuertemente ligadas"* (*"tightly coupled classes"*), esto es una serie de grupo de clases que están conceptualmente asociadas y probablemente son escritas por el mismo programador. En tal caso es común levantar todas las restricciones entre estas clases con declaraciones `friend`. En este caso sólo es necesario que `cell` declare friend a `iterator` y `list`, y que `iterator` declare `friend` a `list`.

- La clase `cell<T>` es declarada privada dentro de `list<T>` ya que normalmente no debe ser accedida por el usuario de la clase, el cual sólo accede los valores a través de `iterator<T>`.

- Para poder sobrecargar los operadores de incremento y dereferenciación (`p++`, `++p` y `*p`) debemos declarar a `iterator<T>` como una clase y no como un `typedef` como en la interfaz básica. Esta clase contiene como único miembro un puntero a celda `cell *ptr`. Como ya no es un `typedef` también debemos declarar los operadores de comparación `p!=q` y `p==q`. (Los operadores `<`, `>`, `<=`, y `>=` también podrían ser definidos, pero probablemente serían $O(n)$). Por supuesto estos operadores comparan simplemente los punteros a las celdas correspondientes. Otro inconveniente es que ahora hay que extraer el campo `ptr` cada vez que se hace operaciones sobre los enlaces, por ejemplo la primera línea de `next()`

  ```
  1. iterator_t q = p->next;
  ```

  se convierte en

  ```
  1. cell *q = p.ptr->next;
  ```

2.1.10.4. Listas doblemente enlazadas

Si es necesario realizar repetidamente la operación `q=L.prev(p)` que retorna la posición `q` anterior a `p` en la lista `L`, entonces probablemente con-

venga utilizar una *"lista doblemente enlazada"*. En este tipo de listas cada celda tiene dos punteros uno al elemento siguiente y otro al anterior.

```
1. class cell {
2.   elem_t elem;
3.   cell *next, *prev;
4.   cell() : next(NULL), prev(NULL) {}
5. };
```

Una ventaja adicional es que en este tipo de implementación la posición puede implementarse como un *"puntero a la celda que contiene el elemento"* y no a la celda precedente, como en las listas simplemente enlazadas (ver §2.1.7.1). Además las operaciones sobre la lista pasan a ser completamente simétricas en cuanto al principio y al fin de la lista.

Notemos también que en este caso al eliminar un elemento en la posición `p`, ésta deja de ser válida efectivamente, ya que la celda a la que apunta desaparece. En la versión simplemente enlazada, en cambio, la celda a la que apunta la posición sigue existiendo, con lo cual la posición en principio sigue siendo válida. Es decir, por ejemplo en el código código 2.4, la línea 8 puede ser reemplazada por `L.erase(q)` *sin actualizar* `q`. Sin embargo, por una cuestión de uniformidad conviene mantener la convención de reasignar siempre la posición, de manera que el código siga valiendo tanto para listas simple como doblemente enlazadas.

2.2. El TAD pila

Básicamente es una lista en la cual todas las operaciones de inserción y borrado se producen en uno de los extremos de la lista. Un ejemplo gráfico es una pila de libros en un cajón. A medida que vamos recibiendo más libros los ubicamos en la parte superior. En todo momento tenemos acceso sólo al libro que se encuentra sobre el *"tope"* de la pila. Si queremos acceder a algún libro que se encuentra más abajo (digamos en la quinta posición desde el tope) debemos sacar los primeros cuatro libros y ponerlos en algún lugar para poder acceder al mismo. La Pila es el típico ejemplo de la estructura tipo *"LIFO"* (por *"Last In First Out"*, es decir *"el último en entrar es el primero en salir"*).

La pila es un subtipo de la lista, es decir podemos definir todas las operaciones abstractas sobre pila en función de las operaciones sobre lista. Esto motiva la idea de usar *"adaptadores"* es decir capas de código (en la forma

de templates de C++) para adaptar cualquiera de las posibles clases de lista (por arreglo, punteros o cursores) a una pila.

Ejemplo 2.3: *Consigna:* Escribir un programa para calcular expresiones aritméticas complejas con números en doble precisión usando *"notación polaca invertida"* (RPN, por *"reverse polish notation"*).

Solución: En las calculadoras con RPN una operación como 2+3 se introduce en la forma 2 3 +. Esto lo denotamos así

$$\mathrm{rpn}[2+3] = 2, 3, + \tag{2.7}$$

donde hemos separados los elementos a ingresar en la calculadora por comas. En general, para cualquier operador binario (como **+**, **-**, ***** o **/**) tenemos

$$\mathrm{rpn}[(a)\,\theta\,(b)] = \mathrm{rpn}(a),\ \mathrm{rpn}(b),\ \theta \tag{2.8}$$

donde a, b son los operandos y θ el operador. Hemos introducido paréntesis alrededor de los operandos a y b ya que (eventualmente) estos pueden ser también expresiones, de manera que, por ejemplo la expresión $(2+3)*(4-5)$ puede escribirse como

$$\begin{aligned} \mathrm{rpn}[(2+3)*(4-5)] &= \mathrm{rpn}[2+3]\,, \mathrm{rpn}[4-5]\,, * \\ &= 2\,, 3\,, +\,, 4\,, 5\,, -\,, * \end{aligned} \tag{2.9}$$

La ventaja de una tal calculadora es que no hace falta ingresar paréntesis, con el inconveniente de que el usuario debe convertir mentalmente la expresión a RPN.

2.2.1. Una calculadora RPN con una pila

La forma de implementar una calculadora RPN es usando una pila. A medida que el usuario entra operandos y operadores se aplican las siguientes reglas

- Si el usuario ingresó un operando, entonces simplemente se almacena en la pila.
- Si ingresó un operador θ se extraen dos operandos del tope de la pila, digamos t el tope de la pila y u el elemento siguiente, se aplica el operador a los dos operandos (en forma invertida) es decir $u\,\theta\,t$ y se almacena el resultado en el tope de la pila.

Ingresa	Tipo	Pila
2	operando	2
3	operando	3,2
+	operador	5
4	operando	4,5
5	operando	5,4,5
-	operador	-1,5
*	operador	-5

Tabla 2.3: Seguimiento del funcionamiento de la calculadora RPN usando una pila. (Los elementos en la pila son enumerados empezando por el tope.)

Por ejemplo, para la expresión (2.9) tenemos un seguimiento como el mostrado en la tabla 2.3. que es el resultado correcto. El algoritmo puede extenderse fácilmente a funciones con un número arbitrario de variables (como `exp()`, `cos()` ...), sólo que en ese caso se extrae el número de elementos apropiados de la pila, se le aplica el valor y el resultado es introducido en la misma. De nuevo, notar que en general debe invertirse el orden de los argumentos al sacarlos de la pila. Por ejemplo la función `rem(a,b)` (resto) retorna el resto de dividir `b` en `a`. Si analizamos la expresión `mod(5,3)` (que debe retornar 2) notamos que al momento de aplicar la función `mod`, tenemos en la pila los elementos `3,5` (el top primero), de manera que la función debe aplicarse a los elementos *en orden invertido*.

2.2.2. Operaciones abstractas sobre pilas

El ejemplo anterior sugiere las siguientes operaciones abstractas sobre pilas

- Insertar un elemento en el tope de la pila.
- Obtener el valor del elemento en el tope de la pila.
- Eliminar el elemento del tope.

2.2.3. Interfaz para pila

```
1. elem_t top();
2. void pop();
```

```
3. void push(elem_t x);
4. void clear();
5. int size();
6. bool empty();
```

Código 2.14: *Interfaz [Archivo: sbas.h]*

Una posible interfaz puede observarse en el código 2.14. Consta de sólo tres funciones básicas a saber,

- `top()` devuelve el elemento en el tope de la pila (sin modificarla).
- `pop()` remueve el elemento del tope (sin retornar su valor!).
- `push(x)` inserta el elemento `x` en el tope de la pila.

Esta interfaz es directamente compatible con STL, ya que, a diferencia con las listas, la pila no tiene iterators.

Casi todas las librerías que implementan pilas usan una interfaz similar a esta con muy pequeñas variantes. En algunos casos, por ejemplo, la función que remueve el elemento también devuelve el elemento del tope.

También hemos agregado tres funciones auxiliares más a saber

- `clear()` remueve todos los elementos de la pila.
- `int size()` devuelve el número de elementos en la pila.
- `bool empty()` retorna verdadero si la pila esta vacía, verdadero en caso contrario.

Notar que `empty()` *no modifica* la pila, mucha gente tiende a confundirla con `clear()`.

2.2.4. Implementación de una calculadora RPN

```
1. bool check2(stack &P,double &v1, double&v2) {
2.   if (P.size()<2) {
3.     cout << "Debe haber al menos 2 elementos en la pila!!\n";
4.     return false;
5.   } else {
6.     v2 = P.top(); P.pop();
7.     v1 = P.top(); P.pop();
```

```
8.     return true;
9.   }
10. }
11.
12. bool check1(stack &P,double &v1) {
13.   if (P.size()<1) {
14.     cout << "Debe haber al menos 1 elemento en la pila!!\n";
15.     return false;
16.   } else {
17.     v1 = P.top(); P.pop();
18.     return true;
19.   }
20. }
21.
22. int main() {
23.   stack P,Q;
24.   const int SIZE=100;
25.   char line[SIZE];
26.   double v1,v2;
27.   // REPL (read, eval print loop)
28.   while(true) {
29.     // Read
30.     cout << "calc> ";
31.     assert(line);
32.     cin.getline(line,SIZE,'\n');
33.     if(!cin) break;
34.     // 'Eval' y 'print' dependiendo del caso
35.     if (!strcmp(line,"+")) {
36.       if (check2(P,v1,v2)) {
37.         P.push(v1+v2);
38.         printf("->%lf\n",P.top());
39.       }
40.     } else if (!strcmp(line,"-")) {
41.       if (check2(P,v1,v2)) {
42.         P.push(v1-v2);
43.         printf("->%lf\n",P.top());
44.       }
45.     } else if (!strcmp(line,"*")) {
46.       if (check2(P,v1,v2)) {
47.         P.push(v1*v2);
48.         printf("->%lf\n",P.top());
49.       }
50.     } else if (!strcmp(line,"/")) {
51.       if (check2(P,v1,v2)) {
52.         P.push(v1/v2);
53.         printf("->%lf\n",P.top());
54.       }
55.     } else if (!strcmp(line,"log")) {
```

```
56.       if (check1(P,v1)) {
57.         P.push(log(v1));
58.         printf("->%lf\n",P.top());
59.       }
60.     } else if (!strcmp(line,"exp")) {
61.       if (check1(P,v1)) {
62.         P.push(exp(v1));
63.         printf("->%lf\n",P.top());
64.       }
65.     } else if (!strcmp(line,"sqrt")) {
66.       if (check1(P,v1)) {
67.         P.push(sqrt(v1));
68.         printf("->%lf\n",P.top());
69.       }
70.     } else if (!strcmp(line,"atan2")) {
71.       if (check2(P,v1,v2)) {
72.         P.push(atan2(v1,v2));
73.         printf("->%lf\n",P.top());
74.       }
75.     } else if (!strcmp(line,"c")) {
76.       printf("vaciando la pila...\n");
77.       P.clear();
78.     } else if (!strcmp(line,"p")) {
79.       printf("pila: ");
80.       while(!P.empty()) {
81.         double x = P.top();
82.         cout << x << " ";
83.         P.pop();
84.         Q.push(x);
85.       }
86.       while(!Q.empty()) {
87.         double x = Q.top();
88.         Q.pop();
89.         P.push(x);
90.       }
91.       cout << endl;
92.     } else if (!strcmp(line,"x")) {
93.       "Saliendo de calc!!\n";
94.       exit(0);
95.     } else {
96.       double val;
97.       int nread = sscanf(line,"%lf",&val);
98.       if (nread!=1) {
99.         printf("Entrada invalida!!: \"%s\"\n",line);
100.        continue;
101.      } else {
102.        P.push(val);
103.        printf("<-%g\n",val);
```

```
104.         }
105.       }
106.     }
107.   }
```

Código 2.15: *Implementación de una calculadora RPN usando una pila. [Archivo: stackcalc.cpp]*

En el código 2.15 vemos una posible implementación de la calculadora usando la interfaz STL de la pila descripta en código 2.14. En el `main()` se declara la pila `P` que es la base de la calculadora. Después de la declaración de algunas variables auxiliares se ingresa en un lazo infinito, que es un ejemplo de lazo *"REPL"* (por *"read, eval, print loop"*), típico en los lenguajes interpretados.

- **read:** Se lee una línea de la consola,
- **eval:** Se evalúa para producir efectos laterales y producir un resultado
- **print:** Se imprime el resultado de la evaluación .

La línea se lee con la función `getline()` del standard input `cin`. Si hay cualquier tipo de error al leer (fin de archivo, por ejemplo) `cin` queda en un estado que retorna `false`, de ahí que basta con verificar `!cin` para ver si la lectura ha sido exitosa. El valor leído queda en el string (de C) `line`. El string tiene un tamaño fijo `SIZE`. También se podría hacer dinámicamente con rutinas más elaboradas y seguras como la `snprintf` o `asprintf` (ver Foundation [b]). Antes de leer la línea se imprime el *prompt* "`calc>` ".

Después de leer la línea se entra en una secuencia de `if-else` (similar a un `switch`). Si la línea entrada es un operador o función, entonces se extrae el número apropiado de operandos de la pila, se aplica la operación correspondiente y el resultado es ingresado en la pila. Es importante verificar que la pila contenga un número apropiado de valores antes de hacer la operación. Por ejemplo, si el usuario entra `+` entonces debemos verificar que al menos haya 2 operandos en la pila. Esto se hace con la función `check2` que verifica que efectivamente haya dos operandos, los extrae de la pila y los pone en las variables `v1` y `v2`. Si no hay un número apropiado de valores entonces `check2` retorna `false`. Para funciones u operadores unarios usamos la función similar `check1`.

Además de los 4 operadores binarios normales y de algunas funciones comunes, hemos incluido un comando para salir de la calculadora (**x**), para limpiar la pila (**c**) y para imprimir la pila (**p**).

Notar que para imprimir la pila se necesita una pila auxiliar **Q**. Los elementos de la pila se van extrayendo de **P**, se imprimen por consola y se guardan en **Q**. Una vez que **P** está vacía, todos los elementos de **Q** son devueltos a **P**.

Una sesión típica, correspondiente a (2.9), sería así

```
1. [mstorti@spider aedsrc]$ stackcalc
2. calc> 2
3. <- 2
4. calc> 3
5. <- 3
6. calc> +
7. -> 5.000000
8. calc> 4
9. <- 4
10. calc> 5
11. <- 5
12. calc> -
13. -> -1.000000
14. calc> *
15. -> -5.000000
16. calc> x
17. [mstorti@spider aedsrc]$
```

2.2.5. Implementación de pilas mediante listas

Como ya mencionamos, la pila se puede implementar fácilmente a partir de una lista, *asumiendo que el tope de la pila está en el comienzo de la lista*. **push(x)** y **pop()** se pueden implementar a partir de **insert** y **erase** en el comienzo de la lista. **top()** se puede implementar en base a **retrieve**. La implementación por punteros y cursores es apropiada, ya que todas estas operaciones son $O(1)$. Notar que si se usa el último elemento de la lista como tope, entonces las operaciones de **pop()** pasan a ser $O(n)$, ya que una vez que, asumiendo que contamos con la posición **q** del último elemento de la lista, al hacer un **pop()**, **q** deja de ser válida y para obtener la nueva posición del último elemento de la lista debemos recorrerla desde el principio. La implementación de pilas basada en listas puede observarse en los código 2.16 y código 2.17. Notar que la implementación es muy simple, ya que prácticamente todas las operaciones son transferidas al tipo lista.

Por otra parte, en la implementación de listas por arreglos tanto la inserción como la supresión en el primer elemento son $O(n)$. En cambio, si podría implementarse con una implementación basada en arreglos *si el tope de la pila está al final*.

El tamaño de la pila es guardado en un miembro `int size_m`. Este contador es inicializado a cero en el constructor y después es actualizado durante las operaciones que modifican la longitud de la lista como `push()`, `pop()` y `clear()`.

```
1.  class stack : private list {
2.  private:
3.    int size_m;
4.  public:
5.    stack();
6.    void clear();
7.    elem_t& top();
8.    void pop();
9.    void push(elem_t x);
10.   int size();
11.   bool empty();
12. };
```

Código 2.16: *Pila basada en listas. Declaraciones. [Archivo: stackbas.h]*

```
1.  stack::stack() : size_m(0) { }
2.
3.  elem_t& stack::top() {
4.    return retrieve(begin());
5.  }
6.
7.  void stack::pop() {
8.    erase(begin()); size_m--;
9.  }
10.
11. void stack::push(elem_t x) {
12.   insert(begin(),x); size_m++;
13. }
14.
15. void stack::clear() {
16.   erase(begin(),end()); size_m = 0;
17. }
```

```
18.
19. bool stack::empty() {
20.   return begin()==end();
21. }
22.
23. int stack::size() {
24.   return size_m;
25. }
```

Código 2.17: *Pila basada en listas. Implementación. [Archivo: stackbas.cpp]*

2.2.6. La pila como un adaptador

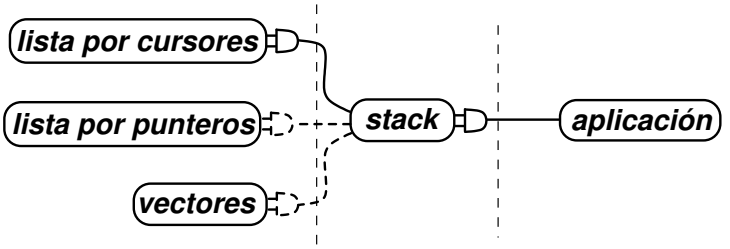


Figura 2.14: La clase pila como un adaptador para varios contenedores básicos de STL

Notar que la pila deriva directamente de la lista, pero con una declaración **private**. De esta forma el usuario de la clase **stack** no puede usar métodos de la clase lista. El hecho de que la pila sea tan simple permite que pueda ser implementada en términos de otros contenedores también, como por ejemplo el contenedor **vector** de STL. De esta forma, podemos pensar a la pila como un *"adaptador"* (*"container adaptor"*), es decir que brinda un subconjunto de la funcionalidad del contenedor original ver figura 2.14. La ventaja de operar sobre el adaptador (en este caso la pila) y no directamente sobre el contenedor básico (en este caso la lista) es que el adaptador puede después conectarse fácilmente a otros contenedores (en la figura representado por enchufes).

2.2.7. Interfaz STL

```
1.  #ifndef AED_STACK_H
2.  #define AED_STACK_H
3.
4.  #include <aedsrc/list.h>
5.
6.  namespace aed {
7.
8.    template<class T>
9.    class stack : private list<T> {
10.   private:
11.     int size_m;
12.   public:
13.     stack() : size_m(0) { }
14.     void clear() { erase(begin(),end()); size_m = 0; }
15.     T &top() { return *begin(); }
16.     void pop() { erase(begin()); size_m--; }
17.     void push(T x) { insert(begin(),x); size_m++; }
18.     int size() { return size_m; }
19.     bool empty() { return size_m==0; }
20.   };
21. }
22. #endif
```

Código 2.18: *Clase pila con templates. [Archivo: stack.h]*

Como la pila en si misma no contiene iteradores no hay necesidad de clases anidadas ni sobrecarga de operadores, de manera que la única diferencia con la interfaz STL es el uso de templates. Una interfaz compatible con STL puede observarse en el código 2.18.

2.3. El TAD cola

Por contraposición con la pila, la cola es un contenedor de tipo *"FIFO"* (por *"First In First Out"*, el primero en entrar es el primero en salir). El ejemplo clásico es la cola de la caja en el supermercado. La cola es un objeto muchas veces usado como buffer o pulmón, es decir un contenedor donde almacenar una serie de objetos que deben ser procesados, manteniendo el orden en el que ingresaron. La cola es también, como la pila, un subtipo de la lista llama también a ser implementado como un adaptador.

2.3.1. Intercalación de vectores ordenados

Ejemplo 2.4: Un problema que normalmente surge dentro de los algoritmos de ordenamiento es el intercalamiento de contenedores ordenados. Por ejemplo, si tenemos dos listas ordenadas **L1** y **L2**, el proceso de intercalamiento consiste en generar una nueva lista **L** ordenada que contiene los elementos en **L1** y **L2** de tal forma que **L** está ordenada, en la forma lo más eficiente posible. Con listas es fácil llegar a un algoritmo $O(n)$ simplemente tomando de las primeras posiciones de ambas listas el menor de los elementos e insertándolo en **L** (ver secciones §4.3.0.2 y §5.6). Además, este algoritmo no requiere memoria adicional, es decir, no necesita alocar nuevas celdas ya que los elementos son agregados a **L** a medida que se eliminan de **L1** y **L2** (Cuando manipula contenedores sin requerir memoria adicional se dice que es *"in place"* (*"en el lugar"*)). Para vectores el problema podría plantearse así.

Consigna: Sea **a** un arreglo de longitud par, tal que las posiciones pares como las impares están ordenadas entre sí, es decir

$$\begin{aligned} a_0 \leq a_2 \leq \cdots \leq a_{n-2} \\ a_1 \leq a_3 \leq \cdots \leq a_{n-1} \end{aligned} \tag{2.10}$$

Escribir un algoritmo que ordena los elementos de **a**.

2.3.1.1. Ordenamiento por inserción

Solución: Consideremos por ejemplo que el arreglo contiene los siguientes elementos.

$$a = \texttt{10 1 12 3 14 5 16 7 18 9 20 51 22 53 24 55 26 57 28 59} \tag{2.11}$$

Verificamos que los elementos en las posiciones pares (`10 12 14 16...`) están ordenados entre sí, como también los que están en las posiciones impares (`1 3 5 7...`). Consideremos primero el algoritmo de *"ordenamiento por inserción"* (ver código 2.19, los algoritmos de ordenamiento serán estudiados en más detalle en un capítulo posterior).

```
1. void inssort(vector<int> &a) {
2.   int n=a.size();
3.   for (int j=1; j<n; j++) {
4.     int x = a[j];
5.     int k = j;
```

```
6.      while (--k>=0 && x<a[k]) a[k+1] = a[k];
7.      a[k+1] = x;
8.    }
9.  }
```

Código 2.19: *Algoritmo de ordenamiento por inserción. [Archivo: inssort.cpp]*

El cursor j va avanzando desde el comienzo del vector hasta el final. Después de ejecutar el cuerpo del lazo sobre j el rango de elementos $[0, j]$ queda ordenado. (Recordar que $[a, b)$ significa los elementos que están en las posiciones entre a y b *incluyendo* a a y *excluyendo* a b). Al ejecutar el lazo para un dado j, los elementos $a_0, \ldots, a_{j-1}$ están ordenados, de manera que basta con *"insertar"* (de ahí el nombre del método) el elemento a_j en su posición correspondiente. En el lazo sobre k, todos los elementos mayores que a_j son desplazados una posición hacia el fondo y el elemento es insertado

en alguna posición $p <= j$, donde corresponde.

```
10 [1]^j 12  3 14  5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
[1]^p 10 12  3 14  5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
 1 10 [12]^j 3 14  5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
 1 10 [12]^p 3 14  5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
 1 10 12 [3]^j 14  5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
 1 [3]^p 10 12 14  5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
 1  3 10 12 [14]^j 5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
 1  3 10 12 [14]^p 5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
 1  3 10 12 14 [5]^j 16  7 18  9 20 51 22 53 24 55 26 57 28 59
 1  3 [5]^p 10 12 14 16  7 18  9 20 51 22 53 24 55 26 57 28 59
 1  3  5 10 12 14 [16]^j 7 18  9 20 51 22 53 24 55 26 57 28 59
 1  3  5 10 12 14 [16]^p 7 18  9 20 51 22 53 24 55 26 57 28 59
 1  3  5 10 12 14 16 [7]^j 18  9 20 51 22 53 24 55 26 57 28 59
 1  3  5 [7]^p 10 12 14 16 18  9 20 51 22 53 24 55 26 57 28 59
 1  3  5  7 10 12 14 16 [18]^j 9 20 51 22 53 24 55 26 57 28 59
 1  3  5  7 10 12 14 16 [18]^p 9 20 51 22 53 24 55 26 57 28 59
 1  3  5  7 10 12 14 16 18 [9]^j 20 51 22 53 24 55 26 57 28 59
 1  3  5  7 [9]^p 10 12 14 16 18 20 51 22 53 24 55 26 57 28 59
```

(2.12)

En (2.12) vemos un seguimiento de algunas de las operaciones de inserción. Cada par de líneas corresponde a la ejecución para un j dado, la primera línea muestra el estado del vector antes de ejecutar el cuerpo del lazo y la segunda línea muestra el resultado de la operación. En ambos casos se indica con una caja el elemento que es movido desde la posición j a la p. Por ejemplo, para $j = 9$ el elemento $a_j = 9$ debe viajar hasta la posición $p = 4$, lo cual involucra desplazar todos los elementos que previamente estaban en el rango $[4, 9)$ una posición hacia arriba en el vector. Este algoritmo funciona, por supuesto, para cualquier vector, independientemente de si las posiciones pares e impares están ordenadas entre si, como estamos asumiendo en este ejemplo.

2.3.1.2. Tiempo de ejecución

Consideremos ahora el tiempo de ejecución de este algoritmo. El lazo sobre j se ejecuta $n-1$ veces, y el lazo interno se ejecuta, en el peor caso $j-1$ veces, con lo cual el costo del algoritmo es, en el peor caso

$$T_{\text{peor}}(n) = \sum_{j=1}^{n-1}(j-1) = O(n^2) \tag{2.13}$$

En el mejor caso, el lazo interno no se ejecuta ninguna vez, de manera que sólo cuenta el lazo externo que es $O(n)$.

En general, el tiempo de ejecución del algoritmo dependerá de cuantas posiciones deben ser desplazadas (es decir $p-j$) para cada j

$$T(n) = \sum_{j=1}^{n-1}(p-j) \tag{2.14}$$

o, tomando promedios

$$T_{\text{prom}}(n) = \sum_{j=1}^{n-1} d_j \tag{2.15}$$

donde d_j es el tamaño promedio del rango que se desplaza.

El promedio debe ser tomado sobre un cierto conjunto de posibles vectores. Cuando tomamos vectores completamente desordenados, se puede ver fácilmente que $d_j = j/2$ ya que el elemento a_j no guarda ninguna relación con respecto a los elementos precedentes y en promedio irá a parar a la mitad del rango ordenado, es decir $p = j/2$ y entonces $j - p = j/2$, de manera que

$$T_{\text{prom}}(n) = \sum_{j=1}^{n-1} d_j = \sum_{j=1}^{n-1} \frac{j}{2} = O(n^2) \tag{2.16}$$

2.3.1.3. Particularidades al estar las secuencias pares e impares ordenadas

Como la intercalación de listas ordenadas es $O(n)$ surge la incógnita de si el algoritmo para arreglos puede ser mejorado. Al estar las posiciones pares e impares ordenadas entre sí puede ocurrir que *en promedio* el desplazamiento sea menor, de hecho, generando vectores en forma aleatoria, pero tales que sus posiciones pares e impares estén ordenada se llega a

la conclusión que el desplazamiento promedio es $O(\sqrt{n})$, de manera que el algoritmo resulta ser $O(n^{3/2})$. Esto representa una gran ventaja contra el $O(n^2)$ del algoritmo de ordenamiento original.

De todas formas, podemos mejorar más aún esto si tenemos en cuenta que las subsecuencias pares e impares están ordenadas. Por ejemplo consideremos lo que ocurre en el seguimiento (2.12) al mover los elementos 18 y 9 que originalmente estaban en las posiciones $q = 8$ y $q + 1 = 9$. Como vemos, los elementos en las posiciones 0 a $q - 1 = 7$ están ordenados. Notar que el máximo del rango ya ordenado $[0, q)$ es menor que el máximo de estos dos nuevos elementos a_q y a_{q+a}, ya que todos los elementos en $[0, q)$ provienen de elementos en las subsecuencias que estaban *antes* de a_q y a_{q+1}.

$$\max_{j=0}^{q-1} a_j < \max(a_q, a_{q+1}) \tag{2.17}$$

por lo tanto después de insertar los dos nuevos elementos, el mayor (que en este caso es 18) quedará en la posición $q + 1$. El menor ($\min(a_q, a_{q+1}) = 9$) viaja una cierta distancia, hasta la posición $p = 4$. Notar que, por un razonamiento similar, todos los elementos en las posiciones $[q+2, n)$ deben ser mayores que $\min(a_q, a_{q+1})$, de manera que los elementos en $[0, p)$ no se moverán a partir de esta inserción.

2.3.1.4. Algoritmo de intercalación con una cola auxiliar

```
1.  void merge(vector<int> &a) {
2.    int n = a.size();
3.    // C = cola vacia . . .
4.    int p=0, q=0, minr, maxr;
5.    while (q<n) {
6.      // minr = min(a_q,a_{q+1}, maxr = max(a_q,a_{q+1}
7.      if (a[q]<=a[q+1]) {
8.        minr = a[q];
9.        maxr = a[q+1];
10.     } else {
11.       maxr = a[q];
12.       minr = a[q+1];
13.     }
14.     // Apendizar todos los elementos del frente de la cola menores que
15.     // min(a_q,a_{q+1}) al rango [0,p), actualizando eventualmente
16.     while ( /* C no esta vacia. . . */) {
17.       x = /* primer elemento de C . . . */;
18.       if (x>minr) break;
```

```
19.       a[p++] = x;
20.       // Saca primer elemento de C . . .
21.     }
22.     a[p++] = minr;
23.     a[p++] = minr;
24.     // Apendizar 'maxr' al rango [0,p) . . .
25.     q += 2;
26.   }
27.   // Apendizar todos los elementos en C menores que
28.   // min(a_q,a_{q+1}) al rango [0,p)
29.   // . . .
30. }
```

Código 2.20: *Algoritmo de intercalación con una cola auxiliar [Archivo: mr-garray1.cpp]*

El algoritmo se muestra en el código 2.20, manteniendo el rango $[p, q)$ en una cola auxiliar C. En (2.18) vemos el seguimiento correspondiente. Los elementos en la cola C son mostrados encerrados en una caja, en el rango $[p, q)$. Notar que si bien, el número de elementos en la cola entra exactamente en ese rango, en la implementación el estado los elementos en ese rango es irrelevante y los elementos están en una cola auxiliar.

$$
\begin{array}{l}
1\,\boxed{10}^{C}\,12\;3\;14\;5\;16\;7\;18\;9\;20\;51\;22\;53\;24\;55\;26\;57\;28\;59\\
1\;\;3\,\boxed{10\;12}^{C}\,14\;5\;16\;7\;18\;9\;20\;51\;22\;53\;24\;55\;26\;57\;28\;59\\
1\;\;3\;\;5\,\boxed{10\;12\;14}^{C}\,16\;7\;18\;9\;20\;51\;22\;53\;24\;55\;26\;57\;28\;59\\
1\;\;3\;\;5\;\;7\,\boxed{10\;12\;14\;16}^{C}\,18\;9\;20\;51\;22\;53\;24\;55\;26\;57\;28\;59\\
1\;\;3\;\;5\;\;7\;\;9\,\boxed{10\;12\;14\;16\;18}^{C}\,20\;51\;22\;53\;24\;55\;26\;57\;28\;59\\
1\;\;3\;\;5\;\;7\;\;9\;10\;12\;14\;16\;18\;20\boxed{51}^{C}\,22\;53\;24\;55\;26\;57\;28\;59\\
1\;\;3\;\;5\;\;7\;\;9\;10\;12\;14\;16\;18\;20\;22\boxed{51\;53}^{C}\,24\;55\;26\;57\;28\;59\\
1\;\;3\;\;5\;\;7\;\;9\;10\;12\;14\;16\;18\;20\;22\;24\boxed{51\;53\;55}^{C}\,26\;57\;28\;59\\
1\;\;3\;\;5\;\;7\;\;9\;10\;12\;14\;16\;18\;20\;22\;24\;26\boxed{51\;53\;55\;57}^{C}\,28\;59\\
1\;\;3\;\;5\;\;7\;\;9\;10\;12\;14\;16\;18\;20\;22\;24\;26\;28\boxed{51\;53\;55\;57\;59}^{C}
\end{array}
\tag{2.18}
$$

Consideremos por ejemplo el paso en el cual se procesan los elementos $a_q = 20$ y $a_{q+1} = 51$ en las posiciones $q = 10$ y $q + 1 = 11$. En ese

momento la cola contiene los elementos 10,12,14,16 y 18. Como son todos menores que $\min(a_q, a_{q+1}) = 20$ apendizamos todos al rango $[0, p = 5)$ de manera que queda $p = 10$. Se apendiza también el 20, con lo cual queda $p = 11$ y finalmente se apendiza $\max(a_q, a_{q+1}) = 51$ a la cola. Como en ese momento la cola esta vacía, depués de la inserción queda en la cola solo el 51.

2.3.2. Operaciones abstractas sobre colas

Del ejemplo anterior se hacen evidentes las siguientes operaciones

- Obtener el elemento en el frente de la cola
- Eliminar el elemento en el frente de la cola
- Agregar un elemento a la cola

2.3.3. Interfaz para cola

Una versión reducida de la interfaz STL para la cola puede observarse en el código 2.21. Al igual que en el caso de la pila, la cola no tiene posiciones, de manera que no necesita clases anidadas ni sobrecarga de operadores, por lo que el código es sencillo, de manera que no presentamos una versión básica, como hemos hecho con las listas y pilas, sino que presentamos directamente la versión compatible STL.

Las operaciones abstractas descriptas se realizan a través de las funciones **`pop()`** y **`front()`**, que operan sobre el principio de la lista, y **`push()`** que opera sobre el fin de la lista. Todas estas operaciones son $O(1)$. Notar que de elegir lo opuesto (**`pop()`** y **`front()`** sobre el fin de la lista y **`push()`** sobre el principio) entonces la operación **`pop()`** sería $O(n)$ ya que acceder al último elemento de la lista (no **`end()`** que es una posición *fuera* de la lista) es $O(n)$.

También hemos agregado, como en el caso de la pila operaciones estándar **`size()`** y **`empty()`**, que también son $O(1)$ y **`clear()`** que es $O(n)$.

```
1. #ifndef AED_QUEUE_H
2. #define AED_QUEUE_H
3.
4. #include <aedsrc/list.h>
5.
6. namespace aed {
```

```
7.
8.   template<class T>
9.   class queue : private list<T> {
10.  private:
11.    int size_m;
12.  public:
13.    queue() : size_m(0) { }
14.    void clear() { erase(begin(),end()); size_m = 0; }
15.    T &front() { return *begin(); }
16.    void pop() { erase(begin()); size_m--; }
17.    void push(T x) { insert(end(),x); size_m++; }
18.    int size() { return size_m; }
19.    bool empty() { return size_m==0; }
20.  };
21. }
22. #endif
```

Código 2.21: *Interfaz STL para cola [Archivo: queue.h]*

2.3.4. Implementación del algoritmo de intercalación de vectores

El algoritmo completo, usando la interfaz STL puede observarse en el código 2.22.

```
1.  void merge(vector<int> &a) {
2.    queue<int> C;
3.    int n = a.size();
4.    if (n==0) return;
5.    if (n% 2) {
6.      cout << "debe haber un numero par de elementos en el vector\n";
7.      exit(1);
8.    }
9.    int p=0,q=0, minr, maxr;
10.
11.   print(a,C,p,q);
12.   while (q<n) {
13.     if (a[q]<=a[q+1]) {
14.       minr = a[q];
15.       maxr = a[q+1];
16.     } else {
17.       maxr = a[q];
```

```
18.       minr = a[q+1];
19.     }
20.     while (!C.empty() && C.front()<=minr) {
21.       a[p++] = C.front();
22.       C.pop();
23.     }
24.     a[p++] = minr;
25.     C.push(maxr);
26.     q += 2;
27.     print(a,C,p,q);
28.   }
29.   while (!C.empty()) {
30.     a[p++] = C.front();
31.     C.pop();
32.   }
33. }
```

Código 2.22: *Algoritmo de intercalación con una cola auxiliar. Implementación con la interfaz STL. [Archivo: mrgarray.cpp]*

2.3.4.1. Tiempo de ejecución

El lazo sobre q se ejecuta $n/2$ veces. Dentro del lazo todas las operaciones son de tiempo constante, salvo los lazos sobre la cola de las líneas 20–23 y 29–32. Las veces que el primer lazo se ejecuta para cada q puede ser completamente variable, pero notar que por cada ejecución de este lazo, un elemento es introducido en el rango $[0, p)$. Lo mismo ocurre para el segundo lazo. Como finalmente todos los elementos terminan en el rango $[0, p)$, el número de veces total que se ejecutan los dos lazos debe ser menor que n. De hecho como para cada ejecución del cuerpo del lazo sobre q se introduce un elemento en $[0, p)$ en la línea 24, el número de veces total que se ejecutan los dos lazos es exactamente igual a $n/2$. De manera que el algoritmo es finalmente $O(n)$.

Sin embargo, el algoritmo no es *in-place*, la memoria adicional está dada por el tamaño de la cola `C`. En el peor caso, `C` puede llegar a tener $n/2$ elementos y en el mejor caso ninguno. En el caso promedio, el tamaño máximo de `C` es tanto como el número de desplazamientos que deben hacerse en el algoritmo de inserción puro, descrito en la sección §2.3.1.3, es decir $O(\sqrt{n})$.

Todo esto esta resumido en la tabla (M es la memoria *adicional* requerida).

	`inssort`	`merge`
$T_{\text{peor}}(n)$	$O(n^2)$	$O(n)$
$T_{\text{prom}}(n)$	$O(n^{3/2})$	$O(n)$
$T_{\text{mejor}}(n)$	$O(n)$	$O(n)$
$M_{\text{peor}}(n)$	$O(n)$	$O(n)$
$M_{\text{prom}}(n)$	$O(n)$	$O(\sqrt{n})$
$M_{\text{mejor}}(n)$	$O(n)$	$O(1)$

Tabla 2.4: Tiempo de ejecución para la intercalación de vectores ordenados. T es tiempo de ejecución, M memoria *adicional* requerida.

2.4. El TAD correspondencia

La *"correspondencia"* o *"memoria asociativa"* es un contenedor que almacena la relación entre elementos de un cierto conjunto universal D llamado el *"dominio"* con elementos de otro conjunto universal llamado el *"contradominio"* o *"rango"*. Por ejemplo, la correspondencia $\mathcal{M}$ que va del dominio de los números enteros en sí mismo y transforma un número j en su cuadrado j^2 puede representarse como se muestra en la figura 2.15. Una restricción es que un dado elemento del dominio o bien no debe tener asignado ningún elemento del contradominio o bien debe tener asignado uno solo. Por otra parte, puede ocurrir que a varios elementos del dominio se les asigne un solo elemento del contradominio. En el ejemplo de la figura a los elementos 3 y -3 del dominio les es asignado el mismo elemento 9 del contradominio. A veces también se usa el término *"clave"* (*"key"*) (un poco por analogía con las bases de datos) para refererirse a un valor del dominio y *"valor"* para referirse a los elementos del contradominio.

Las correspondencias son representadas en general guardando internamente los pares de valores y poseen algún algoritmo para asignar valores a claves, en forma análoga a como funcionan las bases de datos. Por eso, en el caso del ejemplo previo $j \to j^2$, es mucho más eficiente representar la correspondencia como una función, ya que es mucho más rápido y no es necesario almacenar todos los valores. Notar que para las representaciones más usuales de enteros, por ejemplo con 32 bits, harían falta varios gigabytes de RAM. El uso de un contenedor tipo correspondencia es útil justamente cuando no es posible calcular el elemento del contradominio a partir del elemento del dominio. Por ejemplo, un tal caso es una correspondencia entre el número de documento de una persona y su nombre. Es imposible de *"calcu-*

lar" el nombre a partir del número de documento, necesariamente hay que almacenarlo internamente en forma de pares de valores.

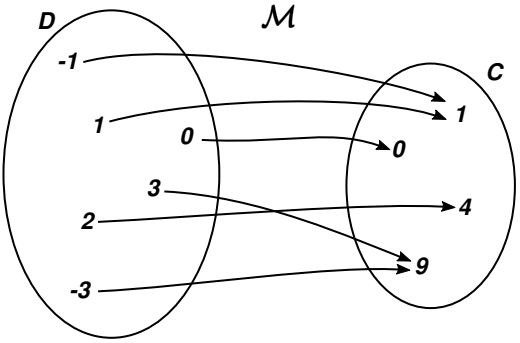


Figura 2.15: Correspodencia entre números enteros $j \rightarrow j^2$

Ejemplo 2.5: *Consigna:* Escribir un programa que memoriza para cada documento de identidad el sueldo de un empleado. Se van ingresando números de documento, si el documento ya tiene un sueldo asignado, entonces esto se reporta por consola, sino el usuario debe entrar un sueldo el cual es asignado a ese número de documento en la tabla. Una posible interacción con el programa puede ser como sigue

```
1. [mstorti@spider aedsrc]$ payroll
2. Ingrese nro. documento > 14203323
3. Ingrese salario mensual: 2000
4. Ingrese nro. documento > 13324435
5. Ingrese salario mensual: 3000
6. Ingrese nro. documento > 13323421
7. Ingrese salario mensual: 2500
8. Ingrese nro. documento > 14203323
9. Doc: 14203323, salario: 2000
10. Ingrese nro. documento > 13323421
11. Doc: 13323421, salario: 2500
12. Ingrese nro. documento > 13242323
13. Ingrese salario mensual: 5000
14. Ingrese nro. documento > 0
15. No se ingresan mas sueldos...
16. [mstorti@spider aedsrc]$
```

Solución: Un posible seudocódigo puede observarse en el código 2.23. El programa entra en un lazo infinito en el cual se ingresa el número de documento y se detiene cuando se ingresa un documento nulo. Reconocemos

las siguientes operaciones abstractas necesarias para manipular correspondencias

- Consultar la correspondencia para saber si una dada clave tiene un valor asignado.
- Asignar un valor a una clave.
- Recuperar el valor asignado a una clave.

```
1.  // declarar 'tabla_sueldos' como 'map' . . .
2.  while(1) {
3.    cout << "Ingrese nro. documento > ";
4.    int doc;
5.    double sueldo;
6.    cin >> doc;
7.    if (!doc) break;
8.    if (/* No tiene 'doc' sueldo asignado?. . . */) {
9.      cout << "Ingrese sueldo mensual: ";
10.     cin >> sueldo;
11.     // Asignar 'doc -> sueldo'
12.     // . . .
13.   } else {
14.     // Reportar el valor almacenado
15.     // en 'tabla_sueldos'
16.     // . . .
17.     cout << "Doc: " << doc << ", sueldo: "
18.          << sueldo << endl;
19.   }
20. }
21. cout << "No se ingresan mas sueldos. . ." << endl;
```

Código 2.23: *Seudocódigo para construir una tabla que representa la correspondencia número de documento → sueldo. [Archivo: payroll4.cpp]*

2.4.1. Interfaz simple para correspondencias

```
1. class iterator_t {/* . . . */};
2.
3. class map {
4.  private:
5.   // . . .
```

```
6.  public:
7.   iterator_t find(domain_t key);
8.   iterator_t insert(domain_t key,range_t val);
9.   range_t& retrieve(domain_t key);
10.  void erase(iterator_t p);
11.  int erase(domain_t key);
12.  domain_t key(iterator_t p);
13.  range_t& value(iterator_t p);
14.  iterator_t begin();
15.  iterator_t next(iterator_t p);
16.  iterator_t end();
17.  void clear();
18.  void print();
19. };
```

Código 2.24: *Interfaz básica para correspondencias. [Archivo: mapbas.h]*

En el código 2.24 vemos una interfaz básica posible para correspondencias. Está basada en la interfaz STL pero, por simplicidad evitamos el uso de clases anidadas para el correspondiente iterator y también evitamos el uso de templates y sobrecarga de operadores. Primero se deben definir (probablemente via **typedef**'s) los tipos que corresponden al domininio (**domain_t**) y al contradominio (**range_t**). Una clase **iterator** (cuyos detalles son irrelevantes para la interfaz) representa las posiciones en la correspondencia. Sin embargo, considerar de que, en contraposición con las listas y a semejanza de los conjuntos, no hay un orden definido entre los pares de la correspondencia. Por otra parte en la correspondencia el iterator *itera sobre los pares de valores* que representan la correspondencia.

En lo que sigue **M** es una correspondencia, **p** es un iterator, **k** una clave, **val** un elemento del contradominio (tipo **range_t**). Los métodos de la clase son

- **p = find(k)**: Dada una clave **k** devuelve un iterator *al par correspondiente* (si existe debe ser único). Si **k** no tiene asignado ningún valor, entonces devuelve **end()**.
- **p = insert(k,val)**: asigna a **k** el valor **val**. Si **k** ya tenía asignado un valor, entonces este nuevo valor reemplaza a aquel en la asignación. Si **k** no tenía asignado ningún valor entonces la nueva asignación es definida. Retorna un iterator al par.

- `val = retrieve(k)`: Recupera el valor asignado a `k`. Si `k` no tiene ningún valor asignado, entonces *inserta una asignación de* k *al valor creado por defecto* para el tipo `range_t` (es decir el que retorna el constructor `range_t()`). *Esto es muy importante y muchas veces es fuente de error.* (Muchos esperan que `retrieve()` de un error en ese caso.) Si queremos recuperar en `val` el valor asignado a `k` *sin insertar accidentalmente una asignación* en el caso que `k` no tenga asignado ningún valor entonces debemos hacer

  ```
  1.  if (M.find(k)!=M.end()) val = M.retrieve(k);
  ```

  `val=M.retrieve(k)` retorna una *referencia* al valor asignado a `k`, de manera que también puede ser usado como miembro izquierdo, es decir, es válido hacer (ver §2.1.4)

  ```
  1.  M.retrieve(k) = val;
  ```

- `k = key(p)` retorna el valor correspondiente a la asignación *apuntada* por `p`.
- `val = value(p)` retorna el valor correspondiente a la asignación *apuntada* por `p`. El valor retornado es una referencia, de manera que también podemos usar `value(p)` como miembro izquierdo (es decir, asignarle un valor como en `value(p)=val`). Notar que, por el contrario, `key(p)` no retorna una referencia.
- `erase(p)`: Elimina la asignación apuntada por `p`. Si queremos eliminar una eventual asignación a la clave `k` entonces debemos hacer

  ```
  1.  p = M.find(k);
  2.  if (p!=M.end()) M.erase(p);
  ```

- `p = begin()`: Retorna un iterator a la primera asignación (en un orden no especificado).
- `p = end()`: Retorna un iterator a una asignación ficticia después de la última (en un orden no especificado).
- `clear()`: Elimina todas las asignaciones.
- `print()`: Imprime toda la tabla de asignaciones.

Con esta interfaz, el programa 2.23 puede completarse como se ve en código 2.25. Notar que el programa es cuidadoso en cuanto a no crear nuevas asignaciones. El `retrieve` de la línea 16 está garantizado que no generará ninguna asignación involuntaria ya que el test del if garantiza que `doc` ya tiene asignado un valor.

```
1.  map sueldo;
```

```
2.  while(1) {
3.    cout << "Ingrese nro. documento > ";
4.    int doc;
5.    double salario;
6.    cin >> doc;
7.    if(!doc) break;
8.    iterator_t q = sueldo.find(doc);
9.    if (q==sueldo.end()) {
10.     cout << "Ingrese salario mensual: ";
11.     cin >> salario;
12.     sueldo.insert(doc,salario);
13.     cout << sueldo.size() << " salarios cargados" << endl;
14.   } else {
15.     cout << "Doc: " << doc << ", salario: "
16.          << sueldo.retrieve(doc) << endl;
17.   }
18. }
19. cout << "No se ingresan mas sueldos. . ." << endl;
```

Código 2.25: *Tabla de sueldos del personal implementado con la interfaz básica de código 2.24. [Archivo: payroll2.cpp]*

2.4.2. Implementación de correspondencias mediante contenedores lineales

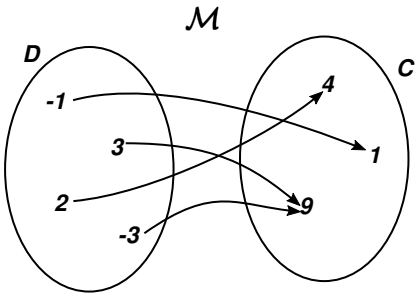


Figura 2.16: Ejemplo de correspondencia.

Tal vez la forma más simple de implementar una correspondencia es guardando en un contenedor todas las asignaciones. Para eso simplemente definimos una clase **elem_t** que simplemente contiene dos campos **first** y **second** con la clave y el valor de la asignación (los nombres de los campos

vienen de la clase **pair** de las STL). Estos pares de elementos (asignaciones) se podrían guardar tanto en un **vector<elem_t>** como en una lista (**list<elem_t>**). Por ejemplo, la correspondecia de enteros a enteros que se muestra en la figura 2.15 se puede representar almacenando los siguientes pares en un contenedor:

$$(-1, 4), (3, 4), (2, 1), (-3, 9) \tag{2.19}$$

A las listas y vectores se les llama contenedores lineales, ya que en ellos existe un ordenamiento natural de las posiciones. En lo que sigue discutiremos la implementación del TAD correspondencia basadas en estos contenedores lineales. Más adelante, en otro capítulo, veremos otras implementaciones más eficientes. Cuando hablemos de la implementación con listas asumiremos una implementación de listas basada en punteros o cursores, mientras que para vectores asumiremos arreglos estándar de C++ o el mismo **vector** de STL. En esta sección asumiremos que las asignaciones son insertadas en el contenedor ya sea al principio o en el final del mismo, de manera que el orden entre las diferentes asignaciones es en principio aleatorio. Más adelante discutiremos el caso en que las asignaciones se mantienen ordenadas por la clave. En ese caso las asignaciones aparecerían en el contenedor como en (2.20).

- **p=find(k)** debe recorrer todas las asignaciones y si encuentra una cuyo campo **first** coincida con **k** entonces debe devolver el iterator correspondiente. Si la clave no tiene ninguna asignación, entonces después de recorrer todas las asignaciones, debe devolver **end()**. El peor caso de **find()** es cuando la clave no está asignada, o la asignación está al final del contenedor, en cuyo caso es $O(n)$ ya que debe recorrer todo el contenedor (n es el número de asignaciones en la correspondencia). Si la clave tiene una asignación, entonces el costo es proporcional a la distancia desde la asignación hasta el origen. En el peor caso esto es $O(n)$, como ya mencionamos, mientras que en el mejor caso, que es cuando la asignación está al comienzo del contenedor, es $O(1)$. La distancia media de la asignación es (si las asignaciones se han ingresado en forma aleatoria) la mitad del número de asociaciones y por lo tanto en promedio el costo será $O(n/2)$.

- **insert()** debe llamar inicalmente a **find()**. Si la clave ya tiene un valor asignado, la inserción es $O(1)$ tanto para listas como vectores. En el caso de vectores, notar que esto se debe a que no es necesario

insertar una nueva asignación. Si la clave no está asignada entonces el elemento se puede insertar al final para vectores, lo cual también es $O(1)$, y en cualquier lugar para listas.

- Un análisis similar indica que `retrieve(k)` también es equivalente a `find()`.
- Para `erase(p)` sí hay diferencias, la implementación por listas es $O(1)$ mientras que la implementación por vectores es $O(n)$ ya que implica mover todos los elementos que están después de la posición eliminada.
- `clear()` es $O(1)$ para vectores, mientras que para listas es $O(n)$.
- Para las restantes funciones el tiempo de ejecución *en el peor caso* es $O(1)$.

Operación	lista	vector
`find(key)`	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
`insert(key,val)`	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
`retrieve(key)`	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
`erase(p)`	$O(1)$	$O(1)/O(n)/O(n)$
`key`, `value`, `begin`, `end`,	$O(1)$	$O(1)$
`clear`	$O(n)$	$O(1)$

Tabla 2.5: Tiempos de ejecución para operaciones sobre correspondencias con contenedores lineales no ordenadas. n es el número de asignaciones en la correspondencia. (La notación es *mejor/promedio/peor*. Si los tres son iguales se reporta uno sólo.)

Estos resultados se sumarizan en la Tabla 2.5. No mostraremos ninguna implementación de correspondencias con contenedores lineales no ordenados ya que discutiremos a continuación la implementación con contenedores ordenados, que es más eficiente.

2.4.3. Implementación mediante contenedores lineales ordenados

Una posibilidad de reducir el tiempo de ejecución es usar contenederos ordenados, es decir ya sea vectores o listas, pero donde los pares de asignación están ordenados de menor a mayor según la clave. Notar que esto

exige que se pueda definir un tal ordenamiento en el conjunto universal del dominio. Si el dominio son los enteros o los reales, entonces se puede usar la relación de orden propia del tipo. Para *"strings"* (cadenas de caracteres) se puede usar el orden *"lexicográfico"*. Para otros tipos compuestos, para los cuales no existe una relación de orden se pueden definir relaciones de orden *ad-hoc* (ver §2.4.4). En algunos casos estas relaciones de orden no tienen ningún otro interés que ser usadas en este tipo de representaciones o en otros algoritmos relacionados.

```
1.  class map;
2.
3.  class elem_t {
4.  private:
5.    friend class map;
6.    domain_t first;
7.    range_t second;
8.  };
9.  // iterator para map va a ser el mismo que para listas.
10. class map {
11. private:
12.   list l;
13.
14.   iterator_t lower_bound(domain_t key) {
15.     iterator_t p = l.begin();
16.     while (p!=l.end()) {
17.       domain_t dom = l.retrieve(p).first;
18.       if (dom >= key) return p;
19.       p = l.next(p);
20.     }
21.     return l.end();
22.   }
23.
24. public:
25.   map() { }
26.   iterator_t find(domain_t key) {
27.     iterator_t p = lower_bound(key);
28.     if (p!=l.end() && l.retrieve(p).first == key)
29.       return p;
30.     else return l.end();
31.   }
32.   iterator_t insert(domain_t key,range_t val) {
33.     iterator_t p = lower_bound(key);
34.     if (p==l.end() || l.retrieve(p).first != key) {
35.       elem_t elem;
```

```
36.       elem.first = key;
37.       p = l.insert(p,elem);
38.     }
39.     l.retrieve(p).second = val;
40.     return p;
41.   }
42.   range_t &retrieve(domain_t key) {
43.     iterator_t q = find(key);
44.     if (q==end()) q=insert(key,range_t());
45.     return l.retrieve(q).second;
46.   }
47.   bool empty() { return l.begin()==l.end(); }
48.   void erase(iterator_t p) { l.erase(p); }
49.   int erase(domain_t key) {
50.     iterator_t p = find(key); int r = 0;
51.     if (p!=end()) { l.erase(p); r = 1; }
52.     return r;
53.   }
54.   iterator_t begin() { return l.begin(); }
55.   iterator_t end() { return l.end(); }
56.   void clear() { l.erase(l.begin(),l.end()); }
57.   int size() { return l.size(); }
58.   domain_t key(iterator_t p) {
59.     return l.retrieve(p).first;
60.   }
61.   range_t &value(iterator_t p) {
62.     return l.retrieve(p).second;
63.   }
64. };
```

Código 2.26: *Implementación de correspondencia mediante listas ordenadas. [Archivo: mapl.h]*

2.4.3.1. Implementación mediante listas ordenadas

En el caso de representar la correspondencia mediante una lista ordenada, los pares son ordenados de acuerdo con su campo clave. Por ejemplo la correspondencia de la figura 2.16, se representaría por una lista como sigue,

$$\mathcal{M} = ((-3, 9), (-1, 4), (2, 1), (3, 4)). \tag{2.20}$$

Ahora `p=find(k)` no debe necesariamente recorrer toda la lista cuando la clave no está, ya que el algoritmo puede detenerse cuando encuentra una

clave *mayor* a la que se busca. Por ejemplo, si hacemos **p=find(0)** en la correspondencia anterior podemos dejar de buscar cuando llegamos al par $(2, 1)$, ya que como los pares están ordenados por clave, todos los pares siguientes deben tener claves mayores que 2, y por lo tanto no pueden ser 0. Sin embargo, al insertar nuevas asignaciones hay que tener en cuenta que no se debe insertar en cualquier posición sino que hay que mantener la lista ordenada. Por ejemplo, si queremos asignar a la clave 0 el valor 7, entonces el par $(0, 7)$ *debe* insertarse entre los pares $(-1, 4)$ y $(2, 1)$, para mantener el orden entre las claves.

Una implementación de la interfaz simplificada mostrada en 2.24 basada en listas ordenadas puede observarse en el código 2.26. Tanto **p=find(key)** como **p=insert(key,val)** se basan en una función auxiliar **p=lower_bound(key)** que retorna la primera posición donde podría insertarse la nueva clave sin violar la condición de ordenamiento sobre el contenedor. Como casos especiales, si todas las claves son mayores que **key** entonces debe retornar **begin()** y si son todas menores, o la correspondencia está vacía, entonces debe retornar **end()**.

La implementación de **p=insert(key,val)** en términos de **p=lower_bound(key)** es simple, **p=lower_bound(key)** retorna un iterator a la asignación correspondiente a **key** (si **key** tiene un valor asignado) o bien un iterator a la posición donde la asignación a **key** debe ser insertada. En el primer caso un nuevo par (de tipo **elem_t**) es construido e insertado usando el método **insert** de listas. Al llegar a la línea 39 la posición **p** apunta al par correspondiente a **key**, independientemente de si este ya existía o si fue creado en el bloque previo.

La implementación de **p=find(key)** en términos de **p=lower_bound(key)** también es simple. Si **p** es **end()** o la asignación correspondiente a **p** contiene *exactamente* la clave **key**, entonces debe retornar **p**. Caso contrario, **p** debe corresponder a una posición dereferenciacible, pero cuya clave no es **key** de manera que en este caso no debe retornar **p** sino **end()**.

El método **val=retrieve(key)** busca una asignación para **key**. Notar que, como mencionamos en §2.4.1 si **key** no está asignado entonces *debe generar una asignación*. El valor correspondiente en ese caso es el que retorna el constructor por defecto (**range_t()**).

Notar que **lower_bound()** es declarado **private** en la clase ya que es sólo un algoritmo interno auxiliar para **insert()** y **find()** .

2.4.3.2. Interfaz compatible con STL

```
1.  template<typename first_t,typename second_t>
2.  class pair {
3.  public:
4.    first_t first;
5.    second_t second;
6.  };
7.
8.  template<typename domain_t,typename range_t>
9.  class map {
10. private:
11.   typedef pair<domain_t,range_t> pair_t;
12.   typedef list<pair_t> list_t;
13.   list_t l;
14.
15. public:
16.   typedef typename list_t::iterator iterator;
17.   map();
18.   iterator find(domain_t key);
19.   range_t & operator[](domain_t key);
20.   bool empty();
21.   void erase(iterator p);
22.   int erase(domain_t key);
23.   iterator begin();
24.   iterator end();
25.   void clear();
26. };
```

Código 2.27: *Versión básica de la interfaz STL para correspondencia. [Archivo: mapstl.h]*

Una versión reducida de la interfaz STL para correspondencia se puede observar en el código 2.27. En el código 2.28 vemos la implementación de esta interfaz mediante listas ordenadas.

```
1.  #ifndef AED_MAP_H
2.  #define AED_MAP_H
3.
4.  #include <aedsrc/list.h>
5.  #include <iostream>
6.
```

```
7.  using namespace std;
8.
9.  namespace aed {
10.
11.   template<typename first_t,typename second_t>
12.   class pair {
13.   public:
14.     first_t first;
15.     second_t second;
16.     pair(first_t f=first_t(),second_t s=second_t())
17.       : first(f), second(s) {}
18.   };
19.
20.   // iterator para map va a ser el mismo que para listas.
21.   template<typename domain_t,typename range_t>
22.   class map {
23.
24.   private:
25.     typedef pair<domain_t,range_t> pair_t;
26.     typedef list<pair_t> list_t;
27.     list_t l;
28.
29.   public:
30.     typedef typename list_t::iterator iterator;
31.
32.   private:
33.     iterator lower_bound(domain_t key) {
34.       iterator p = l.begin();
35.       while (p!=l.end()) {
36.         domain_t dom = p->first;
37.         if (dom >= key) return p;
38.         p++;
39.       }
40.       return l.end();
41.     }
42.
43.   public:
44.     map() { }
45.
46.     iterator find(domain_t key) {
47.       iterator p = lower_bound(key);
48.       if (p!=l.end() && p->first == key)
49.         return p;
50.       else return l.end();
51.     }
52.     range_t & operator[](domain_t key) {
```

```
53.       iterator q = lower_bound(key);
54.       if (q==end() || q->first!=key)
55.         q = l.insert(q,pair_t(key,range_t()));
56.       return q->second;
57.     }
58.     bool empty() { return l.begin()==l.end(); }
59.     void erase(iterator p) { l.erase(p); }
60.     int erase(domain_t key) {
61.       iterator p = find(key);
62.       if (p!=end()) {
63.         l.erase(p);
64.         return 1;
65.       } else {
66.         return 0;
67.       }
68.     }
69.     iterator begin() { return l.begin(); }
70.     iterator end() { return l.end(); }
71.     iterator next(iterator p) { return l.next(p); }
72.     void clear() { l.erase(l.begin(),l.end()); }
73.   };
74. }
75. #endif
```

Código 2.28: *Implementación de correspondencia mediante listas ordenadas con la interfaz STL. [Archivo: map.h]*

- En vez del tipo **elem_t** se define un template **pair<class first_t,class second_t>**. Este template es usado para **map** y otros contenedores y algoritmos de STL. Los campos **first** y **second** de **pair** son públicos. Esto es un caso muy especial dentro de las STL y la programación orientada a objetos en general ya que en general *se desaconseja permitir el acceso a los campos datos de un objeto*. (La motivación para esto es que **pair<>** es una construcción tan simple que se permite violar la regla.) **pair<>** es una forma muy simple de asociar pares de valores en un único objeto. Otro uso de **pair<>** es para permitir que una función retorne dos valores al mismo tiempo. Esto se logra haciendo que retorne un objeto de tipo **pair<>**.
- La clase **map** es un template de las clases **domain_t** y **range_t**. Los elementos de la lista serán de tipo **pair<domain_t,range_t>**.

- Para simplificar la escritura de la clase, se definen dos tipos internos **pair_t** y **list_t**. Los elementos de la lista serán de tipo **pair_t**. También se define el tipo público **iterator** que es igual al iterator sobre la lista, pero esta definición de tipo permitirá verlo también externamente como **map<domain_t,range_t>::iterator**.
- **val=M.retrieve(key)** se reemplaza sobrecargando el operador **[]**, de manera que con esta interfaz la operación anterior se escribe **val=M[key]**. Recordar que tiene el mismo efecto colateral que **retrieve**: *Si* `key` *no tiene ningún valor asignado, entonces* `M[key]` *le asigna uno por defecto.* Además, igual que **retrieve**, **M[key]** retorna una referencia de manera que es válido usarlo como miembro izquierdo, como en **M[key]=val**.

```
1.  map<int,double> sueldo;
2.  while(1) {
3.    cout << "Ingrese nro. documento > ";
4.    int doc;
5.    double salario;
6.    cin >> doc;
7.    if (!doc) break;
8.    map<int,double>::iterator q = sueldo.find(doc);
9.    if (q==sueldo.end()) {
10.     cout << "Ingrese salario mensual: ";
11.     cin >> salario;
12.     sueldo[doc]=salario;
13.   } else {
14.     cout << "Doc: " << doc << ", salario: "
15.          << sueldo[doc] << endl;
16.   }
17. }
18. cout << "No se ingresan mas sueldos..." << endl;
```

Código 2.29: *Tabla de sueldos del personal implementado con la interfaz STL de map (ver código 2.27). [Archivo: payroll3.cpp]*

El programa implementado en el código 2.25 escrito con esta interface puede observarse en el código 2.29.

2.4.3.3. Tiempos de ejecución para listas ordenadas

Consideremos primero **p=lower_bound(k)** *con éxito* (es decir, cuando **k** tiene un valor asignado). El costo es proporcional a la distancia con respecto al origen de la posición donde se encuentra la asignación. Valores de **k** bajos quedarán en las primeras posiciones de la lista y los valores altos al fondo de la misma. En promedio un valor puede estar en cualquier posición de la lista, con lo cual tendrá un costo $O(n/2) = O(n)$. El **p=lower_bound(k)** *sin éxito* (es decir, cuando **k** *no* tiene un valor asignado) en este caso también se detiene cuando llega a un elemento mayor o igual y, usando el mismo razonamiento, también es $O(n/2) = O(n)$. Consecuentemente, si bien hay una ganancia en el caso de listas ordenadas, el *orden* del tiempo de ejecución es prácticamente el mismo que en el caso de listas no ordenadas.

Operación	lista	vector
find(key)	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
M[key] (no existente)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
M[key] (existente)	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
erase(key)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
key, **value**, **begin**, **end**,	$O(1)$	$O(1)$
clear	$O(n)$	$O(1)$

Tabla 2.6: Tiempos de ejecución para operaciones sobre correspondencias con contenedores lineales *ordenados*. n es el número de asignaciones en la correspondencia. (La notación es *mejor/promedio/peor*. Si los tres son iguales se reporta uno sólo.)

find(key) y **M[key]** están basados en **lower_bound** y tienen el mismo tiempo de ejecución ya que para listas las inserciones son $O(1)$. Los tiempos de ejecución para correspondencias por listas ordenadas se sumarizan en la Tabla 2.6.

2.4.3.4. Implementación mediante vectores ordenados

```
1. #ifndef AED_MAPV_H
2. #define AED_MAPV_H
3.
4. #include <iostream>
5. #include <vector>
```

```
6.
7.  using namespace std;
8.
9.  namespace aed {
10.
11.   template<typename first_t,typename second_t>
12.   class pair {
13.   public:
14.     first_t first;
15.     second_t second;
16.     pair() : first(first_t()), second(second_t()) {}
17.   };
18.
19.   // iterator para map va a ser el mismo que para listas.
20.   template<typename domain_t,typename range_t>
21.   class map {
22.
23.   public:
24.     typedef int iterator;
25.
26.   private:
27.     typedef pair<domain_t,range_t> pair_t;
28.     typedef vector<pair_t> vector_t;
29.     vector_t v;
30.
31.     iterator lower_bound(domain_t key) {
32.       int p=0, q=v.size(), r;
33.       if (!q || v[p].first >key) return 0;
34.       while (q-p > 1) {
35.         r = (p+q)/2;
36.         domain_t kr = v[r].first;
37.         if (key > kr) p=r;
38.         else if (key < kr) q=r;
39.         else if (kr==key) return r;
40.       }
41.       if (v[p].first == key) return p;
42.       else return q;
43.     }
44.
45.   public:
46.     map() { }
47.
48.     iterator find(domain_t key) {
49.       int p = lower_bound(key);
50.       if (p == v.size() || v[p].first == key) return p;
51.       else return v.size();
```

```
52.     }
53.     range_t & operator[](domain_t key) {
54.       iterator p = lower_bound(key);
55.       if (p == v.size() || v[p].first != key) {
56.         v.push_back(pair_t());
57.         iterator q = v.size();
58.         while (--q > p) v[q] = v[q-1];
59.         v[p].first = key;
60.       }
61.       return v[p].second;
62.     }
63.     int erase(domain_t key) {
64.       iterator p = find(key); int r = 0;
65.       if (p!=end()) { erase(p); r = 1; }
66.       return r;
67.     }
68.     bool empty() { return v.size()==0; }
69.     void erase(iterator p) {
70.       iterator q = p;
71.       while (q != v.size()) {
72.         v[q] = v[q+1];
73.         q++;
74.       }
75.       v.pop_back();
76.     }
77.     iterator begin() { return 0; }
78.     iterator end() { return v.size(); }
79.     void clear() { v.clear(); }
80.     int size() { return v.size(); }
81.   };
82. }
83. #endif
```

Código 2.30: *Implementación de correspondencia con vectores ordenados. [Archivo: mapv.h]*

La ganancia real de usar contenedores ordenados es en el caso de vectores ya que en ese caso podemos usar el algoritmo de *"búsqueda binaria"* (*"binary search"*) en cuyo caso `p=lower_bound(k)` resulta ser $O(\log n)$ *en el peor caso*. Una implementación de correspondencias con vectores ordenados puede verse en el código 2.30. El código se basa en la clase `vector` de STL, pero en realidad con modificaciones menores se podrían reemplazar

los vectores de STL por arreglos estándar de C. La correspondencia almacena las asignaciones (de tipo **pair_t**) en un **vector<pair_t> v**. Para el tipo **map<>::iterator** usamos directamente el tipo entero.

Veamos en detalle el algoritmo de búsqueda binaria en **lower_bound()**. El algoritmo se basa en ir refinando un rango $[p, q)$ tal que la clave buscada esté garantizado siempre en el rango, es decir $k_p \leq k < k_q$, donde k_p, k_q son las claves en las posiciones p y q respectivamente y k es la clave buscada. Las posiciones en el vector como p, q se representan por enteros comenzando de 0. La posición **end()** en este caso coincide con el entero **size()** (el tamaño del vector). En el caso en que $q =$**size()** entonces asumimos que la clave correspondiente es $k_q = \infty$, es decir más grande que todas las claves posibles.

Si el vector está vacío, entonces **lower_bound** retorna 0 ya que entonces debe insertar en **end()** que en ese caso vale 0. Lo mismo si $k < k_0$, ya que en ese caso la clave va en la primera posición. Si ninguno de estos casos se aplica, entonces el rango $[p, q)$, con $p = 0$ y $m =$**v.size()** es un rango válido, es decir $k_p \leq k < k_q$.

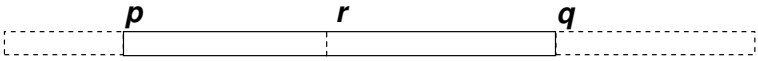


Figura 2.17: Refinando el rango de búsqueda en el algoritmo de búsqueda binaria.

Una vez que tenemos un rango válido $[p, q)$ podemos *refinarlo* (ver figura 2.17) calculando la posición media

$$r = \text{floor}((p + q)/2) \tag{2.21}$$

donde $\text{floor}(x)$ es la parte entera de x ($\text{floor}()$ es parte de la librería **libc**, estándar de C). Esto divide $[p, q)$ en dos subrangos disjuntos $[p, r)$ y $[r, q)$. y comparando la clave **k** con la que está almacenado en la posición r, k_r. Si $k >= k_r$ entonces el nuevo rango es el $[r, q)$, mientras que si no es el $[p, r)$. Notar que si el rango $[p, q)$ es de longitud par, esto es $n = q - p$ es par, entonces los dos nuevos rangos son iguales, de longitud igual a la mitad $n = (q - p)/2$. Si no, uno es de longitud $n = \text{floor}((q - p)/2)$ y el otro de longitud $n = \text{floor}((q - p)/2) + 1$. Ahora bien, a menos que $n = 1$, esto implica que en cada refinamiento la longitud del rango se reduce estrictamente, de manera que en a lo sumo en n pasos la longitud se reduce a longitud 1, en cuyo caso el algoritmo se detiene. En ese caso o bien la

clave buscada esta en **p**, en cuyo caso **lower_bound** retorna **p**, o bien la clave no está y el punto de inserción es **q**.

p=find(k) se implementa fácilmente en términos de **lower_bound()**. Básicamente es idéntica al **find()** de listas en el código 2.26. Por otra parte **operator[]** es un poco más complicado, ya que si la búsqueda no es exitosa (es decir, si k no tiene ningun valor asignado) hay que desplazar todas las asignaciones una posición hacia el final para hacer lugar para la nueva asignación (el lazo de las líneas 55–60).

2.4.3.5. Tiempos de ejecución para vectores ordenados

Ahora estimemos mejor el número de refinamientos. Si el número de elementos es inicialmente una potencia de 2, digamos $n = 2^m$, entonces despúes del primer refinamiento la longitud será 2^{m-1}, después de dos refinamientos 2^{m-2}, hasta que, después de m refinamientos la longitud se reduce a $2^0 = 1$ y el algoritmo se detiene. De manera que el número de refinamientos es $m = \log_2 n$. Puede verse que, si n no es una potencia de dos, entonces el número de refinamientos es $m = \mathrm{floor}(\log_2 n) + 1$. Pero el tiempo de ejecución de **lower_bound** es proporcional al número de veces que se ejecuta el lazo de refinamiento, de manera que el costo de **lower_bound()** es en *el peor caso* $O(\log n)$. Esto representa una significante reducción con respecto al $O(n)$ que teníamos con las listas. Notar de paso que el algoritmo de búsqueda binaria no se puede aplicar a listas ya que estas no son contenedores de *"acceso aleatorio"*. Para vectores, acceder al elemento j-ésimo es una operación $O(1)$, mientras que para listas involucra recorrer toda la lista desde el comienzo hasta la posición entera j, lo cual es $O(j)$.

Como **p=find(k)** está basado en **lower_bound()** el costo de éste es también $O(\log n)$. Para **M[k]** tenemos el costo de **lower_bound()** por un lado pero también tenemos el lazo de las líneas 55–60 para desplazar las asignaciones, el cual es $O(n)$. Los resultados se sumarizan en la Tabla 2.6. Para $M[key]$ hemos separado los casos en que la clave era ya existente de cuando no existía. Notar la implementación por vectores ordenados es óptima en el caso de no necesitar eliminar asignaciones o insertar nuevas.

2.4.4. Definición de una relación de orden

Para implementar la correspondencia con contenedores ordenados es necesario contar con una relación de orden en conjunto universal de las claves. Para los tipos númericos básicos se puede usar el orden usual y

para las cadenas de caracteres el orden lexicográfico (alfabético). Para otros conjuntos universales como por ejemplo el conjunto de los pares de enteros la definición de una tal relación de orden puede no ser trivial. Sería natural asumir que

$$(2,3) < (5,6) \tag{2.22}$$

ya que cada uno de las componentes del primer par es menor que la del segundo par, pero no sabríamos como comparar $(2,3)$ con $(5,1)$. Primero definamos más precisamente qué es una *"relación de orden"*.
Definición: *"$<$" es una relación de orden en el conjunto C si,*

1. *$<$ es transitiva, es decir, si $a < b$ y $b < c$, entonces $a < c$.*

2. *Dados dos elementos cualquiera de C, una y sólo una de las siguientes afirmaciones es válida:*

   - $a < b$,
   - $b < a$
   - $a = b$.

Una posibilidad sería en comparar ciertas funciones escalares del par, como la suma, o la suma de los cuadrados. Por ejemplo definir que $(a,b) < (c,d)$ si y sólo sí $(a+b) < (c+d)$. Una tal definición satisface 1, pero no 2, ya que por ejemplo los pares $(2,3)$ y $(1,4)$ no satisfacen ninguna de las tres condiciones.

Notar que una vez que se define un operador $<$, los operadores $\leq$, $>$ y $\geq$ se pueden definir fácilmente en términos de $<$.

Una posibilidad para el conjunto de pares de enteros es la siguiente $(a,b) < (c,d)$ si $a < c$ o $a = c$ y $b < d$. Notar, que esta definición es equivalente a la definición lexicográfica para pares de letras si usamos el orden alfabético para comparar las letras individuales.

Probemos ahora la transitividad de esta relación. Sean $(a,b) < (c,d)$ y $(c,d) < (e,f)$, entonces hay cuatro posibilidades

- $a < c < e$
- $a < c = e$ y $d < f$
- $a = c < e$ y $b < d$
- $a = c = e$ y $b < d < f$

y es obvio que en cada una de ellas resulta ser $(a,b) < (e,f)$. También es fácil demostrar la condición 2.

Notar que esta relación de orden puede extenderse a cualquier conjunto universal compuesto de pares de conjuntos los cuales individualmente tienen una relación de orden, por ejemplo pares de la forma *(doble,entero)* o *(entero,string)*. A su vez, aplicando recursivamente el razonamiento podemos ordenar n-tuplas de elementos que pertenezcan cada uno de ellos a conjuntos ordenados.

Capítulo 3

Arboles

Los árboles son contenedores que permiten organizar un conjunto de objetos en forma jerárquica. Ejemplos típicos son los diagramas de organización de las empresas o instituciones y la estructura de un sistema de archivos en una computadora. Los árboles sirven para representar fórmulas, la descomposición de grandes sistemas en sistemas más pequeños en forma recursiva y aparecen en forma sistemática en muchísimas aplicaciones de la computación científica. Una de las propiedades más llamativas de los árboles es la capacidad de acceder a muchísimos objetos desde un punto de partida o raíz en unos pocos pasos. Por ejemplo, en mi cuenta poseo unos 61,000 archivos organizados en unos 3500 directorios a los cuales puedo acceder con un máximo de 10 cambios de directorio (en promedio unos 5).

Sorprendentemente *no existe un contenedor STL de tipo árbol*, si bien varios de los otros contenedores (como conjuntos y correspondencias) están implementados internamente en términos de árboles. Esto se debe a que en la filosofía de las STL el árbol es considerado o bien como un subtipo del grafo o bien como una entidad demasiado básica para ser utilizada directamente por los usuarios.

3.1. Nomenclatura básica de árboles

Un árbol es una colección de elementos llamados *"nodos"*, uno de los cuales es la *"raíz"*. Existe una relación de parentesco por la cual cada nodo tiene un y sólo un *"padre"*, salvo la raíz que no lo tiene. El nodo es el concepto análogo al de *"posición"* en la lista, es decir un objeto abstracto que representa una posición en el mismo, no directamente relacionado con

el *"elemento"* o *"etiqueta"* del nodo. Formalmente, el árbol se puede definir recursivamente de la siguiente forma (ver figura 3.1)

- Un nodo sólo es un árbol
- Si n es un nodo y T_1, T_2, ..., T_k son árboles con raíces n_1, ..., n_k entonces podemos construir un nuevo árbol que tiene a n como raíz y donde n_1, ..., n_k son *"hijos"* de n.

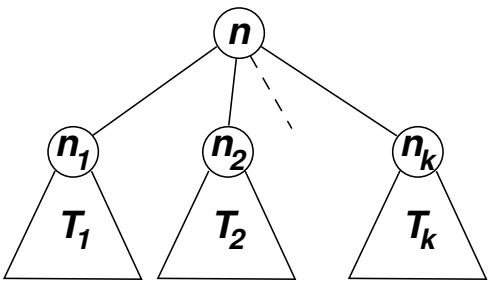


Figura 3.1: Construcción recursiva de un árbol

También es conveniente postular la existencia de un *"árbol vacío"* que llamaremos Λ.

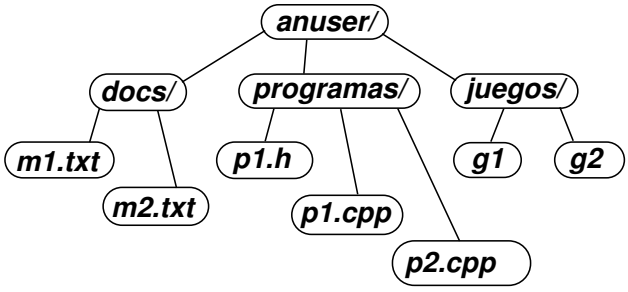


Figura 3.2: Arboles representando un sistema de archivos

Ejemplo 3.1: Consideremos el árbol que representa los archivos en un sistema de archivos. Los nodos del árbol pueden ser directorios o archivos. En el ejemplo de la figura 3.2, la cuenta `anuser/` contiene 3 subdirectorios `docs/`, `programas/` y `juegos/`, los cuales a su vez contienen una serie de archivos. En este caso la relación entre nodos hijos y padres corresponde a la de pertenencia: un nodo a es hijo de otro b, si el archivo a *pertenece* al

directorio b. En otras aplicaciones la relación padre/hijo puede representar querer significar otra cosa.

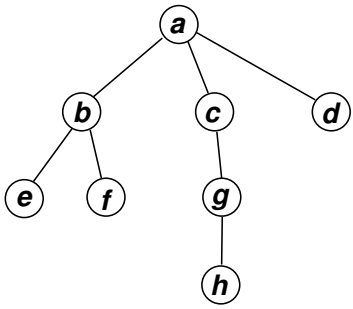


Figura 3.3: Ejemplo simple de árbol

Camino. Si $n_1, n_2, \ldots, n_k$ es una secuencia de nodos tales que n_i es padre de n_{i+1} para $i = 1 \ldots k-1$, entonces decimos que esta secuencia de nodos es un *"camino"* (*"path"*), de manera que $\{$**anuser**, **docs**, **m2.txt**$\}$ es un camino, mientras que $\{$**docs**, **anuser**, **programas**$\}$ no. (Coincidentemente en Unix se llama camino a la especificación completa de directorios que va desde el directorio raíz hasta un archivo, por ejemplo el camino correspondiente a **m2.txt** es **/anuser/docs/m2.txt**.) La *"longitud"* de un camino es igual al número de nodos en el camino menos uno, por ejemplo la longitud del camino $\{$**anuser**, **docs**, **m2.txt**$\}$ es 2. Notar que siempre existe un camino de longitud 0 de un nodo a sí mismo.

Descendientes y antecesores. Si existe un camino que va del nodo a al b entonces decimos que a es antecesor de b y b es descendiente de a. Por ejemplo **m1.txt** es descendiente de **anuser** y **juegos** es antecesor de **g2**. Estrictamente hablando, un nodo es antecesor y descendiente de sí mismo ya que existe camino de longitud 0. Para diferenciar este caso trivial, decimos que a es descendiente (antecesor) propio de b si a es descendiente (antecesor) de b, *pero* $a \neq b$. En el ejemplo de la figura 3.3 a es antecesor propio de c, f y d,

Hojas. Un nodo que no tiene hijos es una *"hoja"* del árbol. (Recordemos que, por contraposición el nodo que no tiene padre es único y es la raíz.) En el ejemplo, los nodos e, f, h y d son hojas.

3.1.0.0.1. Altura de un nodo. La altura de un nodo en un árbol es la máxima longitud de un camino que va desde el nodo a una hoja. Por ejemplo, el árbol de la figura la altura del nodo c es 2. La altura del árbol es la altura de la raíz. La altura del árbol del ejemplo es 3. Notar que, para cualquier nodo n

$$\text{altura}(n) = \begin{cases} 0; & \text{si } n \text{ es una hoja} \\ 1 + \max_{s=\text{hijo de } n} \text{altura}(s); & \text{si no lo es.} \end{cases} \tag{3.1}$$

3.1.0.0.2. Profundidad de un nodo. Nivel. La *"profundidad"* de un nodo es la longitud de único camino que va desde el nodo a la raíz. La profundidad del nodo g en el ejemplo es 2. Un *"nivel"* en el árbol es el conjunto de todos los nodos que están a una misma profundidad. El nivel de profundidad 2 en el ejemplo consta de los nodos e, f y g.

3.1.0.0.3. Nodos hermanos Se dice que los nodos que tienen un mismo padre son *"hermanos"* entre sí. Notar que no basta con que dos nodos estén en el mismo nivel para que sean hermanos. Los nodos f y g en el árbol de la figura 3.3 están en el mismo nivel, pero no son hermanos entre sí.

3.2. Orden de los nodos

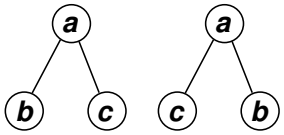


Figura 3.4: Arboles ordenados: el orden de los hijos es importante de manera que los árboles de la figura son diferentes.

En este capítulo, estudiamos árboles para los cuales el *orden* entre los hermanos es relevante. Es decir, los árboles de la figura 3.4 *son diferentes* ya que si bien a tiene los mismos hijos, están en diferente orden. Volviendo a la figura 3.3 decimos que el nodo c está a la derecha de b, o también que c es es el hermano derecho de b. También decimos que b es el *"hijo más a la izquierda"* de a. El orden entre los hermanos se propaga a los hijos, de manera que h está a la derecha de e ya que ambos son descendientes

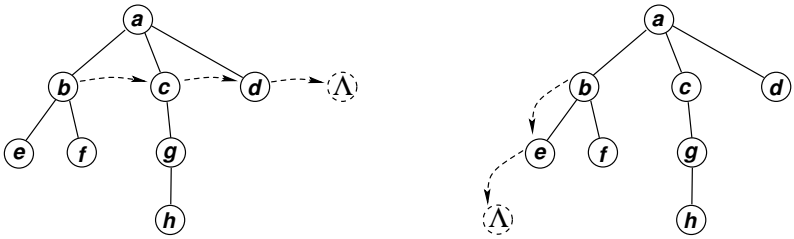


Figura 3.5: Direcciones posibles para avanzar en un árbol.

de c y b, respectivamente. A estos árboles se les llama *"árboles ordenados orientados"* (AOO).

Podemos pensar al árbol como una lista bidimensional. Así como en las listas se puede avanzar linealmente desde el comienzo hacia el fin, en cada nodo del árbol podemos avanzar en dos direcciones (ver figura 3.5)

- Por el hermano derecho, de esta forma se recorre toda la lista de hermanos de izquierda a derecha.

- Por el hijo más izquierdo, tratando de descender lo más posible en profundidad.

En el primer caso el recorrido termina en el último hermano a la derecha. Por analogía con la posición **end()** en las listas, asumiremos que después del último hermano existe un nodo ficticio no dereferenciable. Igualmente, cuando avanzamos por el hijo más izquierdo, el recorrido termina cuando nos encontramos con una hoja. También asumiremos que el hijo más izquierdo de una hoja es un nodo ficticio no dereferenciable. Notar que, a diferencia de la lista donde hay una sola posición no dereferenciable (la posición **end()**), en el caso de los árboles puede haber más de una posiciones ficticias no dereferenciables, las cuales simbolizaremos con Λ cuando dibujamos el árbol. En la figura 3.6 vemos todas las posibles posiciones ficticias $\Lambda_1, \ldots, \Lambda_8$ para el árbol de la figura 3.5. Por ejemplo, el nodo f no tiene hijos, de manera que genera la posición ficticia Λ_2.Tampoco tiene hermano derecho, de manera que genera la posición ficticia Λ_3.

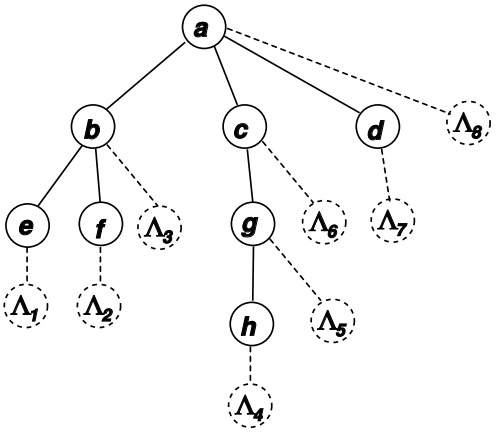


Figura 3.6: Todas las posiciones no dereferenciables de un árbol.

3.2.1. Particionamiento del conjunto de nodos

Ahora bien, dados dos nodos cualquiera m y n consideremos sus caminos a la raíz. Si m es descendiente de n entonces el camino de n está incluido en el de m o viceversa. Por ejemplo, el camino de c, que es a, c, está incluido en el de h, a, c, g, h, ya que c es antecesor de h. Si entre m y n no hay relación de descendiente o antecesor, entonces los caminos se deben bifurcar necesariamente en un cierto nivel. *El orden entre m y n es el orden entre los antecesores a ese nivel*. Esto demuestra que, dados dos nodos cualquiera m y n sólo una de las siguientes afirmaciones puede ser cierta

- $m = n$
- m es antecesor propio de n
- n es antecesor propio de m
- m está a la derecha de n
- n está a la derecha de m

Dicho de otra manera, dado un nodo n el conjunto N de todos los nodos del árbol se puede dividir en 5 conjuntos *disjuntos* a saber

$$N = \{n\} \cup \{\text{descendientes}(n)\} \cup \{\text{antecesores}(n)\} \cup \{\text{derecha}(n)\} \cup \{\text{izquierda}(n)\} \tag{3.2}$$

En la figura 3.7 vemos la partición inducida para los nodos c y f. Notar que en el caso del nodo f el conjunto de los descendientes es vacío ($\emptyset$).

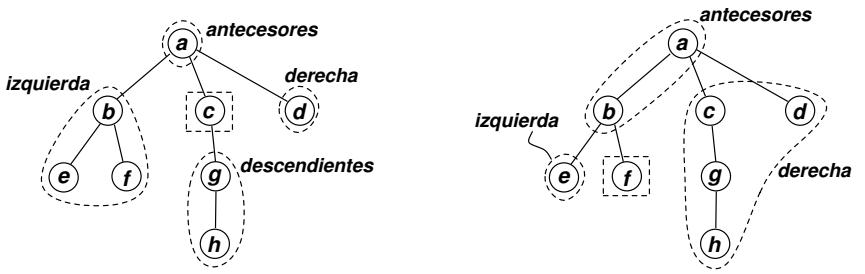


Figura 3.7: Clasificación de los nodos de un árbol con respecto a un nodo. Izquierda: con respecto al nodo c. Derecha: con respecto al nodo f

3.2.2. Listado de los nodos de un árbol

3.2.2.1. Orden previo

Existen varias formas de recorrer un árbol listando los nodos del mismo, generando una lista de nodos. Dado un nodo n con hijos n_1, n_2, $\ldots n_m$, el *"listado en orden previo"* (*"preorder"*) del nodo n que denotaremos como $\mathrm{oprev}(n)$ se puede definir recursivamente como sigue

$$\mathrm{oprev}(n) = (n, \mathrm{oprev}(n_1), \mathrm{oprev}(n_2), \ldots, \mathrm{oprev}(n_m)) \tag{3.3}$$

Además el orden previo del árbol vacío es la lista vacía: $\mathrm{oprev}(\Lambda) = ()$.

Consideremos por ejemplo el árbol de la figura 3.3. Aplicando recursivamente (3.3) tenemos

$$\begin{aligned}
\mathrm{oprev}(a) &= a, \mathrm{oprev}(b), \mathrm{oprev}(c), \mathrm{oprev}(d) \\
&= a, b, \mathrm{oprev}(e), \mathrm{oprev}(f), c, \mathrm{oprev}(g), d \\
&= a, b, \mathrm{oprev}(e), \mathrm{oprev}(f), c, \mathrm{oprev}(g), d \\
&= a, b, e, f, c, g, \mathrm{oprev}(h), d \\
&= a, b, e, f, c, g, h, d
\end{aligned} \tag{3.4}$$

Una forma más visual de obtener el listado en orden previo es como se muestra en la figura 3.8. Recorremos el borde del árbol en el sentido contrario a las agujas del reloj, partiendo de un punto imaginario a la izquierda del nodo raíz y terminando en otro a la derecha del mismo, como muestra la línea de puntos. Dado un nodo como el b el camino pasa cerca de él en varios puntos (3 en el caso de b, marcados con pequeños números en el camino). El orden previo consiste en *listar los nodos una sola vez, la primera vez que el camino*

pasa cerca del árbol. Así en el caso del nodo b, este se lista al pasar por 1. Queda como ejercicio para el lector verificar el orden resultante coincide con el dado en (3.4).

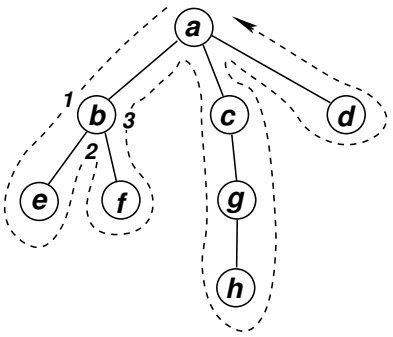


Figura 3.8: Recorrido de los nodos de un árbol en orden previo.

3.2.2.2. Orden posterior

El *"orden posterior"* (*"postorder"*) se puede definir en forma análoga al orden previo pero reemplazando (3.3) por

$$\operatorname{opost}(n) = (\operatorname{opost}(n_1), \operatorname{opost}(n_2), \ldots, \operatorname{opost}(n_m), n) \tag{3.5}$$

y para el árbol del ejemplo resulta ser

$$\begin{aligned} \operatorname{opost}(a) &= \operatorname{opost}(b), \operatorname{opost}(c), \operatorname{opost}(d), a \\ &= \operatorname{opost}(e), \operatorname{opost}(f), b, \operatorname{opost}(g), c, d, a \\ &= e, f, b, \operatorname{opost}(h), g, c, d, a \\ &= e, f, b, h, g, c, d, a \end{aligned} \tag{3.6}$$

Visualmente se puede realizar de dos maneras.

- Recorriendo el borde del árbol igual que antes (esto es en sentido contrario a las agujas del reloj), listando el nodo *la última vez que el recorrido pasa por al lado del mismo*. Por ejemplo el nodo b sería listado al pasar por el punto 3.

- Recorriendo el borde en el sentido opuesto (es decir en el mismo sentido que las agujas del reloj), y listando los nodos la *primera vez* que el

camino pasa cera de ellos. Una vez que la lista es obtenida, *invertimos la lista*. En el caso de la figura el recorrido en sentido contrario daría (a, d, c, g, h, b, f, e). Al invertirlo queda como en (3.6).

Existe otro orden que se llama *"simétrico"*, pero este sólo tiene sentido en el caso de árboles binarios, así que no será explicado aquí.

3.2.2.3. Orden posterior y la notación polaca invertida

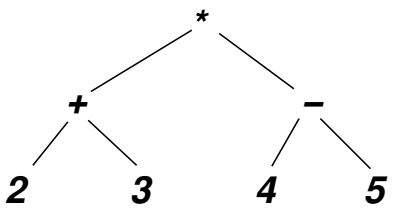


Figura 3.9: Arbol correspondiente a la expresión matemática $(2+3)*(4-5)$

Las expresiones matemáticas como $(2 + 4) * (4 - 5)$ se pueden poner en forma de árbol como se muestra en la figura 3.9. La regla es

- Para operadores binarios de la forma $a + b$ se pone el operador $(+)$ como padre de los dos operandos (a y b). Los operandos pueden ser a su vez expresiones. Funciones binarias como $rem(10, 5)$ (rem es la función resto) se tratan de esta misma forma.

- Operadores unarios (como -3) y funciones (como $\sin(20)$) se escriben poniendo el operando como hijo del operador o función.

- Operadores asociativos con más de dos operandos (como $1+3+4+9$) deben asociarse de a 2 (como en $(((1 + 3) + 4) + 9)$).

De esta forma, expresiones complejas como

$$(3 + \sin(4 + 20) * (5 - e^3)) * (20 + 10 - 7) \tag{3.7}$$

pueden ponerse en forma de árbol, como en la figura 3.10.

El listado en orden posterior de este árbol coincide con la notación polaca invertida (RPN) discutida en la sección §2.2.1.

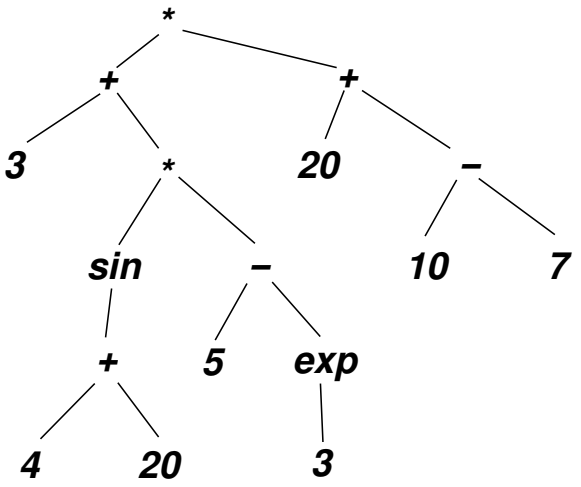


Figura 3.10: Arbol correspondiente a la expresión matemática (3.7)

3.2.3. Notación Lisp para árboles

Una expresión matemática compleja que involucra funciones cuyos argumentos son a su vez llamadas a otras funciones puede ponerse en forma de árbol. Por ejemplo, para la expresión

$$f(g(a,b),h(t,u,v),q(r,s(w))) \tag{3.8}$$

corresponde un árbol como el de la figura 3.11. En este caso cada función es un nodo cuyos hijos son los argumentos de la función. En Lisp la llamada a un función $f(x,y,z)$ se escribe de la forma `(f x y z)`, de manera que la llamada anterior se escribiría como

```
1.  (f (g a b) (h t u v) (q r (s w)))
```

Para expresiones más complejas como la de (3.7), la forma Lisp para el árbol (figura 3.10) da el código Lisp correspondiente

```
1.  (* (+ 3 (* (sin (+ 4 20)) (- 5 (exp 3)))) (+ 20 (- 10 7)))
```

Esta notación puede usarse para representar árboles en forma general, de manera que, por ejemplo, el árbol de la figura 3.3 puede ponerse, en notación Lisp como

```
1. (a (b e f) (c (g h)) d)
```

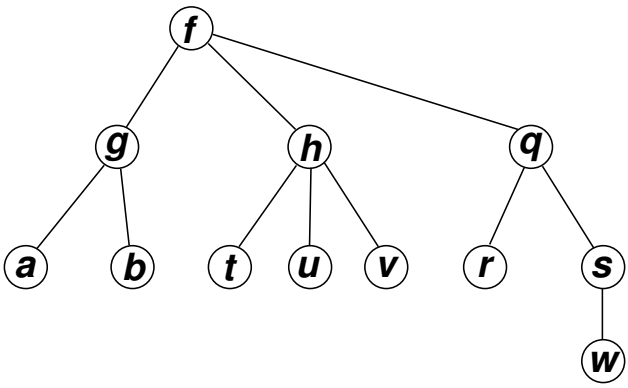


Figura 3.11: Arbol correspondiente a una expresión de llamadas a funciones.

Notemos que el orden de los nodos es igual al del orden previo. Se puede dar una definición precisa de la notación Lisp como para el caso de los órdenes previo y posterior:

$$\mathrm{lisp}(n) = \begin{cases} \text{si } n \text{ es una hoja:} & n \\ \text{caso contrario:} & (n\ \mathrm{lisp}(n_1)\ \mathrm{lisp}(n_2)\ \ldots\ \mathrm{lisp}(n_m)) \end{cases} \tag{3.9}$$

donde $n_1 \ldots n_m$ son los hijos del nodo n.

Es evidente que existe una relación unívoca entre un árbol y su notación Lisp. Los paréntesis dan la estructura adicional que permite establecer la relación unívoca. La utilidad de esta notación es que permite fácilmente escribir árboles en una línea de texto, sin tener que recurrir a un gráfico. Basado en esta notación, es fácil escribir una función que convierta un árbol a una lista y viceversa.

También permite *"serializar"* un árbol, es decir, convertir una estructura *"bidimensional"* como es el árbol, en una estructura unidimensional como es una lista. El serializar una estructura compleja permite almcenarla en disco o comunicarla a otro proceso por mensajes.

3.2.4. Reconstrucción del árbol a partir de sus órdenes

Podemos preguntarnos si podemos reconstruir un árbol a partir de su listado en orden previo. Si tal cosa fuera posible, entonces sería fácil representar árboles en una computadora, almacenando dicha lista. Sin embargo puede verse fácilmente que árboles distintos pueden dar el mismo orden previo (ver figura 3.12) o posterior (ver figura 3.13).

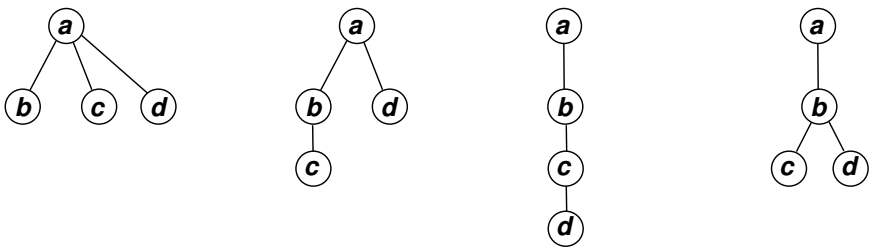


Figura 3.12: Los cuatro árboles de la figura tienen el mismo orden previo (a, b, c, d)

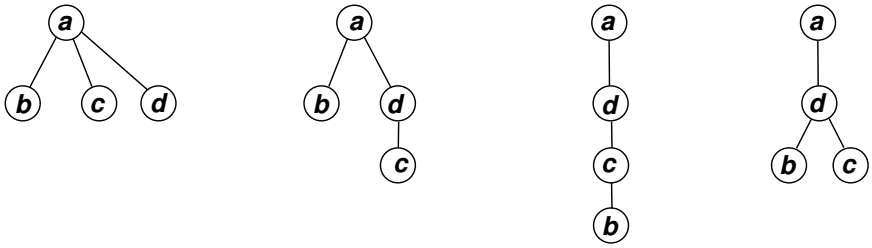


Figura 3.13: Los cuatro árboles de la figura tienen el mismo orden posterior (b, c, d, a)

Sin embargo, es destacable que, dado el orden previo y posterior de un árbol sí se puede reconstruir el árbol. Primero notemos que (3.3) implica que el orden de los nodos queda así

$$\mathrm{oprev}(n) = (n, n_1, \text{descendientes}(n_1), n_2, \text{descendientes}(n_2), \ldots, n_m, \text{descendientes}(r \tag{3.10}$$

mientras que

$$\mathrm{opost}(n) = (\text{descendientes}(n_1), n_1, \text{descendientes}(n_2), n_2, \ldots, \text{descendientes}(n_m), n_m \tag{3.11}$$

Notemos que el primer nodo listado en orden previo es la raíz, y el segundo su primer hijo n_1. Todos los nodos que están *después* de n_1 en orden previo pero *antes* de n_1 en orden posterior son los descendientes de n_1. Prestar atención a que el orden en que aparecen los descendientes de un dado nodo en (3.10) puede no coincidir con el que aparecen en (3.11). De esta forma podemos deducir cuales son los descendientes de n_1. El nodo siguiente, en orden previo, a todos los descendientes de n_1 debe ser el segundo hijo n_2. Todos los nodos que están después de n_2 en orden previo pero antes de n_2

en orden posterior son descendientes de n_2. Así siguiendo podemos deducir cuales son los hijos de n y cuales son descendientes de cada uno de ellos.
Ejemplo 3.2: *Consigna:* Encontrar el árbol A tal que

$$\begin{aligned} \text{orden previo} &= (z, w, a, x, y, c, m, t, u, v) \\ \text{orden posterior} &= (w, x, y, a, t, u, v, m, c, z) \end{aligned} \tag{3.12}$$

Solución: De los primeros dos nodos en orden previo se deduce que z debe ser el nodo raíz y w su primer hijo. No hay nodos antes de w en orden posterior de manera que w no tiene hijos. El nodo siguiente a w en orden previo es a que por lo tanto debe ser el segundo hijo de z. Los nodos que están antes de a pero después de w en orden posterior son x e y, de manera que estos son descendientes de a. De la misma forma se deduce que el tercer hijo de z es c y que sus descendientes son m, t, u, v. A esta altura podemos esbozar un dibujo del árbol como se muestra en la figura 3.14. Las líneas de puntos indican que, por ejemplo, sabemos que m, t, u, v son descendientes de c, pero todavía no conocemos la estructura de ese subárbol.

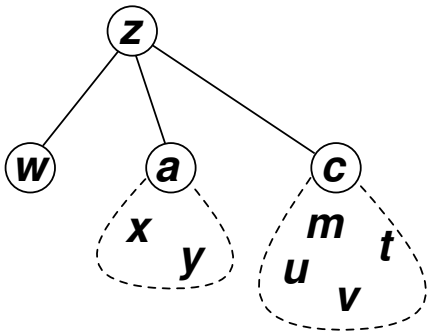


Figura 3.14: Etapa parcial en la reconstrucción del árbol del ejemplo 3.2

Ahora bien, para hallar la estructura de los descendientes de c volvemos a (3.12), y vemos que

$$\begin{aligned} \text{oprev}(c) &= (c, m, t, u, v) \\ \text{opost}(c) &= (t, u, v, m, c) \end{aligned} \tag{3.13}$$

de manera que el procedimiento se puede aplicar recursivamente para hallar los hijos de c y sus descendientes y así siguiendo hasta reconstruir todo el árbol. El árbol correspondiente resulta ser, en este caso el de la figura 3.15, o en notación Lisp `(z w (a x y) (c (m t u v)))`.

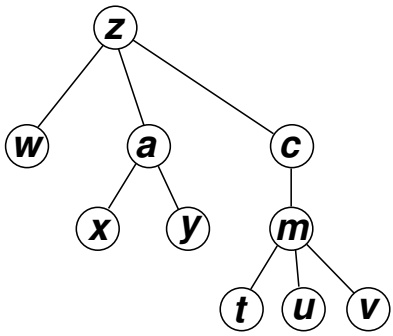


Figura 3.15: Arbol reconstruido a partir de los órdenes previo y posterior especificados en (3.12)

3.3. Operaciones con árboles

3.3.1. Algoritmos para listar nodos

Implementar un algoritmo para recorrer los nodos de un árbol es relativamente simple debido a su naturaleza intrínsecamente recursiva, expresada en (3.3). Un posible algoritmo puede observarse en el código 3.1. Si bien el algoritmo es genérico hemos usado ya algunos conceptos familiares de las STL, por ejemplo las posiciones se representan con una clase `iterator`. Recordar que para árboles se puede llegar al *"fin del contenedor"*, es decir los nodos Λ, en más de un punto del contenedor. El código genera una lista de elementos `L` con los elementos de `T` en orden previo.

```
1. void preorder(tree &T,iterator n,list &L) {
2.   L.insert(L.end(),/* valor en el nodo 'n'... */);
3.   iterator c = /* hijo mas izquierdo de n... */;
4.   while (/* 'c' no es 'Lambda'... */) {
5.     preorder(T,c,L);
6.     c = /* hermano a la derecha de c... */;
7.   }
8. }
```

Código 3.1: *Algoritmo para recorrer un árbol en orden previo. [Archivo: preorder.cpp]*

```
1.  void postorder(tree &T,iterator n,list &L) {
2.    iterator c = /* hijo mas izquierdo de n . . . */;
3.    while (c != T.end()) {
4.      postorder(T,c,L);
5.      c = /* hermano a la derecha de c . . . */;
6.    }
7.    L.insert(L.end(),/* valor en el nodo 'n'. . . */);
8.  }
```

Código 3.2: *Algoritmo para recorrer un árbol en orden posterior. [Archivo: postorder.cpp]*

```
1.  void lisp_print(tree &T,iterator n) {
2.    iterator c = /* hijo mas izquierdo de n . . . */;
3.    if (/* 'c' es 'Lambda'. . . */) {
4.      cout << /* valor en el nodo 'n'. . . */;
5.    } else {
6.      cout << "(" << /* valor de 'n' . . . */;
7.      while (/* 'c' no es 'Lambda'. . . */) {
8.        cout << " ";
9.        lisp_print(T,c);
10.       c = /* hermano derecho de c . . . */;
11.     }
12.     cout << ")";
13.   }
14. }
```

Código 3.3: *Algoritmo para imprimir los datos de un árbol en notación Lisp. [Archivo: lispprint.cpp]*

En el código 3.2 se puede ver un código similar para generar la lista con el orden posterior, basada en (3.5). Similarmente, en código 3.3 puede verse la implementación de una rutina que imprime la notación Lisp de un árbol.

3.3.2. Inserción en árboles

Para construir árboles necesitaremos rutinas de inserción supresión de nodos. Como en las listas, las operaciones de inserción toman un elemento y una posición e insertan el elemento en esa posición en el árbol.

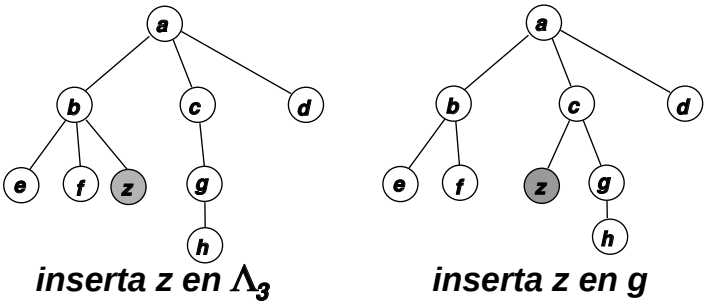


Figura 3.16: Resultado de insertar el elemento z en el árbol de la figura 3.6. *Izquierda:* Inserta z en la posición Λ_3. *Derecha:* Inserta z en la posición g.

- Cuando insertamos un nodo en una posición Λ entonces simplemente el elemento pasa a generar un nuevo nodo en donde estaba el nodo ficticio Λ. Por ejemplo, el resultado de insertar el elemento z en el nodo Λ_3 de la figura 3.6 se puede observar en la figura 3.16 (izquierda). (*Observación:* En un abuso de notación estamos usando las mismas letras para denotar el contenido del nodo que el nodo en sí.)

- Cuando insertamos un nodo en una posición dereferenciable, entonces simplemente el elemento pasa a generar un nuevo nodo hoja en el lugar en esa posición, tal como operaría la operación de inserción del TAD lista en la lista de hijos. Por ejemplo, consideremos el resultado de insertar el elemento z en la posición g. El padre de g es c y su lista de hijos es (g). Al insertar z en la posición de g la lista de hijos pasa a ser (z, g), de manera que z pasa a ser el hijo más izquierdo de c (ver figura 3.16 derecha).

- Así como en listas `insert(p,x)` invalida las posiciones después de `p` (inclusive), en el caso de árboles, una inserción en el nodo `n` invalida las posiciones que son descendientes de `n` y que están a la derecha de `n`.

3.3.2.1. Algoritmo para copiar árboles

```
1. iterator tree_copy(tree &T,iterator nt,
2.                    tree &Q,iterator nq) {
3.   nq = /* nodo resultante de insertar el
4.           elemento de 'nt' en 'nq' . . . */;
```

```
5.    iterator
6.      ct = /* hijo mas izquierdo de 'nt' . . .*/,
7.      cq = /* hijo mas izquierdo de 'nq' . . .*/;
8.    while (/* 'ct' no es 'Lambda'. . . */) {
9.      cq = tree_copy(T,ct,Q,cq);
10.     ct = /* hermano derecho de 'ct'. . . */;
11.     cq = /* hermano derecho de 'cq'. . . */;
12.   }
13.   return nq;
14. }
```

Código 3.4: *Seudocódigo para copiar un árbol. [Archivo: treecpy.cpp]*

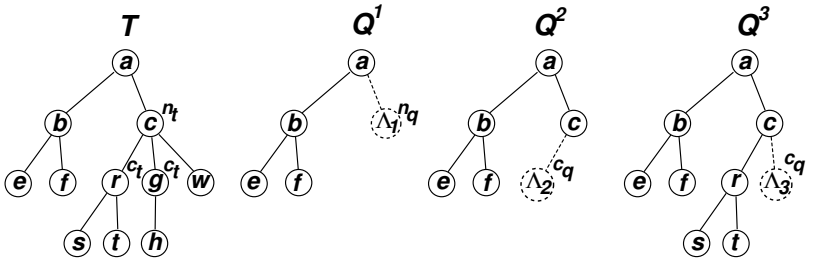


Figura 3.17: Algoritmo para copiar árboles.

Con estas operaciones podemos escribir el seudocódigo para una función que copia un árbol (ver código 3.4). Esta función copia el subárbol del nodo `nt` en el árbol `T` en la posición `nq` en el árbol `Q` y devuelve la posición de la raíz del subárbol insertado en `Q` (actualiza el nodo `nq` ya que después de la inserción es inválido). La función es recursiva, como lo son la mayoría de las operaciones no triviales sobre árboles. Consideremos el algoritmo aplicado al árbol de la figura 3.17 a la izquierda. Primero inserta el elemento que esta en `nt` en la posición `nq`. Luego va copiando cada uno de los subárboles de los hijos de `nq` como hijos del nodo `nt`. El nodo `ct` itera sobre los hijos de `nt` mientras que `cq` lo hace sobre los hijos de `nq`. Por ejemplo, si consideramos la aplicación del algoritmo a la copia del árbol de la figura 3.17 a la izquierda, concentrémonos en la copia del subárbol del nodo c del árbol T al Q.

Cuando llamamos a `tree_copy(T,nt,Q,nq)`, `nt` es c y `nq` es Λ_1, (mostrado como Q^1 en la figura). La línea 3 copia la raíz del subárbol que en este caso es el nodo c insertándolo en Λ_1. Después de esta línea, el árbol queda como se muestra en la etapa Q^2. Como en la inserción en listas, la línea

actualiza la posición **nq**, la cuál queda apuntando a la posición que contiene a c. Luego **ct** y **nt** toman los valores de los hijos más izquierdos, a saber r y Λ_2. Como **ct** no es Λ entonces el algoritmo entra en el lazo y la línea 9 copia todo el subárbol de r en Λ_2, quedando el árbol como en Q^3. De paso, la línea actualiza el iterator **cq**, de manera que **ct** y **cq** quedan apuntando a los dos nodos r en sus respectivos árboles. Notar que en este análisis no consideramos la llamada recursiva a **tree_copy()** sino que simplemente asumimos que estamos analizando la instancia específica de llamada a **tree_copy** donde **nt** es c y no aquéllas llamadas generadas por esta instancia. En las líneas 10–11, los iterators **ct** y **cq** son avanzados, de manera que quedan apuntando a g y Λ_2. En la siguiente ejecución del lazo la línea 9 copiará todo el subárbol de g a Λ_3. El proceso se detiene después de copiar el subárbol de w en cuyo caso **ct** obtendrá en la línea 10 un nodo Λ y la función termina, retornando el valor de **nq** actualizado.

```
1.  iterator mirror_copy(tree &T,iterator nt,
2.                       tree &Q,iterator nq) {
3.    nq = /* nodo resultante de insertar
4.            el elemento de 'nt' en 'nq' */;
5.    iterator
6.      ct = /* hijo mas izquierdo de 'nt' . . .*/,
7.      cq = /* hijo mas izquierdo de 'nq' . . .*/;
8.    while (/* 'ct' no es 'Lambda'. . . */) {
9.      cq = mirror_copy(T,ct,Q,cq);
10.     ct = /* hermano derecho de 'ct' . . . */;
11.   }
12.   return nq;
13. }
```

Código 3.5: *Seudocódigo para copiar un árbol en espejo. [Archivo: mirrorcpy.cpp]*

Con menores modificaciones la función puede copiar un árbol en forma espejada, es decir, de manera que todos los nodos hermanos queden en orden inverso entre sí. Para eso basta con *no avanzar* el iterator **cq** donde se copian los subárboles, es decir eliminar la línea 11 . Recordar que si en una lista se van insertando valores en una posición *sin avanzarla*, entonces los elementos quedan ordenados *en forma inversa* a como fueron ingresados. El algoritmo de copia espejo **mirror_copy()** puede observarse en el código 3.5.

Con algunas modificaciones el algoritmo puede ser usado para obtener la copia de un árbol reordenando las hojas en un orden arbitrario, por ejemplo dejándolas ordenadas entre sí.

3.3.3. Supresión en árboles

Al igual que en listas, solo se puede suprimir en posiciones dereferenciables. En el caso de suprimir en un nodo hoja, solo se elimina el nodo. Si el nodo tiene hijos, eliminarlo equivale a eliminar todo el subárbol correspondiente. Como en listas, eliminando un nodo devuelve la posición del hermano derecho que llena el espacio dejado por el nodo eliminado.

```
1.  iterator_t prune_odd(tree &T,iterator_t n) {
2.    if (/*valor de 'n'. . . */ % 2)
3.      /* elimina el nodo 'n' y refresca. . . */;
4.    else {
5.      iterator_t c =
6.        /* hijo mas izquierdo de 'n' . . . */;
7.      while (/*'c' no es 'Lambda' . . . */)
8.        c = prune_odd(T,c);
9.      n = /* hermano derecho de 'n'. . . */;
10.   }
11.   return n;
12. }
```

Código 3.6: *Algoritmo que elimina los nodos de un árbol que son impares, incluyendo todo su subárbol [Archivo: pruneodd.cpp]*

Por ejemplo consideremos el algoritmo **prune_odd** (ver código 3.6) que *"poda"* un árbol, eliminando todos los nodos de un árbol que son impares *incluyendo sus subárboles.* Por ejemplo, si `T=(6 (2 3 4) (5 8 10))`. Entonces después de aplicar **prune_odd** tenemos `T=(6 (2 4))`. Notar que los nodos 8 y 10 han sido eliminados ya que, si bien son pares, pertenecen al subárbol del nodo 5, que es impar. Si el elemento del nodo es impar todo el subárbol del nodo es eliminado en la línea 3, caso contrario los hijos son podados aplicándoles recursivamente la función. Notar que tanto si el valor contenido en **n** es impar como si no, **n** avanza una posición dentro de **prune_odd**, ya sea al eliminar el nodo en la línea 3 o al avanzar explícitamente en la línea 9.

3.3.4. Operaciones básicas sobre el tipo árbol

Los algoritmos para el listado presentados en las secciones previas, sugieren las siguientes operaciones abstractas sobre árboles

- Dado un nodo (posición o iterator sobre el árbol), obtener su hijo más izquierdo. (Puede retornar una posición Λ).
- Dado un nodo obtener su hermano derecho. (Puede retornar una posición Λ).
- Dada una posición, determinar si es Λ o no.
- Obtener la posición de la raíz del árbol.
- Dado un nodo obtener una referencia al dato contenido en el nodo.
- Dada una posición (dereferenciable o no) y un dato, insertar un nuevo nodo con ese dato en esa posición.
- Borrar un nodo y todo su subárbol correspondiente.

3.4. Interfaz básica para árboles

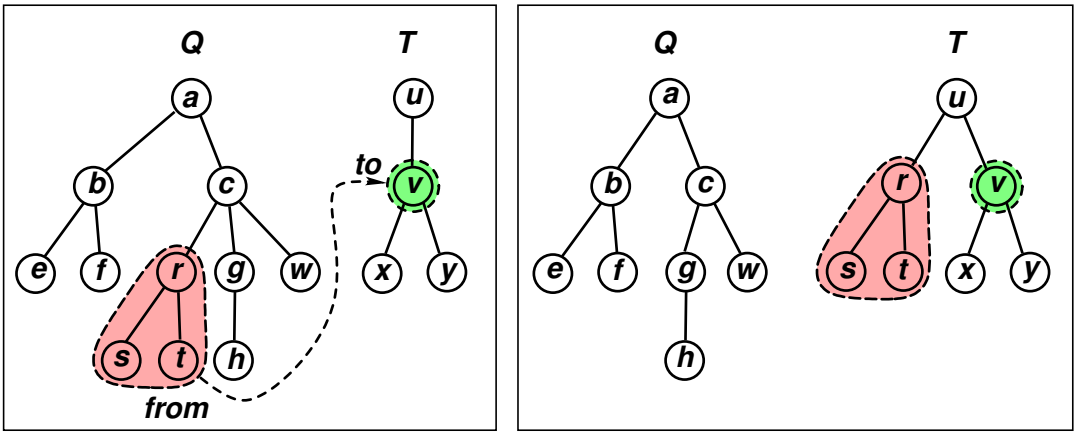
```
1.  class iterator_t {
2.    /* . . . . */
3.  public:
4.    iterator_t lchild();
5.    iterator_t right();
6.  };
7.
8.  class tree {
9.    /* . . . . */
10. public:
11.   iterator_t begin();
12.   iterator_t end();
13.   elem_t &retrieve(iterator_t p);
14.   iterator_t insert(iterator_t p,elem_t t);
15.   iterator_t erase(iterator_t p);
16.   void clear();
17.   iterator_t splice(iterator_t to,iterator_t from);
18. };
```

Código 3.7: *Interfaz básica para árboles. [Archivo: treebas1.h]*

Una interfaz básica, parcialmente compatible con la STL puede observarse en el código 3.7. Como con las listas y correspondencias tenemos una clase `iterator_t` que nos permite iterar sobre los nodos del árbol, tanto dereferenciables como no dereferenciables. En lo que sigue `T` es un árbol, `p`, `q` y `r` son nodos (iterators) y `x` es un elemento de tipo `elem_t`.

- `q = p.lchild()`: Dada una posición dereferenciable `p` retorna la posición del hijo más izquierdo (*"leftmost child"*). La posición retornada puede ser dereferenciable o no.
- `q = p.right()`: Dada una posición dereferenciable `p` retorna la posición del hermano derecho. La posición retornada puede ser dereferenciable o no.
- `T.end()`: retorna un iterator no dereferenciable.
- `p=T.begin()`: retorna la posición del comienzo del árbol, es decir la raíz. Si el árbol está vacío, entonces retorna `end()`.
- `x = T.retrieve(p)`: Dada una posición dereferenciable retorna una referencia al valor correspondiente.
- `q = T.insert(p,x)`: Inserta un nuevo nodo en la posición `p` conteniendo el elemento `x`. `p` puede ser dereferenciable o no. Retorna la posición del nuevo elemento insertado.
- `p = T.erase(p)`: Elimina la posición dereferenciable `p` y todo el subárbol de `p`.
- `T.clear()`: Elimina todos los elementos del árbol (equivale a `T.erase(T.begin())`).
- `T.splice(to,from)`: Elimina todo el subárbol del nodo dereferenciable `from` y lo inserta en el nodo (dereferenciable o no) `to`. `to` y `from` no deben tener relación de antecesor o descendiente y pueden estar en diferentes árboles. Por ejemplo, consideremos el ejemplo de la figura 3.18 (izquierda), donde deseamos mover todo el subárbol del nodo `r` en el árbol `T` a la posición del nodo `v` en el árbol `Q`. El resultado es como se muestra en la parte izquierda de la figura y se obtiene con el llamado `T.splice(r,v)`.

```
1. void preorder(tree &T,iterator_t n,list<int> &L) {
2.   L.insert(L.end(),T.retrieve(n));
3.
4.   iterator_t c = n.lchild();
5.   while (c!=T.end()) {
6.     preorder(T,c,L);
```



T.splice(v,r)

Figura 3.18: Operación splice.

```
7.    c = c.right();
8.   }
9.  }
10. void preorder(tree &T,list<int> &L) {
11.  if (T.begin()==T.end()) return;
12.  preorder(T,T.begin(),L);
13. }
14.
15. //---:---<*>---:---<*>---:---<*>---:---<*>
16. void postorder(tree &T,iterator_t n,list<int> &L) {
17.  iterator_t c = n.lchild();
18.  while (c!=T.end()) {
19.   postorder(T,c,L);
20.   c = c.right();
21.  }
22.  L.insert(L.end(),T.retrieve(n));
23. }
24. void postorder(tree &T,list<int> &L) {
25.  if (T.begin()==T.end()) return;
26.  postorder(T,T.begin(),L);
27. }
28.
29. //---:---<*>---:---<*>---:---<*>---:---<*>
30. void lisp_print(tree &T,iterator_t n) {
31.  iterator_t c = n.lchild();
32.  if (c==T.end()) cout << T.retrieve(n);
33.  else {
```

```
34.     cout << "(" << T.retrieve(n);
35.     while (c!=T.end()) {
36.       cout << " ";
37.       lisp_print(T,c);
38.       c = c.right();
39.     }
40.     cout << ")";
41.   }
42. }
43. void lisp_print(tree &T) {
44.   if (T.begin()!=T.end()) lisp_print(T,T.begin());
45. }
46.
47. //---:---<*>---:---<*>---:---<*>---:---<*>
48. iterator_t tree_copy(tree &T,iterator_t nt,
49.                      tree &Q,iterator_t nq) {
50.   nq = Q.insert(nq,T.retrieve(nt));
51.   iterator_t
52.     ct = nt.lchild(),
53.     cq = nq.lchild();
54.   while (ct!=T.end()) {
55.     cq = tree_copy(T,ct,Q,cq);
56.     ct = ct.right();
57.     cq = cq.right();
58.   }
59.   return nq;
60. }
61.
62. void tree_copy(tree &T,tree &Q) {
63.   if (T.begin() != T.end())
64.     tree_copy(T,T.begin(),Q,Q.begin());
65. }
66.
67. //---:---<*>---:---<*>---:---<*>---:---<*>
68. iterator_t mirror_copy(tree &T,iterator_t nt,
69.                        tree &Q,iterator_t nq) {
70.   nq = Q.insert(nq,T.retrieve(nt));
71.   iterator_t
72.     ct = nt.lchild(),
73.     cq = nq.lchild();
74.   while (ct != T.end()) {
75.     cq = mirror_copy(T,ct,Q,cq);
76.     ct = ct.right();
77.   }
78.   return nq;
79. }
```

```
80.
81. void mirror_copy(tree &T,tree &Q) {
82.   if (T.begin() != T.end())
83.     mirror_copy(T,T.begin(),Q,Q.begin());
84. }
85.
86. //---:---<*>---:---<*>---:---<*>---:---<*>
87. iterator_t prune_odd(tree &T,iterator_t n) {
88.   if (T.retrieve(n) % 2) n = T.erase(n);
89.   else {
90.     iterator_t c = n.lchild();
91.     while (c != T.end()) c = prune_odd(T,c);
92.     n = n.right();
93.   }
94.   return n;
95. }
96.
97. void prune_odd(tree &T) {
98.   if (T.begin() != T.end()) prune_odd(T,T.begin());
99. }
```

Código 3.8: *Diversos algoritmos sobre árboles con la interfaz básica. [Archivo: treetools.cpp]*

Los algoritmos **preorder**, **postorder**, **lisp_print**, **tree_copy**, **mirror_copy** y **prune_odd** descriptos en las secciones previas se encuentran implementados con las funciones de la interfaz básica en el código 3.8.

3.4.1. Listados en orden previo y posterior y notación Lisp

El listado en orden previo es simple. Primero inserta, el elemento del nodo **n** en el fin de la lista **L**. Notar que para obtener el elemento se utiliza el método **retrieve**. Luego se hace que **c** apunte al hijo más izquierdo de **n** y se va aplicando **preorder()** en forma recursiva sobre los hijos **c**. En la línea 7 se actualiza **c** de manera que recorra la lista de hijos de **n**. La función **postorder** es completamente análoga, sólo que el elemento de **n** es agregado *después* de los órdenes posteriores de los hijos.

3.4.2. Funciones auxiliares para recursión y sobrecarga de funciones

En general estas funciones recursivas se escriben utilizando una función auxiliar. En principio uno querría llamar a la función como **preorder(T)** *asumiendo que se aplica al nodo raíz de* T. Pero para después poder aplicarlo en forma recursiva necesitamos agregar un argumento adicional que es un nodo del árbol. Esto lo podríamos hacer usando una función recursiva adicional **preorder_aux(T,n)**. Finalmente, haríamos que **preorder(T)** llame a **preorder_aux**:

```
1. void preorder_aux(tree &T,iterator_t n,list<int> &L) {
2.   /* . . . */
3. }
4. void preorder(tree &T,list<int> &L) {
5.   preorder_aux(T,T.begin(),L);
6. }
```

Pero como C++ admite *"sobrecarga del nombre de funciones"*, no es necesario declarar la función auxiliar con un nombre diferente. Simplemente hay dos funciones **preorder()**, las cuales se diferencian por el número de argumentos.

A veces se dice que la función **preorder(T)** actúa como un *"wrapper"* (*"envoltorio"*) para la función **preorder(T,n)**, que es la que hace el trabajo real. **preorder(T)** sólo se encarga de pasarle los parámetros correctos a **preorder(T,n)**. De paso podemos usar el wrapper para realizar algunos chequeos como por ejemplo verificar que el nodo **n** no sea Λ. Si un nodo Λ es pasado a **preorder(T,n)** entonces seguramente se producirá un error al querer dereferenciar **n** en la línea 2.

Notar que Λ puede ser pasado a **preorder(T,n)** sólo si **T** es vacío, ya que una vez que un nodo dereferenciable es pasado a **preorder(T,n)**, el test de la línea 5 se encarga de no dejar nunca pasar un nodo Λ a una instancia inferior de **preorder(T,n)**. Si **T** no es vacío, entonces **T.begin()** es dereferenciable y a partir de ahí nunca llegará a **preorder(T,n)** un nodo Λ. Notar que es mucho más eficiente verificar que el árbol no este vacío en **preorder(T)** que hacerlo en **preorder(T,n)** ya que en el primer caso la verificación se hace una sola vez para todo el árbol.

La rutina **lisp_print** es básicamente equivalente a las de orden previo y orden posterior. Una diferencia es que **lisp_print** no apendiza a una lista, ya que en ese caso habría que tomar alguna convención para representar los paréntesis. **lisp_print()** simplemente imprime por terminal la notación Lisp del árbol.

3.4.3. Algoritmos de copia

Pasemos ahora a estudiar **tree_copy()** y **mirror_copy()**. Notar el llamado a **insert** en la línea 50. Notar que la línea actualiza el valor de **nq** ya que de otra manera quedaría inválido por la inserción. Notar como las líneas 52–53 y 56–57 van manteniendo los iterators **ct** y **cq** sobre posiciones equivalentes en **T** y **Q** respectivamente.

3.4.4. Algoritmo de poda

Notar que si el elemento en **n** es impar, todo el subárbol de **n** es eliminado, y **n** queda apuntando al hermano derecho, por el **erase** de la línea 88, caso contrario, **n** queda apuntando al hermano derecho por el avance explícito de la línea 92.

3.5. Implementación de la interfaz básica por punteros

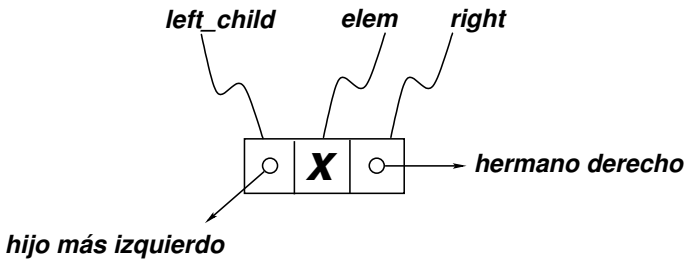


Figura 3.19: Celdas utilizadas en la representación de árboles por punteros.

Así como la implementación de listas por punteros mantiene los datos en celdas enlazadas por un campo **next**, es natural considerar una implementación de árboles en la cual los datos son almacenados en celdas que contienen, además del dato **elem**, un puntero **right** a la celda que corresponde al hermano derecho y otro **left_child** al hijo más izquierdo (ver figura 3.19). En la figura 3.20 vemos un árbol simple y su representación mediante celdas enlazadas.

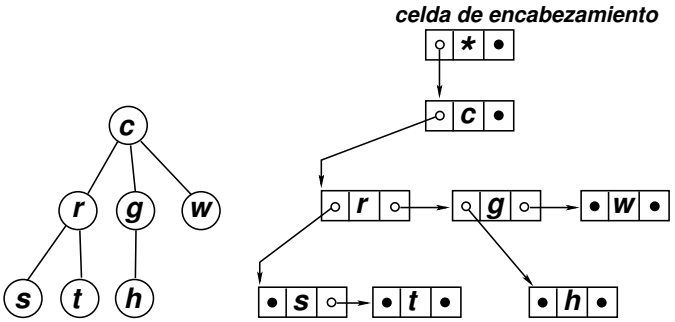


Figura 3.20: Representación de un árbol con celdas enlazadas por punteros.

3.5.1. El tipo iterator

Por analogía con las listas podríamos definir el tipo `iterator_t` como un `typedef` a `cell *`. Esto bastaría para representar posiciones dereferenciables y posiciones no dereferenciables que provienen de haber aplicado `right()` al último hermano, como la posición Λ_3 en la figura 3.21. Por supuesto habría que mantener el criterio de usar *"posiciones adelantadas"* con respecto al dato. Sin embargo no queda en claro como representar posiciones no dereferenciables como la Λ_1 que provienen de aplicar `lchild()` a una hoja.

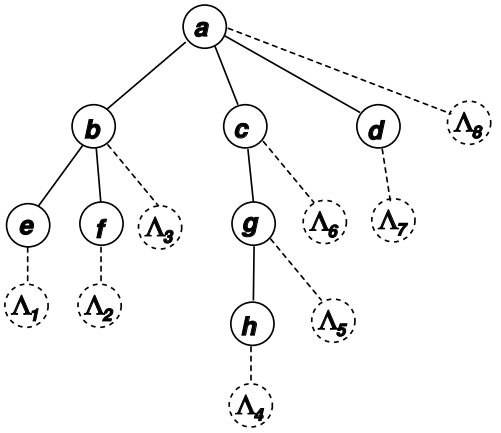


Figura 3.21: Todas las posiciones no dereferenciables de un árbol.

Una solución posible consiste en hacer que el tipo `iterator_t` contenga, además de un puntero a la celda que contiene el dato, punteros a celdas

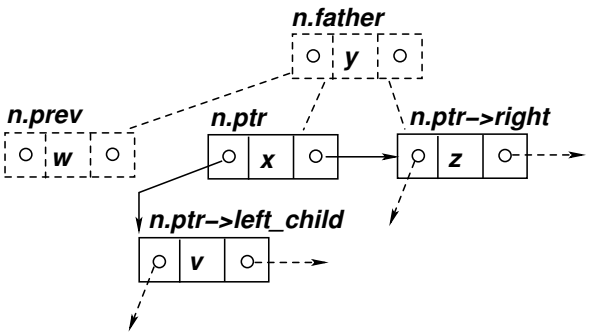


Figura 3.22: Entorno local de un iterator sobre árboles.

que de otra forma serían inaccesibles. Entonces el iterator consiste en *tres punteros a celdas* (ver figura 3.22) a saber,

- Un puntero **ptr** a la celda que contiene el dato. (Este puntero es nulo en el caso de posiciones no dereferenciables).
- Un puntero **prev** al *hermano izquierdo*.
- Un puntero **father** al padre.

Así, por ejemplo a continuación mostramos los juegos de punteros correspondientes a varias posiciones dereferenciables y no dereferenciables en el árbol de la figura 3.6:

- nodo e: **ptr=**e, **prev=NULL**, **father=**b.
- nodo f: **ptr=**f, **prev=**e, **father=**b.
- nodo Λ_1: **ptr=NULL**, **prev=NULL**, **father=**e.
- nodo Λ_2: **ptr=NULL**, **prev=NULL**, **father=**f.
- nodo Λ_3: **ptr=NULL**, **prev=**f, **father=**b.
- nodo g: **ptr=**g, **prev=NULL**, **father=**c.
- nodo Λ_6: **ptr=NULL**, **prev=**g, **father=**c.

Estos tres punteros tienen la suficiente información como para ubicar a todas las posiciones (dereferenciables o no) del árbol. Notar que todas las posiciones tienen un puntero **father** no nulo, mientras que el puntero **prev** puede o no ser nulo.

Para tener un código más uniforme se introduce una celda de encabezamiento, al igual que con las listas. La raíz del árbol, si existe, es una celda hija

de la celda de encabezamiento. Si el árbol está vacío, entonces el iterator correspondiente a la raíz (y que se obtiene llamando a **begin()**) corresponde a **ptr=NULL**, **prev=NULL**, **father=**celda de encabezamiento.

3.5.2. Las clases cell e iterator_t

```
1.  class tree;
2.  class iterator_t;
3.
4.  //---:---<*>---:---<*>---:---<*>---:---<*>
5.  class cell {
6.    friend class tree;
7.    friend class iterator_t;
8.    elem_t elem;
9.    cell *right, *left_child;
10.   cell() : right(NULL), left_child(NULL) {}
11. };
12.
13. //---:---<*>---:---<*>---:---<*>---:---<*>
14. class iterator_t {
15. private:
16.   friend class tree;
17.   cell *ptr,*prev,*father;
18.   iterator_t(cell *p,cell *prev_a, cell *f_a)
19.     : ptr(p), prev(prev_a), father(f_a) { }
20. public:
21.   iterator_t(const iterator_t &q) {
22.     ptr = q.ptr;
23.     prev = q.prev;
24.     father = q.father;
25.   }
26.   bool operator!=(iterator_t q) { return ptr!=q.ptr; }
27.   bool operator==(iterator_t q) { return ptr==q.ptr; }
28.   iterator_t()
29.     : ptr(NULL), prev(NULL), father(NULL) { }
30.
31.   iterator_t lchild() {
32.     return iterator_t(ptr->left_child,NULL,ptr);
33.   }
34.   iterator_t right() {
35.     return iterator_t(ptr->right,ptr,father);
36.   }
37. };
```

```
38.
39.  //---:---<*>---:---<*>---:---<*>---:---<*>
40.  class tree {
41.  private:
42.    cell *header;
43.    tree(const tree &T) {}
44.  public:
45.
46.    tree() {
47.      header = new cell;
48.      header->right = NULL;
49.      header->left_child = NULL;
50.    }
51.    ~tree() { clear(); delete header; }
52.
53.    elem_t &retrieve(iterator_t p) {
54.      return p.ptr->elem;
55.    }
56.
57.    iterator_t insert(iterator_t p,elem_t elem) {
58.      assert(!(p.father==header && p.ptr));
59.      cell *c = new cell;
60.      c->right = p.ptr;
61.      c->elem = elem;
62.      p.ptr = c;
63.      if (p.prev) p.prev->right = c;
64.      else p.father->left_child = c;
65.      return p;
66.    }
67.    iterator_t erase(iterator_t p) {
68.      if(p==end()) return p;
69.      iterator_t c = p.lchild();
70.      while (c!=end()) c = erase(c);
71.      cell *q = p.ptr;
72.      p.ptr = p.ptr->right;
73.      if (p.prev) p.prev->right = p.ptr;
74.      else p.father->left_child = p.ptr;
75.      delete q;
76.      return p;
77.    }
78.
79.    iterator_t splice(iterator_t to,iterator_t from) {
80.      assert(!(to.father==header && to.ptr));
81.      if (from.ptr->right == to.ptr) return from;
82.      cell *c = from.ptr;
83.
84.      if (from.prev) from.prev->right = c->right;
```

```
85.       else from.father->left_child = c->right;
86.
87.       c->right = to.ptr;
88.       to.ptr = c;
89.       if (to.prev) to.prev->right = c;
90.       else to.father->left_child = c;
91.
92.       return to;
93.     }
94.
95.     iterator_t find(elem_t elem) {
96.       return find(elem,begin());
97.     }
98.     iterator_t find(elem_t elem,iterator_t p) {
99.       if(p==end() || retrieve(p) == elem) return p;
100.      iterator_t q,c = p.lchild();
101.      while (c!=end()) {
102.        q = find(elem,c);
103.        if (q!=end()) return q;
104.        else c = c.right();
105.      }
106.      return iterator_t();
107.    }
108.    void clear() { erase(begin()); }
109.    iterator_t begin() {
110.      return iterator_t(header->left_child,NULL,header);
111.    }
112.    iterator_t end() { return iterator_t(); }
```

Código 3.9: *Implementación de la interfaz básica de árboles por punteros. [Archivo: treebas.h]*

Una implementación de la interfaz básica código 3.7 por punteros puede observarse en el código 3.9.

- Tenemos primero las declaraciones *"hacia adelante"* de **tree** e **iterator_t**. Esto nos habilita a declarar **friend** a las clases **tree** e **iterator_t**.
- La clase **cell** sólo declara los campos para contener al puntero al hijo más izquierdo y al hermano derecho. El constructor inicializa los punteros a **NULL**.

- La clase **iterator_t** declara **friend** a **tree**, pero no es necesario hacerlo con **cell**. **iterator_t** declara los campos punteros a celdas y por comodidad declaramos un constructor privado **iterator_t(cell *p,cell *pv, cell *f)** que simplemente asigna a los punteros internos los valores de los argumentos.
- Por otra parte existe un constructor público **iterator_t(const iterator_t &q)**. Este es el *"constructor por copia"* y es utilizado cuando hacemos por ejemplo **iterator_t p(q);** con **q** un iterador previamente definido.
- El operador de asignación de iteradores (por ejemplo **p=q**) es *sintetizado* por el compilador, y simplemente se reduce a una copia *bit a bit* de los datos miembros de la clase (en este caso los tres punteros **prt**, **prev** y **father**) lo cual es apropiado en este caso.
- Haber definido **iterator_t** como una clase (y no como un **typedef**) nos obliga a definir también los operadores **!=** y **==**. Esto nos permitirá comparar nodos por igualdad o desigualdad (**p==q** o **p!=q**). De la misma forma que con las posiciones en listas, los nodos *no* pueden compararse con los operadores de relación de orden (**<**, **<=**, **>** y **>=**). Notar que los operadores **==** y **!=** sólo comparan el campo **ptr** de forma que *todas las posiciones no dereferenciables (Λ) son "iguales" entre sí*. Esto permite comparar en los lazos cualquier posición **p** con **end()**, entonces **p==end()** retornará **true** incluso si **p** no es exactamente igual a **end()** (es decir tiene campos **father** y **prev** diferentes).
- El constructor por defecto **iterator_t()** devuelve un iterator con los tres punteros nulos. Este iterator no debería ser normalmente usado en ninguna operación, pero es invocado automáticamente por el compilador cuando declaramos iterators como en: **iterator p;**
- Las operaciones **lchild()** y **right()** son las que nos permiten movernos dentro del árbol. Sólo pueden aplicarse a posiciones dereferenciables y pueden retornar posiciones dereferenciables o no. Para entender como funcionan consideremos la información contenida en un iterator. Si consideramos un iterator **n** (ver figura 3.23), entonces la posición de **m=n.lchild()** está definida por los siguientes punteros

  - **m.ptr= n.ptr->left_child**
  - **m.prev= NULL** (ya que **m** es un *hijo más izquierdo*)
  - **m.father= n.ptr**

  Por otra parte, si consideramos la función hermano derecho: **q=n.right()**, entonces **q** está definida por los siguientes punteros,

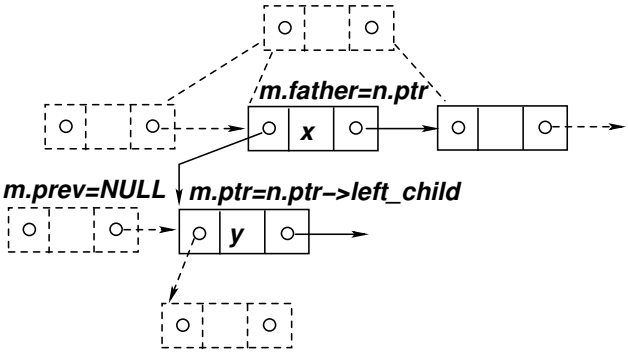


Figura 3.23: La función `lchild()`.

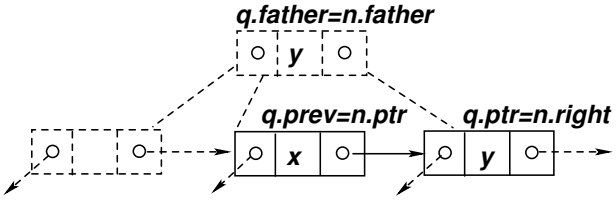


Figura 3.24: La función `right()`.

- `q.ptr= n.ptr->right`
- `q.prev= n.ptr`
- `q.father= n.father`

3.5.3. La clase tree

- La clase **`tree`** contiene un único dato que es un puntero a la celda de encabezamiento. Esta celda es alocada e inicializada en el constructor **`tree()`**.

- La función **`retrieve(p)`** simplemente retorna el dato contenido en la celda apuntada por **`p.ptr`**.

- La función **`insert(p,x)`** aloca una nueva celda **`c`** e inicializa sus campos datos. El dato **`elem`** se obtiene del argumento a la llamada y el puntero al hermano derecho pasa a ser el puntero **`ptr`** de la posición donde se va a insertar, ya que (al igual que con las listas) el valor insertado queda *a la izquierda* de la posición donde se inserta. Como la

nueva celda no va a tener hijos, el puntero `left_child` queda en `NULL` (esto se hace al crear la celda en el constructor de la clase `cell`). Después de insertar y enlazar la nueva celda hay que calcular la posición de la nueva celda insertada, que es el valor de retorno de `insert()`.

- `p.ptr=c`, la nueva celda insertada.
- `p.prev` no se altera ya que el hermano a la izquierda de la nueva celda es el mismo que el de la celda donde se insertó.
- `p.father` tampoco cambia, porque la nueva celda tiene el mismo padre que aquella posición donde se insertó.

Finalmente hay que actualizar los punteros en algunas celdas vecinas.

- Si `p.prev` no es nulo, entonces la celda *no es el hijo más izquierdo* y por lo tanto hay que actualizar el puntero `right` de la celda a la izquierda.
- Caso contrario, la nueva celda *pasa a ser el hijo más izquierdo* de su padre y por lo tanto hay que actualizar el puntero `left_child` de éste.

- En `erase(p)` la línea 70 eliminan todos los subárboles de `p` en forma recursiva. Notar que esta parte del código es genérica (independiente de la implementación particular). Finalmente las líneas 71–76 eliminan la celda correspondiente a `p` actualizando los punteros a las celdas vecinas si es necesario.

- La función `splice(to,from)` primero elimina el subárbol de `from` (líneas 84–85) en forma muy similar a las líneas 73–74 de `erase` pero sin eliminar recursivamente el subárbol, como en `erase()` ya que debe ser insertado en la posición `to`. Esta inserción se hace en las líneas 87–90, notar la similitud de estas líneas con la función `insert()`.

- La función `find(elem)` no fue descrita en la interfaz básica pero es introducida aquí. Retorna un iterator al nodo donde se encuentra el elemento `elem`. La implementación es completamente genérica se hace recursivamente definiendo la función auxiliar `find(elem,p)` que busca el elemento `elem` *en el subárbol del nodo* p.

- `clear()` llama a `erase()` sobre la raíz.

- `begin()` construye el iterator correspondiente a la raíz del árbol, es decir que los punteros correspondientes se obtienen a partir de la celda de encabezamiento como

  - `ptr=header->left_child`
  - `prev=NULL`
  - `father=header`

- Se ha incluido un *"constructor por copia"* (línea 43) en la parte privada. Recordemos que el constructor por copia es usado cuando un usuario declara objetos en la siguente forma

```
1.  tree T2(T1);
```

  es decir, al declarar un nuevo objeto `T2` a partir de uno preexistente `T1`.

  Para todas las clases que contienen punteros a otros objetos (y que pertenecen a la clase, es decir que debe encargarse de alocarlos y desalocarlos, como son las celdas en el caso de listas y árboles) hay que tener cuidado. Si uno deja que el compilador sintetice un constructor por copia entonces la copia se hace *"bit a bit"* con lo cual para los punteros simplemente copia el valor del puntero, no creando duplicados de los objetos apuntados. A esto se le llama *"shallow copy"*. De esta forma el objeto original y el duplicado comparten objetos internos, y eso debe hacerse con cuidado, o puede traer problemas. A menos que uno, por alguna razón prefiera este comportamiento, en general hay que optar por una de las siguientes alternativas

  - Implementar el constructor por copia correctamente, es decir copiando los componentes internos (*"deep copy"*). Este constructor funcionaría básicamente como la función `tree_copy()` descripta más arriba (ver §3.3.2.1). (La implementación de la interfaz avanzada y la de árbol binario están hechas así).
  - Declarar al constructor por copia, implementándolo con un cuerpo vacío y poniéndolo en la parte privada. Esto hace que si el usuario intenta escribir un código como el de arriba, el compilador dará un mensaje de error. Esto evita que un usuario desprevenido que no sabe que el constructor por copia no hace el *"deep copy"*, lo use por accidente. Por ejemplo

```
1. tree T1;
2. // pone cosas en T1...
3. tree T2(T1);
```

```
4. // Obtiene un nodo en T2
5. iterator_t n = T2.find(x);
6. // vacia el arbol T1
7. T1.clear();
8. // Intenta cambiar el valor en 'n'
9. T2.retrieve(n) = y; // ERROR!!
```

En este ejemplo, el usuario obtiene una *"shallow copy"* `T2` del árbol `T1` y genera un iterator a una posición dereferenciable `n` en `T2`. Después de vaciar el árbol `T1` y querer acceder al elemento en `n` genera un error en tiempo de ejecución, ya que en realidad la estructura interna de `T2` era compartida con `T1`. Al vaciar `T1`, se vacía también `T2`.

Con la inclusión de la línea 43 del código 3.9. , la instrucción `tree T2(T1);` genera un error en tiempo de compilación.

- Implementarlo como público, pero que de un error.

```
1. public:
2.   tree(const tree &T) {
3.     error("Constructor por copia no implementado!!");
4.   }
```

De esta forma, si el usuario escribe `tree T2(T1);` entonces compilará pero en tiempo de ejecución, si pasa por esa línea va a dar un error.

3.6. Interfaz avanzada

```
1. #ifndef AED_TREE_H
2. #define AED_TREE_H
3.
4. #include <cassert>
5. #include <iostream>
6. #include <cstddef>
7. #include <cstdlib>
8.
9. namespace aed {
10.
11.   //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
12.   template<class T>
13.   class tree {
14.   public:
15.     class iterator;
16.   private:
```

```
17.     class cell {
18.       friend class tree;
19.       friend class iterator;
20.       T t;
21.       cell *right, *left_child;
22.       cell() : right(NULL), left_child(NULL) {}
23.     };
24.     cell *header;
25.
26.     iterator tree_copy_aux(iterator nq,
27.                            tree<T> &TT,iterator nt) {
28.       nq = insert(nq,*nt);
29.       iterator
30.         ct = nt.lchild(),
31.         cq = nq.lchild();
32.       while (ct!=TT.end()) {
33.         cq = tree_copy_aux(cq,TT,ct);
34.         ct = ct.right();
35.         cq = cq.right();
36.       }
37.       return nq;
38.     }
39.   public:
40.     static int cell_count_m;
41.     static int cell_count() { return cell_count_m; }
42.     class iterator {
43.     private:
44.       friend class tree;
45.       cell *ptr,*prev,*father;
46.       iterator(cell *p,cell *prev_a,cell *f_a) : ptr(p),
47.         prev(prev_a), father(f_a) { }
48.     public:
49.       iterator(const iterator &q) {
50.         ptr = q.ptr;
51.         prev = q.prev;
52.         father = q.father;
53.       }
54.       T &operator*() { return ptr->t; }
55.       T *operator->() { return &ptr->t; }
56.       bool operator!=(iterator q) { return ptr!=q.ptr; }
57.       bool operator==(iterator q) { return ptr==q.ptr; }
58.       iterator() : ptr(NULL), prev(NULL), father(NULL) { }
59.
60.       iterator lchild() { return iterator(ptr->left_child,NULL,ptr); }
61.       iterator right() { return iterator(ptr->right,ptr,father); }
62.
63.       // Prefix:
```

```
64.       iterator operator++() {
65.         *this = right();
66.         return *this;
67.       }
68.       // Postfix:
69.       iterator operator++(int) {
70.         iterator q = *this;
71.         *this = right();
72.         return q;
73.       }
74.     };
75.
76.     tree() {
77.       header = new cell;
78.       cell_count_m++;
79.       header->right = NULL;
80.       header->left_child = NULL;
81.     }
82.     tree<T>(const tree<T> &TT) {
83.       if (&TT != this) {
84.         header = new cell;
85.         cell_count_m++;
86.         header->right = NULL;
87.         header->left_child = NULL;
88.         tree<T> &TTT = (tree<T> &) TT;
89.         if (TTT.begin()!=TTT.end())
90.           tree_copy_aux(begin(),TTT,TTT.begin());
91.       }
92.     }
93.     tree &operator=(tree<T> &TT) {
94.       if (this != &TT) {
95.         clear();
96.         tree_copy_aux(begin(),TT,TT.begin());
97.       }
98.       return *this;
99.     }
100.    ~tree() { clear(); delete header; cell_count_m--; }
101.    iterator insert(iterator p,T t) {
102.      assert(!(p.father==header && p.ptr));
103.      cell *c = new cell;
104.      cell_count_m++;
105.      c->right = p.ptr;
106.      c->t = t;
107.      p.ptr = c;
108.      if (p.prev) p.prev->right = c;
109.      else p.father->left_child = c;
110.      return p;
```

```
111.     }
112.     iterator erase(iterator p) {
113.       if(p==end()) return p;
114.       iterator c = p.lchild();
115.       while (c!=end()) c = erase(c);
116.       cell *q = p.ptr;
117.       p.ptr = p.ptr->right;
118.       if (p.prev) p.prev->right = p.ptr;
119.       else p.father->left_child = p.ptr;
120.       delete q;
121.       cell_count_m--;
122.       return p;
123.     }
124.
125.     iterator splice(iterator to,iterator from) {
126.       assert(!(to.father==header && to.ptr));
127.       if (from.ptr->right == to.ptr) return from;
128.       cell *c = from.ptr;
129.
130.       if (from.prev) from.prev->right = c->right;
131.       else from.father->left_child = c->right;
132.
133.       c->right = to.ptr;
134.       to.ptr = c;
135.       if (to.prev) to.prev->right = c;
136.       else to.father->left_child = c;
137.
138.       return to;
139.     }
140.     iterator find(T t) { return find(t,begin()); }
141.     iterator find(T t,iterator p) {
142.       if(p==end() || p.ptr->t == t) return p;
143.       iterator q,c = p.lchild();
144.       while (c!=end()) {
145.         q = find(t,c);
146.         if (q!=end()) return q;
147.         else c++;
148.       }
149.       return iterator();
150.     }
151.     void clear() { erase(begin()); }
152.     iterator begin() { return iterator(header->left_child,NULL,header); }
153.     iterator end() { return iterator(); }
154.
155.   };
156.
157.   template<class T>
```

```
158.   int tree<T>::cell_count_m = 0;
159.
160.   template<class T>
161.   void swap(tree<T> &T1, tree<T> &T2) { T1.swap(T2); }
162. }
163. #endif
```

Código 3.10: *Interfaz avanzada para árboles. [Archivo: tree.h]*

Una interfaz similar a la descripta en la sección §3.4 pero incluyendo templates, clases anidadas y sobrecarga de operadores puede observarse en el código 3.10.

- La clase **tree** pasa a ser ahora un template, de manera que podremos declarar **tree<int>**, **tree<double>**.
- Las clases **cell** e **iterator** son ahora clases anidadas dentro de **tree**. Externamente se verán como **tree<int>::cell** y **tree<int>::iterator**. Sin embargo, sólo **iterator** es pública y es usada *fuera* de **tree**.
- La dereferenciación de posiciones (nodos) **x=retrieve(p)** se reemplaza por **x=*p**. Para eso debemos *"sobrecargar"* el operador *****. Si el tipo elemento (es decir el tipo **T** del template) contiene campos, entonces vamos a querer extraer campos de un elemento almacenado en un nodo, por lo cual debemos hacer **(*p).campo**. Para poder hacer esto usando el operador **->** (es decir **p->campo**) debemos sobrecargar el operador **->**. Ambos operadores devuelven referencias de manera que es posible usarlos en el miembro izquierdo, como en ***p=x** o **p->campo=z**.
- Igual que con la interfaz básica, para poder hacer comparaciones de iterators debemos sobrecargar también los operadores **==** y **!=**. También tiene definido el constructor por copia.
- El avance por hermano derecho **p = p.right();** ahora se puede hacer con **p++**, de todas formas mantenemos la función **right()** que a veces resulta ser más compacta. Por ejemplo **q = p.right()** se traduce en **q=p; q++;** en la versión con operadores.
- La función estática **cell_count()**, permite obtener el número total de celdas alocadas por todos las instancias de la clase, e incluye las celdas de encabezamiento. Esta función fue introducida para debugging,

normalmente no debería ser usada por los usuarios de la clase. Como es estática puede invocarse como `tree<int>::cell_count()` o también sobre una instancia, `T.cell_count()`.

- Se ha incluido un constructor por copia, de manera que se puede copiar árboles usando directamente el operador `=`, por ejemplo

```
tree<int> T,Q;
// carga elementos en T . . . .
    Q = T;
```

Así como también pasar árboles por copia y definir contenedores que contienen árboles como por ejemplo una lista de árboles de enteros. `list< tree<int> >`. Esta función necesita otra función recursiva auxiliar que hemos llamado `tree_copy_aux()` y que normalmente no debería ser usada directamente por los usuarios de la clase, de manera que la incluimos en la sección privada.

3.6.1. Ejemplo de uso de la interfaz avanzada

```
typedef tree<int> tree_t;
typedef tree_t::iterator node_t;

int count_nodes(tree_t &T,node_t n) {
  if (n==T.end()) return 0;
  int m=1;
  node_t c = n.lchild();
  while(c!=T.end()) m += count_nodes(T,c++);
  return m;
}

int count_nodes(tree_t &T) {
  return count_nodes(T,T.begin());
}

int height(tree_t &T,node_t n) {
  if (n==T.end()) return -1;
  node_t c = n.lchild();
  if (c==T.end()) return 0;
  int son_max_height = -1;
  while (c!=T.end()) {
    int h = height(T,c);
    if (h>son_max_height) son_max_height = h;
    c++;
  }
```

```
26.   return 1+son_max_height;
27.  }
28.
29.  int height(tree_t &T) {
30.   return height(T,T.begin());
31.  }
32.
33.  void
34.  node_level_stat(tree_t &T,node_t n,
35.              int level,vector<int> &nod_lev) {
36.   if (n==T.end()) return;
37.   assert(nod_lev.size()>=level);
38.   if (nod_lev.size()==level) nod_lev.push_back(0);
39.   nod_lev[level]++;
40.   node_t c = n.lchild();
41.   while (c!=T.end()) {
42.    node_level_stat(T,c++,level+1,nod_lev);
43.   }
44.  }
45.
46.  void node_level_stat(tree_t &T,
47.                  vector<int> &nod_lev) {
48.   nod_lev.clear();
49.   node_level_stat(T,T.begin(),0,nod_lev);
50.   for (int j=0;j<nod_lev.size();j++) {
51.    cout << "[level: " << j
52.         << ", nodes: " << nod_lev[j] << "]";
53.   }
54.   cout << endl;
55.  }
56.
57.  //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
58.  int max_node(tree_t &T,node_t n) {
59.   if (n==T.end()) return -1;
60.   int w = *n;
61.   node_t c = n.lchild();
62.   while (c!=T.end()) {
63.    int ww = max_node(T,c++);
64.    if (ww > w) w = ww;
65.   }
66.   return w;
67.  }
68.
69.  int max_node(tree_t &T) {
70.   return max_node(T,T.begin());
```

```
71. }
72.
73. //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
74. int max_leaf(tree_t &T,node_t n) {
75.   if (n==T.end()) return -1;
76.   int w = *n;
77.   node_t c = n.lchild();
78.   if (c==T.end()) return w;
79.   w = 0;
80.   while (c!=T.end()) {
81.     int ww = max_leaf(T,c++);
82.     if (ww > w) w = ww;
83.   }
84.   return w;
85. }
86.
87. int max_leaf(tree_t &T) {
88.   return max_leaf(T,T.begin());
89. }
90.
91. //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
92. int leaf_count(tree_t &T,node_t n) {
93.   if (n==T.end()) return 0;
94.   node_t c = n.lchild();
95.   if (c==T.end()) return 1;
96.   int w = 0;
97.   while (c!=T.end()) w += leaf_count(T,c++);
98.   return w;
99. }
100.
101. int leaf_count(tree_t &T) {
102.   return leaf_count(T,T.begin());
103. }
```

Código 3.11: *Algunos ejemplos de uso de la interfaz avanzada para árboles. [Archivo: treetools2.cpp]*

En el código 3.11 vemos algunos ejemplos de uso de esta interfaz.

- Todo los ejemplos usan árboles de enteros, `tree<int>`. Los `typedef` de las líneas 1-2 permiten definir tipos `tree_t` y `node_t` que abrevian el código.
- Las funciones implementadas son

- `height(T)` su altura,
- `count_nodes(T)` cuenta los nodos de un árbol,
- `leaf_count(T)` el número de hojas,
- `max_node(T)` el máximo valor del elemento contenido en los nodos,
- `max_leaf(T)` el máximo valor del elemento contenido en las hojas,

- `node_level_stat(T,nod_lev)` calcula el número de nodos que hay en cada nivel del árbol, el cual se retorna en el `vector<int> nod_lev`, es decir, `nod_lev[l]` es el número de nodos en el nivel `l`.

3.7. Tiempos de ejecución

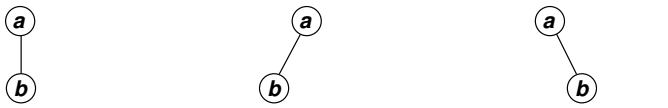
Operación	$T(n)$
`begin()`, `end()`, `n.right()`, `n++`, `n.left_child()`, `*n`, `insert()`, `splice(to,from)`	$O(1)$
`erase()`, `find()`, `clear()`, `T1=T2`	$O(n)$

Tabla 3.1: Tiempos de ejecución para operaciones sobre árboles.

En la Tabla 3.1 vemos los tiempos de ejecución para las diferentes operaciones sobre árboles. Es fácil ver que todas las funciones básicas tienen costo $O(1)$. Es notable que una función como `splice()` también sea $O(1)$. Esto se debe a que la operación de mover todo el árbol de una posición a otra se realiza con una operación de punteros. Las operaciones que no son $O(1)$ son `erase(p)` que debe eliminar todos los nodos del subárbol del nodo `p`, `clear()` que equivale a `erase(begin())`, `find(x)` y el constructor por copia (`T1=T2`). En todos los casos n es o bien el número de nodos del subárbol (`erase(p)` y `find(x,p)`) o bien el número total de nodos del árbol (`clear()`, `find(x)` y el constructor por copia `T1=T2`).

3.8. Arboles binarios

Los árboles que hemos estudiado hasta ahora son *"árboles ordenados orientados"* (AOO) ya que los hermanos están ordenados entre sí y hay una



ordenado orientado ***binario con hijo izquierdo*** ***binario con hijo derecho***

Figura 3.25: Diferentes casos de un árbol con dos nodos.

orientación de los caminos desde la raíz a las hojas. Otro tipo importante de árbol es el *"árbol binario"* (AB) en el cual cada nodo puede tener a lo sumo dos hijos. Además, si un dado nodo n tiene un sólo hijo, entonces este puede ser el hijo derecho o el hijo izquierdo de n. Por ejemplo si consideramos las posibles estructuras de árboles con dos nodos (ver figura 3.25), tenemos que para el caso de un AOO la única posibilidad es un nodo raíz con un nodo hijo. Por otra parte, si el árbol es binario, entonces existen dos posibilidades, que el único hijo sea el hijo izquierdo o el derecho. Dicho de otra forma los AB del centro y la derecha son diferentes, mientras que si fueran AOO entonces serían ambos iguales al de la izquierda.

3.8.1. Listados en orden simétrico

Los listados en orden previo y posterior para AB coinciden con su versión correspondiente para AOO. El *"listado en orden simétrico"* se define recursivamente como

$$\begin{aligned} \mathrm{osim}(\Lambda) &= < \text{lista vacía} > \\ \mathrm{osim}(n) &= (\mathrm{osim}(s_l), n, \mathrm{osim}(s_r)) \end{aligned} \tag{3.14}$$

donde $s_{l,r}$ son los hijos izquierdo y derecho de n, respectivamente.

3.8.2. Notación Lisp

La notación Lisp para árboles debe ser modificada un poco con respecto a la de AOO, ya que debemos introducir algún tipo de notación para un hijo Λ. Básicamente, en el caso en que un nodo tiene un sólo hijo, reemplazamos

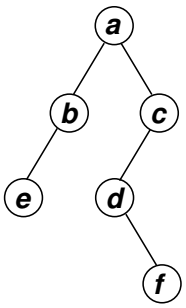


Figura 3.26: Ejemplo de árbol binario.

con un punto la posición del hijo faltante. La definición recursiva es

$$\text{lisp}(n) = \begin{cases} n & ;\ \text{si } s_l = \Lambda \text{ y } s_r = \Lambda \\ (n\ \text{lisp}(s_l)\ \text{lisp}(s_r)) & ;\ \text{si } s_l \neq \Lambda \text{ y } s_r \neq \Lambda \\ (n\ \cdot\ \text{lisp}(s_r)) & ;\ \text{si } s_l = \Lambda \text{ y } s_r \neq \Lambda \\ (n\ \text{lisp}(s_l)\ \cdot) & ;\ \text{si } s_l \neq \Lambda \text{ y } s_r = \Lambda \end{cases} \tag{3.15}$$

Por ejemplo, la notación Lisp del árbol de la figura 3.26 es

$$\text{lisp}(a) = (a\ (b\ e\ \cdot)\ (c\ (d\ \cdot\ f)\ \cdot)) \tag{3.16}$$

3.8.3. Árbol binario lleno

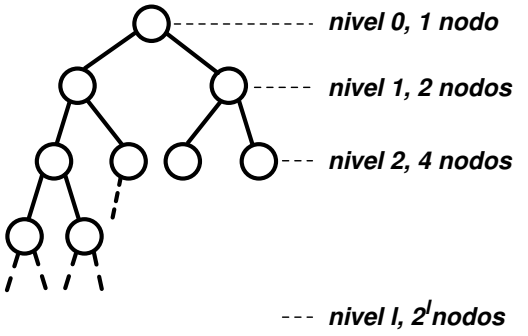


Figura 3.27: Cantidad máxima de nodos por nivel en un árbol binario lleno.

Un AOO no está limitado en cuanto a la cantidad de nodos que puede contener en un dado nivel. Por el contrario (ver figura 3.27) el árbol binario

puede tener a lo sumo dos nodos en el nivel 1, 4 nodos en el nivel 2, y en general, 2^l nodos en el nivel l. En total, un árbol binario de l niveles puede tener a lo sumo

$$n \leq 1 + 2 + 4... + 2^l \tag{3.17}$$

nodos. Pero ésta es una serie geométrica de razón dos, por lo cual

$$n \leq \frac{2^{l+1} - 1}{2 - 1} = 2^{l+1} - 1. \tag{3.18}$$

O bien

$$n < 2^{l+1}. \tag{3.19}$$

Inversamente el número de niveles puede obtenerse a partir del número de nodos como

$$\begin{aligned} l + 1 &> \log_2 n \\ l + 1 &> \text{floor}(\log_2 n) \\ l &\geq \text{floor}(\log_2 n) \end{aligned} \tag{3.20}$$

3.8.4. Operaciones básicas sobre árboles binarios

Las operaciones sobre AB difieren de las de AOO (ver sección §3.3.4) en las funciones que permiten *"moverse"* en el árbol. Si usáramos las operaciones de AOO para acceder a los hijos de un nodo en un AB, entonces para acceder al hijo derecho deberíamos hacer una operación *"hijo-más-izquierdo"* para acceder al hijo izquierdo y después una operación *"hermano-derecho"*. Pero el hijo izquierdo no necesariamente debe existir, por lo cual la estrategia de movimientos en el árbol debe ser cambiada. La solución es que existan dos operaciones independientes *"hijo-izquierdo"* e *"hijo-derecho"*. También, en AB *sólo se puede insertar en un nodo* Λ ya que la única posibilidad de insertar en un nodo dereferenciable, manteniendo el criterio usado para AOO, sería insertar en un hijo izquierdo que no tiene hermano derecho. En ese caso (manteniendo el criterio usado para AOO) el hijo izquierdo debería pasar a ser el derecho y el nuevo elemento pasaría a ser el hijo izquierdo y de todas formas esta operación no agregaría ninguna funcionalidad.

Entonces, las operaciones para el AB son las siguientes.

- Dado un nodo, obtener su *hijo izquierdo*. (Puede retornar una posición Λ).
- Dado un nodo, obtener su *hijo derecho*. (Puede retornar una posición Λ).

- Dada una posición, determinar si es Λ o no.
- Obtener la posición de la raíz del árbol.
- Dado un nodo obtener una referencia al dato contenido en el nodo.
- Dada una posición *no dereferenciable* y un dato, insertar un nuevo nodo con ese dato en esa posición.
- Borrar un nodo y todo su subárbol correspondiente.

Notar que sólo cambian las dos primeras y la inserción con respecto a las de AOO.

3.8.5. Interfaces e implementaciones

3.8.5.1. Interfaz básica

```
1.  class iterator_t {
2.    /* . . . */
3.   public:
4.    iterator_t left();
5.    iterator_t right();
6.  };
7.
8.  class btree {
9.    /* . . . */
10.   public:
11.    iterator_t begin();
12.    iterator_t end();
13.    elem_t & retrieve(iterator_t p);
14.    iterator_t insert(iterator_t p,elem_t t);
15.    iterator_t erase(iterator_t p);
16.    void clear();
17.    iterator_t splice(iterator_t to,iterator_t from);
18.  };
```

Código 3.12: *Interfaz básica para árboles binarios. [Archivo: btreebash.h]*

En el código 3.12 vemos una interfaz posible (recordemos que las STL *no* tienen clases de árboles) para AB. Como siempre, la llamamos básica porque no tiene templates, clases anidadas ni sobrecarga de operadores. Es similar a la mostrada para AOO en código 3.7, la única diferencia es que en la clase `iterator` las funciones `left()` y `right()` retornan los *hijos izquierdo*

y derecho, respectivamente, en lugar de las funciones **lchild()** (que en AOO retornaba el hijo más izquierdo) y **right()** (que en AOO retorna el *hermano* derecho).

3.8.5.2. Ejemplo de uso. Predicados de igualdad y espejo

```
1.  bool equal_p (btree &T,iterator_t nt,
2.            btree &Q,iterator_t nq) {
3.    if (nt==T.end() xor nq==Q.end()) return false;
4.    if (nt==T.end()) return true;
5.    if (T.retrieve(nt) != Q.retrieve(nq)) return false;
6.    return equal_p(T,nt.right(),Q,nq.right()) &&
7.      equal_p(T,nt.left(),Q,nq.left());
8.  }
9.  bool equal_p(btree &T,btree &Q) {
10.   return equal_p(T,T.begin(),Q,Q.begin());
11. }
```

Código 3.13: *Predicado que determina si dos árboles son iguales. [Archivo: equalp.cpp]*

Como ejemplo de uso de esta interfaz vemos en código 3.13 una función predicado (es decir una función que retorna un valor booleano) que determina si dos árboles binarios **T** y **Q** son iguales. Dos árboles son iguales si

- Ambos son vacíos
- Ambos no son vacíos, los valores de sus nodos son iguales y los hijos respectivos de su nodo raíz son iguales.

Como, descripto en §3.4.2 la función se basa en una función auxiliar recursiva que toma como argumento adicionales dos nodos **nt** y **nq** y determina si los subárboles de **nt** y **nq** son iguales entre sí.

La función recursiva primero determina si uno de los nodos es Λ y el otro no o viceversa. En ese caso la función debe retornar **false** inmediatamente. La expresión lógica buscada podría ser

```
1.  if ((nt==T.end() && nq=!Q.end()) ||
2.      (nt!=T.end() && nq==Q.end())) return false;
```

pero la expresión se puede escribir en la forma más compacta usada en la línea 3 usando el operador lógico **xor** (*"o exclusivo"*). Recordemos que

`x xor y` retorna verdadero sólo si uno de los operandos es verdadero y el otro falso. Si el código llega a la línea línea 4 es porque o bien ambos nodos son Λ o bien los dos no lo son. Por lo tanto, si `nt` es Λ entonces ambos lo son y la función puede retornar `true` ya que dos árboles vacíos ciertamente son iguales. Ahora, si el código llega a la línea 5 es porque ambos nodos no son Λ. En ese caso, los valores contenidos deben ser iguales. Por lo tanto la línea 5 retorna `false` si los valores son distintos. Finalmente, si el código llega a la línea línea 6 sólo resta comparar los subárboles derechos de `nt` y `nq` y sus subárboles izquierdos, los cuales deben ser iguales entre sí. Por supuesto estas comparaciones se hacen en forma recursiva.

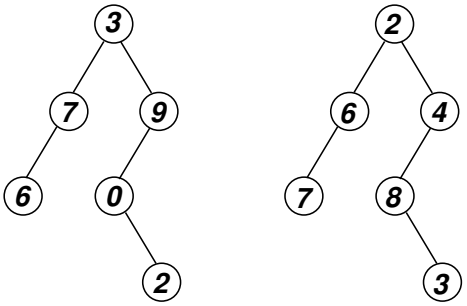


Figura 3.28: Dos árboles semejantes.

```
1.  bool semejante_p (btree &T,iterator_t nt,
2.            btree &Q,iterator_t nq) {
3.    if (nt==T.end() xor nq==Q.end()) return false;
4.    if (nt==T.end()) return true;
5.    return semejante_p(T,nt.right(),Q,nq.right()) &&
6.      semejante_p(T,nt.left(),Q,nq.left());
7.  }
8.  bool semejante_p(btree &T,btree &Q) {
9.    return semejante_p(T,T.begin(),Q,Q.begin());
10. }
```

Código 3.14: *Función predicado que determina si dos árboles son semejantes. [Archivo: semejantep.cpp]*

Modificando ligeramente este algoritmo verifica si dos árboles son *"semejantes"* es decir, son iguales en cuanto a su estructura, sin tener en cuenta

el valor de los nodos. Por ejemplo, los árboles de la figura 3.28 son semejantes entre sí. En forma recursiva la semejanza se puede definir en forma casi igual que la igualdad pero no hace falta que las raíces de los árboles sea igual. Dos árboles son semejantes si

- Ambos son vacíos
- Ambos no son vacíos, y los hijos respectivos de su nodo raíz son semejantes.

Notar que la única diferencia es que no se comparan los valores de las raíces de los subárboles comparados. En el código 3.14 se muestra una función predicado que determina si dos árboles son semejantes. El código es igual al de `equal_p` sólo que se elimina la línea 5.

3.8.5.3. Ejemplo de uso. Hacer espejo "in place"

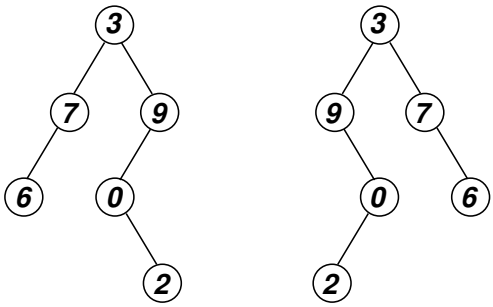


Figura 3.29: Copia espejo del árbol.

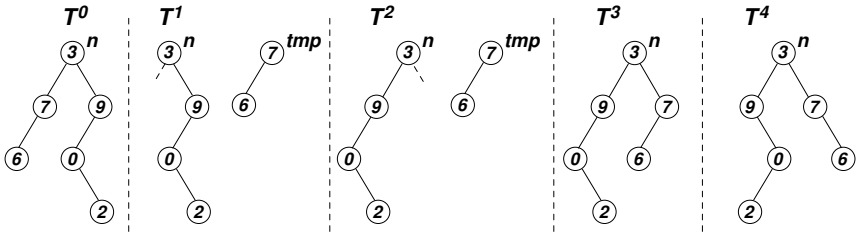


Figura 3.30: Descripción gráfica del procedimiento para copiar convertir *"in place"* un árbol en su espejo.

```
1.  void mirror(btree &T,iterator_t n) {
2.    if (n==T.end()) return;
3.    else {
4.      btree tmp;
5.      tmp.splice(tmp.begin(),n.left());
6.      T.splice(n.left(),n.right());
7.      T.splice(n.right(),tmp.begin());
8.      mirror(T,n.right());
9.      mirror(T,n.left());
10.   }
11. }
12. void mirror(btree &T) { mirror(T,T.begin()); }
```

Código 3.15: *Función para copiar convertir "in place" un árbol en su espejo. [Archivo: btmirror.cpp]*

Consideremos ahora una función **`void mirror(tree &T)`** que modifica el árbol **`T`**, dejándolo hecho igual a su espejo. Notar que esta operación es *"in place"*, es decir se hace en la estructura misma, sin crear una copia. El algoritmo es recursivo y se basa en intercambiar los subárboles de los hijos del nodo **`n`** y después aplicar recursivamente la función a los hijos (ver código 3.15). La operación del algoritmo sobre un nodo **`n`** del árbol se puede ver en la figura 3.30.

- La línea 5 extrae todo el subárbol del nodo izquierdo y lo inserta en un árbol vacío **`tmp`** con la operación **`splice(to,from)`**. Después de hacer esta operación el árbol se muestra como en el cuadro T^1.
- La línea 6 mueve todo el subárbol del hijo derecho al hijo izquierdo, quedando como en T^2.
- La línea 7 mueve todo el árbol guardado en **`tmp`** y que originariamente estaba en el hijo izquierdo al hijo derecho, quedando como en T^3.
- Finalmente en las líneas 8–9 la función se aplica recursivamente a los hijos derecho e izquierdo, de manera que el árbol queda como en T^4, es decir como el espejo del árbol original T^0.

3.8.5.4. Implementación con celdas enlazadas por punteros

Así como la diferencia entre las interfaces de AOO y AB difieren en las funciones **`lchild()`** y **`right()`** que son reemplazadas por **`left()`** y **`right()`**. (Recordar que **`right()`** tiene significado diferente en AOO y AB.)

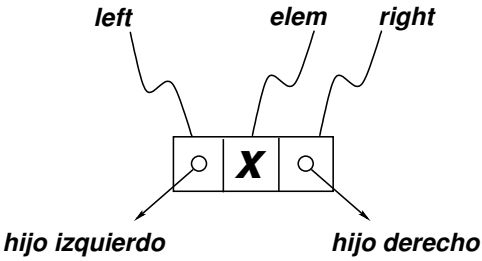


Figura 3.31: Celdas para representación de árboles binarios.

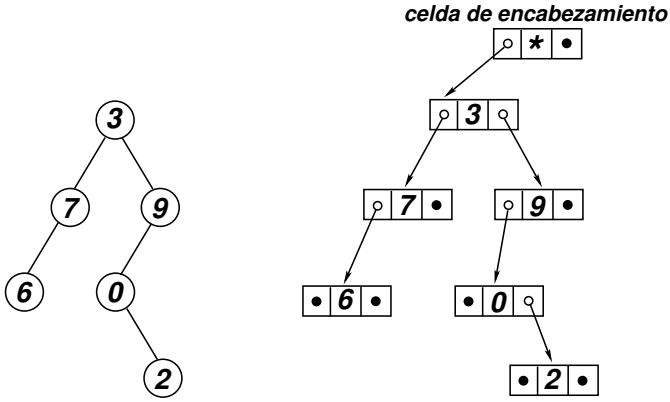


Figura 3.32: Representación de un árbol binario con celdas enlazadas.

Esto induce naturalmente a considerar que en la celda haya dos punteros que apunten al hijo izquierdo y al hijo derecho, como se muestra en la figura 3.31. En la figura 3.32 se observa a la derecha el enlace de las celdas para representar el árbol binario de la izquierda.

```
1.  typedef int elem_t;
2.  class cell;
3.  class iterator_t;
4.
5.  class cell {
6.    friend class btree;
7.    friend class iterator_t;
8.    elem_t t;
9.    cell *right,*left;
10.   cell() : right(NULL), left(NULL) {}
```

```
11.   };
12.
13.   class iterator_t {
14.   private:
15.     friend class btree;
16.     cell *ptr,*father;
17.     enum side_t {NONE,R,L};
18.     side_t side;
19.     iterator_t(cell *p,side_t side_a,cell *f_a)
20.       : ptr(p), side(side_a), father(f_a) { }
21.
22.   public:
23.     iterator_t(const iterator_t &q) {
24.       ptr = q.ptr;
25.       side = q.side;
26.       father = q.father;
27.     }
28.     bool operator!=(iterator_t q) { return ptr!=q.ptr; }
29.     bool operator==(iterator_t q) { return ptr==q.ptr; }
30.     iterator_t() : ptr(NULL), side(NONE),
31.
32.                    father(NULL) { }
33.
34.     iterator_t left() {
35.       return iterator_t(ptr->left,L,ptr);
36.     }
37.     iterator_t right() {
38.       return iterator_t(ptr->right,R,ptr);
39.     }
40.   };
41.
42.   class btree {
43.   private:
44.     cell *header;
45.     iterator_t tree_copy_aux(iterator_t nq,
46.                              btree &TT,iterator_t nt) {
47.       nq = insert(nq,TT.retrieve(nt));
48.       iterator_t m = nt.left();
49.       if (m != TT.end()) tree_copy_aux(nq.left(),TT,m);
50.       m = nt.right();
51.       if (m != TT.end()) tree_copy_aux(nq.right(),TT,m);
52.       return nq;
53.     }
54.   public:
55.     static int cell_count_m;
56.     static int cell_count() { return cell_count_m; }
```

```
57.     btree() {
58.       header = new cell;
59.       cell_count_m++;
60.       header->right = NULL;
61.       header->left = NULL;
62.     }
63.     btree(const btree &TT) {
64.       if (&TT != this) {
65.         header = new cell;
66.         cell_count_m++;
67.         header->right = NULL;
68.         header->left = NULL;
69.         btree &TTT = (btree &) TT;
70.         if (TTT.begin()!=TTT.end())
71.           tree_copy_aux(begin(),TTT,TTT.begin());
72.       }
73.     }
74.     ~btree() { clear(); delete header; cell_count_m--; }
75.     elem_t & retrieve(iterator_t p) { return p.ptr->t; }
76.     iterator_t insert(iterator_t p,elem_t t) {
77.       cell *c = new cell;
78.       cell_count_m++;
79.       c->t = t;
80.       if (p.side == iterator_t::R)
81.         p.father->right = c;
82.       else p.father->left = c;
83.       p.ptr = c;
84.       return p;
85.     }
86.     iterator_t erase(iterator_t p) {
87.       if(p==end()) return p;
88.       erase(p.right());
89.       erase(p.left());
90.       if (p.side == iterator_t::R)
91.         p.father->right = NULL;
92.       else p.father->left = NULL;
93.       delete p.ptr;
94.       cell_count_m--;
95.       p.ptr = NULL;
96.       return p;
97.     }
98.
99.     iterator_t splice(iterator_t to,iterator_t from) {
100.      cell *c = from.ptr;
101.      from.ptr = NULL;
102.      if (from.side == iterator_t::R)
```

```
103.        from.father->right = NULL;
104.      else
105.        from.father->left = NULL;
106.      if (to.side == iterator_t::R) to.father->right = c;
107.      else to.father->left = c;
108.      to.ptr = c;
109.      return to;
110.    }
111.    iterator_t find(elem_t t) { return find(t,begin()); }
112.    iterator_t find(elem_t t,iterator_t p) {
113.      if(p==end() || p.ptr->t == t) return p;
114.      iterator_t l = find(t,p.left());
115.      if (l!=end()) return l;
116.      iterator_t r = find(t,p.right());
117.      if (r!=end()) return r;
118.      return end();
119.    }
120.    void clear() { erase(begin()); }
121.    iterator_t begin() {
122.      return iterator_t(header->left,
123.                        iterator_t::L,header);
124.    }
125.    iterator_t end() { return iterator_t(); }
126.
127.    void lisp_print(iterator_t n) {
128.      if (n==end()) { cout << "."; return; }
129.      iterator_t r = n.right(), l = n.left();
130.      bool is_leaf = r==end() && l==end();
131.      if (is_leaf) cout << retrieve(n);
132.      else {
133.        cout << "(" << retrieve(n) << " ";
134.        lisp_print(l);
135.        cout << " ";
136.        lisp_print(r);
137.        cout << ")";
138.      }
139.    }
140.    void lisp_print() { lisp_print(begin()); }
141.  };
```

Código 3.16: *Implementación de árboles con celdas enlazadas por punteros. Declaraciones. [Archivo: btreebas.h]*

En el código 3.16 se muestra la implementación correspondiente.

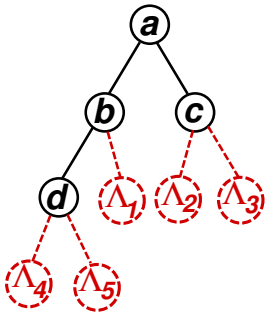


Figura 3.33: Iterators Λ en un árbol binario.

- **La clase iterator.** La clase `iterator_t` contiene un puntero a la celda, y otro al padre, como en el caso del AOO. Sin embargo el puntero `prev` que apuntaba al hermano a la izquierda, aquí ya no tiene sentido. Recordemos que el iterator nos debe permitir ubicar a las posiciones, incluso aquellas que son Λ. Para ello incluimos el iterator un miembro `side` de tipo tipo `enum side_t`, que puede tomar los valores `R` (right) y `L` (left). Por ejemplo consideremos el árbol de la figura 3.33. Además de los nodos $a-d$ existen 5 nodos Λ. Las posiciones de algunos nodos son representadas como sigue

  - nodo b: `ptr`=b, `father`=a, `side=L`
  - nodo c: `ptr`=c, `father`=a, `side=R`
  - nodo d: `ptr`=d, `father`=b, `side=L`
  - nodo Λ_1: `ptr=NULL`, `father`=b, `side=R`
  - nodo Λ_2: `ptr=NULL`, `father`=c, `side=L`
  - nodo Λ_3: `ptr=NULL`, `father`=c, `side=R`
  - nodo Λ_4: `ptr=NULL`, `father`=d, `side=L`
  - nodo Λ_5: `ptr=NULL`, `father`=d, `side=R`

- La comparación de iterators (líneas 28–29) compara sólo los campos `ptr`, de manera que todos los iterators Λ resultan iguales entre sí (ya que tienen `ptr=NULL`). Como `end()` retorna un iterator Λ (ver más abajo), entonces esto habilita a usar los lazos típicos

```
1.  while (c!=T.end()) {
2.    // . . . .
3.    c = c.right();
4.  }
```

- La clase **btree** incluye un contador de celdas **cell_count()** y constructor por copia **btree(const btree &)**, como para AOO.

- La diferencia principal está en **insert(p,x)** y **erase(p)**. En insert se crea la celda (actualizando el contador de celdas) y se inserta el dato (líneas 77–79. Recordar que los campos punteros de la celda quedan en **NULL**, porque así se inicializan en el constructor de celdas. El único campo de celdas que se debe actualizar es, o bien el campo **left** o **right** de la celda padre. Cuál de ellos es el que debe apuntar a la nueva celda se deduce de **p.side** en el iterator. Finalmente se debe actualizar el iterator de forma que **ptr** apunte a la celda creada.

- **erase(p)** elimina primero recursivamente todo el subárbol de los hijos izquierdo y derecho de **p**. Después libera la celda actualizando el campo correspondiente del padre (dependiendo de **p.side**). También se actualiza el contador **cell_count_m** al liberar la celda. Notar la actualización del contador por la liberación de las celdas en los subárboles de los hijos se hace automáticamente dentro de la llamada recursiva, de manera que en **erase(p)** sólo hay que liberar explícitamente a la celda **p.ptr**.

- El código de **splice(to,from)** es prácticamente un erase de **from** seguido de un **insert** en **to**.

- La posición raíz del árbol se elige como el hijo izquierdo de la celda de encabezamiento. Esto es una convención, podríamos haber elegido también el hijo derecho.

- El constructor por defecto de la clase **iterator** retorna un iterator no dereferenciable que no existe en el árbol. Todos sus punteros son nulos y **side** es un valor especial de **side_t** llamado **NONE**. Insertar en este iterator es un error.

- **end()** retorna un iterator no dereferenciable dado por el constructor por defecto de la clase **iterator_t** (descripto previamente). Este iterator debería ser usado sólo para comparar. Insertar en este iterator es un error.

3.8.5.5. Interfaz avanzada

```
1.  template<class T>
2.  class btree {
3.    /* . . . */
4.   public:
5.    class iterator {
6.    /* . . . */
7.    public:
8.      T &operator*();
9.      T *operator->();
10.     bool operator!=(iterator q);
11.     bool operator==(iterator q);
12.     iterator left();
13.     iterator right();
14.   };
15.   iterator begin();
16.   iterator end();
17.   iterator insert(iterator p,T t);
18.   iterator erase(iterator p);
19.   iterator splice(iterator to,iterator from);
20.   void clear();
21. };
```

Código 3.17: *Interfaz avanzada para árboles binarios. [Archivo: btreeh.h]*

En el código 3.17 vemos una interfaz para árboles binarios incluyendo templates, clases anidadas y sobrecarga de operadores. Las diferencias principales son (ver también lo explicado en la sección §3.6)

- La clase es un template sobre el tipo contenido en el dato (`class T`) de manera que podremos declarar `btree<int>`, `btree<double>` ...
- La dereferenciación de nodo se hace sobrecargando los operadores `*` y `->`, de manera que podemos hacer

```
1.  x = *n;
2.  *n = w;
3.  y = n->member;
4.    n->member = v;
```

donde `n` es de tipo iterator, `x,w` con de tipo `T` y `member` es algún campo de la clase `T` (si es una clase compuesta). También es válido hacer `n->f(...)` si `f` es un método de la clase `T`.

3.8.5.6. Ejemplo de uso. El algoritmo apply y principios de programación funcional.

A esta altura nos sería fácil escribir algoritmos que modifican los valores de un árbol, por ejemplo sumarle a todos los valores contenidos en un árbol un valor, o duplicarlos. Todos estos son casos particulares de un algoritmo más general `apply(Q,f)` que tiene como argumentos un árbol `Q` y una *"función escalar"* `T f(T)`. y le aplica a cada uno de los valores nodales la función en cuestión. Este es un ejemplo de *"programación funcional"*, es decir, programación en los cuales los datos de los algoritmos pueden ser también funciones.

C++ tiene un soporte básico para la programación funcional en la cual se pueden pasar *"punteros a funciones"*. Un soporte más avanzado se obtiene usando clases especiales que sobrecargan el operador `()`, a tales funciones se les llama *"functors"*. Nosotros vamos a escribir ahora una herramienta simple llamada `apply(Q,f)` que aplica a los nodos de un árbol una función escalar `t f(T)` pasada por puntero.

```
1.  template<class T>
2.  void apply(btree<T> &Q,
3.         typename btree<T>::iterator n,
4.         T(*f)(T)) {
5.    if (n==Q.end()) return;
6.    *n = f(*n);
7.    apply(Q,n.left(),f);
8.    apply(Q,n.right(),f);
9.  }
10. template<class T>
11. void apply(btree<T> &Q,T(*f)(T)) {
12.   apply(Q,Q.begin(),f);
```

Código 3.18: *Herramienta de programación funcional que aplica a los nodos de un árbol una función escalar. [Archivo: apply.cpp]*

La función se muestra en el código 3.18. Recordemos que para pasar funciones como argumentos, en realidad se pasa *el puntero a la función*. Como las funciones a pasar son funciones que toman como un argumento un elemento de tipo `T` y retornan un elemento del mismo tipo, su *"signatura"* (la forma como se declara) es `T f(T)`. La declaración de punteros a tales funciones se hace reemplazando en la signatura el nombre de la función por

`(*f)`. De ahí la declaración en la línea 11, donde el segundo argumento de la función es de tipo `T(*f)(T)`.

Por supuesto `apply` tiene una estructura recursiva y llama a su vez a una función auxiliar recursiva que toma un argumento adicional de tipo iterator. Dentro de esta función auxiliar el puntero a función `f` se aplica como una función normal, como se muestra en la línea 6.

Si `n` es Λ la función simplemente retorna. Si no lo está, entonces aplica la función al valor almacenado en `n` y después llama `apply` recursivamente a sus hijos izquierdo y derecho.

Otro ejemplo de programación funcional podría ser una función `reduce(Q,g)` que toma como argumentos un árbol `Q` y una función asociativa `T g(T,T)` (por ejemplo la suma, el producto, el máximo o el mínimo) y devuelve el resultado de aplicar la función asociativa a todos los valores nodales, hasta llegar a un único valor. Por ejemplo, si hacemos que `g(x,y)` retorne `x+y` retornará la suma de todas las etiquetas del árbol y si hacemos que retorne el máximo, entonces retornará el máximo de todas las etiquetas del árbol. Otra aplicación pueden ser *"filtros"*, como la función `prune_odd` discutida en la sección §3.3.3. Podríamos escribir una función `remove_if(Q,pred)` que tiene como argumentos un árbol `Q` y una función predicado `bool pred(T)`. La función `remove_if` elimina todos los nodos `n` (y sus subárboles) para cuyos valores la función `pred(*n)` retorna verdadero. La función `prune_odd` se podría obtener entonces simplemente pasando a `remove_if` una función predicado que retorna verdadero si el argumento es impar.

3.8.5.7. Implementación de la interfaz avanzada

```
1.  #ifndef AED_BTREE_H
2.  #define AED_BTREE_H
3.
4.  #include <iostream>
5.  #include <cstddef>
6.  #include <cstdlib>
7.  #include <cassert>
8.  #include <list>
9.
10. using namespace std;
11.
12. namespace aed {
13.
```

```
14.  //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
15.  template<class T>
16.  class btree {
17.  public:
18.    class iterator;
19.  private:
20.    class cell {
21.      friend class btree;
22.      friend class iterator;
23.      T t;
24.      cell *right,*left;
25.      cell() : right(NULL), left(NULL) {}
26.    };
27.    cell *header;
28.    enum side_t {NONE,R,L};
29.  public:
30.    static int cell_count_m;
31.    static int cell_count() { return cell_count_m; }
32.    class iterator {
33.    private:
34.      friend class btree;
35.      cell *ptr,*father;
36.      side_t side;
37.      iterator(cell *p,side_t side_a,cell *f_a)
38.        : ptr(p), side(side_a), father(f_a) { }
39.    public:
40.      iterator(const iterator &q) {
41.        ptr = q.ptr;
42.        side = q.side;
43.        father = q.father;
44.      }
45.      T &operator*() { return ptr->t; }
46.      T *operator->() { return &ptr->t; }
47.      bool operator!=(iterator q) { return ptr!=q.ptr; }
48.      bool operator==(iterator q) { return ptr==q.ptr; }
49.      iterator() : ptr(NULL), side(NONE), father(NULL) { }
50.
51.      iterator left() { return iterator(ptr->left,L,ptr); }
52.      iterator right() { return iterator(ptr->right,R,ptr); }
53.
54.    };
55.
56.    btree() {
57.      header = new cell;
58.      cell_count_m++;
59.      header->right = NULL;
60.      header->left = NULL;
```

```
61.     }
62.     btree<T>(const btree<T> &TT) {
63.       if (&TT != this) {
64.         header = new cell;
65.         cell_count_m++;
66.         header->right = NULL;
67.         header->left = NULL;
68.         btree<T> &TTT = (btree<T> &) TT;
69.         if (TTT.begin()!=TTT.end())
70.           copy(begin(),TTT,TTT.begin());
71.       }
72.     }
73.     btree &operator=(btree<T> &TT) {
74.       if (this != &TT) {
75.         clear();
76.         copy(begin(),TT,TT.begin());
77.       }
78.       return *this;
79.     }
80.     ~btree() { clear(); delete header; cell_count_m--; }
81.     iterator insert(iterator p,T t) {
82.       assert(p==end());
83.       cell *c = new cell;
84.       cell_count_m++;
85.       c->t = t;
86.       if (p.side==R) p.father->right = c;
87.       else p.father->left = c;
88.       p.ptr = c;
89.       return p;
90.     }
91.     iterator erase(iterator p) {
92.       if(p==end()) return p;
93.       erase(p.right());
94.       erase(p.left());
95.       if (p.side==R) p.father->right = NULL;
96.       else p.father->left = NULL;
97.       delete p.ptr;
98.       cell_count_m--;
99.       p.ptr = NULL;
100.      return p;
101.    }
102.
103.    iterator splice(iterator to,iterator from) {
104.      if (from==end()) return to;
105.      cell *c = from.ptr;
106.      from.ptr = NULL;
107.      if (from.side==R) from.father->right = NULL;
```

```
108.       else from.father->left = NULL;
109.
110.       if (to.side==R) to.father->right = c;
111.       else to.father->left = c;
112.       to.ptr = c;
113.       return to;
114.     }
115.     iterator copy(iterator nq,btree<T> &TT,iterator nt) {
116.       nq = insert(nq,*nt);
117.       iterator m = nt.left();
118.       if (m != TT.end()) copy(nq.left(),TT,m);
119.       m = nt.right();
120.       if (m != TT.end()) copy(nq.right(),TT,m);
121.       return nq;
122.     }
123.     iterator find(T t) { return find(t,begin()); }
124.     iterator find(T t,iterator p) {
125.       if(p==end() || p.ptr->t == t) return p;
126.       iterator l = find(t,p.left());
127.       if (l!=end()) return l;
128.       iterator r = find(t,p.right());
129.       if (r!=end()) return r;
130.       return end();
131.     }
132.     void clear() { erase(begin()); }
133.     iterator begin() { return iterator(header->left,L,header); }
134.
135.     void lisp_print(iterator n) {
136.       if (n==end()) { cout << "."; return; }
137.       iterator r = n.right(), l = n.left();
138.       bool is_leaf = r==end() && l==end();
139.       if (is_leaf) cout << *n;
140.       else {
141.         cout << "(" << *n << " ";
142.         lisp_print(l);
143.         cout << " ";
144.         lisp_print(r);
145.         cout << ")";
146.       }
147.     }
148.     void lisp_print() { lisp_print(begin()); }
149.
150.     iterator end() { return iterator(); }
151.   };
152.
153.   template<class T>
154.   int btree<T>::cell_count_m = 0;
```

```
155. }
156. #endif
```

Código 3.19: *Implementación de la interfaz avanzada de árboles binarios por punteros. [Archivo: btree.h]*

En el código 3.19 vemos una posible implementación de interfaz avanzada de AB con celdas enlazadas por punteros.

3.8.6. Arboles de Huffman

Los árboles de Huffman son un ejemplo interesante de utilización del TAD AB. El objetivo es comprimir archivos o mensajes de texto. Por simplicidad, supongamos que tenemos una cadena de N caracteres compuesta de un cierto conjunto reducido de caracteres C. Por ejemplo si consideramos las letras $C = \{a, b, c, d\}$ entonces el mensaje podría ser $abdcdacabbcdba$. El objetivo es encontrar un representación del mensaje en bits (es decir una cadena de 0's y 1's) lo más corta posible. A esta cadena de 0's y 1's la llamaremos *"el mensaje encodado"*. El algoritmo debe permitir recuperar el mensaje original, ya que de esta forma, si la cadena de caracteres representa un archivo, entonces podemos guardar el mensaje encodado (que es más corto) con el consecuente ahorro de espacio.

Una primera posibilidad es el código provisto por la representación binaria ASCII de los caracteres. De esta forma cada caracter se encoda en un código de 8 bits. Si el mensaje tiene N caracteres, entonces el mensaje encodado tendrá una longitud de $l = 8N$ bits, resultando en una longitud promedio de

$$\langle l \rangle = \frac{l}{N} = 8 \text{ bits/caracter} \tag{3.21}$$

Pero como sabemos que el mensaje sólo está compuesto de las cuatro letras a, b, c, d podemos crear un código de dos bits como el $C1$ en la Tabla 3.2, de manera que un mensaje como $abcdcba$ se encoda en 00011011100100. Desencodar un mensaje también es simple, vamos tomando dos caracteres del mensaje y consultando el código lo vamos convirtiendo al carácter correspondiente. En este caso la longitud del código pasa a ser de $\langle l \rangle = 2$ bits/caracter, es decir la cuarta parte del código ASCII. Por supuesto esta gran compresión se produce por el reducido conjunto de caracteres utilizados. Si el conjunto de caracteres fuera de tamaño 8 en vez

de 4 necesitaríamos al menos códigos de 3 bits, resultando en una tasa de 3 bits/caracter. En general si el número de caracteres es n_c la tasa será de

$$\langle l \rangle = \text{ceil}(\log_2 n_c) \tag{3.22}$$

Si consideramos texto común, el número de caracteres puede oscilar entre unos 90 y 128 caracteres, con lo cual la tasa será de 7 bits por caracter, lo cual representa una ganancia relativamente pequeña del 12.5 %.

Letra	Código $C1$	Código $C2$	Código $C3$
a	00	0	0
b	01	100	01
c	10	101	10
d	11	11	101

Tabla 3.2: Dos códigos posibles para un juego de 4 caracteres.

Una forma de mejorar la compresión es utilizar códigos de longitud variable, tratando de asignar códigos de longitud corta a los caracteres que tienen más probabilidad de aparecer y códigos más largos a los que tienen menos probabilidad de aparecer. Por ejemplo, si volvemos al conjunto de caracteres a, b, c, d y si suponemos que a tiene una probabilidad de aparecer del 70 % mientras que b, c, d sólo del 10 % cada uno, entonces podríamos considerar el código $C2$ de la Tabla 3.2. Si bien la longitud en bits de a es menor en $C2$ que en $C1$, la longitud de b y c es mayor, de manera que determinar cual de los dos códigos es mejor no es directo. Consideremos un mensaje típico de 100 caracteres. En promedio tendría 70 a's, 10 b's, 10c's y 10d's, lo cual resultaría en un mensaje encodado de $70\times1+10\times3+10\times3+10\times2 = 150$ bits, resultando en una longitud promedio de $\langle l \rangle$=1.5 bit/caracter. Notemos que en general

$$\begin{aligned} \langle l \rangle &= \frac{150 \text{ bits}}{100 \text{ caracteres}} \\ &= 0.70 \times 1 + 0.1 \times 3 + 0.1 \times 3 + 0.1 \times 2 \\ &= \sum_c P(c) l(c) \end{aligned} \tag{3.23}$$

Esta longitud media representa una compresión de 25 % con respecto al $C1$. Por supuesto, la ventaja del código $C2$ se debe a la gran diferencia en probabilidades entre a y los otros caracteres. Si la diferencia fuera menor, digamos $P(a) = 0.4$, $P(b) = P(c) = P(d) = 0.2$ ($P(c)$ es la probabilidad

de que en el mensaje aparezca un caracter c), entonces la longitud de un mensaje típico de 100 caracteres sería $40 \times 1 + 20 \times 3 + 20 \times 3 + 20 \times 2 = 200$ bits, o sea una tasa de 2 bits/caracter, igual a la de $C1$.

3.8.6.1. Condición de prefijos

En el caso de que un código de longitud variable como el $C2$ sea más conveniente, debemos poder asegurarnos poder desencodar los mensajes, es decir que la relación entre mensaje y mensaje encodado sea unívoca y que exista un algoritmo para encodar y desencodar mensajes en un tiempo razonable. Por empezar los códigos de los diferentes caracteres deben ser diferentes entre sí, pero esto no es suficiente. Por el ejemplo el código $C3$ tiene un código diferente para todos los caracteres, pero los mensaje dba y ccc se encodan ambos como 101010.

El error del código $C3$ es que el código de d empieza con el código de c. Decimos que el código de c es *"prefijo"* del de d. Así, al comenzar a desencodar el mensaje 101010. Cuando leemos los dos primeros bits 10, no sabemos si ya extraer una c o seguir con el tercer bit para formar una d. Notar que también el código de a es prefijo del de b. Por lo tanto, una condición para admitir un código es que cumpla con la *"condición de prefijos"*, a saber que *el código de un caracter no sea prefijo del código de ningún otro caracter*. Por ejemplo los códigos de longitud fija como el $C1$ trivialmente satisfacen la condición de prefijos. La única posibilidad de que violaran la condición sería si los códigos de dos caracteres son iguales, lo cual ya fue descartado.

3.8.6.2. Representación de códigos como árboles de Huffman

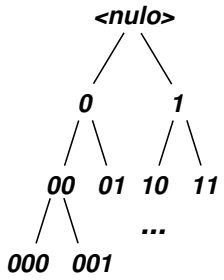


Figura 3.34: Representación de códigos binarios con árboles de Huffman.

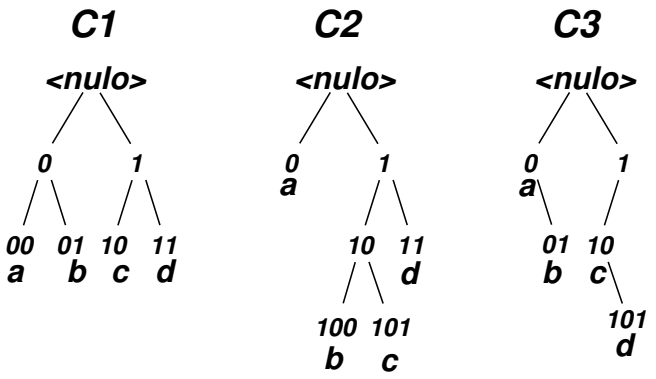


Figura 3.35: Representación de los códigos binarios $C1$, $C2$ y $C3$.

Para códigos de longitud variable como el $C2$ la condición de prefijos se puede verificar caracter por caracter, pero es más simple y elegante si representamos el código como un *"árbol de Huffman"*. En esta representación, asociamos con cada nodo de un AB un código binario de la siguiente manera. Al nodo raíz lo asociamos con el código de longitud 0 para el resto de los nodos la definición es recursiva: si un nodo tiene código $b_0b_1...b_{n-1}$ entonces el hijo izquierdo tiene código $b_0b_1...b_{n-1}0$ y el hijo derecho $b_0b_1...b_{n-1}1$. Así se van generando todos los códigos binarios posibles. Notar que en el nivel l se encuentran todos los códigos de longitud l. Un dado código (como los $C1$-$C3$) se puede representar como un AB marcando los nodos correspondientes a los códigos de cada uno de los caracteres y eliminando todos los nodos que no están contenidos dentro de los caminos que van desde esos nodos a la raíz. Es inmediato ver que la longitud del código de un caracter es la profundidad del nodo correspondiente. Una forma visual de construir el árbol es comenzar con un árbol vacío y comenzar a dibujar los caminos correspondientes al código de cada caracter. De esta manera los árboles correspondientes a los códigos $C1$ a $C3$ se pueden observar en la figura 3.35. Por construcción, a cada una de las hojas del árbol le corresponde un caracter. Notemos que en el caso del código $C3$ el camino del caracter a esta contenido en el de b y el de c está contenido en el de d. De manera que la condición de prefijos se formula en la representación mediante árboles exigiendo que los caracteres estén *sólo en las hojas*, es decir *no en los nodos interiores*.

3.8.6.3. Códigos redundantes

Letra	Código $C2$	Código $C4$
a	0	0
b	100	10100
c	101	10101
d	11	111

Tabla 3.3: Dos códigos posibles para un juego de 4 caracteres.

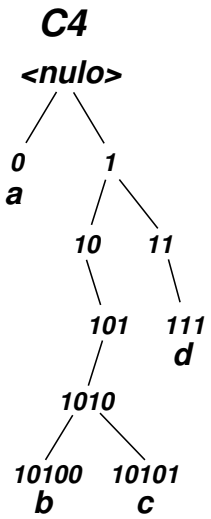


Figura 3.36: Código redundante.

Consideremos ahora el código $C4$ mostrado en la Tabla 3.3 (ver figura 3.36). El código satisface la condición de prefijos, ya que los caracteres están asignados a 4 hojas distintas del árbol. Notemos que el árbol tiene 3 nodos interiores 10, 101 y 11 que *tienen un sólo hijo*. Entonces podemos eliminar tales nodos interiores, *"subiendo"* todo el subárbol de 1010 a la posición del 10 y la del 111 a la posición del 11. El código resultante resulta ser igual al $C2$ que ya hemos visto. Como todos los nodos suben y la profundidad de los nodos da la longitud del código, es obvio que la longitud del código medio será siempre menor para el código $C2$ que para el $C4$, *independientemente de las probabilidades de los códigos de cada caracter*. Decimos entonces que un código como el $C4$ que tiene nodos interiores

con un sólo hijo es *"redundante"*, ya que puede ser trivialmente optimizado eliminando tales nodos y subiendo sus subárboles. Si un AB es tal que no contiene nodos interiores con un solo hijo se le llama *"árbol binario lleno"* (Full Binary Tree, FBT). Es decir, en un árbol binario lleno los nodos son o bien hojas, o bien nodos interiores con sus dos hijos.

3.8.6.4. Tabla de códigos óptima. Algoritmo de búsqueda exhaustiva

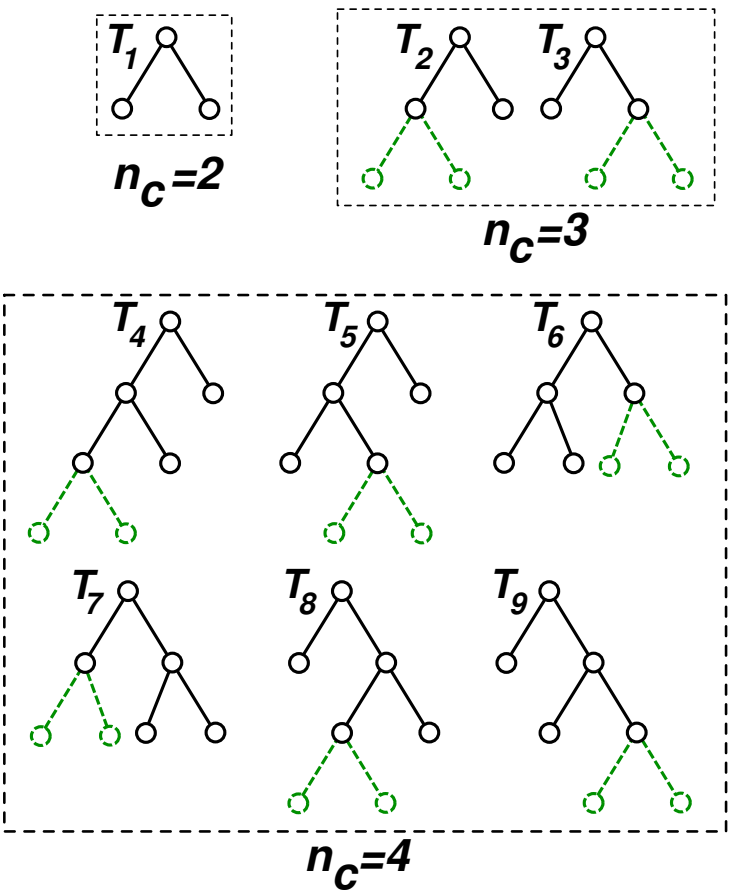


Figura 3.37: Posibles árboles binarios llenos con n_c hojas.

El objetivo ahora es, *dado un conjunto de n_c caracteres, con probabilidades P_0, P_1, ..., P_{n_c-1}, encontrar, de todos los posibles FBT con n_c hojas, aquel que minimiza la longitud promedio del código*. Como se discutió en pri-

mer capítulo, sección §1.1.4, una posibilidad es generar todos los posibles árboles llenos de n_c caracteres, calcular la longitud promedio de cada uno de ellos y quedarnos con el que tiene la longitud promedio mínima. Consideremos, por ejemplo el caso de $n_c = 2$ caracteres. Es trivial ver que hay un solo posible FBT con dos hojas, el cual es mostrado en la figura 3.37 (árbol T_1). Los posibles FBT de tres hojas se pueden obtener del de dos hojas convirtiendo a cada una de las hojas de T_1 en un nodo interior, insertándole dos hijos. Así, el árbol T_2 se obtiene insertándole dos nodos a la hoja izquierda de T_1 y el T_3 agregándole a la hoja derecha. En ambos casos los nodos agregados se marcan en verde. Así siguiendo, por cada FBT de 3 hojas se pueden obtener 3 FBT de 4 hojas, *"bifurcando"* cada una de sus hojas. En la figura T_4, T_5 y T_6 se obtienen bifurcando las hojas de $T2$ y T_7, T_8 y T_9 las hojas de T_3. Como hay 2 FBT de 3 hojas, se obtienen en total 6 FBT de 4 hojas. Siguiendo con el razonamiento, *el número de FBT de n_c hojas es* $(n_c-1)!$. Notar que algunas de las estructuras son redundantes, por ejemplo los árboles T_6 y T_7 son equivalentes.

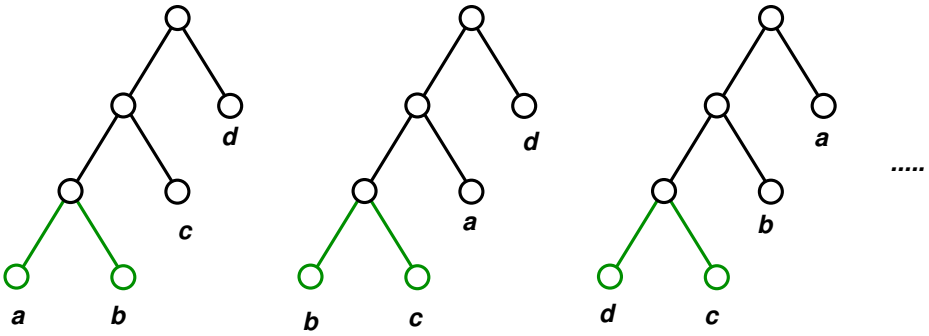


Figura 3.38: Posibles tablas de códigos que se obtienen permutando las letras en las hojas de una estructura dada.

Notemos que hasta ahora hemos considerado solamente *"la estructura del árbol"*. Los códigos se obtienen asignando cada uno de los n_c caracteres a una de las hojas del árbol. Entonces, por cada uno de los árboles de T_4 a T_9 se pueden obtener $4! = 24$ posibles tablas de códigos permutando los caracteres entre sí. Por ejemplo, los tres árboles de la figura 3.38 son algunos de los árboles que se obtienen de la estructura de T_4 en la figura 3.37 permutando las letras entre sí. Como el número de permutaciones de n_c caracteres es $n_c!$, tenemos que en total hay a lo sumo $(n_c - 1)!n_c!$ posibles tablas de códigos. Usando las herramientas desarrolladas en la sección §1.3

(en particular §1.3.9), vemos que

$$T(n_c) = (n_c - 1)!n_c! < (n_c!)^2 = O(n_c^{2n_c}) \tag{3.24}$$

Repasando la experiencia que tuvimos con la velocidad de crecimiento de los algoritmos no polinomiales (ver §1.1.6), podemos descartar esta estrategia si pensamos aplicar el algoritmo a mensajes con $n_c > 20$.

3.8.6.4.1. Generación de los árboles Una forma de generar todos los árboles posibles es usando recursión. Dados una serie de árboles $T_0, T_1, \ldots, T_{n-1}$, llamaremos $\mathrm{comb}(T_0, T_1, \ldots, T_{n-1})$ a la lista de todos los posibles árboles formados combinando $T_0, \ldots, T_{n-1}$. Para $n = 2$ hay una sola combinación que consiste en poner a T_0 como hijo izquierdo y T_1 como hijo derecho de un nuevo árbol. Para aplicar el algoritmo de búsqueda exhaustiva debemos poder generar la lista de árboles $\mathrm{comb}(T_0, T_1, \ldots, T_{n_{\mathrm{char}}-1})$ donde T_j es un árbol que contiene un sólo nodo con el caracter j-ésimo. Todas las combinaciones se pueden hallar tomando cada uno de los posibles pares de árboles (T_i, T_j) de la lista, combinándolo e insertando $\mathrm{comb}(T_i, T_j)$ en la lista, después de eliminar los árboles originales.

$$\begin{aligned}\mathrm{comb}(T_0, T_1, T_2, T_3) = (&\mathrm{comb}(\mathrm{comb}(T_0, T_1), T_2, T_3),\\ &\mathrm{comb}(\mathrm{comb}(T_0, T_2), T_1, T_3),\\ &\mathrm{comb}(\mathrm{comb}(T_0, T_3), T_1, T_2),\\ &\mathrm{comb}(\mathrm{comb}(T_1, T_2), T_0, T_3),\\ &\mathrm{comb}(\mathrm{comb}(T_1, T_3), T_0, T_2),\\ &\mathrm{comb}(\mathrm{comb}(T_2, T_3), T_0, T_1))\end{aligned} \tag{3.25}$$

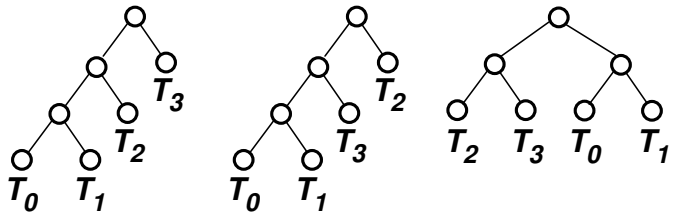


Figura 3.39: Generación de todos los posibles árboles binarios llenos de 4 hojas.

Ahora, recursivamente, cada uno de las sublistas se expande a su vez en 3 árboles, por ejemplo

$$\begin{aligned}\mathrm{comb}(\mathrm{comb}(T_0,T_1),T_2,T_3) = (&\mathrm{comb}(\mathrm{comb}(\mathrm{comb}(T_0,T_1),T_2),T_3),\\ &\mathrm{comb}(\mathrm{comb}(\mathrm{comb}(T_0,T_1),T_3),T_2),\\ &\mathrm{comb}(\mathrm{comb}(T_2,T_3),\mathrm{comb}(T_0,T_1)))\end{aligned} \tag{3.26}$$

Estos tres posibles combinaciones pueden observarse en la figura 3.39. Para la figura hemos asumido que los árboles originales son nodos simples, pero podrían ser a su vez árboles.

Entonces, cada una de las sublistas en (3.25) genera 3 árboles. En general vamos a tener

$$N_{\mathrm{abc}}(n) = (\text{número de posibles pares a escoger de } n \text{ árboles}) \times N_{\mathrm{abc}}(n-1) \tag{3.27}$$

donde $N_{\mathrm{abc}}(n)$ es el número de árboles binarios llenos de n hojas. El número de posibles pares a escoger de n árboles es $n(n-1)/2$. La división por 2 proviene de que no importa el orden entre los elementos del par. De manera que

$$N_{\mathrm{abc}}(n) = \frac{n(n-1)}{2} N_{\mathrm{abc}}(n-1) \tag{3.28}$$

Aplicando recursivamente esta relación llegamos a

$$\begin{aligned} N_{\mathrm{abc}}(n) &= \frac{n(n-1)}{2}\,\frac{(n-1)(n-2)}{2}\cdots\frac{3\cdot 2}{2} \\ &= \frac{n!(n-1)!}{2^{n+1}} = \frac{(n!)^2}{n\,2^{n+1}} \end{aligned} \tag{3.29}$$

Usando la aproximación de Stirling (1.34) llegamos a

$$N_{\mathrm{abc}}(n) = O\left(\frac{n^{2n-1}}{2^n}\right) \tag{3.30}$$

Notar que esta estimación es menor que la obtenida en la sección anterior (3.24) ya que en aquella estimación se cuentan dos veces árboles que se obtienen intercambiando los hijos.

3.8.6.4.2. Agregando un condimento de programación funcional

```
1. typedef void (*traverse_tree_fun) (btree_t &T, void *data);
```

```
2.
3.  typedef list< btree<int> > list_t;
4.  typedef list_t::iterator pos_t;
5.
6.  void comb(list_t &L,traverse_tree_fun f,void *data=NULL) {
7.   if (L.size()==1) {
8.    f(*L.begin(),data);
9.    return;
10.  }
11.  int n=L.size();
12.  for (int j=0; j<n-1; j++) {
13.   for (int k=j+1; k<n; k++) {
14.    btree_t T;
15.    T.insert(T.begin(),-1);
16.    node_t m = T.begin();
17.
18.    pos_t pk=L.begin();
19.    for (int kk=0; kk<k; kk++) pk++;
20.    T.splice(m.left(),pk->begin());
21.    L.erase(pk);
22.
23.    pos_t pj=L.begin();
24.    for (int jj=0; jj<j; jj++) pj++;
25.    T.splice(m.right(),pj->begin());
26.    L.erase(pj);
27.
28.    pos_t p = L.insert(L.begin(),btree_t());
29.    p->splice(p->begin(),T.begin());
30.
31.    comb(L,f,data);
32.
33.    p = L.begin();
34.    m = T.splice(T.begin(),p->begin());
35.    L.erase(p);
36.
37.    pj=L.begin();
38.    for (int jj=0; jj<j; jj++) pj++;
39.    pj = L.insert(pj,btree_t());
40.    pj->splice(pj->begin(),m.right());
41.
42.    pk=L.begin();
43.    for (int kk=0; kk<k; kk++) pk++;
44.    pk = L.insert(pk,btree_t());
45.    pk->splice(pk->begin(),m.left());
46.
47.   }
48.  }
```

```
49. }
```

Código 3.20: *Implementación del algoritmo que genera todos los árboles llenos. [Archivo: allabc.cpp]*

En el código 3.20 vemos una implementación del algoritmo descripto, es decir la función `comb(L,f)` genera todas las posibles combinaciones de árboles contenidos en la lista `L` y les aplica una función `f`. Este es otro ejemplo de *"programación funcional"*, donde uno de los argumentos de la función es, a su vez, otra función. Notemos que no cualquier función puede pasarse a `comb()` sino que tiene que responder a la *"signatura"* dada en la línea 1. Recordemos que la signatura de una función son los tipos de los argumentos de una función. En este caso el tipo `traverse_tree_fun` consiste en una función que retorna `void` y toma como argumentos una referencia a un árbol binario y puntero genérico `void *`. Cualquier función que tenga esa signatura podrá ser pasada a `comb()`.

En nuestro caso, debemos pasar a `comb()` una función que calcula la longitud promedio de código para un dado árbol y retenga el mínimo de todos aquellos árboles que fueron visitados hasta el momento. Como la `f` debe calcular el mínimo de todas las longitudes de código debemos guardar el estado actual del cálculo (la mínima longitud de código calculada hasta el momento) en una variable global. Para evitar el uso de variables globales, que son siempre una fuerte de error, utilizamos una variable auxiliar `void *data` que contiene el estado del cálculo y es pasado a `f` en las sucesivas llamadas.

De esta forma dividimos el problema en dos separados. Primero escribir un algoritmo `comb()` que recorre todos los árboles y le aplica una función `f`, pasándole el estado de cálculo actual `void *data` y, segundo, escribir la función `f` apropiada para este problema. Notar que de esta forma el algoritmo `comb()` es sumamente genérico y puede ser usado para otras aplicaciones que requieren recorrer todos los posibles árboles binarios llenos.

Una estrategia más simple sería escribir una función basada en este mismo algoritmo que genere todos los árboles posibles en una lista. Luego se recorre la lista calculando la longitud media para cada árbol y reteniendo la menor. La implementación propuesta es mucho más eficiente en uso de memoria ya que en todo momento sólo hay un árbol generado y también de tiempo de cálculo ya que en esta estrategia más simple los árboles se deben ir combinando por copia, mientras que en la propuesta el mismo árbol va pasando de una forma a la otra con simples operaciones de `splice()`.

3.8.6.4.3. El algoritmo de combinación El algoritmo verifica primero la condición de terminación de la recursión. Cuando la longitud de la lista es 1, entonces ya tenemos uno de los árboles generados y se le aplica la función **f**. Caso contrario el algoritmo prosigue generando $n(n-1)/2$ listas de árboles que se obtienen combinando alguno de los pares posibles T_i, T_j y reemplazando T_i, T_j por la combinación $\mathrm{comb}(T_i, T_j)$. Una posibilidad es para cada par generar una copia de la lista de árboles, generar la combinación y reemplazar y llamar recursivamente a **comb()**. Pero esto resultaría en copiar toda la lista de árboles (incluyendo la copia de los árboles mismos), lo cual resultaría en un costo excesivo. En la implementación presentada esto se evita aplicando la transformación en las líneas 14–29 y deshaciendo la transformación en las líneas 33–45, después de haber llamado recursivamente a la función.

Detengámonos en la combinación de los árboles. Notemos que el lazo externo se hace sobre enteros, de manera que tanto para **j** como para **k** hay que realizar un lazo sobre la lista para llegar a la posición correspondiente. (Los lazos sobre **kk** y **jj**). En ambos casos obtenemos las posiciones correspondientes en la lista **pj** y **pk**. Para hacer la manipulación creamos un árbol temporario **T**. Insertamos un nodo interior con valor **-1** para diferenciar de las hojas, que tendrán valores no negativos (entre **0** y **nchar-1**). Cada uno de los árboles es movido (con **splice()**) a uno de los hijos de la raíz del árbol **T**. Notar que como **pj** y **pk** son iterators, los árboles correspondientes son ***pj** y ***pk** y para tomar la raíz debemos hacer **pj->begin()** y **pk->begin()**. Después de mover los árboles, la posición es eliminada de la lista con el **erase()** de listas. Notar que primero se elimina **k**, que es mayor que **j**. Si eliminàramos primero **j** entonces el lazo sobre **kk** debería hacerse hasta **k-1** ya que todas las posiciones después de **j** se habrían corrido. Finalmente una nueva posición es insertada en el comienzo de la lista (cualquier posición hubiera estado bien, por ejemplo **end()**) y todo el árbol **T** es movido (con **splice()**) al árbol que está en esa posición.

Después de llamar a **comb()** recursivamente, debemos volver a *"desarmar"* el árbol que está en la primera posición de **L** y retornar los dos subárboles a las posiciones **j** y **k**. Procedemos en forma inversa a la combinación. Movemos todo el árbol en **L.begin()** a **T** y eliminamos la primera posición. Luego buscamos la posición **j**-ésima con un lazo e insertamos un árbol vacío, moviendo todo la rama derecha a esa posición. Lo mismo se hace después con la rama izquierda. La lista de árboles **L** debería quedar después de la línea 45 igual a la que comenzó el lazo en la línea 14, de manera que en realidad todo **comb()** no debe modificar a **L**. Notar que esto

es a su vez necesario para poder llamar a **comb** en la línea 31, ya que si lo modificara no sería posible después desarmar la combinación.

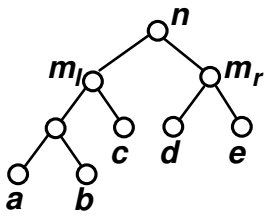


Figura 3.40: Cálculo de la longitud media del código.

3.8.6.4.4. Función auxiliar que calcula la longitud media

```
1.  double codelen(btree_t &T,node_t n,
2.             const vector<double> &prob,double &w) {
3.    if (n.left()==T.end()) {
4.      w = prob[*n];;
5.      return 0.;
6.    } else {
7.      double wl,wr,lr,ll;
8.      ll = codelen(T,n.left(),prob,wl);
9.      lr = codelen(T,n.right(),prob,wr);
10.     w = wr+wl;
11.     return wl+wr+ll+lr;
12.   }
13. }
14.
15. double codelen(btree_t &T,
16.            const vector<double> &prob) {
17.   double ww;
18.   return codelen(T,T.begin(),prob,ww);
19. }
```

Código 3.21: *Cálculo de la longitud media del código. [Archivo: codelen.cpp]*

Consideremos el ejemplo del código de la figura 3.40. Tomando como base la expresión para la longitud media dada por (3.23) definimos la cantidad $\mathrm{cdln}(n)$ como

$$\mathrm{cdln}(n) = P(a)\,3 + P(b)\,3 + P(c)\,2 + P(d)\,2 + P(e)\,2 \tag{3.31}$$

es decir la suma de los productos de las probabilidades de las hojas del árbol por su distancia al nodo en cuestión. En el caso de ser n la raíz del árbol, $\mathrm{cdln}(n)$ es igual a la longitud media. Para nodos que no son la raíz el resultado no es la longitud media del subárbol correspondiente ya que la suma de las probabilidades en el subárbol es diferente de 1. Con un poco de álgebra podemos hallar una expresión recursiva

$$\begin{aligned}\mathrm{cdln}(n) &= P(a)\,3 + P(b)\,3 + P(c)\,2 + P(d)\,2 + P(e)\,2 \\ &= [P(a) + P(b) + P(c)] + [P(a)\,2 + P(b)\,2 + P(c)\,1] \\ &\quad + [P(d) + P(e)] + [P(d)\,1 + P(e)\,1] \\ &= P(m_l) + \mathrm{cdln}(m_l) + P(m_r) + \mathrm{cdln}(m_r)\end{aligned} \tag{3.32}$$

donde

$$\begin{aligned}P(m_l) &= P(a) + P(b) + P(c) \\ P(m_r) &= P(d) + P(e)\end{aligned} \tag{3.33}$$

son la suma de probabilidades de la rama izquierda y derecha y

$$\begin{aligned}\mathrm{cdln}(m_l) &= P(a)\,2 + P(b)\,2 + P(c)\,1 \\ \mathrm{cdln}(m_r) &= P(d)\,1 + P(e)\,1\end{aligned} \tag{3.34}$$

son los valores de la función **cdln** en los hijos izquierdo y derecho.

En el código 3.21 se observa la función que calcula, dado el árbol binario y el vector de probabilidades **prob** la longitud media del código. Esta es una típica función recursiva como las ya estudiadas. La función **codelen(T,n,prob,w)** retorna la función cdln aplicada al nodo **n** aplicando la expresión recursiva, mientras que por el argumento **w** retorna la suma de las probabilidades de todas las hojas del subárbol. Para el nodo raíz del árbol **codelen()** coincide con la longitud media del código. La recursión se corta cuando **n** es una hoja, en cuyo caso **w** es la probabilidad de la hoja en cuestión (lo cual se obtiene de **prob**) y **cdln** es 0.

La función wrapper **codelen(T,prob)** se encarga de pasar los argumentos apropiados a la función recursiva auxiliar.

3.8.6.4.5. Uso de comb y codelen

```
1. struct huf_exh_data {
2.   btree_t best;
3.   double best_code_len;
4.   const vector<double> *prob;
5. };
```

```
6.
7.  void min_code_len(btree_t &T,void *data) {
8.    huf_exh_data *hp = (huf_exh_data *)data;
9.    double l = codelen(T,*(hp->prob));
10.   if (l < hp->best_code_len) {
11.     hp->best_code_len = l;
12.     hp->best = T;
13.   }
14. }
15.
16. //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
17. void
18. huffman_exh(const vector<double> &prob,btree_t &T) {
19.   int nchar = prob.size();
20.   list_t L;
21.   pos_t p;
22.   huf_exh_data h;
23.   h.prob = &prob;
24.   h.best_code_len = DBL_MAX;
25.   for (int j=0; j<nchar; j++) {
26.     p = L.insert(L.end(),btree_t());
27.     p->insert(p->begin(),j);
28.   }
29.   comb(L,&min_code_len,&h);
30.   T.clear();
31.   T.splice(T.begin(),h.best.begin());
32. }
```

Código 3.22: *Función auxiliar para pasar a* `comb()` *[Archivo: hufexh.cpp]*

Ahora debemos escribir una función auxiliar con signatura `void min_code_len(btree_t &T, void *data);` para pasar a `comb()`. Esta debe encargarse de calcular la longitud media y mantener el mínimo de todas las longitudes medias encontradas y el árbol correspondiente. Llamamos a esto el *"estado del cálculo"* y para ello definimos una estructura auxiliar `huf_exh_data`. Además debemos pasar el vector de probabilidades `prob`. Para evitar de pasarlo por una variable global agregamos a `huf_exh_data` un puntero al vector.

Como `comb()` es genérica el estado global se representa por un puntero genérico `void *`. El usuario debe encargarse de convertir el puntero a un puntero a su estructura real, a través de una *"conversión estática"* (*"static cast"*). Esto se hace en la línea 8 donde se obtiene un puntero a la estructura

real de tipo **huf_exh_data** que está detrás del puntero genérico. **T** es una referencia a uno de los árboles generados. Recordemos que **comb()** va a ir llamando a nuestra función **min_code_len()** con cada uno de los árboles que genere. En **min_code_len** simplemente calculamos la longitud promedio con **codelen()** y si es menor que el mínimo actual actualizamos los valores.

Finalmente, la función **huffman_exh(prob,T)** calcula en forma exhaustiva el árbol de menor longitud media usando las herramientas desarrolladas. Declara una lista de árboles **L** e inserta en ellos **nchar** árboles que constan de un sólo nodo (hoja) con el caracter correspondiente (en realidad un entero de 0 a **nchar-1**). También declara la estructura **h** que va a representar el estado del cálculo e inicializa sus valores, por ejemplo inicializa **best_code_len** a **DBL_MAX** (un doble muy grande). Luego llama a **comb()** el cual va a llamar a **min_code_len()** con cada uno de los árboles y un puntero al estado **h**. Una vez terminado **comb()** en **h.best** queda el árbol con el mejor código. Este lo movemos al árbol de retorno **T** con **splice()**.

3.8.6.5. El algoritmo de Huffman

Un algoritmo que permite obtener tablas de código óptimas en tiempos reducidos es el *"algoritmo de Huffman"*. Notemos que es deseable que los caracteres con mayor probabilidad (los más *"pesados"*) deberían estar cerca de la raíz, para tener un código lo más corto posible. Ahora bien, para que un caracter pesado pueda *"subir"*, es necesario que otros caracteres (los más livianos) *"bajen"*. De esta manera podemos pensar que hay una competencia entre los caracteres que tratan de estar lo más cerca posible de la raíz, pero tienden a *"ganar"* los más pesados. Caracteres con probabilidades similares deberían tender a estar en niveles parecidos. Esto sugiere ir apareando caracteres livianos con pesos parecidos en árboles que pasan a ser caracteres *"comodines"* que representan a un conjunto de caracteres.

Ejemplo 3.3: *Consigna:* Dados los caracteres a, b, c, d, e, f con pesos $P(a) = 0.4$, $P(b) = 0.15$, $P(c) = 0.15$, $P(d) = 0.1$, $P(e) = 0.1$, $P(f) = 0.1$, encontrar una tabla óptima de codificación, usando el algoritmo de Huffman.

El algoritmo procede de la siguiente manera,

1. Inicialmente se crean n_c árboles (tantos como caracteres hay). Los árboles tienen una sola hoja asociada con cada uno de los caracteres. Al árbol se le asocia un peso total, inicialmente cada árbol tiene el peso del caracter correspondiente.

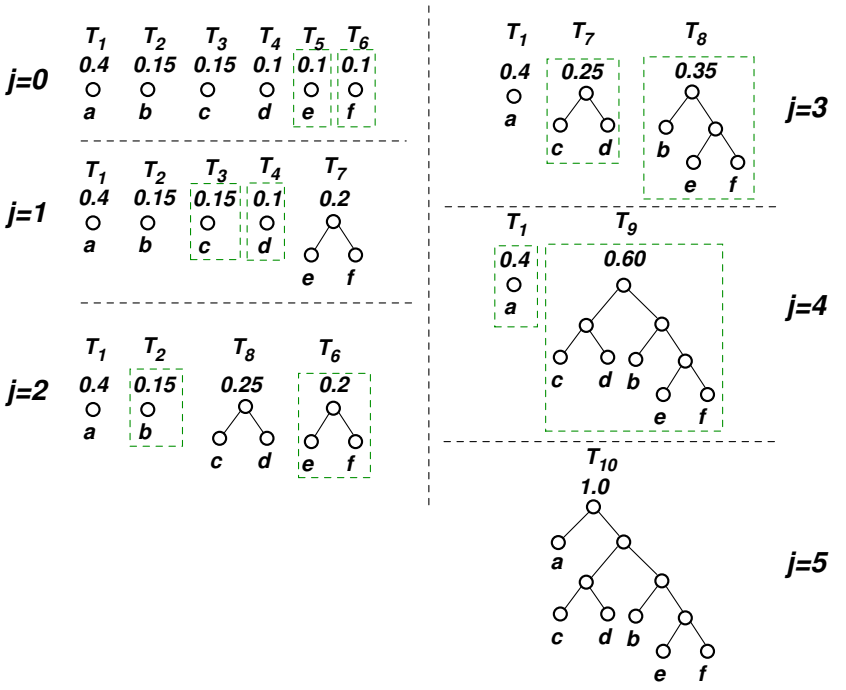


Figura 3.41: Algoritmo de Huffman.

2. En sucesivas iteraciones el algoritmo va combinando los dos árboles con menor peso en uno sólo. Si hay varias posibilidades se debe escoger aquella que da el menor peso del árbol combinado. Como en cada combinación desaparecen dos árboles y aparece uno nuevo en $n_c - 1$ iteraciones queda un sólo árbol con el código resultante.

Por ejemplo, en base a los datos del problema se construyen los 6 árboles de la figura 3.41 (marcados con $j = 0$, j es el número de iteración). De los 6 árboles, los de menor peso son los T_4, T_5 y T_6 correspondientes a los caracteres d, e y f. Todos tienen la misma probabilidad 0.1. *Cuando varios árboles tienen la misma probabilidad se puede tomar cualquiera dos de ellos*. En el ejemplo elegimos T_5 y T_6. Estos dos árboles se combinan como subárboles de un nuevo árbol T_7, con un peso asociado 0.2, igual a la suma de los pesos de los dos árboles combinados. Después de esta operación quedan sólo los árboles T_1, T_2, T_3, T_4 y T_7 con pesos 0.4, 0.15, 0.15, 0.1 y 0.2. Notar que en tanto antes como después de la combinación, la suma de los pesos de los árboles debe dar 1 (como debe ocurrir siempre con las

probabilidades).

Ahora los árboles de menor peso son T_2, T_3 y T_4 con pesos 0.15, 0.15 y 0.1. Cualquiera de las combinaciones T_2 y T_4 o T_3 y T_4 son válidas, dando una probabilidad combinada de 0.25. Notar que la combinación T_2 y T_3 no es válida ya que daría una probabilidad combinada 0.3, mayor que las otras dos combinaciones. En este caso elegimos T_3 y T_4 resultando en el árbol T_8 con probabilidad combinada 0.25. En las figuras se observa las siguientes etapas, hasta llegar al árbol final T_{10}.

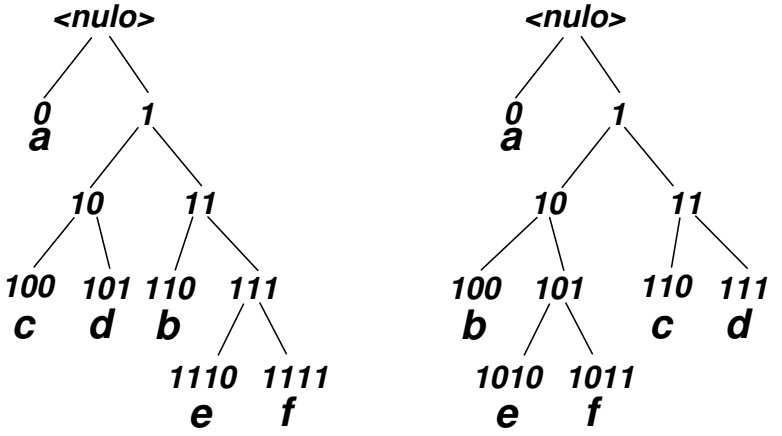


Figura 3.42: Arboles de códigos resultante del algoritmo de Huffman.

En la figura 3.42 se observa a la izquierda el árbol con los códigos correspondientes. La longitud media del código es, usando (3.23),

$$\langle l \rangle = 0.4{\times}1{+}0.15{\times}3{+}0.15{\times}3{+}0.1{\times}3{+}0.1{\times}5{+}0.1{\times}4 = 2.5\,\mathrm{bits/caracter}. \tag{3.35}$$

resultando en una ganancia del 17 % con respecto a la longitud media de 3 que correspondería a la codificación con códigos de igual longitud dada por (3.22).

Notemos que al combinar dos árboles entre sí no estamos especificando cuál queda como hijo derecho y cuál como izquierdo. Por ejemplo, si al combinar T_7 con T_8 para formar T_9 dejamos a T_7 a la derecha, entonces el árbol resultante será el mostrado en la figura 3.42 a la derecha. Si bien el árbol resultante (y por lo tanto la tabla de códigos) es diferente, *la longitud media del código será la misma*, ya que la profundidad de cada caracter, que es su longitud de código, es la misma en los dos árboles. Por ejemplo, e y f tienen longitud 4 en los dos códigos.

A diferencia de los algoritmos heurísticos, la tabla de códigos así generada es *"óptima"* (la mejor de todas), y la longitud media por caracter coincide con la que se obtendría aplicando el algoritmo de búsqueda exhaustiva pero a un costo mucho menor.

3.8.6.6. Implementación del algoritmo

```
1.  struct huffman_tree {
2.    double p;
3.    btree<int> T;
4.  };
5.
6.  void
7.  huffman(const vector<double> &prob,btree<int> &T) {
8.    typedef list<huffman_tree> bosque_t;
9.
10.   // Contiene todos los arboles
11.   bosque_t bosque;
12.   // Numero de caracteres del codigo
13.   int N = prob.size();
14.   // Crear los arboles iniciales poniendolos en
15.   // una lista Los elementos de la lista contienen
16.   // la probabilidad de cada caracter y un arbol
17.   // con un solo nodo. Los nodos interiores del
18.   // arbol tienen un -1 (es solo para
19.   // consistencia) y las hojas tienen el indice
20.   // del caracter. (entre 0 y N-1)
21.   for (int j=0; j<N; j++) {
22.     // Agrega un nuevo elemento a la lista
23.     bosque_t::iterator htree =
24.       bosque.insert(bosque.begin(),huffman_tree());
25.     htree->p = prob[j];
26.     htree->T.insert(htree->T.begin(),j);
27.   }
28.
29.   // Aqui empieza el algoritmo de Huffman.
30.   // Tmp va a contener el arbol combinado
31.   btree<int> Tmp;
32.   for (int j=0; j<N-1; j++) {
33.     // En la raiz de Tmp (que es un nodo interior)
34.     // ponemos un -1 (esto es solo para chequear).
35.     Tmp.insert(Tmp.begin(),-1);
36.     // Tmp_p es la probabilidad del arbol combinado
37.     // (la suma de las probabilidades de los dos subarboles)
```

```
38.     double Tmp_p = 0.0;
39.     // Para 'k=0' toma el menor y lo pone en el
40.     // hijo izquierdo de la raiz de Tmp. Para 'k=1' en el
41.     // hijo derecho.
42.     for (int k=0; k<2; k++) {
43.       // recorre el 'bosque' (la lista de arboles)
44.       // busca el menor. 'qmin' es un iterator al menor
45.       bosque_t::iterator q = bosque.begin(), qmin=q;
46.       while (q != bosque.end()) {
47.         if (q->p < qmin->p) qmin = q;
48.         q++;
49.       }
50.       // Asigna a 'node' el hijo derecho o izquierdo
51.       // de la raiz de 'Tmp' dependiendo de 'k'
52.       btree<int>::iterator node = Tmp.begin();
53.       node = (k==0 ? node.left() : node.right());
54.       // Mueve todo el nodo que esta en 'qmin'
55.       // al nodo correspondiente de 'Tmp'
56.       Tmp.splice(node,qmin->T.begin());
57.       // Acumula las probabilidades
58.       Tmp_p += qmin->p;
59.       // Elimina el arbol correspondiente del bosque.
60.       bosque.erase(qmin);
61.     }
62.     // Inserta el arbol combinado en el bosque
63.     bosque_t::iterator r =
64.       bosque.insert(bosque.begin(),huffman_tree());
65.     // Mueve todo el arbol de 'Tmp' al nodo
66.     // recien insertado
67.     r->T.splice(r->T.begin(),Tmp.begin());
68.     // Pone la probabilidad en el elemento de la
69.     // lista
70.     r->p = Tmp_p;
71.   }
72.   // Debe haber quedado 1 solo elemento en la lista
73.   assert(bosque.size()==1);
74.   // Mueve todo el arbol que quedo a 'T'
75.   T.clear();
76.   T.splice(T.begin(),bosque.begin()->T.begin());
77. }
```

Código 3.23: *Implementación del algoritmo de Huffman. [Archivo: huf.cpp]*

En el el código 3.23 vemos una implementación del algoritmo de Huffman usando la interfaz avanzada mostrada en el código 3.17.

- El código se basa en usar una lista de estructuras de tipo `huffman_tree`, definida en las líneas 1-4 que contienen el árbol en cuestión y su probabilidad `p`.
- Los árboles son de tipo `btree<int>`. En los valores nodales almacenaremos para las hojas un índice que identifica al caracter correspondiente. Este índice va entre 0 y `N-1` donde `N` es el número de caracteres. `prob` es un vector de dobles de longitud `N`. `prob[j]` es la probabilidad del caracter `j`. En los valores nodales de los nodos interiores del árbol almacenaremos un valor `-1`. Este valor no es usado normalmente, sólo sirve como un chequeo adicional.
- El tipo `bosque_t` definido en la línea 8 es un alias para una lista de tales estructuras.
- La función `huffman(prob,T)` toma un vector de dobles (las probabilidades) `prob` y calcula el árbol de Huffman correspondiente.
- En el lazo de la líneas 21–27 los elementos del bosque son inicializados, insertando el único nodo
- El lazo de las líneas 31–77 es el lazo del algoritmo de Huffman propiamente dicho. En cada paso de este lazo se toman los dos árboles de `bosque` con probabilidad menor. Se combinan usando `splice` en un árbol auxiliar `Tmp`. Los dos árboles son eliminados de la lista y el combinado con la suma de las probabilidades es insertado en el bosque.
- Inicialmente `Tmp` está vacío, y dentro del lazo, la última operación es hacer un `splice` de todo `Tmp` a uno de los árboles del bosque, de manera que está garantizado que al empezar el lazo `Tmp` *siempre* está vacío.
- Primero insertamos en `Tmp` un nodo interior (con valor -1). Los dos árboles con pesos menores quedarán como hijos de este nodo raíz.
- La variable `Tmp_p` es la probabilidad combinada. Al empezar el cuerpo del lazo es inicializada a 0 y luego le vamos acumulando las probabilidades de cada uno de los dos árboles a combinar.
- Para evitar la duplicación de código hacemos la búsqueda de los dos menores dentro del lazo sobre `k`, que se hace para `k=0` y `k=1`. Para `k=0` se encuentra el árbol del bosque con menor probabilidad y se inserta en el subárbol izquierdo de `Tmp`. Para `k=1` se inserta al segundo menor en el hijo derecho de de `Tmp`.
- La búsqueda del menor se hace en el lazo de las líneas 46–49. Este es un lazo sobre listas. `q` es un iterator a `list<huffman_tree>` de manera que `*q` es de tipo `huffman_tree` y la probabilidad correspondiente está en `(*q).p` o, lo que es lo mismo `q->p`. Vamos guardando la posición

en la lista del mínimo actual en la posición **qmin**.

- En las líneas 52–53 hacemos que el iterator **node** apunte primero al nodo raíz de **Tmp** y después al hijo izquierdo o derecho, dependiendo de **k**.
- El árbol con menor probabilidad es pasado del elemento del bosque al subárbol correspondiente de **Tmp** en el splice de la línea 56. Recordar que, como el elemento de la lista es de tipo **huffman_tree**, el árbol está en **qmin->T**. El elemento de la lista es borrado en la línea 60.
- Las probabilidades de los dos subárboles se van acumulando en la línea 58.
- Después de salir del lazo debe quedar en el bosque un solo árbol. **bosque.begin()** es un iterator al primer (y único) elemento del bosque. De manera que el árbol está en **bosque.begin()->T**. La probabilidad correspondiente debería ser 1. El **splice** de la línea 76 mueve todo este subárbol al valor de retorno, el árbol **T**.

3.8.6.7. Un programa de compresión de archivos

```
1.  void
2.  huffman_codes(btree<int> &T,btree<int>::iterator node,
3.            const vector<double> &prob,
4.            codigo_t &codigo, vector<codigo_t> &codigos) {
5.    // 'codigo' es el codigo calculado hasta node.
6.    // La funcion se va llamando recursivamente y a
7.    // medida que va bajando en el arbol va
8.    // agregando bits al codigo.
9.    if (*node>=0) {
10.     // Si es una hoja directamente inserta un
11.     // codigo en 'codigos'
12.     codigos[*node] = codigo;
13.     return;
14.   } else {
15.     // Le va pasando 'codigo' a los hijos los
16.     // cuales van agregando codigos a 'codigos'.
17.     // 'codigo' se va pasando por referencia de
18.     // manera que las llamadas recursivas lo deben
19.     // dejar tal como estaba. Por eso, despues
20.     // despues de agregar un 0 hay que sacarlo
21.     // y lo mismo con el 1.
22.     codigo.push_back(0);
23.     huffman_codes(T,node.left(),prob,codigo,codigos);
24.     codigo.pop_back();
```

```
25.
26.     codigo.push_back(1);
27.     huffman_codes(T,node.right(),prob,codigo,codigos);
28.     codigo.pop_back();
29.     return;
30.   }
31. }
32.
33. void
34. huffman_codes(btree<int> &H,const vector<double> &prob,
35.              vector<codigo_t> &codigos) {
36.   // Este es el codigo de un caracter en particular. Es
37.   // pasado por referencia, de manera que hay una sola instancia
38.   // de codigo.
39.   codigo_t codigo;
40.   huffman_codes(H,H.begin(),prob,codigo,codigos);
41. }
42.
43. const int NB = 8;
44. const int bufflen = 1024;
45.
46. void qflush(queue<char> &Q, queue<char_t> &Qbytes,
47.            int &nbits) {
48.   // Convierte 'NB' bytes de 'Q' a un char.
49.   // Si 'Q' queda viacia entonces rellena con 0's.
50.   char_t c=0;
51.   for (int j=0; j<NB; j++) {
52.     int b = 0;
53.     if (!Q.empty()) {
54.       b = Q.front();
55.       Q.pop();
56.       nbits++;
57.     }
58.     c <<= 1;
59.     if (b) c |= 1;
60.     else c &= ~1;
61.   }
62.   Qbytes.push(c);
63. }
64.
65. void bflush(queue<char_t> &Qbytes,
66.            vector<char_t> &buff,int &nbits,
67.            FILE *zip) {
68.   // Numero de bits a ser escrito
69.   int nb = nbits;
70.   if (nb>bufflen*NB) nb = bufflen*NB;
```

```
71.   nbits -= nb;
72.   // Guarda en el archivo la longitud del siguiente bloque
73.   fwrite(&nb,sizeof(int),1,zip);
74.   // Pone en el buffer los 'nb' bits
75.   int nbytes = 0;
76.   while (nb>0) {
77.     buff[nbytes++] = Qbytes.front();
78.     Qbytes.pop();
79.     nb -= NB;
80.   }
81.   fwrite(&buff[0],sizeof(char_t),nbytes,zip);
82. }
83.
84. void hufzip(char *file,char *zipped) {
85.   // Abre el archivo a compactar
86.   FILE *fid;
87.   if (file) {
88.     fid = fopen(file,"r");
89.     assert(fid);
90.   } else fid = stdin;
91.   // Numero total de caracteres posibles. Consideramos
92.   // caracteres de 8 bits, es decir que puede haber 256
93.   // caracteres
94.   const int NUMCHAR=256;
95.   // table[j] va a ser el numero de veces que aparece el
96.   // caracter 'j' en el archivo. De manera que la
97.   // probabilidad del caracter es prob[j] =
98.   // table[j]/(sum_k=0^numchar table[k]) indx[j] es el
99.   // indice asignado al caracter 'j'. Si el caracter 'j' no
100.  // aparece en el archivo entonces hacemos indx[j]=-1 y si
101.  // no le asignamos un numero correlativo de 0 a N-1. N es
102.  // el numero total de caracteres distintos que aprecen en
103.  // el archivo.
104.  vector<int> table(NUMCHAR),indx(NUMCHAR);
105.  // Ponemos los caracteres en una cola de 'char_t'
106.  queue<char_t> fin;
107.  // Contador de cuantos caracteres hay en el archivo
108.  int n = 0;
109.  while(1) {
110.    int c = getc(fid);
111.    if (c==EOF) break;
112.    fin.push(c);
113.    assert(c<NUMCHAR);
114.    n++;
115.    table[c]++;
116.  }
117.  fclose(fid);
```

```
118.  // Detecta cuantos caracteres distintos hay fijandose en
119.  // solo aquellos que tienen table[j]>0. Define prob[k] que
120.  // es la probabilidad de aparecer del caracter con indice
121.  // 'k'
122.  int N=0;
123.  // prob[k] es la probabilidad correspondiente
124.  // al caracter de indice k
125.  vector<double> prob;
126.  // 'letters[k]' contiene el caracter (de 0 a NUMCHAR-1)
127.  // correspondiente al indice 'k'
128.  vector<char_t> letters;
129.  for (int j=0; j<NUMCHAR; j++) {
130.    if (table[j]) {
131.      double p = double(table[j])/double(n);
132.      indx[j] = N++;
133.      letters.push_back((char_t)j);
134.      prob.push_back(p);
135.    } else indx[j] = -1;
136.  }
137.
138.  // H va a contener al arbol de codigos
139.  btree<int> H;
140.  // Calcula el arbol usando el algoritmo de Huffman
141.  huffman(prob,H);
142.
143.  // Construye la tabla de codigos. 'codigos[j]' va a ser
144.  // un vector de enteros (bits)
145.  vector<codigo_t> codigos(N);
146.  // Calcula la tabla de codigos y la longitud media.
147.  huffman_codes(H,prob,codigos);
148.
149.
150.  // Abre el archivo zippeado
151.  FILE *zip;
152.  if (zipped) {
153.    zip = fopen(zipped,"w");
154.    assert(zip);
155.  } else zip = stdout;
156.
157.  // Guarda encabezamiento en archivo zippeado conteniendo
158.  // las probabilidades para despues poder reconstruir el arbol
159.  for (int j=0; j<N; j++) {
160.    fwrite(&prob[j],sizeof(double),1,zip);
161.    fwrite(&letters[j],sizeof(char_t),1,zip);
162.  }
163.  // Terminador (probabilidad negativa)
164.  double p = -1.0;
```

```
165.   fwrite(&p,sizeof(double),1,zip);
166.
167.   vector<char_t> buff(bufflen);
168.   // Cantidad de bits almacenados en buff
169.   int nbits=0;
170.
171.   // Zippea. Va convirtiendo los caracteres de 'fin' en
172.   // codigos y los inserta en la cola 'Q', o sea que 'Q'
173.   // contiene todos elementos 0 o 1. Por otra parte va sacan
174.   // dode a 8 bits de Q y los convierte en un byte en
175.   // 'Qbytes'. O sea que 'Qbytes' contiene caracteres que
176.   // pueden tomar cualquier valor entre 0 y NUMCHAR-1.
177.   queue<char> Q;
178.   queue<char_t> Qbytes;
179.   assert(fid);
180.   while(!fin.empty()) {
181.     // Va tomando de a un elemento de 'fin' y pone todo el
182.     // codigo correspondiente en 'Q'
183.     int c = fin.front();
184.     fin.pop();
185.     assert(c<NUMCHAR);
186.     int k = indx[c];
187.     assert(k>=0 && k<N);
188.     codigo_t &cod = codigos[k];
189.     for (int j=0; j<cod.size(); j++) Q.push(cod[j]);
190.     // Convierte bits de 'Q' a caracteres
191.     while (Q.size()>NB) qflush(Q,Qbytes,nbits);
192.     // Escribe en el archivo zippeado.
193.     while (Qbytes.size()>bufflen) bflush(Qbytes,buff,nbits,zip);
194.   }
195.
196.   // Convierte el resto que puede quedar en Q
197.   while (Q.size()>0) qflush(Q,Qbytes,nbits);
198.   // Escribe el resto de lo que esta en Qbytes en 'zip'
199.   while (Qbytes.size()>0) bflush(Qbytes,buff,nbits,zip);
200.   // Terminador final con longitud de bloque=0
201.   int nb=0;
202.   // Escribe un terminador (un bloque de longitud 0)
203.   fwrite(&nb,sizeof(int),1,zip);
204.   fclose(zip);
205. }
206.
207. int pop_char(queue<char> &Q,btree<int> &H,
208.          btree<int>::iterator &m,int &k) {
209.   // 'm' es un nodo en el arbol. Normalmente deberia estar
210.   // en la raiz pero en principio puede estar en cualquier
211.   // nodo. Se supone que ya se convirtieron una seride de
```

```
212.   // bits. Si en la ultima llamada se llego a sacar un
213.   // caracter entonces 'm' termina en la raiz, listo para
214.   // extraer otro caracter. Entonces 'pop_char' extrae
215.   // tantos caracteres como para llegar a una hoja y, por lo
216.   // tanto, extraer un caracter. En ese caso pasa en 'k' el
217.   // indice correspondiente, vuelve a 'm' a la raiz (listo
218.   // para extraer otro caracter) y retorna 1. Si no, retorna
219.   // 0 y deja a 'm' en el nodo al que llega.
220.   while (!Q.empty()) {
221.     int f = Q.front();
222.     Q.pop();
223.     // El valor binario 0 o 1 almacenado en 'Q' dice que
224.     // hijo hay que tomar.
225.     if (f) m = m.right();
226.     else m = m.left();
227.     // Verificar si llego a una hoja.
228.     if (m.left()==H.end()) {
229.       // Pudo sacar un caracter completo
230.       k = *m;
231.       assert(k != -1);
232.       m = H.begin();
233.       return 1;
234.     }
235.   }
236.   // No pudo sacar un caracter completo.
237.   return 0;
238. }
239.
240. void hufunzip(char *zipped,char *unzipped) {
241.   // Deszippea el archivo de nombre 'zipped' en 'unzipped'
242.   // El vector de probabilidades (esta guardado en
243.   // 'zipped').
244.   vector<double> prob;
245.   // Los caracteres correspondientes a cada indice
246.   vector<char> letters;
247.   // Numero de bits por caracter
248.   const int NB=8;
249.
250.   // Abre el archivo 'zipped', si no es 'stdin'
251.   FILE *zip;
252.   if (zipped) {
253.     zip = fopen(zipped,"r");
254.     assert(zip);
255.   } else zip = stdin;
256.
257.   // Lee la tabla de probabilidades y codigos, estan
258.   // escritos en formato binario
259.   // probabilidad,caracter,probabilidad,caracter,. . . hasta
```

```
260.   // terminar con una probabilidad <0 Los va poniendo en
261.   // 'prob[]' y las letras en 'letters[]'
262.   int N=0;
263.   int nread;
264.   while (true) {
265.     double p;
266.     char c;
267.     nread = fread(&p,sizeof(double),1,zip);
268.     assert(nread==1);
269.     if (p<0.0) break;
270.     N++;
271.     prob.push_back(p);
272.     nread = fread(&c,sizeof(char),1,zip);
273.     assert(nread==1);
274.     letters.push_back(c);
275.   }
276.
277.   // 'H' va a tener el arbol de codigos. 'huffman()'
278.   // calcula el arbol.
279.   btree<int> H;
280.   huffman(prob,H);
281.
282.   // Los codigos se almacenan en un vector de
283.   // codigos.
284.   vector<codigo_t> codigos(N);
285.   // 'huffman_codes()' calcula los codigos y tambien
286.   // la longitud promedio del codigo.
287.   huffman_codes(H,prob,codigos);
288.
289.   // El archivo donde descompacta. Si no se pasa
290.   // el nombre entoces descompacta sobre 'stdout'.
291.   FILE *unz;
292.   if (unzipped) {
293.     unz = fopen(unzipped,"w");
294.     assert(unz);
295.   } else unz = stdout;
296.
297.   // Los bloques de bytes del archivo compactado
298.   // se van leyendo sobre una cola 'Q' y se va
299.   // descompactando directamente sobre el archivo
300.   // descompactado con 'putc' (el cual ya es
301.   // buffereado)
302.   queue<char> Q;
303.   int read=0;
304.   // Posicion en el arbol
305.   btree<int>::iterator m = H.begin();
306.   // indice de caracter que se extrajo
```

```
307.  int k;
308.  // Buffer para poner los bytes que se leen del
309.  // archivo compactado.
310.  vector<char_t> buff;
311.  char_t c;
312.  while (1) {
313.    int nb;
314.    // Lee longitud (en bits) del siguiente bloque.
315.    nread = fread(&nb,sizeof(int),1,zip);
316.    assert(nread==1);
317.
318.    // Detenerse si el siguiente bloque es nulo.
319.    if (!nb) break;
320.
321.    // Redimensionar la longitud del
322.    // buffer apropriadamente.
323.    int nbytes = nb/NB + (nb % NB ? 1 : 0);
324.    if (buff.size()<nbytes) buff.resize(nb);
325.
326.    // Lee el bloque
327.    nread = fread(&buff[0],sizeof(char_t),nbytes,zip);
328.    assert(nread==nbytes);
329.
330.    vector<char_t> v(NB);
331.    int j = 0, read=0;
332.    while (read<nb) {
333.      c = buff[j++];
334.      // Desempaqueta el caracter tn bits
335.      for (int l=0; l<NB; l++) {
336.        int b = (c & 1 ? 1 : 0);
337.        c >>= 1;
338.        v[NB-l-1] = b;
339.      }
340.      for (int l=0; l<NB; l++) {
341.        if (read++ < nb) Q.push(v[l]);
342.      }
343.      // Va convirtiendo bits de 'Q' en
344.      // caracteres. Si 'pop_char()' no puede
345.      // sacar un caracter, entonces va a devolver
346.      // 0 y se termina el lazo. En ese caso 'm'
347.      // queda en la posicion correspondiente en el
348.      // arbol.
349.      while(pop_char(Q,H,m,k)) putc(letters[k],unz);
350.    }
351.  }
352.
353.  assert(!.empty());
```

```
354.   // Cerrar los archivos abiertos.
355.   fclose(zip);
356.   fclose(unz);
357. }
```

Código 3.24: *Programa que comprime archivos basado en el algoritmo de Huffman. [Archivo: hufzipc.cpp]*

Ahora vamos a presentar un programa que comprime archivos utilizando el algoritmo de Huffman.

- Primero vamos a describir una serie de funciones auxiliares. `huffman_codes(T,prob,codigos)` calcula los codigos correspondientes a cada uno de los caracteres que aparecen en el archivo. La función es por supuesto recursiva y utiliza una función auxiliar `huffman_codes(T,node,prob,level,codigo,codigos)`, donde se va pasando el *"estado"* del cálculo por las variables `node` que es la posición actual en el árbol y `codigo` que es un vector de enteros con 0's y 1's que representa el código hasta este nodo. La función agrega un código a la tabla si llega a una hoja (línea 12). Si no, agrega un 0 al código y llama recursivamente a la función sobre el hijo izquierdo y después lo mismo pero sobre el hijo derecho, pero agregando un 1. El código es siempre el mismo ya que es pasado por referencia. Por eso después de agregar un 0 o 1 y llamar recursivamente a la función hay que eliminar el bit, de manera que `codigo` debe quedar al salir en la línea 29 igual que al entrar antes de la línea 22. La función *"wrapper"* llama a la auxiliar pasándole como nodo la raíz y un código de longitud nula.
- Las funciones `hufzip(file,zipped)` y `hufunzip(zipped,unzipped)` comprimen y descomprimen archivos, respectivamente. Los argumentos son cadenas de caracteres que corresponden a los nombres de los archivos. El primer argumento es el archivo de entrada y el segundo el de salida. Si un string es nulo entonces se asume `stdin` o `stdout`.
- `hufzip()` va leyendo caracteres del archivo de entrada y cuenta cuantos instancias de cada caracter hay en el archivo. Dividiendo por el número total de caracteres obtenemos las probabilidades de ocurrencia de cada caracter. Consideramos caracteres de 8 bits de longitud,

de manera que podemos comprimir cualquier tipo de archivos, formateados o no formateados.

- El árbol de Huffman se construye sólo para los **N** caracteres que existen actualmente en el archivo, es decir, aquellos que tienen probabilidad no nula. A cada caracter que existe en el archivo se le asigna un índice **indx** que va entre 0 y **N-1**. Se construyen dos tablas **indx[j]** que da el índice correspondiente al caracter **j** y **letters[k]** que da el caracter correspondiente al índice **k**. Otra posibilidad sería construir el árbol de Huffman para todos los caracteres incluyendo aquellos que tienen probabilidad nula.
- Como de todas formas estos caracteres tendrán posiciones en el árbol por debajo de aquellos caracteres que tienen probabilidad no nula, no afectará a la longitud promedio final.
- Notar que para poder comprimir hay que primero construir el árbol y para esto hay que recorrer el archivo para calcular las probabilidades. Esto obliga a leer los caracteres y mantenerlos en una cola **fin**, con lo cual todo el archivo a comprimir está en memoria principal. Otra posibilidad es hacer dos pasadas por el archivo, una primera pasada para calcular las probabilidades y una segunda para comprimir. Todavía otra posibilidad es comprimir con una tabla de probabilidades construida previamente, en base a estadística sobre archivos de texto o del tipo que se está comprimiendo. Sin embargo, en este caso las tasas de compresión van a ser menores.
- **hufzip()** va convirtiendo los caracteres que va leyendo a códigos de 0's y 1's que son almacenados temporariamente en una cola de bits **Q**. Mientras tanto se van tomando de a **NB=8** bits y se construye con operaciones de bits el caracter correspondiente el cual se va guardando en una cola de bytes **Qbytes**. A su vez, de **Qbytes** se van extrayendo bloques de caracteres de longitud **bufflen** para aumentar la eficiencia de la escritura a disco.
- Para descomprimir el archivo es necesario contar con el árbol que produjo la compresión. La solución utilizada aquí es guardar el vector de probabilidades utilizado **prob** y los caracteres correspondientes, ya que el árbol puede ser calculado unívocamente usando la función **huffman()** a partir de las probabilidades. Para guardar el árbol se van escribiendo en el archivo la probabilidad y el caracter de a uno (líneas 159–162). Para indicar el fin de la tabla se escribe una probabilidad negativa (-1.0). La lectura de la tabla se hace en **hufunzip()** se hace en la líneas 264–275. Ambas funciones (**hufzip** y **hufunzip**)

calculan el árbol usando **huffman()** en las líneas 141 y 280, respectivamente.

- El lazo que comprime son las líneas 180–194. Simplemente va tomando caracteres de **fin** y los convierte a bits, usando el código correspondiente, en **Q**. Simultáneamente va pasando tantas **NB**-tuplas de bits de **Q** a **Qbytes** con la función **qflush()** y tantos bytes de **Qbytes** al archivo zippeado con **bflush()**. La rutina **bflush()** va imprimiendo en el archivo de a **bufflen** bytes.
- En el archivo comprimido se van almacenando los bytes de a bloques de longitud **bufflen** o menor. Los bloques se almacenan grabando primero la longitud del bloque (un entero) (línea 73) y después el bloque de bytes correspondiente (línea 81).
- Después del lazo pueden quedar bits en **Q** y bytes en **Qbytes**. Los lazos de las líneas 197 y 199 terminan de procesar estos restos.
- El archivo se descomprime en las línea 312 de **hufunzip()**. Se leen bloques de bytes en la línea 327 y se van convirtiendo a bits en línea 341.
- La función **pop_char(Q,H,m,k)** va sacando bits de **Q** y moviendo el nodo **m** en el árbol hasta que **m** llega a una hoja (en cuyo caso **pop_char()** retorna un caracter por el argumento **k**, o mejor dicho el índice correspondiente) o hasta que **Q** queda vacía. En este caso **m** es vuelto a la raíz del árbol. Por ejemplo, refiriéndonos al código $C2$ de la figura 3.35 si inicialmente **m** está en el nodo **1** y **Q={0110010110}**, entonces el primer bit 0 de la cola mueve **m** al nodo **10** y el segundo a **101**. A esa altura **pop_char()** devuelve el caracter c ya que ha llegado a una hoja y **m** vuelve a la raíz. La cola de bits queda en **Q={10010110}**. Si volvemos a llamar a **pop_char()** repetidamente va a ir devolviendo los caracteres b (100) y c (101). A esa altura queda **Q={10}**. Si volvemos a llamar a **pop_char()**, **m** va a descender hasta el nodo 10 y **Q** va a quedar vacía. En ese caso **pop_char()** no retorna un caracter. La forma de retornar un caracter es la siguiente: Si **pop_char()** pudo llegar a un caracter, entonces retorna 1 (éxito) y el índice del caracter se pasa a través del argumento **k**, si no retorna 0 (fallo).
- El lazo de la línea 349 extrae tantos caracteres como puede de **Q** y los escribe en el archivo **unzipped** con el macro **putc()** (de la librería estándar de C).
- Al salir del lazo de las líneas 312–351 no pueden quedar bits sin convertir en **Q**, ya que todos los caracteres del archivo comprimido han sido leídos y estos representan un cierto número de caracteres. Si

después de salir del lazo quedan bits, esto quiere decir que hubo algún error en la lectura o escritura. El `assert()` de la línea 353 verifica esto.

- El programa así presentado difícilmente no puede recuperarse de un error en la lectura o escritura, salvo la detección de un error al fin del archivo, lo cual fue descripto en el párrafo anterior. Supongamos por ejemplo que encodamos el mensaje $accdcdcdcdcd$ con el código $C2$. Esto resulta en el string de bits `010110111101111011110111110111`. Ahora supongamos que al querer desencodar el mensaje se produce un error y el tercer bit pasa de 0 a 1, quedando el string `011110111101111011110111110111`. El proceso de decompresión resulta en el mensaje $addaddaddaddaddad$, quedando un remanente de un bit 1 en la cola. Detectamos que hay un error al ver que quedo un bit en la cola sin poder llegar a formar el caracter, pero lo peor es que todo el mensaje desencodado a partir del error es erróneo. Una posibilidad es encodar el código por bloques, introduciendo caracteres de control. La tasa de compresión puede ser un poco menor, pero si un error se produce en alguna parte del archivo los bloques restantes se podrán descomprimir normalmente.

El programa de compresión aquí puede comprimir y descomprimir archivos de texto y no formateados. Para un archivo de texto típico la tasa de compresión es del orden del 35 %. Esto es poco comparado con los compresores usuales como `gzip`, `zip` o `bzip2` que presentan tasas de compresión en el orden del 85 %. Además, esos algoritmos comprimen los archivos *"al vuelo"* (*"on the fly"*) es decir que no es necesario tener todo el archivo a comprimir en memoria o realizar dos pasadas sobre el mismo.

Capítulo 4

Conjuntos

En este capítulo se introduce en mayor detalle el TAD *"conjunto"*, ya introducido en la sección §1.2 y los subtipos relacionados *"diccionario"* y *"cola de prioridad"*.

4.1. Introducción a los conjuntos

Un conjunto es una colección de *"miembros"* o *"elementos"* de un *"conjunto universal"*. Por contraposición con las listas y otros contenedores vistos previamente, todos los miembros de un conjunto deben ser diferentes, es decir no puede haber dos copias del mismo elemento. Si bien para definir el concepto de conjunto sólo es necesario el concepto de igualdad o desigualdad entre los elementos del conjunto universal, en general las representaciones de conjuntos asumen que entre ellos existe además una *"relación de orden estricta"*, que usualmente se denota como $<$. A veces un tal orden no existe en forma natural y es necesario saber definirlo, aunque sea sólo para implementar el tipo conjunto (ver sección §2.4.4).

4.1.1. Notación de conjuntos

Normalmente escribimos un conjunto enumerando sus elementos entre llaves, por ejemplo $\{1, 4\}$. Debemos recordar que no es lo mismo que una lista, ya que, a pesar de que los enumeramos en forma lineal, *no existe un orden preestablecido entre los miembros de un conjunto*. A veces representamos conjuntos a través de una condición sobre los miembros del conjunto

universal, por ejemplo

$$A = \{x \text{ entero } / x \text{ es par } \} \tag{4.1}$$

De esta forma se pueden definir conjuntos con un número infinito de miembros.

La principal relación en los conjuntos es la de *"pertenencia"* $\in$, esto es $x \in A$ si x es un miembro de A. Existe un conjunto especial $\emptyset$ llamado el *"conjunto vacío"*. Decimos que A esta incluido en B ($A \subseteq B$, o $B \supseteq A$) si todo miembro de A también es miembro de B. También decimos que A es un *"subconjunto"* de B y que B es un *"supraconjunto"*de A. Todo conjunto está incluido en sí mismo y el conjunto vacío está incluido en cualquier conjunto. A y B son *"iguales"* si $A \subseteq B$ y $B \subseteq A$, por lo tanto dos conjuntos son distintos si al menos existe un elemento de A que no pertenece a B o viceversa. El conjunto A es un *"subconjunto propio"* (*"supraconjunto propio"*)de B si $A \subseteq B$ ($A \supseteq B$) y $A \neq B$.

Las operaciones más básicas de los conjuntos son la *"unión"*, *"intersección"* y *"diferencia"*. La unión $A \cup B$ es el conjunto de los elementos que pertenecen a A *o* a B mientras que la intersección $A \cap B$ es el de los elementos que pertenecen a A *y* a B. Dos conjuntos son *"disjuntos"* si $A \cap B = \emptyset$. La diferencia $A - B$ está formada por los elementos de A que no están en B. Es fácil demostrar la siguiente igualdad de conjuntos

$$A \cup B = (A \cap B) \cup (A - B) \cup (B - A) \tag{4.2}$$

siendo los tres conjuntos del miembro derecho disjuntos.

4.1.2. Interfaz básica para conjuntos

```
1.  typedef int elem_t;
2.
3.  class iterator_t {
4.  private:
5.    /* . . . */;
6.  public:
7.    bool operator!=(iterator_t q);
8.    bool operator==(iterator_t q);
9.  };
10.
11. class set {
12. private:
```

```
13.    /* . . . */;
14.  public:
15.    set();
16.    set(const set &);
17.    ~set();
18.    elem_t retrieve(iterator_t p);
19.    pair<iterator_t,bool> insert(elem_t t);
20.    void erase(iterator_t p);
21.    int erase(elem_t x);
22.    void clear();
23.    iterator_t next(iterator_t p);
24.    iterator_t find(elem_t x);
25.    iterator_t begin();
26.    iterator_t end();
27.  };
28.  void set_union(set &A,set &B,set &C);
29.  void set_intersection(set &A,set &B,set &C);
30.  void set_difference(set &A,set &B,set &C);
```

Código 4.1: *Interfaz básica para conjuntos [Archivo: setbash.h]*

En el código 4.1 vemos una interfaz básica para conjuntos

- Como en los otros contenedores STL vistos, una clase **iterator** permite recorrer el contenedor. Los iterators soportan los operadores de comparación == y !=.
- Sin embargo, en el conjunto no se puede insertar un elemento en una posición determinada, por lo tanto la función **insert** no tiene un argumento posición como en listas o árboles. Sin embargo **insert** retorna un par, conteniendo un iterator al elemento insertado y un **bool** que indica si el elemento es un nuevo elemento o ya estaba en el conjunto.
- La función **erase(p)** elimina el elemento que está en la posición **p**. La posición **p** debe ser válida, es decir debe haber sido obtenida de un **insert(x)** o **find(x)**. **erase(p)** invalida **p** y todas las otras posiciones obtenidas previamente.
- **count=erase(x)** elimina el elemento **x** si estaba en el conjunto. Si no, el conjunto queda inalterado. Retorna el número de elementos *efectivamente eliminados* del conjunto. Es decir, si el elemento estaba previamente en el conjunto entonces retorna 1, de otra forma retorna 0.

(Nota: Existe otro contenedor relacionado llamado *"multiset"* en el cual pueden existir varias copias de un mismo elemento. En *multiset* el valor de retorno puede ser mayor que 1).

- Como es usual **`begin()`**, **`end()`** y **`next()`** permiten iterar sobre el conjunto.

- Las operaciones binarias sobre conjuntos se realizan con las funciones **`set_union(A,B,C)`**, **`set_intersection(A,B,C)`** y **`set_difference(A,B,C)`** que corresponden a las operaciones $C = A \cup B$, $C = A \cap B$ y $C = A - B$, respectivamente. Notar que estas *no son miembros de la clase*. Todas estas funciones binarias asumen que los conjuntos **`A`**, **`B`** y **`C`** son distintos.

- Una restricción muy importante en todas las funciones binarias es que ninguno de los conjuntos de entrada (ni A ni B) deben *superponerse* (overlap) con C. Esto se aplica también a la versión de las STL, en cuyo caso los *rangos* de entrada no se deben superponer con el de salida.

- **`p=find(x)`** devuelve un iterator a la posición ocupada por el elemento **`x`** en el conjunto. Si el conjunto no contiene a **`x`** entonces devuelve **`end()`**.

4.1.3. Análisis de flujo de datos

Consideremos un programa simple como el mostrado en la figura 4.1 que calcula el máximo común divisor $\gcd(p, q)$ de dos números enteros p, q, mediante el algoritmo de Euclides. Recordemos que el algoritmo de Euclides se basa en la relación recursiva

$$\gcd(p, q) = \begin{cases} q \text{ divide a } p: & q; \\ \text{si no:} & \gcd(q, \text{rem}(p, q)) \end{cases} \tag{4.3}$$

donde asumimos que $p > q$ y $\text{rem}(p, q)$ es el resto de dividir p por q. Por ejemplo, si $p = 30$ y $q = 12$ entonces

$$\gcd(30, 12) = \gcd(12, 6) = 6 \tag{4.4}$$

ya que $\text{rem}(30, 12) = 6$ y 6 divide a 12.

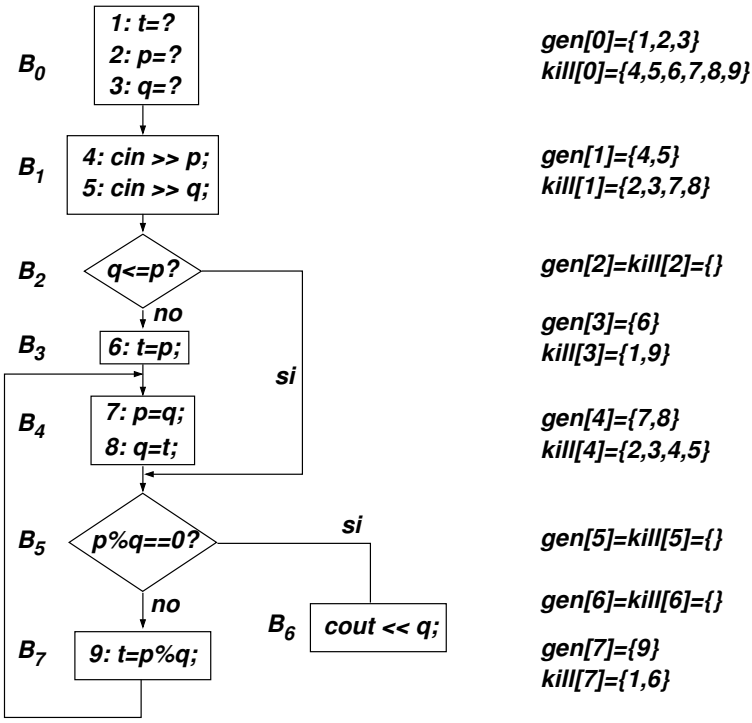


Figura 4.1: Algoritmo de Euclides

Los bloques B_4, B_5 y B_7 son la base recursiva del algoritmo. La lectura de los datos se produce en el bloque B_1 y el condicional del bloque B_2 se encarga de intercambiar p y q en el caso de que $q > p$.

Los bloques representan porciones de código en los cuales el código sucede secuencialmente línea a línea. Los condicionales en los bloques B_2 y B_5 y el lazo que vuelve del bloque B_7 al B_4 rompen la secuencialidad de este código.

En el diseño de compiladores es de importancia saber cuál es la última línea donde una variable puede haber tomado un valor al llegar a otra determinada línea. Por ejemplo, al llegar a la línea 7, `t` puede haber tomado su valor de una asignación en la línea 6 o en la línea 9. Este tipo de análisis es de utilidad para la optimización del código. Por ejemplo si al llegar a un cierto bloque sabemos que una variable `x` sólo puede haber tomado su valor de una asignación constante como `x=20;`, entonces en esa línea se puede reemplazar el valor de `x` por el valor 20. También puede servir para la detec-

ción de errores. Hemos introducido un bloque ficticio B_0 que asigna valores indefinidos (representados por el símbolo "?"). Si alguna de estas asignaciones están activas al llegar a una línea donde la variable es usada, entonces puede ser que se este usando una variable indefinida. Este tipo de análisis es estándar en la mayoría de los compiladores.

Para cada bloque B_j vamos a tener definidos 4 conjuntos a saber

- **`gen[j]`**: las asignaciones que son generadas en el bloque B_j. Por ejemplo en el bloque B_1 se generan las asignaciones 4 y 5 para las variables **`p`** y **`q`**.

- **`kill[j]`**: las asignaciones que son eliminadas en el bloque. Por ejemplo, al asignar valores a las variables **`p`** y **`q`** en el bloque B_1 cualquier asignación a esas variables que llegue al bloque será eliminada, como por ejemplo, las asignaciones 2 y 3 del bloque ficticio B_0. En este caso podemos detectar fácilmente cuáles son las asignaciones eliminadas pero en general esto puede ser más complejo, de manera que, conservativamente, introducimos en **`kill[j]`** todas las asignaciones a variables cuyo valor es reasignado en B_j. En el caso del bloque B_1 las variables reasignadas son **`p`** y **`q`**, las cuales sólo tienen asignaciones en las líneas 2,3,7 y 8 (sin contar las propias asignaciones en el bloque). De manera que **`kill[1]={2,3,7,8}`**. Notar que, por construcción **`gen[j]`** y **`kill[j]`** son conjuntos disjuntos.

- **`defin[j]`** el conjunto total de definiciones que llegan al bloque B_j.

- **`defout[j]`** el conjunto total de definiciones que salen del bloque B_j ya sea porque son generadas en el bloque, o porque pasan a través del bloque sin sufrir reasignación.

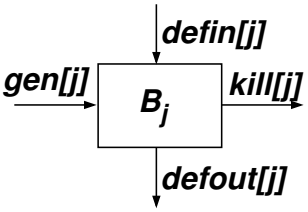


Figura 4.2: Ecuación de balance para las asignaciones que llegan y salen de un bloque.

En este análisis los conjuntos **gen[j]** y **kill[j]** pueden por simple observación del código, mientras que los **defin[j]** y **defout[j]** son el resultado buscado. Para obtenerlos debemos escribir una *"ecuación de balance de asignaciones"* en el bloque, a saber (ver figura 4.2)

$$\mathbf{defout[j]} = (\mathbf{defin[j]} \cup \mathbf{gen[j]}) - \mathbf{kill[j]} \tag{4.5}$$

la cual expresa que *las asignaciones que salen del bloque son aquellas que llegan, más las generadas en el bloque menos las que son eliminadas en el mismo*.

Ahora consideremos las asignaciones que llegan al B_4, es decir **defin[4]**. Estas pueden proceder o bien del bloque B_3 o bien del B_7, es decir

$$\mathbf{defin[4]} = \mathbf{defout[3]} \cup \mathbf{defout[7]} \tag{4.6}$$

En general tenemos que

$$\mathbf{defin[j]} = \sum_{\mathbf{m} \in \mathbf{ent[j]}} \mathbf{defout[m]} \tag{4.7}$$

donde **ent[j]** es el conjunto de bloques cuyas salidas confluyen a la entrada de B_j. En este caso tenemos

$$\begin{aligned}
\mathbf{ent[0]} &= \emptyset \\
\mathbf{ent[1]} &= \{0\} \\
\mathbf{ent[2]} &= \{1\} \\
\mathbf{ent[3]} &= \{2\} \\
\mathbf{ent[4]} &= \{3, 7\} \\
\mathbf{ent[5]} &= \{2, 4\} \\
\mathbf{ent[6]} &= \{5\} \\
\mathbf{ent[7]} &= \{5\}
\end{aligned} \tag{4.8}$$

```
1. void dataflow(vector<set> &gen,
2.               vector<set> &kill,
3.               vector<set> &defin,
4.               vector<set> &defout,
5.               vector<set> &ent) {
6.   int nblock = gen.size();
7.   bool cambio=true;
8.   while (cambio) {
9.     cambio=false;
```

```
10.     for (int j=0; j<nblock; j++) {
11.       // Calcular la entrada al bloque 'defin[j]'
12.       //   sumando sobre los 'defout[m]' que
13.       //   confluyen al bloque j . . .
14.     }
15.     int out_prev = defout[j].size();
16.
17.     cambio=false;
18.     for (int j=0; j<nblock; j++) {
19.       // Calcular el nuevo valor de 'defout[j]'
20.       //   usando la ec. de balance de asignaciones. . .
21.       if (defout[j].size() != out_prev) cambio=true;
22.     }
23.   }
24. }
```

Código 4.2: *Seudocódigo para el análisis de flujo de datos. [Archivo: dataflow1.cpp]*

Un seudocódigo para este algoritmo puede observarse en el código 4.2. La función **dataflow()** toma como argumentos vectores de conjuntos de longitud **nblock** (el número de bloques, en este caso 8). **ent[]**, **gen[]** y **kill[]** son datos de entrada, mientras que **defin[]** y **defout[]** son datos de salida calculados por **dataflow()**. **tmp** es una variable auxiliar de tipo conjunto. El código entra en un lazo infinito en el cual va calculando para cada bloque las asignaciones a la entrada **defin[j]** aplicando (4.7) y luego calculando **defout[j]** mediante (4.5).

El proceso es iterativo, de manera que hay que inicializar las variables **defin[]** y **defout[]** y detectar cuando no hay mas cambios. Notemos primero que lo único que importa son las inicializaciones para **defin[]** ya que cualquier valor que tome **defout[]** al ingresar a **dataflow[]** será sobreescrito durante la primera iteración. Tomemos como inicialización $\mathbf{defin[j]}^0 = \emptyset$. Después de la primera iteración los **defout[j]** tomarán ciertos valores $\mathbf{defout[j]}^0$, posiblemente no nulos, de manera que en la iteración 1 los $\mathbf{defin[j]}^1$ pueden eventualmente ser no nulos, pero vale que

$$\mathbf{defin[j]}^0 \subseteq \mathbf{defin[j]}^1 \tag{4.9}$$

Es fácil ver entonces que, después de aplicar (4.5) valdrá que

$$\mathbf{defout[j]}^0 \subseteq \mathbf{defout[j]}^1 \tag{4.10}$$

Siguiendo el razonamiento, puede verse que siempre seguirá valiendo que

$$\begin{aligned} \texttt{defin[j]}^k &\subseteq \texttt{defin[j]}^{k+1} \\ \texttt{defout[j]}^k &\subseteq \texttt{defout[j]}^{k+1} \end{aligned} \tag{4.11}$$

Nos preguntamos ahora cuantas veces hay que ejecutar el algoritmo. Notemos que, como tanto `defin[j]` como `defout[j]` deben ser subconjuntos del conjunto finito que representan todas las asignaciones en el programa, a partir de una cierta iteración los `defin[j]` y `defout[j]` no deben cambiar más. Decimos entonces que el algoritmo *"convirgió"* y podemos detenerlo.

Notar que (4.11) garantiza que, para detectar la convergencia basta con verificar que el tamaño de ningún `defout[j]` cambie.

j	iter=0	iter=1	iter=2
defin[0]	{}	{}	{}
defout[0]	{1,2,3}	{1,2,3}	{1,2,3}
defin[1]	{}	{1,2,3}	{1,2,3}
defout[1]	{4,5}	{1,4,5}	{1,4,5}
defin[2]	{}	{4,5}	{1,4,5}
defout[2]	{}	{4,5}	{1,4,5}
defin[3]	{}	{}	{4,5}
defout[3]	{6}	{6}	{4,5,6}
defin[4]	{}	{6,9}	{6,9}
defout[4]	{7,8}	{6,7,8,9}	{6,7,8,9}
defin[5]	{}	{7,8}	{4,5,6,7,8,9}
defout[5]	{}	{7,8}	{4,5,6,7,8,9}
defin[6]	{}	{}	{7,8}
defout[6]	{}	{}	{7,8}
defin[7]	{}	{}	{7,8}
defout[7]	{9}	{9}	{7,8,9}

Tabla 4.1: Tres primeras iteraciones para el algoritmo de análisis de flujo de datos.

Las Tablas 4.1 y 4.2 muestran el avance de las iteraciones hasta llegar a convergencia. La iteración 6 coincide con la 5, por lo tanto se comprueba que el algoritmo ha convergido.

```
1. void dataflow(vector<set> &gen,
2.               vector<set> &kill,
3.               vector<set> &defin,
4.               vector<set> &defout,
5.               vector<set> &ent) {
6.   int nblock = gen.size();
```

j	iter=3	iter=4	iter=5 y 6
defin[0]	{}	{}	{}
defout[0]	{1,2,3}	{1,2,3}	{1,2,3}
defin[1]	{1,2,3}	{1,2,3}	{1,2,3}
defout[1]	{1,4,5}	{1,4,5}	{1,4,5}
defin[2]	{1,4,5}	{1,4,5}	{1,4,5}
defout[2]	{1,4,5}	{1,4,5}	{1,4,5}
defin[3]	{1,4,5}	{1,4,5}	{1,4,5}
defout[3]	{4,5,6}	{4,5,6}	{4,5,6}
defin[4]	{4,5,6,7,8,9}	{4,5,6,7,8,9}	{4,5,6,7,8,9}
defout[4]	{6,7,8,9}	{6,7,8,9}	{6,7,8,9}
defin[5]	{1,4,5,6,7,8,9}	{1,4,5,6,7,8,9}	{1,4,5,6,7,8,9}
defout[5]	{1,4,5,6,7,8,9}	{1,4,5,6,7,8,9}	{1,4,5,6,7,8,9}
defin[6]	{4,5,6,7,8,9}	{1,4,5,6,7,8,9}	{1,4,5,6,7,8,9}
defout[6]	{4,5,6,7,8,9}	{1,4,5,6,7,8,9}	{1,4,5,6,7,8,9}
defin[7]	{4,5,6,7,8,9}	{1,4,5,6,7,8,9}	{1,4,5,6,7,8,9}
defout[7]	{4,5,7,8,9}	{4,5,7,8,9}	{4,5,7,8,9}

Tabla 4.2: Iteraciones 3-6 hasta llegar a convergencia para el algoritmo de análisis de flujo de datos.

```
7.   set tmp;
8.   bool cambio=true;
9.   while (cambio) {
10.   for (int j=0; j<nblock; j++) {
11.    defin[j].clear();
12.    iterator_t p = ent[j].begin();
13.    while (p!=ent[j].end()) {
14.     int k = ent[j].retrieve(p);
15.     set_union(defin[j],defout[k],tmp);
16.     defin[j] = tmp;
17.     p = ent[j].next(p);
18.    }
19.   }
20.   cambio=false;
21.   for (int j=0; j<nblock; j++) {
22.    int out_prev = defout[j].size();
23.    set_union(defin[j],gen[j],tmp);
24.    set_difference(tmp,kill[j],defout[j]);
25.    if (defout[j].size()!=out_prev) cambio=true;
26.   }
27.  }
28. }
```

Código 4.3: *Rutina para el análisis de flujo de datos. [Archivo: dtflow.cpp]*

El código 4.3 muestra el código definitivo usando la interfaz básica para conjuntos (ver código 4.1). En las líneas 11–18 se calcula el `defin[j]` a partir de los `defout[k]` que confluyen a la entrada. El iterator `j` recorre los elementos del conjunto `ent[j]` que son los bloques cuya salida llega a la entrada del bloque `j`. Nótese que se debe usar un conjunto auxiliar `tmp` ya que el debido a que los argumentos de entrada a `set_union` deben ser diferentes entre sí. En las líneas 21–26 se calculan los conjuntos `defout[j]` a partir de las entradas `defin[j]` y los `gen[j]` y `kill[j]` usando la relación (4.5). Notar también el uso del conjunto auxiliar `tmp` para evitar la superposición de argumentos. Antes de actualizar `defout[j]` guardamos el número de elementos en una variable `out_prev` para poder después verificar si el conjunto creció, y así determinar si es necesario seguir iterando o no.

4.2. Implementación por vectores de bits

Tal vez la forma más simple de representar un conjunto es guardando un campo de tipo `bool` por cada elemento del conjunto universal. Si este campo es verdadero entonces el elemento está en el conjunto y viceversa. Por ejemplo, si el conjunto universal son los enteros de `0` a `N-1`

$$U = \{j \text{ entero, tal que } 0 \leq j < N\} \tag{4.12}$$

entonces podemos representar a los conjuntos por vectores de valores booleanos (puede ser `vector<bool>`) de longitud `N`. Si `v` es el vector, entonces `v[j]` indica si el entero `j` está o no en el conjunto. Por ejemplo, si el conjunto es `S={4,6,9}` y `N` es 10, entonces el vector de bits correspondiente sería `v={0,0,0,0, 1,0,1,0,0,1}`.

Para insertar o borrar elementos se prende o apaga el bit correspondiente. Todas estas son operaciones $O(1)$. Las operaciones binarias también son muy simples de implementar. Por ejemplo, si queremos hacer la unión `C` $=$ `A` $\cup$ `B`, entonces debemos hacer `C.v[j] = A.v[j] || B.v[j]`. La intersección se obtiene reemplazando `||` con `&&` y la diferencia `C` $=$ `A` $-$ `B` con `C.v[j] = A.v[j] && ! B.v[j]`.

Notar que el tiempo de ejecución de estas operaciones es $O(N)$, donde N *es el número de elementos en el conjunto universal.* La memoria requerida es `N` bits, es decir que también es $O(N)$. Es de destacar que también

se podría usar `vector<T>` con cualquier tipo `T` convertible a un entero, por ejemplo `int`, `char`. `bool` y sus variantes. En cada caso la memoria requerida es `N*sizeof(T)`, de manera que siempre es $O(N)$. Esto representa 8 bits por elemento para `char` o 32 para `int`. En el caso de `bool`, el operador `sizeof(bool)` reporta normalmente 1 byte por elemento, pero la representación interna de `vector<bool>` y en realidad requiere de un sólo bit por elemento.

4.2.1. Conjuntos universales que no son rangos contiguos de enteros

Para representar conjuntos universales U que no son subconjuntos de los enteros, o que no son un subconjunto contiguo de los enteros `[0,N)` debemos definir funciones que establezcan la correspondencia entre los elementos del conjunto universal y el conjunto de los enteros entre $[0, N)$. Llamaremos a estas funciones

```
1.  int indx(elem_t t);
2.  elem_t element(int j);
```

```
1. const int N=50;
2. typedef int elem_t;
3. int indx(elem_t t) { return (t-100)/2; }
4. elem_t element(int j) { return 100+2*j; }
```

Código 4.4: *Funciones auxiliares para definir conjuntos dentro de* $U = \{100, 102, .., 198\}$. *[Archivo: element.cpp]*

```
1.  const int N=52;
2.  typedef char elem_t;
3.  int indx(elem_t c) {
4.    if (c>='a' && c<='z') return c-'a';
5.    else if (c>='A' && c<='Z') return 26+c-'A';
6.    else cout << "Elemento fuera de rango!!\n"; abort();
7.  }
8.  elem_t element(int j) {
9.    assert(j<N);
10.   return (j<26 ? 'a'+j : 'A'+j-26);
11. }
```

Código 4.5: *Funciones auxiliares para definir conjuntos dentro de las letras* `a-z` *y* `A-Z`. *[Archivo: setbasadefs.h]*

Por ejemplo, si queremos representar el conjunto de los enteros pares entre 100 y 198, entonces podemos definir estas funciones como en el código 4.4. Para el conjunto de las letras tanto minúsculas como mayúsculas (en total `N=52`) podemos usar las funciones mostradas en el código código 4.5. Hacemos uso de que el código ASCII para las letras es correlativo. Para las minúsculas va desde 97 a 122 (`a-z`) y para las mayúsculas va desde 65 hasta 90 (`A-Z`). La función `indx()` correspondiente determina en que rango (mayúsculas o minúsculas) está el caracter y restándole la base correspondiente lo convierte a un número entre 0 y 51. La función `element()`, en forma recíproca convierte un entero entre 0 y 51 a un caracter, fijándose primero si está en el rango 0-25, o 26-51.

4.2.2. Descripción del código

```
1.  typedef int iterator_t;
2.
3.  class set {
4.  private:
5.    vector<bool> v;
6.    iterator_t next_aux(iterator_t p) {
7.      while (p<N && !v[p]) p++;
8.      return p;
9.    }
10.   typedef pair<iterator_t,bool> pair_t;
11. public:
12.   set() : v(N,0) { }
13.   set(const set &A) : v(A.v) {}
14.   ~set() {}
15.   iterator_t lower_bound(elem_t x) {
16.     return next_aux(indx(x));
17.   }
18.   pair_t insert(elem_t x) {
19.     iterator_t k = indx(x);
20.     bool inserted = !v[k];
21.     v[k] = true;
22.     return pair_t(k,inserted);
23.   }
24.   elem_t retrieve(iterator_t p) { return element(p); }
```

```
25.     void erase(iterator_t p) { v[p]=false; }
26.     int erase(elem_t x) {
27.       iterator_t p = indx(x);
28.       int r = (v[p] ? 1 : 0);
29.       v[p] = false;
30.       return r;
31.     }
32.     void clear() { for(int j=0; j<N; j++) v[j]=false; }
33.     iterator_t find(elem_t x) {
34.       int k = indx(x);
35.       return (v[k] ? k : N);
36.     }
37.     iterator_t begin() { return next_aux(0); }
38.     iterator_t end() { return N; }
39.     iterator_t next(iterator_t p) { next_aux(++p); }
40.     int size() {
41.       int count=0;
42.       for (int j=0; j<N; j++) if (v[j]) count++;
43.       return count;
44.     }
45.     friend void set_union(set &A,set &B,set &C);
46.     friend void set_intersection(set &A,set &B,set &C);
47.     friend void set_difference(set &A,set &B,set &C);
48.   };
49.
50.   void set_union(set &A,set &B,set &C) {
51.     for (int j=0; j<N; j++) C.v[j] = A.v[j] || B.v[j];
52.   }
53.   void set_intersection(set &A,set &B,set &C) {
54.     for (int j=0; j<N; j++) C.v[j] = A.v[j] && B.v[j];
55.   }
56.   void set_difference(set &A,set &B,set &C) {
57.     for (int j=0; j<N; j++) C.v[j] = A.v[j] && ! B.v[j];
58.   }
```

Código 4.6: *Implementación de conjuntos con vectores de bits. [Archivo: setbasac.h]*

En el código 4.6 vemos una implementación posible con vectores de bits.

- El vector de bits está almacenado en un campo `vector<bool>`. El constructor por defecto dimensiona a `v` de tamaño `N` y con valores 0. El constructor por copia simplemente copia el campo `v`, con lo cual copia el conjunto.

- La clase iterator es simplemente un **typedef** de los enteros, con la particularidad de que el iterator corresponde al índice en el conjunto universal, es decir el valor que retorna la función **indx()**. Por ejemplo, en el caso de los enteros pares entre 100 y 198 tenemos 50 valores posibles del tipo iterator. El iterator correspondiente a 100 es 0, a 102 le corresponde 1, y así siguiendo hasta 198 que le corresponde 50.
- Se elige como iterator **end()** el índice **N**, ya que este no es nunca usado por un elemento del conjunto.
- Por otra parte, los iterators sólo deben avanzar sobre los valores definidos en el conjunto. Por ejemplo, si el conjunto es **S={120,128,180}** los iterators ocupados son 10, 14 y 40. Entonces **p=S.begin();** debe retornar 10 y aplicando sucesivamente **p=S.next(p);** debemos tener **p**=14, 40 y 50 (que es **N=S.end()**).
- La función auxiliar **p=next_aux(p)** (notar que esta declarada como privada) devuelve el primer índice siguiente a **p** ocupado por un elemento. Por ejemplo en el caso del ejemplo, **next_aux(0)** debe retornar 10, ya que el primer índice ocupado en el conjunto siguiente a 0 es el 10.
- La función **next()** (que en la interfaz avanzada será sobrecargada sobre el operador **operator++**) incrementa **p** y luego le aplica **next_aux()** para avanzar hasta el siguiente elemento del conjunto.
- La función **retrieve(p)** simplemente devuelve el elemento usando la función **element(p)**.
- **erase(x)** e **insert(x)** simplemente prenden o apagan la posición correspondiente **v[indx(x)]**. Notar que son $O(1)$.
- **find(x)** simplemente verifica si el elemento está en el conjunto, en ese caso retorna **indx(x)**, si no retorna **N** (que es **end()**).
- **size()** cuenta las posiciones en **v** que están prendidas. Por lo tanto es $O(N)$.
- Las funciones binarias **set_union(A,B,C)**, **set_intersection(A,B,C)** y **set_difference(A,B,C)** hacen un lazo sobre todas las posiciones del vector y por lo tanto son $O(N)$.

4.3. Implementación con listas

4.3.0.1. Similaridad entre los TAD conjunto y correspondencia

Notemos que el TAD CONJUNTO es muy cercano al TAD CORRESPONDENCIA, de hecho podríamos representar conjuntos con correspondencias, simplemente usando las claves de la correspondencia como elementos del

TAD conjunto	TAD Correspondencia
`x = retrieve(p);`	`x = retrieve(p).first;`
`p=insert(x)`	`p=insert(x,w)`
`erase(p)`	`erase(p)`
`erase(x)`	`erase(x)`
`clear()`	`clear()`
`p = find(x)`	`p = find(x)`
`begin()`	`begin()`
`end()`	`end()`

Tabla 4.3: Tabla de equivalencia entre las operaciones del TAD conjunto y el TAD correspondencia.

conjunto e ignorando el valor del contradominio. En la Tabla 4.3 vemos la equivalencia entre la mayoría de las operaciones de la clase. Sin embargo, las operaciones binarias **`set_union(A,B,C)`**, **`set_intersection(A,B,C)`** y **`set_difference(A,B,C)`** no tiene equivalencia dentro de la correspondencia. De manera que si tenemos una implementación del TAD correspondencia, y podemos implementar las funciones binarias, entonces con pequeñas modificaciones adicionales podemos obtener una implementación de conjuntos.

En la sección §2.4, se discutieron las posibles implementaciones de correspondencias con contenedores lineales (listas y vectores) ordenados y no ordenados. Consideremos por ejemplo la posibilidad de extender la implementación de correspondencia por listas ordenadas y no ordenadas a conjuntos. Como ya se discutió, en ambos casos las operaciones de inserción supresión terminan siendo $O(n)$ ya que hay que recorrer el contenedor para encontrar el elemento.

Sin embargo, hay una gran diferencia para el caso de las operaciones binarias. Consideremos por ejemplo **`set_union(A,B,C)`**. En el caso de contenedores no ordenados, la única posibilidad es comparar cada uno de los elementos x_a de A con cada uno de los elementos de B. Si se encuentra el elemento x_a en B, entonces el elemento es insertado en **`C`**. Tal algoritmo es $O(n_A\, n_B)$, donde $n_{A,B}$ es el número de elementos en A y B. Si el número de elementos en los dos contenedores es similar ($n_A \sim n_B \sim n$), entonces vemos que es $O(n^2)$.

4.3.0.2. Algoritmo lineal para las operaciones binarias

Con listas ordenadas se puede implementar una versión de **`set_union(A,B,C)`** que es $O(n)$. Mantenemos dos posiciones **pa** y **pb**, cuyos valores llamaremos x_a y x_b, tales que

- los elementos en A en posiciones anteriores a **pa** (y por lo tanto menores que x_a) son menores que x_b y viceversa,
- todos los valores en B antes de **pb** son menores que x_a.

Estas condiciones son bastante fuertes, por ejemplo si el valor $x_a < x_b$ entonces podemos asegurar que $x_a \notin B$. Efectivamente, en ese caso, todos los elementos anteriores a x_b son menores a x_a y por lo tanto distintos a x_a. Por otra parte los que están después de x_b son mayores que x_b y por lo tanto que x_a, con lo cual también son distintos. Como conclusión, no hay ningún elemento en B que sea igual a x_a, es decir $x_a \notin B$. Similarmente, si $x_b < x_a$ entonces se ve que $x_b \notin A$.

Inicialmente podemos poner **`pa=A.begin()`** y **`pb=B.begin()`**. Como antes de **pa** y **pb** no hay elementos, la condición se cumple. Ahora debemos avanzar **pa** y **pb** de tal forma que se siga cumpliendo la condición. En cada paso puede ser que avancemos **pa**, **pb** o ambos. Por ejemplo, consideremos el caso

$$A = \{1, 3, 5, 7, 10\}, \quad B = \{3, 5, 7, 9, 10, 12\} \tag{4.13}$$

Inicialmente tenemos $x_a = 1$ y $x_b = 3$. Si avanzamos **pb**, de manera que x_b pasa a ser 5, entonces la segunda condición no se satisfará, ya que el 3 en B no es menor que x_a que es 1. Por otra parte, si avanzamos **pa**, entonces sí se seguirán satisfaciendo ambas condiciones, y en general esto será siempre así, mientras *avancemos siempre el elemento menor*. Efectivamente, digamos que los elementos de A son $x_a^0, x_a^1, ..., x_a^{n-1}$, y los de B son $x_b^0, x_b^1, ..., x_b^{m-1}$ y que en un dado momento **pa** está en x_a^j y **pb** en x_b^k. Esto quiere decir que, por las condiciones anteriores,

$$\begin{aligned} x_a^0, x_a^1, ..., x_a^{j-1} < x_b^k \\ x_b^0, x_b^1, ..., x_b^{k-1} < x_a^j \end{aligned} \tag{4.14}$$

Si

$$x_a^j < x_b^k \tag{4.15}$$

entonces en el siguiente paso **pa** avanza a x_a^{j+1}. Los elementos de B antes de **pb** siguen satisfaciendo

$$x_b^0, x_b^1, ..., x_b^{k-1} < x_a^j < x_a^{j+1} \tag{4.16}$$

con lo cual la condición sobre **pa** se sigue cumpliendo. Por otra parte, ahora a los elementos antes de **pa** se agregó x_a^j con lo cual tenemos antes de **pa**

$$x_a^0, x_a^1, ..., x_a^{j-1}, x_a^j \tag{4.17}$$

que cumplen la condición requerida por (4.14) y (4.15). Puede verse las condiciones también se siguen satisfaciendo si $x_a^j > x_b^k$ y avanzamos **pb**, o si $x_a^j = x_b^k$ y avanzamos ambas posiciones.

Para cualquiera de las operaciones binarias inicializamos los iterators con **pa=A.begin()** y **pb=B.begin()** y los vamos avanzando con el mecanismo explicado, es decir siempre el menor o los dos cuando son iguales. El proceso se detiene cuando alguno de los iterators llega al final de su conjunto. En cada paso alguno de los iterators avanza, de manera que es claro que en un número finito de pasos alguno de los iterators llegará al fin, de hecho en menos de $n_a + n_b$ pasos. Las posiciones **pa** y **pb** recorren todos los elementos de alguno de los dos conjuntos, mientras que en el otro puede quedar un cierto *"resto"*, es decir una cierta cantidad de elementos al final de la lista.

Ahora consideremos la operación de **set_union(A,B,C)**. Debemos asegurarnos de insertar todos los elementos de A y B, pero una sola vez y en forma ordenada. Esto se logra si en cada paso insertamos en el fin de C el elemento menor de x_a y x_b (si son iguales se inserta una sola vez). Efectivamente, si en un momento $x_a < x_b$ entonces, por lo discutido previamente x_a seguramente no está en B y podemos insertarlo en C, ya que en el siguiente paso **pa** avanzará, dejándolo atrás, con lo cual seguramente no lo insertaremos nuevamente. Además en pasos previos x_a no puede haber sido insertado ya que si era el menor **pa** habría avanzado dejándolo atrás y si era el mayor no habría sido insertado. El caso en que son iguales puede analizarse en forma similar. Puede verse que los elementos de C quedan ordenados, ya que de x_a y x_b siempre avanzamos el menor. Una vez que uno de los iterators (digamos **pa**) llegó al final, si quedan elementos en B (el *"resto"*) entonces podemos insertarlos directamente al fin de C ya que está garantizado que estos elementos no pueden estar en A.

En el caso de los conjuntos en (4.13) un seguimiento de las operaciones arroja lo siguiente

- x_a=1, x_b=3, inserta 1 en C
- x_a=3, x_b=3, inserta 3 en C
- x_a=5, x_b=5, inserta 5 en C
- x_a=7, x_b=7, inserta 7 en C

- x_a=10, x_b=9, inserta 9 en C
- x_a=10, x_b=10, inserta 10 en C
- **pa** llega a **A.end()**
- Inserta todo el resto de B (el elemento 12) en C.

quedando $C = \{1, 3, 5, 7, 9, 10, 12\}$ que es el resultado correcto.

En el caso de **set_intersection(A,B,C)** sólo hay que insertar x_a cuando $x_a = x_b$ y los restos no se insertan. El seguimiento arroja

- x_a=1, x_b=3,
- x_a=3, x_b=3, inserta 3
- x_a=5, x_b=5, inserta 5
- x_a=7, x_b=7, inserta 7
- x_a=10, x_b=9,
- x_a=10, x_b=10, inserta 10
- **pa** llega a **A.end()**

Para **set_difference(A,B,C)** hay que insertar x_a si $x_a < x_b$ y x_b no se inserta nunca. Al final sólo se inserta el resto de A.

- x_a=1, x_b=3, inserta 1
- x_a=3, x_b=3,
- x_a=5, x_b=5,
- x_a=7, x_b=7,
- x_a=10, x_b=9,
- x_a=10, x_b=10,
- **pa** llega a **A.end()**

Con estos algoritmos, los tiempos de ejecución son $O(n_A + n_B)$ ($O(n)$ si los tamaños son similares).

4.3.0.3. Descripción de la implementación

```
1.  typedef int elem_t;
2.
3.  typedef list<elem_t>::iterator iterator_t;
4.
5.  class set {
6.  private:
7.    list<elem_t> L;
8.  public:
```

```
9.    set() {}
10.   set(const set &A) : L(A.L) {}
11.   ~set() {}
12.   elem_t retrieve(iterator_t p) { return *p; }
13.   iterator_t lower_bound(elem_t t) {
14.     list<elem_t>::iterator p = L.begin();
15.     while (p!=L.end() && t>*p) p++;
16.     return p;
17.   }
18.   iterator_t next(iterator_t p) { return ++p; }
19.   pair<iterator_t,bool> insert(elem_t x) {
20.     pair<iterator_t,bool> q;
21.     iterator_t p;
22.     p = lower_bound(x);
23.     q.second = p==end() || *p!=x;
24.     if(q.second) p = L.insert(p,x);
25.     q.first = p;
26.     return q;
27.   }
28.   void erase(iterator_t p) { L.erase(p); }
29.   void erase(elem_t x) {
30.     list<elem_t>::iterator
31.       p = lower_bound(x);
32.     if (p!=end() && *p==x) L.erase(p);
33.   }
34.   void clear() { L.clear(); }
35.   iterator_t find(elem_t x) {
36.     list<elem_t>::iterator
37.       p = lower_bound(x);
38.     if (p!=end() && *p==x) return p;
39.     else return L.end();
40.   }
41.   iterator_t begin() { return L.begin(); }
42.   iterator_t end() { return L.end(); }
43.   int size() { return L.size(); }
44.   friend void set_union(set &A,set &B,set &C);
45.   friend void set_intersection(set &A,set &B,set &C);
46.   friend void set_difference(set &A,set &B,set &C);
47. };
48.
49. void set_union(set &A,set &B,set &C) {
50.   C.clear();
51.   list<elem_t>::iterator pa = A.L.begin(),
52.     pb = B.L.begin(), pc = C.L.begin();
53.   while (pa!=A.L.end() && pb!=B.L.end()) {
```

```
54.       if (*pa<*pb) { pc = C.L.insert(pc,*pa); pa++; }
55.       else if (*pa>*pb) {pc = C.L.insert(pc,*pb); pb++; }
56.       else {pc = C.L.insert(pc,*pa); pa++; pb++; }
57.       pc++;
58.     }
59.     while (pa!=A.L.end()) {
60.       pc = C.L.insert(pc,*pa);
61.       pa++; pc++;
62.     }
63.     while (pb!=B.L.end()) {
64.       pc = C.L.insert(pc,*pb);
65.       pb++; pc++;
66.     }
67.   }
68.   void set_intersection(set &A,set &B,set &C) {
69.     C.clear();
70.     list<elem_t>::iterator pa = A.L.begin(),
71.       pb = B.L.begin(), pc = C.L.begin();
72.     while (pa!=A.L.end() && pb!=B.L.end()) {
73.       if (*pa<*pb) pa++;
74.       else if (*pa>*pb) pb++;
75.       else { pc=C.L.insert(pc,*pa); pa++; pb++; pc++; }
76.     }
77.   }
78.   // C = A - B
79.   void set_difference(set &A,set &B,set &C) {
80.     C.clear();
81.     list<elem_t>::iterator pa = A.L.begin(),
82.       pb = B.L.begin(), pc = C.L.begin();
83.     while (pa!=A.L.end() && pb!=B.L.end()) {
84.       if (*pa<*pb) { pc=C.L.insert(pc,*pa); pa++; pc++; }
85.       else if (*pa>*pb) pb++;
86.       else { pa++; pb++; }
87.     }
88.     while (pa!=A.L.end()) {
89.       pc = C.L.insert(pc,*pa);
90.       pa++; pc++;
91.     }
92.   }
```

Código 4.7: *Implementación de conjuntos con listas ordenadas. [Archivo: setbas.h]*

En el código 4.7 vemos una posible implementación de conjuntos con listas ordenadas.

- Los métodos de la clase son muy similares a los de correspondencia y no serán explicados nuevamente (ver sección §2.4). Consideremos por ejemplo `p=insert(x)`. La única diferencia con el `insert()` de `map` es que en aquel caso hay que insertar un par que consiste en la clave y el valor de contradominio, mientras que en `set` sólo hay que insertar el elemento.
- Las funciones binarias no pertenecen a la clase y el algoritmo utilizado responde a lo descripto en la sección §4.3.0.2. Notar que al final de `set_union()` se insertan *los dos restos* de A y B a la cola de C. Esto se debe a que al llegar a este punto uno de los dos iterators está en `end()` de manera que sólo uno de los lazos se ejecutará efectivamente.

4.3.0.4. Tiempos de ejecución

Método	$T(N)$
`retrieve(p)`, `insert(x)`, `erase(x)`, `clear()`, `find(x)`, `lower_bound(x)`, `set_union(A,B,C)`, `set_intersection(A,B,C)`, `set_difference(A,B,C)`,	$O(n)$
`erase(p)`, `begin()`, `end()`,	$O(1)$

Tabla 4.4: Tiempos de ejecución de los métodos del TAD conjunto implementado con listas ordenadas.

4.4. Interfaz avanzada para conjuntos

```
1.  template<class T>
2.  class set {
3.  private:
4.    /* . . . */
5.  public:
6.    class iterator {
7.      friend class set;
8.      T & operator*();
9.      T *operator->();
10.     bool operator!=(iterator q);
11.     bool operator==(iterator q);
```

```
12.     }
13.     set() {}
14.     set(const set &A) : L(A.L) {}
15.     ~set() {}
16.     set &operator=(set<T> &);
17.     iterator lower_bound(T t);
18.     pair<iterator,bool> insert(T x);
19.     void erase(iterator p);
20.     int erase(T x);
21.     void clear();
22.     iterator find(T x);
23.     iterator begin();
24.     iterator end();
25.     int size();
26.   };
27.
28.   template<class T>
29.   void set_union(set<T> &A,set<T> &B,set<T> &C);
30.
31.   template<class T>
32.   void set_intersection(set<T> &A,set<T> &B,set<T> &C);
33.
34.   template<class T>
35.   void set_difference(set<T> &A,set<T> &B,set<T> &C);
```

Código 4.8: *Interfaz avanzada para conjuntos [Archivo: seth.h]*

En el código 4.8 se puede ver la interfaz avanzada para conjuntos, es decir utilizando templates, clases anidadas y sobrecarga de operadores. Las diferencias con la interfaz básica son similares a las de otros TAD (por ejemplo `tree<>`).

- La clase `set` pasa a ser ahora un template, de manera que podremos declarar `set<int>`, `set<double>`.
- La clase `iterator` es ahora una clase anidada dentro de `set`. Externamente se verá como `set<int>::iterator`
- La dereferenciación de posiciones (`x=retrieve(p)`) se reemplaza por `x=*p`. sobrecargando el operador `*`. Si el tipo elemento (es decir el tipo `T` del template) contiene campos dato o métodos, podemos escribir `p->campo` o `p->f(...)`. Para esto sobrecargamos los operadores `operator*` y `operator->`.

- Igual que con la interfaz básica, para poder hacer comparaciones de iterators debemos sobrecargar también los operadores `==` y `!=` en la clase iterator.
- `erase(x)` retorna el número de elementos efectivamente eliminados.
- `insert(x)` retorna un `pair<iterator,bool>`. (ver sección §2.4.3.2). El primero es, como siempre, un iterator al elemento insertado. El segundo indica si el elemento realmente fue insertado o ya estaba en el conjunto.

```
1.  template<class T>
2.  class set {
3.  private:
4.    list<T> L;
5.  public:
6.    typedef typename list<T>::iterator iterator;
7.    typedef pair<iterator,bool> pair_t;
8.    set() {}
9.    set(const set &A) : L(A.L) {}
10.   ~set() {}
11.   iterator lower_bound(T t) {
12.     iterator p = L.begin();
13.     while (p!=L.end() && t>*p) p++;
14.     return p;
15.   }
16.   pair_t insert(T x) {
17.     iterator p = lower_bound(x);
18.     if(p==end() || *p!=x) {
19.       p = L.insert(p,x);
20.       return pair_t(p,true);
21.     } else {
22.       return pair_t(end(),false);
23.     }
24.   }
25.   void erase(iterator p) { L.erase(p); }
26.   int erase(T x) {
27.     iterator p = lower_bound(x);
28.     if (p!=end() && *p==x) {
29.       L.erase(p); return 1;
30.     } else return 0;
31.   }
32.   void clear() { L.clear(); }
33.   iterator find(T x) {
34.     iterator p = lower_bound(x);
35.     if (p!=end() && *p==x) return p;
```

```
36.       else return L.end();
37.     }
38.     iterator begin() { return L.begin(); }
39.     iterator end() { return L.end(); }
40.     int size() { return L.size(); }
41.     bool empty() { return !L.size(); }
42.     friend void set_union<>(set<T> &A,set<T> &B,set<T> &C);
43.     friend void set_intersection<>(set<T> &A,set<T> &B,set<T> &C);
44.     friend void set_difference<>(set<T> &A,set<T> &B,set<T> &C);
45.   };
46.
47.   template<class T>
48.   void set_union(set<T> &A,set<T> &B,set<T> &C) {
49.     C.clear();
50.     typename list<T>::iterator pa = A.L.begin(),
51.       pb = B.L.begin(), pc = C.L.begin();
52.     while (pa!=A.L.end() && pb!=B.L.end()) {
53.       if (*pa<*pb) {pc = C.L.insert(pc,*pa); pa++; }
54.       else if (*pa>*pb) {pc = C.L.insert(pc,*pb); pb++; }
55.       else {pc = C.L.insert(pc,*pa); pa++; pb++; }
56.       pc++;
57.     }
58.     while (pa!=A.L.end()) {
59.       pc = C.L.insert(pc,*pa); pa++; pc++;
60.     }
61.     while (pb!=B.L.end()) {
62.       pc = C.L.insert(pc,*pb); pb++; pc++;
63.     }
64.   }
65.
66.   template<class T>
67.   void set_intersection(set<T> &A,set<T> &B,set<T> &C) {
68.     C.clear();
69.     typename list<T>::iterator pa = A.L.begin(),
70.       pb = B.L.begin(), pc = C.L.begin();
71.     while (pa!=A.L.end() && pb!=B.L.end()) {
72.       if (*pa<*pb) pa++;
73.       else if (*pa>*pb) pb++;
74.       else {C.L.insert(pc,*pa); pa++; pb++; }
75.     }
76.   }
77.
78.   // C = A - B
79.   template<class T>
80.   void set_difference(set<T> &A,set<T> &B,set<T> &C) {
81.     C.clear();
82.     typename list<T>::iterator pa = A.L.begin(),
```

```
83.       pb = B.L.begin(), pc = C.L.begin();
84.     while (pa!=A.L.end() && pb!=B.L.end()) {
85.       if (*pa<*pb) {C.L.insert(pc,*pa); pa++; }
86.       else if (*pa>*pb) pb++;
87.       else { pa++; pb++; }
88.     }
89.     while (pa!=A.L.end()) {
90.       pc = C.L.insert(pc,*pa); pa++; pc++;
91.     }
92.   }
```

Código 4.9: *Implementación de la interfaz avanzada para conjuntos con listas ordenadas. [Archivo: setl.h]*

Una implementación de la interfaz avanzada basada en listas ordenadas puede observarse en el código 4.9.

4.5. El diccionario

En algunas aplicaciones puede ser que se necesite un TAD como el conjunto pero sin necesidad de las funciones binarias. A un tal TAD lo llamamos TAD DICCIONARIO. Por supuesto cualquier implementación de conjuntos puede servir como diccionario, por ejemplo con vectores de bits, contenedores lineales ordenados o no. Sin embargo existe una implementación muy eficiente para la cual las inserciones y supresiones son $O(1)$, basada en la estructura *"tabla de dispersión"* (*"hash tables"*). Sin embargo, no es simple implementar en forma eficiente las operaciones binarias para esta implementación, por lo cual no es un buen candidato para conjuntos.

4.5.1. La estructura tabla de dispersión

La idea esencial es que dividimos el conjunto universal en B *"cubetas"* (*"buckets"* o *"bins"*), de tal manera que, a medida que nuevos elementos son insertados en el diccionario, estos son desviados a la cubeta correspondiente. Por ejemplo, si consideramos diccionarios de cadenas de caracteres en el rango `a-z`, es decir letras minúsculas, entonces podemos dividir al conjunto universal en B=26 cubetas. La primera corresponde a todos los strings que comienzan con `a`, la segunda los que comienzan con `b` y así siguiendo hasta la `z`. En general tendremos una función de dispersión `int b = h(elem_t t)`

que nos da el número de cubeta **b** en el cual debe ser almacenado el elemento **t**. En el ejemplo descripto, la función podría implementarse como sigue

```
1.  int h(string t) {
2.    return t[0]-'a';
3.  }
```

En este caso está garantizado que los números de cubetas devueltos por **h()** están en el rango $[0, B)$. En la práctica, el programador de la clase puede proveer funciones de hash para los tipos más usuales (como **int**, **double**, **string**...) dejando la posibilidad de que el usuario defina la función de hash para otros tipos, o también para los tipos básicos si considera que los que el provee son más eficientes (ya veremos cuáles son los requisitos para una buena función de hash). Asumiremos siempre que el tiempo de ejecución de la función de dispersión es $O(1)$. Para mayor seguridad, asignamos al elemento **t** la cubeta **b=h(t)%B**, de esta forma está siempre garantizado que **b** está en el rango $[0, B)$.

Básicamente, las cubetas son guardadas en un arreglo de cubetas (**vector<elem_t> v(B)**). Para insertar un elemento, simplemente calculamos la cubeta a usando la función de dispersión y guardamos el elemento en esa cubeta. Para hacer un **find(x)** o **erase(x)**, calculamos la cubeta y verificamos si el elemento está en la cubeta o no. De esta forma tanto las inserciones como las supresiones son $O(1)$. Este costo tan bajo es el interés principal de estas estructuras.

Pero normalmente el número de cubetas es mucho menor que el número de elementos del conjunto universal N (en muchos casos este último es infinito). En el ejemplo de los strings, todos los strings que empiezan con **a** van a la primera cubeta. Si un elemento es insertado y la cubeta correspondiente ya está ocupada decimos que hay una *"colisión"* y no podemos insertar el elemento en la tabla. Por ejemplo, si $B = 10$ y usamos **h(x)=x**, entonces si se insertan los elementos $\{1, 13, 4, 1, 24\}$ entonces los tres primeros elementos van a las cubetas 1, 3 y 4 respectivamente. El cuarto elemento también va a la cubeta 1, la cual ya está ocupada, pero no importa ya que es el *mismo* elemento que está en la cubeta. El problema es al insertar el elemento 24, ya que va a parar a la cubeta 4 la cual ya está ocupada, pero por *otro* elemento (el 4).

Si el número de elementos en el conjunto n es pequeño con respecto al número de cubetas ($n \ll B$) entonces la probabilidad de que dos elementos vayan a la misma cubeta es pequeña, pero en este caso la memoria requerida (que es el tamaño del vector **v**, es decir $O(B)$) sería mucho mayor que el tamaño del conjunto, con lo cual la utilidad práctica de esta implementa-

ción sería muy limitada. De manera que debe definirse una estrategia para *"resolver colisiones"*. Hay al menos dos formas bien conocidas:

- Usar *"tablas de dispersión abiertas"*.
- Usar redispersión en *"tablas de dispersión cerradas"*.

4.5.2. Tablas de dispersión abiertas

Esta es la forma más simple de resolver el problema de las colisiones. En esta implementación las cubetas no son elementos, sino que son listas (simplemente enlazadas) de elementos, es decir el vector `v` es de tipo `vector< list<elem_t> >`). De esta forma cada cubeta puede contener (teóricamente) infinitos elementos. Los elementos pueden insertarse en las cubetas en cualquier orden o ordenadas. La discusión de la eficiencia en este caso es similar a la de correspondencia con contenedores lineales (ver sección §2.4).

A continuación discutiremos la implementación del TAD diccionario implementado por tablas de dispersión abiertas con listas desordenadas. La inserción de un elemento `x` pasa por calcular el número de cubeta usando la función de dispersión y revisar la lista (cubeta) correspondiente. Si el elemento está en la lista, entonces no es necesario hacer nada. Si no está, podemos insertar el elemento en cualquier posición, puede ser en `end()`. El costo de la inserción es, en el peor caso, cuando elemento no esta en la lista, proporcional al número de elementos en la lista (cubeta). Si tenemos n elementos, el número de elementos por cubeta será, en promedio, n/B. Si el número de cubetas es $B \approx n$, entonces $n/B \approx 1$ y el tiempo d ejecución es $O(1 + n/B)$. El 1 tiene en cuenta acá de que al menos hay que calcular la función de dispersión. Como el término n/B puede ser menor, e incluso mucho menor, que 1, entonces hay que mantener el término 1, de lo contrario estaríamos diciendo que en ese caso el costo de la función es mucho menor que 1. En el peor caso, todos los elementos pueden terminar en una sola cubeta, en cuyo caso la inserción sería $O(n)$. Algo similar pasa con `erase()`.

4.5.2.1. Detalles de implementación

```
1.  typedef int key_t;
2.
3.  class hash_set;
```

```
4.  class iterator_t {
5.    friend class hash_set;
6.  private:
7.    int bucket;
8.    std::list<key_t>::iterator p;
9.    iterator_t(int b,std::list<key_t>::iterator q)
10.     : bucket(b), p(q) { }
11. public:
12.   bool operator==(iterator_t q) {
13.     return (bucket == q.bucket && p==q.p);
14.   }
15.   bool operator!=(iterator_t q) {
16.     return !(*this==q);
17.   }
18.   iterator_t() { }
19. };
20. typedef int (*hash_fun)(key_t x);
21.
22. class hash_set {
23. private:
24.   typedef std::list<key_t> list_t;
25.   typedef list_t::iterator listit_t;
26.   typedef std::pair<iterator_t,bool> pair_t;
27.   hash_set(const hash_set&) {}
28.   hash_set& operator=(const hash_set&) {}
29.   hash_fun h;
30.   int B;
31.   int count;
32.   std::vector<list_t> v;
33.   iterator_t next_aux(iterator_t p) {
34.     while (p.p==v[p.bucket].end()
35.            && p.bucket<B-1) {
36.       p.bucket++;
37.       p.p = v[p.bucket].begin();
38.     }
39.     return p;
40.   }
41. public:
42.   hash_set(int B_a,hash_fun h_a)
43.     : B(B_a), v(B), h(h_a), count(0) { }
44.   iterator_t begin() {
45.     iterator_t p = iterator_t(0,v[0].begin());
46.     return next_aux(p);
47.   }
48.   iterator_t end() {
```

```
49.       return iterator_t(B-1,v[B-1].end());
50.     }
51.     iterator_t next(iterator_t p) {
52.       p.p++; return next_aux(p);
53.     }
54.     key_t retrieve(iterator_t p) { return *p.p; }
55.     pair_t insert(const key_t& x) {
56.       int b = h(x) % B;
57.       list_t &L = v[b];
58.       listit_t p = L.begin();
59.       while (p!= L.end() && *p!=x) p++;
60.       if (p!= L.end())
61.         return pair_t(iterator_t(b,p),false);
62.       else {
63.         count++;
64.         p = L.insert(p,x);
65.         return pair_t(iterator_t(b,p),true);
66.       }
67.     }
68.     iterator_t find(key_t& x) {
69.       int b = h(x) % B;
70.       list_t &L = v[b];
71.       listit_t p = L.begin();
72.       while (p!= L.end() && *p!=x) p++;
73.       if (p!= L.end())
74.         return iterator_t(b,p);
75.       else return end();
76.     }
77.     int erase(const key_t& x) {
78.       list_t &L = v[h(x) % B];
79.       listit_t p = L.begin();
80.       while (p!= L.end() && *p!=x) p++;
81.       if (p!= L.end()) {
82.         L.erase(p);
83.         count--;
84.         return 1;
85.       } else return 0;
86.     }
87.     void erase(iterator_t p) {
88.       v[p.bucket].erase(p.p);
89.     }
90.     void clear() {
91.       count=0;
92.       for (int j=0; j<B; j++) v[j].clear();
93.     }
94.     int size() { return count; }
```

```
95.  };
```

Código 4.10: *Diccionario implementado por tablas de dispersión abiertas con listas desordenadas. [Archivo: hashsetbaso.h]*

En el código 4.10 vemos una posible implementación.

- Usamos `vector<>` y `list<>` de STL para implementar los vectores y listas, respectivamente.
- Los elementos a insertan en el diccionario son de tipo `key_t`.
- El `typedef` de la línea 20 define un tipo para las funciones admisibles como funciones de dispersión. Estas son funciones que deben tomar como argumento un elemento de tipo `key_t` y devuelve un entero.
- La clase contiene un puntero `h` a la función de dispersión. Este puntero se define en el constructor.
- El constructor toma como argumentos el número de cubetas y el puntero a la función de dispersión y los copia en los valores internos. Redimensiona el vector de cubetas `v` e inicializa el contador de elementos `count`.
- La clase iterator consiste de el número de cubeta (`bucket`) y un iterator en la lista (`p`). Las posiciones válidas son, como siempre, posiciones dereferenciables y `end()`.
- Los iterators dereferenciable consisten en un número de cubeta no vacía y una posición dereferenciable en la lista correspondiente.
- El iterator `end()` consiste en el par `bucket=B-1` y `p` el `end()` de esa lista (es decir, `v[bucket].end()`)
- Hay que tener cuidado de, al avanzar un iterator siempre llegar a otro iterator válido. La función privada `next_aux()` avanza cualquier combinación de cubeta y posición en la lista (puede no ser válida, por ejemplo el `end()` de una lista que no es la última) hasta la siguiente posición válida.
- El tiempo de ejecución de `next_aux()` es 1 si simplemente avanza una posición en la misma cubeta sin llegar a `end()`. Si esto último ocurre entonces entra en un lazo sobre las cubetas del cual sólo sale cuando llega a una cubeta no vacía. Si el número de elementos es mayor que B entonces en promedio todas las cubetas tienen al menos un elemento y `next_aux()` a lo sumo debe avanzar a la siguiente cubeta. Es decir, el cuerpo del lazo dentro de `next_aux()` se ejecuta una sola vez.

Si $n \ll B$ entonces el número de cubetas llenas es aproximadamente n y el de vacías $\approx B$ (en realidad esto no es exactamente así ya que hay una cierta probabilidad de que dos elementos vayan a la misma cubeta). Entonces, por cada cubeta llena hay B/n cubetas vacías y el lazo en **`next_aux()`** debe ejecutarse B/n veces. Finalmente, B/n da infinito para $n = 0$ y en realidad el número de veces que se ejecuta el lazo no es infinito sino B.

- **`begin()`** devuelve la primera posición válida en el diccionario. Para ello toma el iterator correspondiente a la posición **`begin()`** de la primera lista y le aplica **`next_aux()`**. Esto puede resultar en una posición dereferenciable o en **`end()`** (si el diccionario está vacío).
- **`insert(x)`** y **`erase(x)`** proceden de acuerdo a lo explicado en la sección previa.
- Notar la definición de referencias a listas en las líneas 57 y líneas 78. Al declarar **`L`** como referencias a listas (por el **`&`**) *no se crea una copia* de la lista.
- **`next(p)`** incrementa la posición en la lista. Pero con esto no basta ya que puede estar en el **`end()`** de una lista que no es la última cubeta. Por eso se aplica **`next_aux()`** que sí avanza a la siguiente posición válida.

4.5.2.2. Tiempos de ejecución

Método	$T(n, B)$ (promedio)	$T(n, B)$ (peor caso)
`retrieve(p)`, `erase(p)`, `end()`	$O(1)$	$O(1)$
`insert(x)`, `find(x)`, `erase(x)`	$O(1 + n/B)$	$O(n)$
`begin()`, `next(p)`	$\begin{cases} \text{si } n = 0, & O(B); \\ \text{si } n \leq B, & O(B/n); \\ \text{si } n > B & O(1); \end{cases}$	$O(B)$
`clear()`	$O(n + B)$	$O(n + B)$

Tabla 4.5: Tiempos de ejecución de los métodos del TAD diccionario implementado con tablas de dispersión abiertas.

En la Tabla 4.5 vemos los tiempos de ejecución correspondientes. El tiempo de ejecución de `next()` y `begin()` se debe a que llaman a `next_aux()`.

4.5.3. Funciones de dispersión

De los costos que hemos analizado para las tablas de dispersión abierta se deduce que para que éstas sean efectivas los elementos deben ser distribuidos uniformemente sobre las cubetas. El diseño de una buena función de dispersión es precisamente ése. Pensemos por ejemplo en el caso de una tabla de dispersión para strings con B=256 cubetas. Como función de dispersión podemos tomar

$$h_1(x) = \text{(Código ASCII del primer caracter de } x) \tag{4.18}$$

Si ésta función de dispersión es usada con strings que provienen por ejemplo de palabras encontradas encontradas en texto usual en algún lenguaje como español, entonces es probable que haya muchas más palabras que comiencen con la letra `a` y por lo tanto vayan a la cubeta 97 (el valor ASCII de `a`) que con la letra `x` (cubeta 120).

```
1. int h2(string s) {
2.   int v = 0;
3.   for (int j=0; j<s.size(); j++) {
4.     v += s[j];
5.     v = v % 256;
6.   }
7.   return v;
8. }
```

Código 4.11: *Función de dispersión para strings [Archivo: hash.cpp]*

Una mejor posibilidad es la función `h2()` mostrada en el código 4.11. Esta función calcula la suma de los códigos ASCII de todos los caracteres del string, módulo 256. Notamos primero que basta con que dos strings tengan un sólo caracter diferente para que sus valores de función de dispersión sean diferentes. Por ejemplo, los strings `argonauta` y `argonautas` irán a diferentes cubetas ya que difieren en un caracter. Sin embargo las palabras `vibora` y `bravio` irán a la misma ya que los caracteres son los mismos, pero en diferente orden (son anagramas la una de la otra). Sin embargo, parece

intuitivo que en general h_2 dispersará mucho mejor los strings que h_1. En última instancia el criterio para determinar si una función de dispersión es buena o no es estadístico. Se deben considerar *"ensambles"* de elementos (en este ejemplo strings) representativos y ver que tan bien se desempeñan las funciones potenciales para esos ensambles.

4.5.4. Tablas de dispersión cerradas

Otra posibilidad para resolver el problema de las colisiones es usar otra cubeta cercana a la que indica la función de dispersión. A estas estrategias se les llama de *"redispersión"*. La tabla es un `vector<key_t>` e inicialmente todos los elementos están inicializados a un valor especial que llamaremos `undef` (*"indefinido"*). Si el diccionario guarda valores enteros positivos podemos usar 0 como `undef`. Si los valores son reales (`double`'s o `float`'s) entonces podemos usar como `undef` el valor `DBL_MAX` (definido en el header `float.h`). Este es el valor más grande representable en esa computadora (en el caso de un procesador de 32bits con el SO GNU/Linux y el compilador GCC es `1.7976931348623157e+308`). También puede ser `NAN`. Si estamos almacenando nombres de personas podemos usar la cadena `<NONE>` (esperando que nadie se llame de esa forma) y así siguiendo. Para insertar un elemento x calculamos la función de dispersión `init`=$h(x)$ para ver cuál cubeta le corresponde *inicialmente*. Si la cubeta está libre (es decir el valor almacenado es `undef`), entonces podemos insertar el elemento en esa posición. Si está ocupado, entonces podemos probar en la siguiente cubeta (en *"sentido circular"*, es decir si la cubeta es $B-1$ la siguientes es la 0) `(init+1)%B`. Si está libre lo insertamos allí, si está ocupada y el elemento almacenado no es x, seguimos con la siguiente, etc... hasta llegar a una libre. Si todas las cubetas están ocupadas entonces la tabla está llena y el programa debe señalar un error (o agrandar la tabla dinámicamente). *Nota:* En las STL la implementación de diccionario corresponde al header `unordered_set` (esto recientemente fue cambiado, en versiones anteriores el nombre era `hash_set` (pero no era parte del estándar)).

Consideremos por ejemplo una tabla de dispersión para enteros con $B = 10$ cubetas y dispersión lineal ($h(x) = x$), insertando los elementos $1, 13, 4, 1, 24, 12, 15, 34, 4, 44, 22, 15, 17, 19$. Al insertar los elementos 1,13 y 4 las cubetas respectivas (0,3,4) están vacías y los elementos son insertados. Al insertar el cuarto elemento (un 1) la cubeta respectiva (la 1) está ocupada, pero el elemento almacenado es el 1, de manera que no hace falta insertarlo nuevamente. El quinto elemento es un 24. Al querer insertarlo

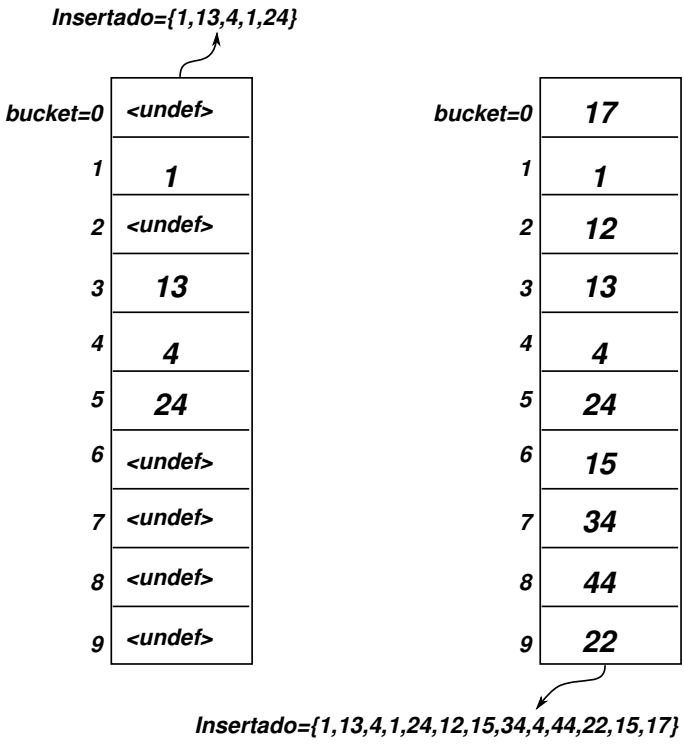


Figura 4.3: Ejemplo de inserción en tablas cerradas.

en la cubeta correspondiente (la 4) vemos que está ocupada y el elemento almacenado es un 4 (no un 24), de manera que probamos en la siguiente. Como está vacía el elemento es insertado. A esta altura la tabla está como se muestra en la figura 4.3 a la izquierda. Después de aplicar el procedimiento a todos los elementos a insertar, hasta el 17, la tabla queda como se muestra en la misma figura, a la derecha. Al querer insertar el 19 vemos que todas las cubetas están ocupadas, de manera que se debe señalar un error (o agrandar la tabla dinámicamente).

Ahora consideremos que ocurre al hacer `p=find(x)`. Por supuesto no basta con buscar en la cubeta $h(x)$ ya que si al momento de insertar esa cubeta estaba llena, entonces el algoritmo de inserción lo insertó en la siguiente cubeta vacía, es decir que x puede estar en otra cubeta. Sin embargo, no hace falta revisar todas las cubetas, si revisamos las cubetas a partir de $h(x)$, es decir la $h(x)+1$, $h(x)+2$... entonces cuando lleguemos a alguna cubeta que contiene a x podemos devolver el iterator correspondiente. Pero

también si llegamos a un **undef**, sabemos que el elemento *"no está"* en el diccionario, ya que si estuviera debería haber sido insertado en alguna cubeta entre $h(x)$ y el siguiente **undef**. De manera que al encontrar el primer **undef** podemos retornar **end()**.

Hasta ahora no hemos discutido como hacer las supresiones. Primero discutiremos en las secciones siguientes el costo de las operaciones de inserción y búsqueda, luego en la sección §4.5.4.4).

4.5.4.1. Costo de la inserción exitosa

Definimos la *"tasa de ocupación"* α como

$$\alpha = \frac{n}{B} \tag{4.19}$$

El costo de insertar un nuevo elemento (una *"inserción exitosa"*) es proporcional al número m de cubetas ocupadas que hay que recorrer hasta encontrar una cubeta libre. Cuando $\alpha \ll 1$, (muy pocas cubetas ocupadas) la probabilidad de encontrar una cubeta libre es grande y el costo de inserción es $O(1)$. A medida que la tabla se va llenando la probabilidad de encontrar una serie de cubetas ocupadas es alta y el costo de inserción ya no es $O(1)$.

Consideremos la cantidad de cubetas ocupadas que debemos recorrer hasta encontrar una cubeta libre en una tabla como la de la figura 4.3 a la izquierda. Consideremos que la probabilidad de que una dada cubeta sea la cubeta inicial es la misma para todas las cubetas. Si la cubeta inicial es una vacía (como las 0,2,6,7,8,9) entonces no hay que recorrer ninguna cubeta ocupada, es decir $m = 0$. Si queremos insertar en la cubeta 1, entonces ésta está ocupada, pero la siguiente está vacía, con lo cual debemos recorrer $m = 1$ cubetas. El peor caso es al querer insertar en una cubeta como la 3, para la cual hay que recorrer $m = 3$ cubetas antes de encontrar la siguiente vacía, que es la 6. Para las cubetas 4 y 5 tenemos $m = 2$ y 1 respectivamente. El m promedio (que denotaremos por $\langle m \rangle$ es la suma de los m para cada cubeta dividido el número de cubetas. En este caso es $(1 + 3 + 2 + 1)/10 = 0.7$. Notar que en este caso en particular es diferente de $\alpha = n/B$ debido a que hay secuencias de cubetas ocupadas contiguas.

Si todas las cubetas tienen la misma probabilidad de estar ocupadas α, entonces la probabilidad de que al insertar un elemento ésta este libre ($m = 0$ intentos) es $P(0) = 1 - \alpha$. Para que tengamos que hacer un sólo intento debe ocurrir que la primera esté llena y la siguiente vacía. La probabilidad de que esto ocurra es $P(1) = \alpha(1 - \alpha)$. En realidad aquí hay una

Nro. de intentos infructuosos (m)	**Probabilidad de ocurrencia** $P(m) = \alpha^m(1-\alpha)$
0	0.250000
1	0.187500
2	0.140625
3	0.105469
4	0.079102
5	0.059326
6	0.044495
7	0.033371
8	0.025028
9	0.018771
10	0.014078

Tabla 4.6: Probabilidad de realizar m intentos en una tabla cerrada cuando la tasa de ocupación es $\alpha = 0.75$

aproximación. Supongamos que $B = 100$ de las cuales 75 están ocupadas. Si el primer intento da con una cubeta ocupada, entonces la probabilidad de que la segunda esté libre es un poco mayor que $25/100 = (1-\alpha)$ ya que sabemos que en realidad de las 99 cubetas restantes hay 25 libres, de manera que la probabilidad de encontrar una libre es en realidad $25/99 \approx 0.253 > 0.25$. Por simplicidad asumiremos que la aproximación es válida. Para dos intentos, la probabilidad es $\alpha^2(1-\alpha)$ y así siguiendo, la probabilidad de que haya que hacer m intentos es

$$P(m) = \alpha^m(1-\alpha) \tag{4.20}$$

Por ejemplo, si la tasa de ocupación es $\alpha = 0.75$, entonces la probabilidad de tener que hacer m intentos es como se muestra en la Tabla 4.6. La cantidad de intentos en promedio será entonces

$$\begin{aligned} \langle m \rangle &= \sum_{m=0}^{B-1} m\, P(m) \\ &= \sum_{m=0}^{B-1} m\, \alpha^m(1-\alpha) \end{aligned} \tag{4.21}$$

Suponiendo que B es relativamente grande y α no está demasiado cerca de uno, podemos reemplazar la suma por una suma hasta $m = \infty$, ya que los términos decrecen muy fuertemente al crecer m. Para hacer la suma necesitamos hacer un truco matemático, para llevar la serie de la forma $m\alpha^m$ a una geométrica simple, de la forma α^m

$$m\alpha^m = \alpha \frac{\mathrm{d}}{\mathrm{d}\alpha} \alpha^m \tag{4.22}$$

de manera que

$$\begin{aligned} \langle m \rangle &= \sum_{m=0}^{\infty} (1-\alpha)\, \alpha \frac{\mathrm{d}}{\mathrm{d}\alpha} \alpha^m \\ &= \alpha\,(1-\alpha) \frac{\mathrm{d}}{\mathrm{d}\alpha} \left(\sum_{k=0}^{\infty} \alpha^m \right) \\ &= \alpha\,(1-\alpha) \frac{\mathrm{d}}{\mathrm{d}\alpha} \left(\frac{1}{1-\alpha} \right) \\ &= \frac{\alpha}{1-\alpha} \end{aligned} \tag{4.23}$$

Por ejemplo, en el caso de tener $B = 100$ cubetas y $\alpha = 0.9$ (90 % de cubetas ocupadas) el número de intentos medio es de $\langle m \rangle = 0.9/(1 - 0.9) = 9$.

4.5.4.2. Costo de la inserción no exitosa

Llamaremos una *"inserción no exitosa"* cuando al insertar un elemento este ya está en el diccionario y por lo tanto en realidad no hay que insertarlo. El costo en este caso es menor ya que no necesariamente hay que llegar hasta una cubeta vacía, el elemento puede estar en cualquiera de las cubetas ocupadas. El número de intentos también depende de en qué momento fue insertado el elemento. Si fue insertado en los primeros momentos, cuando la tabla estaba vacía (α pequeños), entonces es menos probable que el elemento esté en el segmento final de largas secuencias de cubetas llenas. Por ejemplo en el caso de la tabla en la figura 4.3 a la izquierda, el elemento 4 que fue insertado al principio necesita $m = 0$ intentos infructuosos (es encontrado de inmediato), mientras que el 24, que fue insertado después, necesitará $m = 1$.

Si la tabla tiene una tasa de ocupación α, entonces un elemento x puede haber sido insertado en cualquier momento previo en el cual la tasa era α', con $0 \leq \alpha' \leq \alpha$. Si fue insertado al principio ($\alpha' \approx 0$) entonces habrá

que hacer pocos intentos infructuosos para encontrarlo, si fue insertado recientemente ($\alpha' \approx \alpha$) entonces habrá que hacer el mismo número promedio de intentos infructuosos que el que hay que hacer ahora para insertar un elemento nuevo, es decir $\alpha/(1-\alpha)$. Si asumimos que el elemento pudo haber sido insertado en cualquier momento cuando $0 \leq \alpha' \leq \alpha$, entonces el número de intentos infructuosos promedio será

$$\langle m_{\text{n.e.}} \rangle = \frac{1}{\alpha} \int_{\alpha'=0}^{\alpha} \langle m \rangle \, \mathrm{d}\alpha' \tag{4.24}$$

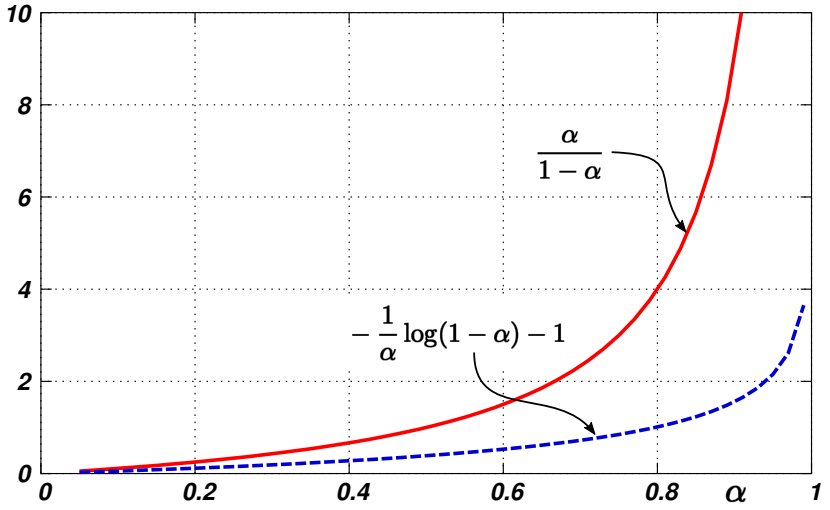


Figura 4.4: Eficiencia de las tablas de dispersión.

Reemplazando $\langle m \rangle$ de (4.23), tenemos que

$$\begin{aligned}
\langle m_{\text{n.e.}} \rangle &= \frac{1}{\alpha} \int_{\alpha'=0}^{\alpha} \frac{\alpha'}{1-\alpha'} \, \mathrm{d}\alpha' \\
&= \frac{1}{\alpha} \int_{\alpha'=0}^{\alpha} \left(\frac{1}{1-\alpha'} - 1 \right) \mathrm{d}\alpha' \\
&= \frac{1}{\alpha} \left(-\log(1-\alpha') - \alpha' \right) \Big|_{\alpha'=0}^{\alpha} \\
&= -\frac{1}{\alpha} \log(1-\alpha) - 1
\end{aligned} \tag{4.25}$$

4.5.4.3. Costo de la búsqueda

Ahora analicemos el costo de **p=find(k)**. El costo es $O(1 + \langle m \rangle)$, donde $\langle m \rangle$ es el número de intentos infructuosos. Si el elemento no está en el diccionario (*búsqueda no exitosa*), entonces el número de intentos infructuosos es igual al de intentos infructuosos para inserción exitosa analizado en la sección §4.5.4.1 dado por (4.23). Por otra parte, si el elemento está en el diccionario (*búsqueda exitosa*), el número de intentos infructuosos es sensiblemente menor, ya que si el elemento fue insertado al comienzo, cuando la tabla estaba vacía es probable que esté en las primeras cubetas, bien cerca de $h(x)$. El análisis es similar al de la inserción no exitosa analizado en la sección §4.5.4.2, dado por (4.25).

4.5.4.4. Supresión de elementos

Al eliminar un elemento uno estaría tentado de reemplazar el elemento por un **undef**. Sin embargo, esto dificultaría las posibles búsquedas futuras ya que ya no sería posible detenerse al encontrar un **undef** para determinar que el elemento no está en la tabla. La solución es introducir otro elemento **deleted** (*"eliminado"*) que marcará posiciones donde previamente hubo alguna vez un elemento que fue eliminado. Por ejemplo, para enteros positivos podríamos usar **undef**=0 y **deleted**=-1. Ahora, al hacer **p=find(x)** debemos recorrer las cubetas siguientes a $h(x)$ hasta encontrar x o un elemento **undef**. Los elementos **deleted** son tratados como una cubeta ocupada más. Sin embargo, al hacer un **insert(x)** de un nuevo elemento, podemos insertarlo en posiciones **deleted** además de **undef**.

4.5.4.5. Costo de las funciones cuando hay supresión

El análisis de los tiempos de ejecución es similar al caso cuando no hay supresiones, pero ahora la tasa de ocupación debe incluir a los elementos **deleted**, es decir en base a la *"tasa de ocupación efectiva"* α' dada por

$$\alpha' = \frac{n + n_{\text{del}}}{B} \tag{4.26}$$

donde ahora n_{del} es el número de elementos **deleted** en la tabla.

Esto puede tener un impacto muy negativo en la eficiencia, si se realizan un número de inserciones y supresiones elevado. Supongamos una tabla con $B =$100 cubetas en la cual se insertan 50 elementos distintos, y a partir de allí se ejecuta un lazo infinito en el cual se inserta un nuevo elemento

al azar y se elimina otro del conjunto, también al azar. Después de cada ejecución del cuerpo del lazo el número de elementos se mantiene en $n = 50$ ya que se inserta y elimina un elemento. La tabla nunca se llena, pero el número de suprimidos n_{del} crece hasta que eventualmente llega a ser igual $B - n$, es decir todas las cubetas están ocupadas o bien tienen **deleted**. En ese caso la eficiencia se degrada totalmente, cada operación (de hecho cada **locate()**) recorre toda la tabla, ya que en ningún momento encuentra un **undef**. De hecho, esto ocurre en forma relativamente rápida, en $B - n$ supresiones/inserciones.

4.5.4.6. Reinserción de la tabla

Si el destino de la tabla es para insertar una cierta cantidad de elementos y luego realizar muchas consultas, pero pocas inserciones/supresiones, entonces el esquema presentado hasta aquí es razonable. Por el contrario, si el ciclo de inserción supresión sobre la tabla va a ser continuo, el incremento en los el número de elementos **deleted** causará tal deterioro en la eficiencia que no permitirá un uso práctico de la tabla. Una forma de corregir esto es *"reinsertar"* los elementos, es decir, se extraen todos los elementos guardándolos en un contenedor auxiliar (vector, lista, etc...) limpiando la tabla, y reinsertando todos los elementos del contenedor. Como inicialmente la nueva tabla tendrá todos los elementos **undef**, y las inserciones no generan elementos **deleted**, la tabla reinsertada estará libre de **deleted**'s. Esta tarea es $O(B + n)$ y se ve compensada por el tiempo que se ahorrará en unas pocas operaciones.

Para determinar cuando se dispara la reinserción se controla la tasa de suprimidos que existe actualmente

$$ß = \frac{n_{\text{del}}}{B} \tag{4.27}$$

Cuando $ß \approx 1 - \alpha$ quiere decir que de la fracción de cubetas no ocupadas $1 - \alpha$, una gran cantidad de ellas dada por la fracción $ß$ está ocupada por suprimidos, degradando la eficiencia de la tabla. Por ejemplo, si $\alpha = 0.5$ y $ß = 0.45$ entonces 50 % de las cubetas está ocupada, y del restante 50 % el 45 % está ocupado por **deleted**. En esta situación la eficiencia de la tabla es equivalente una con un 95 % ocupado.

En la reinserción continua, cada vez que hacemos un borrado hacemos una series de operaciones para dejar la tabla sin ningún **deleted**. Recordemos que al eliminar un elemento no podemos directamente insertar un **undef**

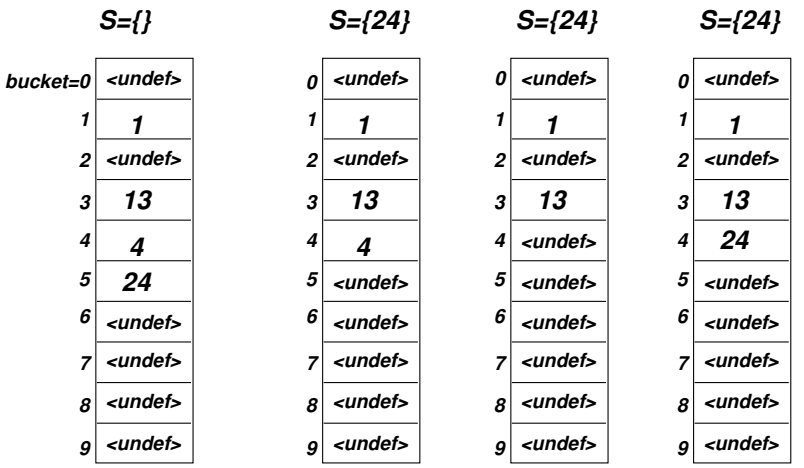


Figura 4.5: Proceso de borrado con redispersión continua.

en su lugar ya que de hacer el algoritmo de búsqueda no encontraría a los elementos que están después del elemento insertado y hasta el siguiente **undef**. Por ejemplo, si consideramos la tabla de la figura 4.5 a la izquierda, entonces si queremos eliminar el elemento 4, no podemos simplemente insertar insertar un **undef** ya que no encontraríamos después el 24. Pero, si quisiéramos eliminar un elemento como el 24 entonces sí podemos insertar allí un **undef**, ya que no queda ningún elemento entre la cubeta ocupada por el 24 y el siguiente **undef**. Ahora volvamos al caso en que queremos eliminar el 4. Podemos eliminar temporariamente el 24, guardándolo en una pila auxiliar S (puede ser cualquier otro contenedor como lista, vector o cola), y reemplazándolo por un **undef**, quedando en el estado mostrado en la segunda tabla desde la izquierda. De esta forma ahora el 4 está justo antes del **undef** y sí podemos reemplazarlo por un **undef**, quedando como la tercera tabla desde la izquierda. Ahora, finalmente, debemos reinsertar todos los elementos que están en S (en este caso sólo el 24), de nuevo en la tabla. En el caso más general, deberíamos guardar en el contenedor auxiliar todos los elementos que están entre la cubeta a suprimir y el siguiente (en sentido circular) **undef**. Es claro que de esta forma no tenemos en ningún momento elementos **deleted** en la tabla, ya que al eliminar el elemento nos aseguramos de que así sea. Las inserciones siguientes seguramente no generan elementos **deleted** (ninguna inserción los puede generar).

4.5.4.7. Costo de las operaciones con supresión

Si se usa reinserción continua, entonces no hay en ningún momentos elementos **deleted** y el número de intentos infructuosos es $\langle m \rangle = \alpha/(1-\alpha)$. Las operaciones **find(x)** e **insert(x)** son ambas $O(1+\langle m \rangle)$, pero **erase(x)** es $O(1+\langle m \rangle^2)$ ya que al reinsertar los elementos hay en promedio $\langle m \rangle$ llamadas a **insert()** el cuál es a su vez $O(1+\langle m \rangle)$.

En la reinserción discontinua, el costo de las tres operaciones es $O(1+\langle m \rangle)$, pero $\langle m \rangle = \alpha'/(1-\alpha')$, es decir que la tasa de ocupación efectiva crece con el número de elementos **deleted**.

4.5.4.8. Estrategias de redispersión

Al introducir las tablas de dispersión cerrada (sección §4.5.4) hemos explicado como la redispersión permite resolver las colisiones, buscando en las cubetas $h(x)+j$, para $j = 0, 1, ..., B-1$, hasta encontrar una cubeta libre. A esta estrategia se le llama *"de redispersión lineal"*. Podemos pensar que si la función de dispersión no es del todo buena, entonces ciertas cubetas o ciertas secuencias de cubetas cercanas pueden tender a llenarse más que otras. Esto puede ocasionar que tiendan a formarse localmente secuencias de cubetas ocupadas, incluso cuando la tasa de ocupación no es tan elevada globalmente. En estos casos puede ayudar el tratar de no dispersar las cubetas en forma lineal sino de la forma $h(x)+d_j$, donde d_0 es 0 y $d_j, j = 1, ..., B-1$ es una permutación de los números de 1 a $B-1$. Por ejemplo, si $B = 8$, podríamos tomar alguna permutación aleatoria como $d_j = 7, 2, 5, 4, 1, 0, 3, 6$. La forma de generar estas permutaciones es similar a la generación de números aleatorios, sin embargo está claro que no son números aleatorios, ya que la secuencia d_j debe ser la misma durante todo el uso de la tabla. Los elementos a tener en cuenta para elegir las posibles funciones de redispersión son las siguientes

- Debe ser determinística, es decir d_j debe ser sólo función de j durante todo el uso de la tabla.
- Debe ser $d_0 = 0$ y los demás elementos d_1 a d_{B-1} deben ser una permutación de los enteros 1 a $B-1$, por ejemplo $d_j = \mathrm{rem}(mj, B)$ es válida si m y B son *"coprimos"* (no tienen factores en común). Por ejemplo si $B = 8$ entonces $m = 6$ no es válida ya que m y B tienen el factor 2 en común y la secuencia generada es $d_j = \{0, 6, 4, 2, 0, 6, 4, 2\}$, en la cual se generan sólo los números pares.

Sin embargo $m = 9$ está bien, ya que es coprimo con B y genera la secuencia $d_j = \{0, 7, 6, 5, 4, 3, 2, 1\}$.

Una posibilidad es generar una permutación aleatoria de los números de 0 a $B - 1$ y guardarla en un vector, como un miembro estático de la clase. De está forma habría que almacenar un sólo juego de d_j para todas las tablas. Esto serviría si tuviéramos muchas tablas relativamente pequeñas de dispersión pero no serviría si tuviéramos una sola tabla grande ya que en ese caso el tamaño de almacenamiento para los d_j sería comparable con el de la tabla misma.

Por otra parte, si los d_j se calculan en tiempo de ejecución cada vez que se va a aplicar una de las funciones **`find()`**, **`insert()`** o **`erase()`** entonces el costo de calcular cada uno de los d_j es importante ya que para aplicar una sola de esas funciones (es decir un sólo **`insert()`** o **`erase()`**) puede involucrar varias evaluaciones de d_j (tantos como intentos infructuosos).

Una posible estrategia para generar los d_j es la siguiente. Asumamos que B es una potencia de 2 y sea k un número entre 0 y $B-1$ a ser definido más adelante. Se comienza con un cierto valor para d_1 y los d_{j+1} se calculan en términos del número anterior d_j con el algoritmo siguiente

$$d_{j+1} = \begin{cases} 2d_j < B & ; 2d_j \\ 2d_j \geq B & ; (2d_j - B) \oplus k \end{cases} \tag{4.28}$$

En esta expresión el operador $\oplus$ indica el operador *"o exclusivo bit a bit"*. Consiste en escribir la expresión de cada uno de los argumentos y aplicarles el operador *"xor"* bit a bit. Por ejemplo, si queremos calcular $25_{10} \oplus 13_{10}$ primero calculamos las expresiones binarias $25_{10} = 11001_2$ y $13_{10} = 1101_2$. El resultado es entonces

$$\begin{aligned} 25_{10} &= 11001_2 \\ 13_{10} &= 01101_2 \\ 25_{10} \oplus 13_{10} &= 10100_2 \end{aligned} \tag{4.29}$$

```
1.  int B=8, k=3, d=5;
2.  for (int j=2; j<B; j++) {
3.    int v = 2*d;
4.    d = (v<B ? v : (v-B)^k);
5.  }
```

Código 4.12: *Generación de los índices de redispersión d_j [Archivo: redisp.cpp]*

Esta operación puede ser implementada en forma eficiente en C++ usando el operador *"bitwise xor"* denotado por ^. El fragmento de código genera los d_j para el caso $B = 8$, tomando $k = 3$ y $d_1 = 5$. En este caso los códigos generados son $d_j = \{0, 5, 1, 2, 4, 3, 6, 7\}$. Sin embargo, no está garantizado que cualquier combinación k y d_1 genere una permutación válida, es decir, para algunas combinaciones puede ser que se generen elementos repetidos. En realidad el valor importante es k, si una dada combinación k, d_1 es válida, entonces ese k será válido con cualquier otro valor de d_1 en el rango $[1, B)$. No existe una forma sencilla de predecir para un dado B cuál es un k válido. Pero, asumiendo que el tamaño de las tablas será a lo sumo de $2^{30} \approx 10^9$ elementos (si fueran enteros de 4bytes, una tal tabla ocuparía unos 4 GByte de memoria), podemos calcular previamente los $k(p)$ apropiados para cada $B = 2^p$.

4.5.4.9. Detalles de implementación

```
1.  typedef int iterator_t;
2.  typedef int (*hash_fun)(key_t x);
3.  typedef int (*redisp_fun)(int j);
4.
5.  int linear_redisp_fun(int j) { return j; }
6.
7.  class hash_set {
8.  private:
9.    hash_set(const hash_set&) {}
10.   hash_set& operator=(const hash_set&) {}
11.   int undef, deleted;
12.   hash_fun h;
13.   redisp_fun rdf;
14.   int B;
15.   int count;
16.   std::vector<key_t> v;
17.   std::stack<key_t> S;
18.   iterator_t locate(key_t x,iterator_t &fdel) {
19.     int init = h(x);
20.     int bucket;
21.     bool not_found = true;
```

```
22.       for (int i=0; i<B; i++) {
23.         bucket = (init+rdf(i)) % B;
24.         key_t vb = v[bucket];
25.         if (vb==x || vb==undef) break;
26.         if (not_found && vb==deleted) {
27.           fdel=bucket;
28.           not_found = false;
29.         }
30.       }
31.       if (not_found) fdel = end();
32.       return bucket;
33.     }
34.     iterator_t next_aux(iterator_t bucket) {
35.       int j=bucket;
36.       while(j!=B && (v[j]==undef || v[j]==deleted)) {
37.         j++;
38.       }
39.       return j;
40.     }
41.   public:
42.     hash_set(int B_a,hash_fun h_a,
43.              key_t undef_a,key_t deleted_a,
44.              redisp_fun rdf_a=&linear_redisp_fun)
45.       : B(B_a), undef(undef_a), v(B,undef_a), h(h_a),
46.         deleted(deleted_a), rdf(rdf_a), count(0)
47.     { }
48.     std::pair<iterator_t, bool>
49.       insert(key_t x) {
50.       iterator_t fdel;
51.       int bucket = locate(x,fdel);
52.       if (v[bucket]==x)
53.         return std::pair<iterator_t,bool>(bucket,false);
54.       if (fdel!=end()) bucket = fdel;
55.       if (v[bucket]==undef || v[bucket]==deleted) {
56.         v[bucket]=x;
57.         count++;
58.         return std::pair<iterator_t,bool>(bucket,true);
59.       } else {
60.         std::cout << "Tabla de dispersion llena!!\n";
61.         abort();
62.       }
63.     }
64.     key_t retrieve(iterator_t p) { return v[p]; }
65.     iterator_t find(key_t x) {
66.       iterator_t fdel;
67.       int bucket = locate(x,fdel);
```

```
68.       if (v[bucket]==x) return bucket;
69.       else return(end());
70.     }
71.     int erase(const key_t& x) {
72.       iterator_t fdel;
73.       int bucket = locate(x,fdel);
74.       if (v[bucket]==x) {
75.         v[bucket]=deleted;
76.         count--;
77.           // Trata de purgar elementos 'deleted'
78.           // Busca el siguiente elemento 'undef'
79.           int j;
80.           for (j=1; j<B; j++) {
81.             op_count++;
82.             int b = (bucket+j) % B;
83.             key_t vb = v[b];
84.             if (vb==undef) break;
85.             S.push(vb);
86.             v[b]=undef;
87.             count--;
88.           }
89.           v[bucket]=undef;
90.           // Va haciendo erase/insert de los elementos
91.           // de atras hacia adelante hasta que se llene
92.           // 'bucket'
93.           while (!S.empty()) {
94.             op_count++;
95.             insert(S.top());
96.             S.pop();
97.           }
98.         return 1;
99.       } else return 0;
100.    }
101.    iterator_t begin() {
102.      return next_aux(0);
103.    }
104.    iterator_t end() { return B; }
105.    iterator_t next(iterator_t p) {
106.      return next_aux(p++);
107.    }
108.    void clear() {
109.      count=0;
110.      for (int j=0; j<B; j++) v[j]=undef;
111.    }
112.    int size() { return count; }
113.  };
```

Código 4.13: *Implementación de diccionario con tablas de dispersión cerrada y redispersión continua. [Archivo: hashsetbash.h]*

En el código 4.13 se puede ver una posible implementación del TAD diccionario con tablas de dispersión cerrada y redispersión continua.

- El **typedef hash_fun** define el tipo de funciones que pueden usarse como funciones de dispersión (toman como argumento un elemento de tipo **key_t** y devuelven un entero). La función a ser usada es pasada en el constructor y guardada en un miembro **hash_fun h**.
- El **typedef redisp_fun** define el tipo de funciones que pueden usarse para la redispersión (los d_j). Es una función que debe tomar como argumento un entero j y devolver otro entero (el d_j). La función a ser usada es pasada por el usuario en el constructor y guardada en un miembro (**redisp_fun rdf**). Por defecto se usa la función de redispersión lineal (**linear_redisp_fun()**).
- El miembro dato **int B** es el número de cubetas a ser usada. Es definido en el constructor.
- El miembro dato **int count** va a contener el número de elementos en la tabla (será retornado por **size()**).
- Las B cubetas son almacenadas en un **vector<key_t> v**.
- Casi todos los métodos de la clase usan la función auxiliar `iterator_t locate(key_t x,iterator_t &fdel)`. Esta función retorna un iterator de acuerdo a las siguiente reglas:
  - Si **x** está en la tabla **locate()** retorna la cubeta donde está almacenado el elemento **x**.
  - Si **x** no está en la tabla, retorna el primer **undef** después (en sentido circular) de la posición correspondiente a **x**.
  - Si no hay ningún **undef** (pero puede haber **deleted**) retorna alguna posición no especificada en la tabla.
  - Retorna por el argumento **fdel** la posición del primer **deleted** entre la cubeta correspondiente a **x** y el primer **undef**. Si no existe ningún **deleted** entonces retorna **end()**.
- La pila **S** es usada como contenedor auxiliar para la estrategia de reinserción. La reinserción se realiza en el bloque de las líneas 78–98

- La clase iterator consiste simplemente de un número de cubeta. `end()` es la cubeta ficticia `B`. La función `q=next(p)` que avanza iteradores, debe avanzar sólo sobre *cubetas ocupadas*. La función `next_aux()` avanza un iterador (en principio inválido) hasta llegar a uno ocupado o a `end()`. La función `q=next(p)` simplemente incrementa `p` y luego aplica `next_aux()`.
- La función `begin()` debe retornar el primer iterator válido (la primera cubeta ocupada) o bien `end()`. Para ello calcula el `next_aux(0)` es decir la primera cubeta válida después de (o igual a) la cubeta 0.

4.6. Conjuntos con árboles binarios de búsqueda

Una forma muy eficiente de representar conjuntos son los árboles binarios de búsqueda (ABB). Un árbol binario es un ABB si es vacío (Λ) o:

- Todos los elementos en los nodos del subárbol izquierdo son menores que el nodo raíz.
- Todos los elementos en los nodos del subárbol derecho son mayores que el nodo raíz.
- Los subárboles del hijo derecho e izquierdo son a su vez ABB.

4.6.1. Representación como lista ordenada de los valores

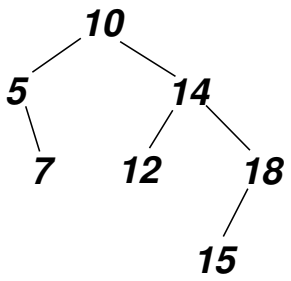


Figura 4.6: Ejemplos de árboles binarios de búsqueda

En la figura 4.6 vemos un posible árbol binario de búsqueda. La condición de ABB implica que si ordenamos todos los valores nodales de menor a mayor, quedan de la siguiente forma

$$\text{valores nodales ordenados} = \{(\text{rama izquierda}), r, (\text{rama derecha})\} \tag{4.30}$$

donde r es el valor nodal de la raíz. Por ejemplo, si ordenamos los valores nodales del ejemplo obtenemos la lista $\{5, 7, 10, 12, 14, 15, 18\}$ donde vemos que los valores de la rama izquierda 5 y 7 quedan a la izquierda de la raíz $r = 10$ y los valores de la rama derecha 12,14,15 y 18 quedan a la izquierda. Este ordenamiento es válido también para subárboles. Si n es el subárbol del nodo n, entonces todos los elementos del subárbol aparecen contiguos en la lista ordenada global y los valores nodales del subárbol guardan entre sí la relación indicada en (4.30). En el ejemplo, si consideramos el subárbol de 14, entonces vemos que todos los valores del subárbol aparecen juntos (en este caso al final) y entre sí respetan (4.30), a saber primero 12 (la rama izquierda), 14 (la raíz del subárbol) y 15,18 (la rama derecha).

4.6.2. Verificar la condición de ABB

```
1.  bool abb_p(aed::btree<int> &T,
2.          aed::btree<int>::iterator n,int &min,int &max) {
3.    aed::btree<int>::iterator l,r;
4.    int minr,maxr,minl,maxl;
5.    min = +INT_MAX;
6.    max = -INT_MAX;
7.    if (n==T.end()) return true;
8.
9.    l = n.left();
10.   r = n.right();
11.
12.   if (!abb_p(T,l,minl,maxl) || maxl>*n) return false;
13.   if (!abb_p(T,r,minr,maxr) || minr<*n) return false;
14.
15.   min = (l==T.end()? *n : minl);
16.   max = (r==T.end()? *n : maxr);
17.   return true;
18. }
19.
20. bool abb_p(aed::btree<int> &T) {
21.   if (T.begin()==T.end()) return false;
22.   int min,max;
23.   return abb_p(T,T.begin(),min,max);
24. }
```

Código 4.14: *Función predicado que determina si un dado árbol es ABB. [Archivo: abbp.cpp]*

En el código 4.14 vemos una posible función predicado **bool abb_p(T)** que determina si un árbol binario es ABB o no. Para ello usa una función recursiva auxiliar **bool abb_p(T,n,min,max)** que determina si el subárbol del nodo **n** es ABB y en caso de que si lo sea retorna a través de **min** y **max** los valores mínimos y máximos del árbol. Si no es ABB los valores de **min** y **max** están indeterminados. La función recursiva se aplica a cada uno de los hijos derechos e izquierdo, en caso de que alguno de estos retorne **false** la función retorna inmediatamente **false**. Lo mismo ocurre si el máximo del subárbol del hijo izquierdo **maxl** es mayor que el valor en el nodo ***n**, o el mínimo del subárbol del hijo derecho es menor que ***n**.

Por supuesto, si el nodo **n** está vacío (es **end()**) **abb_p()** no falla y retorna un valor mínimo de **+INT_MAX** y un valor máximo de **-INT_MAX**. (**INT_MAX** es el máximo entero representable y está definido en el header **float.h**). Estos valores garantizan que las condiciones de las líneas 12 y 13 no fallen cuando alguno de los hijos es Λ.

Si todas las condiciones se satisfacen entonces el mínimo de todo el subárbol de **n** es el mínimo del subárbol izquierdo, es decir **minl**, salvo sí el nodo izquierdo es **end()** en cuyo caso el mínimo es el valor de ***n**. Igualmente, el máximo del subárbol de **n** es **maxr** si el nodo izquierdo no es **end()** y ***n** si lo es.

4.6.3. Mínimo y máximo

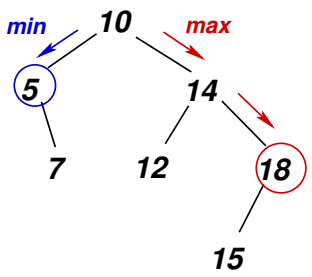


Figura 4.7:

Buscar los mínimos y máximos se puede hacer gráficamente en forma muy simple (ver figura 4.7).

- Para buscar el máximo se debe avanzar siempre por la derecha. Para encontrar el mínimo siempre por la izquierda.

- El listado en orden simétrico da la lista ordenada de los elementos del árbol.

4.6.4. Buscar un elemento

```
1.  node_t find(tree_t t,node_t n,T x) {
2.    if (n==t.end()) return t.end();
3.    if (x<*n) return find(t,n.left(),x)
4.    elsif (x>*n) return find(t,n.right(),x)
5.    else return n;
6.  }
```

La función previa permite encontrar un elemento en el árbol. Retorna la posición (nodo) donde el elemento está o retornar **end()** si el elemento no está.

4.6.5. Costo de mínimo y máximo

Si el árbol esta bien balanceado puede almacenar una gran cantidad de elementos en pocos niveles. Como las funciones descriptas previamente, (mínimo, máximo y buscar un elemento) siguen un camino en el árbol es de esperar que sean muy eficientes. Un *"árbol binario completo"* es un árbol que tiene todos sus niveles completamente ocupados, hasta un cierto nivel d. El árbol completo es el mejor caso en cuanto a balanceo de l árbol. Calculemos cuantos nodos tiene un árbol completo. En el primer nivel $l = 0$ sólo está la raíz, es decir 1 nodo. En el nivel $l = 1$ 2 nodos y en general 2^l nodos en el nivel l. Es decir que un árbol completo de altura d tiene

$$n = 1 + 2 + 2^2 + \ldots + 2^d = 2^{d+1} - 1 \tag{4.31}$$

nodos, y por lo tanto el número de niveles, en función del número de nodos es (ver sección §3.8.3)

$$d = \log_2(n+1) - 1 \tag{4.32}$$

Las funciones que iteran sólo sobre un camino en el árbol, como **insert(x)** o **find(x)**, tienen un costo $O(l)$ donde l es el nivel o profundidad del nodo desde la raíz (ver sección 3.1). En el caso del árbol completo todos los caminos tienen una longitud $l \leq d$, de manera que los tiempos de ejecución son $O(d) = O(\log(n))$.

En realidad, los nodos interiores van a tener una profundidad $l < d$ y no puede preguntarse si esto puede hacer bajar el costo promedio de las

operaciones. En realidad, como el número de nodos crece muy rápidamente, de hecho exponencialmente con el número de nivel, el número de nivel promedio es muy cercano a la profundidad máxima para un árbol completo. Consideremos un árbol completo con profundidad 12, es decir con $n = 2^{13} - 1 = 8191$ nodos. En el último nivel hay $2^{12} = 4096$ nodos, es decir más de la mitad. En el anteúltimo ($l = 11$) nivel hay 2048 nodos (1/4 del total) y en el nivel $l = 10$ hay 1024 (1/8 del total). Vemos que en los últimos 3 niveles hay más del 87 % de los nodos. La profundidad media de los nodos es

$$\langle l \rangle = \frac{1}{n} \sum_{\text{nodo } m} l(m) \tag{4.33}$$

donde m recorre los nodos en el árbol y $l(m)$ es la profundidad del nodo m. Como todos los nodos en el mismo nivel tienen la misma profundidad, tenemos

$$\langle l \rangle = \frac{1}{n} \sum_{l=0}^{d} (\text{Nro. de nodos en el nivel } l) \cdot l \tag{4.34}$$

Para el árbol completo tenemos entonces

$$\langle l \rangle = \frac{1}{n} \sum_{l=0}^{d} 2^l \, l \tag{4.35}$$

Esta suma se puede calcular en forma cerrada usando un poco de álgebra. Notemos primero que $2^l = \mathrm{e}^{l \log 2}$. Introducimos una variable auxiliar α tal que

$$2^l = \mathrm{e}^{\alpha l}\Big|_{\alpha = \log 2} \tag{4.36}$$

Lo bueno de introducir esta variable α es que la derivada con respecto a α de la exponencial baja un factor l, es decir

$$\frac{\mathrm{d}}{\mathrm{d}\alpha} \mathrm{e}^{\alpha l} = l \, \mathrm{e}^{\alpha l}. \tag{4.37}$$

Entonces

$$\begin{aligned} \langle l \rangle &= \frac{1}{n} \sum_{l=0}^{d} 2^l \, l \\ &= \frac{1}{n} \sum_{l=0}^{d} \left[\frac{\mathrm{d}\mathrm{e}^{\alpha l}}{\mathrm{d}\alpha} \right]_{\alpha = \log 2}, \\ &= \frac{1}{n} \frac{\mathrm{d}}{\mathrm{d}\alpha} \left[\sum_{l=0}^{d} \mathrm{e}^{\alpha l} \right]_{\alpha = \log 2}. \end{aligned} \tag{4.38}$$

Ahora la sumatoria es una suma geométrica simple de razón e^{α}, y se puede calcular en forma cerrada

$$\sum_{l=0}^{d} \mathrm{e}^{\alpha l} = \frac{\mathrm{e}^{\alpha(d+1)} - 1}{\mathrm{e}^{\alpha} - 1}. \tag{4.39}$$

Su derivada con respecto a α es

$$\frac{\mathrm{d}}{\mathrm{d}\alpha}\left[\frac{\mathrm{e}^{\alpha(d+1)} - 1}{\mathrm{e}^{\alpha} - 1}\right] = \frac{(d+1)\,\mathrm{e}^{\alpha(d+1)}\,(\mathrm{e}^{\alpha} - 1) - (\mathrm{e}^{\alpha(d+1)} - 1)\,\mathrm{e}^{\alpha}}{(\mathrm{e}^{\alpha} - 1)^2} \tag{4.40}$$

y reemplazando en (4.38) y usando $\mathrm{e}^{\alpha} = 2$ obtenemos

$$\begin{aligned} \langle l \rangle &= \frac{1}{n} \frac{(d+1)\,\mathrm{e}^{\alpha(d+1)}\,(\mathrm{e}^{\alpha} - 1) - (\mathrm{e}^{\alpha(d+1)} - 1)\,\mathrm{e}^{\alpha}}{(\mathrm{e}^{\alpha} - 1)^2} \\ &= \frac{(d+1)\,2^{d+1} - (2^{d+1} - 1)\,2}{2^{d+1} - 1} \\ &= \frac{(d-1)\,2^{d+1} + 2}{2^{d+1} - 1} \\ &\approx d - 1 \end{aligned} \tag{4.41}$$

La última aproximación proviene de notar que $2^{d+1} \gg 1$ para, digamos, $d > 10$.

Esto dice que en un árbol completo la longitud promedio de los caminos es la profundidad del árbol menos uno.

Por el contrario, el peor caso es cuando el árbol está completamente desbalanceado, es decir en cada nivel del árbol hay un sólo nodo y por lo tanto hay n niveles. En este caso $\langle l \rangle = n/2$ y los costos de **`insert()`** y **`find()`** son $O(n)$.

Notar que el balanceo del árbol depende del orden en que son insertados los elementos en el árbol. En la figura 4.8 se muestran los árboles obtenidos al insertar los enteros del 1 al 7, primero en forma ascendente, después descendente y finalmente en el orden $\{4, 2, 6, 1, 3, 5, 7\}$. Vemos que en los dos primeros casos el desbalanceo es total, el árbol degenera en dos listas por hijo derecho en el primer caso y por hijo izquierdo en el segundo. En el tercer caso los elementos son ingresados en forma desordenada y el balanceo es el mejor posible. El árbol resultante es un arbol completo hasta el nivel 3.

Puede demostrarse que si los valores son insertados en forma aleatoria entonces

$$\langle m \rangle \approx 1.4 \log_2 n, \tag{4.42}$$

con lo cual el caso de inserción aleatoria está muy cercano al mejor caso.

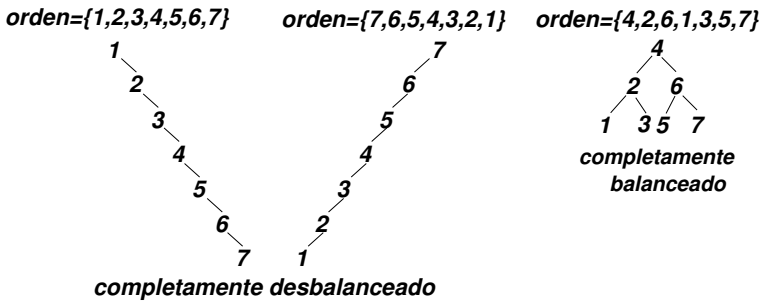


Figura 4.8: El balanceo del árbol depende del orden en que son ingresados los elementos.

4.6.6. Operación de inserción

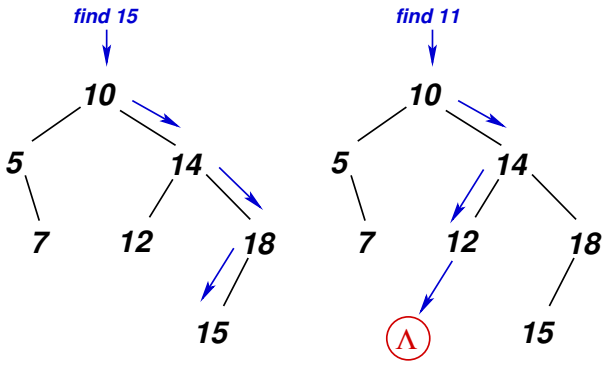


Figura 4.9:

Para insertar un elemento `x` hacemos un `find(x)`, si el nodo retornado es Λ insertamos allí, si no el elemento ya está. Una vez ubicada la posición donde el elemento debe ser insertada, la inserción es $O(1)$ de manera que toda la operación es básicamente la de `find(x)`: $O(\langle m \rangle) = O(\log_2 n)$.

4.6.7. Operación de borrado

Para borrar hacemos de nuevo el `n=find(x)`. Si el elemento no está (*"supresión no exitosa"*), no hay que hacer nada. Si está hay varios casos, dependiendo del número de hijos del nodo que contiene el elemento a eliminar.

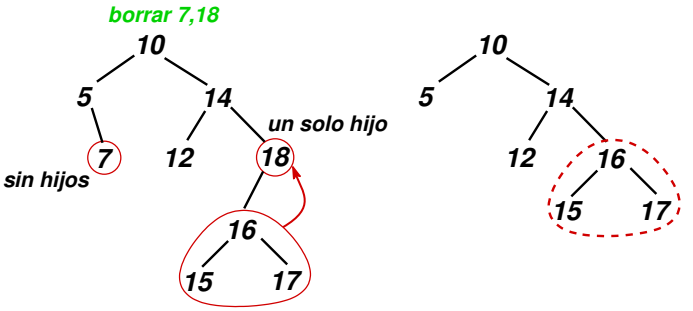


Figura 4.10: Supresión en ABB cuando el nodo no tiene o tiene un solo hijo.

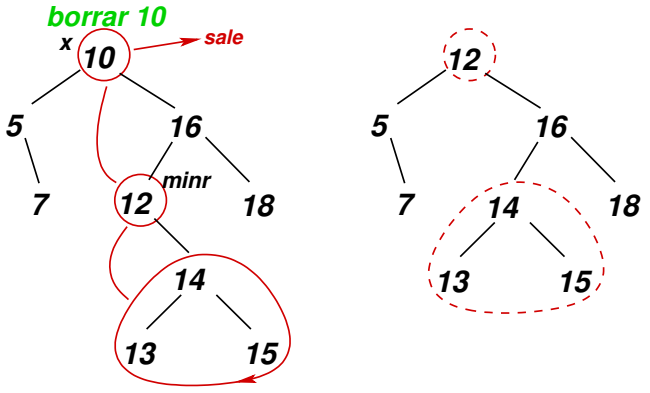


Figura 4.11: Supresión ABB cuando el nodo tiene dos hijos.

- Si no tiene hijos, como el elemento 7 en la figura 4.10 a la derecha, entonces basta con suprimir el nodo. Notar que si tiene hijos, no se puede aplicar directamente el `erase()` de árboles al nodo ya que eliminaría no sólo el elemento en cuestión sino todo su subárbol. Por eso consideramos a continuación los casos en que el nodo tiene uno y dos hijos.
- Si tiene un sólo hijo, como el elemento 18 en la figura 4.10 a la derecha, entonces basta con subir todo el subárbol del hijo existente (en este caso el izquierdo, ocupado por un 16) reemplazando al nodo eliminado.
- En el caso de tener dos hijos, como el elemento `x=10` en la figura 4.11, buscamos algún elemento de la rama derecha (también podría ser de la izquierda) para eliminar de la rama derecha y reemplazar `x` . Para que el árbol resultante siga siendo un árbol binario de búsqueda, este elemento debe ser el mínimo de la rama derecha, (llamémoslo `minr`)

en este caso ocupado por el 12. (También podría usarse el máximo de la rama izquierda.) Notar que, en general **minr** se buscaría como se explicó en la sección §4.6.3, pero a partir del hijo derecho (en este caso el 16). **minr** es eliminado de la rama correspondiente y reemplaza a **x**. Notar que como **minr** es un mínimo, necesariamente no tiene hijo izquierdo, por lo tanto al eliminar **minr** no genera una nueva cascada de eliminaciones. En el ejemplo, basta con subir el subárbol de 14, a la posición previamente ocupada por 12.

4.6.8. Recorrido en el árbol

Hemos discutido hasta aquí las operaciones propias del TAD conjunto, **find(x)**, **insert(x)** y **erase(x)**. Debemos ahora implementar las operaciones para recorrer el árbol en forma ordenada, como es propio del tipo **set<>** en las STL, en particular **begin()** debe retornar un iterator al menor elemento del conjunto y el operador de incremento **n++** debe incrementar el iterator **n** al siguiente elemento en forma ordenada. Estas operaciones son algo complejas de implementar en el ABB.

Para **begin()** basta con seguir el algoritmo explicado en la sección §4.6.3 partiendo de la raíz del árbol.

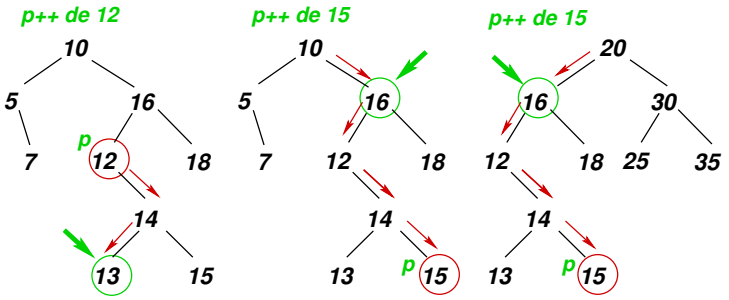


Figura 4.12: Operación p++ en set<> implementado por ABB.

Dado un nodo dereferenciable **n** la operación de incremento **n++** procede de la siguiente forma

- Si **n** tiene hijo derecho (como el 12 en la figura 4.12 a la izquierda) entonces el siguiente es el mínimo de la rama derecha. Por lo tanto basta con bajar por la derecha hasta el hijo derecho (en este caso el 14) y después seguir siempre por la izquierda hasta la última posición dereferenciable (en este caso el 13).

- Si el nodo no tiene hijo derecho, entonces hay que buscar en el camino que llega al nodo el último *"padre derecho"*. En el ejemplo de la figura 4.12 centro, 15 no tiene hijo derecho, por lo cual recorremos el camino desde la raíz que en este caso corresponde a los elementos $\{10, 16, 12, 14, 15\}$. Hay un sólo padre que es derecho en el camino (es decir tal que el nodo siguiente en el camino es hijo izquierdo) y es el 16. Notar que al ser 16 el último padre derecho, esto garantiza que 15 es el máximo de todo el subárbol izquierdo de 16, por lo tanto en si representamos todos los elementos del subárbol de 16 en forma ordenada, 15 está inmediatamente antes que el 16. Como todos los elementos del subárbol de 16 aparecen juntos en la lista ordenada global (ver sección §4.6.1) en la lista 16 será también el siguiente a 15 en la lista ordenada global. Esto ocurre también si el último padre derecho (16 en este caso) es hijo izquierdo como en el ejemplo de la la figura 4.12 a la derecha.

4.6.9. Operaciones binarias

Las funciones binarias requieren un párrafo aparte. Notemos primero que podemos implementar estas funciones en forma genérica. Por ejemplo podemos implementar **`set_union(A,B,C)`** insertando en **`C`** primero todos los elementos de **`A`** y después los de **`B`**. El mismo **`insert(x)`** se encargará de no insertar elementos duplicados. Si pudiéramos garantizar un tiempo $O(\log n)$ para cada inserción, entonces el costo global sería a lo sumo de $O(n \log n)$ lo cual sería aceptable. Sin embargo, en este caso si iteramos sobre los conjuntos con **`begin()`**, **`operator++()`** y **`end()`**, estaríamos insertando los elementos en forma ordenada en **`C`** con lo cual se produciría el peor caso discutido en la sección §4.6.5, en el cual el árbol degenera en una lista y las inserciones cuestan $O(n)$, dando un costo global para **`set_union()`** de $O(n^2)$, lo cual es inaceptable.

Para evitar esto usamos una estrategia en la cual insertamos **`A`** y **`B`** en forma recursiva, insertando primero la raíz y después los árboles izquierdo y derecho. De esta forma puede verse que si los árboles para **`A`** y **`B`** están bien balanceados el árbol resultante para **`C`** también lo estará.

Para **`set_intersection()`** adoptamos una estrategia similar. Para cada nodo en **`A`** insertamos primero el valor nodal si está también contenido en **`B`**, y después aplicamos recursivamente a las ramas izquierda y derecha. La estrategia para **`set_difference()`** es similar sólo que verificamos que el valor nodal *no esté* en **`B`**.

4.6.10. Detalles de implementación

```
1.  // Forward declarations
2.  template<class T>
3.  class set;
4.  template<class T> void
5.  set_union(set<T> &A,set<T> &B,set<T> &C);
6.  template<class T> void
7.  set_intersection(set<T> &A,set<T> &B,set<T> &C);
8.  template<class T> void
9.  set_difference(set<T> &A,set<T> &B,set<T> &C);
10.
11. template<class T>
12. class set {
13. private:
14.   typedef btree<T> tree_t;
15.   typedef typename tree_t::iterator node_t;
16.   tree_t bstree;
17.   node_t min(node_t m) {
18.     if (m == bstree.end()) return bstree.end();
19.     while (true) {
20.       node_t n = m.left();
21.       if (n==bstree.end()) return m;
22.       m = n;
23.     }
24.   }
25.
26.   void set_union_aux(tree_t &t,node_t n) {
27.     if (n==t.end()) return;
28.     else {
29.       insert(*n);
30.       set_union_aux(t,n.left());
31.       set_union_aux(t,n.right());
32.     }
33.   }
34.   void set_intersection_aux(tree_t &t,
35.                             node_t n, set &B) {
36.     if (n==t.end()) return;
37.     else {
38.       if (B.find(*n)!=B.end()) insert(*n);
39.       set_intersection_aux(t,n.left(),B);
40.       set_intersection_aux(t,n.right(),B);
41.     }
42.   }
43.   void set_difference_aux(tree_t &t,
```

```
44.                     node_t n, set &B) {
45.     if (n==t.end()) return;
46.     else {
47.      if (B.find(*n)==B.end()) insert(*n);
48.      set_difference_aux(t,n.left(),B);
49.      set_difference_aux(t,n.right(),B);
50.     }
51.   }
52.   int size_aux(tree_t t,node_t n) {
53.     if (n==t.end()) return 0;
54.     else return 1+size_aux(t,n.left())
55.          +size_aux(t,n.right());
56.   }
57.  public:
58.   class iterator {
59.   private:
60.     friend class set;
61.     node_t node;
62.     tree_t *bstree;
63.     iterator(node_t m,tree_t &t)
64.       : node(m), bstree(&t) {}
65.     node_t next(node_t n) {
66.       node_t m = n.right();
67.       if (m!=bstree->end()) {
68.         while (true) {
69.           node_t q = m.left();
70.           if (q==bstree->end()) return m;
71.           m = q;
72.         }
73.       } else {
74.         // busca el padre
75.         m = bstree->begin();
76.         if (n==m) return bstree->end();
77.         node_t r = bstree->end();
78.         while (true) {
79.           node_t q;
80.           if (*n<*m) { q = m.left(); r=m; }
81.           else q = m.right();
82.           if (q==n) break;
83.           m = q;
84.         }
85.         return r;
86.       }
87.     }
88.   public:
89.     iterator() : bstree(NULL) { }
```

```
90.       iterator(const iterator &n)
91.         : node(n.node), bstree(n.bstree) {}
92.       iterator& operator=(const iterator& n) {
93.         bstree=n.bstree;
94.         node = n.node;
95.       }
96.       const T &operator*() { return *node; }
97.       const T *operator->() { return &*node; }
98.       bool operator!=(iterator q) {
99.         return node!=q.node; }
100.      bool operator==(iterator q) {
101.        return node==q.node; }
102.
103.      // Prefix:
104.      iterator operator++() {
105.        node = next(node);
106.        return *this;
107.      }
108.      // Postfix:
109.      iterator operator++(int) {
110.        node_t q = node;
111.        node = next(node);
112.        return iterator(q,*bstree);
113.      }
114.    };
115.  private:
116.    typedef pair<iterator,bool> pair_t;
117.  public:
118.    set() {}
119.    set(const set &A) : bstree(A.bstree) {}
120.    ~set() {}
121.    pair_t insert(T x) {
122.      node_t q = find(x).node;
123.      if (q == bstree.end()) {
124.        q = bstree.insert(q,x);
125.        return pair_t(iterator(q,bstree),true);
126.      } else return pair_t(iterator(q,bstree),false);
127.    }
128.    void erase(iterator m) {
129.      node_t p = m.node;
130.      node_t qr = p.right(),
131.        ql = p.left();
132.      if (qr==bstree.end() && ql==bstree.end())
133.        p = bstree.erase(p);
134.      else if (qr == bstree.end()) {
135.        btree<T> tmp;
136.        tmp.splice(tmp.begin(),ql);
```

```
137.        p = bstree.erase(p);
138.        bstree.splice(p,tmp.begin());
139.      } else if (ql == bstree.end()) {
140.        btree<T> tmp;
141.        tmp.splice(tmp.begin(),p.right());
142.        p = bstree.erase(p);
143.        bstree.splice(p,tmp.begin());
144.      } else {
145.        node_t r = min(qr);
146.        T minr = *r;
147.        erase(iterator(r,bstree));
148.        *p = minr;
149.      }
150.    }
151.    int erase(T x) {
152.      iterator q = find(x);
153.      int ret;
154.      if (q==end()) ret = 0;
155.      else {
156.        erase(q);
157.        ret = 1;
158.      }
159.      return ret;
160.    }
161.    void clear() { bstree.clear(); }
162.    iterator find(T x) {
163.      node_t m = bstree.begin();
164.      while (true) {
165.        if (m == bstree.end())
166.          return iterator(m,bstree);
167.        if (x<*m) m = m.left();
168.        else if (x>*m) m = m.right();
169.        else return iterator(m,bstree);
170.      }
171.    }
172.    iterator begin() {
173.      return iterator(min(bstree.begin()),bstree);
174.    }
175.    iterator end() {
176.      return iterator(bstree.end(),bstree);
177.    }
178.    int size() {
179.      return size_aux(bstree,bstree.begin()); }
180.    friend void
181.    set_union<T>(set<T> &A,set<T> &B,set<T> &C);
182.    friend void
183.    set_intersection<>(set<T> &A,set<T> &B,set<T> &C);
```

```
184.    friend void
185.    set_difference<>(set<T> &A,set<T> &B,set<T> &C);
186.    friend void f();
187.  };
188.
189.  template<class T> void
190.  set_union(set<T> &A,set<T> &B,set<T> &C) {
191.    C.clear();
192.    C.set_union_aux(A.bstree,A.bstree.begin());
193.    C.set_union_aux(B.bstree,B.bstree.begin());
194.  }
195.
196.  template<class T> void
197.  set_intersection(set<T> &A,set<T> &B,set<T> &C) {
198.    C.clear();
199.    C.set_intersection_aux(A.bstree,
200.                           A.bstree.begin(),B);
201.  }
202.
203.  // C = A - B
204.  template<class T> void
205.  set_difference(set<T> &A,set<T> &B,set<T> &C) {
206.    C.clear();
207.    C.set_difference_aux(A.bstree,
208.                         A.bstree.begin(),B);
209.  }
```

Código 4.15: *Implementación de* `set<>` *son ABB. [Archivo: setbst.h]*

En el código 4.15 vemos una posible implementación de conjuntos por ABB usando la clase **`btree<>`** discutida en el capítulo §3. En este caso pasamos directamente a la interfaz avanzada, es decir compatible con las STL.

- El tipo **`set<T>`** es un template que contiene un árbol binario **`btree<T> bstree`**.
- Los **`typedef tree_t`** y **`node_t`** son abreviaciones (privadas) para acceder convenientemente al árbol subyacente.
- La función **`min(m)`** retorna un iterator al nodo con el menor elemento del subárbol del nodo **`m`**. El nodo se encuentra bajando siempre por el hijo izquierdo (como se explicó en la sección §4.6.3).
- La clase iterator contiene no sólo el nodo en el árbol sino también un puntero al árbol mismo. Esto es necesario ya que algunas operaciones

de árboles (por ejemplo la comparación con **`end()`**) necesitan tener acceso al árbol donde está el nodo.

- **`begin()`** utiliza **`min(bstree.begin())`** para encontrar el nodo con el menor elemento en el árbol.
- La implementación incluye un constructor por defecto (el conjunto esta vacío), un constructor por copia que invoca al constructor por copia de árboles. El operador de asignación es sintetizado automáticamente por el compilador y funciona correctamente ya que utiliza el operador de asignación para **`btree<>`** el cual fue correctamente implementado (hace una *"deep copy"* del árbol.
- En este caso no mantenemos un contador de nodos, por lo cual la función **`size()`** calcula el número de nodo usando una función recursiva auxiliar **`size_aux()`**.
- Las operadores de incremento para iterators (tanto prefijo como postfijo) utilizan una función auxiliar **`next()`** que calcula la posición correspondiente al siguiente nodo en el árbol (en el sentido ordenado, como se describió en la sección §4.6.8). Esta función se complica un poco ya que nuestra clase árbol no cuenta con una función *"padre"*. Entonces, cuando el nodo no tiene hijo derecho y es necesario buscar el último *"padre derecho"*, el camino no se puede seguir desde abajo hacia arriba sino desde arriba hacia abajo, comenzando desde la raíz. De todas formas, el costo es $O(d)$ donde d es la altura del árbol, gracias a que la estructura de ABB nos permite ubicar el nodo siguiendo un camino.
- La función **`erase()`** es implementada eficientemente en términos de **`splice()`** cuando el nodo tiene sus dos hijos.
- Las funciones binarias utilizan funciones recursivas miembros privados de la clase **`set_union_aux(T,n)`**, **`set_intersection_aux(T,n,B)`** y **`set_difference_aux(T,n,B)`** que aplican la estrategia explicada en la sección §4.6.9.

4.6.11. Tiempos de ejecución

4.6.12. Balanceo del árbol

Es claro de los tiempos de ejecución que la eficiencia de esta implementación reside en mantener el árbol lo más balanceado posible. Existen al menos dos estrategias para mantener el árbol balanceado. Los *"árboles AVL"* mantienen el árbol balanceado haciendo rotaciones de los elementos en forma apropiada ante cada inserción o supresión. Por otra parte los *"treaps"*

Método	$T(n)$ (promedio)	$T(n)$ (peor caso)
`*p`, `end()`	$O(1)$	$O(1)$
`insert(x)`, `find(x)`, `erase(p)`, `erase(x)`, `begin()`, `p++`, `++p`	$O(\log n)$	$O(n)$
`set_union(A,B,C)`, `set_intersection(A,B,C)`, `set_difference(A,B,C)`	$O(n \log n)$	$O(n^2)$

Tabla 4.7: Tiempos de ejecución de los métodos del TAD set implementado con ABB.

mantiene el árbol balanceado introduciendo en cada nodo un campo elemento adicional llamado *"prioridad"*. La prioridad es generada en forma aleatoria en el momento de insertar el elemento y lo acompaña hasta el momento en que el elemento sea eliminado. Si bien el árbol sigue siendo un ABB con respecto al campo elemento, además se exige que el árbol sea *"parcialmente ordenado"* con respecto a la prioridad. Un árbol binario parcialmente ordenado es aquel tal que la prioridad de cada nodo es menor o igual que la de sus dos hijos. De esta forma las rotaciones se hacen en forma automática al insertar o eliminar elementos con sólo mirar las prioridades de un nodo y sus dos hijos, mientras que en el caso de los árboles AVL se debe considerar la altura de las ramas derecha e izquierda.

Capítulo 5

Ordenamiento

El proceso de ordenar elementos (*"sorting"*) en base a alguna relación de orden (ver sección §2.4.4) es un problema tan frecuente y con tantos usos que merece un capítulo aparte. Nosotros nos concentraremos en el ordenamiento interna, es decir cuando todo el contenedor reside en la memoria principal de la computadora. El tema del ordenamiento externo, donde el volumen de datos a ordenar es de tal magnitud que requiere el uso de memoria auxiliar, tiene algunas características propias y por simplicidad no será considerado aquí.

5.1. Introducción

Asumiremos normalmente que los elementos a ordenar pertenecen a algún tipo `key_t` con una relación de orden `<` y que están almacenados en un contenedor lineal (vector o lista). El problema de ordenar un tal contenedor es realizar una serie de intercambios en el contenedor de manera de que los elementos queden ordenados, es decir $k_0 \leq k_1 \leq ... \leq k_{n-1}$, donde k_j es la clave del j-ésimo elemento.

Si el ordenamiento se hace sin requerir ningún tipo de memoria adicional (que crezca con n), decimos que es *"in-place"*. Si no, se debe tener en cuenta también como crece la cantidad de memoria requerida.

5.1.1. Relaciones de orden débiles

Ya hemos discutido como se define una relación de orden en la sección §2.4.4. Igual que para definir conjuntos o correspondencias, la relación

de ordenamiento puede a veces ser elegida por conveniencia. Así, podemos querer ordenar un conjunto de números por su valor absoluto. En ese caso al ordenar los números $(1, 3, -2, -4)$ el contenedor ordenado resultaría en $(1, -2, 3, -4)$. En este caso la relación de orden la podríamos denotar por $\lhd$ y la definiríamos como

$$a \lhd u, \quad \text{si } |a| < |u|. \tag{5.1}$$

La relación binaria definida así no resulta ser una relación de orden en el sentido fuerte, como definida en la sección §2.4.4, ya que, por ejemplo, para el par $-2, 2$ no son ciertos $2 \lhd -2$, ni $-2 \lhd 2$ ni $-2 = 2$.

La segunda condición sobre las relaciones de orden puede relajarse un poco y obtener la definición de relaciones de orden débiles: ***Definición:*** *"$\lhd$" es una relación de orden débil en el conjunto C si,*

1. *$\lhd$ es transitiva, es decir, si $a \lhd b$ y $b \lhd c$, entonces $a \lhd c$.*
2. *Dados dos elementos cualquiera a, b de C*

$$(a \lhd b) \;\&\&\; (b \lhd a) = \texttt{false} \tag{5.2}$$

Esta última condición se llama *de antisimetría*. Usamos aquí la notación de C++ para los operadores lógicos, por ejemplo && indica "and" y los valores booleanos **`true`** y **`false`**. Es decir $a \lhd b$ y $b \lhd a$ no pueden ser verdaderas al mismo tiempo (son exclusivas).

Los siguientes, son ejemplos de relaciones de orden débiles pero no cumplen con la condición de antisimetría fuerte.

- Menor en valor absoluto para enteros, es decir

$$a \lhd b \;\text{ si } |a| < |b| \tag{5.3}$$

  La antisimetría fuerte no es verdadera para el par 2, -2, es decir, ni es cierto que $-2 \lhd 2$ ni $2 \lhd -2$ ni $2 = -2$.
- Pares de enteros por la primera componente, es decir $(a, b) \lhd (c, d)$ si $a < c$. En este caso la antisimetría fuerte se viola para $(2, 3)$ y $(2, 4)$, por ejemplo.
- Legajos por el nombre del empleado. Dos legajos para diferentes empleados que tienen el mismo nombre:
  (*nombre*=**`Perez, Juan`**,*DNI*=**`14231235`**) y
  (*nombre*=**`Perez, Juan`**,*DNI*=**`12765987`**) violan la condición.

Si dos elementos satisfacen que $!(a \lhd b)$ && $!(b \lhd a)$ entonces decimos que son *equivalentes* y lo denotamos por $a \equiv b$.

También se pueden definir otras relaciones derivadas como

$$\begin{array}{ll} \text{mayor:} & (a \rhd b) = (b \lhd a) \\ \text{equivalencia:} & (a \equiv b) = !(a \lhd b) \;\&\&\; !(b \lhd a) \\ \text{menor o equivalente:} & (a \trianglelefteq b) = !(b \lhd a) \\ \text{mayor o equivalente:} & (a \trianglerighteq b) = !(a \lhd b) \end{array} \tag{5.4}$$

También en algunos lenguajes (e.g. Perl) es usual definir una función `int cmp(T x,T y)` asociada a una dada relación de orden $\rhd$ que retorna 1, 0 o -1 dependiendo si $x \rhd y$, $x \equiv y$ o $x \lhd y$. En ese caso, el valor de `cmp(x,y)` se puede obtener de

$$\mathrm{cmp}(x, y) = (y \lhd x) - (x \lhd y) \tag{5.5}$$

donde en el miembro derecho se asume que las expresiones lógicas retornan 0 o 1 (como es usual en C). En las Tablas 5.1,5.2,5.3 se puede observar que todos los operadores de comparación se pueden poner en términos de cualquiera de los siguientes $\lhd$, $\trianglelefteq$, $\rhd$, $\trianglerighteq$, $\mathrm{cmp}(\,\cdot\,,\,\cdot\,)$. Notar que los valores retornados por $\mathrm{cmp}(\,\cdot\,,\,\cdot\,)$ son comparados directamente con la relación de orden para enteros $<$, $>$ etc... y no con la relación $\lhd$, $\rhd$, ... y amigos.

La expresión:	Usando: $\lhd$	Usando: $\trianglelefteq$
$a \lhd b$	$a \lhd b$	$!\,(b \trianglelefteq a)$
$a \trianglelefteq b$	$!\,(b \lhd a)$	$a \trianglelefteq b$
$a \rhd b$	$b \lhd a$	$!\,a \trianglelefteq b$
$a \trianglerighteq b$	$!\,a \lhd b$	$b \trianglerighteq a$
$\mathrm{cmp}(a, b)$	$(b \lhd a) - (a \lhd b)$	$(b \trianglelefteq a) - (a \trianglelefteq b)$
$a = b$	$!\,(a \lhd b \,\|\|\, b \lhd a)$	$a \trianglelefteq b \,\&\&\, b \trianglelefteq a$
$a \neq b$	$a \lhd b \,\|\|\, b \lhd a$	$(!\,a \trianglelefteq b) \,\|\|\, (!\,b \trianglelefteq a)$

Tabla 5.1: Equivalencia entre los diferentes operadores de comparación. En términos de $\trianglelefteq$

5.1.2. Signatura de las relaciones de orden. Predicados binarios.

En las STL las relaciones de orden se definen mediante *"predicados binarios"*, es decir, su signatura es de la forma

La expresión:	Usando: $\rhd$	Usando: $\trianglerighteq$
$a \lhd b$	$b \rhd a$	$!\ a \trianglerighteq b$
$a \trianglelefteq b$	$!\ (b \rhd a)$	$b \trianglerighteq a$
$a \rhd b$	$a \rhd b$	$!\ b \trianglerighteq a$
$a \trianglerighteq b$	$!\ b \rhd a$	$a \trianglerighteq b$
$\mathrm{cmp}(a, b)$	$(a \rhd b) - (b \rhd a)$	$(a \trianglerighteq b) - (b \trianglerighteq a)$
$a = b$	$!\ (a \rhd b \,\|\|\, b \rhd a)$	$a \trianglerighteq b \,\&\&\, b \trianglerighteq a)$
$a \neq b$	$a \rhd b \,\|\|\, b \rhd a$	$(!\ a \trianglerighteq b) \,\|\|\, (!\ b \trianglerighteq a)$

Tabla 5.2: Equivalencia entre los diferentes operadores de comparación. En términos de $\trianglerighteq$

La expresión:	Usando: $\mathrm{cmp}(\,\cdot\,,\,\cdot\,)$
$a \lhd b$	$\mathrm{cmp}(a, b) = -1$
$a \trianglelefteq b$	$\mathrm{cmp}(a, b) \leq 0$
$a \rhd b$	$\mathrm{cmp}(a, b) = 1$
$a \trianglerighteq b$	$\mathrm{cmp}(a, b) \geq 0$
$\mathrm{cmp}(a, b)$	$\mathrm{cmp}(a, b)$
$a = b$	$!\ \mathrm{cmp}(a, b)$
$a \neq b$	$\mathrm{cmp}(a, b)$

Tabla 5.3: Equivalencia entre los diferentes operadores de comparación. En términos de $\mathrm{cmp}(\,\cdot\,,\,\cdot\,)$

```
bool (*binary_pred)(T x, T Y);
```

Ejemplo 5.1: *Consigna:* Escribir una relación de orden para comparación lexicográfica de cadenas de caracteres. Hacerlo en forma dependiente e independiente de mayúsculas y minúsculas (*case-sensitive* y *case-insensitive*).

La clase **`string`** de las STL permite encapsular cadenas de caracteres con ciertas características mejoradas con respecto al manejo básico de C. Entre otras cosas, tiene sobrecargado el operador "<" con el orden lexicográfico (es decir, el orden alfabético). El orden lexicográfico consiste en comparar los primeros caracteres de ambas cadenas, si son diferentes entonces es menor aquel cuyo valor ASCII es menor, si son iguales se continua con los segundos caracteres y así siguiendo hasta que eventualmente uno de las dos cadenas se termina. En ese caso si una de ellas continúa es la mayor, si sus longitudes son iguales los strings son iguales. Esta relación de orden es fuerte.

```
1.  bool string_less_cs(const string &a,const string &b) {
2.    int na = a.size();
3.    int nb = b.size();
4.    int n = (na>nb ? nb : na);
5.    for (int j=0; j<n; j++) {
6.      if (a[j] < b[j]) return true;
7.      else if (b[j] < a[j]) return false;
8.    }
9.    return na<nb;
10. }
```

Código 5.1: *Función de comparación para cadenas de caracteres con orden lexicográfico. [Archivo: stringlcs.cpp]*

En el código 5.1 vemos la implementación del predicado binario correspondiente. Notar que este predicado binario termina finalmente comparando caracteres en las líneas 6–7. A esa altura se esta usando el operador de comparación sobre el tipo `char` que es un subtipo de los enteros.

Si ordenamos las cadenas `pepe juana PEPE Juana JUANA Pepe`, con esta función de comparación obtendremos

```
JUANA Juana PEPE Pepe juana pepe
```
(5.6)

Recordemos que en la serie ASCII las mayúsculas están antes que las minúsculas, las minúsculas `a-z` están en el rango 97-122 mientras que las mayúsculas `A-Z` están en el rango 65-90.

```
1.  bool string_less_cs3(const string &a,const string &b) {
2.    return a<b;
3.  }
```

Código 5.2: *Función de comparación para cadenas de caracteres con orden lexicográfico usando el operador "<" de* C++. *[Archivo: stringlcs3.cpp]*

```
1.    template<class T>
2.    bool less(T &x,T &y) {
3.      return x<y;
4.    }
```

Código 5.3: *Template de las STL que provee un predicado binario al operador intrínseco "<" del tipo* T. *[Archivo: lesst.h]*

Como ya mencionamos el orden lexicográfico está incluido en las STL, en forma del operador "<" sobrecargado, de manera que también podría usarse como predicado binario la función mostrada en el código 5.2. Finalmente, como para muchos tipos básicos (**int**, **double**, **string**) es muy común ordenar por el operador "<" del tipo, las STL proveen un template **less<T>** que devuelve el predicado binario correspondiente al operador intrínseco "<" del tipo **T**.

```
1.  char tolower(char c) {
2.    if (c>='A' && c<='Z') c += 'a'-'A';
3.    return c;
4.  }
5.
6.  bool string_less_ci(const string &a,
7.                      const string &b) {
8.    int na = a.size();
9.    int nb = b.size();
10.   int n = (na>nb ? nb : na);
11.   for (int j=0; j<n; j++) {
12.     char
13.       aa = tolower(a[j]),
14.       bb = tolower(b[j]);
15.     if (aa < bb) return true;
16.     else if (bb < aa) return false;
17.   }
18.   return na<nb;
19. }
```

Código 5.4: *Función de comparación para cadenas de caracteres con orden lexicográfico independiente de mayúsculas y minúsculas. [Archivo: stringlci.cpp]*

Si queremos ordenar en forma independiente de mayúsculas/minúsculas, entonces podemos definir una relación binaria que básicamente es

$$(a \lhd b) \;=(\texttt{tolower}(a) < \texttt{tolower}(b)) \tag{5.7}$$

donde la función **tolower()** convierte su argumento a minúsculas. La función **tolower()** está definida en la librería estándar de C (libc) para caracteres. Podemos entonces definir la función de comparación para orden lexicográfico independiente de mayúsculas/minúsculas como se muestra en el código 5.4. La función **string_less_ci()** es básicamente igual a la **string_less_cs()** sólo que antes de comparar caracteres los convierte con **tolower()** a minúsculas. De esta forma **pepe**, **Pepe** y **PEPE** resultan equivalentes.

5.1.3. Relaciones de orden inducidas por composición

La relación de orden para cadenas de caracteres (5.7) es un caso particular de de una forma bastante general de generar relaciones de orden que se obtienen componiendo otra relación de orden con una función escalar. Dada una relación de orden "$<$" y una función escalar $y = f(x)$ podemos definir una relación de orden "$\lhd$" mediante

$$(a \lhd b) = (f(a) < f(b)) \tag{5.8}$$

Por ejemplo, la relación de orden *menor en valor absoluto* definida en (5.3) para los enteros, puede interpretarse como la composición de la relación de orden usual "$<$" con la función valor absoluto. Si la relación de orden "$<$" y $f()$ es biunívoca, entonces "$\lhd$" resulta ser también fuerte. Si estas condiciones no se cumplen, la relación resultante puede ser débil. Volviendo al caso de la relación *menor en valor absoluto*, si bien la relación de partida "$<$" es fuerte, la composición con una función no biunívoca como el valor absoluto resulta en una relación de orden débil.

Notar que la función de mapeo puede ser de un conjunto universal U a otro $U' \neq U$. Por ejemplo, si queremos ordenar un vector de listas por su longitud, entonces el conjunto universal U es el conjunto de las listas, mientras que U' es el conjunto de los enteros.

5.1.4. Estabilidad

Ya vimos que cuando la relación de orden es débil puede haber elementos que son equivalentes entre sí sin ser iguales. Al ordenar el contenedor por una relación de orden débil los elementos equivalentes entre sí deben quedar contiguos en el contenedor ordenado pero el orden entre ellos no está definido en principio.

Un algoritmo de ordenamiento es *"estable"* si aquellos elementos equivalentes entre sí quedan en el orden en el que estaban originalmente. Por ejemplo, si ordenamos `(-3,2,-4,5,3,-2,4)` por valor absoluto, entonces podemos tener

$$\begin{array}{ll} \texttt{(2, -2, -3, 3, -4, 4, 5)}, & \text{estable} \\ \texttt{(-2, 2, -3, 3, 4, -1, 5)}, & \text{no estable} \end{array} \tag{5.9}$$

5.1.5. Primeras estimaciones de eficiencia

Uno de los aspectos más importantes de los algoritmos de ordenamiento es su tiempo de ejecución como función del número n de elementos a ordenar. Ya hemos visto en la sección §1.4.2 el algoritmo de ordenamiento por el método de la burbuja (*"bubble-sort"*), cuyo tiempo de ejecución resultó ser $O(n^2)$. Existen otros algoritmos de ordenamiento simples que también llevan a un número de operaciones $O(n^2)$. Por otra parte, cualquier algoritmo de ordenamiento debe por lo menos recorrer los elementos a ordenar, de manera que al menos debe tener un tiempo de ejecución $O(n)$, de manera que en general todos los algoritmos existentes están en el rango $O(n^\alpha)$ con $1 \leq \alpha \leq 2$ (los algoritmos $O(n \log n)$ pueden considerarse como $O(n^{1+\epsilon})$ con $\epsilon \to 0$).

Si los objetos que se están ordenando son grandes (largas cadenas de caracteres, listas, árboles...) entonces puede ser más importante considerar el número de intercambios que requiere el algoritmo, en vez del número de operaciones. Puede verse también que los algoritmos más simples hacen $O(n^2)$ intercambios (por ej. el método de la burbuja).

5.1.6. Algoritmos de ordenamiento en las STL

La signatura del algoritmo genérico de ordenamiento en las STL es

```
1. void sort(iterator first, iterator last);
2. void sort(iterator first, iterator last, binary_pred f);
```

el cual está definido en el header `algorithm`. Notemos que en realidad `sort()` es una función *"sobrecargada"*, existen en realidad dos funciones `sort()`. La primera toma un rango `[first,last)` de iteradores en el contenedor y ordena ese rango del contenedor, dejando los elementos antes de `first` y después de `last` inalterados. Si queremos ordenar todo el contenedor, entonces basta con pasar el rango `[begin(),end())`. Esta versión de `sort()` no se puede aplicar a cualquier contenedor sino que tiene que ser

un *"contenedor de acceso aleatorio"*, es decir un contenedor en el cual los iteradores soportan operaciones de aritmética entera, es decir si tenemos un iterador `p`, podemos hacer `p+j` (avanzar el iterador `j` posiciones, en tiempo $O(1)$). Los operadores de acceso aleatorio en las STL son `vector<>` y `deque<>`.

El ordenamiento se realiza mediante la relación de orden `operator<` del tipo `T` del cual están compuestos los elementos del contenedor. Si el tipo `T` es una clase definida por el usuario, este debe sobrecargar el `operator<`.

La segunda versión toma un argumento adicional que es la función de comparación. Esto puede ser útil cuando se quiere ordenar un contenedor por un orden diferente a `operator<` o bien `T` no tiene definido `operator<`. Las STL contiene en el header `functional` unos templates `less<T>`, `greater<T>` que devuelven funciones de comparación basados en `operator<` y `operator>` respectivamente. Por ejemplo, si queremos ordenar de mayor a menor un vector de enteros, basta con hacer

```
1.  vector<int> v;
2.  // Inserta elementos en v. . .
3.  sort(v.begin(), v.end(), greater<int>);
```

Usar `less<int>` es totalmente equivalente a usar la versión `sort(first,last)` es decir sin función de comparación.

Si queremos ordenar un vector de strings por orden lexicográfico independientemente de minúsculas/mayúsculas debemos hacer

```
1.  vector<string> v;
2.  // Inserta elementos en v. . .
3.  sort(v.begin(), v.end(), string_less_ci);
```

5.2. Métodos de ordenamiento lentos

Llamamos *"rápidos"* a los métodos de ordenamiento con tiempos de ejecución menores o iguales a $O(n \log n)$. Al resto lo llamaremos *"lentos"*. En esta sección estudiaremos tres algoritmos lentos, a saber burbuja, selección e inserción.

5.2.1. El método de la burbuja

```
1.  template<class T> void
2.  bubble_sort(typename std::vector<T>::iterator first,
3.              typename std::vector<T>::iterator last,
```

```
4.              bool (*comp)(T&,T&)) {
5.    int size = last-first;
6.    for (int j=0; j<size-1; j++) {
7.     for (int k=size-1; k>j; k--) {
8.      if (comp(*(first+k),*(first+k-1))) {
9.       T tmp = *(first+k-1);
10.      *(first+k-1) = *(first+k);
11.      *(first+k) = tmp;
12.     }
13.    }
14.   }
15.  }
16.
17.  template<class T> void
18.  bubble_sort(typename std::vector<T>::iterator first,
19.          typename std::vector<T>::iterator last) {
20.   bubble_sort(first,last,less<T>);
21.  }
```

Código 5.5: *Algoritmo de ordenamiento de la burbuja. [Archivo: bubsort.h]*

El método de la burbuja fue introducido en la sección §1.4.2. Nos limitaremos aquí a discutir la conversión al formato compatible con la STL. El código correspondiente puede observarse en el código 5.5.

- Para cada algoritmo de ordenamiento presentamos las dos funciones correspondientes, con y sin función de comparación.
- Ambas son templates sobre el tipo `T` de los elementos a ordenar.
- La que no tiene operador de comparación suele ser un *"wrapper"* que llama a la primera pasándole como función de comparación `less<T>`.
- Notar que como las funciones no reciben un contenedor, sino un rango de iteradores, no se puede referenciar directamente a los elementos en la forma `v[j]` sino a través del operador de dereferenciación `*p`. Así, donde normalmente pondríamos `v[first+j]` debemos usar `*(first+j)`.
- Recordar que las operaciones aritméticas con iteradores son válidas ya que los contenedores son de acceso aleatorio. En particular, las funciones presentadas son sólo válidas para `vector<>`, aunque se podrían modificar para incluir a `deque<>` en forma relativamente fácil.

- Notar la comparación en la línea 8 usando el predicado binario `comp()` en vez de `operator<`. Si queremos que nuestros algoritmos de ordenamiento puedan ordenar con predicados binarios arbitrarios, la comparación para elementos de tipo `T` se debería hacer siempre usando `comp()`.
- Para la discusión de los diferentes algoritmos haremos abstracción de la posición de comienzo `first`, como si fuera 0. Es decir consideraremos que se está ordenando el rango `[0,n)` donde `n=last-first`.

5.2.2. El método de inserción

```
1.  template<class T> void
2.  insertion_sort(typename
3.             std::vector<T>::iterator first,
4.             typename
5.             std::vector<T>::iterator last,
6.             bool (*comp)(T&,T&)) {
7.    int size = last-first;
8.    for (int j=1; j<size; j++) {
9.      T tmp = *(first+j);
10.     int k=j-1;
11.     while (comp(tmp,*(first+k))) {
12.       *(first+k+1) = *(first+k);
13.       if (--k < 0) break;
14.     }
15.     *(first+k+1) = tmp;
16.   }
17. }
18.
19.
20. template<class T> void
21. insertion_sort(typename
22.             std::vector<T>::iterator first,
23.             typename
24.             std::vector<T>::iterator last) {
25.   insertion_sort(first,last,less<T>);
26. }
```

Código 5.6: *Algoritmo de ordenamiento por inserción. [Archivo: inssorta.h]*

En este método (ver código 5.6) también hay un doble lazo. En el lazo sobre j el rango $[0, j)$ está ordenado e insertamos el elemento j en el

rango $[0, j)$, haciendo los desplazamientos necesarios. El lazo sobre k va recorriendo las posiciones desde j hasta 0 para ver donde debe insertarse el elemento que en ese momento está en la posición j.

5.2.3. El método de selección

```
1.  template<class T> void
2.  selection_sort(typename
3.             std::vector<T>::iterator first,
4.             typename
5.             std::vector<T>::iterator last,
6.             bool (*comp)(T&,T&)) {
7.    int size = last-first;
8.    for (int j=0; j<size-1; j++) {
9.      typename std::vector<T>::iterator
10.       min = first+j,
11.       q = min+1;
12.     while (q<last) {
13.       if (comp(*q,*min)) min = q;
14.       q++;
15.     }
16.     T tmp = *(first+j);
17.     *(first+j) = *min;
18.     *min = tmp;
19.   }
20. }
```

Código 5.7: *Algoritmo de ordenamiento por selección. [Archivo: selsort.h]*

En este método (ver código 5.7) también hay un doble lazo (esta es una característica de todos los algoritmos lentos). En el lazo j se elige el menor del rango $[j, N)$ y se intercambia con el elemento en la posición j.

5.2.4. Comparación de los métodos lentos

- En los tres métodos el rango **`[0,j)`** está siempre ordenado. Este rango va creciendo hasta que en la iteración **`j=n`** ocupa todo el vector y por lo tanto queda ordenado.
- Además en los métodos de burbuja y selección el rango **`[0,j)`** está en su posición definitiva, es decir los elementos en ese rango son los **`j`**

menores elementos del vector, de manera que en las iteraciones sucesivas no cambiarán de posición. Por el contrario, en el método de inserción el elemento `j`-ésimo viaja hasta la posición que le corresponde de manera que hasta en la última ejecución del lazo sobre `j` todas las posiciones del vector pueden cambiar.

- Tanto burbuja como selección son exactamente $O(n^2)$. Basta ver que en ambos casos el lazo interno se ejecuta incondicionalmente `j` veces.
- En el caso de inserción el lazo interno puede ejecutarse entre 0 y `j` veces, dependiendo de donde se encuentre el punto de inserción para el elemento `j`. El mejor caso es cuando el vector está ordenado, y el punto de inserción en ese caso es directamente la posición `k=j-1` para todos los `j`. En este caso el lazo interno se ejecuta una sola vez para cada valor de `j` con lo cual el costo total es $O(n)$.
- El peor caso para inserción es cuando el vector está ordenado en forma inversa (de mayor a menor). En ese caso, para cada `j` el elemento `j` debe viajar hasta la primera posición y por lo tanto el lazo interno se ejecuta j veces. El costo total es en este caso $\sim n^2/2$.
- En el caso promedio (es decir cuando los valores del vector están aleatoriamente distribuidos) la posición de inserción `k` está en el medio del rango `[0,j)` de manera que el lazo interno se ejecuta $\sim j/2$ veces y el costo total es $\sim n^2/4$.
- En cuanto al número de intercambios, ambos burbuja e inserción hacen en el peor caso (cuando el vector está ordenado en forma inversa) `j` intercambios en el lazo interior y $\sim n^2/2$ en total.
- En el mejor caso (cuando el vector está ordenado) ambos hacen 0 intercambios. En el caso promedio ambos hacen $\sim n^2/4$ intercambios (Hay un 50 % de probabilidad de que el intercambio se haga o no).
- Selección hace siempre sólo n intercambios. Notar que el intercambio se hace fuera del *lazo* interno.

Debido a estas consideraciones resulta que *inserción* puede ser de interés cuando el vector está parcialmente ordenado, es decir hay relativamente pocos elementos fuera de posición.

Por otra parte *selección* puede ser una opción interesante cuando se debe minimizar el número de intercambios. Sin embargo, veremos en la siguiente sección que, ordenando *"indirectamente"* los elementos, cualquier se puede lograr que cualquier método de ordenación haga sólo n intercambios.

5.2.5. Estabilidad

Una forma de verificar si un algoritmo de ordenamiento es estable o no es controlar que la estabilidad no se viole en ningún intercambio. Por ejemplo en el caso del método de la burbuja la estabilidad se mantiene ya que el intercambio se realiza sólo en las líneas 9–11. Pero como el intercambio sólo se realiza si `*(first+k)` es *estrictamente menor* que `*(first+k-1)` y los dos elementos están en posiciones consecutivas *el intercambio nunca viola la estabilidad*.

En el caso del método de inserción pasa algo similar. En las líneas 11–14 el elemento `*(first+j)` es intercambiado con todo el rango `[first+k,first+j)` que son elementos *estrictamente mayores* que `*(first+j)`.

En cambio, en el caso del método de selección, después de buscar la posición del mínimo en el lazo de las líneas 12–15, el intercambio se realiza en las líneas 16–18. Pero al realizar este intercambio, el elemento `*(first+j)`, que va a ir a la posición `min`, puede estar cambiando de posición relativa con elementos equivalentes en el rango `(first+j,min)`, violando la estabilidad.

5.3. Ordenamiento indirecto

Si el intercambio de elementos es muy costoso (pensemos en largas listas, por ejemplo) podemos reducir notablemente el número de intercambios usando *"ordenamiento indirecto"*, el cual se basa en ordenar un vector de cursores o punteros a los objetos reales. De esta forma el costo del intercambio es en realidad el de intercambio de los cursores o punteros, lo cual es mucho más bajo que el intercambio de los objetos mismos.

```
1.  template<class T>
2.  void apply_perm(typename std::vector<T>::iterator first,
3.              typename std::vector<T>::iterator last,
4.              std::vector<int> &indx) {
5.    int size = last-first;
6.    assert(indx.size()==size);
7.    int sorted = 0;
8.    T tmp;
9.    while (sorted<size) {
10.     if(indx[sorted]!=sorted) {
11.       int k = sorted;
12.       tmp = *(first+k);
13.       while (indx[k]!=sorted) {
```

```
14.         int kk = indx[k];
15.         *(first+k)=*(first+kk);
16.         indx[k] = k;
17.         k = kk;
18.       }
19.       *(first+k) = tmp;
20.       indx[k] = k;
21.     }
22.     sorted++;
23.   }
24. }
25.
26. template<class T>
27. void ibubble_sort(typename std::vector<T>::iterator first,
28.                    typename std::vector<T>::iterator last,
29.                    bool (*comp)(T&,T&)) {
30.   int size = last-first;
31.   std::vector<int> indx(size);
32.   for (int j=0; j<size; j++) indx[j] = j;
33.
34.   for (int j=0; j<size-1; j++) {
35.     for (int k=size-1; k>j; k--) {
36.       if (comp(*(first+indx[k]),*(first+indx[k-1]))) {
37.         int tmp = indx[k-1];
38.         indx[k-1] = indx[k];
39.         indx[k] = tmp;
40.       }
41.     }
42.   }
43.   apply_perm<T>(first,last,indx);
44. }
45.
46. template<class T>
47. void ibubble_sort(typename std::vector<T>::iterator first,
48.                    typename std::vector<T>::iterator last) {
49.  ibubble_sort(first,last,less<T>);
50. }
```

Código 5.8: *Método de la burbuja con ordenamiento indirecto. [Archivo: ibub.h]*

En el código 5.8 vemos el método de la burbuja combinado con ordenamiento indirecto. Igual que para el ordenamiento directo existen versiones con y sin función de comparación (**ibubble_sort(first,last,comp)**

y **ibubble_sort(first,last)**). Se utiliza un vector auxiliar de enteros **indx**. Inicialmente este vector contiene los enteros de 0 a **size-1**. El método de la burbuja procede en las líneas 34–42, pero en vez de intercambiar los elementos en el contenedor real, se mueven los cursores en **indx**. Por ejemplo, después de terminar la ejecución del lazo para **j=0**, en vez de quedar el menor en ***first**, en realidad queda la posición correspondiente al menor en **indx[0]**. En todo momento **indx[]** es una permutación de los enteros en **[0,size)**. Cuando termina el algoritmo de la burbuja, al llegar a la línea 43, **indx[]** contiene la permutación que hace ordena **[first,last)**. Por ejemplo el menor de todos los elementos está en la posición **first+indx[0]**, el segundo menor en **first+indx[1]**, etc... La última operación de **ibubble_sort()** es llamar a la función **apply_perm(first,last,indx)** que aplica la permutación obtenida a los elementos en **[first,last)**. Puede verse que esta función realiza n intercambios o menos. Esta función es genérica y puede ser combinada con cualquier otro algoritmo de ordenamiento indirecto. En la figura 5.1 vemos como queda el vector **v[]** con los elementos desordenados, y la permutación **indx[]**.

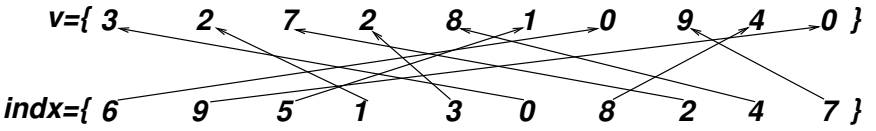


Figura 5.1: Ordenamiento indirecto.

El código en las líneas 34–42, es básicamente igual al de la versión directa **bubble_sort()** sólo que donde en ésta se referencian elementos como ***(first+j)** en la versión indirecta se referencian como ***(first+indx[j])**. Por otra parte, donde la versión indirecta intercambia los elementos ***(first+k-1)** y ***(first+k)**, la versión indirecta intercambia los índices en **indx[k-1]** y **indx[k]**.

En el caso de usar ordenamiento indirecto debe tenerse en cuenta que se requiere memoria adicional para almacenar **indx[]**.

5.3.1. Minimizar la llamada a funciones

Si la función de comparación se obtiene por composición, y la función de mapeo es muy costosa. Entonces tal vez convenga generar primero un vector auxiliar con los valores de la función de mapeo, ordenarlo y después

aplicar la permutación resultante a los elementos reales. De esta forma el número de llamadas a la función de mapeo pasa de ser $O(n^2)$ a $O(n)$. Por ejemplo, supongamos que tenemos un conjunto de árboles y queremos ordenarlos por la suma de sus valores nodales. Podemos escribir una función `int sum_node_val(tree<int> &A);` y escribir una función de comparación

```
1.  bool comp_tree(tree<int> &A,tree<int> &A) {
2.    return sum_node_val(A) < sum_node_val(B);
3.  }
```

Si aplicamos por ejemplo `bubble_sort`, entonces el número de llamadas a la función será $O(n^2)$. Si los árboles tienen muchos nodos, entonces puede ser preferible generar un vector de enteros con los valores de las sumas y ordenarlo. Luego se aplicaría la permutación correspondiente al vector de árboles.

En este caso debe tenerse en cuenta que se requiere memoria adicional para almacenar el vector de valores de la función de mapeo, además del vector de índices para la permutación.

5.4. El método de ordenamiento rápido, quick-sort

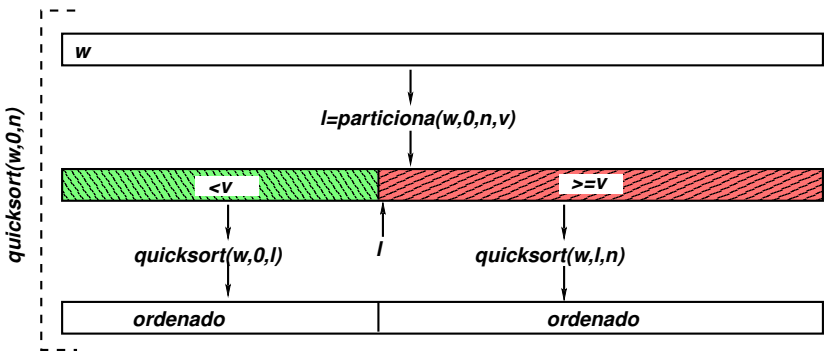


Figura 5.2: Esquema *"dividir para vencer"* para el algoritmo de ordenamiento rápido, *quick-sort*.

Este es probablemente uno de los algoritmos de ordenamiento más usados y se basa en la estrategia de *"dividir para vencer"*. Se escoge un elemento del vector v llamado *"pivote"* y se *"particiona"* el vector de manera de dejar los elementos $\geq v$ a la derecha (rango $[l, n)$, donde l es la posición del primer elemento de la partición derecha) y los $< v$ a la izquierda (rango

$[0, l)$). Está claro que a partir de entonces, los elementos en cada una de las particiones quedarán en sus respectivos rangos, ya que todos los elementos en la partición derecha son estrictamente mayores que los de la izquierda. Podemos entonces aplicar **quick-sort** recursivamente a cada una de las particiones.

```
1. void quicksort(w,j1,j2) {
2.   // Ordena el rango [j1,j2) de 'w'
3.   if (n==1) return;
4.   // elegir pivote v . . .
5.   l = partition(w,j1,j2,v);
6.   quicksort(w,j1,l);
7.   quicksort(w,l,j2);
8. }
```

Código 5.9: *Seudocódigo para el algoritmo de ordenamiento rápido. [Archivo: qsortsc.cpp]*

Si garantizamos que cada una de las particiones tiene al menos un elemento, entonces en cada nivel de recursividad los rangos a los cuales se le aplica quick-sort son estrictamente menores. La recursión termina cuando el vector a ordenar tiene un sólo elemento.

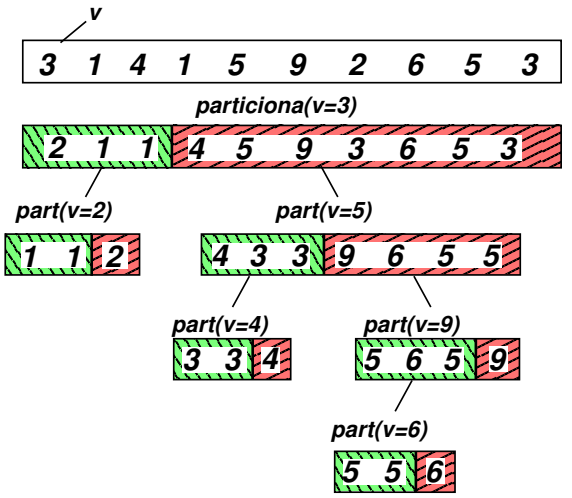


Figura 5.3: Ejemplo de aplicación de quick-sort

En la figura 5.3 vemos un ejemplo de aplicación de quick-sort a un vector de enteros. Para el ejemplo hemos elegido como estrategia para elegir el pivote tomar el mayor de los dos primeros elementos distintos. Si la secuencia a ordenar no tiene elementos distintos, entonces la recursión termina. En el primer nivel los dos primeros elementos distintos (de izquierda a derecha) son el 3 y el 1, por lo que el pivote será $v = 3$. Esto induce la partición que se observa en la línea siguiente, donde tenemos los elementos menores que 3 en el rango $[0, 3)$ y los mayores o iguales que 3 en el rango $[3, 10)$. Todavía no hemos explicado cómo se hace el algoritmo de partición, pero todavía esto no es necesario para entender como funciona quick-sort. Ahora aplicamos quick-sort a cada uno de los dos rangos. En el primer caso, los dos primeros elementos distintos son 2 y 1 por lo que el pivote es 2. Al particionar con este pivote quedan dos rangos en los cuales no hay elementos distintos, por lo que la recursión termina allí. Para la partición de la derecha, en cambio, todavía debe aplicarse quick-sort dos niveles más. Notar que la estrategia propuesta de elección del pivote garantiza que cada una de las particiones tendrá al menos un elemento ya que si los dos primeros elementos distintos de la secuencia son a y b, y $a < b$, por lo cual $v = b$, entonces al menos hay un elemento en la partición derecha (el elemento b) y otro en la partición izquierda (el elemento a).

5.4.1. Tiempo de ejecución. Casos extremos

El tiempo de ejecución del algoritmo aplicado a un rango $[j_1, j_2)$ de longitud $n = j_2 - j_1$ es

$$T(n) = T_{\mathrm{part-piv}}(n) + T(n_1) + T(n_2) \tag{5.10}$$

donde $n_1 = l - j_1$ y $n_2 = j_2 - l$ son las longitudes de cada una de las particiones. $T_{\mathrm{part-piv}}(n)$ es el costo de particionar la secuencia y elegir el pivote.

El mejor caso es cuando podemos elegir el pivote de tal manera que las particiones resultan ser bien balanceadas, es decir $n_1 \approx n_2 \approx n/2$. Si además asumimos que el algoritmo de particionamiento y elección del pivote son $O(n)$, es decir

$$T_{\mathrm{part-piv}} = cn \tag{5.11}$$

(más adelante se explicará un algoritmo de particionamiento que satisface

esto) Entonces

$$\begin{aligned} T(n) &= T_{\text{part}-\text{piv}}(n) + T(n_1) + T(n_2) \\ &= cn + T(n/2) + T(n/2) \end{aligned} \tag{5.12}$$

Llamando $T(1) = d$ y aplicando sucesivamente

$$\begin{aligned} T(2) &= c + 2T(1) = c + 2d \\ T(4) &= 4c + 2T(2) = 3 \cdot 4c + 4d \\ T(8) &= 8c + 2T(4) = 4 \cdot 8c + 8d \\ T(16) &= 16c + 2T(8) = 5 \cdot 16c + 16d \\ \vdots &= \quad \vdots \\ T(2^p) &= (p+1)n(c+d) \end{aligned} \tag{5.13}$$

pero como $n = 2^p$ entonces $p = \log_2 n$ y por lo tanto

$$T(n) = O(n \log n). \tag{5.14}$$

Por otro lado el peor caso es cuando la partición es muy desbalanceada, es decir $n_1 = 1$ y $n_2 = n - 1$ o viceversa. En este caso tenemos

$$\begin{aligned} T(n) &= cn + T(1) + T(n-1) \\ &= cn + d + T(n-1) \end{aligned} \tag{5.15}$$

y aplicando sucesivamente,

$$\begin{aligned} T(2) &= 2c + 2d \\ T(3) &= 3c + d + (2c + 2d) = 5c + 3d \\ T(4) &= 4c + d + (5c + 3d) = 9c + 4d \\ T(5) &= 5c + d + (9c + 4d) = 14c + 5d \\ \vdots &= \quad \vdots \\ T(n) &= \left(\frac{n(n+1)}{2} - 2 \right) c + nd = O(n^2) \end{aligned} \tag{5.16}$$

El peor caso ocurre, por ejemplo, si el vector está inicialmente ordenado y usando como estrategia para el pivote el mayor de los dos primeros distintos. Si tenemos en **v**, por ejemplo los enteros 1 a 100 ordenados, que podemos denotar como un rango $[1, 100]$, entonces el pivote sería inicialmente 2. La partición izquierda tendría sólo al 1 y la derecha sería el rango $[2, 99]$. Al particionar $[2, 99]$ tendríamos el pivote 3 y las particiones serían 2 y $[3, 99]$ (ver figura 5.4). Puede verificarse que lo mismo ocurriría si el vector está ordenado al revés.

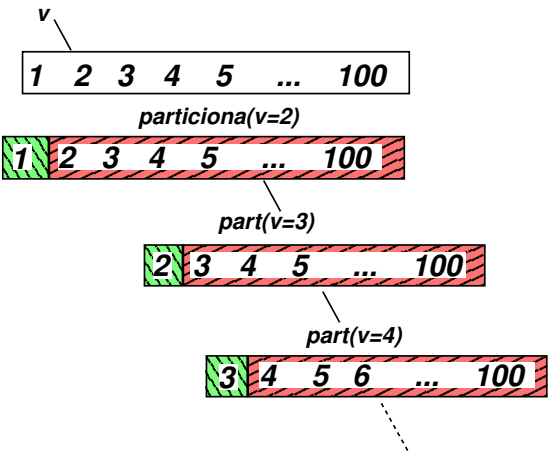


Figura 5.4: Particionamiento desbalanceado cuando el vector ya está ordenado.

5.4.2. Elección del pivote

En el caso promedio, el balance de la partición dependerá de la estrategia de elección del pivote y de la distribución estadística de los elementos del vector. Compararemos a continuación las estrategias que corresponden a tomar la *"mediana"* de los k primeros distintos. Recordemos que para k impar la mediana de k elementos es el elemento que queda en la posición del medio del vector (la posición $(k-1)/2$ en base 0) después de haberlos ordenado. Para k par tomamos el elemento en la posición $k/2$ (base 0) de los primeros k distintos, después de ordenarlos.

No confundir la mediana con el *"promedio"* o *"media"* del vector que consiste en sumar los valores y dividirlos por el número de elementos. Si bien es muy sencillo calcular el promedio en $O(n)$ operaciones, la mediana requiere en principio ordenar el vector, por lo que claramente no se puede usar la mediana como estrategia para elegir el pivote. En lo que resta, cuando hablamos de elegir la mediana de k valores del vector, asumimos que k es un valor constante y pequeño, más concretamente que no crece con n.

En cuanto al promedio, tampoco es una buena opción para el pivote. Consideremos un vector con 101 elementos 1,2,...,99,100,100000. La mediana de este vector es 51 y, por supuesto da una partición perfecta, mientras que el promedio es 1040.1, lo cual daría una pésima partición con 100 elementos en la partición izquierda y 1 elemento en la derecha. Notemos que esta mala partición es causada por la no uniformidad en la distribución

de los elementos, para distribuciones más uniformes de los elementos tal vez, es posible que el promedio sea un elección razonable. De todas formas el promedio es un concepto que es sólo aplicable a tipos para los cuales las operaciones algebraicas tienen sentido. No es claro como podríamos calcular el promedio de una serie de cadenas de caracteres.

Volviendo En el caso $k = 2$ tomamos de los dos primeros elementos distintos el que está en la posición 1, es decir el mayor de los dos, de manera que $k = 2$ equivale a la estrategia propuesta en las secciones anteriores. El caso del balance perfecto (5.12) se obtiene tomando como pivote la mediana de todo el vector, es decir $k = n$.

Para una dada estrategia de elección del pivote podemos preguntarnos, cual es la probabilidad $P(n, n_1)$ de que el pivote genere subparticiones de longitud n_1 y $n - n_1$, con $1 \leq n_1 < n$. Asumiremos que los elementos del vector están distribuidos aleatoriamente. Si, por ejemplo, elegimos como pivote el primer elemento del vector (o sea la mediana de los primeros $k = 1$ distintos), entonces al ordenar el vector este elemento puede terminar en cualquier posición del vector, ordenado de manera que $P(n, n_1) = 1/(n-1)$ para cualquier $n_1 = 1, .., n - 1$. Por supuesto, recordemos que esta no es una elección aceptable para el pivote en la práctica ya que no garantizaría que ambas particiones sean no nulas. Si el primer elemento resultara ser el menor de todos, entonces la partición izquierda resultaría ser nula. En la figura 5.6 vemos esta distribución de probabilidad. Para que la curva sea independiente de n hemos graficado $nP(n, n_1)$ en función de n_1/n.

$$
\begin{array}{cccc}
a & b & x & x \\
a & x & b & x \\
a & x & x & b \\
b & a & x & x \\
x & a & b & x \\
x & a & x & b \\
b & x & a & x \\
x & b & a & x \\
x & x & a & b \\
b & x & x & a \\
x & b & x & a \\
x & x & b & a
\end{array}
$$

Figura 5.5: Posibilidades al ordenar un vector 5 elementos. a y b son los primeros dos elementos antes de ordenar.

Ahora consideremos la estrategia propuesta en las secciones anteriores, es decir el mayor de los dos primeros elementos distintos. Asumamos por

simplicidad que todos los elementos son distintos. Sean a y b los dos primeros elementos del vector, entonces después de ordenar los elementos estos elementos pueden terminar en cualquiera de las posiciones j, k del vector con la misma probabilidad. Si la longitud del vector es $n = 4$, entonces hay $n(n-1) = 12$ posibilidades esencialmente distintas como se muestra en la figura 5.5.

Notemos que las primeras tres corresponden a que a termine en la posición 0 (base 0) y b en cada una de las tres posiciones restantes. Las siguientes 3 corresponden a a en la posición 1 (base 0), y así siguiendo. En el primero de los doce casos el pivote sería b y terminaría en la posición 1. Revisando todos los posibles casos tenemos que en 2 casos (líneas 1 y 4) el pivote termina en la posición 1, en 4 casos (líneas 2, 5, 7 y 8) termina en la posición 2 y en 6 casos (líneas 3, 6, 9, 10, 11 y 12) termina en la posición 3. Notemos que en este caso es más probable que el pivote termine en las posiciones más a la derecha que en las que están más a la izquierda. Por supuesto esto se debe a que estamos tomando el mayor de los dos. Una forma de contar estas posibilidades es considerar que para que el mayor esté en la posición j debe ocurrir que a esté en la posición j y b en las posiciones 0 a $j-1$, o que b quede en la posición j y a en las posiciones 0 a $j-1$, o sea un total de $2j$ posibilidades.

Ahora veamos que ocurre si tomamos como estrategia para la elección del pivote la mediana de los primeros $k = 3$ distintos. En este caso, si denotamos los tres primeros distintos como a, b y c, entonces existen $n(n-1)(n-2)$ casos distintos: a en cualquiera de las n posiciones, b en cualquiera de las $n-1$ restantes y c en cualquiera de las $n-2$ restantes. Para que el pivote quede en la posición j debería ocurrir que, por ejemplo, a quede en la posición j, b en una posición $[0, j)$ y c en una posición (j, n), o sea $j(n-j-1)$ posibilidades. Las restantes posibilidades se obtienen por permutación de los elementos a, b y c, en total

$$
\begin{array}{llll}
a \text{ en la posición } j, & b \text{ en } [0,j) & c \text{ en } (j,n) \\
a \text{ en la posición } j, & c \text{ en } [0,j) & b \text{ en } (j,n) \\
b \text{ en la posición } j, & a \text{ en } [0,j) & c \text{ en } (j,n) \\
b \text{ en la posición } j, & c \text{ en } [0,j) & a \text{ en } (j,n) \\
c \text{ en la posición } j, & a \text{ en } [0,j) & b \text{ en } (j,n) \\
c \text{ en la posición } j, & b \text{ en } [0,j) & a \text{ en } (j,n)
\end{array}
$$

O sea que en total hay $6j(n-j-1)$ posibilidades. Esto está graficado en la figura 5.6 como $k = 3$. Notemos que al tomar $k = 3$ hay mayor probabi-

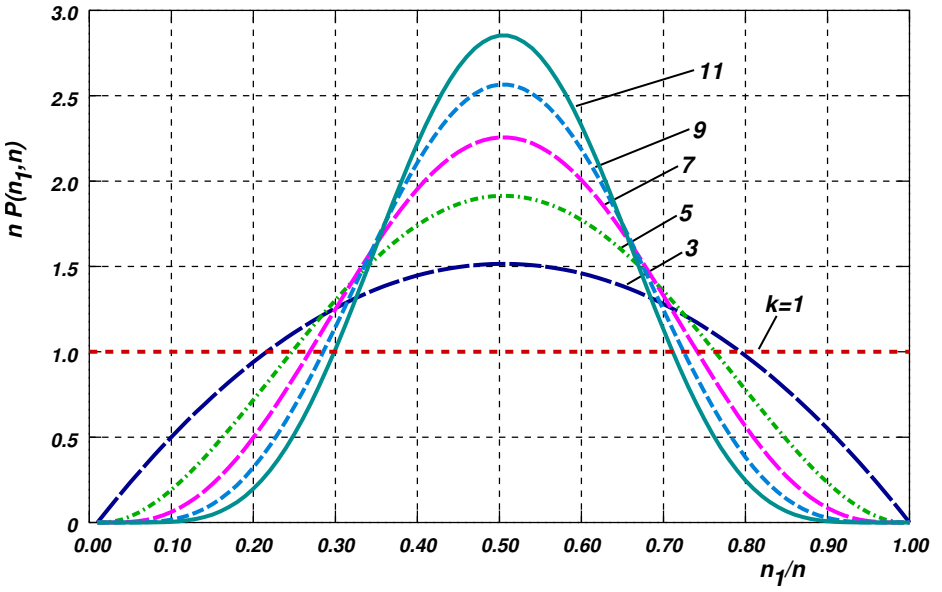


Figura 5.6: Probabilidad de que el pivote aparezca en una dada posición n_1, para diferentes valores de k.

lidad de que el pivote particione en forma más balanceada. En la figura se muestra la distribución de probabilidades para los k impares y se observa que a medida que crece k ha más certeza de que el pivote va a quedar en una posición cercana al centro de la partición.

5.4.3. Tiempo de ejecución. Caso promedio.

Hemos obtenido el tiempo de ejecución en el mejor (5.12) y el peor (5.15) caso como casos particulares de (5.10). Conociendo la distribución de probabilidades para las diferentes particiones podemos obtener una expresión general para el caso promedio sumando sobre todas las posibles posiciones finales del pivote, desde $j = 0$ hasta $n - 1$ multiplicado por su correspondiente probabilidad

$$T(n) = cn + \sum_{n_1=1}^{n-1} P(n, n_1)(T(n_1) + T(n - n_1)). \tag{5.17}$$

Notar que en la suma para un n dado sólo aparecen los $T(n_1)$ para $n_1 < n$, con lo cual puede fácilmente implementarse en un pequeño programa que

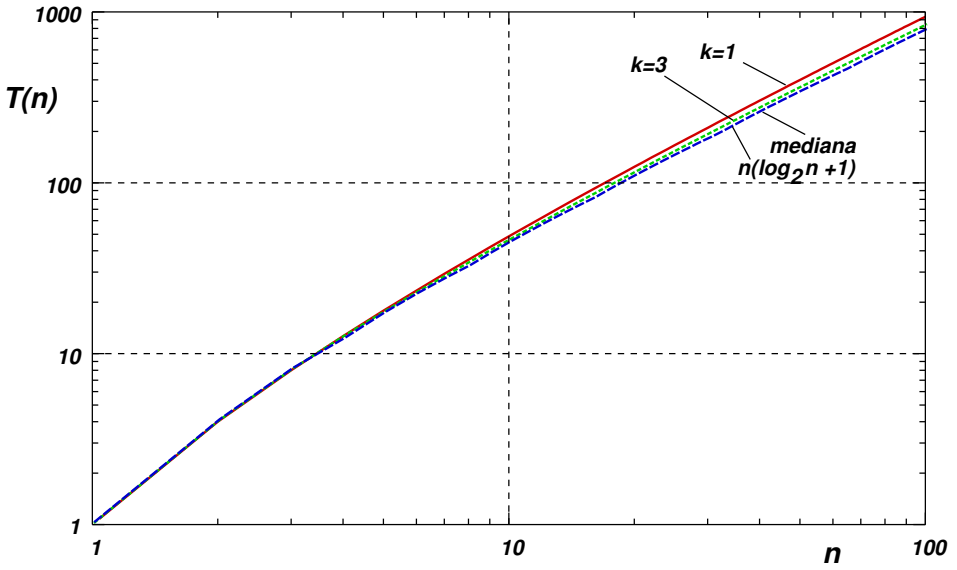


Figura 5.7:

va calculando los valores de $T(n)$ para $n = 1, 2, 3, ...$ En la figura 5.7 se observan los tiempos de ejecución así calculados para los casos $k = 1$ y $k = 3$ y también para el caso de elegir como pivote la mediana (equivale a $k = n$). Para los cálculos, se ha tomado $c = 1$ (en realidad puede verse que la velocidad de crecimiento no depende del valor de c, mientras sea $c > 0$). También se graficó la función $n(\log_2 n + 1)$ que corresponde al mejor caso (5.12), al menos para $n = 2^p$. Observamos que el tiempo de ejecución no presenta una gran dependencia con k y en todos los casos está muy cerca del mejor caso, $O(n \log_2 n)$. Esto demuestra (al menos *"experimentalmente"*) que en el caso promedio, e incluso para estrategias muy simples del pivote, el tiempo de ejecución en el caso promedio es

$$T_{\text{prom}}(n) = O(n \log_2 n) \tag{5.18}$$

Una demostración rigurosa de esto puede encontrarse en Aho et al. [1987].

5.4.4. Dispersión de los tiempos de ejecución

Sin embargo, la estrategia en la elección del pivote (en este caso el valor de k) sí tiene una incidencia notable en la *"dispersión"* de los valores de tiempos de ejecución, al menos para valores de k pequeños. Es decir, si bien

para todos los valores de k el tiempo de ejecución promedio es $O(n \log n)$ para valores de k altos, es más probable que la partición sea siempre balanceada, debido a que las campanas de probabilidad (ver figura 5.6) son cada vez más concentradas cerca de $n_1 = n/2$. De hecho, cuando tomamos como pivote la mediana ($k = n$) el balanceo es siempre perfecto, a medida que reducimos el valor de k es más probable que para ciertas distribuciones de los elementos del vector los tiempos de ejecución sean más grande que promedio.

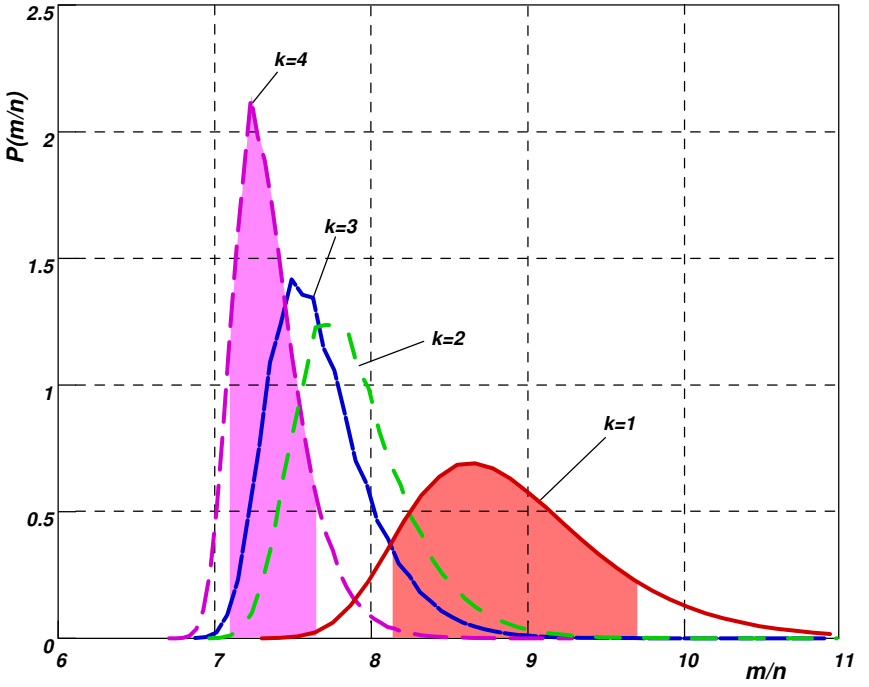


Figura 5.8: Dispersión de valores en el número de operaciones m al ordenar con quick-sort un vector generado aleatoriamente.

Para verificar esto realizamos un experimento en el que generamos un cierto número muy grande N (en este caso usamos concretamente $N = 10^6$) de vectores de longitud n y contamos el número de operaciones m al aplicar quick-sort a ese vector. En realidad lo que contamos es la suma m del largo de las particiones que se van generando. En el caso de la figura 5.3, esta suma daría $m = 30$ (no se cuenta la longitud del vector inicial, es decir sólo se suman las longitudes de las particiones rojas y verdes en la figura). En la figura 5.8 graficamos la probabilidad $P(\xi)$ de ocurrencia de un dado

valor de $\xi = m/n$. En el caso de la figura 5.3 tendríamos $m = 30$ y $n = 10$, por lo tanto $\xi = 30/10 = 3$. La probabilidad $P(\xi)$ se obtiene dividiendo el eje de las abscisas ξ en una serie de intervalos $[\xi_i, \xi_{1+1}]$, y contando para cada intervalo el número de vectores N_i (de los N simulados) cuyo valor de ξ cae en el intervalo. La probabilidad correspondiente al intervalo es entonces

$$P(\xi) \approx \frac{N_i}{N}. \tag{5.19}$$

Haciendo tender el número de simulaciones en el experimento N a infinito el miembro derecho tiende a la probabilidad $P(x)$.

Basta con observar la gráfica para ver que a medida que k se incrementa la distribución de los valores es más concentrada, resultando en una campana más delgada y puntiaguda. Esto se puede cuantificar buscando cuáles son los valores de ξ que delimitan el 80 % de los valores centrales. Por ejemplo, se observa que para el valor más bajo $k = 1$ el 80 % de los valores está entre ξ =8.1 y 9.8 (ancho de la campana 1.7), mientras que para k =4 el 80 % de los valores está entre ξ =7.1 y 7.65 (ancho de la campana 0.55). En la figura se muestran sombreadas las áreas que representan el 80 % central de los valores para $k = 1$ y $k = 4$.

5.4.5. Elección aleatoria del pivote

Vimos que tomando la mediana de los primeros k valores podemos hacer que cada vez sea menos probable obtener tiempos de ejecución demasiado altos, o mejor dicho, mucho más altos que el promedio *si el vector está inicialmente desordenado*. Pero si el vector está inicialmente ordenado, entonces en todos los niveles la partición va a ser desbalanceada independientemente del valor de k. El problema es que la situación en que el vector tiene cierto grado de ordenamiento previo es relativamente común, pensemos por ejemplo en un vector que está inicialmente ordenado y se le hacen un pequeño número de operaciones, como insertar o eliminar elementos, o permutar algunos de ellos. Cuando queremos volver a ordenar el vector estaríamos en un caso bastante cercano al peor.

Para decirlo en forma más rigurosa, cuando calculamos el tiempo promedio asumimos que todas las posibles permutaciones de los elementos del vector son igualmente probables, lo cual es cierto si, por ejemplo, generamos el vector tomando los elementos con un generador aleatorio. En la práctica es común que secuencias donde los elementos estén parcialmente ordenados sean más frecuentes y es malo que éste sea justo el peor caso. Una

solución puede ser *"desordenar"* inicialmente el vector, aplicando un algoritmo como el `random_shuffle()` de STL [SGI, 1999]. La implementación de `random_shuffle()` es $O(n)$, con lo cual no cambia la tasa de crecimiento. Sin embargo podría poner a quick-sort en desventaja con otros algoritmos que también son $O(n \log n)$ como `heap_sort()`. Otro inconveniente es que el `random_shuffle()` inicial haría prácticamente imposible una implementación estable del ordenamiento.

5.4.6. El algoritmo de partición

Una implementación eficiente del algoritmo de partición es clave para la eficiencia de quick-sort. Mantenemos dos cursores `l` y `r` de tal manera que todos los elementos a la izquierda de `l` son estrictamente menores que el pivote `v` y los que están a la derecha de `r` son mayores o iguales que `v`. Los elementos a la izquierda de `l` son la partición izquierda que va a ir creciendo durante el algoritmo de partición, y lo mismo para `r`, *mutatis mutandis*. Inicialmente podemos poner `l=first` y `r=last-1`, ya que corresponde a que ambas particiones sean inicialmente nulas. A partir de allí vamos aplicando un proceso iterativo que se muestra en el seudocódigo 5.10

```
1.  int partition(w,first,last,v) {
2.    // Particiona el rango [j1,j2) de 'w'
3.    // con respecto al pivote 'v'
4.    if (n==1) return (w[first]<v ? first : last);
5.    int middle = (first+last)/2;
6.    l1 = partition(w,first,middle,v);
7.    l2 = partition(w,middle,last,v);
8.    // Intercambia [l1,middle) con [middle,l2)
9.    swap(l1,middle,l2);
10. }
```

Código 5.10: *Seudocódigo para el algoritmo de particionamiento. [Archivo: partsc.cpp]*

Avanzar `l` lo más a la derecha posible significa avanzar `l` hasta encontrar un elemento mayor o igual que `v`. Notar que intercambiar los elementos, garantiza que en la siguiente ejecución del lazo cada uno de los cursores `l` y `r` avanzarán al menos una posición ya que, después de intercambiar, el elemento en `l` será menor que `v` y el que está en `r` será mayor o igual que `v`. El algoritmo termina cuando `l` y `r` se *"cruzan"*, es decir cuando `l>r`. En ese

caso `l` representa el primer elemento de la partición derecha, el cual debe ser retornado por `partition()` para ser usado en quick-sort.

```
1.  template<class T>
2.  typename std::vector<T>::iterator
3.  partition(typename std::vector<T>::iterator first,
4.          typename std::vector<T>::iterator last,
5.          bool (*comp)(T&,T&),T &pivot) {
6.    typename std::vector<T>::iterator
7.      l = first,
8.      r = last;
9.    r--;
10.   while (true) {
11.     T tmp = *l;
12.     *l = *r;
13.     *r = tmp;
14.     while (comp(*l,pivot)) l++;
15.     while (!comp(*r,pivot)) r--;
16.     if (l>r) break;
17.   }
18.   return l;
19. }
```

Código 5.11: *Algoritmo de partición para quick-sort. [Archivo: qspart.h]*

5.4.7. Tiempo de ejecución del algoritmo de particionamiento

En el código 5.11 vemos una implementación de `partition()`. Recordemos que para que las estimaciones del tiempo de ejecución de quick-sort sean válidas, en particular (5.14) y (5.18), debe valer (5.11), es decir que el tiempo de particionamiento sea lineal. El tiempo de particionamiento es el del lazo de las líneas 10–17, es decir la suma de los lazos de avances de `l` y retrocesos de `r` más los intercambios. Ahora bien, en cada avance de `l` y retroceso de `l` la longitud del rango `[l,r]` que es la que todavía falta particionar se reduce en uno, de manera que a lo sumo puede haber $n =$`last-first` avances y retrocesos. Por otra parte, por cada intercambio debe haber al menos un avance de `l` y un retroceso de `r` de manera que hay a lo sumo $n/2$ intercambios, con lo cual se demuestra que todo el tiempo de `partition()` es a lo sumo $O(n)$.

5.4.8. Búsqueda del pivote por la mediana

```
1.  template<class T>
2.  int median(typename std::vector<T>::iterator first,
3.        typename std::vector<T>::iterator last,
4.        std::vector<T> &dif, int k,
5.        bool (*comp)(T&,T&)) {
6.    typename std::vector<T>::iterator
7.      q = first;
8.    int ndif=1;
9.    dif[0] = *q++;
10.   while (q<last) {
11.     T val = *q++;
12.     int j;
13.     for (j=0; j<ndif; j++)
14.       // Aca debe compararse por 'equivalente'
15.       // es decir usando comp
16.       if (!comp(dif[j],val)
17.          && !comp(val,dif[j])) break;
18.     if (j==ndif) {
19.       dif[j] = val;
20.       ndif++;
21.       if (ndif==k) break;
22.     }
23.   }
24.   typename std::vector<T>::iterator
25.     s = dif.begin();
26.   bubble_sort(s,s+ndif,comp);
27.   return ndif;
28. }
```

Código 5.12: *Función para calcular el pivote en quick-sort. [Archivo: qsmedian.h]*

En el código 5.12 podemos ver una implementación de la función que calcula el pivote usando la estrategia de la mediana de los k primeros. La función **ndif=median(first,last,dif,k,comp)** busca los k primeros elementos distintos del rango **[first,last)** en el vector **dif** y retorna el número exacto de elementos distintos encontrados (que puede ser menor que **k**). Recordemos que el número de elementos distintos es usado en quick-sort para cortar la recursión. Si no hay al menos dos elementos distintos entonces no es necesario particionar el rango.

El algoritmo es $O(n)$ mientras que k sea fijo, esto es, que no crezca con n. Se recorre el rango y se van introduciendo los nuevos elementos distintos en el vector **dif**. Para ver si un elemento es distinto se compara con todos los elementos previamente insertados en **dif**. En muy importante que la comparación debe realizarse por *equivalencia* (ver línea 17), y no por *igualdad*. Es decir dos elementos a y b son equivalentes si **comp(a,b) && comp(b,a)** es verdadero. Para relaciones de orden débiles esto es muy importante ya que si todos los elementos en el rango son equivalentes pero no iguales (pensemos en $(-1, 1, -1)$ con la relación de orden (5.3), menor en valor absoluto), entonces si **partition()** comparara por igualdad reportaría dos elementos distintos, pero después al particionar una de las particiones resultaría vacía y entraría en un lazo infinito.

5.4.9. Implementación de quick-sort

```
1.  template<class T> void
2.  quick_sort(typename std::vector<T>::iterator first,
3.         typename std::vector<T>::iterator last,
4.         bool (*comp)(T&,T&)) {
5.    int size = last-first;
6.    int max_bub_size = 9;
7.    if (size<max_bub_size) {
8.      bubble_sort(first,last,comp);
9.      return;
10.   }
11.   if (size<=1) return;
12.   int k=3;
13.   std::vector<T> dif(k);
14.   int ndif = median(first, last, dif, k, comp);
15.   if (ndif==1) return;
16.   T pivot = dif[ndif/2];
17.   typename std::vector<T>::iterator l;
18.   l = partition(first,last,comp,pivot);
19.   quick_sort(first,l,comp);
20.   quick_sort(l,last,comp);
21. }
22.
23. template<class T> void
24. quick_sort(typename std::vector<T>::iterator first,
25.         typename std::vector<T>::iterator last) {
26.   quick_sort(first,last,less<T>);
27. }
```

Código 5.13: *Algoritmo de ordenamiento rápido (quick-sort) [Archivo: qsort.h]*

En el código 5.13 vemos la implementación de la rutina principal de quick-sort.

- Una mejora para quick-sort consiste en usar para las particiones más pequeñas otro algoritmo, por ejemplo el método de la burbuja. Recordemos que si bien la tasa de crecimiento $O(n^2)$ nos indica que para valores grandes de n burbuja será siempre más ineficiente que quick-sort, para valores pequeños puede ser más eficiente y de hecho lo es. Por eso, para longitudes de rangos menores que **`max_bub_size`** la función llama a **`bubble_sort()`** en vez de llamar recursivamente a **`quick_sort()`**. La constante **`max_bub_size`** óptima se halla por prueba y error y en nuestro caso se ha elegido **`max_bub_size=9`**.
- Notemos que en este caso **`quick_sort()`** corta la recursión de varias formas posibles, a saber:
  - Cuando cambia a **`bubble_sort()`** como se mencionó en el punto anterior.
  - Cuando la longitud del rango es 1. Esto sólo puede ocurrir si se elige **`max_bub_size=0`** para evitar cambiar a llamar a **`bubble_sort()`** (por la razón que fuere).
  - Cuando no se detectan dos o más elementos distintos (mejor dicho, no equivalentes).
- Las últimas tres líneas de **`quick_sort()`** simplemente reflejan el seudocódigo 5.9.

5.4.10. Estabilidad

Quick-sort es estable si el algoritmo de partición lo es, y tal cual como está implementado aquí, el algoritmo de *partición no es estable*, ya que al hacer el intercambio en **`partition()`** un elemento puede ser intercambiado con un elemento equivalente.

```
1. template<class T>
2. typename std::vector<T>::iterator
3. stable_partition(typename std::vector<T>::iterator first,
```

```
4.          typename std::vector<T>::iterator last,
5.          bool (*comp)(T&,T&),T &pivot) {
6.    int size = (last-first);
7.    if (size==1) return (comp(*first,pivot)? last : first);
8.    typename std::vector<T>::iterator
9.     middle = first + size/2,
10.    l1, l2;
11.   l1 = stable_partition(first,middle,comp,pivot);
12.   l2 = stable_partition(middle,last,comp,pivot);
13.   range_swap<T>(l1,middle,l2);
14.   return l1+(l2-middle);
15. }
```

Código 5.14: *Seudocódigo para el algoritmo de partición estable. [Archivo: stabpart.h]*

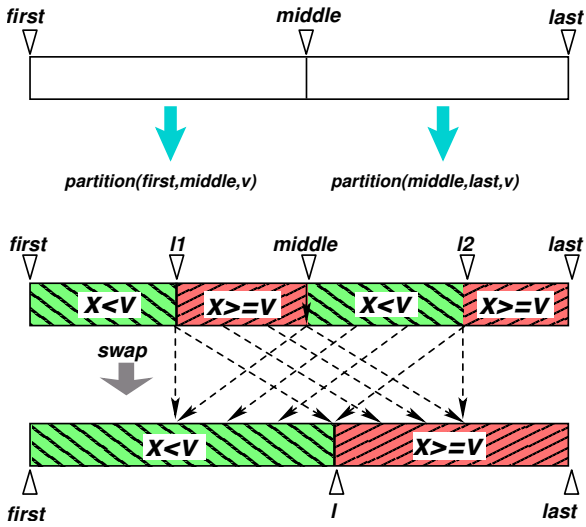


Figura 5.9: Algoritmo de partición estable.

Es sencillo implementar una variante estable de `partition()` como se observa en el seudo código 5.14. El algoritmo de partición es recursivo. Primero dividimos el rango `[first,last)` por el punto medio `middle`. Aplicamos recursivamente `partition()` a cada una de los rangos izquierdo (`[first,middle)`) y derecho (`[middle,last)`) retornando los puntos de separación de cada una de ambas particiones `l1` y `l2`. Una vez que ambos

rangos están particionados sólo hace falta intercambiar (*"swap"*) los rangos `[l1,middle)` y `[middle,l2)`. Si el particionamiento de ambos subrangos fue estable y al hacer el swap mantenemos el orden relativo de los elementos en cada uno de los rangos, entonces la partición de `[first,last)` será estable, ya que los elementos de `[middle,l2)` son estrictamente mayores que los de `[l1,middle)`.

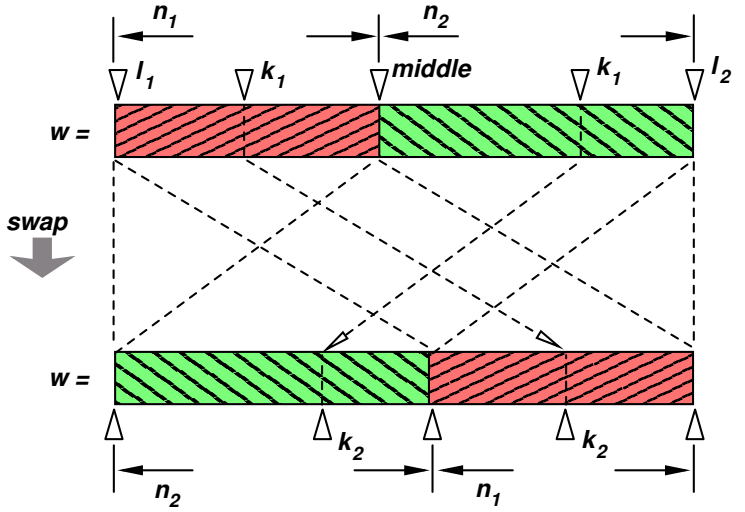


Figura 5.10:

5.4.11. El algoritmo de intercambio (swap)

Llamaremos a esta operación `swap(l1,middle,l2)`. Notemos que a cada posición `l1+k1` en el rango `[l1,l2)` le corresponde, después del intercambio, otra posición `l1+k2` en el rango `[l1,l2)` . La relación entre `k1` y `k2` es biunívoca, es decir, la operación de intercambio es una *"permutación"* de los elementos del rango. Si llamamos `n1` y `n2` las longitudes de cada uno de los rangos a intercambiar, entonces tenemos

$$k_2 = \begin{cases} k_1 + n_2; & \text{si } k_1 < n_1 \\ k_1 - n_1; & \text{si } k_1 \geq n_1 \end{cases} \tag{5.20}$$

o recíprocamente,

$$k_1 = \begin{cases} k_2 + n_1; & \text{si } k_2 < n_2 \\ k_2 - n_2; & \text{si } k_2 \geq n_2 \end{cases} \tag{5.21}$$

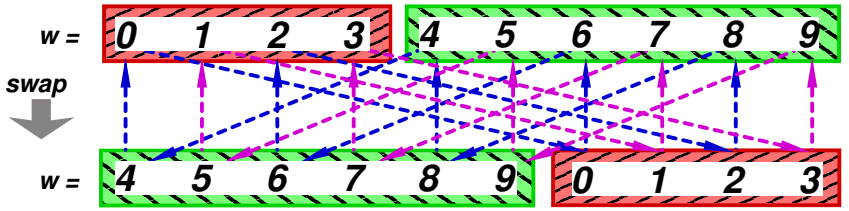


Figura 5.11: Algoritmo de intercambio de rangos.

Para describir el algoritmo es más simple pensar que tenemos los elementos del rango `[l1,l2)` en un vector `w` de longitud `n1+n2`. Consideremos por ejemplo el caso $n_1 = 4$, $n_2 = 6$ (ver figura 5.11). De acuerdo con (5.21) el elemento que debe ir a la primera posición es el que esta en la posición 4. Podemos guardar el primer elemento (posición 0) en una variable temporaria `tmp` y traer el 4 a la posición 0. A su vez podemos poner en 4 el que corresponde allí, que está inicialmente en la posición 8 y así siguiendo se desencadenan una serie de intercambios hasta que el que corresponde poner en la posición a rellenar es el que tenemos guardado en la variable temporaria (por ahora el 0).

```
1. T tmp = w[0];
2. int k2 = 0;
3. while (true) {
4.   int k1 = (k2<n2 ? k2+n1 : k2-n2);
5.   if (k1==0) break;
6.   w[k2] = w[k1];
7.   k2 = k1;
8. }
9. w[k2] = tmp;
```

Código 5.15: *Algoritmo de rotación para el swap de rangos. [Archivo: swapsc.cpp]*

En el código 5.15 vemos como sería el algoritmo de rotación. En el ejemplo, los intercambios producidos serían

$$\texttt{tmp} \leftarrow \texttt{w}[0] \leftarrow \texttt{w}[4] \leftarrow \texttt{w}[8] \leftarrow \texttt{w}[2] \leftarrow \texttt{w}[6] \leftarrow \texttt{tmp} \tag{5.22}$$

Esta rotación de los elementos se muestra en la figura con flechas azules. Podemos ver que esto rota 5 elementos, los otros 5 rotan de la misma forma

si comenzamos guardando en **tmp** el elemento de la posición 1, trayendo al 1 el de la posición 5, y así siguiendo

$$\texttt{tmp} \leftarrow \texttt{w}[1] \leftarrow \texttt{w}[5] \leftarrow \texttt{w}[9] \leftarrow \texttt{w}[3] \leftarrow \texttt{w}[7] \leftarrow \texttt{tmp} \tag{5.23}$$

Notar que los elementos que se rotan son exactamente los que fueron rotados previamente, incrementados en uno.

Si $n_1 = n_2$ (por ejemplo $n_1 = n_2 = 5$), entonces hay cinco rotaciones de dos elementos,

$$\begin{aligned}
&\texttt{tmp} \leftarrow \texttt{w}[0] \leftarrow \texttt{w}[5] \leftarrow \texttt{tmp}\\
&\texttt{tmp} \leftarrow \texttt{w}[1] \leftarrow \texttt{w}[6] \leftarrow \texttt{tmp}\\
&\texttt{tmp} \leftarrow \texttt{w}[2] \leftarrow \texttt{w}[7] \leftarrow \texttt{tmp}\\
&\texttt{tmp} \leftarrow \texttt{w}[3] \leftarrow \texttt{w}[8] \leftarrow \texttt{tmp}\\
&\texttt{tmp} \leftarrow \texttt{w}[4] \leftarrow \texttt{w}[9] \leftarrow \texttt{tmp}
\end{aligned} \tag{5.24}$$

Si n_1 divide a n_2 (por ejemplo $n_1 = 2$ y $n_2 = 8$), entonces se generan 2 rotaciones de 5 elementos, a saber

$$\begin{aligned}
&\texttt{tmp} \leftarrow \texttt{w}[0] \leftarrow \texttt{w}[2] \leftarrow \texttt{w}[4] \leftarrow \texttt{w}[6] \leftarrow \texttt{w}[8] \leftarrow \texttt{tmp}\\
&\texttt{tmp} \leftarrow \texttt{w}[1] \leftarrow \texttt{w}[3] \leftarrow \texttt{w}[5] \leftarrow \texttt{w}[7] \leftarrow \texttt{w}[9] \leftarrow \texttt{tmp}
\end{aligned} \tag{5.25}$$

Observando con detenimiento podemos encontrar una regla general, a saber que el número de rotaciones es $m = \gcd(n_1, n_2)$, donde $\gcd(n_1, n_2)$ es el máximo común divisor de n_1 y n_2. Además, como el número de rotaciones por el número de elementos rotados en cada rotación debe ser igual al número total de elementos $n_1 + n_2$ debemos tener que el número de elementos rotados en cada rotación es $(n_1 + n_2)/m$. Las rotaciones empiezan en los elementos 0 a $m - 1$.

```
1.  int gcd(int m,int n) {
2.    int M,N;
3.    if (m>n) {
4.      M=m; N=n;
5.    } else {
6.      N=m; M=n;
7.    }
8.    while (true) {
9.      int rest = M% N;
10.     if (!rest) return N;
11.     M = N; N= rest;
12.   }
13. }
```

```
14.
15.  template<class T>
16.  void range_swap(typename std::vector<T>::iterator first,
17.                  typename std::vector<T>::iterator middle,
18.                  typename std::vector<T>::iterator last) {
19.   int
20.    n1 = middle-first,
21.    n2 = last-middle;
22.   if (!n1 || !n2) return;
23.   int m = gcd(n1,n2);
24.   for (int j=0; j<m; j++) {
25.    T tmp = *(first+j);
26.    int k2 = j;
27.    while (true) {
28.     int k1 = (k2<n2 ? k2+n1 : k2-n2);
29.     if (k1==j) break;
30.     *(first+k2) = *(first+k1);
31.     k2 = k1;
32.    }
33.    *(first+k2) = tmp;
34.   }
35.  }
```

Código 5.16: *Algoritmo de intercambio de dos rangos consecutivos. [Archivo: swap.h]*

El algoritmo de particionamiento estable se observa en el código 5.16. La función **`int gcd(int,int)`** calcula el máximo común divisor usando el algoritmo de Euclides (ver sección §4.1.3). El tiempo de ejecución de **`swap()`** es $O(n)$, donde n, ya que en cada ejecución del lazo un elemento va a su posición final.

Si reemplazamos en quick-sort (código 5.13) la función **`partition()`** (código 5.11) por **`stable_partition()`** (código 5.14) el algoritmo se hace estable.

5.4.12. Tiempo de ejecución del quick-sort estable

Sin embargo el algoritmo de particionamiento estable propuesto ya no es $O(n)$. De hecho el tiempo de ejecución de **`stable_partition()`** se puede analizar de la misma forma que el de quick-sort mismo en el mejor caso. Es decir, su tiempo de ejecución satisface una relación recursiva como (5.12)

donde el tiempo cn ahora es el tiempo de **`swap()`**. Por lo tanto el tiempo de ejecución de **`stable_partition()`** es $O(n \log n)$ en el peor caso.

Ahora para obtener el tiempo de ejecución de la versión estable de quicksort en el mejor caso volvemos a (5.12) teniendo en cuenta la nueva estimación para el algoritmo de partición

$$T(n) = n \log n + 2T(n/2) \tag{5.26}$$

Ambas (5.26) y (5.12) son casos especiales de una relación de recurrencia general que surge naturalmente al considerar algoritmos de tipo *"dividir para vencer"*

$$T(n) = f(n) + 2T(n/2), \tag{5.27}$$

donde $f(n) = n \log n$ en el caso de (5.26) y $f(n) = cn$ para (5.12). Aplicando recursivamente obtenemos

$$\begin{aligned} T(2) &= f(2) + 2T(1) = f(2) + 2d \\ T(4) &= f(4) + 2T(2) = f(4) + 2f(2) + 4d \\ T(8) &= f(8) + 2T(4) = f(8) + 2f(4) + 4f(2) + 8d \\ \vdots &= \quad \vdots \end{aligned} \tag{5.28}$$

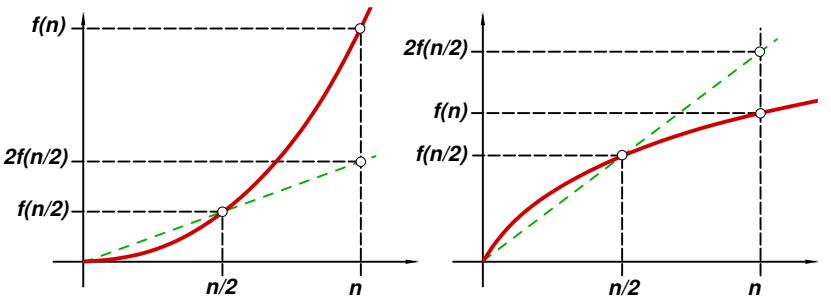


Figura 5.12: Ejemplo de crecimiento *"más que lineal"* (izquierda) y menos que lineal (derecha)

Si $f(n)$ es una función *"cóncava hacia arriba"* (ver figura 5.12, izquierda) entonces es válido que

$$2f(n/2) \leq f(n), \tag{5.29}$$

mientras que si es cóncava hacia abajo (ver figura 5.12, derecha) entonces

$$2f(n/2) \geq f(n). \tag{5.30}$$

También decimos que la función tiene crecimiento *"más que lineal"* o *"menos que lineal"*, respectivamente. Si la función crece más que linealmente, como en el caso de $n \log n$, entonces podemos acotar

$$\begin{aligned} 2f(2) &\leq f(4) \\ 2f(4) &\leq f(8) \\ 4f(2) \leq 2f(4) &\leq f(8) \\ \vdots \quad &\quad \vdots \end{aligned} \tag{5.31}$$

de manera que

$$\begin{aligned} T(8) &\leq 3f(8) + 8d \\ T(16) = f(16) + 2T(8) &\leq f(16) + 6f(8) + 16d \leq 4f(16) + 16d \\ \vdots \quad &\quad \vdots \end{aligned} \tag{5.32}$$

y, en general

$$T(n) \leq (\log n)\, f(n) + nd. \tag{5.33}$$

Si lo aplicamos a quick-sort estable con $f(n) = n \log n$ llegamos a

$$T(n) = O(n\, (\log n)^2) \tag{5.34}$$

5.5. Ordenamiento por montículos

```
1.  // Fase inicial
2.  // Pone todos los elementos en S
3.  while (!L.empty()) {
4.    x = *L.begin();
5.    S.insert(x);
6.    L.erase(L.begin());
7.  }
8.  // Fase final
9.  // Saca los elementos de S usando 'min'
10. while (!S.empty()) {
11.   x = *S.begin();
12.   S.erase(S.begin());
13.   L.push(L.end(),x);
```

Código 5.17: *Algoritmo de ordenamiento usando un conjunto auxiliar. [Archivo: heapsortsc.cpp]*

Podemos ordenar elementos usando un **set<>** de STL ya que al recorrer los elementos con iteradores estos están guardados en forma ordenada. Si logramos una implementación de **set<>** tal que la inserción y supresión de elementos sea $O(\log n)$, entonces basta con insertar los elementos a ordenar y después extraerlos recorriendo el conjunto con **begin()** y **operator++()**. Si, por ejemplo, asumimos que los elementos están en una lista, el algoritmo sería como se muestra en el seudocódigo 5.17.

- La representación por *árboles binarios de búsqueda* (ver sección §4.6.5) sería en principio válida, aunque si los elementos no son insertados en forma apropiada el costo de las inserciones o supresiones puede llegar a $O(n)$. Hasta ahora no hemos discutido ninguna implementación de **set<>** en el cual inserciones y supresiones sean $O(\log n)$ *siempre*.
- Una desventaja es que el conjunto no acepta elementos diferentes, de manera que esto serviría sólo para ordenar contenedores con elementos diferentes. Esto se podría remediar usando un **multiset** [SGI, 1999].
- Esta forma de ordenar no es in-place y de hecho las representaciones de **set<>** requieren una considerable cantidad de memoria adicional por elemento almacenado.

5.5.1. El montículo

El *"montículo"* (*"heap"*) es una estructura de datos que permite representar en forma muy conveniente un TAD similar al conjunto llamado *"cola de prioridad"*. La cola de prioridad difiere del conjunto en que no tiene las operaciones binarias ni tampoco operaciones para recorrer el contenedor como **end()** y **operator++()**. Si tiene una función **min()** que devuelve una posición al menor elemento del conjunto, y por lo tanto es equivalente a **begin()**.

El montículo representa la cola de prioridad almacenando los elementos en un árbol binario con las siguientes características

- Es *"parcialmente ordenado"* (PO), es decir el padre es siempre menor o igual que sus dos hijos.
- Es *"parcialmente completo"*: Todos los niveles están ocupados, menos el último nivel, en el cual están ocupados todos los lugares más a la izquierda.

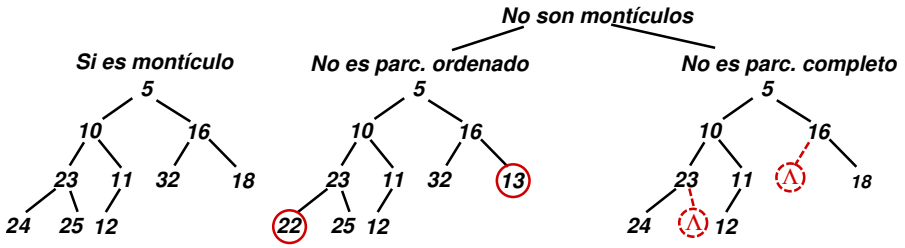


Figura 5.13: Ejemplo de árboles que cumplen y no cumplen la condición de montículo.

En la figura 5.13 vemos tres árboles binarios de los cuales sólo el de más a la izquierda cumples con todas las condiciones de montículo. El del centro no cumple con la condición de PO ya que los elementos 22 y 13 (marcados en rojo) son menores que sus padres. Por otra parte, el de la derecha satisface la condición de PO pero no es parcialmente completo debido a los nodos marcados como Λ.

Notemos que la propiedad de PO es recursiva. Podemos decir que un árbol binario es PO si la raíz es menor que sus hijos (es decir, es *localmente* PO)) y los subárboles de sus hijos son PO. Por otra parte la propiedad de parcialmente completo *no es recursiva*.

5.5.2. Propiedades

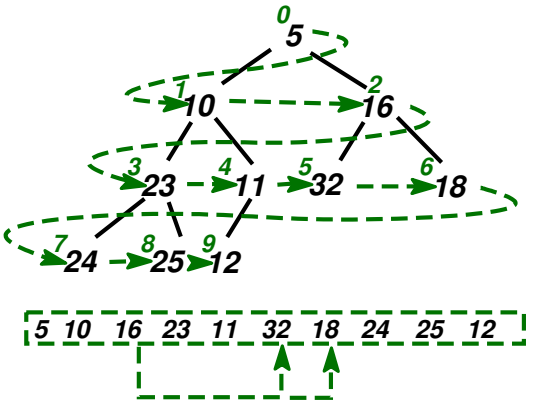


Figura 5.14: Almacenamiento por niveles de los elementos de un montículo en un arreglo.

La condición de que sea PO implica que el mínimo está siempre en la raíz. La condición de parcialmente completo permite implementarlo eficientemente en un vector. Los elementos son almacenados en el vector *por orden de nivel*, es decir primero la raíz, en la posición 0 del vector, después los dos hijos de la raíz de izquierda a derecha en las posiciones 1 y 2, después los 4 elementos del nivel 2 y así siguiendo. Esto se representa en la figura 5.14 donde los números en verde representan la posición en el vector. Notemos que las posiciones de los hijos se pueden calcular en forma algebraica a partir de la posición del padre

$$\begin{aligned} \text{hijo izquierdo de } j &= 2j+1, \\ \text{hijo derecho de } j &= 2j+2. \end{aligned} \tag{5.35}$$

Notemos también que cada subárbol del montículo es a su vez un montículo.

5.5.3. Inserción

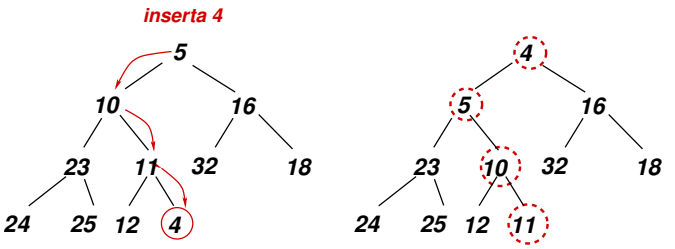


Figura 5.15: Inserción en un montículo.

Para poder usar el montículo para poder ordenar debemos poder insertar nuevos elementos, manteniendo la propiedad de montículo. El procedimiento consiste en insertar inicialmente el elemento en la *primera posición libre*, es decir la posición libre lo más a la izquierda posible del último nivel semicompleto o, si no hay ningún nivel semicompleto, la primera posición a la izquierda del primer nivel vacío. En la figura 5.15 vemos un ejemplo en el cual insertamos el elemento 4 en un montículo que hasta ese momento contiene 10 elementos. Una vez así insertado el elemento, se cumple la condición de parcialmente completo pero probablemente no la de PO. Esta última se restituye haciendo una serie de intercambios. Notar que los intercambios de elementos no pueden quebrar la propiedad de parcialmente completo.

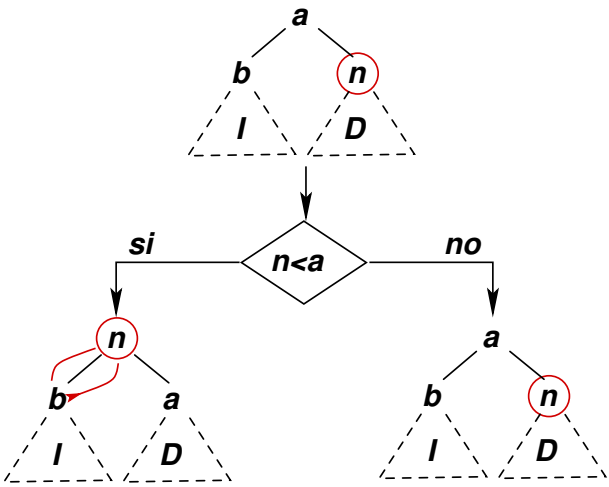


Figura 5.16: Esquema del restablecimiento de la propiedad de montículo al subir el nuevo elemento.

Para restituir la propiedad de PO vamos intercambiando el elemento insertado con su padre, si éste es estrictamente mayor. El proceso se detiene al encontrar un padre que no es mayor o equivalente al elemento. En cierta forma, este procedimiento es similar al que usa el método de la burbuja para ir *"flotando"* cada uno de los elementos en el lazo interno, con la salvedad que aquí el proceso se realiza sobre el camino que va desde el nuevo elemento a la raíz. Por supuesto, en el montículo pueden existir elementos iguales (en realidad *equivalentes*), es decir *no es un conjunto*. Cada vez que se intercambia el elemento con su padre el subárbol de la nueva posición donde va el elemento recupera la propiedad de montículo. Consideremos por ejemplo el caso de la figura 5.16 donde el nuevo elemento n ha subido hasta ser raíz del subárbol D. Queremos ver como se restituye la propiedad de montículo al intercambiar (o no) n con su padre a. El caso en que el nuevo elemento es raíz del subárbol izquierdo es exactamente igual, *mutatis mutandis*. Notemos que el subárbol I del hijo izquierdo de a también debe ser un montículo ya que no fue tocado durante el proceso de subir n hasta su posición actual (todo el camino por donde subió n debe estar contenido dentro de D. Ahora supongamos que $a \leq n$, entonces no intercambiamos a con n como se muestra en la parte inferior derecha de la figura. Es claro que se verifica localmente la condición de PO y como a su vez cada uno de los subárboles I y D son PO, todo el subárbol de a es PO. Si por otra parte

$a > n$ (en la parte inferior izquierda de la figura) entonces al intercambiar a con n es claro que D quedará como un montículo, ya que hemos reemplazado la raíz por un elemento todavía menor. Por otra parte I ya era un montículo y lo seguirá siendo porque no fue modificado. Finalmente la condición de PO se satisface localmente entre n, b y a ya que $n < a$ y como el la condición se satisfacía localmente antes de que subiera n, debía ser $a \leq b$, por lo tanto $n < a \leq b$.

5.5.4. Costo de la inserción

Notemos que cada intercambio es $O(1)$ por lo tanto todo el costo de la inserción es básicamente orden de la longitud l del camino que debe subir el nuevo elemento desde la nueva posición inicial hasta algún punto, eventualmente en el peor de los casos hasta la raíz. Pero como se trata de un árbol parcialmente completo la longitud de los caminos contenidos en el árbol está acotada por (ver sección §3.8.3)

$$T(n) = O(l) = O(\log_2 n). \tag{5.36}$$

Notemos que esta estimación es válida *en el peor caso*. A diferencia del ABB (ver sección §4.6), el montículo no sufre de problemas de *"balanceo"* ya que siempre es mantenido en un árbol parcialmente completo.

Este algoritmo de inserción permite implementar la línea 5 en el seudocódigo 5.17 en $O(n \log_2 n)$.

5.5.5. Eliminar el mínimo. Re-heap.

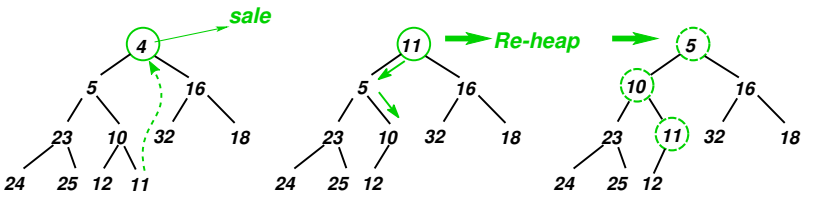


Figura 5.17: Procedimiento de eliminar el elemento mínimo en montículos.

Ahora vamos a implementar las líneas 11–12, esto es una operación combinada en la cual se recupera el valor del mínimo y se elimina este valor del contenedor. En el montículo el elemento menor se encuentra en la raíz, por construcción, es decir en la primera posición del vector. Para eliminar el

elemento debemos rellenar el *"hueco"* con un elemento. Para mantener la propiedad de parcialmente completo, subimos a la raíz el último elemento (el más a la derecha) del último nivel semicompleto. Por ejemplo, considerando el montículo inicial de la figura 5.17 a la izquierda, entonces al eliminar el 4 de la raíz, inmediatamente subimos el 11 para *"rellenar"* el hueco creado, quedando el árbol como se muestra en la figura, en el centro. Este árbol satisface todas las condiciones de montículo menos localmente la condición PO en la raíz. Al proceso de realizar una serie de intercambios para restaurar la condición de montículo en un árbol en el cual la única condición que se viola es la de PO (localmente) en la raíz se le llama *"rehacer el montículo"* (*"re-heap"*). Este proceso consiste en ir bajando el elemento de la raíz, intercambiándolo con el menor de sus hijos hasta encontrar una hoja, o una posición en la cual los dos hijos son mayores o equivalentes que el elemento que baja. En el caso de la figura, el 11 (que subió a la raíz para reemplazar al 4 que se fue), baja por el 5, ya que 5 es el menor de los dos hijos. Luego por el 10 y se detiene allí ya que en ese momento su único hijo es el 12 que es mayor que él. La situación final es como se muestra a la derecha.

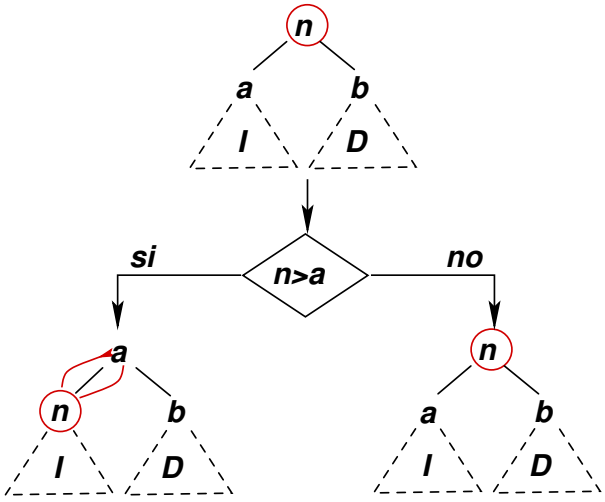


Figura 5.18: Intercambio básico en el re-heap. Asumimos $a \leq b$.

Ahora para entender mejor porque funciona este algoritmo consideremos el caso general mostrado en la figura 5.18. Tenemos un árbol que satisface las propiedades de montículo en todos sus puntos salvo, eventualmente, la condición de PO localmente en la raíz. Llamemos n al nodo de la raíz y a, b sus hijos. Para fijar ideas asumamos que $a \leq b$, ya que el caso $b > a$ es

exactamente igual, *mutatis mutandis*. Ahora si $n > a$ entonces intercambiamos n con a, quedando como en la parte inferior izquierda de la figura, caso contrario el árbol queda igual. Queremos ver que cada vez que el nodo n va bajando una posición la condición de PO se viola (eventualmente) sólo en el nodo que está bajando y en ese caso se viola localmente. Consideremos primero el caso $n > a$, podemos ver que se ha restablecido la condición PO en la raíz, ya que $a < n$ y habíamos asumido que $a \leq b$. Entonces a esta altura la condición de PO puede violares solamente en la raíz del subárbol izquierdo I. Por otra parte si $n \leq a$ entonces todo el árbol es un montículo ya que $n \leq a$ y $n \leq b$ (ya que $a \leq b$).

5.5.6. Costo de re-heap

De nuevo, el costo de re-heap está dado por el número de intercambios hasta que el elemento no baja más o llega a una hoja. En ambos casos el razonamiento es igual al de la inserción, como el procedimiento de bajar el elemento en el re-heap se da por un camino, el costo es a los sumo $O(l)$ donde l es la longitud del máximo camino contenido en el montículo y, por lo tanto $O(\log_2 n)$. El costo de toda la segunda fase del algoritmo (seudo código 5.17) es $O(n \log_2 n)$.

5.5.7. Implementación in-place

Con el procedimiento de inserción y re-heap podemos ya implementar un algoritmo de ordenamiento $O(n \log_2 n)$. Alocamos un nuevo vector de longitud n que contendrá el montículo. Vamos insertando todos los elementos del vector original en el montículo, el cual va creciendo hasta llenar todo el vector auxiliar. Una vez insertados todos los elementos, se van eliminando del montículo, el cual va disminuyendo en longitud, y se van re-insertando en el vector original de adelante hacia atrás. Después de haber re-insertado todos los elementos, el vector original queda ordenado y se desaloca el vector auxiliar.

Sin embargo, la implementación descrita no es in-place, ya que necesita un vector auxiliar. Sin embargo, es fácil modificar el algoritmo para que sea in-place. Notemos que a medida que vamos sacando elementos del vector para insertarlos en el montículo en la primera fase y en el otro sentido en la segunda fase, en todo momento la suma de las longitudes del vector y del montículo es exactamente n de manera que podemos usar el mismo vector

original para almacenar el vector ordenado y el montículo en la fase inicial y el vector ordenado y el montículo en la fase final.

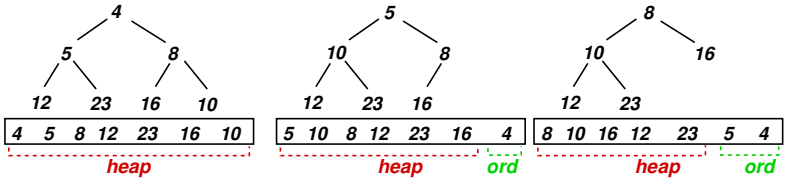


Figura 5.19: Implementación in-place de la fase final de heap-sort.

Para ilustrar mostramos en la figura 5.19 los primeros pasos de la fase final de heap-sort. En la figura de la izquierda se ve el vector que tiene 7 elementos. ocupado totalmente por el montículo. Ahora sacamos el elemento menor que es un 4. Como el tamaño del montículo se reduce en uno, queda al final del vector exactamente un lugar para guardar el elemento eliminado. En el paso siguiente se extrae el nuevo mínimo que es 5, el cual va la segunda posición desde el final. A esta altura el vector contiene al montículo en sus 5 primeros elementos y el vector ordenado ocupa los dos últimos elementos. A medida que se van sacando elementos del montículo, este se reduce en longitud y la parte ordenada del final va creciendo hasta que finalmente todo el vector queda ocupado por el vector ordenado.

Sin embargo, tal cual como se describió aquí, el vector queda ordenado de mayor a menor, lo cual no es exactamente lo planeado en un principio. Pero invertirlo es un proceso muy simple y $O(n)$ de manera que es absorbido en el costo global $O(n \log_2 n)$. Sin embargo existe una posibilidad mejor aún, en vez de usar un montículo minimal, como se describió hasta aquí, podemos usar uno maximal (que es exactamente igual, pero donde el padre es mayor o igual que los hijos). De esta forma en la fase final vamos siempre extrayendo el máximo, de manera que el vector queda ordenado de menor a mayor.

5.5.8. El procedimiento make-heap

La fase inicial tiene como objetivo convertir al vector, inicialmente desordenado, en un montículo. Existe una forma más eficiente de realizar esta tarea mediante un procedimiento llamado *make-heap*. En este procedimiento consiste en aplicar re-heap a cada uno de los nodos interiores (aquellos que no son hojas) *desde abajo hacia arriba*. Consideremos por ejemplo el árbol de la figura 5.20 a la izquierda. Aplicamos inicialmente re-heap a los

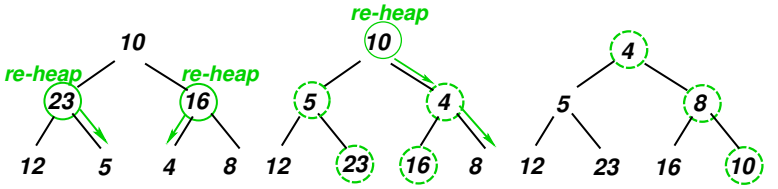


Figura 5.20: Procedimiento make-heap

nodos que están en el nivel 1 (el segundo desde abajo hacia arriba). Recordemos que el re-heap asume que la condición de PO se viola únicamente en la raíz. Como los árboles de los nodos en el nivel 1 tienen profundidad 1, esta condición ciertamente se cumple. Una vez aplicado el re-heap a cada uno de los nodos del nivel 1 el árbol queda como en la figura del centro. Ahora están dadas las condiciones para aplicar el re-heap el árbol en su totalidad ya que la condición de PO sólo se viola eventualmente en la raíz. En general, si ya hemos aplicado el re-heap a todos los nodos del nivel l entonces ciertamente podemos aplicárselo a cualquier nodo del nivel $l-1$ ya que sus hijos derecho e izquierdo pertenecen al nivel l y por lo tanto sus subárboles ya son montículos.

Puede verse fácilmente que este algoritmo es $O(n \log_2 n)$ ya que cada re-heap es, a lo sumo, $O(\log_2 n)$ y se hacen $O(n/2)$ re-heaps. Sin embargo puede demostrarse que en realidad el make-heap es $O(n)$, es decir que tiene un costo aún menor. Notemos que esto no afecta la velocidad de crecimiento global de heap-sort ya que sigue dominando el costo de la etapa final que es $O(n \log_2 n)$, pero de todas formas el reemplazar la etapa inicial implementada mediante inserciones por el make-heap, baja prácticamente el costo global de heap-sort a la mitad.

Para ver que el costo de make-heap es $O(n)$ contemos los intercambios que se hacen en los re-heap. Notemos que al aplicarle los re-heap a los nodos del nivel j, de los cuales hay 2^j, el número de intercambios máximos que hay que hacer es $l-j$ de manera que para cada nivel son a lo sumo $2^j(l-j)$ intercambios. La cantidad total de intercambios es entonces

$$T(n) = \sum_{j=0}^{l} 2^j (l-j) \tag{5.37}$$

Notemos que podemos poner (5.37) como

$$T(n) = l\sum_{j=0}^{l} 2^j - \sum_{j=0}^{l} j2^j \tag{5.38}$$

La primera sumatoria es una serie geométrica

$$\sum_{j=0}^{l} 2^j = \frac{2^{l+1}-1}{2-1} = 2^{l+1} - 1. \tag{5.39}$$

La segunda sumatoria se puede calcular en forma cerrada en forma similar a la usada en la sección §4.5.4.1. Notemos primero que podemos reescribirla como

$$\sum_{j=0}^{l} j2^j = \sum_{j=0}^{l} j\mathrm{e}^{\alpha j}\Bigg|_{\alpha\,=\,\log 2}, \tag{5.40}$$

pero

$$j\mathrm{e}^{\alpha j} = \frac{\mathrm{d}}{\mathrm{d}\alpha}\mathrm{e}^{\alpha j} \tag{5.41}$$

de manera que

$$\sum_{j=0}^{l} j2^j = \sum_{j=0}^{l} \left(\frac{\mathrm{d}}{\mathrm{d}\alpha}\mathrm{e}^{\alpha j}\right)\Bigg|_{\alpha\,=\,\log_2} = \frac{\mathrm{d}}{\mathrm{d}\alpha}\left(\sum_{j=0}^{l}\mathrm{e}^{\alpha j}\right)\Bigg|_{\alpha\,=\,\log_2} \tag{5.42}$$

pero ahora la sumatoria es una suma geométrica de razón e^{α}, de manera que

$$\sum_{j=0}^{l}\mathrm{e}^{\alpha j} = \frac{\mathrm{e}^{\alpha(l+1)}-1}{\mathrm{e}^{\alpha}-1} \tag{5.43}$$

y

$$\frac{\mathrm{d}}{\mathrm{d}\alpha}\left(\frac{\mathrm{e}^{\alpha(l+1)}-1}{\mathrm{e}^{\alpha}-1}\right) = \frac{(l+1)\mathrm{e}^{\alpha(l+1)}(\mathrm{e}^{\alpha}-1) - (\mathrm{e}^{\alpha(l+1)}-1)\mathrm{e}^{\alpha}}{(\mathrm{e}^{\alpha}-1)^2} \tag{5.44}$$

de manera que,

$$\begin{aligned}\sum_{j=0}^{l} j2^j &= \frac{(l+1)\mathrm{e}^{\alpha(l+1)}(\mathrm{e}^{\alpha}-1) - (\mathrm{e}^{\alpha(l+1)}-1)\mathrm{e}^{\alpha}}{(\mathrm{e}^{\alpha}-1)^2}\Bigg|_{\alpha\,=\,\log_2} \\ &= (l+1)2^{l+1} - 2(2^{l+1}-1) \\ &= (l-1)2^{l+1} + 2.\end{aligned} \tag{5.45}$$

Reemplazando en (5.39) y (5.45) en (5.37) tenemos que

$$\begin{aligned} T(n) &= \sum_{j=0}^{l} 2^j(l-j) = (2^{l+1}-1)l - (l-1)2^{l+1} - 2 \\ &= 2^{l+1} - l - 2 = O(2^{l+1}) = O(n) \end{aligned} \tag{5.46}$$

5.5.9. Implementación

```
1.  template<class T> void
2.  re_heap(typename std::vector<T>::iterator first,
3.        typename std::vector<T>::iterator last,
4.        bool (*comp)(T&,T&),int j=0) {
5.   int size = (last-first);
6.   T tmp;
7.   while (true) {
8.    typename std::vector<T>::iterator
9.     higher,
10.    father = first + j,
11.    l = first + 2*j+1,
12.    r = l + 1;
13.   if (l>=last) break;
14.   if (r<last)
15.    higher = (comp(*l,*r) ? r : l);
16.   else higher = l;
17.   if (comp(*father,*higher)) {
18.    tmp = *higher;
19.    *higher = *father;
20.    *father = tmp;
21.   }
22.   j = higher - first;
23.  }
24. }
25.
26. template<class T> void
27. make_heap(typename std::vector<T>::iterator first,
28.        typename std::vector<T>::iterator last,
29.           bool (*comp)(T&,T&)) {
30.  int size = (last-first);
31.  for (int j=size/2-1; j>=0; j--)
32.   re_heap(first,last,comp,j);
33. }
34.
35. template<class T> void
36. heap_sort(typename std::vector<T>::iterator first,
```

```
37.         typename std::vector<T>::iterator last,
38.         bool (*comp)(T&,T&)) {
39.   make_heap(first,last,comp);
40.   typename std::vector<T>::iterator
41.    heap_last = last;
42.   T tmp;
43.   while (heap_last>first) {
44.    heap_last--;
45.    tmp = *first;
46.    *first = *heap_last;
47.    *heap_last = tmp;
48.    re_heap(first,heap_last,comp);
49.   }
50. }
51.
52. template<class T> void
53. heap_sort(typename std::vector<T>::iterator first,
54.         typename std::vector<T>::iterator last) {
55.   heap_sort(first,last,less<T>);
```

Código 5.18: *Algoritmo de ordenamiento por montículos. [Archivo: heap-sort.h]*

En el código 5.18 vemos una posible implementación del algoritmo de ordenamiento por montículos. El montículo utilizado es maximal.

- La función **re_heap(first,last,comp,j)** realiza el procedimiento descripto re-heap sobre el subárbol de l nodos **j** (relativo a **first**).
- Durante la segunda fase **re_heap()** será llamado siempre sobre la raíz del montículo (**j=0**) pero en la fase inicial de **make_heap()** será llamado sobre los nodos interiores.
- Los iteradores **father**, **l** y **r** apuntan al nodo padre y sus dos hijos donde el padre es el nodo que inicialmente está en la raíz y va bajando por el mayor (recordemos que el montículo es maximal).
- El algoritmo termina cuando el **father** es una hoja, lo cual se detecta en la línea 13.
- El iterator **higher** es igual a verb+r+ y **l**, dependiendo de cuál de ellos sea el mayor. Si el elemento en **higher** es mayor que el de **father** entonces los elementos son intercambiados.
- La línea 16 contempla el caso especial de que **father** tenga un sólo hijo. (Como el 12 en la figura 5.14.)

- `make_heap()` simplemente aplica `re_heap()` a los nodos interiores que van de `size/2-1` hasta `j=0`. (De abajo hacia arriba).
- `heap_sort(first,last,comp)` aplica `make_heap()` para construir el montículo en el rango `[first,last)`.
- En la segunda fase, el iterator `heap_last` marca la separación entre el montículo `[first,heap_last)` y el vector ordenado `[heap_last,last)`. Inicialmente `heap_last=last` y se va reduciendo en uno con cada re-heap hasta que finalmente `heap_last=first`.
- Las líneas 45–47 extraen el mínimo del montículo y suben el último elemento del mismo, insertando el mínimo en el frente del vector ordenado. Finalmente la línea 48 restituye la propiedad de montículo.

5.5.10. Propiedades del ordenamiento por montículo

Lo notable de heap-sort es que el tiempo de ejecución es $O(n \log n)$ *en el peor caso* y además es in-place. Sin embargo, en el caso promedio quicksort resulta ser más rápido (por un factor constante) por lo que muchas veces es elegido.

Heap-sort no es estable ya que en el momento de extraer el mínimo del montículo e intercambiarlo y subir el último elemento del mismo, no hay forma de garantizar que no se pase por encima de elementos equivalentes.

5.6. Ordenamiento por fusión

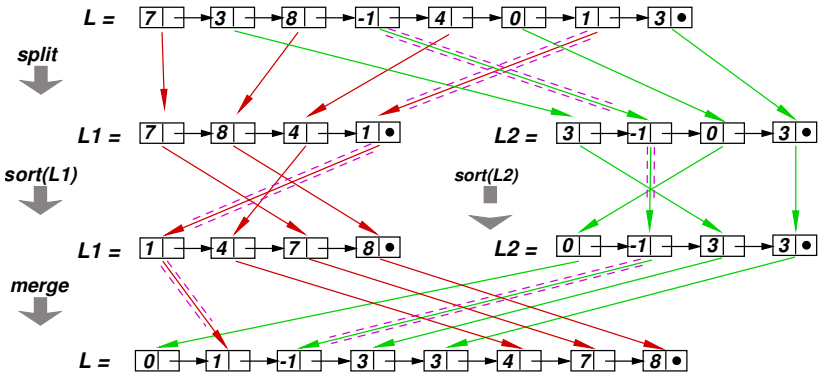


Figura 5.21: Ordenamiento de listas por fusión con splitting par/impar

```
1.  void merge_sort(list<T> &L,bool (comp*)(T&,T&)) {
2.    list<T>::iterator p = L.begin();
3.    if (p==L.end() || ++p==L.end()) return;
4.    list<T> L1,L2;
5.    // Separacion: separar L en dos sublistas de
6.    // tamano similar 'L1' y 'L2' . . .
7.    merge_sort(L1,comp);
8.    merge_sort(L2,comp);
9.    // Fusion: concatenar las listas 'L1' y 'L2' en 'L' . . .
10. }
```

Código 5.19: *Seudocódigo para el algoritmo de ordenamiento por fusión. [Archivo: mergesortsc.cpp]*

Conceptualmente, uno de los algoritmos rápidos más simples de comprender es el algoritmo de *"ordenamiento por fusión"* o *"intercalamiento"* (*"merge-sort"*). Si pensamos en el ordenamiento de listas, entonces el esquema sería como se muestra en el seudocódigo 5.17. Como quick-sort, la estrategia es también típica de *"dividir para vencer"* e intrínsecamente recursiva. Inicialmente (ver figura 5.21) la lista se divide (*"split"*) en dos sublistas del mismo tamaño y se aplica recursivamente **merge_sort()** a cada una de las listas. Luego estas se concatenan (también *"fusionan"* o *"merge"*) en **L** manteniendo el orden, como en **set_union()** (ver sección §4.3.0.2) para conjuntos por listas ordenadas. Si la división se hace en forma balanceada y las operaciones de división y concatenación se pueden lograr en tiempo $O(n)$ entonces el análisis de costo es similar al de quick-sort en el mejor caso y finalmente se obtiene un tiempo $O(n \log n)$. Ya hemos visto que el algoritmo de concatenación para listas ordenadas es $O(n)$ y para dividir la lista en dos de igual tamaño simplemente podemos ir tomando un elemento de **L** e ir poniéndolo alternadamente en **L1** y **L2**, lo cual es $O(n)$. A esta forma de separar la lista en dos de igual tamaño lo llamamos *"splitting par/impar"*. De manera que ciertamente es posible implementar merge-sort para listas en tiempo $O(n \log n)$.

El concepto de si el ordenamiento es *in-place* o no cambia un poco para listas. Si bien usamos contenedores auxiliares (las listas **L1** y **L2**), la cantidad total de celdas en juego es siempre n, si tomamos la precaución de ir eliminando las celdas de **L** a medidas que insertamos los elementos en las listas auxiliares, y *viceversa* al hacer la fusión. De manera que podemos decir que merge-sort es in-place.

Merge-sort es el algoritmo de elección para listas. Es simple, $O(n \log n)$ en el peor caso, y es *in-place*, mientras que cualquiera de los otros algoritmos rápidos como quick-sort y heap-sort se vuelven cuadráticos (o peor aún) al querer adaptarlos a listas, debido a la falta de iteradores de acceso aleatorio. También merge-sort es la base de los algoritmos para *ordenamiento externo*.

5.6.1. Implementación

```
1.  template<class T> void
2.  merge_sort(std::list<T> &L,bool (*comp)(T&,T&)) {
3.    std::list<T> L1,L2;
4.    list<T>::iterator p = L.begin();
5.    if (p==L.end() || ++p==L.end()) return;
6.    bool flag = true;
7.    while (!L.empty()) {
8.      std::list<T> &LL = (flag ? L1 : L2);
9.      LL.insert(LL.end(),*L.begin());
10.     L.erase(L.begin());
11.     flag = !flag;
12.   }
13.
14.   merge_sort(L1,comp);
15.   merge_sort(L2,comp);
16.
17.   typename std::list<T>::iterator
18.     p1 = L1.begin(),
19.     p2 = L2.begin();
20.   while (!L1.empty() && !L2.empty()) {
21.     std::list<T> &LL =
22.       (comp(*L2.begin(),*L1.begin()) ? L2 : L1);
23.     L.insert(L.end(),*LL.begin());
24.     LL.erase(LL.begin());
25.   }
26.   while (!L1.empty()) {
27.     L.insert(L.end(),*L1.begin());
28.     L1.erase(L1.begin());
29.   }
30.   while (!L2.empty()) {
31.     L.insert(L.end(),*L2.begin());
32.     L2.erase(L2.begin());
33.   }
34. }
35.
36. template<class T>
```

```
37.  void merge_sort(std::list<T> &L) {
38.    merge_sort(L,less<T>);
```

Código 5.20: *Algoritmo de ordenamiento por fusión. [Archivo: mergesort.h]*

- Para merge-sort hemos elegido una signatura diferente a la que hemos utilizado hasta ahora y que es usada normalmente en las STL para vectores. `merge_sort()` actúa directamente sobre la lista y no sobre un rango de iteradores. Sin embargo, sería relativamente fácil adaptarla para un rango usando la función `splice()` para extraer el rango en una lista auxiliar, ordenarla y volverla al rango original. Estas operaciones adicionales de `splice()` serían $O(1)$ de manera que no afectarían el costo global del algoritmo.
- El lazo de las líneas 7–12 realiza la separación en las listas auxiliares. Mantenemos una bandera lógica `flag` que va tomando los valores `true`/`false` alternadamente. Cuando `flag` es `true` extraemos un elemento de `L` y lo insertamos en `L1`. Si es `false` lo insertamos en `L2`.
- Notar el uso de la *referencia a lista* `LL` para evitar la duplicación de código. Dependiendo de `flag`, la referencia `LL` *"apunta"* a `L1` o `L2` y después la operación de pasaje del elemento de `L` a `L1` o `L2` se hace via `LL`.
- A continuación `merge_sort()` se aplica recursivamente para ordenar cada una de las listas auxiliares.
- El código de las líneas 17–33 realiza la fusión de las listas. El código es muy similar a `set_union()` para conjuntos implementados por listas ordenadas (ver sección §4.3.0.2). Una diferencia es que aquí si hay elementos duplicados no se eliminan, como se hace con conjuntos.
- En la fusión se vuelve a utilizar una referencia a lista para la duplicación de código.

5.6.2. Estabilidad

Es fácil implementar la etapa de fusión (*"merge"*) en forma estable, basta con tener cuidado al elegir el primer elemento de `L1` o `L2` en la línea 22 cuando ambos elementos son equivalentes. Notar que, de la forma como está implementado allí, cuando ambos elementos son equivalentes se elige el de la lista `L1`, lo cual es estable. Si reemplazáramos por

```
std::list<T> &LL =
   (comp(*L1.begin(),*L2.begin()) ? L1 : L2);
```

entonces en caso de ser equivalentes estaríamos tomando el elemento de la lista **L2**. Por supuesto, ambas implementaciones serían equivalentes si no consideramos la estabilidad.

Entonces, si implementamos la fusión en forma estable y asumimos que las etapas de ordenamiento serán (recursivamente) estables, sólo falta analizar la etapa de split. Puede verse que la etapa de split, tal cual como está implementada aquí (split par/impar) es inestable. Por ejemplo, si nos concentramos en la figura 5.21, de los dos 1's que hay en la lista, queda antes el segundo (que estaba originalmente en la posición 6. Esto se debe a la etapa de split, donde el primer uno va a la lista **L2**, mientras que el segundo va a la lista **L1**.

5.6.3. Versión estable de split

```
1.    int size = L.size();
2.    if (size==1) return;
3.    std::list<T> L1,L2;
4.    int n1 = size/2;
5.    int n2 = size-n1;
6.    for (int j=0; j<n1; j++) {
7.     L1.insert(L1.end(),*L.begin());
8.     L.erase(L.begin());
9.    }
10.   for (int j=0; j<n2; j++) {
11.    L2.insert(L2.end(),*L.begin());
12.    L.erase(L.begin());
13.   }
```

Código 5.21: *Versión estable de split. [Archivo: stabsplit.h]*

Se puede modificar fácilmente la etapa de split de manera que sea estable, basta con poner los primeros $\mathrm{floor}(n/2)$ elementos en **L1** y los restantes $\mathrm{ceil}(n/2)$ en **L2**, como se muestra en el código 5.21, el esquema gráfico de ordenamiento de un vector aleatorio de ocho elementos puede verse en la figura 5.22. Notar que ahora en ningún momento el camino de los 1's se cruzan entre sí.

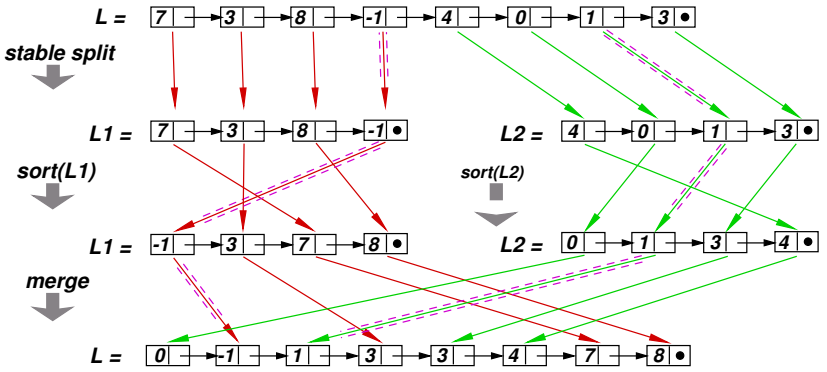


Figura 5.22: Versión estable de `merge_sort()`

5.6.4. Merge-sort para vectores

```
1.  template<class T> void
2.  merge_sort(typename std::vector<T>::iterator first,
3.         typename std::vector<T>::iterator last,
4.         typename std::vector<T> &tmp,
5.         bool (*comp)(T&,T&)) {
6.    int
7.      n = last-first;
8.    if (n==1) return;
9.    int n1 = n/2, n2 = n-n1;
10.   typename std::vector<T>::iterator
11.     middle = first+n1,
12.     q = tmp.begin(),
13.     q1 = first,
14.     q2 = first+n1;
15.
16.   merge_sort(first,middle,tmp,comp);
17.   merge_sort(first+n1,last,tmp,comp);
18.
19.   while (q1!=middle && q2!=last) {
20.     if (comp(*q2,*q1)) *q++ = *q2++;
21.     else *q++ = *q1++;
22.   }
23.   while (q1!=middle) *q++ = *q1++;
24.   while (q2!=last) *q++ = *q2++;
25.
26.   q1=first;
27.   q = tmp.begin();
28.   for (int j=0; j<n; j++) *q1++ = *q++;
```

```
29.  }
30.
31.  template<class T> void
32.  merge_sort(typename std::vector<T>::iterator first,
33.         typename std::vector<T>::iterator last,
34.            bool (*comp)(T&,T&)) {
35.   std::vector<T> tmp(last-first);
36.   merge_sort(first,last,tmp,comp);
37.  }
38.
39.  template<class T> void
40.  merge_sort(typename std::vector<T>::iterator first,
41.             typename std::vector<T>::iterator last) {
42.   merge_sort(first,last,less<T>);
43.  }
```

Código 5.22: *Implementación de merge-sort para vectores con un vector auxiliar. [Archivo: mergevec.h]*

Es muy simple implementar una versión de merge-sort para vectores, si se usa un vector auxiliar, es decir no *in-place*. El proceso es igual que para listas, pero al momento de hacer la fusión, esta se hace sobre el vector auxiliar. Este vector debe ser en principio tan largo como el vector original y por una cuestión de eficiencia es creado en una función *"wrapper"* auxiliar y pasada siempre por referencia.

- Para vectores no es necesario hacer explícitamente el *split* ya que basta con pasar los extremos de los intervalos. Es decir, la operación de split es aquí un simple cálculo del iterator **`middle`**, que es $O(1)$.
- El vector auxiliar **`tmp`** se crea en la línea 35. Este vector auxiliar es pasado a una función recursiva **`merge_sort(first,last,tmp,comp)`**.
- El intercalamiento o fusión se realiza en las líneas 19–24. El algoritmo es igual que para listas, pero los elementos se van copiando a elementos del vector **`tmp`** (el iterator **`q`**).
- El vector ordenado es recopiado en **`[first,last)`** en las líneas 26.

Esta implementación es estable, $O(n \log n)$ en el peor caso, pero no es in-place. Si relajamos la condición de estabilidad, entonces podemos usar el algoritmo de intercalación discutido en la sección §2.3.1. Recordemos que ese algoritmo es $O(n)$ y no es in-place, pero requiere de menos memoria adicional, $O(\sqrt{n})$ en el caso promedio, $O(n)$ en el peor caso, en comparación con el algoritmo descripto aquí que que requiere $O(n)$ siempre,

5.6.5. Ordenamiento externo

```
1.  void merge_sort(list<block> &L,bool (comp*)(T&,T&)) {
2.    int n = L.size();
3.    if (n==1) {
4.      // ordenar los elementos en el unico bloque de
5.      // 'L' . . . .
6.    } else {
7.      list<T> L1,L2;
8.      // Separacion: separar L en dos sublistas de
9.      // tamano similar 'L1' y 'L2' . . .
10.     int
11.       n1 = n/2,
12.       n2 = n-n1;
13.     list<block>::iterator
14.       p = L.begin(),
15.       q = L1.begin();
16.     for (int j=0; j<n1; j++)
17.       q = L1.insert(q,*p++);
18.
19.     q = L2.begin();
20.     for (int j=0; j<n2; j++)
21.       q = L2.insert(q,*p++);
22.
23.     // Sort individual:
24.     merge_sort(L1,comp);
25.     merge_sort(L2,comp);
26.
27.     // Fusion: concatenar las listas
28.     // 'L1' y 'L2' en 'L' . . .
29.   }
30. }
```

Código 5.23: *Algoritmo para ordenar bloques de dato. Ordenamiento externo. [Archivo: extsortsc.cpp]*

De todos los algoritmos de ordenamiento vistos, merge-sort es el más apropiado para ordenamiento externo, es decir, para grandes volúmenes de datos que no entran en memoria principal. Si tenemos n objetos, entonces podemos dividir a los n objetos en m bloques de $b = n/m$ objetos cada uno. Cada bloque se almacenará en un archivo independiente. El algoritmo procede entonces como se muestra en el código 5.23.

- A diferencia del merge-sort de listas, cuando la longitud de la lista se reduce a uno, esto quiere decir que la lista tiene un sólo bloque, no un solo elemento, de manera que hay que ordenar los elementos del bloque entre sí. Esto se puede hacer cargando todos los elementos del bloque en un vector y ordenándolos con algún algoritmo de ordenamiento interno.
- Además, cuando se hace la fusión de las listas de bloques en la línea 28 se debe hacer la fusión elemento a elemento (no por bloques).
- Por supuesto las operaciones sobre los bloques deben ser implementadas en forma *"indirecta"*, es decir sin involucrar una copia explícita de los datos. Por ejemplo, si los bloques son representados por archivos entonces podríamos tener en las listas los nombres de los archivos.

Merge-sort externo es estable si el algoritmo de ordenamiento para los bloques es estable y si la fusión se hace en forma estable.

En el código **extsort.cpp**, que acompaña este libro se ha implementado el algoritmo descripto. El algoritmo genera un cierto número **Nb** de bloques de **M** enteros. Para ordenar cada bloque (archivo) se ha usado el **sort()** de STL para vectores. El **sort()** de STL no es estable, de manera que esta implementación tampoco lo es, pero de todas formas estamos ordenando enteros por $<$, que es una relación de orden fuerte, de manera que la estabilidad no es relevante.

5.7. Comparación de algunas implementaciones de algoritmos de ordenamiento

En la figura 5.23 vemos una comparación de varias implementaciones de algoritmos de ordenamiento.

- **STL[vec]** es la versión de sort que viene en el header **<algorithm>** de **C++** estándar con la versión de **g++** usada en este libro.
- **merge-sort[vec,st]** es la versión de merge-sort estable para vectores (no in-place) descripta en la sección §5.6.4.
- **libc-sort** es la rutina de ordenamiento que viene con el compilador **gcc** y que forma parte de la librería estándar de C (llamada **libc**).
- **heap-sort** es el algoritmo de ordenamiento por montículos implementado en este libro.
- **quick-sort[st]** es el algoritmo de ordenamiento rápido, en su versión estable descripta en la sección §5.4.10.

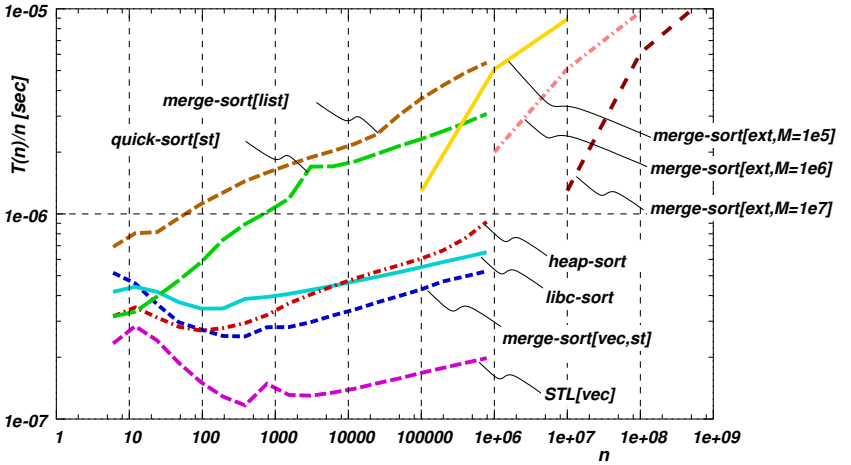


Figura 5.23: Tiempos de ejecución para varios algoritmos de ordenamiento.

- **`merge-sort[list]`** Es la implementación de merge-sort para listas descripta en la sección §5.6.1.
- **`merge-sort[ext,M=]`** es la versión de merge-sort externa descripta en la sección §5.6.5. Se han hecho experimentos con varios valores del tamaño del bloque **`M`**.

Podemos hacer las siguientes observaciones

- Para resaltar las diferencias entre los diferentes algoritmos se ha graficado en las ordenadas $T(n)/n$. Para un algoritmo estrictamente lineal (es decir $O(n)$) este valor debería ser constante. Como todos los algoritmos genéricos son, en el mejor de los casos, $O(n \log n)$ se observa un cierto crecimiento. De todas formas este cociente crece a los sumo un factor 10 o menos en 5 órdenes de magnitud de variación de n.
- Para algunos algoritmos se observa un decrecimiento para valores bajos de n (entre 10 y 100). Esto se debe a ineficiencias de las implementaciones para pequeños valores de n.
- El algoritmo de ordenamiento interno más eficiente de los comparados resulta ser la versión que viene con las STL. Esto puede deberse a que la implementación con template puede implementar *"inline"* las funciones de comparación.
- Merge-sort para vectores resulta ser muy eficiente (recordemos que además es estable, pero no es in-place). Sin embargo, para listas resulta ser notablemente más lento. Esto se debe sobre todo a que en

general la manipulación de listas es más lenta que el acceso a vectores.

- La versión estable de quick-sort es relativamente ineficiente, sin embargo es el único algoritmo rápido, estable e *in-place* de los discutidos (recordemos que es $O(n(\log n)^2)$).
- Los algoritmos de ordenamiento interno se ha usado hasta $n = 10^6$. El *merge-sort* se ha usado hasta cerca de $n = 10^9$. Notar que a esa altura el tamaño de los datos ordenados es del orden de 4 GBytes.
- El algoritmo de ordenamiento externo parece tener una velocidad de crecimiento mayor que los algoritmos de ordenamiento interno. Sin embargo esto se debe a la influencia del tamaño del bloque usado. Para valores de n mucho mayores que M el costo tiende a desacelerarse y debería ser $O(n \log n)$.

Indice alfabético

Bibliografía

A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison Wesley, 1987.

Free Software Foundation. *GNU Compiler Collection GCC manual*, a. GCC version 3.2, http://www.gnu.org.

Free Software Foundation. *The GNU C Library Reference Manual*, b. Version 2.3.x of the GNU C Library, edition 0.10, http://www.gnu.org/software/libc/libc.html.

R. Hernández, J.C. Lázaro, R. Dormido, and S. Ros. *Estructuras de Datos y Algoritmos*. Prentice Hall, 2001.

D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1981.

SGI. *Standard Template Library Programmer's Guide*, 1999. http://www.sgi.com/tech/stl.