Estructura de datos en C++

Luis Joyanes Aguilar
Lucas Sánchez García
Ignacio Zahonero Martínez

Mc Graw

ESTRUCTURA DE DATOS EN C++

Luis Joyanes
Ignacio Zahonero
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Sofware
Facultad de Informática, Escuela Universitaria de Informática
Universidad Pontificia de Salamanca *campus* Madrid

MADRID • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEV A DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • SAN LOUIS • TOKIO • TORONTO

ESTRUCTURAS DE DATOS EN C++

McGraw-Hill/Interamericana de de España, S. A. U.

www.mcgraw-hill.es

ISBN: 978-84-481-5645-9
Depósito legal: M.

Editor: Carmelo Sánchez González
Técnico editorial: Israel Sebastián
Diseño de cubierta: Luis Sanz Cantero
Compuesto por: Gráficas Blanco, S. L.
Impreso en

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Contenido

Prólogo

Dos de las disciplinas clásicas en todas las carreras relacionadas con la Informática y las Ciencias de la Computación son: *Estructuras de Datos* y *Algoritmos* o bien una sola disciplina, si ambas se estudian integradas en *Algoritmos y Estructuras de Datos.* El estudio de estructuras de datos y de algoritmos es tan antiguo como el nacimiento de la programación y se ha convertido en estudio obligatorio en todos los currículos desde finales de los años sesenta y sobre todo en la década de los setenta cuando apareció el Lenguaje Pascal de la mano del profesor suizo Niklaus Wirtz, y posteriormente en la década de los ochenta con la aparición de su obra —ya clásica— *Algorithms and Data Structures* en 1986. ***Estructuras de Datos en C++*** trata sobre el estudio de las estructuras de datos dentro del marco de trabajo de los tipos abstractos de datos (**TAD**) y objetos, bajo la óptica del análisis, diseño de algoritmos y programación, realizando las implementaciones de los algoritmos en C++.

C++ es un *superconjunto* y una extensión de C, tópico conocido por toda la comunidad de programadores del mundo. Cabe preguntarse como hacen muchos autores, profesores, alumnos y profesionales: *¿Se debe aprender primero C y luego C++?* Stroustrup —creador y padre de C++— junto con una gran mayoría de programadores contesta así: "No sólo no es innecesario aprender primero C, sino que además es una mala idea". Nosotros no somos tan radicales y pensamos que se puede llegar a C++ procediendo de ambos caminos. En el caso de un libro de Estructuras de Datos como el que Vd. tiene en sus manos, la problemática es la misma, por lo que se puede aprender a analizar y diseñar estructuras de datos directamente desde C++ . Pero en cualquier forma y en apoyo de nuestra teoría anterior, hemos introducido en los primeros capítulos los conceptos básicos necesarios para seguir el contenido de la obra tanto si usted ya es programador de C++ como si procediese de C y no tuviese esa formación específica en C++.

¿Porqué en C++ y no en Java, porqué no en C/C++ o en Visual Basic/C# ? Muchas Facultades y Escuelas de Ciencias y de Ingeniería, así como Institutos Tecnológicos y Centros de Formación Profesional, comienzan sus cursos de Estructuras de Datos con el soporte de C y muchas otras con el soporte de C++ o Java. De hecho, en nuestra propia universidad, en asignaturas relacionadas con esta disciplina se aprende a diseñar y construir estructuras de datos utilizando C, C++ o Java, a veces, indistintamente. ¿Existe una solución ideal? Evidentemente, consideramos que no y cada una de ellas tiene sus ventajas y sus inconvenientes, y es la decisión del maestro y profesor, responsable de su formación, quien debe elegir aquella que considera más recomendable para sus alumnos teniendo en cuenta el entorno y contexto donde se desarrolla su labor, ya que siempre pensará en su mejor futuro y por esta razón siempre la encajará dentro del currículo específico de su carrera en el lugar que considere más oportuno.

El primer problema que se suele presentar al estudiante de *algoritmos y estructuras de datos* que, probablemente, procederá de un curso de nivel básico, medio o avanzado de *introducción o fundamentos de programación* o bien de *iniciación de algoritmos*, es precisamente el modo de afrontar información compleja desde el principio. Al permitir C++ el empleo de los dos paradigmas clásicos de programación: *procedimental* y *orientada a objetos*, el aprendizaje de la disciplina le será más fácil ya que podrá adaptarse en función de su formación. Pensando en aquellos lectores que no hayan seguido ningún curso de programación orientada a objetos (POO) y dado que la POO es una herramienta de programación y organización muy potente y con grandes ventajas para la enseñanza y posterior tarea profesional, se han incluido capítulos específicos de clases, objetos, clases derivadas, herencia, polimorfismo y plantillas, que pese a no conformar un curso de introducción a POO si se han escrito pensando en servir de fundamentos básicos a modo de introducción, recordatorio o nueva formación.

Por otra parte, la mayoría de los estudiantes de informática, ciencias de la computación, ingeniería de sistemas o de telecomunicaciones, o cualesquiera otra carrera de ciencias o ingeniería, o de estudios de formación profesional de nivel superior, requieren conocer bien el flujo C-C++ y viceversa. El enfoque orientado a objetos permite la implementación de las estructuras de datos con clases y objetos aportando a las estructuras de datos su verdadera potencialidad, si bien hay que reconocer que el paradigma estructurado para aprender la idea de algoritmo y estructuras de datos está muy extendido entre múltiples profesores, alumnos, etc. El libro está soportando la comprensión del tipo abstracto de datos (TAD) con un estilo que permite la formación de las estructuras de datos orientadas a objetos.

Además de estas ventajas, existen otras, que si bien se pueden considerar menores, no por ello menos importantes y son de gran incidencia en la formación en esta materia. Por ejemplo, algunas de las funciones de Entrada/Salida (tan importantes en programación) son más fáciles en C++ que en C (véase el caso de números enteros), otros tipos de datos tales como cadenas y números reales se pueden formatear más fácilmente en C. Otro factor importante para los principiantes es el conjunto de mensajes de error y advertencias proporcionadas por un compilador durante el desarrollo del programa.

Se estudian estructuras de datos con un objetivo fundamental: aprender a escribir programas más eficientes. También cabe aquí hacerse la pregunta ¿Por qué se necesitan programas más eficientes cuando las nuevas computadoras son más rápidas cada año (en el momento de escribir este prólogo, las frecuencias de trabajo de las computadoras personales domésticas son de 3 GHz o superiores —o bien 1,6 a 1,8 GHz en el caso de procesadores de doble núcleo tanto de AMD como de Intel—, y las memorias centrales de 1 GB a 4 GB, son prácticamente usuales en la mayoría de las PCs y claro está son el nivel de partida en profesionales). La razón tal vez resida en el hecho de que nuestras metas no se amplían a medida que se aumentan las características de las computadoras. La potencia de cálculo y las capacidades de almacenamiento aumentan la eficacia y ello conlleva un aumento de los resultados de las máquinas y de los programas desarrollados para ellas.

La búsqueda de la eficiencia de un programa no debe chocar con un buen diseño y una codificación clara y legible. La creación de programas eficientes tiene poco que ver con "trucos de programación" sino al contrario se basan en una buena organización de la información y buenos algoritmos. Un programador que no domine los principios básicos de diseños claros y limpios probablemente no escribirá programas eficientes. A la inversa, programas claros requieren organizaciones de datos claras y algoritmos claros, precisos y transparentes.

La mayoría de los departamentos informáticos reconocen que las destrezas de buena programación requieren un fuerte énfasis en los principios básicos de ingeniería de software. Por consiguiente, una vez que un programador ha aprendido los principios para diseñar e imple-

mentar programas claros y precisos, el paso siguiente es estudiar los efectos de las organizaciones de datos y los algoritmos en la eficiencia de un programa.

EL ENFOQUE DEL LIBRO

En esta obra se muestran numerosas técnicas de representación de datos. El contexto de las mismas se engloban en los siguientes principios:

1. Cada estructura de datos tiene sus costes y sus beneficios. Los programadores y diseñadores necesitan una comprensión rigurosa y completa de cómo evaluar los costes y beneficios para adaptarse a los nuevos retos que afronta la construcción de la aplicación. Estas propiedades requieren un conocimiento o comprensión de los principios del análisis de algoritmos y también una consideración práctica de los efectos significativos del medio físico empleado (p.e. datos almacenados en un disco frente a memoria principal).
2. Los temas relativos a costes y beneficios se consideran dentro del concepto de elemento de compensación. Por ejemplo, es bastante frecuente reducir los requisitos de tiempo en beneficio de un incremento de requisitos de espacio en memoria o viceversa.
3. Los programadores no deben reinventar la rueda continuamente. Por consiguiente, los estudiantes necesitan aprender las estructuras de datos utilizadas junto con los algoritmos correspondientes.
4. Los datos estructurados siguen a las necesidades. Los estudiantes deben aprender a evaluar primero las necesidades de la aplicación, a continuación, encontrar una estructura de datos en correspondencia con sus funcionalidades.

Esta edición, fundamentalmente, describe *estructuras de datos,* métodos de organización de grandes cantidades de datos y *algoritmos* junto con el *análisis de los mismos,* en esencia estimación del tiempo de ejecución de algoritmos. A medida que las computadoras se vuelven más y más rápidas, la necesidad de programas que pueden manejar grandes cantidades de entradas se vuelve más críticas y su eficiencia aumenta a medida que estos programas pueden manipular más y mejores organizaciones de datos. Analizando un algoritmo antes de que se codifique realmente, los estudiantes pueden decidir si una determinada solución será factible y rigurosa. Por ejemplo se pueden ver cómo diseños e implementaciones cuidadas pueden reducir los costes en tiempo y memoria de algoritmos. Por esta razón, se dedica un capítulo en exclusiva a tratar los conceptos fundamentales de *análisis de algoritmos*, y en un gran número de algoritmos se incluyen explicaciones de tiempos de ejecución para poder medir la complejidad y eficiencia de los mismos.

El método didáctico que sigue nuestro libro ya lo hemos seguido en otras obras nuestras y busca preferentemente enseñar al lector a pensar en la resolución de un problema siguiendo un determinado método ya conocido o bien creado por el propio lector, una vez esbozado el método, se estudia el algoritmo correspondiente junto con las etapas que pueden resolver el problema. A continuación se escribe el algoritmo, en ocasiones en *pseudocódigo* que al ser en español facilitará el aprendizaje al lector, y siempre en C++ para que el lector pueda verificar su programa antes de introducirlos en la computadora; se incluyen a veces la salida en pantalla resultante de la ejecución correspondiente en la máquina.

Uno de los objetivos fundamentales del libro es enseñar al estudiante, simultáneamente, buenas reglas de programación y análisis de algoritmos de modo que puedan desarrollar los programas con la mayor eficiencia posible.

EL LIBRO COMO TEXTO DE REFERENCIA UNIVERSITARIA Y PROFESIONAL

El estudio de *Algoritmos* y de *Estructuras de Datos* son disciplinas académicas que se incorporan a todos los planes de estudios universitarios de Ingeniería e Ingeniería Técnica en Informática, Ingeniería de Sistemas Computacionales y Licenciaturas en Informática, así como a los planes de estudio de Informática en Formación Profesional y en institutos politécnicos. Suele considerarse también a estas disciplinas como ampliaciones de las asignaturas de Programación, en cualquiera de sus niveles.

En el caso de España, los actuales planes de estudios y los futuros —contemplados en la Declaración de Bolonia (EEES, Espacio Europeo de Educación Superior)—, de Ingeniería Técnica en Informática e Ingeniería Informática, contemplan materias troncales relativas tanto a Algoritmos como a Estructuras de Datos. Igual sucede en los países iberoamericanos donde también es común incluir estas disciplinas en los *currículum* de carreras de Ingeniería de Sistemas y Licenciaturas en Informática. ACM, la organización profesional norteamericana más prestigiosa a nivel mundial, incluye en las recomendaciones de sus diferentes currículos y carreras relacionadas con informática el estudio de materias de algoritmos y estructuras de datos. En el conocido *Computing Curricula* de 1992 se incluyen descriptores recomendados de *Programación y Estructura de Datos* , y en los últimos currículo publicados, *Computing Curricula* 2001 y 2005, se incluyen en el área **PF** de *Fundamentos de Programación (Programming Fundamentals, PF1 a PF4),* **AL** de *Algoritmos y Complejidad (Algorithms and Complexity, AL1 a AL3).* En este libro se ha incluido los descriptores más importantes tales como *Algoritmos y Resolución de Problemas, Estructuras de datos fundamentales, Recursión, Análisis de algoritmos básicos y estrategias de algoritmos.* Además se incluyen un estudio de algoritmos de estructuras discretas tan importantes como *Árboles y Grafos.*

ORGANIZACIÓN DEL LIBRO

El libro está concebido como libro didáctico y teórico pero con un enfoque muy práctico, por lo que se incluyen gran número de ejemplos y ejercicios resueltos. Se pretende enseñar los principios básicos requeridos para seleccionar o diseñar los algoritmos y las estructuras de datos que ayudarán a resolver mejor los problemas que no a memorizar una gran cantidad de implementaciones. Los lectores deben tener conocimientos a nivel de iniciación o nivel medio en programación. Es deseable, haber cursado al menos un curso de un semestre de introducción a los algoritmos y a la programación, con ayuda de alguna herramienta de programación, preferentemente en lenguaje C++, pero podría bastar un curso de introducción a los algoritmos y programación; pensando en estos lectores en la página web oficial del curso podrá encontrar guías didácticas de introducción al lenguaje C++. El libro busca de modo prioritario enseñar al lector técnicas de programación de algoritmos y estructuras de datos. Se pretende aprender a programar practicando el análisis de los problemas y su codificación en C++.

El libro está pensado para un curso completo anual o bien dos semestres, para ser estudiado de modo independiente —por esta razón se incluyen las explicaciones y conceptos básicos de la teoría de algoritmos y estructuras de datos— o bien de modo complementario, exclusivamente como apoyo de otros libros de teoría o simplemente del curso impartido por el maestro o profesor en su aula de clase. Para aquellos lectores que deseen contrastar o comparar el diseño de algoritmos y estructuras de datos en otros lenguajes tales como Pascal y C, desde un enfoque práctico, le enumeramos otras obras nuestras: Joyanes, L.; Centenera, P.; Sánchez, L.,

Zahonero I.; Fernández, M. *Estructuras de datos. Libro de problema*s. McGraw-Hill, 1999, con un enfoque en Pascal, Joyanes, L.; Sánchez, L.; Zahonero, I.; Fernández, M. *Estructuras de datos en C.* McGraw-Hill/Schaum, 2005, con un enfoque en C).

Contenido

El contenido del libro sigue los programas clásicos de las disciplinas *Estructura de Datos y/o Estructuras de Datos y de la Información* respetando las directrices emanadas de los currículo del 91 y las actualizadas del 2001 y 2005 de ACM/IEEE, así como de los planes de estudio de Ingeniero Informático e Ingenieros Técnicos en Informática de España y los de Ingenieros de Sistemas y Licenciados en Informática de muchas universidades latinoamericanas.

Aunque no se ha realizado una división formal del libro en partes, se puede considerar desde el punto de vista práctico que su contenido se agrupa en cuatro grandes partes. La Parte I: ***Abstracción y desarrollo del software con C++*** presenta los importantes conceptos de algoritmo, tipo abstracto de datos, clases, objetos, plantillas (templates) y *genericidad*, y describe las estructuras de datos más simples tales como los *arrays (arreglos)* cadenas o estructuras. La Parte II: ***Análisis y diseño de algoritmos (recursividad, ordenación y búsqueda)*** describe el importante concepto de análisis de un algoritmo y las diferentes formas de medir su complejidad y eficiencia examina los algoritmos más utilizados en la construcción de cualquier programa tales como los relativos a búsqueda y ordenación, así como las potentes técnicas de manipulación de la recursividad. La Parte III: ***Estructuras de datos lineales (abstracción de datos, listas, pilas, colas, colas de prioridada, tablas hash, la biblioteca STL, contenedores e iteradores)*** constituyen una de las partes avanzadas del libro y que suele formar parte de cursos de nivel medio/alto en organización de datos. Por último, la Parte IV: ***Estructuras de datos no lineales (árboles, grafos y sus algoritmos)*** constituyen también una de las partes avanzadas del libro; su conocimiento y manipulación permitirán al programador obtener el máximo aprovechamiento en el diseño y construcción de sus programas. La descripción más detallada de los capítulos correspondientes se reseñan a continuación:

Capítulo 1. Desarrollo de *software*. Tipos abstractos de datos. Los tipos de datos y necesidad de su organización en estructuras de datos es la parte central de este capítulo. El tratamiento de la abstracción de datos, junto con el reforzamiento de los conceptos de algoritmos y programas, y su herramienta de representación más característica, el *pseudocódigo* completa el capítulo.

Capítulo 2. Clases y objetos. La programación orientada a objetos es hoy día, el eje fundamental de la programación de computadoras. Su núcleo esencial son los conceptos de clases y objetos. En el capítulo se consideran los conceptos teóricos de encapsulación de datos y tipos abstractos de datos como soporte de una clase y de un objeto; también se analizan el modo de construcción y destrucción de objetos, así como conceptos tan importantes como las funciones amiga y los miembros estáticos de una clase.

Capítulo 3. Tipos de datos básicos. Arrays, cadenas, estructuras y tipos enumerados. Se revisan en este capítulo los conceptos básicos de tipos de datos como fundamentos para el diseño y construcción de las clases explicadas en el Capítulo 2. Los diferentes tipos de *arrays (arreglos)* se describen y detallan junto con la introducción a la importante clase `string`.

Capítulo 4. Clases derivadas: herencia y polimorfismo. Uno de los conceptos más empleados en programación orientada a objetos y que ayudará al programador de un modo eficiente al diseño de estructura de datos son las clases derivadas. La propiedad de herencia, junto con el polimorfismo ayudan a definir con toda eficacia las clases derivadas. Otro término fundamental en POO son las clases abstractas que permitirán la construcción de clases deriva-

das. C++, es uno de los pocos lenguajes de programación orientada a objetos que soporta herencia simple y herencia múltiple, aunque en este caso particular el lector deberá estudiar con detenimiento este concepto ya que a sus grandes posibilidades también se pueden añadir grandes problemas de diseño.

Capítulo 5. Genericidad: plantillas *(templates)*. Una de las propiedades más destacadas de C++ es el soporte de la *genericidad* y, por consiguiente, la posibilidad de ejecutar programación genérica. La definición y buen uso de las plantillas (`templates`) es uno de los objetivos de este capítulo. Diferenciar y hacer buen uso de las plantillas de clases y las plantillas de funciones son otro de los objetivos de este capítulo.

Capítulo 6. Análisis y eficiencias de algoritmos. El estudio de algoritmos es uno de los objetivos más ambiciosos de esta obra. El capítulo hace una revisión de sus propiedades más importantes; representación, eficiencia y exactitud. Se describe la notación *O grande* utilizada preferentemente en el análisis de algoritmos.

Capítulo 7. Algoritmos recursivos. La recursividad es una de las características más sobresalientes en cualquier tipo de programación, Los algoritmos recursivos abundan en la vida ordinaria y el proceso de abstracción que identifica estos algoritmos debe conducir a un buen diseño de dichos algoritmos. Los algoritmos más sobresalientes y de mayor difusión en el mundo de la programación se explican con detalle en el capítulo. Así se describen algoritmos como: *mergesort* (ordenación por mezclas), *backtracking* (vuelta atrás) y otros.

Capítulo 8. Algoritmos de ordenación y búsqueda. Las operaciones más frecuentes en el proceso de estructura de datos, son: la ordenación y búsqueda de datos específicos. Los algoritmos más populares y eficientes de proceso de estructuras de datos internas se describen en el capítulo junto con un análisis de su complejidad.

Capítulo 9. Algoritmos de ordenación de archivos. Los archivos (ficheros) de datos son, posiblemente, las estructuras de datos más diseñadas y utilizadas por los programadores de aplicaciones y programadores de sistemas. Los conceptos de flujos y archivos de C++ junto con los métodos clásicos y eficientes de ordenación de archivos se describen en profundidad en el capítulo.

Capítulo 10. Listas. Una lista enlazada es una estructura de datos lineal de gran uso en la vida diaria de las personas y de las organizaciones. Su implementación mediante listas enlazadas es el objetivo central de este capítulo. Variantes de las listas enlazadas simples como doblemente enlazadas y circulares, son también , motivo de estudio en el capítulo.

Capítulo 11. Pilas. La pila es una estructura de datos simple, y cuyo concepto forma también parte, como las listas, en un elevado porcentaje, de la vida diaria de las personas y organizaciones. El tipo de dato *Pila* se puede implementar con arrays o con listas enlazadas y describe ambos algoritmos y sus correspondientes implementaciones en C++.

Capítulo 12. Colas . Al igual que las pilas, las colas conforman otra estructura que abunda en la vida ordinaria. La implementación del **TAD** *(Tipo Abstracto de Dato)* cola se puede hacer con *arrays* (arreglos), listas enlazadas e incluso listas circulares. Así mismo se analiza también en el capitulo el concepto de *bicola* o cola de doble entrada..

Capítulo 13. Cola de prioridades y montículos. Un tipo especial de cola, la cola de prioridades, utilizado en situaciones especiales, para resolución de problemas, junto con el concepto de montículo (*heap*, en inglés) se analizan detalladamente, junto con un método de ordenación por montículos muy eficiente, sobre todo en situaciones complejas y difíciles. Asimismo se analiza en el capitulo el concepto de montículo binomial.

Capítulo 14. Tablas de dispersión: Funciones *hash*. Las tablas aleatorias *hash* junto con los problemas de resolución de colisiones y los diferentes tipos de direccionamiento conforman este capítulo.

Capítulo 15. Biblioteca estándar de plantillas STL. El importante concepto de biblioteca de plantillas de clases se estudia en este capítulo. En particular, la biblioteca **STL** de C++. Los *contenedores* e *iteradores* son dos términos importantes para la programación genérica con plantillas y su conocimiento y diseño son muy importantes en la formación del programador.

Capítulo 16. Árboles. Árboles binarios y árboles ordenados. Los árboles son estructuras de datos no lineales y jerárquicas muy notables. Estas estructuras son notablemente de gran importancia en programación avanzada. Los árboles binarios y los árboles binarios de búsqueda se describen con rigor y profundidad por su importancia en el mundo actual de la programación tanto tradicional *(fuera de línea)* como en la Web *(en línea)*.

Capítulo 17. Árboles de búsqueda equilibrados. Árboles B. Este capítulo se dedica a la programación avanzada de árboles de búsqueda equilibrada y árboles B. Estas estructuras de datos son complejas y su diseño y construcción requiere de estrategias y métodos eficientes para su implementación; sin embargo su uso puede producir grandes mejoras al diseño y construcción de programas que sería muy difícil por otros métodos.

Capítulo 18. Grafos. Los grafos son una de las herramientas más empleadas en matemáticas, estadística, investigación operativa y en numerosos campos científicos. El estudio de la teoría de Grafos se realiza fundamentalmente como elemento de *Matemática Discreta* o *Matemática Aplicada.* El conocimiento profundo de la teoría de grafos junto con los algoritmos de implementación es fundamental para conseguir el mayor rendimiento de las operaciones con datos, sobre todo si estos son complejos en su organización. Un programador de alto nivel no puede dejar de conocer en toda su profundidad la teoría de grafos y sus operaciones

Los **Anexos A**, **B**, **C** y **D** los puede encontrar el lector en la página web oficial del libro y forman parte de *Lecturas recomendadas* a todos aquellos lectores que deseen profundizar en programación avanzada de Árboles y Grafos.

Anexo A. Eliminación de árboles AVL (*Lectura recomendada del Capítulo 17*).

Anexo B. Eliminación de árboles B (*Lectura recomendada del Capítulo 17*).

Anexo C. Listas de adyacencia. Puntos de articulación de un grafo (*Lectura recomendada del Capítulo 18*).

Anexo D. Grafos para redes de flujo (*Lectura recomendada del Capítulo 18*).

CÓDIGO C++ DISPONIBLE

Los códigos en C++ C de todos los programas de este libro están disponibles en la Web (Internet) —en formato Word para que puedan ser utilizados directamente y evitar su "tecleado" en el caso de los programas largos, o bien simplemente, para seleccionar, recortar, modificar… por el lector a su conveniencia, a medida que avanza en su formación—. Estos códigos fuente se encuentran en la página oficial del libro `http://www.mch.es/joyanes`.

AGRADECIMIENTOS

Muchos profesores y colegas españoles y latinoamericanos nos han alentado a escribir esta obra, continuación/complemento de nuestra antiguas y todavía disponibles en librería, *Estructura de Datos* cuyo enfoque era en el clásico lenguaje Pascal y Algoritmos y estructuras de datos una perspectiva en C. A todos ellos queremos mostrarles nuestro agradecimiento y como siempre brindarles nuestra colaboración si así lo desean.

A los muchos instructores, maestros y profesores tanto amigos como anónimos de Universidades e Institutos Tecnológicos y Politécnicos de España y Latinoamérica que siempre apoyan nuestras obras y a los que desgraciadamente nunca podremos agradecer individualmente ese apoyo; al menos que conste en este humilde homenaje, nuestro eterno agradecimiento y reconocimiento por ese cariño que siempre prestan a nuestras obras. Como saben aquellos que nos conocen siempre estamos a su disposición en la medida que, físicamente, nos es posible. Gracias a todos, esta obra es posible, en un porcentaje muy alto, por vuestra ayuda y colaboración.

Y como no, a los estudiantes, a los lectores autodidactas y no autodidactas, que siguen nuestras obras. Su apoyo es un gran acicate para seguir nuestra tarea. También gracias queridos lectores.

Pero si importantes son en esta obra, nuestros colegas y lectores españoles y latinoamericanos, no podemos dejar de citar al equipo humano que desde la editorial siempre cuida nuestras obras y sobre todo nos dan consejos, sugerencias, propuestas, nos "soportan" nuestros retrasos, nuestros "cambios" en la redacción, etc. **A Carmelo Sánchez** nuestro editor —y sin embargo amigo— de McGraw-Hill, que en esta ocasión, para no ser menos, nos ha vuelto a asesorar tanto en la fase de realización como en todo el proceso editorial hasta su publicación final. Por último a nuestro nuevo editor y amigo José Luis García Jurado que se ha hecho cargo de la fase final de la edición de este libro y que en tan breve espacio de tiempo nos ha prestado todo su apoyo, comprensión y *saber hacer.*

Madrid, marzo de 2007.

LOS AUTORES

CAPÍTULO 1

Desarrollo de *software*. Tipos abstractos de datos

Objetivos

Con el estudio de este capítulo usted podra:

- Conocer el concepto de *software* algoritmo y sistema operativo.
- Especificar algoritmos en pseudocódigo.
- Definir los tipos de abstractos de datos.
- Conocer el concepto de objeto.

Contenido

Conceptos clave

- Abstracción.
- Algoritmo.
- Clase.
- Depuración.
- Documentación.
- Estructura de datos.
- Herencia.
- Mantenimiento.
- Objetos.
- Polimorfismo.
- Programación estructurada.
- Programación orientada a objetos.
- Sistema operativo.
- *Software*.
- TAD.
- Tipo de dato.

INTRODUCCIÓN

La principal razón para que las personas aprendan lenguajes y técnicas de programación es utilizar la computadora como una herramienta para resolver problemas. Este capítulo introduce al lector en la metodología a seguir para la resolución de problemas con computadoras y en el diseño de algoritmos examinanando el concepto de *Abstracción de Datos*. La *Abstracción de Datos* es la técnica de inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, facilitar la escritura del programa. La técnica de abstracción de datos es una técnica potente de propósito general que cuando se utiliza adecuadamente, puede producir programas más cortos, más legibles y flexibles.

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como `int`, `char` y `float` en C, C++ y Java. Lenguajes de programación, como C++, tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina *tipo abstracto de dato,* **TAD** *(Abstract Data Type,* **ADT***).* El término abstracto se refiere al medio en que un programador abstrae algunos conceptos de programación creando un nuevo tipo de dato.

La modularización de un programa utiliza la noción de *tipo abstracto de dato* (**TAD**) siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado **TAD**.

Los paradigmas más populares soportados por el lenguaje C++ son: programación estructurada y programación orientada a objetos.

1.1. EL *SOFTWARE* (LOS PROGRAMAS)

El *software* de una computadora es un conjunto de instrucciones de programa detalladas que controlan y coordinan los componentes *hardware* de una computadora y controlan las operaciones de un sistema informático. El auge de las computadoras en el siglo pasado y en el actual siglo XXI, se debe esencialmente al desarrollo de sucesivas generaciones de *software* potentes y cada vez más *amistosas* («fáciles de utilizar»)

Las operaciones que debe realizar el *hardware* son especificadas por una lista de instrucciones, llamadas programas, o *software*. Un programa de *software* es un conjunto de **sentencias** o **instrucciones** dadas al computador. El proceso de escritura o codificación de un programa se denomina **programación** y las personas que se especializan es esta actividad se denominan **programadores**. Existen dos tipos importantes de *software*: *software* del sistema y *software* de *aplicaciones*. Cada tipo realiza una función diferente.

***Software* del sistema** es un conjunto generalizado de programas que gestiona los recursos del computador, tal como el procesador central, enlaces de comunicaciones y dispositivos periféricos. Los programadores que escriben *software* del sistema se llaman **programadores de sistemas. *Software* de aplicaciones** el conjunto de programas escritos por empresas o usuarios individuales o en equipo y que instruyen a la computadora para que ejecute una tarea específica. Los programadores que escriben *software* de aplicaciones se llaman **programadores de aplicaciones**.

Los dos tipos de *software* están relacionados entre sí, de modo que los usuarios y los programadores pueden hacer así un uso eficiente del computador. En la Figura 1.1 se muestra una vista organizacional de un computador donde se muestran los diferentes tipos de *software* a modo de capas de la computadora desde su interior (el *hardware*) hacia su exterior (usuario): las dife-

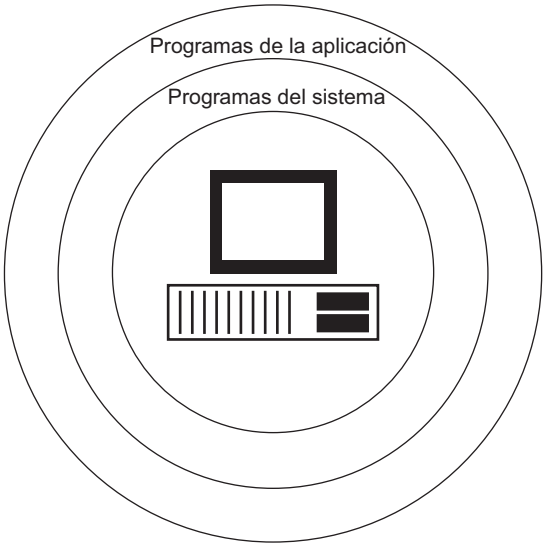


Figura 1.1. Relación entre programas de aplicación y programas del sistema.

rentes capas funcionan gracias a las instrucciones específicas (instrucciones máquina) que forman parte del *software* del sistema y llegan al *software* de aplicación, programado por los programadores de aplicaciones, que es utilizado por el usuario y que no requiere ser un especialista.

1.1.1. *Software* del sistema

El *software del sistema* coordina las diferentes partes de un sistema de computadora y conecta e interactúa entre el *software* de aplicación y el *hardware* del computador. El *software* del sistema gestiona el *hardware* del computador. Otro tipo de *software* del sistema que gestiona, controla las actividades de la computadora y realizan tareas de proceso comunes, se denomina *utility* o **utilidades** (en algunas partes de Latinoamérica, **utilerías**). El *software* del sistema que gestiona y controla las actividades del computador se denomina sistema operativo. Otro *software* del sistema son los programas traductores o de traducción de lenguajes de computador que convierten los programas escritos en lenguajes de programación, entendibles por los programadores, en lenguaje máquina que entienden las computadoras

El ***software* del sistema** es el conjunto de programas indispensables para que la máquina funcione; se denominan también *programas del sistema*. Estos programas son, básicamente, *el sistema operativo, los editores de texto,* los *compiladores*/*intérpretes* (lenguajes de programación) y los *programas de utilidad.*

1.1.2. *Software* de aplicación

El *software* de aplicación tiene como función principal asistir y ayudar a un usuario de un computador para ejecutar tareas específicas. Los programas de aplicación se pueden desarrollar con diferentes lenguajes y herramientas de *software*. Por ejemplo: una aplicación de procesa-

miento de textos (*word processing)* tal como *Word* de Microsoft o *Writely* de Google que ayuda a crear documentos, una hoja de cálculo tales como *Lotus 1-2-*3 o *Excel* que ayudan a automatizar tareas tediosas o repetitivas de cálculos matemáticos o estadísticos, a generar diagramas o gráficos, presentaciones visuales como *PowerPoint*; o a crear bases de datos como *Acces* u *Oracle* que ayudan a crear archivos y registros de datos.

Los usuarios, normalmente, compran el *software* de aplicaciones en discos CDs o DVDs (antiguamente en disquetes) o los descargan (bajan) de la Red Internet y han de instalar el *software* copiando los programas correspondientes de los discos en el disco duro de la computadora. Cuando compre estos programas asegúrese que son compatibles con su computador y con su sistema operativo. Existe una gran diversidad de programas de aplicación para todo tipo de actividades tanto de modo personal, como de negocios, navegación y manipulación en Internet, gráficos y presentaciones visuales, etc.

Los *lenguajes de programación* sirven para escribir programas que permitan la comunicación usuario/máquina. Unos programas especiales llamados *traductores* (**compiladores** o **intérpretes**) convierten las instrucciones escritas en lenguajes de programación en instrucciones escritas en lenguajes máquina (0 y 1, *bits*) que ésta pueda entender.

Los *programas de utilidad*[1] facilitan el uso de la computadora. Un buen ejemplo es un *editor de* textos que permite la escritura y edición de documentos. Este libro ha sido escrito con un editor de textos o *procesador de palabras* (“***word procesor***”).

Los programas que realizan tareas concretas, nóminas, contabilidad, análisis estadístico, etc. es decir, los programas que podrá escribir en C++ o Java, se denominan *programas de aplicación*. A lo largo del libro, se verán pequeños programas de aplicación que muestran los principios de una buena programación de una computadora.

1.1.3. Sistema operativo

Un sistema operativo **SO** (*Operating System, OS*) es tal vez la parte más importante del *software* del sistema y es el *software* que controla y gestiona los recursos del computador. En la práctica, el sistema operativo es la colección de programas de computador que controla la interacción del usuario y el hardware del computador. El sistema operativo es el administrador principal del computador, y por ello a veces, se le compara con el director de una orquesta ya que este *software* es el responsable de dirigir todas las operaciones del computador y gestionar todos sus recursos.

El sistema operativo asigna recursos, planifica el uso de recursos y tareas del computador, y monitoriza las actividades del sistema informático. Estos recursos incluyen memoria, dispositivos de **E/S** (Entrada/Salida), y la **UCP** (**U**nidad **C**entral de **P**roceso). El sistema operativo proporciona servicios tales como asignar memoria a un programa y manipulación del control de los dispositivos de E/S tales como el monitor el teclado o las unidades de disco. La Tabla 1.1 muestra algunos de los sistemas operativos más populares utilizados en enseñanza y en informática profesional.

Cuando un usuario interactúa con un computador, la interacción está controlada por el sistema operativo. Un usuario se comunica con un sistema operativo a través de una interfaz de usuario de ese sistema operativo. Los sistemas operativos modernos utilizan una interfaz gráfica de usuario, **IGU** (*Graphical User Interface, GUI*) que hace uso masivo de iconos, botones, barras y cuadros de diálogo para realizar tareas que se controlan por el teclado o el ratón (mouse) entre otros dispositivos.

---

[1] *Utility*: programa de utilidad.

Tabla 1.1. Sistemas operativos —actuales y antiguos— utilizados en educación y en la empresa

Sistema operativo	Características
Windows Vista[2]	Nuevo sistema operativo de Microsoft presentado a primeros de 2006, pero que se ha lanzado en noviembre de 2006.
Windows XP	Sistema operativo más utilizado en la actualidad, tanto en el campo de la enseñanza, como en la industria y negocios. Su fabricante es Microsoft.
Windows 98/ME/2000	Versiones anteriores de Windows pero que todavía hoy son muy utilizados.
UNIX	Sistema operativo abierto, escrito en C y todavía muy utilizado en el campo profesional.
Linux	Sistema operativo de *software* abierto, gratuito y de libre distribución, similar a UNIX, y una gran alternativa a Windows. Muy utilizado actualmente en servidores de aplicaciones para Internet.
Mac OS	Sistema operativo de las computadoras Apple Macintosh.
DOS y OS/2	Sistemas operativos creados por Microsoft e IBM respectivamente, ya poco utilizados pero que han sido la base de los actuales sistemas operativos.
CP/M	Sistema operativo de 8 bits para las primeras microcomputadoras nacidas en la década de los setenta.
Symbian	Sistema operativo para teléfonos móviles apoyado fundamentalmente por el fabricante de teléfonos celulares Nokia.
PalmOS	Sistema operativo para agendas digitales, PDA, del fabricante Palm.
Windows Mobile, CE	Sistema operativo para teléfonos móviles (celulares) con arquitectura y apariencias similares a Windows XP; actualmente en su versión 6.0

Normalmente, el sistema operativo se almacena de modo permanente en un chip de memoria de sólo lectura (**ROM**) de modo que esté disponible tan pronto el computador se pone en marcha ("*se enciende*" o "*se prende*"). Otra parte del sistema operativo puede residir en disco y se almacena en memoria RAM en la inicialización del sistema por primera vez en una operación que se llama c*arga* del sistema (*booting*).

Uno de los programas más importante es el **sistema operativo**, que sirve, esencialmente, para facilitar la escritura y uso de sus propios programas. El sistema operativo dirige las operaciones globales de la computadora, instruye a la computadora para ejecutar otros programas y controla el almacenamiento y recuperación de archivos (programas y datos) de cintas y discos. Gracias al sistema operativo es posible que el programador pueda introducir y grabar nuevos programas, así como instruir a la computadora para que los ejecute. Los sistemas operativos pueden ser, *monousuarios* (un solo usuario) y *multiusuarios*, o tiempo compartido (diferentes usuarios); atendiendo al número de usuarios y *monocarga* (una sola tarea) o *multitarea* (múltiples tareas) según las tareas (procesos) que puede realizar simul-

[2] Microsoft presentó, a nivel mundial, a finales de noviembre de 2006 y comercializa desde primeros de 2007, un nuevo sistema operativo *Windows Vista*, actualización de Windows XP pero con numerosas funcionalidades, especialmente de Internet y de seguridad, incluyendo en el sistema operativo programas que actualmente se comercializan independientes, tales como programas de reproducción de música, vídeo, y, fundamentalmente, un sistema de representación gráfica muy potente que permitirá construir aplicaciones en tres dimensiones, así como un buscador, un sistema antivirus y otras funcionalidades importantes.

táneamente. C++ corre prácticamente en todos los sistemas operativos, Windows XP, Windows 95, Windows NT, Windows 2000, UNIX, Linux, Vista..., y en casi todas las computadoras personales actuales PC, Mac, Sun, etc.

1.1.3.1. *Tipos de sistemas operativos*

Las diferentes características especializadas del sistema operativo permiten a los computadores manejar muchas tareas diferentes así como múltiples usuarios de modo simultáneo o en paralelo o bien de modo secuencial. En base a sus características específicas los sistemas operativos se pueden clasificar en varios grupos.

Multiprogramación/Multitarea

La *multiprogramación* permite a múltiples programas compartir recursos de un sistema de computadora en cualquier momento a través del uso concurrente de una UCP. Sólo un programa utiliza realmente la UCP en cualquier momento dado, sin embargo, las necesidades de entrada/salida pueden ser atendidas en el mismo momento. Dos o más programas están activos al mismo tiempo, pero no utilizan los recursos del computador simultáneamente. Con multiprogramación, un grupo de programas se ejecutan alternativamente y se alternan en el uso del procesador. Cuando se utiliza un sistema operativo de un único usuario, la multiprogramación toma el nombre de **multitarea**.

Multiprogramación

Método de ejecución de dos o más programas concurrentemente utilizando la misma computadora. La UCP ejecuta sólo un programa pero puede atender los servicios de entrada/salida de los otros al mismo tiempo.

Tiempo compartido (múltiples usuarios, *time sharing*)

Un *sistema operativo multiusuario* es un sistema operativo que tiene la capacidad de permitir que muchos usuarios compartan simultáneamente los recursos de proceso de la computadora. Centenas o millares de usuarios se pueden conectar al computador que asigna un tiempo de computador a cada usuario, de modo que a medida que se libera la tarea de un usuario, se realiza la tarea del siguiente, y así sucesivamente. Dada la alta velocidad de transferencia de las operaciones, la sensación es de que todos los usuarios están conectados simultáneamente a la UCP, con cada usuario recibiendo únicamente en un tiempo de máquina determinado.

Multiproceso

Un sistema operativo trabaja en multiproceso cuando puede enlazar a dos o más UCP para trabajar en paralelo en un único sistema de computadora. El sistema operativo puede asignar múltiples UCP para ejecutar diferentes instrucciones del mismo programa o de programas diferentes simultáneamente, dividiendo el trabajo entre las diferentes UCP.

La multiprogramación utiliza proceso concurrente con una UCP; el multiproceso utiliza proceso simultáneo con múltiples UCP.

1.2. RESOLUCIÓN DE PROBLEMAS Y DESARROLLO DE *SOFTWARE*

En este apartado se estudiará el proceso de desarrollo de *software* y sus fases. Estas fases ocurren en todo el *software* incluyendo los programas pequeños que se suelen utilizar en la etapa de formación de un estudiante o profesional. En los capítulos siguientes se aplicarán las fases de desarrollo de *software* a colecciones organizadas de datos. Estas colecciones organizadas de datos se denominan *estructuras de datos* y la representación y manipulación de tales estructuras de datos constituyen la esencia fundamental de este libro.

La creación de un programa requiere de técnicas similares a la realización de otros proyecto de ciencia e ingeniería, ya que, en la práctica, un programa no es más que una solución desarrollada para resolver un problema concreto. La escritura de un programa es casi la última etapa de un proceso en el que se determina primero ¿cuál es el problema? y el método que se utilizará para resolver el problema. Cada campo de estudio tiene su propio nombre para denominar el método sistemático utilizado para resolver problemas mediante el diseño de soluciones adecuadas. En ciencia y en ingeniería el método se conoce como método científico, mientras que cuando se realiza análisis cuantitativo se suele conocer como enfoque o *método sistemático*.

Las técnicas utilizadas por los desarrolladores profesionales de *software* para llegar a soluciones adecuadas para la resolución de problemas se denomina *proceso de desarrollo de software*. Aunque el número y nombre de las fases puede variar según los modelos, métodos y técnicas utilizadas.

En general, las fases de desarrollo de *software* se suelen considerar las siguientes:

- Análisis y especificación del problema.
- Diseño de una solución.
- Implementación (codificación).
- Pruebas, ejecución, corrección y depuración.
- Documentación.
- Mantenimiento y evaluación.

1.2.1. Modelos de proceso de desarrollo de *software*

A lo largo de la historia del desarrollo de *software* desde la década de los cincuenta del siglo pasado se han propuesto muchos modelos. Pressman en la última edición de su conocida obra *Ingeniería del software*, plantea la siguiente división [PRESSMAN 05][3]:

1. Modelos normativos (prescriptivos).
2. Modelos en cascada.
3. Modelos de proceso incremental
   - Modelo incremental.
   - Modelo **DRA** (**D**esarrollo **R**ápido de **A**plicaciones).
4. Modelos de proceso evolutivo
   - Modelo de prototipado (construcción de prototipos).
   - Modelo en espiral.
   - Modelo de desarrollo concurrente.

[3] [PRESSMAN 05] Roger Pressman. *Ingeniería del software. Un enfoque práctico*. México DF: McGraw-Hill, 2005, pp 48-101. Esta obra es una de las mejores referencias para el estudio de ingeniería de *software*.

5. Modelos especializados de proceso.
   - Modelo basado en componentes.
   - Modelo de métodos formales.
   - Desarrollo de *software* orientado a aspectos.
6. El proceso unificado (**RUP** de Booch, Rumbaugh y Jacobson)
7. Métodos Ágiles [Beck 01]
   - Programación Extrema (**XP**, *Extreme Programming*).

En la bibliografía recomendada y en la página *web* oficial del libro puede encontrar amplias referencia para el caso de que se desee estudiar y profundizar en ingeniería de *software*. En este capítulo y siguientes nos centraremos en las fases comunes de desarrollo de *software* como elementos centrales para la resolución de problemas.

1.2.2. Análisis y especificación del problema

El análisis de un problema se requiere para asegurarse de que el problema está bien definido y comprendido con claridad. La determinación de que el problema está definido claramente se hace después de que la persona entienda cuáles son las salidas requeridas y qué entradas son necesarias. Para realizar esta tarea, el análista debe tener una comprensión de cómo se pueden utilizar las entradas para producir las salidas deseadas.

Después de un análisis profundo del problema se debe realizar la **especificación** que es una descripción precisa y lo más exacta posible del problema; en realidad, es como un *contrato previo* para solución.

La especificación del problema no suele ser una tarea fácil sobre todo por la complejidad que entrañan la mayoría de los problemas del mundo real. La descripción inicial de un problema no suele ser clara y casi siempre, al principio, suele ser imprecisa y vaga.

La formulación de una especificación del problema requiere una descripción precisa y lo más completa posible de la *entrada* del problema (información disponible para la resolución del problema) y la *salida* requerida. Además, se requiere información adicional tal como: *hardware* y *software* necesarios, tiempo de respuesta, plazos de entrada, facilidad de uso, robustez del *software*, etc.

La persona que realiza el análisis debe tener una perspectiva inicial lo más amplia posible y comprender el propósito principal de los que el problema o sistema pretende conseguir. En sistemas grandes el análisis, lo realiza normalmente un analista de sistemas, y en programas individuales, el análisis se realiza directamente por el programador.

Con independencia de cómo y por quién se realice el análisis, a la conclusión del mismo se debe tener una comprensión muy clara de:

¿Qué debe hacer el sistema o el programa?
¿Qué salidas debe producir?
¿Qué entradas se requieren para obtener las salidas deseadas?

1.2.3. Diseño

La fase de diseño de la solución se inicia una vez que se tiene la especificación del problema y consiste en la formulación de los pasos o etapas para resolver el problema. Dos metodologías son las más utilizadas en el diseño: diseño *descendente* o *estructurado* que se apoya en *progra-*

mación estructurada y diseño *orientado* a *objetos* que se basa en la *programación orientada a objetos*.

El diseño de una solución requiere el uso de algoritmos. Un **algoritmo** es un conjunto de instrucciones o pasos para resolver un problema Los algoritmos deben procesar los datos necesarios para la resolución del problema. Los datos se organizan en *almacenes* o *estructuras* de datos. Los programas se compondrán de algoritmos que manipulan o procesan las estructuras de datos.

Una buena técnica para el diseño de un algoritmo, como se ha comentado (diseño descendente), es descomponer el problema en subproblemas o subtareas más pequeñas y sencillas, a continuación descomponer cada subproblema o subtarea en otra más pequeña y así sucesivamente, hasta llegar a subtareas que sean fáciles de implantar en C++ o en cualquier otro lenguaje de programación.

Los algoritmos se suelen escribir en *pseudocódigo* o en otras herramientas de programación como *diagramas de flujo* o *diagramas N-S*. Hoy día la herramienta más utilizada es el pseudocódigo o lenguaje algorítmico, consistentete en un conjunto de palabras —en español, inglés, etcétera— que representan tareas a realizar y una sintaxis de uso como cualquier otro lenguaje.

> *Técnica de diseño descendente*. 1) Descomponer una tarea en subtareas; a continuación cada subtarea en tareas más pequeñas. 2) Diseñar el algoritmo que describe cada tarea.

Otra técnica o método de diseño muy utilizado en la actualidad es el *diseño orientado a objetos*. El diseño descendente se basa en la descomposición de un problema en un conjunto de tareas y en la realización de los algoritmos que resuelven esas tareas, mientras que el diseño orientado a objetos se centra en la localización de módulos y objetos del mundo real. Estos objetos del mundo real están formados por datos y operaciones que actúan sobre los datos y modelan, a su vez, a los objetos del mundo real, e interactúan entre sí para resolver el problema concreto.

El diseño orientado a objetos ha conducido a la programación orientada a objetos, como uno de los métodos de programación más populares y más utilizados en el pasado siglo XX y actual XXI.

Consideraciones prácticas de diseño

Una vez que se ha realizado un análisis y una especificación del problema se puede desarrollar una solución. El programador se encuentra en una situación similar a la de un arquitecto que debe dibujar los planos de una casa; la casa deber cumplir ciertas espeficaciones y cumplir las necesidades de su propietario, pero puede ser diseñada y construida de muchas formas posibles. La misma situación se presenta al programador y al programa a construir.

En programas pequeños, el algoritmo seleccionado suele ser muy simple y consta de unos pocos cálculos que deben realizarse. Sin embargo, lo más normal es que la solución inicial debe ser refinada y organizada en subsistemas más pequeños, con especificaciones de cómo interactuar entre sí y los interfaces correspondientes. Para conseguir este objetivo, la descripción de la solución comienza desde el requisito de nivel más alto y prosigue en modo descendente hacia las partes que deben construir para conseguir este requisito.

Una vez que se ha desarrollado una estructura inicial se refinan las tareas hasta que éstas se encuentren totalmente definidas. El proceso de refinamiento de una solución continúa hasta

que los requisitos más pequeños se incluyan dentro de la solución. Cuando el diseño se ha terminado, el problema se resuelve mediante un *programa* o un *sistema de módulos* (funciones, clases...), *bibliotecas de funciones* o de *clases*, *plantillas*, *patrones*, etc.

1.2.4. Implementación (codificación)

La *codificación* o *implementación* implica la traducción de la solución de diseño elegida en un programa de computadora escrito en un lenguaje de programación tal como **C++** o **Java**. Si el análisis y las especificaciones han sido realizadas con corrección y los algoritmos son eficientes, la etapa de codificación normalmente es un proceso mecánico y casi automático de buen uso de las reglas de sintaxis del lenguaje elegido.

El código fuente, sea cual sea el lenguaje de programación en el cual se haya escrito, debe ser legible, comprensible y correcto (fiable). Es preciso seguir buenas prácticas de programación. Los buenos hábitos en la escritura de programas facilita la ejecución y prueba de los mismos. Tenga presente que los programas, subprogramas (funciones) y bibliotecas escritos por estudiantes suelen tener pocas líneas de programa (decenas, centenas...); sin embargo, los programas que resuelven problemas del mundo real contienen centenares, y millares e incluso millones de líneas del código fuente y son escritos por equipos de programadores. Estos programas se usan, normalmente, durante mucho tiempo y requieren un mantenimiento que en muchos casos se realiza por programadores distintos a los que escribieron el programa original. Por estas razones, es muy importante escribir programas que se puedan leer y comprender con facilidad así como seguir hábitos y reglas de lecturas que conduzcan a programas correctos (fiables).

> ***Recuerde***
>
> Escribir un programa sin diseño es como construir una casa sin un plano (proyecto).

1.2.5. Pruebas y depuración

La *prueba* y *corrección* de un programa pretende verificar que dicho programa funciona correctamente y cumple realmente todos sus requisitos. En teoría las pruebas revelan todos los errores existentes en el programa. En la práctica, esta etapa requiere la comprobación de todas las combinaciones posibles de ejecución de las sentencias de un programa; sin embargo, las pruebas requieren, en muchas ocasiones, mucho tiempo y esfuerzo, que se traduce a veces en objetivo imposible excepto en programas que son muy sencillos.

Los errores pueden ocurrir en cualquiera de las fases del desarrollo de *software*. Así, puede suceder que las especificaciones no contemplen de modo preciso la información de entrada, o los requisitos dados por el cliente; también puede suceder que los algoritmos no estén bien diseñados y contengan errores lógicos o por el contrario que los módulos o unidades de programa no estén bien codificados o la integración de las mismas en el programa principal no se haya realizado correctamente. La *detección* y *corrección de errores* es una parte importante del desarrollo de *software*, dado que dichos errores pueden aparecer en cualquier fase del proceso.

Debido a que las pruebas exhautivas no suelen ser factibles ni viables en la mayoría de los programas se necesitan diferentes métodos y filosofías de prueba. Una de las responsabilidades

de la ciencia de *ingeniería de software* es la construcción sistemática de un conjunto de pruebas (*test*) de entradas que permitan descubrir errores. Si las pruebas revelan un **error** (*bug*), el proceso de **depuración** —detección, localización, corrección y verificación— se puede iniciar. Es importante advertir que aunque *las pruebas puedan detectar la presencia de un error, no necesariamente implica la ausencia de errores*. Por consiguiente, el hecho de que una prueba o test, revele la existencia de un error no significa que uno indefinible pueda existir en otra parte del programa.

Para atrapar y corregir errores de un programa, es importante desarrollar un conjunto de datos de prueba que se puedan utilizar para determinar si el programa proporciona respuestas correctas. De hecho, una etapa aceptada en el desarrollo formal de *software* es planificar los procedimientos de pruebas y crear pruebas significativas antes de escribir el código. Los procedimientos de prueba de un programa deben examinar cada situación posible bajo la cual se ejecutará el programa. El programa debe ser comprobado con datos en un rango razonable así como en los límites y en las áreas en las que el programa indique al usuario que los datos no son válidos. El desarrollo de buenos procedimientos y datos de prueba en problemas complejos pueden ser más difíciles que la escritura del propio código del programa.

Verificación y validación

La prueba del *software* es un elemento de un tema más amplio que suele denominarse verificación y validación. **Verificación** es el conjunto de actividades que aseguran que el *software* funciona correctamente con una amplia variedad de datos. **Validación** es el conjunto diferente de actividades que aseguran que el *software* construido se corresponde con los requisitos del cliente [PRESMAN 05][4].

En la práctica, la verificación pretende comprobar que los documentos del programa, los módulos y restantes unidades son correctos, completos y consistentes entre sí y con los de las fases precedentes; la validación, a su vez, se ocupa de comprobar que estos productos se ajustan a la especificación del problema. Boehm [BOEHM 81][5] estableció que la verificación era la respuesta a: "*¿Estamos construyendo el producto correctamente?*" mientras que la validación era la respuesta a: "*¿Estamos construyendo el programa correcto?*".

En la prueba de *software* convencional en posible aplicar diferentes clases de pruebas. Pressman distingue las siguientes:

— *Prueba de unidad:* se centra el esfuerzo de verificación en la unidad más pequeña del diseño de *software*, el componente o módulo (función o subprograma) de *software* que se comprueban individualmente.
— *Prueba de integración:* se comprueba si las distintas unidades del programa se han unido correctamente. Aquí es muy importante descubrir errores asociados con la interfaz.

En el caso de *software* orientado a objetos cambia el concepto de unidad que pasa a ser la **clase** o la **instancia de una clase (objeto)** que empaqueta los *atributos* (datos) y las *operaciones* (funciones) que manipulan estos datos. La prueba de la clase en el *software* orientado a objetos es la equivalente a la prueba de unidad para el *software* convencional. La prueba de integración se realiza sobre clases que colaboran entre sí.

---

[4] *Ibid*, p. 364.
[5] Boehm.

Otro tipo de pruebas importantes son las pruebas del sistema. Una prueba del sistema comprueba que el sistema global del programa funciona correctamente; es decir las funciones, las clases, las bibliotecas, etc. Las pruebas del sistema abarcan una serie de pruebas diferentes cuyo propósito principal es verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas. Estas pruebas se corresponden con las distintas etapas del desarrollo del *software*.

Elección de datos de prueba

Para que los datos de prueba sean buenos, necesitan cumplir dos propiedades [MAINSA 01][6]:

1. Se necesita conocer cuál es la salida que debe producir un programa correcto para cada entrada de prueba.
2. Las entradas de prueba deben incluir aquellas entradas que más probabilidad tengan de producir errores.

Aunque un programa se compile, se ejecute y produzca una salida que parezca correcta no significa que el programa sea correcto. Si la respuesta correcta es 211492 y el programa obtiene 211491, algo está equivocado. A veces, el método más evidente para encontrar el valor de salida correcto es utilizar lápiz y papel utilizando un método distinto al empleado en el programa. Puede ayudarle utilizar valores de entrada más pequeños o simplemente valores de entrada cuya salida sea conocida.

Existen diferentes métodos para encontrar datos de prueba que tengan probabilidad de producir errores. Uno de los más utilizados se denomina *valores de frontera*. Un *valor frontera* de un problema es una entrada que produce un tipo de comportamiento diferente, por ejemplo, la función C++

```
int comprobar_hora (int hora)

//la hora de día está en el rango 0 a 23, tiene dos valores frontera 0
//(referir a 0, no es válido) y 23 (superior a 23 no es válida, ya que 24
//es un nuevo día); de igual modo si se deja contemplar el hecho de
//mañana (AM) o tarde (PM), los valores frontera serán 0 y 11, 12 y 23, //
respectivamente.
```

En general, no existen definiciones de valores frontera y es en las especificaciones del problema donde se pueden obtener dichos valores. Una buena regla suele ser la siguiente: *"Si no puede comprobar todas las entradas posible, al menos compruebe los valores frontera. Por ejemplo, si el rango de entrada va de 0 a 500.000, asegure la prueba de 0 y 500.000, y será buena práctica probar 0, 1 y –1 siempre que sean valores válidos"*.

1.2.6. Depuración

La **depuración** es el proceso de fijación o localización de errores. La detección de una entrada de prueba que produce un error es sólo parte del problema de prueba y depuración. Después que se encuentra una entrada de prueba errónea se debe determinar exactamente por qué ocurre

[6] Mainsa.

el error y, a continuación, depurar el programa. Una vez que se ha corregido un error debe volver a ejecutar el programa.

En programas sencillos la depuración se puede realizar con mayor o menor dificultad, pero en programas grandes el *seguimiento* (*traza* o *rastreo*) de errores es casi imposible sin ayuda de una herramienta de *software* denominada *depurador* (*debugger*). Un depurador ejecuta el código del programa línea a línea, o puede ejecutar el código harta que se produzca una cierta condición. El uso de un depurador puede especificar cuáles son las condiciones que originan la ejecución anómala de un programa. Los errores más frecuentes de un programa son:

- *Errores de sintaxis:* faltas gramaticales de la sintaxis del lenguaje de programación.
- *Errores en tiempo de ejecución:* se producen durante la ejecución del programa.
- *Errores lógicos:* normalmente errores de diseño del algoritmo.

EJEMPLOS

Errores de sintaxis	`double presupuesto //error, falta el ;` `cin presupuesto;  //error, falta >>`
Error de ejecución	`Cálculo de división por cero, obtener raíces de números negativos, valores fuera de rango.`
Error lógico	`Mal planteamiento en el diseño de un algoritmo.`

1.2.7. Documentación

El desarrollo de *software* requiere un gran esfuerzo de documentación de las diferentes etapas. En la práctica, muchos de los documentos clave (críticos) se crean durante las fases de análisis, diseño, codificación y prueba. La documentación completa del *software* presenta todos los documentos en un manual que sea útil a los programadores y a su organización.

Aunque el número de documentos puede variar de un proyecto a otro, y de una organización a otra, esencialmente existen cinco documentos imprescindibles en la documentación final de un programa:

1. Descripción del problema (especificaciones).
2. Cambio y desarrollo de los algoritmos.
3. Listados de programas, bien comentados.
4. Ejecución de las pruebas de muestra.
5. Manual del usuario.

La documentación del *software* comienza en la fase de análisis y especificación del problema y continúa en la fase de mantenimiento y evolución.

1.2.8. Mantenimiento

Una vez que el *software* se ha depurado completamente y el conjunto de programas, biblioteca de funciones y clases, etc., se han terminado y funcionan correctamente, el uso de los mismos se puede extender en el tiempo durante grandes períodos (normalmente meses o años).

La fase de mantenimiento del *software* está relacionada con corrección futura de problemas, revisión de especificaciones, adición de nuevas características, etc. El mantenimiento requiere, normalmente, un esfuerzo importante ya que si bien el desarrollo de un programa puede durar días, meses o años, el mantenimiento se puede extender a años e incluso décadas. Un ejemplo típico estudiado en numerosos cursos de ingeniería de *software* fue el esfuerzo realizado para asegurar que los programas existentes funcionan correctamente al terminar el siglo XX conocido como el efecto del año 2000.

Estadísticamente está demostrado que los sistemas de *software*, especialmente los desarrollados para resolver problemas complejos presentan errores que no fueron detectados en las diferentes pruebas y que se detectan cuando el *software* se pone en funcionamiento de modo comercial o profesional. La reparación de estos errores es una etapa importante y muy costosa del mantenimiento de un programa.

Además de estas tareas de mantenimiento existen muchas otras tareas dentro de esta etapa: "mejoras en la eficiencia, añadir nuevas características (por ejemplo, nuevas funcionalidades), cambios en el *hardware*, en el sistema operativo, ampliar número máximo de usuarios...". Otros cambios pueden venir derivados de cambios en normativas legales, en la organización de la empresa, en la difusión del producto a otros países, etc.

Estudios de *ingeniería de software* demuestran que el porcentaje del presupuesto de un proyecto *software* y del tiempo de programador/analista/ingeniero de *software* ha ido creciendo por décadas. Así, en la década de los setenta, se estimaba el porcentaje de mantenimiento entre un 35-40 por 100, en la década de los ochenta, del 40 al 60 por 100, y se estima que en la década actual del siglo XXI, puede llegar en aplicaciones para la web, videojuegos, *software* de inteligencia de negocios, de gestión de relaciones con los clientes (**CRM)**, etc., hasta un 80 o un 90 por 100 del presupuesto total del desarrollo de un producto *software*.

Por todo lo anterior, es muy importante que los programadores diseñen programas legibles, bien documentados y bien estructurados, bibliotecas de funciones y de clases con buena documentación, de modo que los programas sean fáciles de comprender y modificar y en consecuencia fáciles de mantener.

1.3. CALIDAD DEL *SOFTWARE*

El *software* de calidad debe cumplir las siguientes características:

Corrección: Capacidad de los productos *software* de realizar exactamente las tareas definidas por su especificación.

Legibilidad y comprensibilidad: Un sistema debe ser fácil de leer y lo más sencillo posible. Estas características se ven favorecidas con el empleo de abstracciones, sangrado y uso de comentarios.

Extensibilidad: Facilidad que tienen los productos de adaptarse a cambios en su especificación. Existen dos principios fundamentales para conseguir esto, diseño simple y descentralización.

Robustez: Capacidad de los productos *software* de funcionar incluso en situaciones anormales.

Eficiencia: La eficiencia de un *software* es su capacidad para hacer un buen uso de los recursos del computador. Un sistema eficiente es aquél cuya velocidad es mayor con el menor espacio de memoria ocupada. Las grandes velocidades de los microprocesadores (unidades centrales de proceso) actuales, junto con el aumento considerable de las memorias centrales (cifras típi-

cas usuales superan siempre 1-4 GB), hacen que disminuya algo la importancia concedida a ésta, debiendo existir un compromiso entre legibilidad, *modificabilidad* y eficiencia.

Facilidad de uso: La *utilidad* de un sistema está relacionada con su facilidad de uso. Un *software* es fácil de utilizar cuando el usuario puede comunicarse con él de manera cómoda.

Transportabilidad (*portabilidad*): *La transportabilidad o portabilidad* es la facilidad con la que un *software* puede ser transportado sobre diferentes sistemas físicos o lógicos.

***Verificabilidad*:** La *verificabilidad*, "facilidad de verificación" de un *software*, es su capacidad para soportar los procedimientos de validación y de aceptar juegos de test o ensayo de programas.

Reutilización: Capacidad de los productos de ser reutilizados, en su totalidad o en parte, en nuevas aplicaciones.

Integridad: La integridad es la capacidad de un *software* para proteger sus propios componentes contra los procesos que no tengan el derecho de acceso.

Compatibilidad: Facilidad de los productos para ser combinados con otros y usados en diferentes plataformas *hardware* o *software*.

1.4. ALGORITMOS

El término **resolución de un problema** se refiere al proceso completo que abarca desde la descripción inicial del problema hasta el desarrollo de un programa de computadora que lo resuelva. La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto. Los pasos para la resolución de un problema son:

1. *Diseño del algoritmo* que describe la secuencia ordenada de pasos —sin ambigüedades— que conducen a la solución de un problema dado. (*Análisis del problema y desarrollo del algoritmo*).
2. Expresar el algoritmo como un *programa* en un lenguaje de programación adecuado. (*Fase de codificación*).
3. *Ejecución y validación* del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo indicando *cómo* hace el algoritmo la tarea solicitada, y eso se traduce en la construcción de un algoritmo. El resultado final del diseño es una solución que debe ser fácil de traducir a estructuras de datos y estructuras de control de un lenguaje de programación específico.

Las dos herramientas más comúnmente utilizadas para diseñar algoritmos son: *diagramas de flujo y pseudocódigos.*

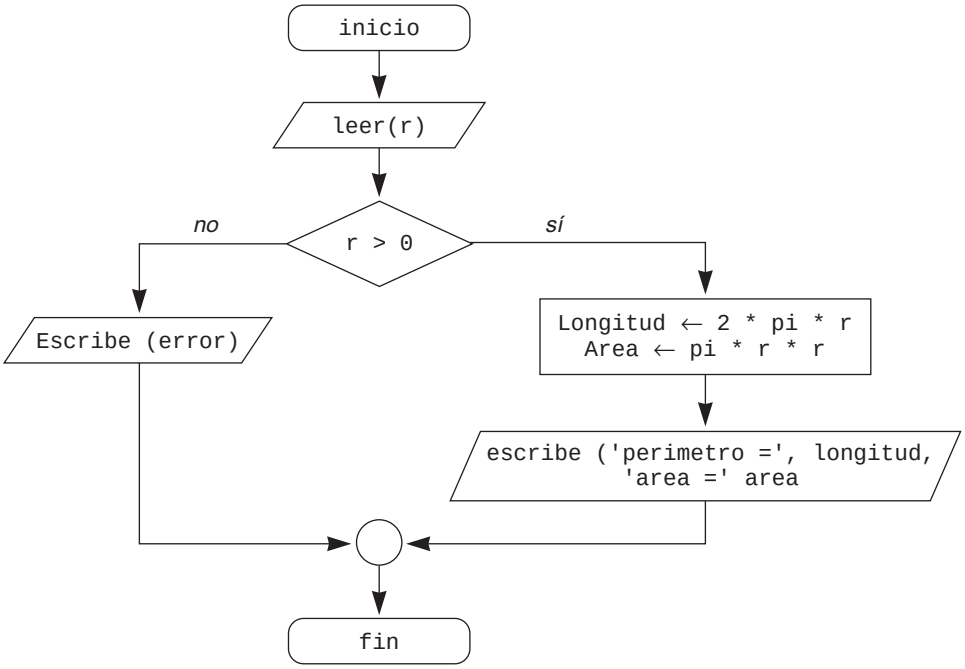
Diagramas de flujo (*flowchart*)

Es una representación gráfica de un algoritmo.

EJEMPLO 1.1. Diagrama de flujo que representa un algoritmo que lea el radio de un círculo, calcule su perímetro y su área.

Se declaran las variables reales `r`, `longitud` y `area`, así como la constante `pi`

```
constantes pi = 3.14
variables real: r, longitud, area
```



Pseudocódigo

En esencia el pseudocódigo se puede definir como *un lenguaje de especificación de algoritmos.*

EJEMPLO 1.2. Realizar un algoritmo que lea tres números; si el primero es positivo calcule el producto de los tres números , y en otro caso calcule la suma.

Se usan tres variables enteras `Numero1, Numero2, Numero3` , en las que se leen los datos, y otras dos variables `Producto` y `Suma` en las que se calcula, o bien el producto, o bien la suma. El algoritmo que resuelve el problema es el siguiente.

```
Entrada Numero1,Numero2 y Numero3
Salida  Suma o el Producto

inicio
1 leer los tres números Numero1, Numero2, Numero3
2 si el Numero1 es positivo
    calcular el producto de los tres números
    escribir el producto
3 si el Numero1 es no positivo
    calcular la suma de los tres números
    escribir la suma
fin
```

El *algoritmo en pseudocódigo* es:

```
algoritmo Producto__Suma
variables
    entero: Numero1, Numero2, Numero3, Producto, Suma
inicio
     Leer(Numero1, Numero2, Numero3)
     si (Numero1 > 0) entonces
         Producto ← Numero1* Numero2 * Numero3
         Escribe('El producto de los números es:', Producto)
     sino
         Suma ← Numero1 + Numero2 + Numero3
         Escribe('La suma de los números es: ', Suma)
     fin si
fin
```

El *algoritmo escrito* en C++ es:

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int Numero1, Numero2, Numero3, Producto, Suma;
    cin >> Numero1 >> Numero2 >>Numero3;
    if(Numero1 > 0)
    {
        Producto = Numero1* Numero2 * Numero3;
        cout <<"El producto de los números es" << Producto;
    }
    else
    {
        Suma = Numero1 + Numero2 + Numero3;
        cout <<" La suma de los números es:" << Suma;
    }
    return 0;
}
```

El **algoritmo** es la especificación concisa del método para resolver un problema con indicación de las acciones a realizar. Un **algoritmo** es un conjunto finito de reglas que dan una secuencia de operaciones para resolver un determinado problema. Es, por consiguiente, *un método para resolver un problema que tiene en general una entrada y una salida*. Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser *preciso* e indicar el orden de realización de cada paso.
- Un algoritmo debe estar bien *definido*. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser *finito*. Si se sigue un algoritmo, se debe terminar en algún momento; es decir, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: *Entrada, Proceso* y *Salida*.

EJEMPLO 1.3. Diseñar un algoritmo que permita saber si un número entero positivo es primo o no. Un número es primo si sólo puede dividirse por sí mismo y por la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo, 9, 8, 6, 4, 12, 16, 20, etc., no son primos, ya que son divisibles por números distintos a ellos mismos y a la unidad. Así, 9 es divisible por 3, 8 lo es por 2, etc. El algoritmo de resolución del problema pasa por dividir sucesivamente el número por 2, 3, 4, etc.

Entrada: `dato n entero positivo`
Salida: `es o no primo.`
Proceso:

1. **Inicio**.
2. Hacer x igual a 2 (x = 2, x variable que representa a los divisores del número que se buscan).
3. Dividir n por x (n/x).
4. **Si** el resultado de n/x es entero, **entonces** n no es un número primo y bifurcar al punto 7; en caso contrario, continuar el proceso.
5. Suma 1 a x (x ← x + 1).
6. **Si** x es igual a n, **entonces** n es un número primo ; **en caso contrario**, bifurcar al punto 3.
7. **Fin**.

El algoritmo anterior escrito en pseudocódigo y C++ es el siguiente:

Algoritmo en pseudocódigo	**Algoritmo en C++**

algoritmo primo
inicio
  variables
  entero: n, x;
  lógico: primo;
  leer(n);
  x ← 2;
  primo ← verdadero;
  mientras primo **y** (x < n) **hacer**
  si n mod x <> 0 **entonces**
  x ← x+1
  sino
  primo ← falso
  fin si
  fin mientras
  si (primo) **entonces**
  escribe('es primo')
  sino
  escribe('no es primo')
  fin si
fin
	#include <cstdlib>
	#include <iostream>
	using namespace std;
	int main(int argc, char *argv[])
	{
	  int n, x;
	  bool primo;
	  cin >> n;
	  x = 2;
	  primo = true;
	  while (primo &&(x < n))
	  if (n % x != 0)
	  x = x+1;
	  else
	  primo = false;
	  if (primo)
	  cout << "es primo";
	  else
	  cout << " no es primo";
	  return 0;
	}

EJEMPLO 1.4. Calcular la media de una serie de números positivos, suponiendo que los datos se leen desde un terminal. Un valor de cero —como entrada— indicará que se ha alcanzado el final de la serie de números positivos.

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, el algoritmo en forma descriptiva es:

```
inicio
  1. Inicializar contador de numeros C y variable suma S a cero (S ← 0,
     C ← 0).
  2. Leer un numero en la variable N (leer(N))
  3. Si el numero leído es cero: (si (N = 0) entonces)
     3.1 Si se ha leído algún número (Si C>0)
         • calcular la media; (media ← S/C)
         • imprimir la media; (Escribe(media))
     3.2 si no se ha leído ningún número (Si C = 0)
         • escribir no hay datos.
     3.3 fin del proceso.
  4. Si el numero leído no es cero : (Si (N <> 0) entonces
     • calcular la suma; (S ← S+N)
     • incrementar en uno el contador de números; (C←C+1)
     • ir al paso 2.
fin
```

El algoritmo anterior escrito en pseudocódigo y en C++ es el siguiente:

Algoritmo escrito en pseudocódigo	**Algoritmo escrito en C++**

algoritmo media
inicio
variables
entero: N, C, S;
real: media;
C ← 0;
S ← 0;
repetir
leer(N)
si N <> 0 **entonces**
S ← S + N;
C ← C +1 ;
fin si
hasta N = 0
si C > 0 **entonces**
media ← S / C
escribe(media)
sino
escribe("no datos")
fin si
fin
	#include <cstdlib>
	#include <iostream>
	using namespace std;
	int main(int argc, char *argv[])
	{
	int N, C, S;
	float media;
	C = 0; S = 0;
	do
	{
	cin >> N;
	if(N != 0)
	{
	S = S + N;
	C = C + 1 ;
	}
	} while (N != 0);
	if(C > 0)
	{media = S / C;
	cout << media;
	}
	else
	cout << “no datos”;
	return 0;
	}

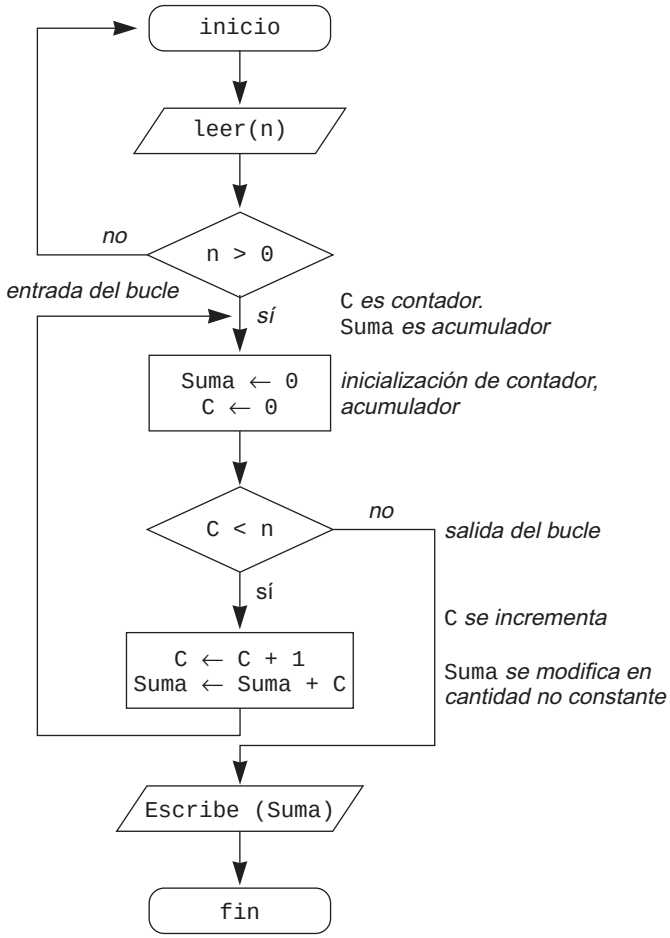
EJEMPLO 1.5. Algoritmo que lee un número entero positivo *n*, y suma los *n* primeros número naturales.

Inicialmente se asegura la lectura del número natural `n` positivo. Mediante un contador `C` se cuentan los números naturales que se suman, y en el acumulador `Suma` se van obteniendo las sumas parciales. Además del diagrama de flujo se realiza un seguimiento para el caso de la entrada `n = 5`.

```
Variables Entero: n, Suma, C
```

seguimiento

paso	n	C	Suma
0	5	0	0
1	5	1	1
2	5	2	3
3	5	3	6
4	5	4	10
5	5	5	15



El algoritmo anterior escrito en pseudocódigo y C++ es el siguiente:

Algoritmo escrito en pseudocódigo	**Algoritmo escrito en C++**

algoritmo suma_n_naturales
inicio
  variables
  entero: Suma, C, n;
repetir
  leer(n)
hasta n>0
C ← 0;
Suma ← 0;
mientras C < n **hacer**
  C ← C + 1;
  Suma ← Suma + C;
fin mientras
escribe(Suma)
fin
	#include <cstdlib>
	#include <iostream>
	using namespace std;
	
	int main(int argc, char *argv[])
	{
	  int Suma, C, n;
	  do
	  cin >> n;
	  while (n <= 0);
	  C = 0; Suma = 0;
	  while (C < n)
	  {
	  C = C + 1;
	  Suma = Suma + C;
	  }
	  cout << Suma;
	  return 0;
	}

1.5. ABSTRACCIÓN EN LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación son las herramientas mediante las cuales los diseñadores de programas pueden implementar los modelos abstractos. La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: *abstracción de datos* (perteneciente a los datos) y *abstracción de control* (perteneciente a las estructuras de control).

Desde comienzos del decenio de los sesenta, en que se desarrollaron los primeros lenguajes de programación de alto nivel, ha sido posible utilizar las abstracciones más primitivas de ambas categorías (variables, tipos de datos, funciones, control de bucles (lazos), etc.).

1.5.1. Abstracciones de control

Los microprocesadores ofrecen directamente sólo dos mecanismos para controlar el flujo y ejecución de las instrucciones: *secuencia* y *salto*. Los primeros lenguajes de programación de alto nivel introdujeron las estructuras de control: *sentencias de bifurcación* (`if`) y *bucles* (`for`, `while`, `do-loop`, `do-while`, etc.).

Las estructuras de control describen el orden en que se ejecutan las sentencias o grupos de sentencia (*unidades de programa*). Las unidades de programa se utilizan como bloques básicos de la clásica descomposición "descendente". En todos los casos, los subprogramas constituyen una herramienta potente de abstracción ya que durante su implementación, el programador describe en detalle cómo funcionan. Cuando el subprograma se llama, basta con conocer lo que hace y no cómo lo hace. De este modo, se convierten en cajas negras que amplían el lenguaje de programación a utilizar. En general, los subprogramas son los mecanismos más ampliamente utilizados para reutilizar código, a través de colecciones de subprogramas en bibliotecas.

Las abstracciones y estructuras de control se clasifican en estructuras de control a nivel de sentencia y a nivel de unidades. La abstracción de control a nivel de unidad se conoce como *abstracción procedimental*.

Abstracción procedimental (por procedimientos o funciones)

Es esencial para diseñar *software* modular y fiable. La *abstracción procedimental* se basa en la utilización de procedimientos o funciones sin preocuparse de cómo se implementan. Esto es posible sólo si conocemos qué hace el procedimiento; esto es, conocemos la sintaxis y semántica que utiliza el procedimiento o función. La abstracción aparece en los subprogramas debido a las siguientes causas:

- Con el nombre de los subprogramas, un programador puede asignar una descripción abstracta que captura el significado global del subprograma. Utilizando el nombre en lugar de escribir el código permite al programador aplicar la acción en términos de su descripción de alto nivel en lugar de sus detalles de bajo nivel.
- Los subprogramas proporcionan ocultación de la información. Las variables locales y cualquier otra definición local se encapsulan en el subprograma, ocultándolas realmente de forma que no se pueden utilizar fuera del subprograma. Por consiguiente, el programador no tiene que preocuparse sobre las definiciones locales.
- Los parámetros de los subprogramas, junto con la ocultación de la información anterior, permiten crear subprogramas que constituyen entidades de *software* propias. Los detalles locales de la implementación pueden estar ocultos mientras que los parámetros se pueden utilizar para establecer la interfaz *público.*

En C++ la abstracción procedimental se establece con los *métodos* o *funciones miembro* de clases.

Otros mecanismos de abstracción de control

La evolución de los lenguajes de programación ha permitido la aparición de otros mecanismos para la abstracción de control, tales como *manejo de excepciones, corrutinas, unidades concurrentes* o *plantillas* (*templates*). Estas construcciones las soportan los lenguajes de programación basados y orientados a objetos, tales como C++ C#, Java, Modula-2, Ada, Smalltalk o Eiffel.

1.5.2. Abstracciones de datos

Los primeros pasos hacia la abstracción de datos se crearon con lenguajes tales como FORTRAN, COBOL y ALGOL 60, con la introducción de tipos de variables diferentes, que manipulan enteros, números reales, caracteres, valores lógicos, etc. Sin embargo, estos tipos de datos no podían ser modificados y no siempre se ajustaban al tipo necesitado. Por ejemplo, el tratamiento de cadenas es una deficiencia en FORTRAN, mientras que la precisión y fiabilidad para cálculos matemáticos es muy alta.

La siguiente generación de lenguajes, PASCAL, SIMULA-67 y ALGOL 68, ofreció una amplia selección de tipos de datos y permitió al programador modificar y ampliar los tipos de datos existentes mediante construcciones específicas (por ejemplo, *arrays* y registros). Además, SIMULA-67 fue el primer lenguaje que mezcló datos y procedimientos mediante la construcción de clases, que eventualmente se convirtió en la base del desarrollo de programación orientada a objetos.

La *abstracción de datos* es la técnica de programación que permite inventar o definir nuevos tipos de datos (tipos de datos definidos por el usuario) adecuados a la aplicación que se desea realizar. La abstracción de datos es una técnica muy potente que permite diseñar programas más cortos, legibles y flexibles. La esencia de la abstracción es similar a la utilización de un tipo de dato, cuyo uso se realiza sin tener en cuenta cómo está representado o implementado.

Los tipos de datos son abstracciones y el proceso de construir nuevos tipos se llaman abstracciones de datos. Los nuevos tipos de datos definidos por el usuario se llaman *tipos abstractos de datos* (ADT, **A**bstract **D**ata **T**ypes).

El concepto de tipo, tal como se definió en PASCAL y ALGOL 68, ha constituido un hito importante hacia la realización de un lenguaje capaz de soportar programación estructurada. Sin embargo, estos lenguajes no soportan totalmente una metodología orientada a objetos. La abstracción de datos útil para este propósito, no sólo clasifica objetos de acuerdo a su estructura de representación, sino que se clasifican de acuerdo al comportamiento esperado. Tal comportamiento es expresable en términos de operaciones que son significativas sobre esos datos, y las operaciones son el único medio para crear, modificar y acceder a los objetos.

En términos más precisos, Ghezzi indica que un tipo de dato definido por el usuario se denomina *tipo abstracto de dato* (**TAD**) si:

- Existe una construcción del lenguaje que le permite asociar la representación de los datos con las operaciones que lo manipulan;
- La representación del nuevo tipo de dato está oculta de las unidades de programa que lo utilizan [GHEZZI 87][7].

Las **clases** de C++, C# y Java cumplen las dos condiciones: agrupa los datos junto a las operaciones, y su representación queda oculta de otras clases.

Los tipos abstractos de datos proporcionan un mecanismo adicional mediante el cual se realiza una separación clara entre la *interfaz* y la *implementación* del tipo de dato. La **implementación de un tipo abstracto de dato** consta de:

1. *Representación*: elección de las estructuras de datos.
2. *Operaciones*: elección de los algoritmos.

La interfaz del tipo abstracto de dato se asocia con las operaciones y datos *visibles* al exterior del **TAD**.

1.6. TIPOS ABSTRACTOS DE DATOS

Algunos lenguajes de programación tienen características que nos permiten ampliar el lenguaje añadiendo sus propios tipos de datos. Un tipo de dato definido por el programador se denomina *tipo abstracto de datos* (**TAD**) para diferenciarlo del tipo fundamental (predefinido) de datos. Por ejemplo, en C++ el tipo `Punto`, que representa a las coordenadas `x` e `y` de un sistema de coordenadas rectangulares, no existe. Sin embargo, es posible implementar el tipo abstracto de datos, considerando los valores que se almacenan en las variables y qué operaciones están disponibles para manipular estas variables. En esencia un tipo abstracto de datos es un tipo que consta de datos (estructuras de datos propias) y operaciones que se pueden realizar sobre esos datos. Un **TAD** se compone de *estructuras de datos* y los *procedimientos* o *funciones* que manipulan esas estructuras de datos.

[7] GHEZZI 87.

Un tipo abstracto de datos puede definirse mediante la ecuación:

TAD = Representación (datos) + Operaciones (funciones y procedimientos)

La estructura de un tipo abstracto de dato, desde un punto de vista global, se compone de la interfaz y de la implementación (Figura 1.2).

Las estructuras de datos reales elegidas para almacenar la representación de un tipo abstracto de datos son invisibles a los usuarios o clientes. Los algoritmos utilizados para implementar cada una de las operaciones de los **TAD** están encapsuladas dentro de los propios **TAD**. La característica de ocultamiento de la información significa que los objetos tienen *interfaces públicos*. Sin embargo, las representaciones e implementaciones de esos *interfaces* son *privados*.

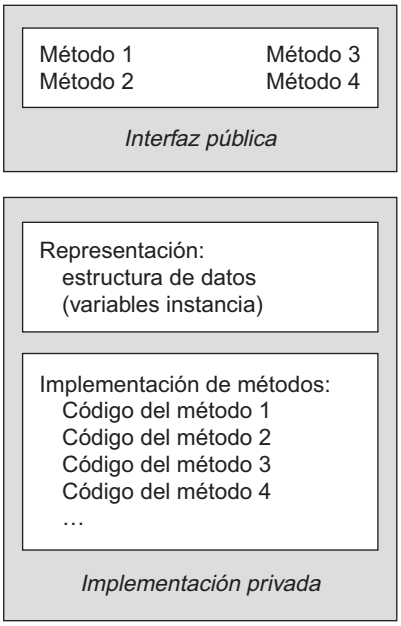


Figura 1.2. Estructura de un tipo abstracto de datos (**TAD**).

1.6.1. Ventajas de los tipos abstractos de datos

Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

1. Permite una mejor conceptualización y *modelización* (modelado) del mundo real. Mejora la representación y la comprensibilidad. Clasifica los objetos basados en estructuras y comportamientos comunes.
2. Mejora la robustez del sistema. Permiten la especificación del tipo de cada variable, de tal forma que se facilita la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.

3. Mejora el rendimiento (prestaciones). Para sistemas tipificados, el conocimiento de los objetos permite la optimización de tiempo de compilación.
4. Separa la implementación de la especificación. Permite la modificación y mejora de la implementación sin afectar al interfaz público del tipo abstracto de dato.
5. Permite la extensibilidad del sistema. Los componentes de *software* reutilizables son más fáciles de crear y mantener.
6. Recoge mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

Un programa que maneja un **TAD** lo hace teniendo en cuenta las operaciones o funcionalidad que tiene, sin interesarse por la representación física de los datos. Es decir, los *usuarios* de un **TAD** se comunican con éste a partir de la interfaz que ofrece el **TAD** mediante funciones de acceso. Podría cambiarse la implementación de tipo de datos sin afectar al programa que usa el **TAD** ya que para el programa está *oculta*.

Las unidades de programación de lenguajes que pueden implementar un **TAD** reciben distintos nombres:

Modula-2	*módulo*
Ada	*paquete*
C++	*clase*
C#	*clase*
Java	*clase*

En estos lenguajes se definen la *especificación* del TAD, que declara las operaciones y los datos, y la *implementación*, que muestra el código fuente de las operaciones y que permanece oculto al exterior del módulo.

EJEMPLO 1.6. Clase `hora` que tiene datos separados de tipo `int` para horas, minutos y segundos. Un constructor inicializará este dato a 0, y otro lo inicializará a valores fijos. Una función miembro deberá visualizar la hora en formato 11:59:59. Otra función miembro sumará dos objetos de tipo hora pasados como argumentos. Una función principal `main()` creará dos objetos inicializados y otro que no está inicializado. Se suman los dos valores inicializados y se deja el resultado en el objeto no inicializado. Por último, se muestra el valor resultante.

```
#include <cstdlib>
#include <iostream>
using namespace std;

class hora
{
private:
    int horas, minutos, segundos;
public:
    hora(){ horas = 0; minutos = 0; segundos = 0; }
    hora(int h, int m, int s){horas = h; minutos = m; segundos = s;}
    void visualizar();
    void sumar(hora h1, hora h2 );
};
```

```
void hora::visualizar()
{
    cout << horas << ":" << minutos << ":" << segundos << endl;
}

void hora::sumar(hora h1, hora h2 )
{
    segundos = h2.segundos + h1.segundos;
    minutos = h2.minutos + h1.minutos + segundos / 60;
    segundos = segundos % 60;
    horas = h2.horas + h1.horas + minutos / 60;
    minutos = minutos % 60;
}

int main(int argc, char *argv[])
{
    hora h1(10,40,50), h2(12,35,40), h;
    h1.visualizar();
    h2.visualizar();
    h.sumar(h1,h2);
    h.visualizar();
    return 0;
}
```

1.6.2. Especificación de los TAD

El objetivo de la especificación es describir el comportamiento del **TAD**; consta de dos partes, la descripción matemática del conjunto de datos, y de las operaciones definidas en ciertos elementos de ese conjunto de datos.

La especificación del **TAD** puede tener un enfoque *informal*, éste describe los datos y las operaciones relacionadas en *lenguaje natural*. Otro enfoque más riguroso, *especificación formal*, supone suministrar un conjunto de *axiomas* que describen las operaciones en su aspecto *sintáctico* y *semántico*.

1.6.2.1. *Especificación informal de un TAD*

Consta de dos partes:

1. Detallar en los datos del tipo, los valores que pueden tomar.
2. Describir las operaciones, relacionándolas con los datos.

El formato que generalmente se emplea, primero especifica el nombre del **TAD** y los datos:

```
TAD nombre del tipo (valores y su descripción)
```

A continuación, cada una de las operaciones con sus argumentos, y una descripción funcional en lenguaje natural, con este formato:

```
Operación(argumentos).
```

Descripción funcional

A continuación se especifica, siguiendo esos pasos, el tipo abstracto de datos `Conjunto`:

TAD Conjunto `(colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).`

Las operaciones básicas sobre conjuntos son las siguientes:

`Conjuntovacio`	Crea un conjunto sin elementos.
`Añadir(Conjunto,elemento)`	Comprueba si el elemento forma parte del conjunto, en caso negativo se añade. La operación modifica al conjunto.
`Retirar(Conjunto, elemento)`	Si el elemento pertenezca al conjunto se retira. La operación modifica al conjunto.
`Pertenece(Conjunto,elemento)`	Verifica si el elemento forma parte del conjunto, en cuyo caso devuelve *cierto*.
`Esvacio(Conjunto)`	Verifica si el conjunto no tiene elementos, en cuyo caso devuelve *cierto*.
`Cardinal(Conjunto)`	Devuelve el número de elementos del conjunto.
`Union (Conjunto, Conjunto)`	Realiza la operación matemática unión de dos conjuntos. La operación devuelve un conjunto con los elementos comunes y no comunes de ambos.

Se puede especificar más operaciones sobre conjuntos, todo dependerá de la aplicación que se quiera dar al **TAD**.

Norma

La especificación informal de un **TAD** tiene como objetivo describir los datos del tipo y las operaciones según la funcionalidad que tienen. No sigue nor mas rígidas, simplemente indica, de forma comprensible, la acción que realiza cada operación.

1.6.2.2. *Especificación formal de un TAD*

La especificación formal proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones. La descripción ha de incluir una parte de sintaxis, en cuanto a los tipos de los argumentos y el tipo del resultado, y una parte de semántica que detalla para unos valores particulares de los argumentos la expresión del resultado que se obtiene. La especificación formal ha de ser lo bastante *potente* para que cumpla el objetivo de verificar la corrección de la implementación del **TAD**.

El esquema que se sigue consta de una cabecera con el nombre del **TAD** y los datos:

```
TAD nombre del tipo (valores que toma los datos del tipo)
```

A continuación, la sintaxis de las operaciones que lista las operaciones mostrando los tipos de los argumentos y el tipo del resultado:

```
Operación(Tipo argumento, ...)  ->  Tipo resultado
```

Se continúa con la *semántica* de las operaciones. Ésta se construye dando unos valores particulares a los argumentos, a partir de los cuales se obtiene una expresión resultado. Éste puede tener referencias a tipos ya definidos, valores de tipo lógico o referencias a otras operaciones del propio **TAD**.

```
Operación(valores particulares argumentos) ⇒ expresión resultado
```

Al realizar la especificación formal siempre hay operaciones definidas por sí mismas, se denominan *constructores* del **TAD**. Mediante los constructores se generan todos los posibles valores del **TAD**. Normalmente, se elige como constructor la operación que inicializa (por ejemplo, `Conjuntovacío` en el `TAD Conjunto`), y la operación que añade un dato o elemento (operación común a la mayoría de los tipos abstractos de datos). Se acostumbra a marcar con un asterisco a las operaciones que son constructores.

A continuación, se especifica formalmente el `TAD Conjunto`. Para formar la expresión resultado se hace uso, si es necesario, de la sentencia alternativa **`si-entonces-sino`**.

`TAD Conjunto(`*colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones*`).`

Sintaxis

```
*Conjuntovacio[8]                  ->   Conjunto
*Añadir(Conjunto, Elemento)[8]     ->   Conjunto
Retirar(Conjunto, Elemento)        ->   Conjunto
Pertenece(Conjunto, Elemento)      ->   boolean
Esvacio(Conjunto)                  ->   boolean
Cardinal(Conjunto)                 ->   entero
Union(Conjunto, Conjunto)          ->   Conjunto
```

Semántica ∀ `e1,e2 ∈ Elemento` y ∀ `C,D ∈ Conjunto`

```
Añadir(Añadir(C, e1), e1)            ⇒  Añadir(C, e1)
Añadir(Añadir(C, e1), e2)            ⇒  Añadir(Añadir(C, e2), e1)
Retirar(Conjuntovacio, e1)           ⇒  Conjuntovacio
Retirar(Añadir(C, e1), e2)           ⇒  si e1 = e2 entonces
                                           C sino
                                           Añadir(Retirar(C, e2), e1)
Pertenece(Conjuntovacio, e1)         ⇒  falso
Pertenece(Añadir(C, e2), e1)         ⇒  si e1 = e2 entonces cierto sino
                                           Pertenece(C, e1)
Esvacio(Conjuntovacio)               ⇒  cierto
```

[8] El asterisco representa a un constructor del TAD. En este caso Conjuntovacio y Añadir que crean un conjunto vacío y añaden un elemento a un conjunto.

```
Esvacio(Añadir(C, e1))                ⇒  falso
Cardinal(Conjuntovacio)               ⇒  Cero
Cardinal(Añadir(C, e1))               ⇒  si Pertenece(C, e1) entonces
                                         Cardinal(C) sino 1 + Cardinal(C)
Union(Conjuntovacio, Conjuntovacio)   ⇒  Conjuntovacio
Union(Conjuntovacio, Añadir(C, e1))   ⇒  Añadir(C, e1)
Union(Añadir(C, e1), D)               ⇒  Añadir(Union(C, D), e1)
```

1.7. PROGRAMACIÓN ESTRUCTURADA

La programación orientada a objetos se desarrolló para tratar de paliar diversas limitaciones que se encontraban en anteriores enfoques de programación. Para apreciar las ventajas de la POO, es preciso constatar las limitaciones citadas y cómo se producen con los lenguajes de programación tradicionales.

C, Pascal y FORTRAN, y lenguajes similares, se conocen como *lenguajes procedimentales* (por procedimientos). Es decir, cada sentencia o instrucción señala al compilador para que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir por cinco, etc. En resumen, un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias. En el caso de pequeños programas, estos principios de organización (denominados *paradigma)* se demuestran eficientes. El programador sólo tiene que crear esta lista de instrucciones en un lenguaje de programación, compilar en la computadora y ésta, a su vez, ejecuta estas instrucciones.

Cuando los programas se vuelven más grandes, cosa que lógicamente sucede cuando aumenta la complejidad del problema a resolver, la lista de instrucciones aumenta considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Los programadores pueden controlar, de modo normal, unos centenares de líneas de instrucciones. Para resolver este problema los programas se descompusieron en unidades más pequeñas que adoptaron el nombre de *funciones* (*procedimientos*, *subprogramas* o *subrutinas* en otros lenguajes de programación). De este modo un programa orientado a procedimientos se divide en funciones, de modo que cada función tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de romper un programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas *módulos* (normalmente, en el caso de C, denominadas **archivos** o **ficheros**); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias). Esta característica hace que a medida que los programas se hacen más grandes y complejos, el paradigma estructurado comienza a dar señales de debilidad y resulta muy difícil terminar los programas de un modo eficiente. Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos. Tal vez las dos razones más evidentes son éstas; primero, las funciones tienen acceso ilimitado a los datos globales; segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental, proporcionan un modelo pobre del mundo real.

1.7.1. Datos locales y datos globales

En un programa procedimental, por ejemplo escrito en C, existen dos tipos de datos. *Datos locales* que son ocultos en el interior de la función y son utilizados, exclusivamente, por la

función. Estos datos locales están estrechamente relacionados con sus funciones y están protegidos de modificaciones por otras funciones.

Otros tipos de datos son los *datos globales* a los cuales se puede acceder desde *cualquier* función del programa. Es decir, dos o más funciones pueden acceder a los mismos datos siempre que éstos sean globales. En la Figura 1.3 se muestra la disposición de variables locales y globales en un programa procedimental.

Un programa grande (Figura 1.4) se compone de numerosas funciones y datos globales y ello conlleva una multitud de conexiones entre funciones y datos que dificulta su comprensión y lectura.

Todas estas conexiones múltiples originan diferentes problemas. En primer lugar, hacen difícil diseñar la estructura del programa. En segundo lugar, el programa es difícil de modificar ya qué cambios en datos globales pueden necesitar la reescritura de todas las funciones que acceden a los mismos. También puede suceder que estas modificaciones de los datos globales pueden no ser aceptadas por todas o algunas de las funciones.

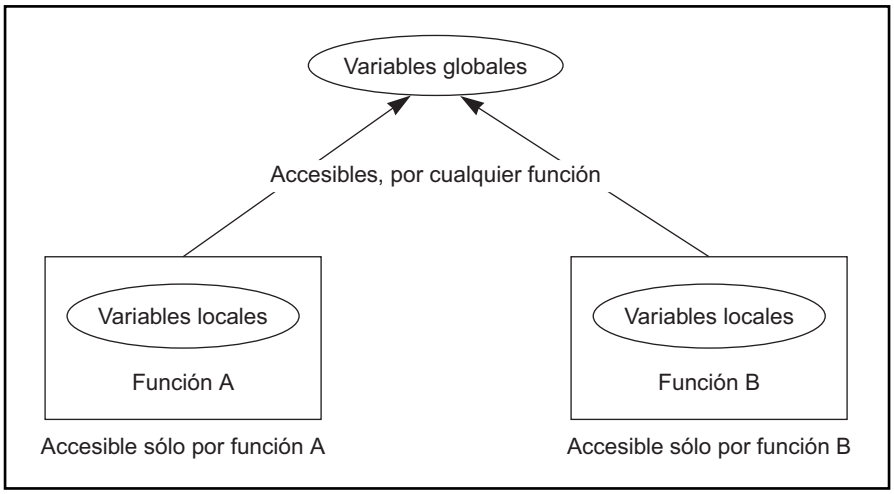


Figura 1.3. Datos locales y globales.

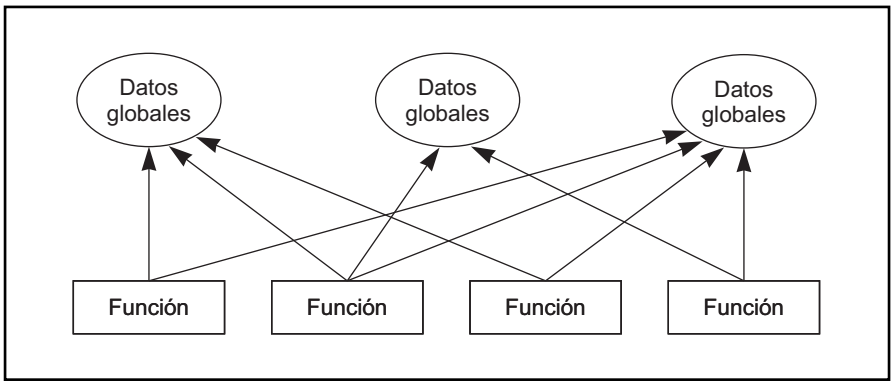


Figura 1.4. Un programa procedimental.

1.7.2. Modelado del mundo real

Otro problema importante de la programación estructurada reside en el hecho de que la disposición separada de datos y funciones no se corresponden con los modelos de las cosas del mundo real. En el mundo físico se trata con objetos físicos tales como personas, autos o aviones. Estos objetos no son como los datos ni como las funciones. Los objetos complejos o no del mundo real tienen *atributos* y *comportamiento.*

Los **atributos** o características de los objetos son, por ejemplo: en las personas, su edad, su profesión, su domicilio, etc.; en un auto, la potencia, el número de matrícula, el precio, número de puertas, etc.; en una casa, la superficie, el precio, el año de construcción, la dirección, etcétera. En realidad, los atributos del mundo real tienen su equivalente en los datos de un programa; toman un valor específico, tal como 200 metros cuadrados, 20.000 dólares, cinco puertas, etc.

El **comportamiento** son las acciones que ejecutan los objetos del mundo real como respuesta a un determinado estímulo. Si usted pisa los frenos en un auto, el coche (carro) se detiene; si acelera, el auto aumenta su velocidad, etc. El comportamiento, en esencia, es como una función: se llama a una función para hacer algo (visualizar la nómina de los empleados de una empresa).

Por estas razones, ni los datos ni las funciones, por sí mismas, modelan los objetos del mundo real de un modo eficiente.

> La programación estructurada mejora la claridad, fiabilidad y facilidad de mantenimiento de los programas; sin embargo, para programas grandes o a gran escala, presentan retos de difícil solución.

1.8. PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos, tal vez el paradigma de programación más utilizado en el mundo del desarrollo de *software* y de la ingeniería de *software* del siglo XXI, trae un nuevo enfoque a los retos que se plantean en la programación estructurada cuando los problemas a resolver son complejos. Al contrario que la programación *procedimental* que enfatiza en los algoritmos, la POO enfatiza en los datos. En lugar de intentar ajustar un problema al enfoque *procedimental* de un lenguaje, POO intenta ajustar el lenguaje al problema. La idea es diseñar formatos de datos que se correspondan con las características esenciales de un problema.

La idea fundamental de los lenguajes orientados a objetos es combinar en una única unidad o módulo, tanto los datos como las funciones que operan sobre esos datos. Tal unidad se llama **objeto**.

Las funciones de un objeto se llaman *funciones miembro* en C++ o *métodos* (éste es el caso de Smalltalk, uno de los primeros lenguajes orientados a objetos), y son el único medio para acceder a sus datos. Los datos de un objeto, se conocen también como *atributos* o *variables de instancia.* Si se desea leer datos de un objeto, se llama a una función miembro del objeto. Se accede a los datos y se devuelve un valor. No se puede acceder a los datos directamente. Los datos están ocultos, de modo que están protegidos de alteraciones accidentales. Los datos y las funciones se dice que están *encapsulados en una única entidad. El encapsulamiento de datos y la ocultación* de los datos son términos clave en la descripción de lenguajes orientados a objetos.

Si se desea modificar los datos de un objeto, se conoce exactamente cuáles son las funciones que interactúan con las funciones miembro del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa. Un programa C++ se compone, normalmente, de un número de objetos que se comunican unos con otros mediante la llamada a otras funciones miembro. La organización de un programa en C++ se muestra en la Figura 1.5. La llamada a una función miembro de un objeto se denomina *enviar un mensaje* a otro objeto.

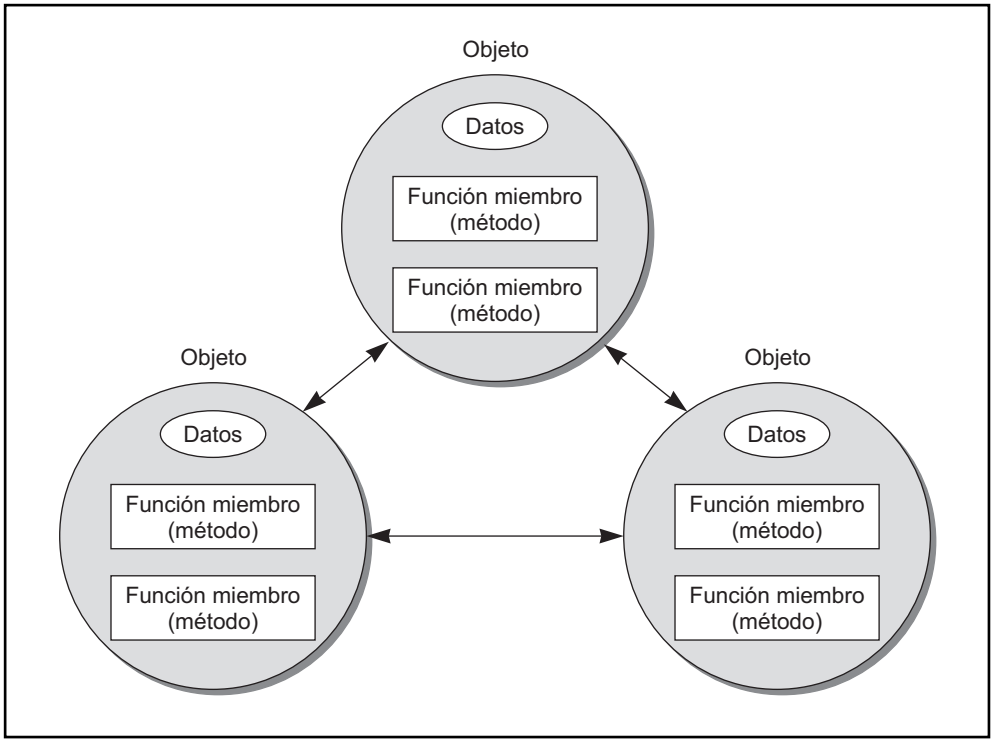


Figura 1.5. Organización típica de un programa orientado a objetos.

En el paradigma orientado a objetos, el programa se organiza como un conjunto finito de objetos que contiene datos y operaciones (funciones miembro en C++) que llaman a esos datos y que se comunican entre sí mediante mensajes.

1.8.1. Propiedades fundamentales de la orientación a objetos

Existen diversas características ligadas a la orientación a objetos. Todas las propiedades que se suelen considerar, no son exclusivas de este paradigma, ya que pueden existir en otros paradigmas, pero en su conjunto definen claramente los lenguajes orientados a objetos. Estas propiedades son:

- *Abstracción* (tipos abstractos de datos y clases).
- *Encapsulado* o *encapsulamiento* de datos.
- *Ocultación* de datos.
- *Herencia*.
- *Polimorfismo*.

C++ soporta todas las características anteriores que definen la orientación a objetos, aunque hay numerosas discusiones en torno a la consideración de C++ como lenguaje orientado a objetos. La razón es que, en contraste con lenguajes tales como Smalltalk, Java o C#, C++ no es un lenguaje orientado a objetos puro. C++ soporta orientación a objetos pero es compatible con C y permite que programas C++ se escriban sin utilizar características orientadas a objetos. De hecho, C++ es un lenguaje *multiparadigma* que permite *programación estructurada*, *procedimental*, *orientada a objetos* y *genérica*.

1.8.2. Abstracción

La abstracción es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta los restantes aspectos. El término **abstracción**, que se suele utilizar en programación, se refiere al hecho de diferenciar entre las propiedades externas de una entidad y los detalles de la composición interna de dicha entidad. Es la abstracción la propiedad que permite ignorar los detalles internos de un dispositivo complejo tal como una computadora, un automóvil, una lavadora o un horno de microondas, etc., y usarlo como una única unidad comprensible. Mediante la abstracción se diseñan y fabrican estos sistemas complejos en primer lugar y, posteriormente, los componentes más pequeños de los cuales están compuestos. Cada componente representa un nivel de abstracción en el cual el uso del componente se aísla de los detalles de la composición interna del componente. La abstracción posee diversos grados denominados niveles de abstracción.

En consecuencia, la abstracción posee diversos grados de complejidad que se denominan *niveles de abstracción* que ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. En el modelado orientado a objetos de un sistema esto significa centrarse en *qué* es *y qué hace* un objeto y no en *cómo* debe implementarse. Durante el proceso de abstracción es cuando se decide qué características y comportamiento debe tener el modelo.

Aplicando la abstracción se es capaz de construir, analizar y gestionar sistemas de computadoras complejos y grandes que no se podrían diseñar si se tratara de modelar a un nivel detallado. En cada nivel de abstracción se visualiza el sistema en términos de componentes, denominados **herramientas abstractas**, cuya composición interna se ignora. Esto nos permite concentrarnos en cómo cada componente interactúa con otros componentes y centrarnos en la parte del sistema que es más relevante para la tarea a realizar en lugar de perderse a nivel de detalles menos significativos.

En estructuras o registros, las propiedades individuales de los objetos se pueden almacenar en los miembros. Para los objetos es de interés *cómo* están organizados sino también *qué* se puede hacer con ellos. Es decir, las operaciones que forman la composición interna de un objeto son también importantes. El primer concepto en el mundo de la orientación a objetos nació con los tipos abstractos de datos (**TAD**). Un tipo abstracto de datos describe no sólo los atributos de un objeto, sino también su comportamiento (las operaciones). Esto puede incluir también una descripción de los estados que puede alcanzar un objeto.

Un medio de reducir la complejidad es la abstracción. Las características y los procesos se reducen a las propiedades esenciales, son resumidas o combinadas entre sí. De este modo, las características complejas se hacen más manejables.

EJEMPLO 1.7. Diferentes modelos de abstracción del término coche (carro).

- Un `coche (carro)` es la combinación (o composición) de diferentes partes, tales como motor, carrocería, cuatro ruedas, cinco puertas, etc.
- Un `coche (carro)` es un concepto común para diferentes tipos de coches. Pueden clasificarse por el nombre del fabricante (Audi, BMW, Seat, Toyota, Chrisler...), por su categoría (turismo, deportivo, todoterreno...), por el carburante que utilizan (gasolina, gasoil, gas, híbrido...).

La abstracción `coche` se utilizará siempre que la marca, la categoría o el carburante no sean significativos. Así, un `carro (coche)` se utilizará para transportar personas o ir de `Carchelejo` a `Cazorla`.

1.8.3. Encapsulación y ocultación de datos

El *encapsulado* o *encapsulación de datos* es el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación. En el caso de los objetos que poseen las mismas características y comportamiento se agrupan en clases, que no son más que unidades o módulos de programación que encapsulan datos y operaciones.

La ocultación de datos permite separar el aspecto de un componente, definido por su *interfaz* con el exterior, de sus detalles internos de implementación. Los términos **ocultación** de la **información** (*information hiding*) y **encapsulación de datos** (*data encapsulation*) se suelen utilizar como sinónimos, pero no siempre es así, a veces se utilizan en contextos diferentes. En C++ no es lo mismo, los datos internos están protegidos del exterior y no se pueden acceder a ellos más que desde su propio interior y, por tanto, no están ocultos. El acceso al objeto está restringido sólo a través de una interfaz bien definida.

El diseño de un programa orientado a objetos contiene, al menos, los siguientes pasos:

1. Identificar los *objetos* del sistema.
2. Agrupar en *clases* a todos los objetos que tengan características y comportamiento comunes.
3. Identificar los *datos* y *operaciones* de cada una de las clases.
4. Identificar las *relaciones* que pueden existir entre las clases.

En C++, un **objeto** es un elemento individual con su propia identidad; por ejemplo, un libro, un automóvil... Una **clase** puede describir las propiedades genéricas de un ejecutivo de una empresa (`nombre`, `título`, `salario`, `cargo`...) mientras que un objeto representará a un ejecutivo específico (`Luis Mackoy`, director general). En general, una clase define qué datos se utilizan para representar un objeto y las operaciones que se pueden ejecutar sobre esos datos.

Cada clase tiene sus propias características y comportamiento; en general, una clase define los datos que se utilizan y las operaciones que se pueden ejecutar sobre esos datos. Una clase describe un conjunto de objetos. En el sentido estricto de programación, una clase es un tipo de datos. Diferentes variables se pueden crear de este tipo. En programación orientada a objetos, éstas se llaman *instancias.* Las instancias son, por consiguiente, la realización de los objetos descritos en una clase. Estas instancias constan de datos o atributos descritos en la clase y se pueden manipular con las operaciones definidas dentro de ellas.

Los términos *objeto* e *instancia* se utilizan frecuentemente como sinónimos (especialmente en C++). Si una variable de tipo `Carro` se declara, se crea un objeto `Carro` (una instancia de la clase `Carro`).

Las operaciones definidas en los objetos se llaman *métodos*. Cada operación llamada por un objeto se interpreta como un *mensaje* al objeto, que utiliza un método específico para procesar la operación.

En el diseño de programas orientados a objetos se realiza en primer lugar el diseño de las clases que representan con precisión aquellas cosas que trata el programa. Por ejemplo, un programa de dibujo, puede definir clases que representan rectángulos, líneas, pinceles, colores, etcétera. Las definiciones de clases, incluyen una descripción de operaciones permisibles para cada clase, tales como desplazamiento de un círculo o rotación de una línea. A continuación se prosigue el diseño de un programa utilizando objetos de las clases.

El diseño de clases fiables y útiles puede ser una tarea difícil. Afortunadamente, los lenguajes POO facilitan la tarea ya que incorporan clases existentes en su propia programación. Los fabricantes de *software* proporcionan numerosas bibliotecas de clases, incluyendo bibliotecas de clases diseñadas para simplificar la creación de programas para entornos tales como Windows, Linux, Macintosh, Unix o Vista. Uno de los beneficios reales de C++ es que permite la reutilización y adaptación de códigos existentes y ya bien probados y depurados.

1.8.4. Objetos

El objeto es el centro de la programación orientada a objetos. Un **objeto** es algo que se visualiza, se utiliza y juega un rol o papel. Si se programa con enfoque orientado a objetos, se intenta descubrir e implementar los objetos que juegan un rol en el dominio del problema y en consecuencia del programa. La estructura interna y el comportamiento de un objeto, en una primera fase, no tiene prioridad. Es importante que un objeto tal como un carro o una casa juegan un rol.

Dependiendo del problema, diferentes aspectos de un dominio son relevantes. Un carro puede ser ensamblado por partes tales como un motor, una carrocería, unas puertas o puede ser descrito utilizando propiedades tales como su velocidad, su kilometraje o su fabricante. Estos atributos indican el objeto. De modo similar una persona, también se puede ver como un objeto, del cual se disponen diferentes atributos. Dependiendo de la definición del problema, esos atributos pueden ser el nombre, apellido, dirección, número de teléfono, color del cabello, altura, peso, profesión, etc.

Un objeto no necesariamente ha de realizar algo concreto o tangible. Puede ser totalmente abstracto y también puede describir un proceso. Por ejemplo, un partido de baloncesto o de *rugby* puede ser descrito como un objeto. Los atributos de este objeto pueden ser los jugadores, el entrenador, la puntuación y el tiempo transcurrido de partido.

Cuando se trata de resolver un problema con orientación a objetos, dicho problema no se descompone en funciones como en programación estructurada tradicional, caso de C, sino en objetos. El pensar en términos de objetos tiene una gran ventaja: se asocian los objetos del problema a los objetos del mundo real.

¿Qué tipos de cosas son objetos en los programas orientados a objetos? La respuesta está limitada por su imaginación aunque se pueden agrupar en categorías típicas que facilitarán su búsqueda en la definición del problema de un modo más rápido y sencillo.

- Recursos Humanos:
  - — Empleados.
  - — Estudiantes.
  - — Clientes.

— Vendedores.
— Socios.

- Colecciones de datos:

— Arrays (arreglos).
— Listas.
— Pilas.
— Árboles.
— Árboles binarios.
— Grafos.

- Tipos de datos definidos por usuarios:

— Hora.
— Números complejos.
— Puntos del plano.
— Puntos del espacio.
— Ángulos.
— Lados.

- Elementos de computadoras:

— Menús.
— Ventanas.
— Objetos gráficos (rectángulos, círculos, rectas, puntos…).
— Ratón (mouse).
— Teclado.
— Impresora.
— USB.
— Tarjetas de memoria de cámaras fotográficas.

- Objetos físicos:

— Carros.
— Aviones.
— Trenes.
— Barcos.
— Motocicletas.
— Casas.

- Componentes de videojuegos:

— Consola.
— Mandos.
— Volante.
— Conectores.
— Memoria.
— Acceso a Internet.

La correspondencia entre objetos de programación y objetos del mundo real es el resultado eficiente de combinar datos y funciones que manipulan esos datos. Los objetos resultantes

ofrecen una mejor solución al diseño del programa que en el caso de los lenguajes orientados a procedimientos.

Un **objeto** se puede definir desde el punto de vista conceptual como una entidad individual de un sistema y que se caracteriza por un estado y un comportamiento. Desde el punto de vista de implementación, un **objeto** es una entidad que posee un conjunto de *datos* y un conjunto de *operaciones* (*funciones* o *métodos*).

El estado de un objeto viene determinado por los valores que toman sus datos, cuyos valores pueden tener las restricciones impuestas en la definición del problema. Los datos se denominan también *atributos* y componen la estructura del objeto y las operaciones —también llamadas *métodos*— representan los servicios que proporciona el objeto.

La representación gráfica de un objeto en **UML** se muestra en la Figura 1.6.

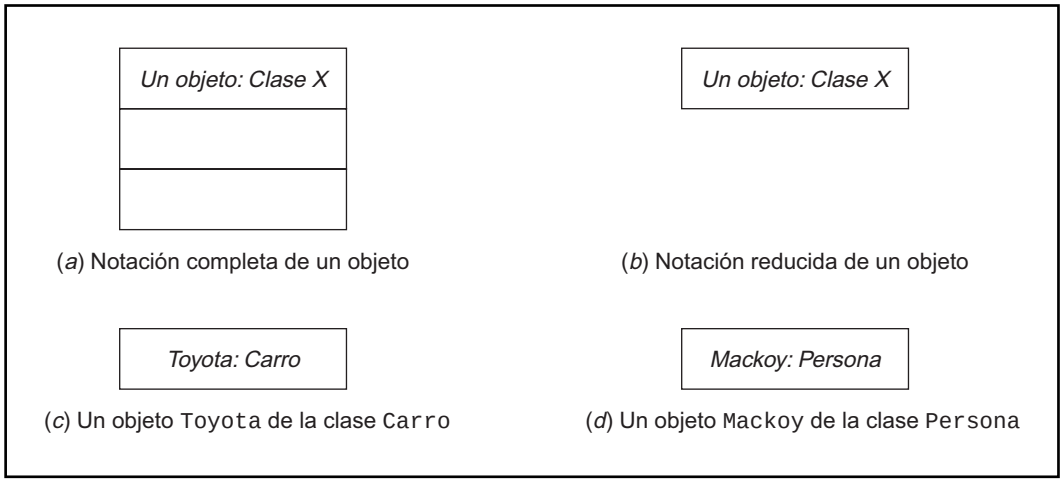


Figura 1.6. Representación de objetos en UML (Lenguaje Unificado de Modelado).

1.8.5. Clases

En POO los objetos son miembros o instancias de clases. En esencia, una **clase** es un tipo de datos al igual que cualquier otro tipo de dato definido en un lenguaje de programación. La diferencia reside en que la clase es un tipo de dato que contiene datos y funciones. Una clase describe muchos objetos y es preciso definirla, aunque su definición no implica creación de objetos (Figura 1.7).

Una **clase** es, por consiguiente, una descripción de un número de objetos similares. `Madonna`, `Sting`, `Prince`, `Juanes`, `Carlos Vives` o `Juan Luis Guerra` son miembros u objetos de la clase `"músicos de rock"`. Un objeto concreto, `Juanes` o `Carlos Vives`, son *instancias* de la clase `"músicos de rock"`.

En **C++** una clase es una estructura de dato o tipo de dato que contiene funciones (métodos) como miembros y datos. Una clase es una descripción general de un conjunto de objetos similares. Por definición, todos los objetos de una clase comparten los mismos atributos (datos) y las mismas operaciones (métodos). Una clase encapsula las abstracciones de datos y operaciones necesarias para describir una entidad u objeto del mundo real.

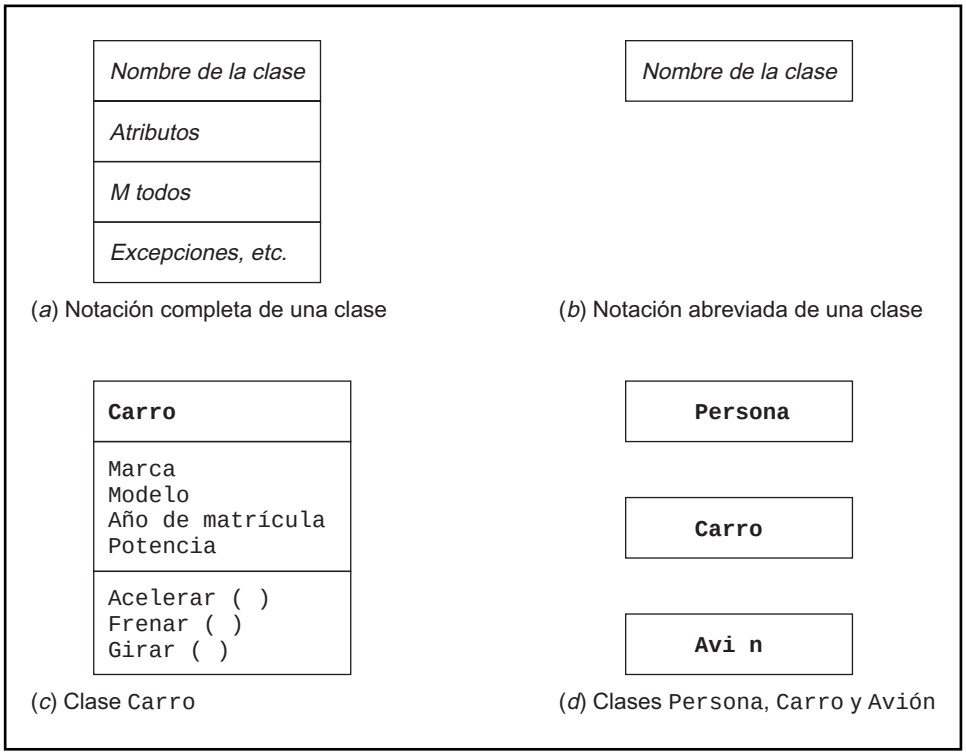


Figura 1.7. Representación de clases en UML.

Una clase se representa en **UML** mediante un rectángulo que contiene en una banda con el nombre de la clase y opcionalmente otras dos bandas con el nombre de sus atributos y de sus operaciones o métodos (Figura 1.8).

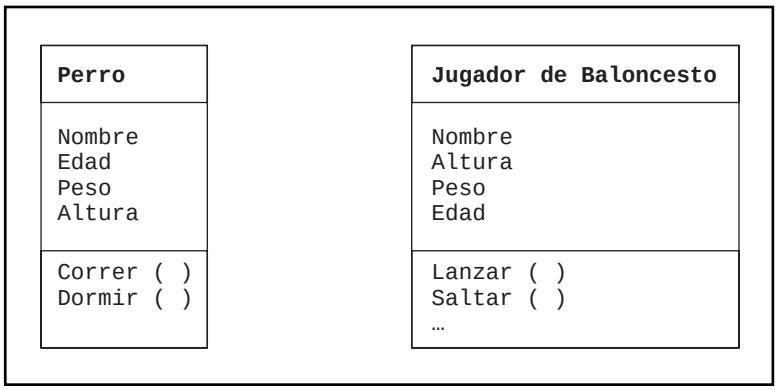


Figura 1.8. Representación de clases en **UML** con atributos y métodos.

1.8.6. Generalización y especialización: herencia

La *generalización* es la propiedad que permite compartir información entre dos entidades evitando la redundancia. En el comportamiento de objetos existen con frecuencia propiedades que son comunes en diferentes objetos y esta propiedad se denomina *generalización*.

Por ejemplo, máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas, etc., son todos electrodomésticos (aparatos del hogar). En el mundo de la orientación a objetos, cada uno de estos aparatos es un **`subclase`** de la clase `Electrodoméstico` y a su vez `Electrodoméstico` es una **`superclase`** de todas las otras clases (`máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas,...`). El proceso inverso de la generalización por el cual se definen nuevas clases a partir de otras ya existentes se denomina *especialización*.

En orientación a objetos, el mecanismo que implementa la propiedad de generalización se denomina **herencia**. La herencia permite definir nuevas clases a partir de otras clases ya existentes, de modo que presentan las mismas características y comportamiento de éstas, así como otras adicionales.

La idea de clases conduce a la idea de herencia. Clases diferentes se pueden conectar unas con otras de modo jerárquico. Como ya se ha comentado anteriormente con las relaciones de generalización y especialización, en nuestras vidas diarias se utiliza el concepto de clases divididas en subclases. La clase `Animal` se divide en `Anfibios`, `Mamíferos`, `Insectos`, `Pájaros`, etc., y la clase `Vehículo` en `Carros`, `Motos`, `Camiones`, `Buses`, etc.

El principio de la división o clasificación es que cada subclase comparte características comunes con la clase de la que procede o se deriva. Los carros, motos, camiones y buses tienen ruedas, motores y carrocerías; son las características que definen a un vehículo. Además de las características comunes con los otros miembros de la clase, cada subclase tiene sus propias características. Por ejemplo, los camiones tienen una cabina independiente de la caja que transporta la carga; los buses tienen un gran número de asientos independientes para los viajeros que ha de transportar, etc. En la Figura 1.9 se muestran clases pertenecientes a una jerarquía o herencia de clases.

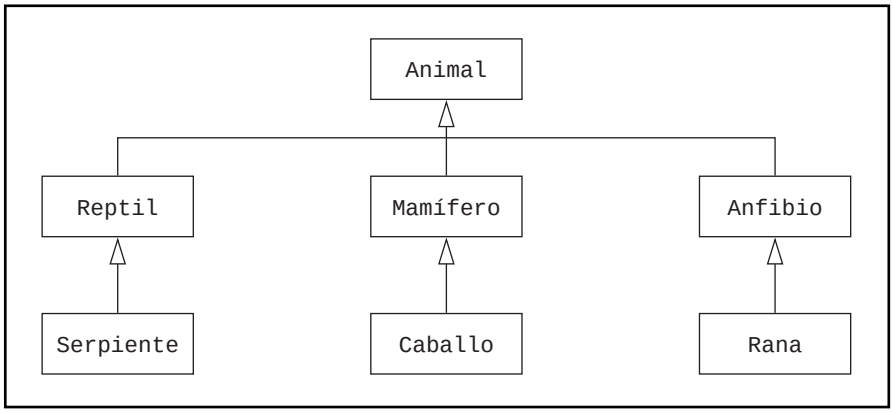


Figura 1.9. Herencia de clases en UML.

De modo similar una clase se puede convertir en padre o raíz de otras subclases. En C++ la clase original se denomina *clase base* y las clases que se derivan de ella se denominan *clases*

derivadas y siempre son una especialización o *concreción* de su clase base. A la inversa, la clase base es la generalización de la clase derivada. Esto significa que todas las propiedades (atributos y operaciones) de la clase base se heredan por la clase derivada, normalmente suplementada con propiedades adicionales.

1.8.7. Reusabilidad

Una vez que una clase ha sido escrita, creada y depurada, se puede distribuir a otros programadores para utilizar en sus propios programas. Esta propiedad se llama *reusabilidad*[9] o *reutilización.* Su concepto es similar a las funciones incluidas en las bibliotecas de funciones de un lenguaje procedimental como C que se pueden incorporar en diferentes programas.

En C++, el concepto de herencia proporciona una extensión o ampliación al concepto de *reusabilidad.* Un programador puede considerar una clase existente y sin modificarla, añadir competencias y propiedades adicionales a ella. Esto se consigue derivando una nueva clase de una ya existente. La nueva clase heredará las características de la clase antigua, pero es libre de añadir nuevas características propias.

La facilidad de reutilizar o reusar el *software* existente es uno de los grandes beneficios de la POO: muchas empresas consiguen con la reutilización de clases en nuevos proyectos la reducción de los costes de inversión en sus presupuestos de programación. ¿En esencia cuáles son las ventajas de la herencia? Primero, se utiliza para consistencia y reducir código, propiedades comunes de varias clases sólo necesitan ser implementadas una vez y sólo necesitan modificarse una vez si es necesario. Tercero, el concepto de abstracción de la funcionalidad común está soportada.

1.8.8. Polimorfismo

Además de las ventajas de consistencia y reducción de código, la herencia, aporta también otra gran ventaja: facilitar el polimorfismo. **Polimorfismo** es la propiedad de que un operador o una función actúen de modo diferente en función del objeto sobre el que se aplican. En la practica, el polimorfismo significa la capacidad de una operación de ser interpretada sólo por el propio objeto que lo invoca. Desde un punto de vista práctico de ejecución del programa, el polimorfismo se realiza en tiempo de ejecución ya que durante la compilación no se conoce qué tipo de objeto y por consiguiente qué operación ha sido llamada.

La propiedad de **polimorfismo** es aquella en que una operación tiene el mismo nombre en diferentes clases, pero se ejecuta de diferentes formas en cada clase. Así, por ejemplo, la operación de `abrir` se puede dar en diferentes clases: abrir una puerta, abrir una ventana, abrir un periódico, abrir un archivo, abrir una cuenta corriente en un banco, abrir un libro, etc. En cada caso se ejecuta una operación diferente aunque tiene el mismo nombre en todos ellos `"abrir"`. El polimorfismo es la propiedad de una operación de ser interpretada sólo por el objeto al que pertenece. Existen diferentes formas de implementar el polimorfismo y variará dependiendo del lenguaje de programación. Veamos el concepto con ejemplos de la vida diaria.

En un taller de reparaciones de automóviles existen numerosos carros (coches), de marcas diferentes, de modelos diferentes, de tipos diferentes, potencias diferentes, etc. Constituyen una

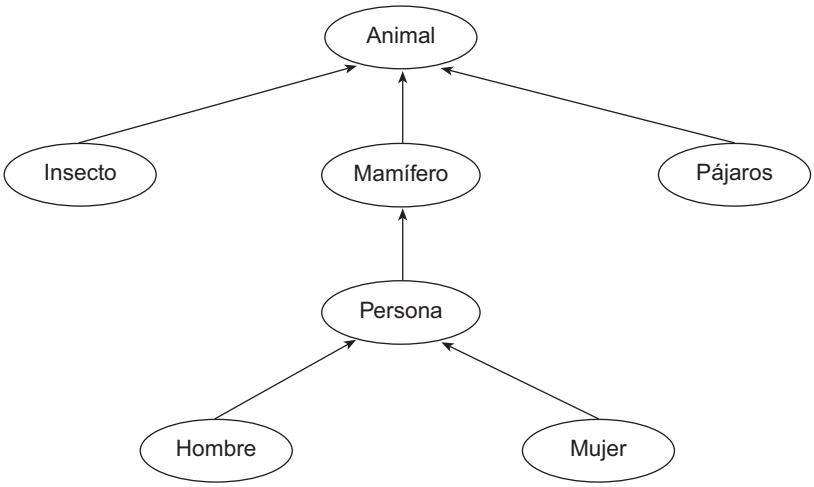
[9] El término proviene del concepto ingles *reusability*. La traducción no ha sido aprobada por la RAE, pero se incorpora al texto por su gran uso y difusión entre los profesionales de la informática.

clase o colección heterogénea de carros (coches). Supongamos que se ha de realizar una operación común `"cambiar los frenos del carro"`. La operación a realizar es la misma, incluye los mismos principios; sin embargo, dependiendo del coche, en particular, la operación será muy diferente, incluirá diferentes acciones en cada caso. Otro ejemplo a considerar y relativo a los operadores "+" y "*" aplicados a números enteros o números complejos; aunque ambos son números, en un caso la suma y multiplicación son operaciones simples, mientras que en el caso de los números complejos al componerse de parte real y parte imaginaria, será necesario seguir un método específico para tratar ambas partes y obtener un resultado que también será un número complejo.

El uso de operadores o funciones de forma diferente, dependiendo de los objetos sobre los que están actuando se llama polimorfismo (una cosa con diferentes formas). Sin embargo, cuando un operador existente, tal como + o =, se le permite la posibilidad de operar sobre nuevos tipos de datos, se dice entonces que el operador está sobrecargado. La sobrecarga es un tipo de polimorfismo y una característica importante de la POO.

EJEMPLO 1.8. Definir una jerarquía de clases para: animal, insecto, mamíferos, pájaros, persona hombre y mujer. Realizar un definición en pseudocódigo de las clases.

Las clases de objeto `Mamífero`, `Pájaro` e `Insecto` se definen como *subclases* de `Animal`; la clase de objeto `Persona`, como una subclase de `Mamífero`, y un `Hombre` y una `Mujer` son subclases de `Persona`.



Las definiciones de clases para esta jerarquía puede tomar la siguiente estructura:

```
clase Animal
  atributos (propiedades)
    string: tipo;
    real: peso;
    (...algun tipo de habitat...):habitat;
  operaciones
```

```
    crear()→ criatura;
    predadores(criatura)→ fijar(criatura);
    esperanza_vida(criatura) → entero;
    ...
fin criatura

clase Insecto hereda Animal
    atributos (propiedades)
      cadena: nombre
    operaciones
    ...
fin Insecto

clase Mamifero hereda Animal;
    atributos (propiedades)
      real: periodo_gestacion;
    operaciones
      ...
fin Mamifero

clase Pajaro hereda Animal
    atributos (propiedades)
      cadena: nombre
      entero: Alas
    operaciones
    ...
fin Pajaro

clase Persona hereda Mamifero;
    atributos (propiedades)
      string: apellido, nombre;
      date: fecha_nacimiento;
      pais: origen;
fin Persona

clase Hombre hereda Persona;
    atributos (propiedades)
      mujer: esposa;
      ...
    operaciones
      ...
fin Hombre

clase Mujer hereda Persona;
    propiedades
      esposo: hombre;
      string: nombre;
      ...
fin Mujer
```

RESUMEN

Un método general para la resolución de un problema con computadora tiene las siguientes fases:

1. *Análisis del programa*.
2. *Diseño del algoritmo*.
3. *Codificación.*

El sistema más idóneo para resolver un problema es descomponerlo en módulos más sencillos y luego, mediante diseños descendentes y refinamiento sucesivo, llegar a módulos fácilmente codificables. Estos módulos se deben codificar con las estructuras de control de programación estructurada.

Los tipos abstractos de datos (**TAD**) describen un conjunto de objetos con la misma representación y comportamiento. Los tipos abstractos de datos presentan una separación clara entre la interfaz externa de un tipo de datos y su implementación interna. La implementación de un tipo abstracto de datos está oculta. Por consiguiente, se pueden utilizar implementaciones alternativas para el mismo tipo abstracto de dato sin cambiar su interfaz.

La especificación de un tipo abstracto de datos se puede hacer de manera *informal*, o bien, de forma más rigurosa, una especificación *formal*. En la especificación *informal* se describen literalmente los datos y la funcionalidad de las operaciones. La especificación formal describe los datos, la sintaxis y la semántica de las operaciones considerando ciertas operaciones como axiomas, que son los constructores de nuevos datos. Una buena especificación formal de un tipo abstracto de datos debe poder verificar la *bondad* de la implementación.

Las características más importantes de la programación orientada a objetos son: abstracción, encapsulamiento, herencia, polimorfismo, clases y objetos.

Una **clase** es un tipo de dato definido por el usuario que sirve para representar objetos del mundo real. Un objeto de una clase tiene dos componentes: un conjunto de atributos y un conjunto de operaciones. Un objeto es una instancia de una clase. Herencia es la relación entre clases que se produce cuando una nueva clase se crea utilizando las propiedades de una clase ya existente. Polimorfismo es la propiedad por la que un mensaje puede significar cosas diferentes dependiendo del objeto que lo recibe.

EJERCICIOS

1.1. Diseñar el diagrama de flujo de algoritmo que visualice y sume la serie de números 4, 8, 12, 16, ..., 400.

1.2. Diseñar un diagrama de flujo y un algoritmo para calcular la velocidad (en m/s) de los corredores de la carrera de 1.500 metros. La entrada consistirá en parejas de números (minutos, segundos) que dan el tiempo del corredor; por cada corredor, el algoritmo debe imprimir el tiempo en minutos y segundos así como la velocidad media. Ejemplo de entrada de datos: (3,53) (3,40) (3,46) (3,52) (4,0) (0,0); el último par de datos se utilizará como fin de entrada de datos.

1.3. Escribir un algoritmo que encuentre el salario semanal de un trabajador, dada la tarifa horaria y el número de horas trabajadas diariamente.

1.4. Escribir un diagrama de flujo y un algoritmo que cuente el número de ocurrencias de cada letra en una palabra leída como entrada. Por ejemplo, "`Mortimer`" contiene dos "`m`", una "`o`", dos "`r`", una "`y`", una "`t`" y una "`e`".

1.5. Dibujar un diagrama jerárquico de objetos que represente la estructura de un coche (carro).

1.6. Dibujar diagramas de objetos que representen la jerarquía de objetos del modelo `Figura` geométrica.

1.7. Construir el **TAD** Natural para representar a los números naturales, con las operaciones: `Cero`, `Sucesor`, `EsCero`, `Igual`, `Suma`, `Antecesor`, `Diferencia` y `Menor`. Realizar la especificación informal y formal considerando como constructores las operaciones `Cero` y `Sucesor`.

1.8. Diseñar el **TAD** Bolsa como una colección de elementos no ordenados y que pueden estar repetidos. Las operaciones del tipo abstracto: `CrearBolsa`, `Añadir` (un elemento), `BolsaVacia` (verifica si tiene elementos), `Dentro` (verifica si un elementos pertenece a la bolsa), `Cuantos` (determina el número de veces que se encuentra un elemento), `Union` y `Total`. Realizar la especificación informal y formal considerando como constructores las operaciones `CrearBolsa` y `Añadir`.

1.9. Diseñar el **TAD** `Complejo` para representar a los números complejos. Las operaciones que se deben definir: `AsignaReal` (asigna un valor a la parte real), `AsignaImaginaria` (asigna un valor a la parte imaginaria), `ParteReal` (devuelve la parte real de un complejo), `ParteImaginaria` (devuelve la parte imaginaria de un complejo), `Modulo` de un complejo y `Suma` de dos números complejos. Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

1.10 . Diseñar el tipo abstracto de datos Matriz con la finalidad de representar matrices matemáticas. Las operaciones a definir: `CrearMatriz` (crea una matriz, sin elementos, de `m` filas por `n` columnas), `Asignar` (asigna un elemento en la fila *`i`* columna *`j`*), `ObtenerElemento` (obtiene el elemento de la fila *`i`* y columna *`j`*), `Sumar` (realiza la suma de dos matrices de las mismas dimensiones), `ProductoEscalar` (obtiene la matriz resultante de multiplicar cada elemento de la matriz por un valor). Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

CAPÍTULO 2
Clases y objetos

Objetivos

Con el estudio de este capítulo usted podrá:

- Entender el concepto de encapsulación de datos a través de las clases.
- Definir clases como una estructura que encierra datos y métodos.
- Especificar tipos abstractos de datos a través de una clase.
- Establecer controles de acceso a los miembros de una clase.
- Identificar los métodos de una clase con el comportamiento o funcionalidad de los objetos.

Contenido

Conceptos clave

- Componentes.
- Constructores.
- Encapsulación.
- Especificadores de acceso: `public`, `protected`, `private`.
- Ocultación de la información.
- Reutilización.

INTRODUCCIÓN

La modularización de un programa utiliza la noción de *tipo abstracto de dato* (**TAD**) siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado **TAD**.

Una **clase** es un tipo de dato que contiene código (métodos) y datos. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, tal como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. En el capítulo se aprenderá a crear (definir y especificar) y utilizar clases individuales.

2.1. CLASES Y OBJETOS

Las tecnologías orientadas a objetos combinan la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos como instancias o ejemplares de ellas, son los elementos clave sobre los que se articula la orientación a objetos.

2.1.1. ¿Qué son objetos?

En el mundo real, las personas identifican los objetos como cosas que pueden ser percibidas por los cinco sentidos. Los objetos tienen propiedades específicas, tales como posición, tamaño, color, forma, textura, etc., que definen su estado. Los objetos también tienen ciertos comportamientos que los hacen diferentes de otros objetos.

Booch[1], define un *objeto* como "algo que tiene un estado, un comportamiento y una identidad". Supongamos una máquina de una fábrica. El *estado* de la *máquina* puede estar *funcionando/parada* ("on/of"), su potencia, velocidad máxima, velocidad actual, temperatura, etc. Su *comportamiento* puede incluir acciones para arrancar y parar la máquina, obtener su temperatura, activar o desactivar otras máquinas, condiciones de señal de error o cambiar la velocidad. Su *identidad* se basa en el hecho de que cada instancia de una máquina es única, tal vez identificada por un número de serie. Las características que se eligen para enfatizar en el estado y el comportamiento se apoyarán en cómo un objeto máquina se utilizará en una aplicación. En un diseño de un programa orientado a objetos, se crea una abstracción (un modelo simplificado) de la máquina basado en las propiedades y comportamiento que son útiles en el tiempo.

Martin/Odell definen un objeto como "cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos".

Un *mensaje* es una instrucción que se envía a un objeto y que cuando se recibe ejecuta sus acciones. Un mensaje incluye un identificador que contiene la acción que ha de ejecutar el objeto junto con los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario de un objeto se comunica con el objeto mediante su *interfaz*, un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. Una

[1] Booch, Grady. *Análisis y diseño orientado a objetos con aplicaciones*. Madrid : Díaz de Santos/Addison-Wesley, 1995. (libro traducido del inglés por Luis Joyanes, coautor de esta obra).

interfaz se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico tal como una máquina de fax tiene una interfaz de usuario bien definida; esa interfaz incluye el mecanismo de avance del papel, botones de marcado, receptor y el botón "enviar". El usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles.

2.1.2. ¿Qué son clases?

En términos prácticos, una **clase** es un tipo definido por el usuario. Las clases son los bloques de construcción fundamentales de los programas orientados a objetos. Booch denomina a una clase como "un conjunto de objetos que comparten una estructura y comportamiento comunes".

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como *servicios* o *métodos.* Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de la clase. Estos datos se conocen como *atributos, variables* o *variables instancia*. El término *atributo* se utiliza en análisis y diseño orientado a objetos y el término *variable instancia* se suele utilizar en programas orientados a objetos.

2.2. DECLARACIÓN DE UNA CLASE

Antes de que un programa pueda crear objetos de cualquier clase, la clase debe ser *declarada* y los métodos definidos (implementados). La declaración de una clase significa dar a la misma un nombre, darle nombre a los elementos que almacenan sus datos y describir los métodos que realizarán las acciones consideradas en los objetos.

Formato

```
class NombreClase
{
     lista_de_miembros
};
```

NombreClase	Nombre definido por el usuario que identifica a la clase (puede incluir letras, números y subrayados como cualquier identificador).
lista_de_miembros	Datos y funciones miembros de la clase.

Las *declaraciones* o *especificaciones* no son código de programa ejecutable. Se utilizan para asignar almacenamiento a los valores de los atributos usados por el programa y reconocer los métodos que utilizará el programa. En C++ los métodos se denominan funciones miembro, normalmente en la declaración sólo se escribe el prototipo de la función. Las declaraciones de las clases se sitúan en archivos `.h` (`NombreClase.h`) y la implementación de las funciones miembro en el archivo `.cpp` (`NombreClase.cpp`).

EJEMPLO 2.1. Definición de una clase llamada `Punto` que contiene las coordenadas *x* e *y* de un punto en un plano.

La declaración de la clase se guarda en el archivo *Punto.h*:

```
//archivo Punto.h
class Punto
{
private:
   int x, y;                    // coordenadas x, y
public:
   Punto(int x_, int y_)        // constructor
   {
       x = x_;
       y = y_;
   }
   Punto() { x = y = 0;}        // constructor sin argumentos
   int leerX() const;           // devuelve el valor de x

   int leerY() const;           // devuelve el valor de y
   void fijarX(int valorX)      // establece el valor de x
   void fijarY(int valorY)      // establece el valor de y
};
```

La definición de las funciones miembro se realiza en el archivo *Punto.cpp*:

```
#include "Punto.h"
int Punto::leerX() const
{
    return x;
}
int Punto::leerY() const
{
    return y;
}
void Punto::fijarX(int valorX)
{
    x = valorX;
}
void Punto::fijarY(int valorY)
{
    y = valorY;
}
```

2.2.1. Objetos

Una vez que una clase ha sido definida, un programa puede contener una *instancia* de la clase, denominada un *objeto de la clase*. Un objeto se crea de forma estática, de igual forma que se define una variable. También de forma dinámica, con el operador `new` aplicado a un constructor de la clase. Por ejemplo, un objeto de la clase `Punto` inicializado a las coordenadas `(2,1)`:

```
Punto p1(2, 1);                 // objeto creado de forma estática
Punto* p2 = new Punto(2, 1);    // objeto creado dinámicamente
```

Formato para crear un objeto

```
NombreClase varObj(argumentos_constructor);
```

Formato para crear un objeto dinámico

```
NombreClase* ptrObj;
ptrObj = new NombreClase(argumentos_constructor);
```

Toda clase tiene una o más funciones miembro, denominadas constructores, para inicializar el objeto cuando es creado; tienen el mismo nombre que el de la clase, no tienen tipo de retorno y pueden estar sobrecargados.

El *operador de acceso* a un miembro del objeto, selector punto (`.`), selecciona un miembro individual de un objeto de la clase. Por ejemplo:

```
Punto p2;                       // llama al constructor sin argumentos
p2.fijarX(10);
cout << " Coordenada  x es " << p2.leerX();
```

El otro *operador de acceso* es el selector *flecha* (`->`), selecciona un miembro de un objeto desde un puntero a la clase. Por ejemplo:

```
Punto* p;
p = new Punto(2, -5);           // crea objeto dinámico
cout << " Coordenada  y es " << p -> leerY();
```

2.2.2. Visibilidad de los miembros de la clase

Un principio fundamental en programación orientada a objetos es la *ocultación de la información*. Significa que determinados datos del interior de una clase no se puede acceder por funciones externas a la clase. El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos *privados*. A los datos o métodos privados sólo se puede acceder desde dentro de la clase. Por el contrario, los datos o métodos públicos son accesibles desde el exterior de la clase (véase la Figura 2.1).

Para controlar el acceso a los miembros de la clase se utilizan tres diferentes *especificadores de acceso:* `public`, `private` y `protected`. Cada miembro de la clase está precedido del especificador de acceso que le corresponde.

Formato

```
class NombreClase
{
private:
     declaraciones de miembros privados;
protected:
     declaraciones de miembros protegidos;
```

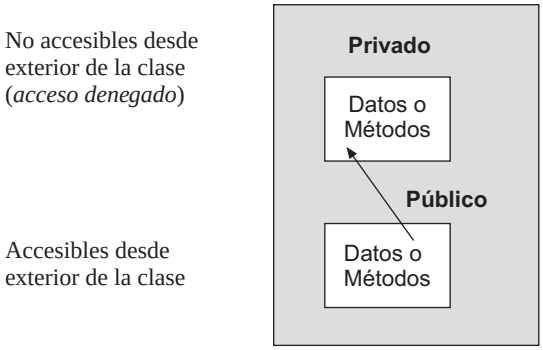


Figura 2.1. Secciones pública y privada de una clase.

```
public:
     declaraciones de miembros públicos;
};
```

Por omisión, los miembros que aparecen a continuación de la llave de inicio de la clase, {, son privados. A los miembros que siguen a la etiqueta `private` sólo se puede acceder por funciones miembro de la misma clase. A los miembros `protected` sólo se puede acceder por funciones miembro de la misma clase y de las clases derivadas. A los miembros que siguen a la etiqueta `public` se puede acceder desde dentro y desde el exterior de la clase. Las secciones `public`, `protected` y `private` pueden aparecer en cualquier orden.

EJEMPLO 2.2. Declaración de la clase `Foto` y `Marco` con miembros declarados con distinta visibilidad.

```
class Foto
{
private:
     int nt;
     char opd;
protected:
     string q;
public:
     Foto(string r)    // constructor
     {
         nt = 0;
         opd = 'S';
         q = r;
     }
     double mtd();
};

class Marco
{
private:
     double p;
     string t;
```

```
public:
     Marco();      // constructor
     void poner()
     {
         Foto* u = new Foto("Paloma");
         p = u -> mtd();
         t = "**" + u -> q + "**";
     }
};
```

Tabla 2.1. Visibilidad, "x" indica que el acceso está per mitido

Tipo de miembro	Miembro de la misma clase	Miembro de una clase derivada	Miembro de otra clase (externo)
`private`	x		
`protected`	x	x	
`public`	x	x	x

Aunque las secciones *públicas*, *privadas* y *protegidas* pueden aparecer en cualquier orden, los programadores suelen seguir ciertas reglas en el diseño que citamos a continuación, y que usted puede elegir la que considere más eficiente.

1. Poner los miembros privados primero, debido a que contiene los atributos (datos).
2. Se pone los miembros públicos primero debido a que los métodos y los constructores son la interfaz del usuario de la clase.

En realidad, tal vez el uso más importante de los especificadores de acceso es implementar la ocultación de la información. El *principio de ocultación* de la información indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida que permite que los detalles de implementación de los objetos sean ignorados. Por consiguiente, los datos y métodos públicos forman la interfaz externa del objeto, mientras que los elementos *privados* son los aspectos internos del objeto que no necesitan ser accesibles para usar el objeto.

> El principio de *encapsulamiento* significa que las estr ucturas de datos inter nas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

2.2.3. Funciones miembro de una clase

La declaración de una clase incluye la declaración o prototipo de las funciones miembros (métodos). Aunque la implementación se puede incluir dentro del cuerpo de la clase (`inline`), normalmente se realiza en otro archivo (con extensión `.cpp`) que constituye la definición de la clase. La Figura 2.2 muestra la declaración de la clase `Producto`.

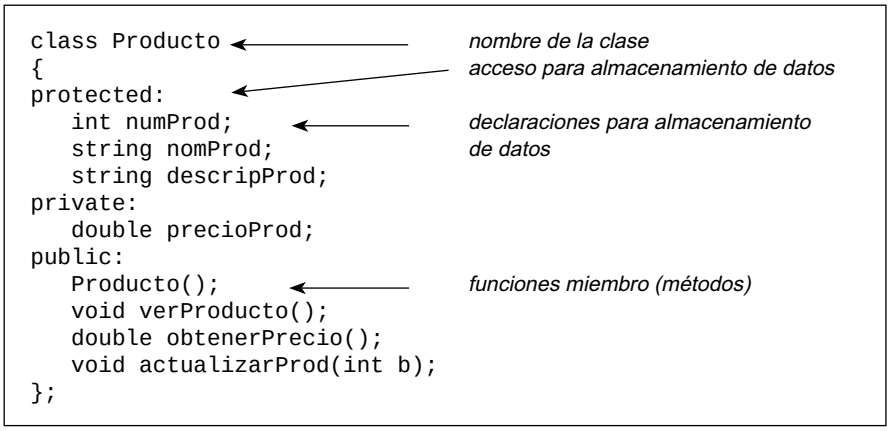


Figura 2.2. Definición típica de una clase.

EJEMPLO 2.3. La clase `Racional` representa un número racional. Por cada dato, numerador y denominador, se proporciona una función miembro que devuelve su valor y otra función para asignar numerador y denominador. Tiene un constructor que inicializa un objeto a 0/1.

En esta ocasión las funciones miembro se implementan directamente en el cuerpo de la clase.

```
   // archivo Racional.h
class Racional
{
private:
    int numerador;
    int denominador;
public:
    Racional()
    {
        numerador = 0;
        denominador = 1;
    }
    int leerN() const { return numerador; }
    int leerD() const { return denominador; }
    void fijar (int n, int d)
    {
        numerador = n;
        denominador = d;
    }
};
```

2.2.4. Funciones en línea y fuera de línea

Las funciones miembro definidas dentro del cuerpo de la declaración de la clase se denominan definiciones de funciones *en línea* (`inline`). Para el caso de funciones más grandes, es preferible codificar sólo el prototipo de la función dentro del bloque de la clase y codificar la im-

plementación de la función en el exterior. Esta forma permite al creador de una clase ocultar la implementación de la función al usuario de la clase proporcionando sólo el código fuente del archivo de cabecera, junto con un archivo de implementación de la clase precompilada.

En el siguiente ejemplo, `FijarEdad()` de la clase `Lince` se declara pero no se define en la declaración de la clase:

```
class Lince
{
public:
   void FijarEdad(int a);
private:
   int edad;
   string habitat;
};
```

La implementación de una función miembro externamente a la declaración de la clase, se hace en una definición de la función *fuera de línea*. Su nombre debe ser precedido por el nombre de la clase y el signo de puntuación :: denominado *operador de resolución de ámbito*. El operador :: permite al compilador conocer que `FijarEdad()` pertenece a la clase `Lince` y es, por consiguiente, diferente de una función global que pueda tener el mismo nombre o de una función que tenga ese nombre que puede existir en otra clase. La siguiente función global, por ejemplo, puede coexistir dentro del mismo ámbito que `Lince::FijarEdad()`:

```
// función global:
void FijarEdad(int valx)
{
   // ...
}
// función en el ámbito de Lince:
void Lince::FijarEdad(int a)
{
    edad = a;
}
```

2.2.5. La palabra reservada `inline`

La decisión de elegir funciones en línea y fuera de línea es una cuestión de eficiencia en tiempo de ejecución. Una función en línea se ejecuta normalmente más rápida, ya que el compilador inserta una copia «fresca» de la función en un programa en cada punto en que se llama a la función. La definición de una función miembro en línea no garantiza que el compilador lo haga realmente en línea; es una decisión que el compilador toma, basado en los tipos de las sentencias dentro de la función y cada compilador de C++ toma esta decisión de modo diferente.

Si una función se compila en línea, se ahorra tiempo de la UCP (CPU) al no tener que ejecutar una instrucción "*call*" (llamar) para bifurcar a la función y no tener que ejecutar una instrucción `return` para retornar al programa llamador. Si una función es corta y se llama cientos de veces, se puede apreciar un incremento en eficiencia cuando actúa como función en línea.

Una función localizada fuera del bloque de la definición de una clase se puede beneficiar de las ventajas de las funciones en línea si está precedida por la palabra reservada `inline`:

```
inline void Lince::FijarEdad(int a)
{
    edad = a;
}
```

Dependiendo de la implementación de su compilador, las funciones que utilizan la palabra reservada `inline` se puede situar en el mismo archivo de cabecera que la definición de la clase. Las funciones que no utilizan `inline` se sitúan en el mismo módulo de programa, pero no el archivo de cabecera. Estas funciones se sitúan en un archivo `.cpp`.

Ejercicio 2.1

Definir una clase `DiaAnyo` con los atributos `mes` y `día`, los métodos `igual()` y `visualizar()`. El mes se registra como un valor entero (1, Enero, 2, Febrero, etc.). El día del mes se registra en otra variable entera día. Escribir un programa que compruebe si una fecha es su cumpleaños.

La función *principal*, `main()`, crea un objeto `DiaAnyo` y llama al método `igual()` para determinar si coincide la fecha del objeto con la fecha de su cumpleaños, que se ha leído de dispositivo de entrada.

```
// archivo DiaAnyo.h

class DiaAnyo
{
private:
   int dia, mes;
public:
    DiaAnyo(int d, int m);
    bool igual(const DiaAnyo& d) const;
    void visualizar() const;
};
```

La implementación de las funciones miembro se guarda en el archivo `DiaAnyo.cpp`:

```
#include <iostream>
using namespace std;
#include "DiaAnyo.h"

DiaAnyo::DiaAnyo(int d, int m)
{
    dia = d;
    mes = m;
}

bool DiaAnyo::igual(const DiaAnyo& d)const
{
    if ((dia == d.dia) && (mes == d.mes))
        return true;
    else
        return false;
}
```

```
void DiaAnyo::visualizar() const
{
    cout << "mes = " << mes << " , dia = " << dia << endl;
}
```

Por último, el archivo `DemoFecha.cpp` contiene la función `main()`, crea los objetos y se envian mensajes.

```
#include <iostream>
using namespace std;
#include "DiaAnyo.h"

int main()
{
    DiaAnyo* hoy;
    DiaAnyo* cumpleanyos;
    int d, m;

    cout << "Introduzca fecha de hoy, dia: ";
    cin >> d;
    cout << "Introduzca el número de mes: ";
    cin >> m;
    hoy = new DiaAnyo(d, m);
    cout << "Introduzca su fecha de nacimiento, dia: ";
    cin >> d;
    cout << "Introduzca el número de mes: ";
    cin >> m;
    cumpleanyos = new DiaAnyo(d, m);
    cout << " La fecha de hoy es ";
    hoy -> visualizar();
    cout << " Su fecha de nacimiento es ";
    cumpleanyos -> visualizar();
    if (hoy -> igual(*cumpleanyos))
        cout << "¡Feliz cumpleaños ! " << endl;
    else
        cout << "¡Feliz dia ! " << endl;
    return 0;
}
```

2.2.6. Sobrecarga de funciones miembro

Al igual que sucede con las funciones no miembro de una clase, las funciones miembro de una clase se pueden sobrecargar. Una función miembro se puede sobrecargar pero sólo en su propia clase.

Las mismas reglas utilizadas para sobrecargar funciones ordinarias se aplican a las funciones miembro: dos miembros sobrecargados no puede tener el mismo número y tipo de parámetros. La sobrecarga permite utilizar un mismo nombre para una función y ejecutar la función definida más adecuada a los parámetros pasados durante la ejecución del programa. La ventaja fundamental de trabajar con funciones miembro sobrecargadas es la comodidad que aporta a la programación.

Para ilustrar la sobrecarga, veamos la clase `Vector` donde aparecen diferentes funciones miembro con el mismo nombre y diferentes tipos de parámetros.

```
class Vector
{
public:
    int suma(int m[], int n);    //funcion 1
    int suma(const Vector& v);   //funcion 2
    float suma(float m, float n);      //funcion 3
    int suma();                  //funcion 4
};
```

Normas para la sobrecarga

No pueden existir dos funciones en el mismo ámbito con igual signatura (lista de parámetros).

2.3. CONSTRUCTORES Y DESTRUCTORES

Un *constructor* es una función miembro que se ejecuta automáticamente cuando se crea un objeto de una clase. Sirve para inicializar los miembros de la clase.

El constructor tiene el mismo nombre que la clase. Cuando se define no se puede especificar un valor de retorno, nunca devuelve un valor. Sin embargo, puede tomar cualquier número de argumentos.

Reglas

1. El constructor tiene el mismo nombre que la clase.
2. Puede tener cero, o más argumentos.
3. No tiene tipo de retorno.

EJEMPLO 2.4. La clase `Rectángulo` tiene un constructor con cuatro parámetros.

```
class Rectangulo
{
private:
    int izdo, superior;
    int dcha, inferior;
public:
    Rectangulo(int iz, int sr, int d, int inf) // constructor
    {
        izdo = iz; superior = sr;
        dcha = d; inferior = inf;
    }
    // definiciones de otros métodos miembro
};
```

Al crear un objeto se pasan los valores al constructor, con la misma sintaxis que la de llamada a una función. Por ejemplo:

```
Rectangulo Rect(4, 4, 10, 10);
Rectangulo* ptr = new Rectangulo(25, 25, 75, 75);
```

Se han creado dos instancias de `Rectangulo`, pasando valores concretos al constructor de la clase.

2.3.1. Constructor por defecto

Un constructor que no tiene parámetros, o que se puede llamar sin argumentos, se llama *constructor por defecto*. Un constructor por defecto normalmente inicializa los miembros dato de la clase con valores por defecto.

Regla

C++ crea automáticamente un constr uctor por def ecto cuando no e xisten otros constructores. Tal constructor inicializa el objeto a ceros binar ios.

EJEMPLO 2.5. El constructor de la clase `Complejo` tiene dos argumentos con v alores por defecto `0` y `1` respectivamenete, por tanto, puede llamarse sin argumentos .

```
class Complejo
{
private                          :
   double x;
   double y;
public:
   Complejo(double r = 0.0, double i = 1.0)
   {
      x = r;
      y = i;
   }
};
```

Cuando se crea un objeto `Complejo` puede inicializarse a los valores por defecto, o bien a otros valores.

```
Complejo z1;   // z1.x == 0.0, z1.y == 1.0
Complejo* pz = new Complejo(-2, 3);  // pz -> x == -2, pz -> y == 3
```

2.3.2. Constructores sobrecargados

Al igual que se puede sobrecargar un método de una clase, también se puede sobrecargar el constructor de una clase. De hecho los constructores sobrecargados son bastante frecuentes, proporcionan diferentes alternativas de inicializar objetos.

Regla

Para prevenir a los usuarios de la clase de crear un objeto sin parámetros, se puede: 1) omitir el constructor por defecto; o bien, 2) hacer el constructor privado.

EJEMPLO 2.6. La clase `EquipoSonido` se define con tres constructores; un constructor por defecto, otro con un argumento de tipo cadena y el tercero con tres argumentos.

```
class EquipoSonido
{
private:
    int potencia, voltios;
    string marca;
public:
    EquipoSonido()   // constructor por defecto
    {
        marca = "Sin marca"; potencia = voltios = 0;
    }
    EquipoSonido(string mt)
    {
        marca = mt; potencia = voltios = 0;
    }
    EquipoSonido(string mt, int p, int v)
    {
        marca = mt;
        potencia = p;
        voltios = v;
    }
};
```

La instanciación de un objeto `EquipoSonido` puede hacerse llamando a cualquier constructor. A continuación se crean tres objetos:

```
EquipoSonido rt;   // constructor por defecto
EquipoSonido ft("POLASIT");
EquipoSonido gt("PARTOLA", 35, 220);
```

2.3.3. Array de objetos

Los objetos se pueden estructurar como un array. Cuando se crea un array de objetos éstos se inicializan llamando al constructor sin argumentos. Por consiguiente, siempre que se prevea organizar los objetos en un array, la clase debe tener un constructor que pueda llamarse sin parámetros.

Precaución

Tenga cuidado con la escritura de una clase con sólo un constructor con argumentos. Si se omite un constructor que pueda llamarse sin argumento no será posible crear un array de objetos.

EJEMPLO 2.7. Se crean arrays de objetos de tipo `Complejo` y `EquipoSonido`.

```
Complejo zz[10];  // crea 10 objetos, cada uno se inicializa a 0,1
EquipoSonido* pq; // declaración de un puntero
int n;
cout << "Número de equipos: "; cin >> n;
pq = new EquipoSonido[n];
```

2.3.4. Constructor de copia

Este tipo de constructor se activa cuando al crear un objeto se inicializa con otro objeto de la misma clase. Por ejemplo:

```
Complejo z1(1, -3);      // z1 se inicializa con el constructor
Complejo z2 = z1;        /* z2 se inicializa con z1, actúa el
                            constructor de copia              */
```

También se llama al constructor de copia cuando se pasa un objeto por valor a una función, o bien cuando la función devuelve un objeto. Por ejemplo:

```
extern Complejo resultado(Complejo d);
```

para llamar a esta función se pasa un parámetro de tipo `Complejo`, un objeto. En esta transferencia se llama al constructor de copia. Lo mismo ocurre cuando la misma función devuelve (`return`) el resultado, un objeto de la clase `Complejo`.

El constructor de copia es una función miembro de la clase, su prototipo:

```
NombreClase(const NombreClase& origen);
```

el argumento `origen` es el objeto copiado, `z1` en el primer ejemplo. La definición del constructor de copia para la clase `Complejo`:

```
Complejo(const Complejo& origen)
{
    x = origen.x;
    y = origen.y;
}
```

No es siempre necesario definir el constructor de copia, por defecto se realiza una copia miembro a miembro. Sin embargo, cuando la clase tenga atributos (punteros) que representen memoria dinámica, *buffer dinámico*, sí debe definirse, para realizar una *copia segura*, reservando memoria y copiando en esa memoria o *buffer* los elementos.

2.3.5. Asignación de objetos

El operador de asignación, `=`, se puede utilizar con objetos, de igual forma que con datos de tipo simple. Por ejemplo:

```
Racional r1(1, 3);
Racional r2(2, 5);
r2 = r1;         // r2 toma los datos del objeto r1
```

Por defecto, la asignación se realiza miembro a miembro. El numerado de `r2` toma el valor del numerador de `r1` y el denominador de `r2` el valor del denominador de `r1`.

C++ permite cambiar la forma de asignar objetos de una clase. Para ello se implementa una función miembro de la clase especial (se denomina sobrecarga del operador `=`) con este prototipo:

```
nombreClase& operator = (const nombreClase&);
```

A continuación se implementa esta función en la clase `Persona`:

```
class Persona
{
private:
     int edad;
     string nom, apell;
     string dni;
public:
     // sobrecarga del operador de asignación
     Persona& operator = (const Persona& p)
     {
          if (this == &p) return *this; // es el mismo objeto
          edad = p.edad;
          nom = p.nom;
          apell = p.apell;
          dni = p.dni;
          return *this;
     }
```

En esta definición se especifica que no se tome acción si el objeto que se asigna es él mismo: `if (this == &p)`.

En la mayoría de las clases no es necesario definir el operador de asignación ya que, por defecto, se realiza una asignación miembro a miembro. Sin embargo, cuando la clase tenga miembros de tipo puntero (memoria dinámica, *buffer dinámico*) sí debe definirse, para que la asignación sea *segura*, reservando memoria y asignando a esa memoria o *buffer* los elementos.

EJEMPLO 2.8. La clase `Libro` se define con un array dinámico de páginas (clase `Pagina`). El constructor establece el nombre del autor , el número de páginas y crea el arr ay. Además, se escribe el constructor de copia y la sobrecarga del oper ador de asignación.

```
// archivo Libro.h

class Libro
{
private:
    int numPags, inx;
    string autor;
    Pagina* pag;
    // ...
```

```
public:
    Libro(string a, int n);     // constructor
    Libro(const Libro& cl);     // constructor de copia
    Libro& operator = (const Libro& al);  // operador de asignación
    // ... funciones miembro
};

// archivo Libro.cpp

Libro::Libro(string a, int n)
{
    autor = a;
    inx = 0;
    numPags = n;
    pag = new Pagina[numPags];
}
// constructor de copia
Libro::Libro(const Libro& cl)
{
    autor = cl.a;
    inx = cl.inx;
    numPags = cl.numPags;
    pag = new Pagina[numPags];  // copia segura
    for (int i = 0; i < inx; i++)
       pag[i] = cl.pag[i];
}
// operador de asignación
Libro& operator = (const Libro& al)
{
    if (this == &p) return *this;
    autor = al.a;
    inx = al.inx;
    numPags = al.numPags;
    pag = new Pagina[numPags];  // copia segura
    for (int i = 0; i < inx; i++)
        pag[i] = al.pag[i];
    return *this;
}
```

2.3.6. Destructor

Un objeto se libera, se destruye, cuando se sale del ámbito de definición. También, un objeto creado dinámicamente, con el operador `new`, se libera al aplicar el operador `delete` al puntero que lo referencia. Por ejemplo:

```
Punto* p = new Punto(1,2);
if (...)
{
    Punto p1(2, 1);
    Complejo z1(8, -9);
}   // los objetos p1 y z1 se destruyen
delete p;
```

El destructor tiene el mismo nombre que clase, precedido de una tilde (~). Cuando se define, no se puede especificar un valor de retorno, ni argumentos:

```
~NombreClase()
{
    ;
}
```

El destructor es necesario implementarlo cuando el objeto contenga memoria reserva dinámicamente.

EJEMPLO 2.9. Se declara la clase `Equipo` con dos atributos de tipo puntero, un constructor con valores por defecto y el destructor.

El constructor define una array de `n` objetos `Jugador` con el operador `new`. El destructor libera la memoria reservada.

```
class Equipo
{
private:
    Jugador* jg;
    int numJug;
    int actual;
public:
    Equipo(int n = 12)
    {
        jg = new Jugador[n];
        numJug = n;  actual = 0;
    }
    ~Equipo()    // destructor
    {
         if (jg != 0) delete [] jg;
    }
}
```

2.4. AUTOREFERENCIA DEL OBJETO: `THIS`

`this` es un puntero al objeto que envía un *mensaje*, o simplemente, un puntero al objeto que llama a una función miembro de la clase (ésta no debe ser `static`). Este puntero no se define, internamente se define:

```
const NombreClase* this;
```

por consiguiente, no puede modificarse. Las variables y funciones de las clase están referenciados, implícitamente, por `this`. Por ejemplo, la siguiente clase:

```
class Triangulo
{
private:
    double base, altura;
```

```
public:
    double area() const
    {
        return base*altura /2.0;
    }
};
```

En la función `area()` se hace referencia a las variables instancia `base` y `altura`. ¿A la `base`, `altura` de qué objeto? El método es común para todos los objetos `Triangulo`. Aparentemente no distingue entre un objeto y otro, sin embargo, cada variable instancia implícitamente está cualificada por `this`, es como si se hubiera escrito:

```
public double area()
{
    return this -> base * this -> altura/2.0;
}
```

Fundamentalmente `this` tiene dos usos:

- Seleccionar explícitamente un miembro de una clase con el fin de dar mas claridad o de evitar colisión de identificadores. Por ejemplo, en la clase `Triangulo`:

  ```
  void datosTriangulo(double base, double altura)
  {
      this -> base = base;
      this -> altura = altura;
  }
  ```

  Se ha evitado, con `this`, la colisión entre argumentos y variables instancia.
- Que una función miembro devuelva el mismo objeto que le llamó. De esa manera se pueden hacer llamadas en cascada a funciones de la misma clase. De nuevo en la clase `Triangulo`:

  ```
  const Triangulo& datosTriangulo(double base, double altura)
  {
      this -> base = base;
      this -> altura = altura;
      return *this;
  }

  const Triangulo& visualizar() const
  {
      cout << " Base = " << base << endl;
      cout << " Altura = " << altura << endl;
      return *this;
  }
  ```

  Ahora se pueden realizar esta concatenación de llamadas:

  ```
  Triangulo t;
  t.datosTriangulo(15.0, 12.0).visualizar();
  ```

2.5. CLASES COMPUESTAS

Una *clase compuesta* es aquella que contiene miembros dato que son asimismo objetos de clases. Antes de crear el cuerpo de un constructor de una clase compuesta, se deben construir los miembros dato individuales en su orden de declaración. La clase `Estudiante` contiene miembros dato de tipo `Expediente` y `Dirección`:

```
class Expediente
{
public:
    Expediente();      // constructor por defecto
    Expediente(int idt);

    // ...
};

class Direccion
{
public:
    Direccion();       // constructor por defecto
    Direccion(string d);
    // ...
};

class Estudiante
{
public:
    Estudiante()
    {
        PonerId(0);
        PonerNotaMedia(0.0);
    }
    void PonerId(long);
    void PonerNotaMedia(float);
private:
    long id;
    Expediente exp;
    Direccion dir;
    float NotMedia;
};
```

Aunque `Estudiante` contiene `Expediente` y `Dirección`, el constructor de `Estudiante` no tiene acceso a los miembros privados o protegidos de `Expediente` o `Dirección`. Cuando un objeto `Estudiante` sale fuera de alcance, se llama a su destructor. El cuerpo de `~Estudiante()` se ejecuta antes que los destructores de `Expediente` y `Dirección`. En otras palabras, el orden de las llamadas a destructores a clases compuestas es exactamente el opuesto al orden de llamadas de constructores.

La llamada al constructor con argumentos de los miembros de una clase compuesta se hace desde el constructor de la clase compuesta. Por ejemplo, este constructor de `Estudiante` inicializa su expediente y dirección:

```
Estudiante::Estudiante(int expediente, string direccion)
        :exp(expediente), dir(direccion)  // lista de inicialización
```

```
{
    PonerId(0);
    PonerNotaMedia(0.0);
}
```

Regla

El orden de creación de un objeto compuesto es en, primer lugar, los objetos miembros en orden de aparición; a continuación el cuerpo del constructor de la clase compuesta.

La llamada al constructor de los objetos miembros se realiza en la lista de inicialización del constructor de la clase compuesta, con la sintaxis siguiente:

```
Compuesta(arg1, arg2, arg3,...): miembro1(arg1,...), miembro2(arg2,...)
{
    // cuerpo del constructor de clase compuesta
}
```

2.6. MIEMBROS STATIC DE UNA CLASE

Cada instancia de una clase, cada objeto, tiene su propia copia de las variables de la clase. Cuando interese que haya miembros que no estén ligados a los objetos sino a la clase y, por tanto, comunes a todos los objetos, éstos se declaran `static`.

2.6.1. Variables `static`

Las variables de clase `static` son compartidas por todos los objetos de la clase. Se declaran de igual manera que otra variable, añadiendo, como prefijo, la palabra reservada `static`. Por ejemplo:

```
class Conjunto
{
    static int k;
    static Lista lista;
    // ...
}
```

Los miembros `static` de una clase deben ser inicializados explícitamente fuera del cuerpo de la clase. Así, los miembros `k` y `lista`:

```
int Conjunto::k = 0;
Lista Conjunto::lista = NULL;
```

Dentro de las clases se accede a los miembros `static` de la manera habitual, simplemente con su nombre. Desde fuera de la clase se accede con el nombre de la clase, el selector. y el nombre de la variable, por ejemplo:

```
cout << " valor de k = " << Conjunto.k;
```

Formato

El símbolo :: (operador de resolución de ámbitos) se utiliza en sentencias de ejecución que accede a los miembros estáticos de la clase. Por ejemplo, la expresión `Punto::X` se refiere al miembro dato estático `X` de la clase `Punto`.

Ejercicio 2.2

Dada una clase se quiere conocer en todo momento los objetos activos en la aplicación.

Se declara la clase `Ejemplo` con dos constructores y el constructor de copia. Todos incrementan la variable `static cuenta`, en `1`. De esa manera cada nuevo objeto queda contabilizado. También se declara el destructor para decrementar `cuenta` en `1`. `main()` crea objetos y visualiza la variable que contabiliza el número de sus objetos.

```
// archivo Ejemplo.h
class Ejemplo
{
private:
    int datos;
public:
    static int cuenta;
    Ejemplo();
    Ejemplo(int g);
    Ejemplo(const Ejemplo&);
    ~Ejemplo();
};

// definición de la clase, archivo Ejemplo.cpp

#include "Ejemplo.h"

int Ejemplo::cuenta = 0;

Ejemplo::Ejemplo()
{
    datos = 0;
    cuenta++;          // nuevo objeto
}

Ejemplo::Ejemplo(int g)
{
   datos = g;
   cuenta++;           // nuevo objeto
}

Ejemplo::Ejemplo(const Ejemplo& org)
{
    datos = org.datos;
    cuenta++;          // nuevo objeto
}
```

```
Ejemplo::~Ejemplo()
{
    cuenta--;
}

// programa de prueba, archivo Demostatic.cpp

#include <iostream>
using namespace std;
#include "Ejemplo.h"

int main()
{
    Ejemplo d1, d2;

    cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
    if (true)
    {
        Ejemplo d3(88);
        cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
    }
    cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
    Ejemplo* pe;
    pe = new Ejemplo();
    cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
    delete pe;
    cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
    return 0;
}
```

2.6.2. Funciones miembro `static`

Los métodos o funciones miembro de las clases se llaman a través de los objetos. En ocasiones, interesa definir funciones que estén controlados por la clase, incluso que no haga falta crear un objeto para llamarlos, son las funciones miembro `static`. La llamada a estas funciones de clase se realiza a través de la clase: `NombreClase::metodo()`, respetando las reglas de visibilidad. También se pueden llamar con un objeto de la clase, no es recomendable debido a que son métodos dependientes de la clase y no de los objetos.

Los métodos definidos como `static` no tienen asignado la referencia `this`, por eso sólo pueden acceder a miembros `static` de la clase. Es un error que una función miembro `static` acceda a miembros de la clase no `static`.

EJEMPLO 2.10. La clase `SumaSerie` define tres variables `static`, y un método `static` que calcula la suma cada vez que se llama.

```
class SumaSerie
{
private:
    static long n;
    static long m;
public:
    static long suma()
```

```
    {
        m += n;
        n = m - n;
        return m;
    }
};
long SumaSerie::n = 0;
long SumaSerie::m = 1;
```

2.7. FUNCIONES AMIGAS (FRIEND)

Con el mecanismo amigo (`friend`*)* se permite que funciones no miembros de una clase puedan acceder a sus miembros privados o protegidos. Se puede hacer `friend` de una clase una función global, o bien otra clase. En el siguiente ejemplo la función global `distancia()` se declara `friend` de la clase `Punto`.

```
double distancia(const Punto& P2)
{
    double d;
    d = sqrt((double)(P2.x * P2.x + P2.y * P2.y));
}
class Punto
{
    friend double distancia(const Punto& P2);
    // ...
};
```

Si `distancia()` no fuera *amiga* de `Punto` no podrá acceder directamente a los miembros privado `x` e `y`.

Es muy habitual sobrecargar el operador `<<` para mostrar por pantalla los objetos de una clase con `cout`. Esto quiere decir que al igual que se escribe un número entero, por ejemplo:

```
int k = 9;
cout << " valor de k = " << k;
```

se pueda escribir un objeto `Punto`:

```
Punto p(1, 5);
cout << " Punto " << p;
```

Para conseguir esto hay que definir la función global `operator<<` y hacerla *amiga* de la clase, en el ejemplo de la clase `Punto`:

```
ostream& operator << (ostream& pantalla, const Punto& mp)
{
    pantalla << " x = " << mp.x << ", y = " << mp.y << endl;
    return pantalla;
}
class Punto
{
```

```
    friend
    ostream& operator << (ostream& pantalla, const Punto& mp);
    // ...
};
```

Una clase completa se puede hacer amiga de otra clase. De esta forma todas las funciones miembro de la clase amiga pueden acceder a los miembros protegidos de la otra clase. Por ejemplo, la clase `MandoDistancia` se hace *amiga* de la clase `Television`:

```
class MandoDistancia { ... };
class Television
{
   friend class MandoDistancia;
   // ...
}
```

Regla

La declaración de amistad empieza por la palabra reservada `friend`, sólo puede aparecer dentro de la declaración de una clase. Se puede situar en cualquier parte la clase, es práctica recomendada agrupar todas las declaraciones `friend` inmediatamente a continuación de la cabecera de la clase.

2.8. TIPOS ABSTRACTOS DE DATOS EN C++

La estructura más adecuada, en C++, para implementación un **TAD** es la clase. Dentro de la clase va a residir la representación de los datos junto a las operaciones (funciones miembro de la clase). La interfaz del tipo abstracto queda perfectamente determinado con la etiqueta `public`, que se aplicará a las funciones de la clase que representen operaciones.

Por ejemplo, si se ha especificado el **TAD** `PuntoTres` para representar la abstracción punto en el espacio tridimensional, la clase que implenta el tipo:

```
class PuntoTres
{
private:
   double x, y, z;              // representación de los datos
public:                         // operaciones
   double distancia(PuntoTres p);
   double modulo();
   double anguloZeta();
    ...
};
```

2.8.1. Implementación del TAD Conjunto

En el Apartado 1.6.2 se ha especificado el **TAD** `Conjunto`, la implementación se realiza con la clase `Conjunto`. Se supone que los elementos del conjunto son cadenas (`string`), aunque se podría generalizar creando una clase plantilla (`template`), pero se prefiere simplificar el desarrollo y más adelante utilizar la genericidad.

La declaración de la clase está en el archivo `Conjunto.h`, la implementación de las funciones, la definición, en `Conjunto.cpp`. Los elementos del conjunto se guardan en un array que crece dinámicamente.

Archivo `Conjunto.h`

```
const int M = 20;
class Conjunto
{
private;             // representación
   string* cto;
   int cardinal;
   int capacidad;
public:              // operaciones
   Conjunto(); const;
   Conjunto (const Conjunto& org);
   bool esVacio() const;
   void annadir(string elemento);
   void retirar(string elemento);
   boolean pertenece(string elemento) const;
   int cardinal() const;
   Conjunto union(const Conjunto& c2);
};
```

Archivo `Conjunto.cpp`

```
#include <iostream>
using namespace std;
#include "Conjunto.h"

Conjunto::Conjunto()
{
    cto = new string[M];
    cardinal = 0;
    capacidad = M;
}

Conjunto::Conjunto(const Conjunto& org)
{
    cardinal = org.cardinal;
    capacidad = org.capacidad;
    cto = new string[capacidad];  // copia segura
    for (int i = 0; i < cardinal; i++)
       cto[i] = opg.cto[i];
}

bool Conjunto::esVacio() const
{
    return (cardinal == 0);
}

void Conjunto::annadir(string elemento)
{
    if (!pertenece(elemento))
```

```
    {
        // amplia el conjunto si no hay posiciones libres
       if (cardinal == capacidad)
       {
           string* nuevoCto;
           nuevoCto = new string[capacidad + M];
           for (int k = 0; k < cardinal; k++)
              nuevoCto[k] = cto[k];
           delete cto;
           cto = nuevoCto;
           capacidad += M;
      }
      cto[cardinal++] = elemento;
    }
}

void Conjunto::retirar(string elemento) const
{
    if (pertenece(elemento))
    {
        int k = 0;
        while (!(cto[k] == elemento)) k++;

        // mueve a la izqda desde elemento k+1 hasta última posición
        for (; k < cardinal ; k++)
            cto[k] = cto[k+1];
        cardinal--;
    }
}

bool Conjunto::pertenece(string elemento) const
{
    int k = 0;
    bool encontrado = false;

    while (k < cardinal && !encontrado)
    {
        encontrado = cto[k] == elemento;
        k++;
    }
    return encontrado;
}

int Conjunto::cardinal() const
{
    return this -> cardinal;
}

Conjunto Conjunto::union(const Conjunto& c2)
{
    int k;

    Conjunto u;   // conjunto unión
       // primero copia el primer operando de la unión
    for (k = 0; k < cardinal; k++)
       u.cto[k] = cto[k];
```

```
      u.cardinal = cardinal;
         // añade los elementos de c2 no incluidos
      for (k = 0; k < c2.cardinal; k++)
         u.annadir(c2.cto[k]);
      return u;
   }
```

RESUMEN

En la mayoría de los lenguajes de programación orientados a objetos, y en particular en C++, los tipos abstractos de datos se implementan mediante **clases**.

Una **clase** es un tipo de dato definido por el programador que sirve para representar objetos del mundo real. Un objeto de una clase tiene dos componentes: un conjunto de atributos o variables instancia y un conjunto de comportamientos (métodos, funciones miembro). Los atributos también se llaman variables instancia o miembros dato y los comportamientos se llaman métodos miembro.

```
class Circulo
{
private:
    double centroX, centroY;
    double radio;
public:
   double superficie();
};
```

Un objeto es una instancia de una clase y una variable cuyo tipo sea la clase es un objeto de la clase.

```
Circulo unCirculo;          // objeto Circulo
Circulo tipocirculo [10];   // array de 10 objetos
```

La declaración de una clase, en cuanto a visibilidad de sus miembros, tiene tres secciones: *pública*, *privada* y *protegida*. La sección pública contiene declaraciones de los atributos y el comportamiento del objeto que son accesibles a los usuarios del objeto. Los constructores inicializan los objetos y se recomenda su declaración en la sección pública. La sección privada contiene los métodos miembro y los miembros dato que son ocultos o inaccesibles a los usuarios del objeto. Estos métodos miembro y atributos dato son accesibles sólo dentro del objeto. Los miembros de una clase con visibilidad `protected` son accesibles desde el interior de la clase y para las clases derivadas.

Un **constructor** es una función miembro con el mismo nombre que su clase. Un constructor no puede devolver un tipo pero puede ser sobrecargado.

```
class Racional
{
public:
    Racional (int nm, int dnm);
    Racional(const Racional& org);
}
```

El **constructor** es un método especial que se invoca cuando se crea un objeto. Se utiliza, normalmente, para inicializar los atributos de un objeto. Por lo general, al menos se define un construc-

tor sin argumentos, llamado constructor por defecto. En caso de no definirse constructor, implícitamente queda definido un constructor sin argumentos que inicializa el objeto a ceros binarios.

El proceso de crear un objeto se llama *instanciación* (creación de instancia). En C++ se crea un objeto como se define una variable, o bien dinámicamente con el operador `new` y un constructor de la clase.

```
Racional R(1, 2);
Racional* pr = new Racional(4, 5);
```

En C++ la liberación de objetos ocurre cuando termina el bloque dentro del cual se ha definido, o cuando se aplica el operador `delete` a un puntero al objeto. La liberación la realiza una función miembro especial, denominada destructor y de prototipo: `~NombreClase()`. Realmente sólo es necesario definir el destructor cuando el objeto contenga memoria reservada dinámicamente con el operador `new`.

Los miembros de un clase definidos como `static` no están ligados a los objetos de la clase sino que son comunes a todos los objetos, que son de la clase. Se cualifican con el nombre de la clase, por ejemplo:

```
Matematica::fibonacci(5);
```

EJERCICIOS

2.1. ¿Qué está mal en la siguiente definición de la clase?

```
class Buffer
{
private:
    char* datos;
    int cursor ;
    Buffer(int n)
    {
        datos = new char[n];
    };
public: static int Long( return cursor;)
}
```

2.2. Se desea realizar la clase `Vector3d` que permita manipular vectores de tres componentes (coordenadas `x`, `y`, `z`) de acuerdo a las siguientes normas:

- Sólo posee una método constructor.
- Tiene una función miembro `igual()` que permite saber si dos vectores tienen sus componentes o coordenadas iguales.

2.3. Incluir en la clase `Vector3d` del Ejercicio 2.2 la función `normaMax` que permita obtener la norma de un vector (**Nota:** La norma de un vector $v = (x, y, z)$ es $\sqrt{(x^2 + y^2 + z^2)}$.

2.4. Realizar la clase `Complejo` que permita la gestión de números complejos (un número complejo consta de dos números reales de tipo `double` : una parte real + una parte imaginaria). Las operaciones a implementar son las siguientes:

- `establecer()` permite inicializar un objeto de tipo `Complejo` a partir de dos componentes `double`.
- `imprimir()` realiza la visualización formateada de un `Complejo`.
- `agregar()` (sobrecargado) para añadir, respectivamente, un `Complejo` a otro y añadir dos componentes `double` a un `Complejo`.

2.5. Añadir a la clase `Complejo` del Ejercicio 2.4 las siguientes operaciones :

- Suma: `a + c = (A + C, (B + D)i)`.
- Resta: `a – c = (A – C, (B – D)i)`.
- Multiplicación: `a * c=(A*C-B*D, (A*D+B*C)i)`.
- Multiplicación: `x * c = (x*C, x*Di)`, donde *x* es `real`.
- Conjugado: `~a = (A ,-Bi)`.
  Siendo `a = A+Bi; c = C+Di`.

2.6. Implementar la clase `Hora`. Cada objeto de esta clase representa una hora específica del día, almacenando las horas, minutos y segundos como enteros. Se ha de incluir un constructor, métodos de acceso, una método `adelantar(int h, int m, int s)` para adelantar la hora actual de un objeto existente, una método `reiniciar(int h, int m, int s)` que reinicializa la hora actual de un objeto existente y una método `imprimir()`.

PROBLEMAS

2.1. Implementar la clase `Fecha` con miembros dato para el mes, día y año. Cada objeto de esta clase representa una fecha, que almacena el día, mes y año como enteros. Se debe incluir un constructor por defecto, funciones de acceso, la función `reiniciar (int d, int m, int a)` para reiniciar la fecha de un objeto existente, la función `adelantar(int d, int m, int a)`para avanzar una fecha existente (día `d`, mes `m`, y año `a`) y la función `imprimir()`. Escribir una función auxiliar de utilidad, `normalizar()`, que asegure que los miembros dato están en el rango correcto $1 \leq$ `año`, $1 \leq$ `mes` ≤ 12, `día` $\leq$ `días (Mes)`, siendo `días(Mes)` otra función que devuelve el número de días de cada mes.

2.2. Ampliar el programa 2.1 de modo que pueda aceptar años bisiestos. **Nota:** un año es bisiesto si es divisible por 400, o si es divisible por 4 pero no por 100. Por ejemplo el año 1992 y 2000 son años bisiestos y 1997 y 1900 no son bisiestos.

CAPÍTULO 3

Tipos de datos básicos:

Arrays, cadenas, estructuras y tipos enumerados

Objetivos

Con el estudio de este capítulo usted podrá:

- Revisar los conceptos básicos de tipos de datos.
- Diferenciar entre un tipo simple y un tipo estructurado.
- Organizar colecciones de items en una misma estructura de programación.
- Definir no sólo arrays de una dimensión sino de dos o mas dimensiones.
- Declarar variables `string` e inicializarla a una cadena constante.
- Conocer las distintas operaciones de la clase `string`.

Contenido

Conceptos clave

- Array (arreglo).
- Cadena.
- Estructura de datos.
- Indexación.
- Lista y tabla.
- Matriz
- Tipo de dato.
- Tipo enumerado

INTRODUCCIÓN

En este capítulo se examinan los tipos básicos: *array* o *arreglo*, cadenas (*string*), estructuras y tipos enumerados. Aprenderá el concepto y tratamiento de los arrays. Un array (arreglo) almacena elementos del mismo tipo, tales como veinte enteros, cincuenta números de coma flotante o quince caracteres. Los arrays pueden ser de una dimensión (vectores), los mas utilizados; de dos dimensiones (tablas o matrices); de tres o más dimensiones. En el capítulo se examina el manejo de cadenas como objetos de la clase `string` (como contenedor de la biblioteca STL) que representa a una secuencia de caracteres y las operaciones con cadenas mas comunes.

3.1. TIPOS DE DATOS

Los lenguajes de programación tradicionales, tales como Pascal y C, proporcionan *tipos de datos* para clasificar diversas clases de datos. La ventajas de utilizar tipos en el desarrollo de *software* [TREMBLAY 03] son:

- Apoyo y ayuda en la prevención y detección de errores.
- Apoyo y ayuda a los desarrolladores de software y la comprensión y organización de ideas acerca de sus objetos.
- Ayuda en la identificación y descripción de propiedades únicas de ciertos tipos.

Los tipos son un enlace importante entre el mundo exterior y los elementos datos que manipulan los programas. El uso de estos tipos permite a los desarrolladores limitar su atención a tipos específicos de datos. Los tipos específicos tienen propiedades únicas. Por ejemplo, el tamaño es una propiedad determinante en los arrays y en los cadenas; sin embargo no es una propiedad esencial en los valores lógicos, *verdadero* (`true`) y *falso* (`false`).

> **Definición** Un *tipo de dato* es un conjunto de valores y operaciones asociadas a esos valores.

En los lenguajes de programación hay disponibles un gran número de tipos de datos. Entre ellos se pueden destacar: *tipos primitivos de datos*, *tipos compuestos* y *tipos agregados.*

3.1.1. Tipos primitivos de datos

Los tipos de datos más simples son los *tipos de datos primitivos,* también denominados *datos atómicos,* porque no se construyen a partir de otros tipos y son entidades únicas no descomponibles en otros.

Un **tipo de dato atómico** es un conjunto de datos atómicos con propiedades idénticas. Estas propiedades diferencian un tipo de dato atómico de otro. Los tipos de datos atómicos se definen por un conjunto de valores y un conjunto de operaciones que actúan sobre esos valores.

Tipo de dato atómico

1. Un conjunto de valores.
2. Un conjunto de operaciones sobre esos valores.

EJEMPLO 3.1. Diferentes tipos de datos atómicos.

Enteros

valores	`–∞ , …, –3, –2, –1, 0, 1, 2, 3, …, +∞`
operaciones	`*, +, –, /, %,++,––, …`

Coma flotante

valores	`- , …-.6, 0.0, .5 …2.5,`
operaciones	`*, +, -, %, /, …`

Carácter

valores	`\0, …, 'A', 'B', …, 'a', 'b',….`
operaciones	`<, >, …`

Lógico

valores	`verdadero, falso`
operaciones	`and, or, not,…`

Los tipos numéricos son, probablemente, los tipos primitivos más fáciles de entender, debido a que las personas están familiarizadas con los números. Sin embargo, los números pueden también ser difíciles de comprender por los diferentes métodos en que son representados en las computadoras. Por ejemplo, los números decimales y binarios tienen representaciones diferentes en las máquinas. Debido a que los números de bits que representan a los números es finito, sólo los subconjuntos de enteros y reales se pueden representar. A consecuencia de ello se pueden presentar situaciones de desbordamiento (*underflow* y *overflow*) positivo y negativo al realizar operaciones aritméticas. Normalmente, los rangos numéricos y la precisión varía de una máquina a otra máquina. Para eliminar estas inconsistencias, algunos lenguajes modernos como Java y C# tienen especificaciones precisas para el número de bits, el rango y la precisión numérica de cada operación para cada tipo numéricos.

```
0         23        786       456       999
7.56      4.34      0.897     1.23456   99.999
```

El tipo de dato `boolean` (lógico) suele considerarse como el tipo más simple debido a que sólo tiene dos valores posibles: *verdadero* (`true`) y *falso* (`false`). El formato sintáctico de estas constantes lógicas puede variar de un lenguaje de programación a otra. Algunos lenguajes ofrecen un conjunto rico de operaciones *lógicas* que otros. Por ejemplo, algunos lenguajes tienen construcciones que permiten a los programadores especificar operaciones condicionales o en cortocircuito. Ejemplo, en C++ en los operadores lógicos `&&` y `||`, sólo se evalúa el segundo operando si su valor se necesita para determinar el valor de la expresión global.

El tipo **carácter (char)** consta del conjunto de caracteres disponibles para un lenguaje específico en una computadora específica. En algunos lenguajes de programación, un carácter es un símbolo indivisible, mientras que una cadena es una secuencia de cero o más caracteres. Las cadenas se pueden manipular de muchas formas, pero los caracteres implican pocas manipulaciones. Los tipos carácter, normalmente, son dependientes de la máquina.

```
'Q'    'a'    '8'    '9'    'k'
```

Los códigos de carácter más utilizados son **ASCII** (el código universal más extendido) y **EBCDIC** (utilizado por las primeras máquinas de **IBM**). La aparición del lenguaje Java, trajo consigo la representación de caracteres en **Unicode**, un conjunto internacional de caracteres que unifica a muchos conjuntos de caracteres, incluyendo inglés, español, italiano, griego, alemán, latín, hebreo o indio.

3.1.2. Tipos de datos compuestos y agregados

Los datos compuestos son el tipo opuesto a los tipos de datos atómicos. Los datos compuestos se pueden romper en subcampos que tengan significado. Un ejemplo sencillo es el número de su teléfono `51199110101`. Realmente, este número consta de varios campos, el código del país (`51`, Perú), el código del área (`1`, Lima) y el número propiamente dicho, que corresponde a un celular porque empieza con `9`.

En algunas ocasiones se conocen también a los datos compuestos como datos o tipos agregados. Los **tipos agregados** son tipos de datos cuyos valores son colecciones de elementos de datos. Un tipo agregado se compone de tipos de datos previamente definitivos. Existen tres tipos agregados básicos: arrays, secuencias y registros.

Un ***array*** o **arreglo**[1] es, normalmente, una colección de datos de tamaño o longitud fija, cada uno de cuyos datos es accesible en tiempo de ejecución mediante la evaluación de las expresiones que representan a los subíndices o índices correspondientes. Todos los elementos de un array deben ser del mismo tipo.

```
array de enteros: [4, 6, 8, 35, 46, 810]
```

Una **secuencia** o **cadena** (**`string`**) es, en esencia, un array cuyo tamaño puede variar en tiempo de ejecución. Por consiguiente, las secuencias son similares a arrays dinámicos o flexibles.

```
Cadena = "Aceite picual de Carchelejo"
```

Un **registro** puede contener elementos datos agregados y primitivos. Cada elemento agregado, eventualmente, se descomponen en *campos* formados por elementos primitivos. Un registro se puede considerar como un tipo o colección de datos de tamaño fijo. Al contrario que en los arrays, en los que todos sus elementos deben ser del mismo tipo de datos, los campos de los registros pueden ser de tipos diferentes de datos. A los campos de los registros se accede mediante identificadores.

---

[1] El término inglés *array* se traduce en casi toda Latinoamérica por **arreglo**, mientras que en España se ha optado por utilizar el término en inglés o bien su traducción por “lista” , “tabla” o “matriz”.

El registro es el tipo de dato más próximo a la idea de objeto. En realidad el concepto de objeto en un desarrollo orientado a objeto es una generalización del tipo registro.

```
Registro {
    Dato1
    Dato2
    Dato3
    ...
}
```

En C++, el tipo registro se denomina estructura. Una ***estructura*** (`struct`) es un tipo definido por el usuario compuesto de otros tipos de variables. La sintaxis básica es:

```
struct nombreEstructura
{
    tipo nombreVar;
    tipo nombreVar;
    // ...
};
```

3.2. LA NECESIDAD DE LAS ESTRUCTURAS DE DATOS

A pesar de la gran potencia de las computadoras actuales, la eficiencia de los programas sigue siendo una de las características más importantes a considerar. Los problemas complejos que cada vez más procesan las computadoras obliga sobre todo a pensar en su eficiencia dado el elevado tamaño que suelen alcanzar. Hoy, mas que nunca, los profesionales deben formarse en técnicas de construcción de programas eficientes.

En sentido general, una estructura de datos es cualquier representación de datos y sus operaciones asociadas. Bajo esta óptica, cualquier representación de datos incluso un número entero o un número de coma flotante almacenado en la computadora es una sencilla estructura de datos. En sentido más específico, una estructura de datos es una organización o estructuración de una colección de elementos dato. Así una lista ordenada de enteros almacenados en un array es un ejemplo de tal estructuración.

> Una **estructura de datos** es una agregación de tipos de datos compuestos y atómicos en un conjunto con relaciones bien definidas . Una estructura significa un conjunto de reglas que contienen los datos juntos.

Las estructuras de datos pueden estar *anidadas*. Se puede tener una estructura de datos que conste de otras estructuras de datos.

> ***Estructura de datos***
>
> 1. Una combinación de elementos cada uno de los cuales es o bien un tipo de dato u otra estructura de datos.
> 2. Un conjuntos de asociaciones o relaciones (estructura) que implica a los elementos combinados.

La Tabla 3.1 recoge las definiciones de dos estructuras de datos clásicas: *arrays* y registros.

Tabla 3.1. Ejemplos de estructura de datos

Array	Registro
Secuencias homogéneas de datos o tipos de datos conocidos como elementos.	Combinación de datos heterogéneos en una estructura única con una clave identificativa.
Asociación de posición entre los elementos.	Ninguna asociación entre los elementos.

La mayoría de los lenguajes de programación soportan diferentes estructuras de datos. Además esos mismos lenguajes suelen permitir a los programadores crear sus propias nuevas estructuras de datos con el objetivo fundamental de resolver del modo más eficiente posible una aplicación.

Será necesario que la elección de una estructura de datos adecuada, requiera también la posibilidad de poder realizar operaciones sobre dichas estructuras. La elección de la estructura de datos adecuadas, redundará en una mayor eficiencia del programa y sobre todo en una mejor resolución del problema en cuestión. Una elección inadecuada de la estructura de datos puede conducir a programas lentos, largos y poco eficientes.

Una solución se dice **eficiente** si resuelve el problema dentro de las *restricciones de recursos requeridas*. Restricciones de recursos pueden ser: espacio total disponible para almacenar los datos (considerando memoria principal independiente de las restricciones de espacio de discos, fijos, CD, DVD, flash...); tiempo permitido para ejecutar cada subtarea. Se suele decir que una solución es eficiente cuando requiere menos recursos que las alternativas conocidas.

3.2.1. Etapas en la selección de una estructura de datos

Los pasos a seguir para seleccionar una estructura de datos que resuelva un problema son [SHAFFER 97]:

1. Analizar el problema para determinar las restricciones de recursos que debe cumplir cada posible solución.
2. Determinar las operaciones básicas que se deben soportar y cuantificar las restricciones de recursos para cada operación. Ejemplos de operaciones básicas incluyen inserción de un datos en la estructura de datos, suprimir un dato de la estructura de datos o encontrar un dato determinado en dicha estructura.
3. Seleccionar la estructura de datos que cumple mejor los requisitos o requerimientos.

Este método de tres etapas para la selección de una estructura de datos es una vista centrada en los datos. Primero, se diseñan los datos y las operaciones que se realizan sobre ellos, a continuación viene la representación de esos datos y por último viene la implementación de esa representación.

Algunas consideraciones importantes para la elección de la estructura de datos adecuada, son:

- ¿Todos los datos se insertan en la estructura de datos al principio o se entremezclan con otras operaciones?

- ¿Se pueden eliminar los datos?
- ¿Los datos se procesan en un orden bien definido o se permite el acceso aleatorio?

Resumen

Un tipo es una colección de valores. Ejemplo: el tipo `bool` consta de los valores `true` y `false`. Los enteros también forman un tipo.

Un tipo de dato es un tipo junto con una colección de operaciones que manipulan el tipo. Ejemplo, una variable entera es un miembro del tipo de dato entero. La suma es un ejemplo de una operación sobre tipos de datos enteros.

Un elemento dato es una pieza de información o un registro cuyo valores se especifica a partir de un tipo. Un elemento dato se dice que es un miembro de un tipo de dato. El entero es un elemento de datos simple y a que no contiene subpartes.

Un registro de una cuenta corriente de un banco puede contener varios campos o piezas de información tales como nombre, número de la cuenta, saldo y dirección. Tal registro es un dato agregado.

3.3. ARRAYS (ARREGLOS)

Un *array* (lista o tabla) o **arreglos** es una secuencia de objetos del mismo tipo. Los objetos se llaman elementos del array y se numeran consecutivamente `0, 1, 2, 3 ...` El tipo de elementos almacenados en el array puede ser cualquier tipo de dato de C++, incluyendo clases definidas por el usuario.

Un array puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o los alumnos de un curso. Cada item del array se denomina *elemento*.

Los elementos de un array se numeran, como ya se ha comentado, consecutivamente `0, 1, 2, 3,...` Estos números se denominan *valores índice* o *subíndice* del array. El término "subíndice" se utiliza ya que se especifica igual que en matemáticas como una secuencia tal como $a_0, a_1, a_2 \ldots$ Estos números localizan la posición del elemento dentro del array, proporcionando *acceso directo* al array.

Si el nombre del array es `a`, entonces `a[0]` es el nombre del elemento que está en la posición `0`, `a[1]` es el nombre del elemento que está en la posición `1`, etc. En general, el elemento *i-ésimo* está en la posición `i-1`. Este método de numeración se denomina *indexación basada en cero*. Su uso tiene el efecto de que el índice de un elemento del array es el número de "pasos" desde el elemento inicial `a[0]` a ese elemento. Por ejemplo, `a[3]` está a 3 pasos o posiciones del elemento `a[0]`. De modo que si el array tiene *n* elementos, sus nombres son `a[0], a[1], ... , a[n-1].`

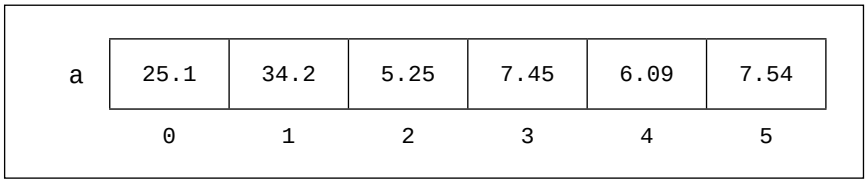


Figura 3.1. Array a de seis elementos.

El diagrama de la Figura 3.1 representa realmente una región de la memoria de la computadora ya que un array se almacena siempre con sus elementos en una secuencia de posiciones de memoria contigua.

3.3.1. Declaración de un array

Al igual que con cualquier tipo de variable, se debe declarar un array antes de utilizarlo. Un array se declara de modo similar a otros tipos de datos, excepto que se debe indicar al compilador el *tamaño* o *longitud* del array. Para indicar al compilador el *tamaño* o *longitud* del array se debe hacer seguir al nombre, el tamaño encerrado entre corchetes. La *sintaxis* para declarar un array de una dimensión determinada es:

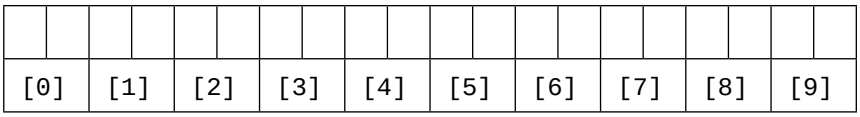
```
tipo nombreArray[numeroDeElementos];
```

Por ejemplo, para crear un array (lista) de diez variables enteras, se escribe:

```
int numeros[10];
```

Esta declaración hace que el compilador reserve espacio suficiente para contener diez valores enteros. La Figura 3.2 muestra el esquema de un array de diez elementos; cada elemento puede tener su propio valor.

Array (**arreglo**) *de datos enteros*: **a**



Un array de enteros se almacena en bytes consecutivos de memoria. Cada elemento utiliza tantos bytes como el tamaño de un entero (se supone dos). Se accede a cada elemento de array mediante un índice que comienza en cero.

Figura 3.2. Almacenamiento de un array en memoria.

Se puede acceder a cada elemento del array utilizando un índice en el array. Por ejemplo,

```
cout << numeros[4] << endl;
```

visualiza el valor del elemento 5 del array. Los arrays en C++ siempre comienzan en el elemento `0`.

Precaución

C++ no comprueba que los índices del array están dentro del rango definido. Así, por ejemplo, se puede intentar acceder a `numeros[12]` y el compilador no producirá ningún error, lo que puede producir un fallo en su programa, dependiendo del contexto en que se encuentre el error.

> ***Nota***
>
> Todos los subíndices de los arrays comienzan con 0.

3.3.2. Inicialización de un array

Se deben asignar valores a los elementos del array antes de utilizarlos, tal como se asignan valores a variables. Para asignar valores a cada elemento del array de enteros `precios`, se puede escribir:

```
precios[0] = 10;
precios[1] = 20;
precios[2] = 30;
...
```

La primera sentencia fija `precios[0]` al valor `10`, `precios[1]` al valor `20`, etc. Sin embargo, este método no es práctico cuando el array contiene muchos elementos. El método utilizado normalmente es inicializar el array completo en una sola sentencia.

Cuando se inicializa un array, el tamaño del array se puede determinar automáticamente por las constantes de inicialización. Estas constantes se separan por comas y se encierran entre llaves, como en los siguientes ejemplos:

```
int numeros[6] = {10, 20, 30, 40, 50, 60};
int n[] = {3, 4, 5}               //Declara un array de 3 elementos
char c[] = {'L','u','i','s'};     //Declara un array de 4 elementos
```

Los arrays de caracteres se pueden inicializar con una constante de cadena, como en

```
char s[] = "Mona Lisa";
```

> ***Nota***
>
> C++ puede dejar los corchetes vacíos, sólo cuando se asignan valores iniciales al array, tal como
>
> ```
> int cuenta[] = {15, 25, -45, 0, 50};
> ```
>
> El compilador asigna automáticamente cinco elementos a `cuenta`.

El método de inicializar arrays mediante valores constantes después de su definición es adecuado cuando el número de elementos del array es pequeño. Por ejemplo, para inicializar un array (lista) de `10` enteros a los valores `10` a `1`:

```
int cuenta[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
```

Se pueden asignar constantes simbólicas como valores numéricos, de modo que las sentencias siguientes son válidas:

```
const int ENE = 31, FEB = 28, MAR = 31, ABR = 30, MAY = 31,
          JUN = 30, JUL = 31, AGO = 31, SEP = 30, OCT = 31,
          NOV = 30, DIC = 31;
...
int meses[12] = {ENE, FEB, MAR, ABR, MAY, JUN,
                 JUL, AGO, SEP, OCT, NOV, DIC};
```

Se pueden asignar valores a un array utilizando un bucle `for` o bien `while/do-while`, éste suele ser el sistema más empleado normalmente. Por ejemplo, para visualizar en orden descendente el array `cuenta`:

```
for (int i = 9; i >= 0; i--)
    cout << "\n cuenta descendente" << i << "=" << cuenta[i];
```

EJEMPLO 3.2. El siguiente programa asigna `NUM` (ocho enteros), mediante `cin`; a continuación visualiza el total de los números .

```
#include <iostream >
using namespace std;
const int NUM = 8;

int main()
{
    int nums[NUM];
    int total = 0;

    for (int i = 0; i < NUM; i++)
    {
        cout << "Por favor, introduzca el número";
        cin >> nums[i];
        total += nums[i];
    }
    cout << "El total de números es" << total << endl;
    return 0;
}
```

3.4. ARRAYS MULTIDIMENSIONALES

Los arrays vistos anteriormente se conocen como arrays *unidimensionales* (una sola dimensión) y se caracterizan por tener un solo subíndice. Estos arrays se conocen también por el término *listas*. Los *arrays* o **arreglos** *multidimensionales* son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los arrays más usuales son los *unidimensionales* y los de dos dimensiones, conocidos también por el nombre de *tablas* o *matrices*. Sin embargo, es posible crear arrays de tantas dimensiones como requieran sus aplicaciones, esto es, tres, cuatro o más dimensiones.

Un array de dos dimensiones equivale a una tabla con múltiples filas y múltiples columnas (Figura 3.3).

Obsérvese, que en el array bidimensional de la Figura 3.3 si las filas se etiquetan de `0` a `m` y las columnas de `0` a `n`, el número de elementos que tendrá el array será el resultado del pro-

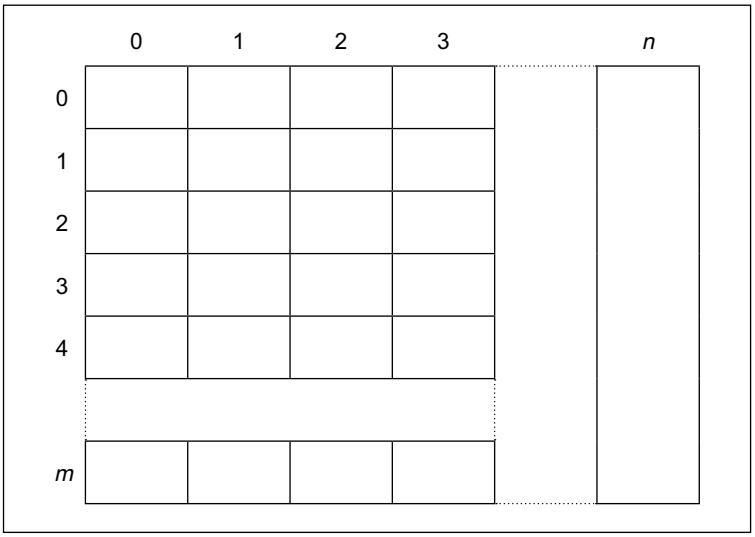


Figura 3.3. Estructura de un array de dos dimensiones.

ducto `(m+1) x (n+1)`. El sistema de localizar un elemento será por las coordenadas representadas por su número de fila y su número de columna *(a, b)*. La sintaxis para la declaración de un array de dos dimensiones es:

```
<tipo de datoElemento> <nombre array> [<NúmeroDeFilas<] [<NúmeroDeColumnas>]
```

Algunos ejemplos de declaración de tablas son:

```
char Pantalla[25][80];
int equipos[4][30];
double matriz[4][2];
```

Atención

Al contrario que otros lenguajes, C++ requiere que cada dimensión esté encerrada entre corchetes. La sentencia `int equipos[4, 30]` no es válida.

Un array de dos dimensiones en realidad es un *array de arrays*. Es decir, es un array unidimensional, y cada elemento no es un valor entero de coma flotante o carácter, sino que cada elemento es otro array.

Los elementos de los arrays se almacenan en memoria de modo que el subíndice más próximo al nombre del array es la fila y el otro subíndice, la columna. En la Tabla 3.2 se representan todos los elementos y sus posiciones relativas en memoria del array `int tabla[4] [2]` suponiendo que el tamaño de un entero son dos bytes.

Tabla 3.2. Un array bidimensional

Elemento	Posición relativa de memoria
tabla[0][0]	0
tabla[0][1]	2
tabla[1][0]	4
tabla[1][1]	6
tabla[2][0]	8
tabla[2][1]	10
tabla[3][0]	12
tabla[3][1]	14

3.4.1. Inicialización de arrays bidimensionales

Los arrays multidimensionales se pueden inicializar, al igual que los de una dimensión, cuando se declaran. La inicialización consta de una lista de constantes separadas por comas y encerradas entre llaves, como en las siguientes definiciones:

1.
```
int tabla[2][3] = {51, 52, 53, 54, 55, 56};
```

   o bien en los formatos:

```
int tabla[2][3]= { {51, 52, 53},
                   {54, 55, 56} };
int tabla[2][3]= { {51, 52, 53}, {54, 55, 56} };
int tabla[2][3]= {
                   {51, 52, 53}
                   {54, 55, 56}
                 };
```

2.
```
int tabla[3][4] = {
                     {1, 2, 3, 4},
                     {5, 6, 7, 8},
                     {9, 10, 11, 12}
                   };
```

Consejo

Los arrays multidimensionales (a menos que sean globales) no se inicializan a valores específicos a menos que se les asignen valores en el momento de la declaración o en el programa. Si se inicializan uno o más elementos, pero no todos, C++ rellena el resto con ceros o valores nulos (`'\0'`). Si se desea inicializar a cero un array multidimensional, utilice una sentencia tal como ésta.

```
float ventas[3][4] = {0.0};
```

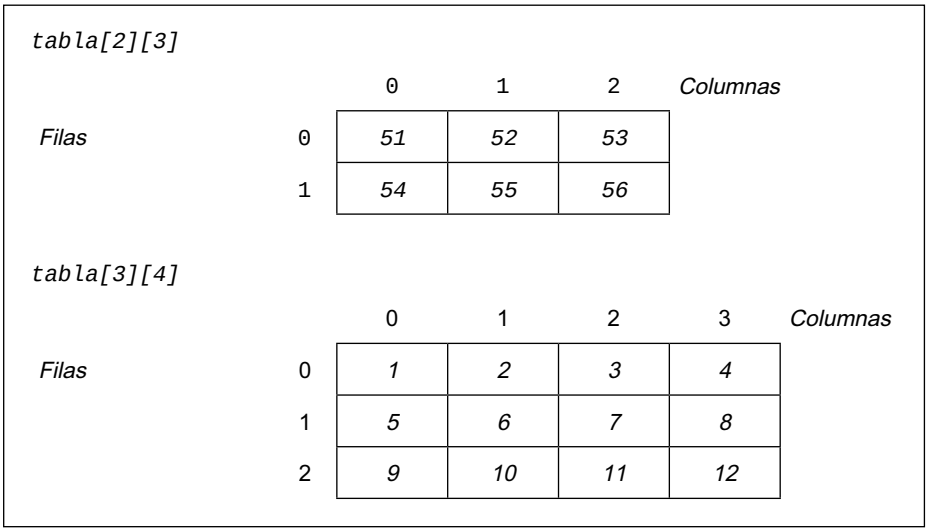


Figura 3.4. Tablas de dos dimensiones.

3.4.2. Acceso a los elementos de los arrays bidimensionales

Se puede acceder a los elementos de arrays bidimensionales de igual forma que a los elementos de un array unidimensional. La diferencia reside en que en los elementos bidimensionales deben especificarse los índices de la fila y la columna.

El formato general para asignación directa de valores a los elementos es:

Inserción de elementos

```
<nombre array>[índice fila][índice columna]= valor elemento;
```

extracción de elementos

```
<variable> = <nombre array> [índice fila] [índice columna];
```

Algunos ejemplos de inserciones pueden ser:

```
Tabla[2][3] = 4.5;
Resistencias[2][4] = 50;
```

y de extracción de valores:

```
Ventas = Tabla[1][1];
Dia = Semana[3][6];
```

Mediante bucles anidados se accede a todos los elementos de arrays bidimensionales. Su sintaxis es:

```
for (int indiceFila = 0; indiceFila < numFilas; ++indiceFila)
    for (int indiceCol = 0; indiceCol < numCol; ++indiceCol)
        Procesar elemento[indiceFila][indiceCol]
```

EJEMPLO 3.3. Se define una matriz de `N*M` elementos que se leen del teclado , a continuación se visualizan.

```
const int N = 2;
const int M = 4;

float discos[N] [M];
int fila, col;
for (fila = 0; fila < N; fila++)
{
    for (col = 0; col < M; col++)
    {
        cin >> discos[fila][col];
    }
}

// Visualizar la tabla
for (fila = 0; fila < N; fila++)
{
    cout << "Precio fila " << fila << " : ";
    for (col = 0; col < M; col++)
    {
        cout << discos[fila][col] << " ";
    }
    cout << endl;
}
```

3.4.3. *Arrays* de más de dos dimensiones

C++ proporciona la posibilidad de almacenar varias dimensiones, aunque raramente los datos del mundo real requieren más de dos o tres dimensiones.

Un *array* tridimensional se puede considerar como un conjunto de arrays bidimensionales combinados juntos para formar, en profundidad, una tercera dimensión. El cubo se construye con filas (dimensión vertical), columnas (dimensión horizontal) y planos (dimensión en profundidad). Por consiguiente, un elemento dado se localiza especificando su plano, fila y columna. Una definición de un array tridimensional `equipos` es:

```
int equipos[3][15][10];
```

Un ejemplo típico de un array de tres dimensiones es el modelo *libro*, en el que cada página de un libro es un *array* bidimensional construido por filas y columnas. Así, por ejemplo, cada página tiene cuarenta y cinco líneas que forman las filas del array y ochenta caracteres por línea, que forman las columnas del *array*. Por consiguiente, si el libro tiene quinientas páginas, existirán quinientos planos y el número de elementos será `500 x 80 x 45 = 1.800.000`.

3.5. UTILIZACIÓN DE ARRAYS COMO PARÁMETROS

En C++ *todos los arrays se pasan por referencia* (dirección). Esto significa que cuando se llama a una función y se utiliza un array como parámetro, se debe tener cuidado de no modificar los arrays en una función llamada. C++ trata automáticamente la llamada a la función como si

hubiera situado el operador de dirección & delante del nombre del array. La Figura 3.5 ayuda a comprender el mecanismo.

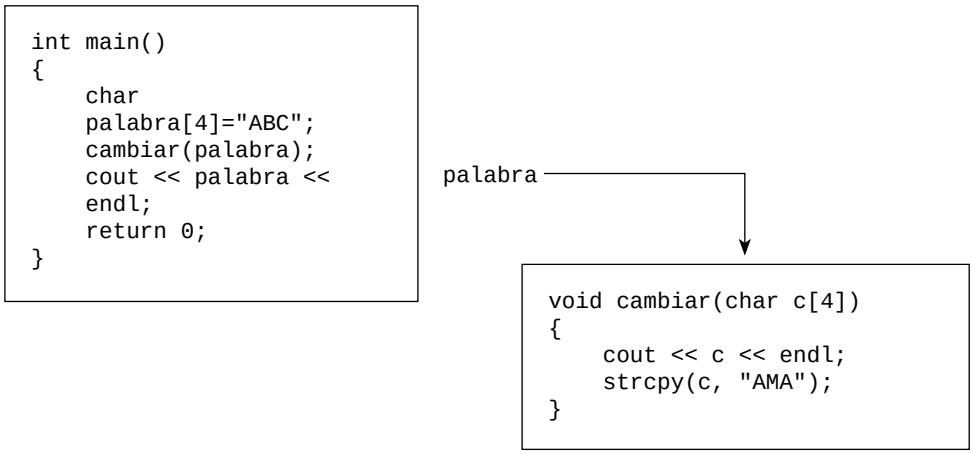


Figura 3.5. Paso de un array por dirección.

Dadas las declaraciones

```
const int MAX = 100;
double datos[MAX];
```

se puede declarar una función que acepte un array de valores `double` como parámetro. Se puede prototipar una función `SumaDeDatos`, de modo similar a:

```
double SumaDeDatos(double datos[MAX]);
```

Incluso mejor si se dejan los corchetes en blanco y se añade un segundo parámetro que indica el tamaño del array:

```
double SumaDeDatos(double datos[], int n);
```

A la función `SumaDeDatos` se pueden entonces pasar argumentos de tipo array junto con un entero `n`, que informa a la función sobre cuantos valores contiene el array. Por ejemplo, esta sentencia visualiza la suma de valores de los datos del array:

```
cout << "Suma de " << SumaDeDatos(datos, MAX) << endl;
```

La función `SumaDeDatos` no es difícil de escribir. Un simple bucle `while` suma los elementos del array y una sentencia `return` devuelve el resultado de nuevo al llamador:

```
double SumaDeDatos(double datos[], int n)
{
    double suma = 0;
    while (n > 0)
        suma += datos[--n];
    return suma;
}
```

Cuando se pasa un array a una función se pone el nombre del array, realmente se pasa *sólo* la dirección de la celda de memoria donde comienza el array que está representado por el nombre del array. La función puede cambiar entonces el contenido del array accediendo directamente a las celdas de memoria en donde se almacenan los elementos del array. Así, aunque el nombre del array se pasa por valor, sus elementos se pueden cambiar como si se hubieran pasado por referencia.

Consejo

Se pueden utilizar dos métodos alter nativos para permitir que una función conozca el número de elementos asociados con un arr ay pasado como argumento:

- Situar un valor de señal al final del arr ay, que indique a la función que se ha de detener el proceso en ese momento.
- Pasar un segundo argumento que indica el número de elementos del arr ay.

Ejemplo 3.4. Se escribe dos funciones con argumentos arr ay. La función `main()` define el array que se pasa a las funciones .

```
#include <iostream>
using namespace std;
const int LONG = 100;
void leerArray(double [], int&);
void imprimirArray (const double [], const int);

int main()
{
    double a[LONG];
    int n;
    leerArray(a, n);
    cout << "El array tiene " << n << " elementos\n son:\n";
    imprimirArray(a, n);
    return 0;
}

void leerArray(double a[], int& n)
{
    n = 0;
    cout << "Introduzca datos. Para terminar pulsar 0:\n";
    for (n = 0; n < LONG; n++)
    {
         cout << n << " : ";
         cin >> a[n];
         if (a[n] == 0) break;
    }
}

void imprimirArray(const double a[], const int n)
{
    for (int i = 0; i < n; i++)
         cout << "\t" << i << " : " << a[i] << endl;
}
```

3.6. CADENAS

Una *cadena* es un tipo de dato compuesto, un array de caracteres (`char`), terminado por un carácter *nulo* (`'\0'`), `NULL`.

Una cadena (también llamada *constante de cadena* o *literal de cadena*) es `"ABC"`. Cuando la cadena aparece dentro de un programa se verá como si se almacenarán cuatro elementos: `'A', 'B', 'C'` y `'\0'`. En consecuencia, se considerará que la cadena `"ABC"` es un array de cuatro elementos de tipo `char`.

> El número total de caracteres de una cadena es siempre igual a la longitud de la cadena más 1.

Ejemplos

1. `char cad[] = "Marisa";`
   `cad` tiene siete caracteres; `'M'`, `'a'`, `'r'`, `'i'`, `'s'`, `'a'` y `'\0'`
2. `cout << cad;`
   el sistema escribirá los caracteres de `cad` hasta que el carácter `NULL ('\0')` se encuentre.
3. `cin >> bufer;`
   el sistema copiará caracteres de `cin` a `bufer` hasta que se encuentre un carácter espacio en blanco e insertará el carácter `NULL ('\0')`. El usuario ha de asegurarse que el `bufer` se define como una cadena lo suficiente grande para contener la entrada.

> Las funciones declaradas en el archivo de cabecera <string.h> se utilizan para manipular cadenas.

3.6.1. Declaración de variables de cadena

Las cadenas se declaran como los restantes tipos de arrays. El operador postfijo`[]` contiene el tamaño máximo del array. El tipo base, naturalmente, es `char`, o bien `unsigned char`:

```
char texto[81];
unsigned char texto[121];
```

Observe que el tamaño de la cadena ha de incluir el carácter `'\0'`. En consecuencia, para definir un array de caracteres que contenga la cadena `"ABCDEF"`, escriba

```
char UnaCadena[7];
```

> A veces se puede encontrar una declaración como ésta: `char *s;`
> ¿Es realmente una cadena `s`? No, no es. Es una variable puntero a un carácter (podrá apuntar a una cadena).

3.6.2. Inicialización de variables de cadena

La inicialización de una variable cadena en su declaración se realiza mediante una constante cadena, es decir, una secuencia de caracteres encerrados entre apóstrofes (").

```
char texto[81] = "Esto es una cadena";
char cadenatest[] = "¿Cuál es la longitud de esta cadena?";
char* ptrCadena = "Porcentajes de oliva";
```

Las cadena `texto` puede contener `80` más el carácter nulo. La cadena `cadenatest` se declara con una especificación de array incompleto y se completa, sólo con el inicializador, a `36` caracteres y el carácter `'\0'`. También, la variable `ptrCadena` referencia a una cadena constante.

¿Cómo se puede inicializar una cadena fuera de la declaración? Será necesario utilizar una función de cadena, generalmente `strcpy()`. Por ejemplo:

```
char buff[121];
strcpy(buff, "Aniversario del Quijote");
```

3.6.3. Lectura de cadenas

La lectura usual de datos es con el objeto `cin` y el operador `>>`, cuando se aplica a datos de cadena producirá normalmente anomalías. Por ejemplo, el siguiente segmento:

```
char nombre[30];      // Define array de caracteres
cin >> nombre;        // Lee la cadena Nombre
cout << nombre;
```

define `nombre` como un array de caracteres de 30 elementos. Suponga que introduce la entrada `Luis Mariano`, cuando se ejecute se visualizará en pantalla `Luis`. Es decir, la palabra `Mariano` no se ha asignado a la variable `nombre`. La razón es que el objeto `cin` termina la operación de lectura siempre que se encuentra un espacio en blanco.

Entonces, ¿cuál será el método correcto para lectura de cadenas, cuando estas cadenas contienen más de una palabra (caso muy usual)? El método recomendado será utilizar una función denominada **`getline()`**, en unión con `cin`, en lugar del operador `>>`. La función `getline` permitirá a `cin` leer la cadena completa, incluyendo cualquier espacio en blanco. La sintaxis de la función **`getline()`** es:

```
istream& getline(signed char* buffer, int long, char separador = '\n');
```

Reglas

- La llamada `cin.getline(cad, n, car)` lee todas las entradas hasta la primera ocurrencia del carácter separador `car` en `cad`.
- Si el carácter especificado `car` es el carácter de nueva línea `'\n'`, la llamada anterior es equivalente a `cin.getline(cad, n)`.

EJEMPLO 3.5. El siguiente programa lee una cadena que puede estar formada por espacios en blanco.

```
#include <iostream>
using namespace std;

int main()
{
    char nombre[80];
    cout << "Introduzca su nombre ";
    cin.getline(nombre, sizeof(nombre));
    cout << "Hola " << nombre << " ¿cómo está usted?" << endl;
    return 0;
}
```

3.7. LA BIBLIOTECA `string.h`

La biblioteca estándar de C++ contiene la biblioteca de cadena `string.h`, que contiene las funciones de manipulación de cadenas utilizadas más frecuentemente. El uso de las funciones de cadena tienen una variable parámetro declarada similar a:

```
char *s1
```

Esto significa que la función espera un puntero a una cadena. Cuando se utiliza la función, se puede usar un puntero a una cadena o se puede especificar el nombre de una variable array `char`. Cuando se pasa un array a una función, C++ pasa automáticamente la dirección del array. La Tabla 3.3 resume algunas de las funciones de cadena más usuales.

Tabla 3.3. Funciones de *<string.h>*

Función	Cabecera de la función y prototipo
memcpy()	`void* memcpy(void* s1, const void* s2, size_t n);` Reemplaza los primeros *n* bytes de `*s1` con los primeros *n* bytes de `*s2`. Devuelve `s1`.
strcat()	`char *strcat(char *destino, const char *fuente);` Añade la cadena *fuente* al final de *destino*, *concatena*.
strchr()	`char* strchr(char* s1, int ch);` Devuelve un puntero a la primera ocurrencia de `ch` en `s1`. Devuelve `NULL` si `ch` no está en `s1`.

strcmp()	`int strcmp(const char *s1, const char *s2);`
	Compara alfabéticamente la cadena `s1` a `s2` y devuelve:
	`0` *si* `s1 = s2`
	`< 0` *si* `s1 < s2`
	`> 0` *si* `s1 > s2`

strcmpi()	`int strcmpi(const char *s1, const char *s2);` Igual que `strcmp()`, pero sin distinguir entre mayúsculas y minúsculas.
strcpy()	`char *strcpy(char *dest, const char *fuente);` Copia la cadena *fuente* a la cadena *destino*.

(*continúa*)

Tabla 3.3. Funciones de <*string.h*> (*continuación*)

Función	Cabecera de la función y prototipo
strcspn()	`size_t strcspn(const char* s1, const char* s2);` Devuelve la longitud de la subcadena más larga de `s1` que comienza con el carácter `s1[0]` y no contiene ninguno de los caracteres de la cadena `s2`.
strlen()	`size_t strlen (const char *s);` Devuelve la longitud de la cadena `s`.
strncat()	`char* strncat(char* s1, const char*s2, size_t n);` Añade los primeros n caracteres de `s2` a `s1`. Devuelve `s1`. Si `n >= strlen(s2)`, entonces `strncat(s1, s2, n)` tiene el mismo efecto que `strcat(s1, s2)`.
strncmp()	`int strncmp(const char* s1, const char* s2, size_t n);` Compara `s1` con la subcadena formada por los primeros n caracteres de `s2`. Devuelve un entero negativo, cero o un entero positivo, según que `s1` lexicográficamente sea menor, igual o mayor que la subcadena `s2`. Si `n ≥ strlen(s2)`, entonces `strncmp(s1, s2, n)` y `strcmp(s1, s2)` tienen el mismo efecto.
strnset()	`char *strnset(char *s, int ch, size_t n);` Copia *n* veces el carácter `ch` en la cadena `s` a partir de la posición inicial de `s (s[0])`. El máximo de caracteres que copia es la longitud de `s`.
strpbrk()	`char* strpbrk(const char* s1, const char* s2);` Devuelve la dirección de la primera ocurrencia en `s1` de cualquiera de los caracteres de `s2`. Devuelve `NULL` si ninguno de los caracteres de `s2` aparece en `s1`.
strrchr()	`char* strrchr(const char* s, int c);` Devuelve un puntero a la última ocurrencia de `c` en `s`. Devuelve `NULL` si `c` no está en `s`. La búsqueda la hace en sentido inverso, desde el final de la cadena al primer carácter, hasta que encuentra el carácter `c`.
strspn()	`size_t strspn(const char* s1, const char* s2);` Devuelve la longitud de la subcadena izquierda (`s1[0]...`) más larga de `s1` que contiene únicamente caracteres de la cadena `s2`.
strstr()	`char *strstr(const char *s1, const char *s2);` Busca la cadena `s2` en `s1` y devuelve un puntero a la subcadena de `s1` donde se encuentra `s2`.
strtok()	`char* strtok(char* s1, const char* s2);` Analiza la cadena `s1` en tokens (componentes léxicos) , estos delimitados por caracteres de la cadena `s2`. La llamada inicial a `strtok(s1, s2)` devuelve la dirección del primer tocken y sitúa `NULL` al final del tocken. Después de la llamada inicial, cada llamada sucesiva a `strtok(NULL, s2)` devuelve un puntero al siguiente token encontrado en `s1`. Estas llamadas cambian la cadena `s1`, reemplazando cada separador por el carácter `NULL`.

3.8. CLASE `string`

C++ dispone de un conjunto de clases genéricas agrupadas en la biblioteca **STL**. Las clases genéricas son, normalmente, clases *contenedoras* de objetos de cualquier tipo. Entonces, muchas implementaciones de C++, como **DEV-C++**, especializan un contenedor para el tipo `char`.

Esta especialización se le da el nombre `string` . De esta forma el usuario puede utilizar clase `string` para manejar cadenas. Por ejemplo:

```
string mipueblo = "Lupiana";
string rotulo;
rotulo = "Lista de pasajeros\n";
```

3.8.1. Variables `string`

Al ser `string` una clase se pueden definir variables, punteros, arrays... de tipo `string` como de cualquier otras clase. A continuación se define dos variables (se crean objetos) `string`. La inicialización se hace llamando a los respectivos constructores:

```
string vacia;
string tema("Marionetas de la vida");
```

La inicialización de una variable cadena se puede hacer con un literal, como se puede observar a continuación:

```
string texto = "Esto es una cadena";
```

Constructores de un objeto `string`

Los constructores de objetos `string` permiten construir desde un objeto cadena vacía hasta un objeto cadena a partir de otra cadena.

1. ***Constructor de cadena vacía***

```
string c;
string* pc = new string();
```

Se ha creado un objeto cadena sin caracteres, es una referencia a una cadena vacía. Si a continuación se obtiene su longitud (`length()`) ésta será cero.

2. ***Constructor de string a partir de otro string***

```
string c1 = "Maria de las Mercedes";
string *pc2;
pc2 = new string(c1);
```

Con este constructor se crea un objeto cadena a partir de otro objeto cadena ya creado.

3. ***Constructor de string a partir de un literal***

```
string c1("Ría de las Mercedes ");
```

3.8.2. Concatenación

El operador + permite aplicarse sobre objetos `string` para dar como resultado otro objeto `string` que es la unión o concatenación de ambas. Por ejemplo:

```
string c1 = "Ángela";
string c2 = "Paloma";
string c3 = c1 + c2;       // genera una nueva cadena: AngelaPaloma
string cd ("clásica");
cd = "Música" + cd;        // genera la cadena Musicaclasica
```

El operador + está sobrecargado o redefinido en la clase `string` para poder concatenar cadenas.

3.8.3. Longitud y caracteres de una cadena

La función **`length()`**, de la clase `string`, permite obtener el número de caracteres (*longitud*) de la cadena. Por ejemplo, la ejecución del siguiente segmento de código escribe 10 que es la longitud de la cadena asignada:

```
string digits;
digits = "0123456789";
cout << digits.length() << endl;
```

En ocasiones interesa obtener caracteres individuales de una cadena. Con variables `string`, la forma más sencilla es utilizar el operador `[]`, como si fuera un array. Para investigar cada uno de los caracteres de la cadena se construye un bucle cuyo fin queda determinado por la función `length()`. Por ejemplo:

```
for (int i = 0; i < digits.length(); i++)
{
    cout << digits[i] << endl;
}
```

3.8.4. Comparación de cadenas

La clase `string` redefine los operadores relacionales con el fin de comparar cadenas alfabéticamente. Estos métodos comparan los caracteres de dos cadenas utilizando el código numérico de su representación.

EJEMPLO 3.6. Se realizan comparaciones entre cadenas utilizando los operadores relacionales.

```
string c1 = "Universo Jamaicano";
string c2 = "Universo Visual";

if (c1 < c2)
    cout << c1 << ", es alfabéticamente menor que " << c2 << endl;
else if (c1 > c2)
    cout << c1 << ", es alfabéticamente mayor que " << c2 << endl;
cout << "Entrada de cadenas, termina con FIN" << endl;
    // bucle condicional, termina con una cadena clave
while (c1 != "FIN")
```

```
{
    cin >> c1;
    cout << "Cadena leída: " << c1 << endl;
}
```

3.9. ESTRUCTURAS

Una *estructura* (`struct`) es un tipo de dato definido por el usuario que puede encapsular uno o más tipos existentes. La sintaxis básica es:

```
struct nombreEstructura
{
    tipo1 nombreVar1;
    tipo2 nombreVar2;
    // ...
};
```

Una estructura se utiliza cuando se tratan elementos que tienen múltiples propiedades. Por ejemplo, se desea escribir una aplicación que gestiones los empleados de una empresa. Cada empleado tiene características diferentes: número de identificación del empleado, nombre, salario, edad, etc.

```
struct empleado
{
    unsigned int id;
    string nombre;
    float salario;
    int edad;
};
```

Una vez que se ha definido una variable, se pueden crear variables de ese tipo:

```
nombreEstructura miVar;

empleado e;
```

La sintaxis empleada para referenciar a los miembros individuales de una estructura utiliza el operador `"."` para acceder a los mismos.

```
miVar.nombremiembro = valor;
```

EJEMPLO 3.7. Suponiendo que se ha creado la variable `e1` de tipo `empleado` se pueden asignar valores a la variable de estructura.

```
e1.id = 4044;
e1.nombre = "Marcos";
e1.salario = 1499;
e1.edad = 66;
```

En el caso de conocer los valores de los miembros de la estructura cuando se crea una variable nueva del tipo estructura, se pueden asignar simultáneamente dichos valores:

```
empleado e1 = {4044, "Marcos", 1499, 66};
```

3.10. ENUMERACIONES

Un tipo enumerado o de enumeración es un tipo de dato cuyos valores se definen por una lista de constantes de tipo entero (`int`). Un tipo enumerado es muy similar a una lista de constantes declaradas. Los tipos enumerados pueden definir sus propias secuencias de modo que se puedan declarar variables con valores en esas secuencias.

Cuando se define un tipo enumerado se pueden utilizar cualquier valor entero y puede tener cualquier número de constantes definidas en un tipo enumerado.

Ejemplo 3.8. Tipo enumerado `diasMes`.

```
enum diasMes { DIAS_ENE = 31, DIAS_FEB = 28, DIAS_MAR = 31,
               DIAS_APR = 30, DIAS_MAY = 31, DIAS_JUN = 30,
               DIAS_JUL = 31, DIAS_AGO = 31, DIAS_SEP = 30,
               DIAS_OCT = 31, DIAS_NOV = 30, DIAS_DIC = 31 };
```

En un tipo enumerado si no se especifica ningún valor numérico, a los identificadores de la definición se asignan valores consecutivos, comenzando con `0`. Por ejemplo, la definición del tipo

```
enum rosaVientos {NORTE, SUR, ESTE, OESTE};
```

es equivalente a

```
enum rosaVientos {NORTE = 0, SUR = 1, ESTE = 2, OESTE = 3};
```

Otros ejemplos de tipos enumerados:

```
enum luna {AM, PM};
enum dia {LUN, MAR, MIE, JUE, VIE, SAB, DOM};
enum color {BLANCO, AZUL, VERDE, ROJO};
```

y variables declaradas de enumeración

```
dia a, b, c;
color c1, c2:
```

EJERCICIO 3.1. Aplicación del tipo enumerado `color`.

```
#include <iostream>
using namespace std;

int main()
{
    enum color {BLANCO, AZUL, VERDE, ROJO};
    color rotulador = ROJO;
    int x;
    cout << "\n El color es " << rotulador << endl;
    cout << "Introduzca un valor: "; cin >> x;
    rotulador = (color)x;
```

```
    if (rotulador == ROJO)
        cout << "El rotulador es rojo" << endl;
    else if (rotulador == VERDE)
        cout << "El rotulador es verde" << endl;
    else if (rotulador == AZUL)
        cout << "El rotulador es azul" << endl;
    else if (rotulador == BLANCO)
        cout << "El rotulador es blanco" << endl;
    else
        cout << "El color es indefinido" << endl;
    return 0;
}
```

Si se ejecuta el programa se obtiene la siguiente salida:

```
El color es 3
Introduzca un valor: 1
El rotulador es verde
```

Regla

El nombre de un tipo de dato definido por el usuar io se puede omitir en la definición,

```
enum {BLANCO, AZUL, VERDE, ROJO} rotulador;
```

`rotulador` es un tipo enumerado con los valores válidos:

```
BLANCO, AZUL, VERDE, ROJO
```

RESUMEN

Los datos atómicos, son datos simples que no se pueden descomponer. Un tipo de dato atómico es un conjunto de datos atómicos con propiedades idénticas. Los tipos de datos atómicos se definen por un conjunto de valores y un conjunto de operaciones que actúan sobre esos valores.

Una estructura de datos es un agregado de datos atómicos y datos compuestos en un conjunto con relaciones bien definidas. Un array es un tipo de dato estructurado que se utiliza para localizar y almacenar elementos de un tipo de dato dado. Existen arrays de una dimensión, de dos dimensiones... y multidimensionales.

En C++ los arrays se definen especificando el tipo de dato del elemento, el nombre del array y el tamaño de cada dimensión del array. Para acceder a los elementos del array se deben utilizar sentencias de asignación directas, sentencias de lectura/escritura o bucles (mediante las sentencias `for`, `while` o `do-while`). Los arrays de caracteres contienen cadenas de textos. La biblioteca string.h ofrece una amplia variedad de funciones para el proceso de cadenas.

La clase `string`, disponible en las últimas implementaciones de C++, facilita el manejo de de cadenas. Los constructores de la clase permiten inicializar el objeto cadena con otra cadena, o bien con un literal. Se puede asignar una variable `string` a otra, concatenar con el operador +, comparar alfabéticamente con los operadores relacionales.

EJERCICIOS

Para los ejercicios 3.1 a 3.6 suponga las declaraciones:

```
int i,j,k;
int Primero[21];
int Segundo[21];
int Tercero[7][8];
```

Determinar la salida de cada segmento de programa si las entradas son las indicadas en negrita.

3.1.
```
for (i = 1; i <= 6; i++)
    cin >> Primero[i];
for (i = 3; i > 0; i--)
    cout << Primero[2*i] << " ";
.................
3   7  4   -1   0   6
```

3.2.
```
cin >> k ;
for (i = 3; i <= k;)
    cin >> Segundo[i++];
j = 4;
cout << Segundo[k] << " " << Segundo[j + 1];
.............
6   3   0   1   9
```

3.3.
```
for (i = 0; i < 10; i++)
    Primero[i] = i + 3;
cin >> j >> k;
for (i = j; i <= k;)
    cout << Primero[i++];
............
7   2   3   9
```

3.4.
```
for (i = 0, i < 12; i++)
    cin >> Primero[i];
for (j = 0; j < 6; j++)
    Segundo[j] = Primero[2 * j] + j;
for (k = 3; k <= 7; k++)
    cout << Primero[k + 1] << " " << Segundo [k – 1];
...........
2   7   3   4   9  -4   6  -5   0   5   -8   1
```

3.5.
```
for (j = 0; j < 7; )
    cin >> Primero[j++];
i = 0;
j = 1;
while ((j < 6) && (Primero[j – 1] < Primero[j]))
{
    i++,j++;
}
for (k = -1; k < j + 2; )
    cout << Primero[++k];
..........
20   60   70   10    0   40   30   90
```

3.6.
```
for (i = 0; i < 3; i++)
    for (j = 0; j < 12; j++)
        Tercero[i][j] = i + j + 1;
for (i = 0; i< 3; i++)
{
    j = 2;
    while (j < 12)
    {
        cout << i << " " << j << " " << Tercero[i][j]);
        j += 3;
    }
}
```

3.7. Escribir un programa que lea el array

```
4    7    1    3    5
2    0    6    9    7
3    1    2    6    4
```

y lo escriba como

```
4    2    3
7    0    1
1    6    2
3    9    6
5    7    4
```

3.8. Dado el array

```
4    7    -5    4    9
0    3    -2    6    -2
1    2    4    1    1
6    1    0    3    -4
```

escribir una función que encuentre la suma de todos los elementos que no pertenecen a la diagonal principal.

3.9. Escribir una función que intercambie la fila i-ésima por la j-ésima de un array de dos dimensiones, `m x n`.

3.10. Escribir una función que tenga como entrada una cadena y devuelva el número de vocales, de consonantes y de dígitos de la cadena.

3.11. ¿Qué diferencias y analogías existen entre las variables `c1`, `c2`? La declaración es:

```
string c1 = "Cadena de favores";
char c2[] = "Filibusteros de hoy";
```

3.12. Definir un array de cadenas para poder leer un texto compuesto por un máximo de 80 líneas. Escribe un método para leer el texto; el método debe de tener dos argumentos, uno el texto y el segundo el número de líneas.

3.13. Escribir un función que reciba una matriz `A`, y devuelva la matriz transpuesta de `A`.

3.14. Escribir una función que acepte como parámetro un array que puede contener números enteros duplicados. El método debe sustituir cada valor repetido por –5 y devolver el vector modificado y el número de entradas modificadas.

PROBLEMAS

3.1. Un texto de n líneas tiene ciertos caracteres que se consideran comodines. Hay dos comodines, el # y el ? El primero indica que se ha de sustituir por la fecha actual, en formato día (nn) *de* mes (nombre) año (aaaa), por ejemplo 21 de abril 2001. El otro comodín indica que se debe reemplazar por un nombre. Escribir un programa que lea las líneas del texto y cree un array de cadenas, cada elemento referencia a una cadena que es el resultado de realizar las sustituciones indicadas. La fecha y el nombre se ha de obtener del flujo de entrada.

3.2. Escribir un programa que permita visualizar el triángulo de Pascal:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
```

En el triángulo de Pascal cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando un array de una sola dimensión.

3.3. Se sabe que en las líneas de que forma un texto hay valores numéricos enteros, representan los Kg de patatas recogidos en una finca. Los valores numéricos están separados de las palabras por un blanco, o el carácter fin de línea. Escribir un programa que lea el texto y obtenga la suma de los valores numéricos.

3.4. Escribir un programa que lea una cadena clave y un texto de como máximo 50 líneas. El programa debe de eliminar las líneas que contengan la clave.

3.5. Se quiere sumar números grandes, tan grandes que no pueden almacenarse en variables de tipo `long`. Por lo que se ha pensado en introducir cada número como una cadena de caracteres y realizar la suma extrayendo los dígitos de ambas cadenas.

3.6. Escribir un programa que visualice un cuadrado mágico de orden impar *n* comprendido entre 3 y 11; el usuario debe elegir el valor de *n*. Un cuadrado mágico se compone de números enteros comprendidos entre 1 y *n*. La suma de los números que figuran en cada fila, columna y diagonal son iguales.

Ejemplo

```
8   1   6
3   5   7
4   9   2
```

Un método de generación consiste en situar el número 1 en el centro de la primera fila, el número siguiente en la casilla situada por encima y a la derecha, y así sucesivamente. El cuadrado es cíclico: la línea encima de la primera es, de hecho, la última y la columna a derecha de la última es la primera. En el caso de que el número generado caiga en una casilla ocupada, se elige la casilla situada encima del número que acaba de ser situado.

3.7. El juego del ahorcado se juega con dos personas (o una persona y una computadora. Un jugador selecciona una palabra y el otro jugador trata de adivinar la palabra adivinando letras individuales. Diseñar un programa para jugar al ahorcado. *Sugerencia:* almacenar una lista de palabras en un vector y seleccionar palabras aleatoriamente.

3.8. Escribir un programa que lea las dimensiones de una matriz, lea y visualice la matriz y a continuación encuentre el mayor y menor elemento de la matriz y sus posiciones.

3.9. Si x representa la media de los números x_1, x_2, ... x_n, entonces la *varianza* es la media de los cuadrados de las desviaciones de los números de la media.

$$Varianza = \frac{1}{n}\sum_{i=1}^{n}(x_1 - x)^2$$

Y la desviación estándar es la raíz cuadrada de la varianza. Escribir un programa que lea una lista de números reales, los cuente y a continuación calcule e imprima su media, varianza y desviación estándar. Utilizar una función para calcular la media, otra para calcular la varianza y otro para la desviación estándar.

3.10. Los resultados de las últimas elecciones a alcalde en el pueblo *x* han sido los siguientes:

Distrito	*Candidato A*	*Candidato B*	*Candidato C*	*Candidato D*
1	194	48	206	45
2	180	20	320	16
3	221	90	140	20
4	432	50	821	14
5	820	61	946	18

Escribir un programa que haga las siguientes tareas:

a) Imprimir la tabla anterior con cabeceras incluidas.
b) Calcular e imprimir el número total de votos recibidos por cada candidato y el porcentaje del total de votos emitidos. Asimismo, visualizar el candidato más votado.
c) Si algún candidato recibe más del 50 por 100 de los datos, el programa imprimirá un mensaje declarándole ganador.
d) Si ningún candidato recibe más del 50 por 100 de los datos, el programa debe imprimir el nombre de los dos candidatos más votados, que serán los que pasen a la segunda ronda de las elecciones.

3.11. Una agencia de venta de vehículos automóviles distribuye quince modelos diferentes y tiene en su plantilla diez vendedores. Se desea un programa que escriba un informe mensual de las ventas por vendedor y modelo, así como el número de automóviles vendidos por cada vendedor y el número total de cada modelo vendido por todos los vendedores. Asimismo, para entregar el premio al mejor vendedor, necesita saber cuál es el vendedor que más coches ha vendido.

Modelo / Vendedor	1	2	3	4 15
1	4	8	1	4
2	12	4	25	14
3	15	3	4	7
.				
.				
10				

3.12. Escribir un programa que lea una línea de caracteres, y visualice la línea de tal forma que las vocales sean sustituidas por el carácter que más veces se repite en la línea.

3.13. Escribir un programa que encuentre dos cadenas introducidas por teclado que sean anagramas. Se considera que dos cadenas son anagramas si contienen exactamente los mismos caracteres en el mismo o en diferente orden. Hay que ignorar los blancos y considerar que las mayúsculas y las minúsculas son iguales.

3.14. Se dice que una matriz tiene un *punto de silla* si alguna posición de la matriz es el menor valor de su fila, y a la vez el mayor de su columna. Escribir un programa que tenga como entrada una matriz de números reales, y calcule la posición de un *punto de silla* (si es que existe).

3.15. Escribir un programa en el que se genere aleatóriamente un array de 20 números enteros. El array ha de quedar de tal forma que la suma de los 10 primeros elementos sea mayor que la suma de los 10 últimos elementos. Mostrar el array original y el array con la distribución indicada.

CAPÍTULO 4

Clases derivadas: herencia y polimorfismo

Objetivos

Con el estudio de este capítulo usted podrá:

- Conocer el concepto de clases derivadas.
- Aprender a realizar herencia simple y múltiple.
- Conocer el concepto de ligadura y funciones virtuales.
- Manejar el concepto de polimorfismo.

Contenido

Conceptos clave

- Clase abstracta.
- Clase base.
- Clase derivada.
- Constructor.
- Declaración de acceso.
- Destructor.
- Especificadores de acceso.
- Función virtual.
- Herencia.
- Herencia pública y privada.
- Herencia múltiple.
- Herencia simple.
- Ligadura dinámica.
- Polimorfismo.

INTRODUCCIÓN

En este capítulo se introduce el concepto de *herencia* y se muestra cómo crear *clases derivadas*. La herencia hace posible crear jerarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases. La *herencia* es la propiedad que permite definir nuevas clases usando como base a clases ya existentes. La nueva clase (*clase derivada*) hereda los atributos y comportamiento que son específicos de ella. La herencia es una herramienta poderosa que proporciona un marco adecuado para producir software fiable, comprensible, bajo coste, adaptable y reutilizable.

El *polimorfismo* permite que diferentes objetos respondan de modo diferente al mismo mensaje. El polimorfismo adquiere su máxima potencia cuando se utiliza en unión de la herencia. El polimorfismo hace los sistema más flexible, sin perder ninguna de las ventajas de la compilación estática de tipos que tienen lugar en tiempo de compilación.

4.1. CLASES DERIVADAS

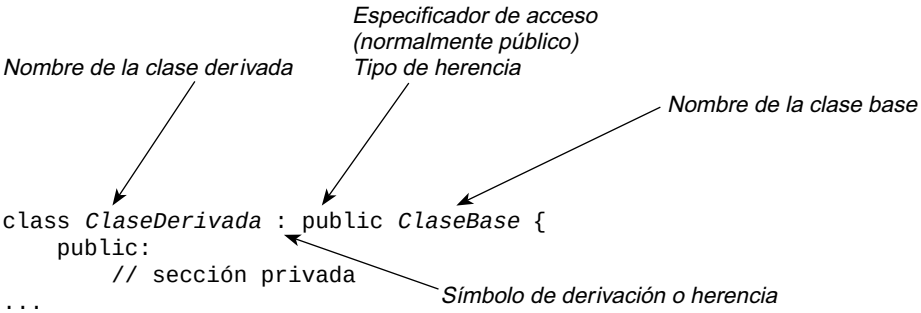
La *herencia* es la relación que existe entre dos clases, en la que una clase denominada *derivada* se crea a partir de otra ya existente, denominada *clase base*. Así, por ejemplo, si existe una clase `Figura` y se desea crear una clase `Triángulo`, esta clase `Triángulo` puede derivarse de `Figura` ya que tendrá en común con ella un estado y un comportamiento, aunque luego tendrá sus características propias. `Triángulo` *es-un* tipo de `Figura`. Otro ejemplo, puede ser **`Programador`** *es-un* de **`Empleado`**.

Figura 4.1. Clases derivadas.

C++ soporta el mecanismo de *derivación* que permite crear clases derivadas, de modo que la nueva clase *hereda* todos los miembros datos y las funciones miembro que pertenecen a la clase ya existente.

4.1.1. Declaración de una clase derivada

La sintaxis para la declaración de una clase derivada es:



Especificador de acceso public, significa que los miembros públicos de la clase base son miembros públicos de la clase derivada.

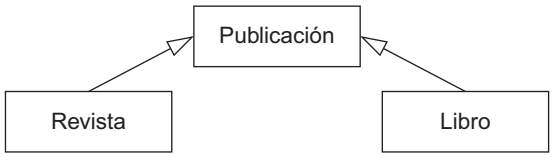
Herencia pública, es aquella en que el especificador de acceso es `public` (*público*).
Herencia privada, es aquella en que el especificador de acceso es `private` (*privado*).
Herencia protegida, es aquella en que el especificador de acceso es `protected` (*protegido*).

El especificador de acceso que declara el tipo de herencia es opcional (`public`, `private` o `protected`); si se omite el especificador de acceso, se considera por defecto `private`. La *clase base* (*ClaseBase*) es el nombre de la clase de la que se deriva la nueva clase. La *lista de miembros* consta de datos y funciones miembro:

```
class nombre_clase : especificador_acceso ClaseBase {
   lista_de_miembros;
};
```

EJEMPLO 4.1. Representar la jer arquía de clases de pub licaciones que se distr ibuyen en una librería: revistas, libros, etc.

Todas las publicaciones tienen en común una editorial y una fecha de publicación. Las revistas tienen una determinada periodicidad lo que implica el número de ejemplares que se publican al año, y por ejemplo, el número de ejemplares que se ponen en circulación controlados oficialmente (por ejemplo, en España la OJD). Los libros, por el contrario, tienen un código de ISBN y el nombre del autor.



Las clases en C++ se especifican así:

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Publicacion {
public:
  void NombrarEditor(const char *S);
  void PonerFecha(unsigned long fe);

private:
  string editor;
  unsigned long fecha;
};

class Revista : public Publicacion {
public:
  void NumerosPorAnyo(unsigned n);
  void FijarCirculacion(unsigned long n);

private:
  unsigned numerosPorAnyo;
  unsigned long circulacion;
};

class Libro : public Publicacion {
public:
  void PonerISBN(const char *s);
  void PonerAutor(const char *s);
private:
  string ISBN;
  string autor;
};
```

Así, en el caso de un objeto `Libro`, éste contiene miembros dato y funciones heredadas del objeto `Publicación`, así como `ISBN` y nombre del `autor`. En consecuencia serán posibles las siguientes operaciones:

```
Libro L;
L.NombrarEditor("McGraw-Hill");
L.PonerFecha(010906);
L.PonerISBN("84-481-4643-3");
L.PonerAutor("Luis Joyanes, Lucas Sánchez");
```

Por el contrario, las siguientes operaciones sólo se pueden ejecutar sobre objetos `Revista`:

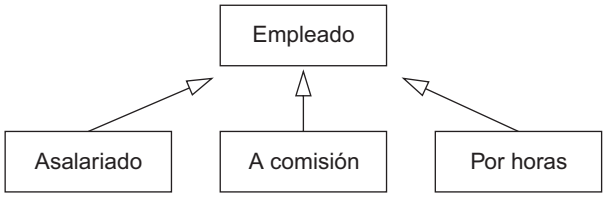
```
Revista R;
R.NumerosPorAnyo(12);
R.FijarCirculacion(200000L);
```

Si no existe la posibilidad de utilizar la herencia, sería necesario hacer una copia del código fuente de una clase, darle un nuevo nombre y añadirle nuevas operaciones y/o miembros dato.

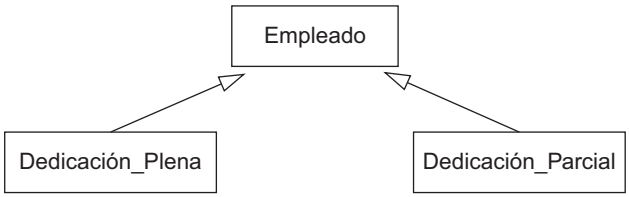
4.1.2. Consideraciones de diseño

A veces, es difícil decidir cuál es la relación de herencia más óptima entre clases en el diseño de un programa. Consideremos, por ejemplo, el caso de los empleados o trabajadores de una empresa. Existen diferentes tipos de clasificaciones según el criterio de selección (se suele llamar *discriminador*) y pueden ser: modo de pago (sueldo fijo, por horas, a comisión); dedicación a la empresa (plena o parcial) o estado de su relación laboral con la empresa (fijo o temporal).

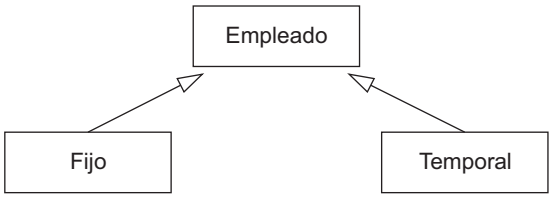
Una vista de los empleados basada en el modo de pago puede dividir a los empleados con salario mensual fijo; empleados con pago por horas de trabajo y empleados a comisión por las ventas realizadas:



Una vista de los empleados basada en el estado de dedicación a la empresa: dedicación plena o dedicación parcial.



Una vista de empleados basada en el estado laboral del empleado con la empresa: fija o temporal.



Una dificultad a la que suele enfrentarse el diseñador es que en los casos anteriores un mismo empleado puede pertenecer a diferentes grupos de trabajadores. Un empleado con dedicación plena puede ser remunerado con un salario mensual. Un empleado con dedicación parcial puede ser remunerado mediante comisiones y un empleado fijo puede ser remunerado por horas.

4.2. TIPOS DE HERENCIA

En una clase existen secciones públicas, privadas y protegidas. Los elementos públicos son accesibles a todas las funciones; los elementos privados son accesibles sólo a los miembros de

la clase en que están definidos y los elementos protegidos pueden ser accedidos por clases derivadas debido a la propiedad de la herencia. En correspondencia con lo anterior existen tres tipos de herencia: *pública*, *privada* y *protegida*. Normalmente, el tipo de herencia más utilizada es la herencia pública.

Con independencia del tipo de herencia, una clase derivada no puede acceder a variables y funciones privadas de su clase base. Para ocultar los detalles de la clase base y de clases y funciones externas a la jerarquía de clases, una clase base utiliza normalmente elementos protegidos en lugar de elementos privados. Suponiendo herencia pública, los elementos protegidos son accesibles a las funciones miembro de todas las clases derivadas.

Tabla 4.1. Tipos de herencia y accesos que permiten

Tipo de herencia	Acceso en la clase base	Acceso en la clase derivada
`public`	`public`	`public`
	`protected`	`protected`
	`private`	*`inaccesible`*
`protected`	`public`	`protected`
	`protected`	`protected`
	`private`	*`inaccesible`*
`private`	`public`	`private`
	`protected`	`private`
	`private`	`inaccesible`

La Tabla 4.1 resume los efectos de los tres tipos de herencia en la accesibilidad de los miembros de la clase derivada. La entrada *inaccesible* indica que la clase derivada no tiene acceso al miembro de la clase base.

Norma

Por defecto, la herencia es privada. Si accidentalmente se olvida la palabra reservada `public`, los elementos `public` y `protected` de la clase base se heredan con acceso `private`. El tipo de herencia es, por consiguiente, una de las primeras cosas que se debe verificar si un compilador devuelve un mensaje de error que indique que las variables o funciones son inaccesibles.

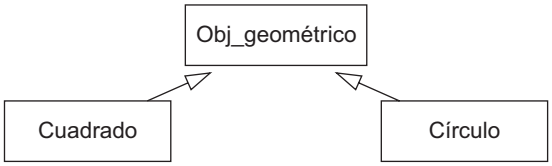
4.2.1. Herencia pública

En general, *herencia pública* significa que una clase derivada tiene acceso a los elementos públicos y protegidos de su clase base. Los elementos públicos se heredan como elementos públicos; los elementos protegidos permanecen protegidos; lo elementos privados no se heredan. La herencia pública se representa con el especificador `public` en la derivación de clases.

Formato

```
class: Clase Derivada: public Clase Base {
public:
   // sección pública
private:
   // sección privada
};
```

EJEMPLO 4.2. Considérese la jerarquía `obj_geometrico`, `cuadrado` y `circulo`.



La clase `obj_geom` de objetos geométricos se declara como sigue:

```
class obj_geom {
public:
  obj_geom(float x = 0, float y = 0) : xC(x), yC(y) {}
  void imprimircentro() const
  {
    cout << xC << " " << yC << endl;
  }
protected:
  float xC, yC;
};
```

Un círculo se caracteriza por su centro y su radio. Un cuadrado se puede representar también por su centro y uno de sus cuatro vértices. Declaremos las dos figuras geométricas como clases derivadas.

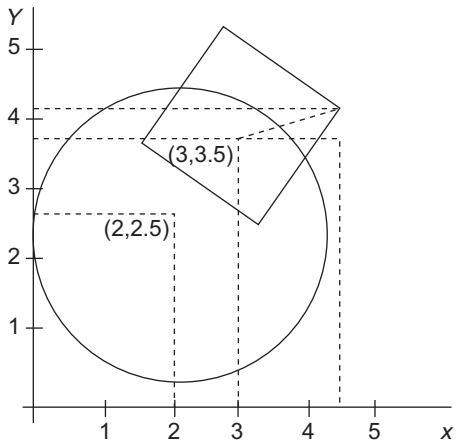


Figura 4.2. Círculo (centro: 2, 2.5), cuadrado (centro: 3, 3.5).

```
const float PI = 3.14159265;

class circulo : public obj_geom {
public:
  circulo(float x_C, float y_C, float r) : obj_geom (x_C, y_C)
  {
     radio = r;
  }
  float area() const {return PI * radio * radio; }
private :
  float radio;
};
class cuadrado :public obj_geom {
public :
  cuadrado(float x_C, float y_C, float x, float y)
          : obj_geom(x_C, y_C)
  {
     x1 = x;
     y1 = y;
  }
  float area() const
  {
     float a, b;
     a = x1 - xC;
     b = y1 - yC;
     return 2 * (a * a + b * b);
}
private:
  float x1, y1;
};
```

Todos los miembros públicos de la clase base `obj_geom` se consideran también como miembros públicos de la clase derivada `cuadrado`. La clase `cuadrado` se deriva públicamente de `obj_geom`. Se puede escribir

```
cuadrado C(3, 3.5, 4.37, 3.85);
C.imprimircentro();
```

Aunque `imprimircentro` no sucede directamente en la declaración de la clase `cuadrado` es, no obstante, una de sus funciones miembro públicas ya que es un miembro público de la clase `obj_geom` de la que se deriva públicamente `cuadrado`. Otro punto observado es el uso de `xC` e `yC` en la función miembro `area` de la clase `cuadrado`. Éstos son miembros protegidos de la clase base `obj_geom`, por lo que tienen acceso a ellos desde la clase derivada.

Una función `main` que utiliza las clases `cuadrado` y `circulo` y la salida que se produce tras su ejecución:

```
int main()
{
  circulo C(2, 2.5, 2);
  cuadrado Cuad(3, 3.5, 4.37, 3.85);
  cout << " centro del circulo : "; C.imprimircentro();
```

```
  cout << " centro del cuadrado : " ; Cuad.imprimircentro();
  cout << "Area del circulo : " ; C.area() << endl;
  cout << "Area del cuadrado : " ; Cuad.area() << endl;
  return 0 ;
}

Centro del circulo : 2 2.5
Centro del cuadrado : 3 3.5
Area del circulo : 12.5664
Area del cuadrado : 3.9988
```

Regla

Con herencia pública, los miembros, de la clase derivada, heredados de la clase base tienen la misma protección que en la clase base . La herencia púb lica se utiliza en la práctica casi siempre ya que modela directamente la relación ***es-un***.

4.2.2. Herencia privada

La herencia privada significa que un usuario de la clase derivada no tiene acceso a ninguno de sus elementos de la clase base. El formato es:

```
class ClaseDerivada: private ClaseBase
{
public:
  // sección pública
protected:
  // sección protegida
private:
  // sección privada
};
```

Con herencia privada, los miembros públicos y protegidos de la clase base se vuelven miembros privados de la clase derivada. En efecto, los usuarios de la clase derivada no tienen acceso a las facilidades proporcionadas por la clase base. Los miembros privados de la clase base son inaccesibles a las funciones miembro de la clase derivada.

La herencia privada se utiliza con menos frecuencia que la herencia pública. Este tipo de herencia oculta la clase base del usuario y así es posible cambiar la implementación de la clase base o eliminarla toda sin requerir ningún cambio al usuario de la interfaz. Cuando un especificador de acceso no está presente en la declaración de una clase derivada, se utiliza herencia privada.

4.2.3. Herencia protegida

Con herencia protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada y los miembros privados de la clase base se vuelven inaccesibles. La herencia protegida es apropiada cuando las facilidades o aptitudes de la clase base son útiles en la implementación de la clase derivada, pero no son parte de la interfaz que

el usuario de la clase ve. La herencia protegida es todavía menos frecuente que la herencia privada.

EJEMPLO 4.3. Declarar una clase base (*Base*) y tres clases der ivadas de ella, `D1`, `D2` y `D3`.

```
class Base {
public:
  int i1;
protected:
  int i2;
private:
  int i3;
};

class D1: private Base {
  void f();
};
class D2: protected Base {
  void g();
};
class D3: public Base {
  void h();
};
```

Ninguna de las subclases tienen acceso al miembro `i3` de la clase `Base`. Las tres clases pueden acceder a los miembros `i1` e `i2`. En la definición de la función miembro `f()` se tiene:

```
void D1::f() {
  i1 = 0;    // Correcto
  i2 = 0;    // Correcto
  i3 = 0;    // Error
};
```

4.2.4. Operador de resolución de ámbito

Si se utiliza herencia privada o protegida, existe un método de hacer a los miembros de la clase base accesibles en una clase derivada: utilizar una *declaración de acceso*. Esto se consigue nombrando uno de los miembros de la clase base en un lugar apropiado de la clase derivada.

```
class D4: protected base {
public
  Base::i1;        // declaración de acceso
};
```

Al declarar `i1` en la sección pública de `D4` se hace a `i1` público en `D4`. Se puede, entonces, escribir

```
D4 d4;
d4.i1 = 0;         // CORRECTO
```

4.2.5. Constructores-Inicializadores en herencia

Una clase derivada es una especialización de una clase base. En consecuencia, el constructor de la clase base debe ser llamado para crear un objeto de la clase base antes de que el constructor de la clase derivada realice su tarea. Haciendo un símil sucede lo mismo que con objetos de las clases derivadas; el objeto de la clase base debe existir antes de convertirse en un objeto de la clase derivada.

Regla

1. Los constructores de las clases base se invocan antes del constructor de la clase derivada; los constructores de la clase base se invocan en la secuencia en que están especificados.
2. Si una clase base es, a su vez, una clase derivada, sus constructores se invocan también en secuencia: constructor base, constructor derivada.
3. Los constructores no se heredan, aunque los constructores por defecto y de copia, se generan si se requiere.

EJEMPLO 4.4

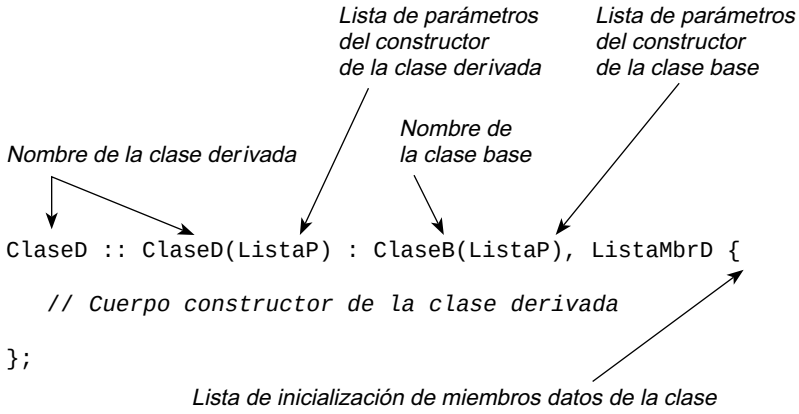
```
class B1 {
public:
  B1() { cout << "C-B1" << endl; }
};

class B2 {
public:
  B2() { cout << "C-B2" << endl; }
};

class D: public B1, B2 {
public:
  D() { cout << "C-D" << endl; }
};

D d1;
```

Este ejemplo visualiza

```
C-B1
C-B2
C-D
```

4.2.6. Sintaxis del constructor

La sintaxis de un constructor de una clase derivada es:



Obsérvese, que la primera línea incluye una llamada al constructor de la clase base. El constructor de la clase base se llama antes de que se ejecute el cuerpo del constructor de la clase derivada. Esta secuencia tiene sentido ya que el objeto base constituye el fundamento del objeto derivado (se necesita el objeto base antes de convertirse en objeto derivado). El constructor de una clase derivada tiene que realizar dos tareas:

- Inicializar el objeto base.
- Inicializar todos los miembros dato.

La clase derivada tiene un *constructor-inicializador*, que llama a uno o más constructores de la clase base. El inicializador aparece inmediatamente después de los parámetros del constructor de la clase derivada y está precedido por dos puntos (`:`).

EJEMPLO 4.5. La clase `Punto3D` es una clase derivada de la clase `Punto`.

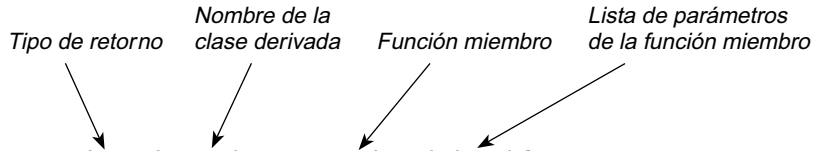
Formato general: `Punto3D::Punto3D(listaP):`*`inicializador-constructor`*

En la implementación del constructor de `Punto3D` se pasan los parámetros `xv` e `yv` al constructor de `Punto`.

```
class Punto {
public:
  Punto(int xv, int yv){ x = xv; y = yv; }
private:
  int x, y;
};

class Punto3D : public Punto {
public:
  Punto3D(int xv, int yv, int zv): Punto(xv, yv){
                FijarZ(zv);
  }
  void FijarZ(int zv){ z= zv; }
private:
  int z;
};
```

4.2.7. Sintaxis de la implementación de una función miembro

La sintaxis para la definición de la implementación de las funciones miembro de una clase derivada es idéntica a la sintaxis de la definición de la implementación de una clase base.



4.3. DESTRUCTORES

Los destructores no se heredan, aunque se genera un destructor por defecto si se requiere. Un destructor normalmente sólo se utiliza cuando un constructor correspondiente ha asignado espacio de memoria que debe ser liberado. Los destructores se manejan como los constructores excepto que todo se hace en orden inverso.

EJEMPLO 4.6. Declaración de una clase `C1` y de una clase der ivada `C2` de `C1` con una llamada a constructores y por defecto a sus destructores.

```
class C1 {
public:
  C1(int n);
  ~C1();
private:
  int *pi, l;
};

C1::C1(int n) : l(n)
{
  cout << l << " enteros se asignan " << endl;
  pi = new int[l];
}
C1::~C1()
{
   cout << l << " enteros son liberados " << endl;
   delete[] pi;
}
class C2 : public C1 {
public :
  C2(int n);
  ~C2();
private :
  char *pc;
  int l;
} ;
```

```
C2 ::C2(int n) :C1(n), l(n)
{
  cout << l << " caracteres son asignados " << endl;
  pc = new char[l];
}
C2::~C2()
{
  cout << l << " caracteres son liberados " << endl;
  delete[] pc;
}
int main()
{
  C2 a(50), b(100);
}
```

Cuando de ejecuta el programa, se visualiza:

```
50 enteros se asignan.
50 caracteres se asignan.
100 enteros se asignan.
100 caracteres se asignan.
100 caracteres son liberados.
100 enteros son liberados.
50 caracteres son liberados.
50 enteros son liberados.
```

4.4. HERENCIA MÚLTIPLE

Herencia multiple es un tipo de herencia en la que una clase hereda el estado (estructura) y el comportamiento de más de una clase base. Es decir, existen múltiples clases base (*ascendientes* o *padres*) para la clase derivada (*descendiente* o *hija*).

La herencia múltiple entraña un concepto más complicado que la herencia simple, no sólo con respecto a la sintaxis sino también al diseño e implementación del compilador. En la Figura 4.3 se muestran diferentes ejemplos de herencia múltiple.

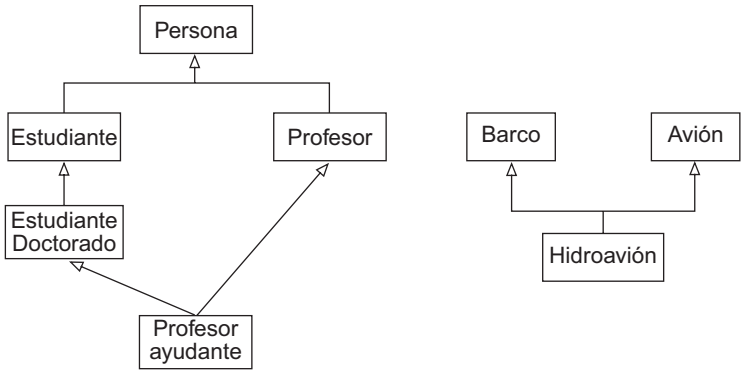


Figura 4.3. Ejemplos de herencia múltiple.

Regla

En herencia simple, una clase derivada hereda exactamente de una clase base (tiene sólo un padre). Herencia múltiple implica múltiples clases bases (tiene varios padres una clase derivada).

La herencia múltiple siempre se puede eliminar y convertirla en herencia simple si el lenguaje de implementación no la soporta o considera que tendrá dificultades en etapas posteriores a la implementación real. La sintaxis de la herencia múltiple es:

```
class CDerivada: [virtual][tipo_acceso] Base1,
                 [virtual][tipo_acceso] Base2,
                 [virtual][tipo_acceso] Basen {
public:
  // sección pública
private:
  // sección privada
...
};
```

`CDerivada`	Nombre de la clase derivada.
`tipo_acceso`	`public`, `private` o `protected`, con las mismas reglas que la herencia simple.
`Base1, Base2,...`	Clases base con nombres diferentes.
`virtual..`	La palabra reservada `virtual` es opcional y especifica una clase base compatible.

Funciones o datos miembro que tengan el mismo nombre en `Base1`, `Base2`, `Basen`, serán motivo de ambigüedad.

Regla

Asegúrese especificar un tipo de acceso en todas las clases base para evitar el acceso privado por omisión. Utilice explícitamente `private` cuando lo necesite para manejar la legibilidad.

```
class Derivada: public Base1, private Base2 {...}
```

4.4.1. Características de la herencia múltiple

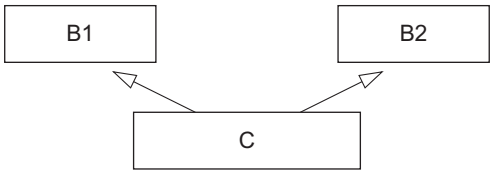
La herencia múltiple plantea diferentes problemas tales como la *ambigüedad* por el uso de nombres idénticos en diferentes clases base, y la *dominación* o *preponderancia* de funciones o datos.

Los problemas que se pueden presentar cuando se diseñan clases con herencia múltiple son:

- **colisiones de nombres** de diferentes clases base (dos o más clases base tiene el mismo identificador para algún elemento de su interfaz. Se resuelve con el operador de ámbito `::`).
- **herencia repetida** de una misma clase base (una clase puede heredar indirectamente dos copias de una clase base. Se resuelve con el operador de resolución ámbito `::`).

EJEMPLO 4.7. Herencia múltiple con colisión de nombre . Repetición de un atr ibuto en las clases base y der ivada.

Las clases `B1` y `B2` tienen un atributo entero `x`. La clase derivada `C` es derivada de las clases `B1` y `B2`, por lo que tiene dos atributos `x` asociados a las clases: `B` hereda de `B1`; y `B` hereda de `B2`. La clase `C` hereda públicamente de la clase `B1` y `B2`, por lo que puede acceder a los tres atributos con el mismo nombre `x`. Estos atributos son: uno local a `C`, y otros dos de `B1` y `B2`. La función miembro `verx` puede retornar el valor de `x`, pero puede ser el de `B1`, `B2`, `C` resuelto con el operador de ámbito `::`. La función miembro `probar` es pública y amiga de la clase `C`, por lo que puede modificar los tres distintos atributos `x` de cada una de las clases. Para resolver el problema del acceso a cada uno de ellos se usa el operador de ámbito `::`

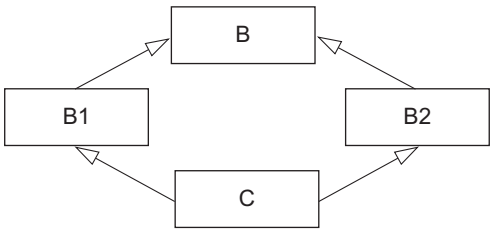



```
class B1
{
protected:
   int x;
};

class B2
{
protected:
   int x;
};

class C: public B1,B2
{
protected:
   int  x;
public:
   friend void probar();      // es amiga y tiene acceso a x
   int verx()
   {
      return x;               // x de C
      return B1::x;           // x de B1
      return B2::x;           // x de B2
   }
};
```

```
void probar()
{
   C c1;
   c1.x = 12;                // atributo x de C
   c1.B1::x = 123;           // atributo x de B1 resolución de colisión
   c1.B2::x = 143;           // atributo x de B2 resolución de colisión
}
```

EJEMPLO 4.8. Herencia múltiple repetida. Repetición de una clase a través de dos clases derivadas.

Las clases `B1` y `B2` tienen un mismo atributo entero `x`. La clase derivada `C` también tiene el mismo atributo entero `x`. La clase `C` hereda públicamente de la clase `B1` y `B2`, por lo que puede acceder a los tres atributos con el mismo nombre `x`. La función miembro `fijarx()` fija los atributos de las dos `x` resolviendo el problema con el operador de ámbito `::`.




```
class B
{
protected:
   int x;
};

class B1: public B
{
protected:
   float y;
};

class B2: public B
{
protected:
   float z;
};

class C: public B1, B2
{
protected:
   int t;
public:
   void fijax(int xx1, int xx2)
      {
         B1::x = xx1;
         B2::x = xx2;
      }
};
```

Regla

Cuando la dominación crea ambigüedades , deben realizarse llamadas directas en la clase `Derivada` a la respectiva clase `Base`

```
class C: public B1, B2 {
public:
   void fijax (int xx1, int xx2)
   {
      B1::x = xx1;
      B2::x = xx2;
   }
};
```

4.5. CLASES ABSTRACTAS

Una clase abstracta, normalmente, ocupa una posición alta en la jerarquía de clases que le permite actuar como un depósito de métodos y atributos compartidos para las subclases de nivel inmediatamente inferior. Las clases abstractas definen un tipo generalizado y sirven solamente para describir nuevas clases.

Las clases abstractas no tienen instancias directamente. Se utilizan para agrupar otras clases y capturar información que es común al grupo. Sin embargo, las subclases de clases abstractas se corresponden a objetos del mundo real y pueden tener instancias. Las superclases que se crean a partir de subclases con atributos y comportamientos comunes, y que sirven para derivar otras clases que comparten sus características, son clases abstractas.

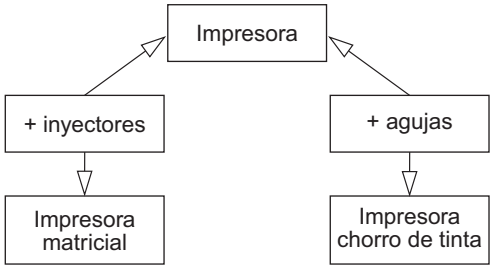
En C++, una clase abstracta tiene al menos una función miembro que se declara pero no se define; su definición se realiza en la clase derivada. Estas funciones miembro se denominan funciones virtuales puras, cuya definición formal es:

```
virtual tipo nombrefuncion(argumentos) = 0;
```

Las siguientes reglas se aplican a las clases abstractas:

- Una clase abstracta debe tener al menos una función virtual pura.
- Una clase abstracta no se puede utilizar como un tipo de argumento o como un tipo de retorno de una función auque sí un puntero a ella.
- No se puede declarar una instancia de una clase abstracta.
- Se puede utilizar un puntero o referencia a una clase abstracta.
- Una clase derivada que no proporcione una definición de una función virtual pura, también es una clase abstracta.
- Cada clase (derivada de una clase abstracta) que proporciona una definición de todas sus funciones virtuales es una clase concreta.
- Sólo está permitido crear punteros a las clases abstracta y pasárselos a funciones.

EJEMPLO 4.9. Una clase abstracta puede ser una impresora.




```
class impresora
{
   public:
      virtual int arrancar() = 0;
};

class ChorroTinta: public impresora
{
   protected:
      float chorro;
   public:
      int arrancar( ) {return 100;}
};

class matricial: public impresora
{
   protected:
      int agujas;
   public:
      int arrancar( ) {return 10;}
};

int main(int argc, char *argv[])
{
   impresora im;        //no puede declararse es abstracta
   matricial m;
   ChorroTinta ch;
   return 0;
}
```

4.6. LIGADURA

Ligadura representa generalmente, una conexión entre una entidad y sus propiedades. Si la propiedad se limita a funciones, ligadura es la conexión entre la llamada a función y el código que se ejecuta tras la llamada. Desde el punto de vista de atributos, la *ligadura* es el proceso de asociar un atributo a un nombre.

La ligadura se clasifica según sea el tiempo o momento de la ligadura: *estática* y *dinámica*. Ligadura estática se produce antes de la ejecución (durante la compilación), mientras que la ligadura dinámica ocurre durante la ejecución.

En un lenguaje de programación con ligadura estática, todas las referencias se determinan en tiempo de compilación. La mayoría de los lenguajes procedimentales son de ligadura estática.

La ligadura dinámica supone que el código a ejecutar en respuesta a un mensaje no se determinará hasta el momento de la ejecución. Unicamente la ejecución del programa determinará la ligadura efectiva entre las diversas que son posibles (una para cada clase derivada).

La principal ventaja de la ligadura dinámica frente a la ligadura estática es que la ligadura dinámica ofrece un alto grado de flexibilidad y ofrece además diversas ventajas prácticas y manejar jerarquías de clases de un modo muy simple. Entre las desventajas está que la ligadura dinámica es menos eficiente que la ligadura estática.

Los lenguajes orientados a objetos que siguen estrictamente el paradigma orientado a objetos ofrecen sólo ligadura dinámica.

La ligadura en C++ es, por defecto, estática. La ligadura dinámica se produce cuando se hace preceder a la declaración de la función con la palabra reservada `virtual`. Sin embargo, puede darse el caso de ligadura estática, pese a utilizar `virtual`, a menos que el receptor se utilice como un puntero o como una referencia.

4.7. FUNCIONES VIRTUALES

Por omisión, las funciones C++ tienen ligadura estática; si la palabra reservada `virtual` precede a la declaración de una función, esta función se llama virtual, y le indica al compilador que puede ser definida (implementado su cuerpo) en una clase derivada y que en este caso la función se invocará directamente a través de un puntero. Se debe calificar una función miembro de una clase con la palabra reservada `virtual` sólo cuando exista una posibilidad de que otras clases puedan ser derivadas de aquélla.

Un uso común de las funciones virtuales es la declaración de clases abstractas y la implementación del polimorfismo.

EJEMPLO 4.10. Declaración e implementación de funciones vir tuales

Si se considera la clase `figura` como la clase base de la que se derivan otras clases, tales como `rectangulo`, `circulo` y `triangulo`. Cada `figura` debe tener la posibilidad de calcular su área y poder dibujarla. En esta caso, la clase `figura` declara las funciones virtuales `calcular_area` y `dibujar`, que son implementadas en las clases `circulo` y `triángulo`. Cada clase derivada específica debe definir sus propias versiones concretas de las funciones que han sido declaradas virtuales en la clase base. Por consiguiente, si se derivan las clases `circulo` y `rectangulo` de la clase `figura`, se deben definir las funciones miembro `calcular_area` y `dibujar` en cada clase.

```
class figura
{
public:
  virtual double calcular_area(void) const = 0;
  virtual void dibujar(void) const = 0;
```

```
  // otras funciones miembro que definen un interfaz a todos los
  // tipos de figuras geométricas
};

class circulo : public figura
{
public:
  double calcular_area(void) const;
  void dibujar(void) const;
  // ...
private:
  double xc, yc;       // coordenada del centro
  double radio;        // radio del círculo
};

#define PI 3.14159     // valor de "pi"
// Implementación de calcular_area

double circulo::calcular_area(void) const
{
  return PI * radio * radio;
}
// Implementación de la función dibujar
void circulo::dibujar(void) const
{
  // ...
}
class triangulo : public figura
{
  public:
     double calcular_area(void) const;
     void dibujar(void) const;
     // ...
  private:
     //
};
```

Cuando se declaran las funciones `dibujar` y `calcular_area` en la clase derivada, se puede añadir opcionalmente la palabra reservada `virtual` para destacar que estas funciones son verdaderamente virtuales. Las definiciones de las funciones no necesitan la palabra reservada `virtual`.

4.7.1. Ligadura dinámica mediante funciones virtuales

Las funciones virtuales se tratan igual que cualquier otra función miembro de una clase. Como ejemplo considérense las siguientes llamadas a las funciones virtuales `dibujar` y `calcular_area`:

```
circulo c1;
triangulo t1
double area = c1.calcular_area();      // calcular área del círculo
c1.dibujar();                          // dibujar un circulo
```

```
double area = t1.calcular_area();  // calcular área de un triangulo
t1.dibujar();                      // dibujar un triángulo
```

El uso anterior es similar al de cualquier función miembro. En este caso, el compilador C++ puede determinar que se llama a la función `calcular_area` de la clase `circulo` y a la función `dibujar` de la clase `triangulo`. De hecho, el compilador hace llamadas directas a estas funciones, y las llamadas a funciones se enlazan a un código específico en tiempo de enlace. Esta es la ligadura que hemos denominado anteriormente *estática*.

Sin embargo, el caso más interesante es la llamada a las funciones a través de un puntero a `figura`, tal como:

```
figura* s[10];      // punteros a 10 objetos figuras
int i, numfiguras = 10;
// crea figuras y almacena punteros en array s
// dibujar las figuras
    for (i = 0; i < numfiguras; i++)
        figura[i] -> dibujar();
```

En este caso se produce ligadura dinámica; el compilador C++ no puede determinar cuál es la implementación específica de la función `dibujar()` que se ha de llamar.

Un puntero a una clase derivada es también un puntero a la clase base

En C++ se puede utilizar una referencia o un puntero a cualquier clase base, en lugar de una referencia o puntero a la clase derivada, sin una conversión explícita de tipos. De modo que si `circulo` y `triangulo` se derivan de la clase `figura`, se puede llamar a una función que requiera un puntero a círculo o triángulo con un puntero a figura.

Lo opuesto no es cierto; no se puede llamar a una función que requiera un puntero a figura con un puntero a círculo o triángulo.

En C++ las funciones virtuales siguen una regla concreta: la función se debe declarar como `virtual` en la primera clase en que está presente. Esta regla significa que normalmente las funciones virtuales se declaran en la clase de nivel más alto de una jerarquía.

4.8. POLIMORFISMO

En POO, el *polimorfismo* permite que diferentes objetos respondan de modo diferente al mismo mensaje. El polimorfismo adquiere su máxima potencia cuando se utiliza en unión de herencia.

EJEMPLO 4.11. Polimorfimso y funciones virtuales.

Si `Poligono` es una clase base de la que cada figura geométrica hereda características comunes. C++ permite que cada clase utilice funciones o métodos `Area`, `Visualizar`, `Perimetro`, `PuntoInterior` como nombre de una función miembro de las clases derivadas. Es en estas clases derivadas donde se definen las funciones miembro.

```
class Poligono
{    // superclase
public:
  virtual float Perimetro();
  virtual float Area();
  virtual bool PuntoInterior();
  virtual void Visualizar();

};
// la clase Rectángulo debe definir las funciones virtuales que use de
//la clase polígono
class Rectangulo : public Poligono
{
private:
  float Alto, Bajo,Izquierdo, Derecho;
public:
    float Perimetro();
    float Area();
    bool PuntoInterior();
    void Visualizar();
    void fijarRectangulo();
};

//la clase Triángulo debe definir las funciones virtuales que use de
//la clase polígono

class Triangulo : public Poligono{
     float uno, dos, tres;
public:
  float Area();
  bool PuntoInterior();
  void fijarTriangulo();
}
```

El polimorfismo permite utilizar el mismo interfaz, tal como métodos (funciones miembro) denominados `Area` y `PuntoInterior`, para trabajar con toda clase de polígonos.

La forma más adecuada de usar polimorfismo es a través de punteros. Supongamos que se dispone de una colección de objetos `Poligono` en una estructura de datos, tal como un array o una lista enlazada. El array almacena simplemente punteros a objetos `poligono`. Estos punteros apuntan a cualquier tipo de polígono. Cuando se actúa sobre estos polígonos, basta simplemente recorrer el array con un bucle e invocar a la función miembro apropiada mediante el puntero a la instancia. Naturalmente, para realizar esta tarea las funciones miembro deben ser declaradas como virtuales en la clase `Poligono`, que es la clase base de todos los polígonos.

Para poder utilizar polimorfismo en C++ se deben seguir las siguientes reglas:

1. Crear una jerarquía de clases con las operaciones importantes definidas por las funciones miembro declaradas como virtuales en la clase base.
2. Las implementaciones específicas de las funciones virtuales se deben hacer en las clases derivadas. Cada clase derivada puede tener su propia versión de las funciones. Por ejemplo, la implementación de la función `dibujar` varía de una figura a otra.
3. Las instancias de estas clases se manipulan a través de una referencia o un puntero. Este mecanismo es la ligadura dinámica y es la esencia del uso polimórfico en C++.

Se obtiene ligadura dinámica sólo cuando las funciones miembro virtuales se invocan a través de un puntero, de la clase base a una instancia de una clase y derivada.

4.9. LIGADURA DINÁMICA FRENTE A LIGADURA ESTÁTICA

La ligadura dinámica se implementa en C++ mediante *funciones virtuales*. Con ligadura dinámica, la selección del código a ejecutar cuando se llama a una función virtual se retrasa hasta el tiempo de ejecución. Esto significa que cuando se llama a una función virtual, el código ejecutable determina en tiempo de ejecución cuál es la versión de la función que se llama. Recordemos que las funciones virtuales son polimórficas y, por consiguiente, tienen diferentes implementaciones para clases diferentes de la familia.

La ligadura estática se produce cuando se define una función polimórfica para diferentes clases de una familia y el código real de la función se conecta o enlaza en tiempo de compilación. Las funciones sobrecargadas se enlazan estáticamente. Con funciones sobrecargadas, el compilador puede determinar cuál es la función a llamar basada en el número y tipos de datos de los parámetros de función. Sin embargo, las funciones virtuales tienen la misma interfaz dentro de una familia de clases dada. Por consiguiente, los punteros se deben utilizar durante el tiempo de ejecución para determinar cuál es la función a llamar.

Las funciones virtuales se declaran en una clase base en C++ utilizando la palabra reservada **`virtual`**. Cuando se declara una función como una función virtual de una clase base, el compilador conoce cuál es la definición de la clase base que se puede anular en una clase derivada. La definición de la clase base se anula (reemplaza) definiendo una implementación diferente para la misma función en la clase derivada. Si la definición de la clase base no se anula en una clase derivada, entonces la definición de la clase base está disponible a la clase derivada.

4.10. VENTAJAS DEL POLIMORFISMO

El polimorfismo hace su sistema más flexible, sin perder ninguna de las ventajas de la compilación estática de tipos que tienen lugar en tiempo de compilación. El polimorfismo en C++ es una herramienta muy potente y que puede ser utilizada en muchas situaciones diferentes. Las aplicaciones más frecuentes del polimorfismo son:

Estructuras de datos heterogéneos. Con polimorfismo se pueden crear y manejar fácilmente estructuras de datos heterogéneos, que son fáciles de diseñar y dibujar, sin perder la comprobación de tipos de los elementos utilizados.
Gestión de una jerarquía de clases. Las jerarquías de clases son colecciones de clases altamente estructuradas, con relaciones de herencia que se pueden extender fácilmente.

RESUMEN

Una clase nueva que se crea a partir de una clase ya existente, utilizando herencia, se denomina *clase derivada* o *subclase*. La clase padre se denomina *clase base* o *superclase.*

Herencia simple es la relación entre clases que se produce cuando una nueva clase se crea utilizando las propiedades de una clase ya existente. La nueva clase se denomina *clase derivada.*

La *Herencia múltiple*, se produce cuando una clase se deriva de dos o más clases base. Aunque es una herramienta potente, puede crear problemas, especialmente de colisión o conflictos de nombres.

Polimorfismo es la propiedad de que algo, tome diferentes formas. En un lenguaje orientado a objetos el polimorfismo es la propiedad por la que un mensaje puede significar cosas diferentes dependiendo del objeto que lo recibe. La razón por la que el polimorfismo es útil se debe a que propociona la capacidad de manipular instancias de clases derivadas a través de un conjunto de operaciones definidas en su clase base. Cada clase derivada puede implementar las operaciones definidas en la clase base.

Una clase abstracta es aquella que contiene al menos una función virtual pura. Una función virtual pura es una función miembro que se declara utilizando un especificador puro que significa que el prototipo de la función termina en un = o. La función se debe implementar en cualquier clase derivada.

EJERCICIOS

4.1. Definir una clase base persona que contenga información de propósito general común a todas las personas (nombre, dirección, fecha de nacimiento, sexo, etc.). Diseñar una jerarquía de clases que contemple las clases siguientes: `estudiante`, `empleado`, `estudiante_empleado`.

Escribir un programa que lea un archivo de información y cree una lista de personas: *a*) general; *b*) estudiantes; *c*) empleados; *d*) estudiantes empleados. El programa deber permitir ordenar alfabéticamente por el primer apellido.

4.2. Implementaar una jerarquía `Librería` que tenga al menos una docena de clases. Considérese una *librería* que tenga colecciones de libros de literatura, humanidades, tecnología, etc.

4.3. Diseñar una jerarquía de clases que utilice como clase base o raíz una clase LAN (red de área local).

Las subclases derivadas deben representar diferentes topologías, como *estrella*, *anillo*, *bus y hub***.** Los miembros datos deben representar propiedades tales como *soporte de transmisión*, *control de acceso*, *formato del marco de datos*, *estándares*, *velocidad de transmisión*, etc. Se desea simular la actividad de los nodos de tal LAN.

La red consta de nodos, que pueden ser dispositivos tales como computadoras personales, estaciones de trabajo, máquinas FAX, etc. Una tarea principal de LAN es soportar comunicaciones de datos entre sus nodos. El usuario del proceso de simulación debe, como mínimo poder:

- Enumerar los nodos actuales de la red LAN.
- Añadir un nuevo nodo a la red LAN.
- Quitar un nodo de la red LAN.
- Configurar la red, proporcionándole una topología de *estrella* o en *bus*.

- Especificar el tamaño del paquete, que es el tamaño en bytes del mensaje que va de un nodo a otro.
- Enviar un paquete de un nodo especificado a otro.
- Difundir un paquete desde un nodo a todos los demás de la red.
- Realizar estadísticas de la LAN, tales como tiempo medio que emplea un paquete.

4.4. Implementar una clase `Automovil` (*Carro*) dentro de una jerarquía de herencia múltiple. Considere que, además de ser un *Vehículo*, un automóvil es también una *comodidad, un símbolo de estado social, un modo de transporte*, etc. `Automovil` debe tener al menos tres clases base y al menos tres clases derivadas.

4.5. Escribir una clase `FigGeometrica` que represente figuras geométricas tales como *punto, línea, rectángulo, triángulo* y similares. Debe proporcionar métodos que permitan dibujar, ampliar, mover y destruir tales objetos. La jerarquía debe constar al menos de una docena de clases.

4.6. Implementar una jerarquía de tipos datos numéricos que extienda los tipos de datos fundamentales tales como `int` y `float`, disponibles en C++. Las clases a diseñar pueden ser `Complejo`, `Fracción`, `Vector`, `Matriz`, etc.

4.7. Diseñar la siguiente jerarquía de clases:

	Persona Nombre edad visualizar()		
Estudiante		*Profesor*	
nombre	*heredado*	nombre	*heredado*
edad	*heredado*	edad	*heredado*
id	*definido*	salario	*definido*
visualizar()	*redefinido*	visualizar()	*heredada*

Escribir un programa que manipule la jerarquía de clases, lea un objeto de cada clase y lo visualice.

a) Sin utilizar funciones virtuales.
b) Utilizando funciones virtuales.

CAPÍTULO 5

Genericidad: plantillas *(templates)*

Objetivos

Con el estudio de este capítulo usted podrá:

- Conocer el concepto de Genericidad.
- Definir plantillas para usar la Genericidad en C++.
- Diferenciar las plantillas del polimorfismo.
- Manejar las plantillas de funciones.
- Usar y definir plantillas de clases.

Contenido

Conceptos clave

- Función de plantilla.
- Genericidad.
- Plantillas en C++.
- Plantillas frente a polimorfismo.
- Polimorfismo.
- `template`.
- Tipo genérico.
- Tipo parametrizado.
- `typename`.

INTRODUCCIÓN

Una de las ideas claves en el mundo de la programación es la posibilidad de diseñar clases y funciones que actúen sobre tipos arbitrarios o genéricos. Para definir clases y funciones que operan sobre tipos arbitrarios, se definen los *tipos parametrizados* o *tipos genéricos*. La mayoría de los lenguajes de programación orientados a objetos proporcionan soporte para la genericidad: los *paquetes* en Ada, las *plantillas* (*templates*) en C++. C++ proporciona la característica **plantilla** (*template*), que permite a los tipos ser parámetros de clases y funciones. C++ soporta esta propiedad a partir de la versión 2.1 de AT&T.

Las *plantillas* o *tipos parametrizados* se pueden utilizar para implementar estructuras y algoritmos que son en su mayoría independientes del tipo de objetos sobre los que operan. Por ejemplo, una plantilla `Pila` puede describir cómo implementar una pila de objetos arbitrarios; una vez que la plantilla se ha definido, los usuarios pueden escribir el código que utiliza pilas de tipos de datos reales, cadenas, enteros, punteros, etc.

5.1. GENERICIDAD

La genericidad es una propiedad que permite definir una clase (o una función) sin especificar el tipo de datos de uno o más de sus miembros (parámetros). De esta forma se puede cambiar la clase para adaptarla a los diferentes usos sin tener que reescribirla.

La razón de la genericidad se basa principalmente en el hecho de que los algoritmos de resolución de numerosos problemas no dependen del tipo de datos que procesa, y sin embargo, cuando se implementan en un lenguaje de programación, los programas que resuelven cada algoritmo serán diferentes para cada tipo de dato que procesan.

Ejemplos típicos de clases genéricas son *pilas*, *colas*, *listas*, *conjuntos*, *diccionarios*, *arrays*, etc. Estas clases se definen con independencia del tipo de los objetos contenidos, y es el usuario de la clase quien deberá especificar el tipo de argumento de la clase en el momento que se instancia.

La genericidad ha sido definida de diversas formas; en nuestro caso hemos seleccionado la definición dada por Bertrand Meyer[1] (autor del lenguaje Eiffel).

En el caso más común, los parámetros representan tipos. Los módulos reales, denominados instancias del módulo genérico, se obtienen proporcionando tipos reales para cada uno de los parámetros genéricos.

La genericidad se implementa en Ada mediante unidades genéricas y en C++ con plantilla de clases y plantillas de funciones.

5.2. PLANTILLAS DE FUNCIONES

Una *plantilla de funciones* especifica un conjunto infinito de funciones sobrecargadas. Cada función de este conjunto es una *función plantilla* y una instancia de la plantilla de función. Una función plantilla apropiada se produce automáticamente por el compilador cuando sea necesario.

---

[1] *Genericidad* es la capacidad de definir módulos parametrizados. Tal módulo, denominado módulo genérico, no es directamente útil; más bien es un patrón de módulos.

5.2.1. Fundamentos teóricos

Supongamos que se desea escribir una función `min(a, b)` que devuelve el valor más pequeño de sus argumentos.

C++ impone una declaración precisa de los tipos de argumentos necesarios que recibe `min()`, así como el tipo de valor devuelto por `min()`. Es necesario utilizar diferentes funciones sobrecargadas `min()`, cada una de las cuales se aplica a un tipo de argumento específico. Un programa que hace uso de funciones `min()` es:

```
// archivo PLANFUN.CPP
#include <iostream>
using namespace std;

// datos enteros (int)
int min(int a, int b)
{
  if (a <= b)
    return a;
  return b;
}

// datos largos
long min(long a, long b)
{
  if (a <= b)
    return a;
  return b;
}

// datos double
double min(double a, double b)
{
  if (a <= b)
    return a;
  return b;
}

int main()
{
  int ea = 1, eb = 5;
  cout << "(int):" << min(ea, eb) << endl;

  long la = 10000, lb = 4000;
  cout << "(long):" << min(la, lb) << endl;

  double da = 423.654, db = 789.10;
  cout << "(double):" << min(da, db) << endl;
}
```

Al ejecutar el programa se visualiza:

```
(int) : 1
(long): 4000
(double): 423.654
```

Obsérvese que las diferentes funciones `min()` tienen un cuerpo idéntico, pero como los argumentos son diferentes, se diferencian entre sí en los prototipos. En este caso sencillo se podría evitar esta multiplicidad de funciones definiendo una macro con `#define`:

```
#define min(a, b) ((a) <= (b) ? (a) : (b))
```

Sin embargo, con la macro se perderán los beneficios de las verificaciones de tipos que efectúa C++ para evitar errores. Para seguir disponiendo de las ventajas de las verificaciones de tipos se requieren las plantillas de funciones.

5.2.2. Definición de plantilla de función

Una *plantilla de función* o *función plantilla* especifica un conjunto infinito de funciones sobrecargadas y describe las propiedades genéricas de una función.

La innovación clave en las plantillas de funciones es representar el tipo de dato utilizado por la función no como un tipo específico tal como `int`, sino por un nombre que representa a cualquier tipo. Normalmente, este tipo se representa por `T` (aunque puede ser cualquier nombre que decida el programador, tal como `Tipo`, `UnTipo`, `Complejo` o similar; es decir, cualquier identificador distinto de una palabra reservada).

La sintaxis de una plantilla de funciones tiene dos formatos, según se utilice la palabra reservada `class` o `typename`. Ambos formatos se pueden utilizar: `class`, es el formato clásico y que incorpora todos los compiladores y, `typename`, es el formato introducido por el estándar ANSI/ISO C++ para utilizar en lugar de `class`.

Sintaxis

```
1. template <class T>
2. template <typename T>
```

`T` es un parámetro tipo, que puede ser reemplazado por cualquier tipo, y que también se conoce como un argumento de la plantilla. También se puede especificar un puntero (`T *` parámetro) o una referencia (`T &` parámetro).

Sintácticamente, no hay ninguna diferencia entre las dos palabras reservadas, `class` y `typename`, y se pueden usar, una u otra, indistintamente.

Una definición de plantilla comienza con la palabra reservada `template` seguida por una *lista de parámetros de la plantilla*, que es una lista de uno o más parámetros de plantilla, separados por comas, encerrados entre corchetes tipo ángulo (< y >). *La lista de parámetros no puede estar vacía*.

Reglas prácticas

- La palabra reservada `class` se puede utilizar en lugar de `typename` y tienen el mismo significado en este contexto (una recomendación puede ser : utilizar `class` cuando el argumento deba ser una clase y `typename` cuando el argumento pueda ser cualquier tipo).

- La palabra reservada `template` indica al compilador que el código que sigue es una plantilla o patrón de funciones, no la cabecera o definición real de una función.
- Los parámetros de tipo (y los argumentos para las plantillas de clase) aparecen entre corchetes de desigualdad (< >).
- Las plantillas de funciones, al contrario que las funciones ordinarias, no se pueden separar en un archivo de cabecera que tenga su cabecera y un archivo compilado por separado que contenga sus definiciones. Las definiciones se deben compilar con

Las cabeceras de las plantillas de funciones no se pueden guardar en un archivo de cabecera `nombre.h` y las definiciones en un archivo de cabecera `nombre.ccp` que se compilen de modo separado. Una práctica habitual es poner todo en el mismo archivo.

La función puede declararse con un parámetro formal o con múltiples parámetros formales y devolver, inclusive, un valor de tipo `T`. Algunas posibles declaraciones pueden ser:

1.
```
template <class T> T & f(T parámetro)
{
  // cuerpo de la función
}
```
2.
```
template <typename T> T f(int a, T b)
{
  // cuerpo de la función
}
```
3.
```
template <class T> T f(T a, T b)
{
  // cuerpo de la función
}
```

Se pueden declarar también dos parámetros tipo `T1` y `T2` distintos.

4.
```
template <typename T1, class T2> T1 f(T1 a, T2 b)
{
  // cuerpo de la función
}
```

Una función plantilla se puede declarar externa (`extern`), en línea (`inline`) o estática (`static`), de igual forma que una función no plantilla. El especificador correspondiente se sitúa a continuación de la línea de parámetros formales (*nunca delante de la palabra reservada template*).

```
// declaración correcta
template <class T> inline T f(T a, T b)
{
  // cuerpo de la función
}

// declaración incorrecta
extern template <class T> T f(T a, T b)
{
  // cuerpo de la función
}
```

5.2.3. Un ejemplo de plantilla de funciones

Se desea diseñar una plantilla que calcule el menor valor de dos datos dados. Sea, por ejemplo, la función plantilla `min()`, que se define como:

```
template <class T> T min(T a, T b)
{
  if (a <= b)
    return a;
  else
    return b;
}
```

La sintaxis anterior especifica que la función `min()` está parametrizada en la función del *tipo de datos* `T`. Para ver la diferencia entre la función plantilla `min()` y las restantes funciones, obsérvese el siguiente programa:

```
void main()
{
    int ea = 1, eb = 5;
    cout << "(int):" << min(ea, eb) << endl;

    long la = 10000, lb = 4000;
    cout << "(long):" << min(la, lb) << endl;

    double da = 423.654, db = 789.10;
    cout << "(double):" << min(da, db) << endl;
}
```

Cuando el compilador encuentra una llamada de la forma `min(a, b)`, *instancia* la función `min()` a partir de los tipos de parámetros `a` y `b` utilizados en la llamada de la función. Así, el tipo genérico `T` es sustituido por parámetros `ea` y `eb`, `la` y `lb`, etc.

EJEMPLO 5.1. La plantilla función `min()` se puede declarar también así (archivo `min.h`):

```
template <typename T> T min(const T a, const T b)
{
  return a < b ? a : b;
}
```

Algunos ejemplos de llamadas a la plantilla función `min()`:

```
#include "min.cpp"
int i, j, k;
double u, v, w;

...
k = min(i, j);
w = min(u, v);
```

Estas dos llamadas a `min` harán que C++ genere dos funciones plantilla con los siguientes prototipos:

```
int min(const int a, const int b);
double min(const double a, const double b);
```

EJEMPLO 5.2. El archivo `minimaxi.h` declara dos plantillas de funciones: `mini()` y `maxi()`.

```
//Archivo minimaxi.h

//plantilla de función maxi
template <class T> T maxi(T a, T b)
{
  if (a > b)
    return a;
  else
    return b;
}

// plantilla de función mini

template <class T> T mini(T a, T b)
{
  if (a < b)
    return a;
  else
    return b;
}
```

Un programa que utiliza funciones plantilla se muestra en el listado siguiente.

```
#include <iostream>
using namespace std;
#include "minimaxi.h"

int main()
{
  int e1 = 100, e2 = 200;
  double d1 = 3.141592, d2 = 2.718283;

  cout << "maxi(e1, e2) es igual a: " << maxi(e1, e2) << "\n";
  cout << "maxi(d1, d2) es igual a: " << maxi(d1, d2) << "\n";
  return 0;
}
```

EJEMPLO 5.3. La función plantilla `intercambio` intercambia dos valores del mismo tipo.

```
#include <iostream>
using namespace std;
template <class T>
```

```
void intercambio(T& a, T& b) {
  T aux = a;
  a = b;
  b = aux;
}

int main() {
                                          // intercambio de enteros
  int x = 50;
  int y = 120;
  cout << "antes del intercambio, x = "
       << x << ", y = " << y << endl;
  intercambio(x, y);
  cout << "después del intercambio, x = "
       << x << ", y = " << y << endl;
                                          // intercambio de cadenas
  string c1 = "Marta";
  string c2 = "Lorena";
  cout << "antes del intercambio, c1 = "
       << c1 << ", c2 = " << c2 << endl;
  intercambio(c1, c2);
  cout << "después del intercambio, c1 = "
       << c1 << ", c2 = " << c2 << endl;
  return 0;
}
```

5.2.4. Función plantilla `ordenar`

Se puede utilizar la plantilla `intercambio` para construir una función plantilla `ordenar`:

```
template <class T>
void ordenar(T* v, int n)
{
  for (int indice = 0; indice < n; indice++)
  {
    int posmin = indice;
    for( int i = indice + 1; indice < n; i++)
      if (v[i] < v[posmin])
          posmin = i;
    intercambio(v[indice], v[posmin]);
  }
}
```

Ahora las siguientes declaraciones

```
int rango_ent[30];
float rango_real[10];
```

permiten las siguientes llamadas a la función `ordenar()`:

```
ordenar(rango_ent, 30);         //llama a ordenar(int *, int);
ordenar(rango_real, 10);        //llama a ordenar(float *, int);
```

Estas llamadas provocan una instanciación de la función `ordenar` para el tipo concreto de la llamada que, a su vez, realiza una instanciación de la función `intercambio`.

5.2.5. Problemas en las funciones plantilla

Cuando se declaran plantillas de funciones con más de un parámetro, para evitar errores es preciso tener mucha precaución. Si se considera la función `maxi` definida anteriormente:

```
template <class T> T maxi(T a, T b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

Las posibles llamadas a la función necesariamente deben tener el mismo tipo de elemento como parámetro.

```
void main()
{
  char c1 = 'j' , c2 = '1';
  int n1 = 25, n2 = 65;
  long n3 = 50000;
  float n4 = 84.25, n5 = 9.999;

  maxi(c1, c2);               // correcto
  maxi(n1, n2);               // correcto
  maxi(n4, n5);               // correcto
  maxi(c1, n1);               // error c1 y n1 tienen tipos distintos
  maxi(n3, n4);               // error n3 y n4 tienen tipos distintos
  maxi(n2, n3);               // error n2 y n3 tienen tipos distintos
}
```

Obsérverse que el error se produce porque `a` *y* `b` *son del mismo tipo de dato, mientras que en las llamadas que producen error el tipo no es indéntico; por ejemplo,* `c1`, `n1`.

Cuando una llamada a la función se realiza con dos parámetros actuales de distinto tipo se produce un error de compilación, ya que no hay conversión implícita de tipos. Ahora bien, en la práctica este error es muy sencillo de evitar. Basta con hacer una conversión explícita de tipos en la llamada. Así, por ejemplo, para evitar el error de compilación en la llamada `maxi(n3, n4)`; basta con convertir el tipo de dato el primer parámetro a `float`. Es decir, la siguiente llamada no produce ningún tipo de error en la compilación.

```
maxi((float)n3, n4);
```

5.3. PLANTILLAS DE CLASES

Las *plantillas de clase* permiten definir clases genéricas que pueden manipular diferentes tipos de datos. Una aplicación importante es la implementación de *contenedores*, clases que contienen objetos de un tipo dato, tales como vectores (*arrays*), listas, secuencias ordenadas, tablas de dispersión (*hash*).

Así, es posible utilizar una clase plantilla para crear una pila genérica, por ejemplo, que se puede instanciar para diversos tipos de datos predefinidos y definidos por el usuario. Puede tener también clases plantillas para colas, vectores (*arrays*), matrices, listas, árboles, tablas (*hash*), grafos y cualquier otra estructura de datos de propósito general.

Las plantillas de clases se utilizan en toda la biblioteca estándar para contenedores (`list<>`, `map<>`, etc), números complejos (`complex<>`) e incluso con cadenas (`basic_string`) o con operaciones de E/S (`basic_istream`), etc.

5.3.1. Definición de una plantilla de clase

Al igual que en las plantillas de funciones, se escribe una plantilla de clase y, a continuación, el compilador generará el código real cuando se está utilizando la plantilla por primera vez. Incluso la sintaxis es la misma:

```
template <class nombretipo>
class tipop
{
    //...
};
```

donde *`nombretipo`* es el nombre del tipo definido por el usuario utilizado por la plantilla —tipo genérico, `T`— y *`tipop`* es el nombre del tipo parametrizado para la plantilla, es decir, `tipop` es su *clase genérica*. **T** no está limitado a clases o tipos de datos definidos por el usuario y puede tomar incluso el valor de tipos de datos aritméticos (`integer`, `char`, `float`, etc.).

Ejemplos

1.
```
template <typename T>
class Punto3D
{
    T x, y, z;
};
```

2.
```
template <class T>
class Vector
{
private:
    T* buf;

public:
    Vector();
    void acumular(const T& a);
    T suma();
};
```

Una plantilla de clases comienza con la palabra reservada `template` seguida por una lista de parámetros de la plantilla. Con la excepción de la lista de parámetros de la plantilla, la definición de una plantilla de clase es similar a cualquier otra clase. Una plantilla de clases puede definir datos, funciones y miembros tipo; puede utilizar etiquetas de visibilidad para

controlar el acceso a los miembros; define constructores y destructores, etc. En la definición de la clase y sus miembros, se puede utilizar los parámetros de la plantilla como tipos incorporados.

Sintaxis

```
template <typename T> class NomClase
{
    // ...
};
```

Alternativamente:

```
template <class T> class NomClase
{
    // ...
};
```

De acuerdo a la sintaxis propuesta, la plantilla para una clase genérica `Pila` cuyo número máximo de elementos sea `50` se puede escribir así:

```
// archivo PILAGEN.H
// pila genérica de 50 elementos como máximo

template <class T>
class Pila
{
  T datos[50];
  int elementos;
public:
     // constructor de la pila vacia
  Pila(): elementos(0) {}
     // añadir un elemento a la pila
  void Meter(T elem)
  {
    if (elementos < 50)
    {
      datos[elementos] = elem;
      elementos++;
    }
    else
      cout << " errror pila llena";
  }

     // obtener y borrar un elemento de la pila
  T sacar()
  { if (elementos > 0)
    {
      elementos--;
      return datos[elementos];
    }
```

```
    else cout << " error pila vacia";
  }
     // número de elementos reales en la pila
  int Numero(){ return elementos;}

     // ¿está la pila vacía?
  bool vacia(){ return elementos == 0;}
};
```

El prefijo `template <class T>`, o alternativamente `template <typename T>`, en la declaración de clases indica que se declara una plantilla de clase y que se utilizará `T` como el tipo genérico. Por consiguiente, `Pila` es una clase parametrizada con el tipo `T` como parámetro.

Nota

El ámbito del argumento plantilla `T` es todo el cuerpo de la clase genérica.

Con esta definición de la plantilla de clases `Pila` se pueden crear pilas de diferentes tipos de datos, tales como:

```
Pila <int> pila_ent;            // Una pila para variables int
Pila <float> pila_real;         // Una pila para variables float
```

Consideremos la siguiente especificación de la clase plantilla `cola`, que contiene dos parámetros:

```
template <class elem, int tamano> class cola
{
  int tam;
     ...
public:
    cola();
    cola(int n);
    bool esvacia();
    ...
};
```

La clase plantilla `cola` tiene dos parámetros plantilla: una variable de tipo `elem`, que especifica el tipo de los elementos de la cola, y `tamaño`, que especifica el tamaño de la cola.

Al igual que con las plantillas de funciones, se pueden instanciar las plantillas de clases. Una *clase plantilla* es una clase construida a partir de una plantilla de clases. La plantilla de clases ha de ser instanciada para manipular los objetos del tipo adecuado. Es decir, cuando el compilador se encuentra especificado del tipo plantilla, tal como:

```
cola <int, 2048> a;
```

la primera vez toma los argumentos dados para la plantilla y construye una definición de clase automáticamente (Figura 5.1).

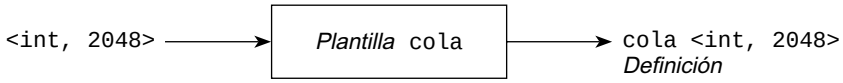


Figura 5.1. Instanciación de una plantilla.

Algunas definiciones de objetos que ilustran el uso de la clase plantilla `cola`:

```
cola <int, 2048> a;
cola <char, 512> b;
cola <char, 1024> c;
cola <char, 512*2> d;
```

Dos nombres de clase plantillas se refieren a las mismas clases, sólo si los nombres de plantillas son idénticos y sus argumentos tienen valores idénticos. En consecuencia, sólo los objetos `c` y `d` tienen los mismos tipos.

La implementación de una clase plantilla requerirá unas funciones constructor, destructor y miembros.

Así, una definición de un constructor de plantilla tiene el formato:

```
template <declaraciones-parámetro-plantilla>
         nombre-clase> <parámetros-plantilla> :: nombre_clase
{
  // ...
}
```

El cuerpo del constructor de la plantilla `cola`:

```
template <class elem, int tamano>
  cola <elem, tamano> :: cola (int n)
{
  tam = n;
}
```

Las definiciones de destructores son similares a las definicones de los constructores.

Una definición de una función miembro de una plantilla de la clase cola tiene el formato siguiente:

```
template <declaraciones-parámetros-plantilla> tipo-resultado
         nombre-clase <parámetros-plantilla> ::
         nombre-func-miembro(declaraciones-parámetros)
{
  // ...
}
```

Como ejemplo de la sintaxis anterior se puede definir la función `vacía` de la clase plantilla `cola`:

```
template <class elem, int tamaño> bool cola <elem, tamaño>::esvacia()
{
  return tam == 0;
}
```

EJEMPLO 5.4. La clase genér ica `Punto` tiene dos datos miembro de tipo genér ico T. Se declara la clase genér ica y, a continuación, se definen las oper aciones.

```
template <typename T> class Punto
{
private:
    T x, y;
public:
    Punto(T x1 = 0, T y1 = 0): x(x1),y(y1){;}
    T getX() const;
    T getY() const;
    void setX(T x1);
    void setY(T y1);
    void mostrar ();
};

template <typename T>
T Punto<T> :: getX() const
{
  return x;
}
template <typename T>
T Punto<T> :: getY() const
{
  return y;
}
template <typename T>
void Punto<T> :: setX(T x1)
{
  x = x1;
}
template <typename T>
void Punto<T> :: setY(T y1)
{
  y = y1;
}
template <typename T>
void Punto<T> :: mostrar ()
{
  cout << "(" << x << ", " << y << ")" << endl;
}
```

5.3.2. Utilización de una plantilla de clase

La manipulación de una plantilla de clase requiere tres etapas:

- Declaración del tipo parametrizado (por ejemplo, `Pila`).
- Implementación de la pila.
- Creación de una instancia específica de pila (por ejemplo, datos de tipo entero `-int-` o carácter `-char-`).

Así, por ejemplo, supongamos que se desea crear un tipo parametrizado `Pila` con las funciones miembro `poner` y `quitar`.

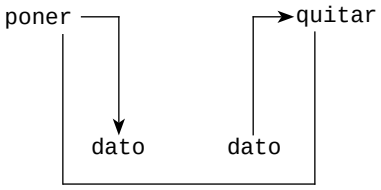


Figura 5.2. Estructura de datos `Pila`.

Declaración de la plantilla Pila con un número variable de elementos

```
template <class T, int nE1 = 100>
class Pila
{
  T datos[nE1];
  int nElementos;
public:
    // constructor
  Pila(): nElementos(0){};
    // añade un elemento a la pila
  void Poner (T elem);
    // obtener y borrar un elemento de la pila
  T Quitar();
    //obtener el numero de elementos
  int num_elementos();
    //¿está la pila vacía?
  bool Vacia();
    // ¿está la pila llena?
  bool Llena();
};
```

Implementación de la pila

```
template <class T, int nE1>
void Pila <T, nE1> :: Poner(T elem)
{
  if (nElementos > nE1)
  {
    datos[nElementos] = elem;
    nElementos++;
  }
  else
    cout << " pila llena ";
}

template <class T, int nE1>
T Pila <T, nE1> :: Quitar()
{
  if (nElementos > 0)
  {
    nElementos--;
    return datos[nElementos];
  }
```

```
  else cout << " error está vacia "<< endl;
}
template <class T, int nE1>
int Pila <T, nE1> :: num_elementos()
{
  return nElementos ;
}
template <class T, int nE1>
bool Pila <T, nE1> :: Vacia()
{
  return (nElementos == 0);
}
template <class T, int nE1>
bool Pila <T, nE1> :: Llena()
{
  return (nElementos == nE1);
}
```

Instanciación de la plantilla de clases

Una instancia de una pila específica se puede crear mediante:

```
Pila <int> pila_ent;            // pila de 100 enteros.
Pila <char> pila_car;           // pila de 100 caracteres.
Pila <double, 5> miniPila;      // pila de 5 reales dobles.
Pila <float, 1000> maxiPila ;   // pila de 1000 reales.
```

Nota

Mediante un tipo parametrizado no se puede utilizar una pila que se componga de tipos diferentes. Objetos polimórficos pueden realizar el ef ecto de tener tipos dif erentes de objetos procesados a la vez, aunque no siempre esto es lo que se requiere .

```
int main()
{
  // pila de enteros
  Pila <int, 6> p1;
  p1.Poner(6);
  p1.Poner(12);
  p1.Poner(18);

  cout <<"Número de elementos :" << p1.num_elementos() << endl;
  cout <<"Quitar 1 :" << p1.Quitar() << endl;
  cout <<"Quitar 2 :" << p1.Quitar() << endl;
  cout <<"Quitar 3 :" << p1.Quitar() << endl;
  cout <<"Número de elementos :" << p1.num_elementos() << endl;

  // Pila de enteros largos
  Pila <long,6> p2;

  p2.Poner(60000L);
  p2.Poner(1000000L);
  p2.Poner(2000000L);
```

```
  cout << "Número de elementos :" << p2.num_elementos() << endl;
  cout << "Quitar 1 :" << p2.Quitar() << endl;
  cout << "Quitar 2 :" << p2.Quitar() << endl;
  cout << "Quitar 3 :" << p2.Quitar() << endl;
  cout << "Número de elementos :" << p2.num_elementos() << endl;

  Pila <double,6> p3;

  p3.Poner(6.6);
  p3.Poner(12.12);
  p3.Poner(18.18);

  cout << "Número de elementos :" << p3.num_elementos() << endl;
  cout << "Quitar 1 :" << p3.Quitar() << endl;
  cout << "Quitar 2 :" << p3.Quitar() << endl;
  cout << "Quitar 3 :" << p3.Quitar() << endl;
  cout << "Número de elementos :" << p3.num_elementos()<< endl;
}
```

El resultado de ejecución del programa anterior es:

```
Número de elementos: 3
Quitar 1: 18
Quitar 2: 12
Quitar3: 6
Números de elementos: 0

Número de elementos: 3
Quitar 1: 2000000
Quitar 2: 1000000
Quitar 3: 60000
Número de elementos: 0

Número de elementos: 3
Quitar 1: 18.18
Quitar 2: 12.12
Quitar 3: 6.6
```

EJEMPLO 5.5. Un programa utiliza la clase genérica `Punto` para crear un punto de tipo `int` y otro de tipo `double`.

```
#include <iostream>
using namespace std;
#include "Punto.h"

int main(int argc, char *argv[])
{
    double x1, y1;
    int x2, y2;
    Punto <double> p1;    // constructor por defecto
    Punto <int> p2(8, -9);
    cout << "Coordenadas del punto 1: ";
    cin >> x1 >> y1;
    p1.setX(x1); p1.setY(y1);
```

```
    cout << "Coordenadas del punto 2: ";
    cin >> x2 >> y2;
    p2.setX(x2); p2.setY(y2);
    cout << " p1, de tipo double: "; p1.mostrar();
    cout << " p2, de tipo int: "; p2.mostrar();
    return 0;
}
```

5.3.3. Argumentos de plantillas

Los argumentos de plantilla no se restringen a tipos, aunque éste sea uno predominante. Los parámetros de una plantilla pueden ser cadenas de caracteres, nombres de funciones y expresiones de constantes. Un caso interesante es el uso de una constante entera para definir el «tamaño» de una estructura de datos de tipo genérico. Por ejemplo, el siguiente código declara un conjunto genérico de `n` elementos:

```
template <class T, int n>
class Conjunto
{
  T datos[n];
  // ...
};
```

Este argumento constante puede incluso tener un valor por defecto, tal como argumentos normales de funciones. La regla de compatibilidad de tipos entre instancias de argumentos de plantilla permanece igual: dos instancias son compatibles si sus argumentos tipo son iguales y sus argumentos expresiones tienen el mismo valor. Esta regla significa que las declaraciones siguientes definen dos objetos compatibles:

```
Conjunto <char, 100> c1;
Conjunto <char, 25*4> c2;
```

El argumento constante de una plantilla debe ser conocido en tiempo de compilación, en definitiva al crear un objeto de esa plantilla el argumento debe ser una constante. Por ejemplo:

```
template <class T, int n, int m>
class Matriz
{
  T mat[n][m];
  // ...
};
```

Esta regla significa que las declaración siguiente para definir un objeto matriz es errónea:

```
int filas, columnas;
cout << "Número de filas/columnas: ";
cin >> filas >> columnas;
Matriz<double, filas, columnas> m1;
```

El error es debido a que filas y columnas no son constantes. Esta otra declaración define una matriz con los valores constantes 3 y 5:

```
Matriz <double, 3, 5> m1;
```

Otra regla de los argumentos plantilla de una función, es que todos deben afectar al tipo de, al menos, uno de los argumentos de las funciones generadas a partir de la plantilla de función. Por ejemplo, el siguiente prototipo de función es correcto:

```
template <class T1, class T2> void convierte(T1 x, T2 y);
```

Sin embargo, el siguiente prototipo no es correcto ya que el argumento constante `n` no afecta a los argumentos de la función:

```
template <class T, int n > void datos (T v[]);
```

5.3.4. Declaración friend en plantillas

La declaración de *amistad*, `friend`, también es posible en el contesto de clases genéricas. Una función global declarada `friend` de una clase genérica puede acceder a los miembros privados o protegidos de cualquier instancia de esa clase genérica. Por ejemplo, la función `getDatos()` puede acceder a cualquier tipo de miembro de la clase genérica `Buffer`:

```
extern void getDatos();

template <class T, int n>
class Buffer
{
  friend void getDatos();
    // ...
```

Una clase genérica se puede declarar *amiga* de otra clase genérica para el mismo argumento plantilla. Por ejemplo, la clase genérica `Conjunto` se declara `friend` de la clase genérica `Elemento`.

```
template <class T>
class Elemento
{
  friend class Conjunto<T>;
private:
    T elemento;
  // ...

template <class T>
class Conjunto
{
public:
    int cardinal();
    bool pertenece(T elm);
    Conjunto<T> union (Conjunto<T> & u2);
    // ...
```

Con esta declaración las operaciones de cualquier tipo de `Conjunto` acceden al elemento privado de la clase `Elemento`.

5.4. MODELOS DE COMPILACIÓN DE PLANTILLAS[2]

Cuando el compilador ve una definición de plantilla, no se genera código inmediatamente. El compilador produce instancias específicas de tipos de la plantilla sólo cuando vea una llamada de la plantilla, tal como cuando se llama una plantilla de función, o un objeto de una plantilla de clase.

Normalmente, cuando se invoca a una función, el compilador necesita ver sólo una declaración de esa función. De modo similar, cuando se define un objeto de un tipo clase, la definición de la clase debe estar disponible, pero las definiciones de las funciones miembro no necesitan estar presentes. Como resultado se ponen las declaraciones de la función y las definiciones de la clase en archivos cabecera y las definiciones de funciones ordinarias y miembros de la clase en archivos fuente.

Las plantillas son diferentes [Lippman, 2005]. Para generar una *instantación* el compilador debe tener que acceder al código fuente que define la plantilla. Cuando se llama a una plantilla de función o una función miembro de una plantilla de clase, el compilador necesita la definición de la función; se necesita el código fuente que está en los archivos fuente.

C++ estándar define dos modelos para compilación del código de las plantillas [Lippman, 2005]. Ambos modelos estructuran los programas de un modo similar: las definiciones de las clases y las declaraciones de las funciones van en archivos de cabeza y las definiciones de miembros y funciones van en archivos fuente. Los dos modelos difieren en el modo que las definiciones de los archivos fuentes se ponen disponibles al compilador. Todos los compiladores soportan el modelo denominado «inclusión» y sólo algunos compiladores soportan el modo de «compilación separada».

5.4.1. Modelo de compilación de inclusión

En el modelo de inclusión, el compilador debe ver la definición de cualquier plantilla que se utilice. La solución que se adopta es incluir en el archivo de cabecera no sólo las declaraciones, sino también las definiciones. Esta estrategia permite mantener la separación de los archivos de cabecera y los archivos de implementación, aunque se incluye una directiva `#include` en el archivo de cabecera para que inserte las definiciones del archivo `.ccp`.

EJEMPLO 5.6. Escribir los archivos necesarios para implementar una clase

Normalmente, las definiciones se hacen disponibles añadiendo una directiva `#include` a las cabeceras que declaran las plantillas de clases o de función. Esta `#include` lleva los archivos fuente que contienen las definiciones variables.

```
//archivo de cabecera demo.h
#ifndef DEMO_H
```

[2] Lippman, Lajoie y Moo realizan una excelente explicación de programación genérica y plantillas en [Lippman, 2005] que recomendamos al lector. [Lippman, 2005]. C++ *Primer*, 4.ª ed. Addison-Wesley, 2005.

```
#define DEMO_H
template<class T>
int comparar(const T&, const T&);
                                    //otras declaraciones
#include "demo.cpp"                 //definiciones de comparar
#endif

//implementación del archivo demo.cpp

template<class T> int comparar(const T &a, const T &b)
{
  if (a < b) return -1;
  if (b < a) return 1;
  return 0;
}
//otras definiciones.
```

Este método permite mantener la separación de los archivos de cabecera y los archivos de implementación pero asegura que el compilador verá ambos archivos cuando se compila el código que utiliza la plantilla.

Algunos compiladores que utilizan el modelo de inclusión pueden generar instanciaciones múltiples. Si dos o más archivos fuente compilados por separado utilizan la misma plantilla, estos compiladores generan una instanciación para la plantilla en cada archivo. Esto supondrá que una plantilla se puede instanciar más de una vez.

5.4.2. Modelo de compilación separada

Este modelo de compilación es muy parecido al modelo típico de C++ y permite escribir las declaraciones y funciones en dos archivos (extensiones `.h` y `.cpp`). La única condición es que se debe utilizar la palabra reservada `export` para conseguir la compilación separada de definiciones de plantillas y declaraciones de funciones de plantillas. Sin embargo, este modelo de compilación no está implementada en la mayoría de los compiladores.

La palabra reservada `export` indica que una definición dada puede ser necesaria para generar instanciaciones en otros archivos. Una plantilla puede ser definida como exportada solamente una vez en un programa. El compilador decide cómo localizar la definición de la plantilla cuando se necesiten generar estas implantaciones. La palabra reservada `export` no necesita aparecer en la declaración de la plantilla. Normalmente, se indica que una plantilla de funciones sea *exportada* como parte de su definición. Se debe incluir la palabra reservada `export` antes de la palabra reservada `template`.

EJEMPLO 5.7. La declaración de la plantilla de función se pone en un archiv o de cabecera, pero la declaración no debe especificar `export`.

```
  //definición de la plantilla en un archivo compilado por separado.
export template<typename T>
T suma(T t1, T t2);
```

El uso de `export` en una plantilla de clase es un poco más complicado. Como es normal, la declaración de la clase debe ir en un archivo de cabecera. El cuerpo de la clase en la cabe-

cera no utiliza la palabra reservada `export`. Si se utiliza `export` en la cabecera, entonces esa cabecera se utilizará en un único archivo fuente del programa.

```
  //cabecera de la plantilla de clase está en el archivo.
  //de cabecera compartido.
template <class T> class Pila {...};
  //Archivo pila.ccp declara Pila como exportada.
export template <class T> class Pila;
#include "Pila.cpp"
  //definiciones de funciones miembro de Pila.
```

Los miembros de una clase exportada se declaran automáticamente como exportados; también se pueden declarar miembros individuales de una plantilla de clase como exportados. En este caso, la palabra `export` no se especifica en la plantilla de clase; sólo se especifica en las definiciones de los miembros específicos que se exportan. La definición se debe situar dentro del archivo de cabecera que define la plantilla de clase.

Nota

La compilación separada es muy interesante pero no es fácil su implementación. Por otra parte, esta característica sólo está implementada en las últimas generaciones de compiladores y pudiera suceder que su propio compilador no la incorpore.

RESUMEN

En el mundo real, cuando se define una clase o función, se puede desear poder utilizarla con objetos de tipos diferentes, sin tener que reescribir el código varias veces. Las últimas versiones de C++ incorporan las plantillas (*templates*) que permiten declarar una clase sin especificar el tipo de uno o más miembros datos (esta operación se puede retardar hasta que un objeto de esa clase se define realmente). De modo similar, se puede definir una función sin especificar el tipo de uno o más parámetros hasta que la función se llama.

Para declarar una familia completa de clases o funciones se puede utilizar la cláusula `template`. Con plantillas sólo se necesita seleccionar la clase característica de la familia.

Las plantillas proporcionan la implementación de tipos parametrizado o genéricos. La *genericidad* es una construcción muy importante en lenguajes de programación orientados a objetos. Una definición muy acertada se debe a Meyer:

«Genericidad es la capacidad de definir módulos parametrizados. Tal módulo se denomina módulo genérico, no es útil directamente; en su lugar, es un patrón de módulos. En la mayoría de los casos, los parámetros representan tipos. Los módulos reales denominados instancias del módulo genérico se obtienen proporcionando tipos reales para cada uno de los parámetros genéricos» [MEYER 88].

El propósito de la genericidad es definir una clase (o una función) sin especificar el tipo de uno o más de sus miembros (parámetros).

Las plantillas permiten especificar un rango de funciones relacionadas (sobrecargados), denominadas *funciones plantilla*, o un rango de clases relacionadas, denominadas *clases plantilla*.

Todas las definiciones de plantillas de funciones comienzan con la palabra reservada `template`, seguida por una lista de parámetros formales encerrados entre ángulos (< >); cada parámetro formal debe estar precedido por la palabra reservada `class`, o bien `typename`. Algunos patrones de sintaxis son:

```
1. template <class T>
2. template <class TipoElemento>
3. template <typename T>
4. template <class TipoBorde, class TipoRelleno>
```

Las plantillas de clases proporcionan el medio para describir una clase genéricamente y se instancian con versiones de esta clase genérica de tipo específico. Una plantilla típica tiene el siguiente formato:

```
template <class T>
class Demo
{
  T v;
  ...
public:
  Demo (const T & val):v(val) {}
  ...
};
```

Las definiciones de las clases (*instanciaciones*) se generan cuando se declara un objeto de la clase especificando un tipo específico. Por ejemplo, la declaración:

```
Demo <short> ic;
```

Hace que el compilador genere una declaración de la clase en el que cada ocurrencia del tipo de parámetro `T` en la plantilla se reemplaza por el tipo real `short` en la declaración de la clase. En este caso, el nombre de la clase es `Demo<short>` y no `Demo`.

El objetivo de la genericidad es permitir reutilizar el código comprobado sin tener que copiarlo manualmente. Esta propiedad simplifica la tarea de programación y hace los programas más fiables.

- C++ proporciona las plantillas (`templates`) para proporcionar *genericidad* y polimorfismo paramétrico. El mismo código se utiliza con diferentes tipos, donde el tipo es un parámetro del cuerpo del código.
- Una plantilla de función es un mecanismo para generar una nueva función.
- Una plantilla de clases es un mecanismo para la generación de una nueva clase.
- Un parámetro de una plantilla puede ser o bien un tipo o un valor.
- El tipo de los parámetros de la plantilla es una definición de una plantilla de función se puede utilizar para especificar el tipo de retorno y los tipos de parámetros de la función generada.
- C++ estándar describe una biblioteca estándar de plantilla que, en parte, incluye versiones de plantillas sobre tareas típicas de computación tales como búsqueda y ordenación. En el paradigma orientado a objetos, el programa se organiza como un conjunto finito de objetos que contiene datos y operaciones (funciones miembro en C++) que llaman a esos datos y que se comunican entre sí mediante mensajes.

EJERCICIOS

5.1. Definir plantillas de funciones `min()` y `Max()` que calculen el valor mínimo y máximo de un vector de `n` elementos.

5.2. Realizar un programa que utilice las funciones plantilla del ejercicio anterior para calcular los valores máximos de vectores de enteros, de doble precisión (`double`) y de carácter (`char`).

5.3. Escribir una clase plantilla que pueda almacenar una pequeña base de datos de registros.

5.4. Realizar un programa que utilice la plantilla del ejercicio anterior para crear un objeto de la clase de base de datos.

5.5. ¿Cómo se implementan funciones genéricas en C? Compárela con plantillas de funciones.

5.6. Declarar una plantilla para la función `gsort` para ordenar arrays de un tipo dado.

5.7. Definir una función plantilla que devuelva el valor absoluto de cualquier tipo de dato incorporado o predefinido pasado a ella.

CAPÍTULO 6

Análisis y eficiencia de algoritmos

Objetivos

Con el estudio de este capítulo usted podrá:

- Revisar los conceptos básicos de tipos de datos.
- Introducirse en las ideas fundamentales de estructuras de datos.
- Revisar el concepto de algoritmo y programa.
- Conocer y entender la utilización de la herramienta de programación conocida por "pseudocódigo".
- Entender los conceptos de análisis, verificación y eficiencia de un algoritmo.
- Conocer las propiedades matemáticas de la notación O.
- Conocer la complejidad de las sentencias básicas de todo programa C++.

Contenido

Conceptos clave

- Algoritmo.
- Complejidad.
- Eficiencia.
- Notación asintótica.
- Pseudocódigo.
- Rendimiento de un programa.

INTRODUCCIÓN

La representación de la información es fundamental en ciencias de la computación y en informática. El propósito principal de la mayoría de los programas de computadoras es almacenar y recuperar información, además de realizar cálculos. De modo práctico, los requisitos de almacenamiento y tiempo de ejecución exigen que tales programas deban organizar su información de un modo que soporte procesamiento eficiente. Por estas razones, el estudio de estructuras de datos y los algoritmos que las manipulan constituyen el núcleo central de la *informática* y de la computación.

En el capítulo se revisan los conceptos básicos de dato, abstracción, algoritmos y programas, así como los criterios relativos a análisis y eficiencia de algoritmos.

6.1. ALGORITMOS Y PROGRAMAS

Un **algoritmo** es un método, proceso conjunto de instrucciones utilizadas para resolver un problema específico. Un problema puede ser resuelto mediante muchos algoritmos. Un algoritmo dado correcto resuelve un problema definido y determinado (por ejemplo, calcula una función determinada). En este libro se explican muchos algoritmos y para algunos problemas se proponen diferentes algoritmos, como es el caso del problema típico de ordenación de listas.

La ventaja de conocer varias soluciones a un problema es que las diferentes soluciones pueden ser más eficientes para variaciones específicas del problema, o para diferentes entradas del mismo problema. Por ejemplo, un algoritmo de ordenación puede ser el mejor, para ordenar conjuntos pequeños de números, otro puede ser el mejor para ordenar conjuntos grandes de números, y un tercero puede ser el mejor para ordenar cadenas de caracteres de longitud variable.

Desde un punto de vista más formal y riguroso, un algoritmo es "un conjunto ordenado de pasos o instrucciones ejecutables y no ambiguas para resolver un determinado problema". Las etapas o pasos que sigue el algoritmo deben tener una estructura bien establecida en términos del orden en que se ejecutan. Esto no significa que las etapas se deban ejecutar en secuencia, una primera etapa, después una segunda, etc. Algunos algoritmos conocidos como *algoritmos paralelos*, por ejemplo, contienen más de una secuencia de etapas, cada una diseñada para ser ejecutada por procesadores diferentes en una máquina multiprocesador. En tales casos, los algoritmos globales no poseen un único hilo conductor de etapas que conforman el escenario de primera etapa, segunda etapa, etc. En su lugar, la estructura del algoritmo es el de múltiples hilos conductores que bifurcan y reconectan a medida que los diferentes procesadores ejecutan las diferentes partes de la tarea global.

Durante el diseño de un algoritmo, los detalles de un lenguaje de programación específico se pueden obviar frente a la simplicidad de una solución. Generalmente, el diseño se escribe en español (o en inglés, o en otro idioma hablado). Se utiliza un tipo de lenguaje mixto entre el español y un lenguaje de programación universal. Esta mezcla se conoce como pseudocódigo *(*o *seudocódigo).*

6.1.1. Propiedades de los algoritmos

Un algoritmo debe cumplir diferentes propiedades[1]:

1. *Especificación precisa de la entrada.* La forma más común del algoritmos es una transformación que toma un conjunto de valores de entrada y ejecuta algunas manipulacio-

[1] En [BUDD 98], [JOYANES 03], [BROOKSHEAR 03] y [TREMBLAY 03] más las referencias incluidas al final del libro en la bibliografía, puede encontrar ampliación sobre teoría y práctica de algoritmos.

nes par producir un conjunto de valores de salida. Un algoritmo debe dejar claros el número y tipo de valores de entrada y las condiciones iniciales que deben cumplir esos valores de entrada para conseguir que las operaciones tengan éxito.

2. *Especificación precisa de cada instrucción.* Cada etapa de un algoritmo debe ser definida con precisión. Esto significa que no puede haber *ambigüedad* sobre las acciones que se deban ejecutar en cada momento.
3. *Exactitud, Corrección.* Un algoritmo debe ser *exacto, correcto.* Se debe poder demostrar que el algoritmo resuelve el problema. Con frecuencia, esto se plasma en el formato de un argumento, lógico o matemático, al efecto de que si las condiciones de entrada se cumplen y se ejecutan los pasos del algoritmos entonces se producirá la salida deseada. En otras palabras, se debe calcular la función deseada, convirtiendo cada entrada a la salida correcta. Un algoritmo se espera resuelva un problema.
4. *Etapas bien definidas y concretas.* Un algoritmo se compone de una serie de *etapas concretas.* Concreta significa que la acción descrita por esa etapa está totalmente comprendida por la persona o máquina que debe ejecutar el algoritmo. Cada etapa debe ser ejecutable en una cantidad finita de tiempo. Por consiguiente, el algoritmos nos proporciona una "receta" para resolver el problema en etapas y tiempos concretos.
5. *Número finito de pasos.* Un algoritmo se debe componer de un número *finito* de pasos. Si la descripción del algoritmo se compone de un número infinito de etapas, nunca se podrá implementar como un programa de computador. La mayoría de los lenguajes que describen algoritmos (español, inglés o pseudocógio) proporcionan un método para ejecutar acciones repetidas, conocidas como iteraciones que controlan las salidas de bucles o secuencias repetitivas.
6. *Un algoritmo debe terminar.* En otras palabras, no puede entrar en un bucle infinito.
7. *Descripción del resultado o efecto.* Por último, debe estar claro cuál es la tarea que el algoritmo debe ejecutar. La mayoría de las veces, esta condición se expresa con la producción de un valor como resultado que tenga ciertas propiedades. Con menor frecuencia, los algoritmos se ejecutan para un *efecto lateral*, tal como imprimir un valor en un dispositivo de salida. En cualquier caso la salida esperada debe estar especificada completamente.

EJEMPLO 6.1. ¿Es un algoritmo la instrucción siguiente?

```
Escribir una lista de todos los enteros positivos
```

Es imposible ejecutar la instrucción anterior dado que hay infinitos enteros positivos. Por consiguiente, cualquier conjunto de instrucciones que implique esta instrucción no es un algoritmo.

6.1.2. Programas

Normalmente, se considera que un programa de computadora es una representación concreta de un algoritmo en un lenguaje de programación. Naturalmente, hay muchos programas que son ejemplos del mismo algoritmo, dado que cualquier lenguaje de programación moderno se puede utilizar para implementar cualquier algoritmo (aunque algunos lenguajes facilitarán su tarea al programador mejor que otros). Por definición, un algoritmo debe

proporcionar suficiente detalle para que se pueda convertir en un programa cuando se necesite.

Para recordar

1. Un problema es una función o asociación de entradas con salidas.
2. Un algoritmo es una receta para resolver un problema cuyas etapas son concretas y no ambiguas.
3. El algoritmo debe ser correcto, finito y debe terminar para todas las entradas.
4. Un programa es una "ejecución" (instanciación) de un algoritmo en un lenguaje de programación de computadora.

El diseño de un algoritmo para ser implementado por un programa de computadora debe tener dos características principales:

1. Que sea fácil de entender, codificar y depurar.
2. Que consiga la mayor eficiencia a los recursos de la computadora.

Idealmente, el programa resultante debería ser el más eficiente. ¿Cómo medir la eficiencia de un algoritmo o programa? El método correspondiente se denomina *análisis de algoritmos* y permite medir la dificultad inherente a un problema. En este capítulo se desarrollará el concepto y la forma de medir la medida la eficiencia.

6.2. EFICIENCIA Y EXACTITUD

De las características, antes analizadas, que deben cumplir los algoritmos destacan dos por su importancia en el desarrollo de algoritmos y en la construcción de programas: eficiencia y exactitud que se examinarán y utilizarán amplia y profusamente en los siguientes capítulos.

Existen numerosos enfoques a la hora de resolver un problema. ¿Cómo elegir el más adecuado entre ellos? Entre las líneas de acción fundamentales en el diseño de computadoras se suelen plantear dos objetivos (a veces conflictivos y contradictorios entre sí) [SHAFFER 97]:

1. Diseñar un algoritmo que sea fácil de entender, codificar y depurar.
2. Diseñar un algoritmo que haga un uso eficiente de los recursos de la computadora.

Idealmente, el programa resultante debe cumplir ambos objetivos. En estos casos, se suele decir que tal programa es "elegante". Entre los objetivos centrales de este libro está la medida de la eficiencia de algoritmo, así como su diseño correcto o exacto.

6.2.1. Eficiencia de un algoritmo

Raramente existe un único algoritmo para resolver un problema determinado. Cuando se comparan dos algoritmos diferentes que resuelven el mismo problema, normalmente, se encontrará que un algoritmo es un orden de magnitud más eficiente que el otro. En este sentido lo importante, es que el programador sea capaz de reconocer y elegir el algoritmo más eficiente.

¿Entonces, qué es eficiencia? La ***eficiencia*** de un algoritmo es la propiedad mediante la cual un algoritmo debe alcanzar la solución al problema en el tiempo más corto posible y/o utilizando la cantidad más pequeña posible de recursos físicos, y que sea compatible con su exactitud o corrección. Un buen programador buscará el algoritmo más eficiente dentro del conjunto de aquellos que resuelven con exactitud un problema dado.

¿Cómo medir la eficiencia de un algoritmo o programa informático? Uno de los métodos más sobresalientes es el ***análisis de algoritmos***. El análisis de algoritmos permite medir la dificultad inherente de un problema. Los restantes capítulos utilizan con frecuencia las técnica de análisis de algoritmos siempre que éstos se diseñan. Esta característica le permitirá comparar algoritmos para la resolución de problemas en términos de eficiencia.

Aunque las máquinas actuales son capaces de ejecutar millones de instrucciones por segundo, la eficiencia permanece como un reto o preocupación a resolver. Con frecuencia, la elección entre algoritmos eficientes e ineficientes pueden mostrar la diferencia entre una solución práctica a un problema y una no práctica. En los primeros tiempos de la informática moderna (décadas sesenta a ochenta) las computadoras eran muy lentas y tenían pequeña capacidad de memoria. Los programas tenían que ser diseñados cuidadosamente para hacer uso de los recursos escasos, tales como almacenamiento y tiempo de ejecución. Los programadores gastaban horas intentando recortar radicalmente segundos a los tiempos de ejecución de sus programas o intentando comprimir los programas en un pequeño espacio en memoria utilizando todo tipo de tecnologías de comprensión y reducción de tamaño. La eficiencia de un programa se medía en aquella época como un factor dependiente del binomio *espacio-tiempo.*

Hoy, la situación ha cambiado radicalmente. Los costes del hardware han caído drásticamente mientras que los costes humanos han aumentado considerablemente. El tiempo de ejecución y el espacio de memoria ya no son factores críticos como lo fueron anteriormente. Hoy día el esfuerzo considerable que se requería para conseguir la eficiencia máxima, no es tan acusado, excepto en algunas aplicaciones, como por ejemplo, aplicaciones de sistemas en tiempo real con factores críticos de ejecución. Pese a todo, la eficiencia sigue siendo un factor decisivo en el diseño de algoritmos y posterior construcción de programas.

Existen diferentes métodos con los que se trata de medir la eficiencia de los algoritmos; entre ellos, están los que se basan en el número de operaciones que debe efectuar un algoritmo para realizar una tarea; otros métodos se centran en tratar de medir el tiempo que se emplea en llevar a cabo una determinada tarea ya que lo importante al usuario final es que ésta se efectúe de forma correcta y en el menor tiempo posible. Sin embargo, estos métodos presentan varias dificultades ya que cuando se trata de generalizar la medida hecha, ésta depende de factores como la máquina en que se efectuó, el ambiente del procesamiento y el tamaño de la muestra, entre otros factores.

Brassard y *Bratley* acuñaron, en 1988, el término **algorítmica** o **algoritmia** *(algorithmics)*[2] que definía como "el estudio sistemático de las técnicas fundamentales utilizadas para diseñar y analizar algoritmos eficientes". Este estudio fue ampliado posteriormente en 1997[3] con la consideración de que la determinación de la eficiencia de un algoritmo se podía expresar en el tiempo requerido para realizar la tarea en función del tamaño de la muestra e independiente del ambiente en que se efectúe.

El estudio de la eficiencia de los algoritmos se centra, fundamentalmente, en el análisis de la ejecución de bucles ya que en el caso de funciones lineales —no contienen bucles— la efi-

---

[2] Giles Brassard y Paul Bratley. *Algorthmics. Theory and Practice*. Englewood Cliffs, N. Y.: Prentice-Hall, 1988.

[3] *Fundamental of algorithmics* (Prentice-Hall, 1997). Este libro fue traducido al español, y publicado también en Prentice-Hall (España), por un equipo de profesores de la Universidad Pontificia de Salamanca, dirigidos por el co-autor de este libro, profesor Luis Joyanes.

ciencia es función del número de instrucciones que contiene. En este caso, su eficiencia depende de la velocidad de las computadoras y generalmente no es un factor decisivo en la eficiencia global de un programa.

Al crear programas que se ejecutan muchas veces a lo largo de su vida y/o tienen grandes cantidades de datos de entrada, las consideraciones de eficiencia, no se pueden descartar. Además, existen en la actualidad un gran número de aplicaciones informáticas que requieren características especiales de *hardware* y *software* en las que los criterios de eficiencia deben ser siempre tenidas en cuenta.

Por otra parte, las consideraciones espacio-tiempo se han ampliado con los nuevos avances en tecnologías de hardware de computadoras. Las consideraciones de espacio implican hoy diversos tipos de memoria: principal, caché, flash, archivos, discos duros USB, y otros formatos especializados. Asimismo con el uso creciente de redes de computadoras de alta velocidad, existen muchas consideraciones de eficiencia a tener en cuenta en entornos de informática o computación distribuida.

Nota

La eficiencia como factor espacio-tiempo debe estar estrechamente relacionada con la buena calidad, el funcionamiento y la facilidad de mantenimiento del programa.

6.2.2. Eficiencia de bucles

En general, el formato de la eficiencia se puede expresar mediante una función:

$$f(n) = eficiencia$$

Es decir, la eficiencia del algoritmo se examina como una función del número de elementos a ser procesados.

Bucles lineales

En los bucles se repiten las sentencias del *cuerpo del bucle* un número determinado de veces, que determina la eficiencia del mismo. Normalmente, en los algoritmos los bucles son el término dominante en cuanto a la eficiencia del mismo.

EJEMPLO 6.2. ¿Cuántas veces se repite el cuerpo del bucle en el siguiente código?

```
1. i = 1
2. mientras (i <= n)
       código de la aplicación
       i = i + 1
   fin_mientras
```

Si `n` es un entero, por ejemplo de valor `100`, la respuesta es `100` veces. El número de iteraciones es directamente proporcional al factor del bucle, `n`. Como la eficiencia es directamente proporcional al número de iteraciones la función que expresa la eficiencia es:

```
f(n) = n
```

EJEMPLO 6.3. ¿Cuántas veces se repite el cuerpo del bucle en el siguiente código?

```
1. i = 1
2. mientras (i <= n)
       código de la aplicación
       i = i + 2
   fin_ mientras
```

La respuesta no siempre es tan evidente como en el ejercicio anterior. Ahora el contador `i` avanza de `2` en `2`, por lo que la respuesta es `n/2`. En este caso el factor de eficiencia es:

```
f(n) = n / 2
```

Bucles algorítmicos

Consideremos un bucle en el que su variable de control se multiplique o divida dentro de dicho bucle. ¿Cuántas veces se repetirá el cuerpo del bucle en los siguientes segmentos de programa?

```
1. i = 1
2. mientras (i < 1000)
       { código de la aplicación }
       i = i * 2
   fin_mientras
```

```
1.  i = 1000
2.  mientras (i >= 1)
        { código aplicación }
        i = i/2
    fin_mientras
```

La Tabla 6.3 contiene las diferentes iteraciones y los valores de la variable `i`.

Tabla 6.1. Análisis de los bucles de multiplicación y división.

Bucle de multiplicar		**Bucle de dividir**	
Iteración	**Valor de i**	**Iteración**	**Valor de i**
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
salida	1024	*salida*	0

En ambos bucles se ejecutan `10` iteraciones. La razón es que en cada iteración el valor de `i` se dobla en el bucle de multiplicar y se divide por la mitad en el bucle de división. Por

consiguiente, el número de iteraciones es una función del multiplicador o divisor, en este caso 2.

Bucle de multiplicar	$2^{\text{iteraciones}} < 1000$
Bucle de división	$1000/2^{\text{iteraciones}} >= 1$

Generalizando, el análisis se puede decir que las iteraciones de los bucles especificados se determinan por la siguiente formula:

```
f(n) = [log2 n]      //parte entera de login
```

Bucles anidados

El total de iteraciones de bucles anidados (bucles que contienen a otros bucles) se determina multiplicando el número de iteraciones del bucle interno por el número de iteraciones del bucle externo.

```
iteraciones = iteraciones del bucle externo x iteraciones bucle interno
```

Existen tres tipos de bucles anidados: *lineal logarítmico, cuadráticos dependientes y cuadráticos* que con análisis similares a los anteriores nos conducen a ecuaciones de eficiencia contempladas en la Tabla 6.2.

Tabla 6.4. Fórmulas de eficiencia.

Lineal logarítmica	$f(n) = [n\log_2 n]$
Dependiente cuadrática	$f(n) = n\frac{(n+1)}{2}$
Cuadrática	$f(n) = n^2$

6.3.3. Análisis de rendimiento

La medida del rendimiento de un programa se consigue mediante la complejidad del espacio y del tiempo y de un programa.

La *complejidad del espacio* de un programa es la cantidad de memoria que se necesita para ejecutar hasta la compleción (*terminación*). El avance tecnológico proporciona hoy en día memoria abundante, por esa razón el análisis de algoritmos se centra, fundamentalmente, en el tiempo de ejecución, si bien puede estudiarse de forma análoga a la del tiempo.

La *complejidad del tiempo* de un programa es la cantidad de tiempo de computadora que se necesita para ejecutarse. Se utiliza una función, *T(n)*, para representar el número de unidades de tiempo tomadas por un programa o algoritmo para cualquier entrada de tamaño *n*. Si la función *T(n)* de un programa es *T(n) = c * n* entonces el tiempo de ejecución es linealmente proporcional al tamaño de la entrada sobre la que se ejecuta. Tal programa se dice que es de *tiempo lineal* o simplemente *lineal*.

EJEMPLO 6.4. Tiempo de ejecución lineal de una función que calcula una serie de *n* términos.

```
double serie(double x, int n)
{
  double s;
  int i;
  s = 0.0;                        // tiempo t1
  for (i = 1; i <= n; i++)        // tiempo t2
  {
    s += i*x;                     // tiempo t3
  }
  return s;                       // tiempo t4
}
```

La función *T(n)* del método es:

```
T(n) = t1 + n * t2 + n * t3 + t4
```

El tiempo crece a medida que lo hace *n*, por ello es preferible expresar el tiempo de ejecución de tal forma que indique el comportamiento que va a tener la función con respecto al valor de *n*.

Considerando todas las reflexiones anteriores, si *T(n)* es el tiempo de ejecución de un programa con entrada de tamaño *n*, será posible valorar *T(n)* como el número de sentencias ejecutadas por el programa, y la evaluación se podrá efectuar desde diferentes puntos de vista:

Peor caso. Indica el tiempo peor que se puede tener. Este análisis es perfectamente adecuado para algoritmos cuyo tiempo de respuesta sea crítico, por ejemplo, para el caso del programa de control de una central nuclear. Es el que se emplea en este libro.

Mejor caso. Indica el tiempo mejor que podemos tener.

Caso medio. Se puede computar *T(n)* como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entradas de tamaño *n*. El tiempo de ejecución medio es a veces una medida más realista de lo que el rendimiento será en la práctica, pero es, normalmente, mucho más difícil de calcular que el tiempo de ejecución en el caso peor.

6.4. NOTACIÓN *O-GRANDE*

La alta velocidad de las computadoras actuales (frecuencias del procesador de 3 GHz. ya son usuales en computadoras comerciales) hace que la medida exacta de la eficiencia de un algoritmo no sea una preocupación vital en el diseño pero si el orden general de magnitud de la misma. Si el análisis de dos algoritmos muestra que uno ejecuta `25` iteraciones mientras que otro ejecuta `40`, la práctica muestra que ambos son muy rápidos, entonces ¿cómo valorar las diferencias? Ahora bien, si un algoritmo realiza `25` iteraciones y otro `2500` iteraciones, entonces debemos estar preocupados por la rapidez o la lentitud de uno y otro.

Anteriormente, en el Ejemplo 6.4, se ha expresado mediante la fórmula *f(n)* el número de sentencias ejecutadas para *n* datos. El factor dominante que se debe considerar para determinar el orden de magnitud de la fórmula es el denominado factor de eficiencia. Por consiguiente, no se necesita determinar la medida completa de la eficiencia, basta con calcular el factor que

determina la magnitud. Este factor se define como *"O grande"*, que representa *"está en el orden de"* y se expresa como *O(n)* , es decir, *"en el orden de n"*.

La notación *O* indica la cota superior del tiempo de ejecución de un algoritmo o programa. Así, en lugar de decir que un algoritmo emplea un tiempo de *4n-1* en procesar un array de longitud *n*, se dirá que emplea un tiempo *O(n)* que se lee *"O grande de n"*, o bien *"O de n"* y que informalmente significa *"algunos tiempos constantes n"*.

Con la notación O se e xpresa una aproximación de la relación entre el tamaño de un problema y la cantidad de proceso necesar io para hacerlo. Por ejemplo, si:

$f(n) = n^2 - 2n + 3$ *entonces* $f(n)$ *es* $O(n^2)$.

6.4.1. Descripción de tiempos de ejecución con la notación *O*

Sea *T(n)* el tiempo de ejecución de un programa, medido como una función de la entrada de tamaño *n*. Se dice que *"T(n) es O(g(n))"* si g*(n)* acota superiormente a *T(n)*. De modo más riguroso, *T(n)* es *O(g(n))* si existe un entero n_o y una constante $c > 0$ tal que para todos los enteros $n \geq no$ se tiene que $T(n) \leq cg(n)$.

EJEMPLO 6.5. Dada la función $f(n) = n^3 + 3n + 1$ encontr ar su **"O grande"** (complejidad asintótica).

Para valores de $n \geq 1$ se puede demostrar que:

$$f(n) = n^3 + 3n + 1 \leq n^3 + 3n^3 + 1n^3 = 5\ n^3$$

Escogiendo la constante $c = 5$ y $n_0 = 1$ se satisface la desigualdad $f(n) \leq 5\ n^3$. Entonces se puede asegurar que:

$$f(n) = O(n^3)$$

Ahora bien, también se puede asegurar que $f(n) = O(n^4)$ y que $f(n) = O(n^5)$ y así sucesivamente con potencias mayores de n. Sin embargo, lo que realmente interesa es la cota superior mas ajustada que informa de la tasa de crecimiento de la función con respecto a n.

Una función $f(x)$ puede estar acotada super iormente por un número indefinido de funciones a partir de ciertos valores x_0,

$$f(x) \leq g(x) \leq h(x) \leq k(x) \ldots$$

La complejidad asintótica de la función $f(x)$ se considera que es la cota superior más ajustada:

$$f(x) = O(g(x))$$

> ***Nota***
>
> La expresión la eficiencia de un algoritmo se simplifica con la función *O* grande.

Si un algoritmo es cuadrático, se dice entonces que su eficiencia es $O(n^2)$. Esta función expresa cómo crece el tiempo de proceso del algoritmo, de tal forma que una eficiencia $O(n^2)$ muestra que crece con el cuadrado del número de entradas.

6.4.2. Determinar la notación *O*

La notación *O grande* se puede obtener a partir de f(n) utilizando los siguientes pasos:

1. En cada término, establecer el coeficiente del término en 1.
2. Mantener el término mayor de la función y descartar los restantes. Los términos se ordenan de menor a mayor:

$$log_2 n \quad n \quad n log_2 n \quad n^2 \quad n^3 \quad \ldots \quad n^k \quad 2^n \quad n!$$

EJEMPLO 6.6. Calcular la función O grande de eficiencia de las siguientes funciones:

a. $f(n) = n(n+1)/2 = 1/2\ n^2 + 1/2\ n$
b. $f(n) = a_j n^k + a_{j-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$

Caso a

1. Se eliminan todos los coeficientes y se obtiene:

   $n^2 + n$

2. Se eliminan los factores más pequeños:

   n^2

3. La notación O correspondiente es:

   $O(f(n)) = O(n^2)$

Caso b

1. Se eliminan todos los coeficientes y se obtiene:

   $f(n) = n^k + n^{k-1} + \ldots + n^2 + n + 1$

2. Se eliminan los factores más pequeños y el término de exponente mayor es el primero:

   n^k

3. La notación *O* correspondiente es:

   $O(f(n)) = O(n^k)$

6.4.3. Propiedades de la notación O

De la definición conceptual de la notación *O* se deducen las siguientes propiedades de la notación *O*.

1. Siendo *c* una constante, *c*O(f(n)) = O(f(n))*

   Por ejemplo si $f(n) = 3n^4$, entonces $f(n) = 3*O(n^4) = O(n^4)$

2. *O(f(n)) + O(g(n)) = O(f(n)+g(n))*.

   Por ejemplo, si $f(n) = 2e^n$ y $g(n) = 2n^3$:

   $$O(f(n)) + O(g(n)) = O(f(n)+g(n)) = O(2e^n + 2\ n^3) = O(e^n)$$

3. *Maximo(O(f(n)),O(g(n))) = O(Maximo(f(n),g(n))*.

   Por ejemplo,

   $$Maximo(O(\log(n)),O(n)) = O(Maximo(\log(n),n)) = O(n).$$

4. *O(f(n)) * O(g(n)) = O(f(n)*g(n))*.

   Por ejemplo, si $f(n) = 2n^3$ y $g(n) = n$:

   $$O(f(n))* O(g(x)) = O(f(x)*g(x)) = O(2n^3 * n) = O(n^4)$$

5. *O(loga(n)) = O(logb(n)) para a,b > 1*

Las funciones logarítmicas son de orden logarítmico, independientemente de la base del logaritmo.

6. *O(log(n!)) = O(n*log(n))*

7. Para k > 1 $\quad O(\sum_{i=1}^{n} i^k) = O(n^{k+1})$

8. $O(\sum_{i=2}^{n} \log(i)) = O(n\log(n))$

6.5. COMPLEJIDAD DE LAS SENTENCIAS BASICAS DE C++

Al analizar la complejidad de un método no recursivo, se han de aplicar las propiedades de la notación *O* y las siguientes consideraciones relativas al orden que tienen las sentencias, fundamentalmente a las estructuras de control.

- Las sentencias de asignación, son de orden constante *O(1)*.
- La complejidad de una sentencia de selección es el máximo de las complejidades del bloque `then` y del bloque `else`.
- La complejidad de una sentencia de selección múltiple (`switch`) es el máximo de las complejidades de cada uno de los bloques `case`.
- Para calcular la complejidad de un bucle, condicional o automático, se ha de estimar el número máximo de iteraciones para el *peor caso*; entonces la complejidad del bucle es el producto del número de iteraciones por la complejidad de las sentencias que forman el cuerpo del bucle.
- La complejidad de un bloque se calcula como la suma de las complejidades de cada sentencia del bloque.
- La complejidad de la llamada a una función es de orden 1, complejidad constante. Es necesario considerar la complejidad de la función llamada.

EJEMPLO 6.7. Determinar la complejidad de la función:

```
double mayor(double x, double y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

El método consta de una sentencia de selección, cada alternativa tiene complejidad constante, *O(1)*, entonces la complejidad de la función `mayor()`es *O(1)*.

EJEMPLO 6.8. Determinar la complejidad de la siguiente función:

```
void escribeVector(doublex[], int n)
{
  int j;
  for (j = 0; j < n; j++)
  {
    cout << x[j];
  }
}
```

La función consta de un bucle que se ejecuta *n* veces, *O(n)*. El cuerpo del bucle consiste en ejecutar una sentencia: complejidad constante *O(1)*. Como conclusión, la complejidad de la función es: `O(n)*O(1) = O(n)`.

EJEMPLO 6.9. Determinar la complejidad de la función:

```
double suma(double d[], int n)
{
  int k ;
  k = s = 0;
```

```
  while (k < n)
  {
    s += d[k];
    if (k == 0)
        k = 2;
    else
        k *= 2;
  }
  return s;
}
```

`suma()` está formado por una sentencia de asignación múltiple, *O(1)*, un bucle condicional y la sentencia que devuelve control de complejidad constante, *O(1)*. Por consiguiente, la complejidad de la función es la del bucle. Es necesario determinar el numero máximo de iteraciones que va a realizar el bucle y la complejidad del cuerpo del bucle. El número de iteraciones es igual al número de veces que el algoritmo multiplica por dos a la variable `k` . Si *t* es el número de veces que `k` se puede multiplicar hasta alcanzar el valor de `n`, que hace que termine el bucle, entonces $k = 2^t$.

$$0,\ 2,\ 2^2,\ 2^3\ ,\ \ldots\ ,\ 2^t \geq n$$

Tomando logaritmos: $t \geq \log_2 n$; por consiguiente, el máximo de iteraciones es:

$t = \log_2 n$.

El cuerpo del bucle consta de una sentencia simple y un sentencia de selección de complejidad *O(1)*, entonces tiene complejidad constante, *O(1)*. Con todas estas consideraciones, la complejidad del bucle y también de la función es:

$O(\log_2 n)*O(1) = O(\log_2 n)$; *complejidad logarítmica* `O(log n)`

EJEMPLO 6.10. Determinar la complejidad de la función:

```
void traspuesta(float d[][M], int n)
{
  int i, j;
  for (i = n - 2; i > 0; i--)
  {
    for (j = i + 1; j < n; j++)
    {
      float t;
      t = d[i][j];
      d[i][j] = d[j][i];
      d[j][i] = t;
      }
  }
}
```

La función consta de dos bucles `for` anidados. El bucle interno está formado por tres sentencias de complejidad constante, *O(1)*. El bucle externo siempre realiza `n-1` veces el bucle

interno. A su vez, el bucle interno realiza *k* veces su bloque de sentencias, *k* varía de 1 a *n – 1*. De modo que el número total de iteraciones es:

$$C = \sum_{k=1}^{n-1} k$$

El desarrollo del sumatorio produce la expresión:

$$(n-1) + (n-2) + \ldots + 1 = \frac{n * (n - 1)}{2}$$

Aplicando las propiedades de la notación *O* se deduce que la complejidad de la función es $O(n^2)$. El término que domina en el tiempo de ejecución es n^2, se dice que la *complejidad es cuadrática*.

RESUMEN

Una de las herramientas típicas más utilizadas para definir algoritmo es el pseudocódigo. El pseudocódigo es una representación en español (o en inglés, brasilero, etc.) del código requerido para un algoritmo.

La eficiencia de un algoritmo se define generalmente en función del número de elementos a procesar y el tipo de bucle que se va a utilizar. Las eficiencias de los diferentes bucles son:

Bucle lineal:	`f(n) = n`
Bucle logarítmico:	`f(n) = log n`
Bucle logarítmico lineal:	`f(n) = n * log n`
Bucle cuadrático dependiente:	`f(n) = n(n+1)/2`
Bucle cuadrático dependiente:	$f(n) = n^2$
Bucle cúbico:	$f(n) = n^3$

Normalmente, no se necesita determinar la medida completa de la eficiencia, basta con calcular el factor que *domina* tal magnitud. Este factor se define como *"O grande"*, que representa *"está en el orden de"* y se expresa como *O(n)*. La notación *O* indica la cota superior del tiempo de ejecución de un algoritmo o programa. Así, en lugar de decir que un algoritmo emplea un tiempo de *2n + 1* en procesar un array de longitud *n*, se dirá que emplea un tiempo *O(n)* que se lee *"O grande de n"*, o bien *"O de n"*.

EJERCICIOS

6.1. El siguiente algoritmo pretende calcular el cociente entero de dos enteros positivos (un dividendo y un divisor) contando el número de veces que el divisor se puede restar del dividendo antes de que se vuelva de menor valor que el divisor. Por ejemplo, 14/3 proporcionará el resultado 4 ya que 3 se puede restar de 14 cuatro veces. ¿Es correcto?, justifique su respuesta. Calcule su complejidad.

```
Cuenta ← 0;
Resto ← Dividendo;
```

```
repetir
  Resto ← Resto – Divisor
  Cuenta ← Cuenta + 1
hasta_que (Resto < Divisor)
Cociente ← Cuenta
```

6.2. El siguiente algoritmo está diseñado para calcular el producto de dos enteros negativos `x` e `y` por acumulación de la suma de copias de `y` (es decir, 4 por 5 se calcula acumulando la suma de cuatro cinco veces). ¿Es correcto?, justifique su respuesta. Calcule su complejidad.

```
producto ← y;
cuenta ← 1;
mientras (cuenta < x) hacer
  producto ← producto + y;
  cuenta ← cuenta + 1
fin_mientras
```

6.3. Determinar la *O-grande* de los algoritmos escritos en los Ejercicios 6.1 y 6.2.

6.4. Diseñar un algoritmo que calcule el número de veces que una cadena de caracteres aparece como una subcadena de otra cadena. Por ejemplo, `abc` aparece dos veces en la cadena `abcdabc` y la cadena `aba` aparece dos veces en la cadena `ababa`. Calcule su complejidad.

6.5. Diseñar un algoritmo que determine el día de la semana de cualquier fecha desde enero de 1.700. Por ejemplo, el 17 de agosto de 2001, era viernes. Calcule su complejidad.

6.6. Determinar la *O-grande* de los algoritmos que resuelven los Ejercicios 6.4 y 6.5.

6.7. Diseñar un algoritmo para determinar si un número *n* es primo. (Un número primo sólo puede ser divisible por él mismo y por la unidad). Calcule su complejidad.

6.8. Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura (*`S = ½ Base x Altura`*). Calcule su complejidad.

6.9. Calcular y visualizar la longitud de la circunferencia y el área de un círculo de radio dado.

6.10. Escribir un algoritmo que indique si una palabra leída del teclado es un palíndromo. Un *palíndromo* (capicúa) es una palabra que se lee igual en ambos sentidos como “radar”. Calcule su complejidad.

6.11. Escribir un algoritmo que cuente el número de ocurrencias de cada letra en una palabra leída como entrada. Por ejemplo, “`Mortimer`” contiene dos “`m`”, una “`o`”, dos “`r`”, una “`y`”, una “`t`” y una “`e`”. Calcule su complejidad.

6.12. Calcular la eficiencia de los siguientes algoritmos:

a)
```
i = 1
mientras (i <= n)
  j = 1
  mientras (j <= n)
    j = j * 2
```

```
      fin_mientras
        i = i + 1
   fin_mientras
```

b)
```
i = 1
mientras (i <= n)
  j = 1
  mientras (j <= i)
    j = j + 1
  fin_mientras
    i = i + 1
fin_mientras
```

c)
```
i = 1
mientras (i <= 10)
  j = 1
  mientras (j <= 10)
    j = j + 1
  fin_mientras
    i = i + 1
fin_mientras
```

CAPÍTULO 7

Algoritmos Recursivos

Objetivos

Con el estudio de este capítulo usted podrá:

- Conocer el funcionamiento de la recursividad.
- Distinguir entre recursividad y recursividad indirecta.
- Resolver problemas numéricos sencillos mediante funciones recursivas.
- Aplicar la técnica algorítmica *divide y vence* para la resolución de problemas.
- Conocer la técnica algorítmica de resolución de problemas *vuelta atrás: backtracking*.
- Aplicar la técnica de backtracking al problema de la *selección óptima.*

Contenido

Conceptos clave

- Backtracking.
- Búsqueda exhaustiva.
- Caso base.
- Complejidad.
- *Divide y vence*.
- Inducción.
- Iteración versus recursión.
- Mejor selección.
- Recursividad.
- Torres de Hanoi.

INTRODUCCIÓN

La *recursividad* (recursión) es aquella propiedad que posee una función por la cual puede llamarse a sí misma. Aunque la recursividad se puede utilizar como una alternativa a la iteración, una solución recursiva es normalmente menos eficiente, en términos de tiempo de computadora, que una solución iterativa, debido a las operaciones auxiliares que llevan consigo las invocaciones suplementarias a las funciónes; sin embargo, en muchas circunstancias el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver. Por esta causa, la recursión es una herramienta poderosa e importante en la resolución de problemas y en la programación. Diversas técnicas algorítmicas utilizan la recursión, son los algoritmos *divide y vence* y los algoritmos de *vuelta atrás*.

7.1. LA NATURALEZA DE LA RECURSIVIDAD

Una función *recursiva* es aquella que se llama a sí mismo, bien directamente o bien indirectamente a través de otra función. La recursividad es un tópico importante examinado frecuentemente en cursos en los que se trata de resolución de algoritmos y en cursos relativos a Estructuras de Datos.

En matemáticas, existen numerosas funciones que tienen carácter recursivo; de igual modo numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo. Por ejemplo, en la búsqueda en páginas Web de “Sierra de Lupiana”, puede ocurrir que aparezcan direcciones (enlaces) que nos lleven a otras páginas, éstas a su vez a otras nuevas y así hasta completar todo lo relativo a la búsqueda inicial.

Una función que tiene sentencias entre las que se encuentra al menos una que llama a la propia función se dice que es *recursiva.* Así, la organización recursiva de una función `funcion1` sería la siguiente:

```
void funcion1(...)
{
  ...
  funcion1();     // llamada recursiva
  ...
}
```

EJEMPLO 7.1. Planteamiento recursivo de la función matemática que suma los n pr imeros números enteros positivos.

Como punto de partida se puede afirmar que para `n = 1` se tiene que la suma `S(1) = 1`. Para `n = 2` se puede escribir `S(2) = S(1) + 2`; en general y aplicando la inducción matemática se tiene:

```
S(n) = S(n-1) + n
```

Se está definiendo la función suma `S()` respecto de sí misma, siempre para un caso más pequeño (`n-1`). `S(2)` respecto a `S(1)`, `S(3)` respecto a `S(2)` y en general `S(n)` respecto a `S(n-1)`.

El algoritmo que resuelve el problema de la suma de los **n** primeros enteros de forma interactiva, utiliza un simple bucle `for`:

```
suma = 0;
for (int contador = 1; contador <= n; contador ++)
   suma += contador;
```

El algoritmo que determina la suma de modo *recursivo* ha de tener presente una condición de salida, o condición de parada. Así, en el caso del cálculo de `S(6)`, la definición es `S(6) = 6 + S(5)`, que de acuerdo a la definición es `5 + S(4)`, este proceso continúa hasta `S(1) = 1` por definición. En matemáticas, la definición de una función en términos de sí misma se denomina definición **inductiva** y conduce naturalmente a una implementación recursiva. El **caso base** de `S(1) = 1` es esencial dado que se detiene, potencialmente, una cadena de llamadas recursivas. Este caso base o condición de salida deben fijarse en cada solución recursiva.

En consecuencia, la implementación del algoritmo mencionado, que calcula la suma de los `n` primeros enteros:

```
long sumaNenteros (int n)
{
  if (n == 1)
    return 1;
  else
    return n + sumaNenteros(n - 1);
}
```

EJEMPLO 7.2. Definir la naturaleza recursiva de la **serie de Fibonacci** : 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Se observa en esta serie que comienza con *0* y *1*, y tiene la propiedad de que cada elemento es la suma de los dos elementos anteriores, por ejemplo:

```
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
...
```

Entonces se puede establecer que:

```
fibonacci(0) = 0
fibonacci(1) = 1
...
fibonacci(n)= fibonacci(n - 1) + fibonacci(n - 2)
```

y la definición recursiva será:

```
fibonacci(n) = n                                        si n = 0  o  n = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)      si n > = 2
```

Obsérvese, que la definición recursiva de los números de fibonacci es diferente de las definiciones recursivas de la suma del Ejemplo 7.1. Así por ejemplo,

```
fibonacci(6) = fibonacci(5) + fibonacci(4)
```

es decir, que para calcular `fibonacci(6)` ha de aplicarse `fibonacci` en modo recursivo dos veces, y así sucesivamente.

La implementación del algoritmo iterativo equivalente sería:

```
long fibonacci (int n)
{
  if (n == 0 || n == 1)
    return n;
  fibinf = 0;
  fibsup = 1;
  for (int i = 2; i <= n; i++)
  {
    int x;
    x = fibinf;
    fibinf = fibsup;
    fibsup = x + fibinf;
  }
  return fibsup;
}
```

El tiempo de ejecución del algoritmo crece linealmente con `n` ya que el bucle es el término dominante. Se puede afirmar que `t(n)` es `O(n)`.

La versión recursiva:

```
long fibonacci (int n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

En cuanto al tiempo de ejecución del algoritmo recursivo ya no es tan elemental establecer una cota superior. Observe que, por ejemplo, para calcular `fibonacci(6)` se calcula, recursivamente, `fibonacci(5)` y cuando termine este `fibonacci(4)`. A su vez, el cálculo de `fibonacci(5)` supone calcular `fibonacci(4)` y `fibonacci(3)`; se está repitiendo el cálculo de `fibonacci(4)`, es una pérdida de tiempo. Por inducción matemática se puede demostrar que el número de llamadas recursivas crece exponencialmente, *t(n)* es $O(\phi^n)$ con $\phi = \frac{1+\sqrt{5}}{2}$.

A tener en cuenta

La formulación recursiva de una función matemática puede ser muy ineficiente sobre todo si se repiten cálculos realizados anteriormente. En estos caso el algoritmo iterativo, aunque no sea tan evidente, es notablemente más eficiente.

7.2. FUNCIONES RECURSIVAS

Una función **recursiva** es una función que se invoca a sí mismo de forma directa o indirecta. En **recursión directa** el código de la función `f()` contiene una sentencia que invoca a `f()`, mientras que en **recursión indirecta** la función `f()` invoca a una función `g()` que invoca a su vez a la función `p()`, y así sucesivamente hasta que se invoca de nuevo a la función `f()`.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia, la definición recursiva debe incluir una *condición de salida*, se denomina **componente base**, en la cual *f(n)* se defina directamente (es decir, no recursivamente) para uno o más valores de *n*.

En definitiva, debe existir una *"forma de salir"* de la secuencia de llamadas recursivas. Así, en el algoritmo que calcula la suma de los n primeros enteros:

$$S(n) = \begin{cases} 1 & n = 1 \\ n + S(n - 1) & n > 1 \end{cases}$$

la condición de salida o componente base es *S (n) = 1* para *n = 1*.

En el ejemplo del algoritmo recursivo de la serie de Fibonacci:

$$F_0 = 0, \quad F_1 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{para} \quad n > 1$$

$F_0 = 0$ y $F_1 = 1$ constituyen el componente base o condiciones de salida y $F_n = F_{n-1} + F_{n-2}$ es el componente recursivo. Una función recursiva correcta debe incluir un componente base o condición de salida ya que en caso contrario se produce una recursión infinita.

> Una función es recursiva si se llama a sí mismo, directamente o bien indirectamente a través de otra función `g()`. Es necesario contemplar un caso base que determina la salida de las llamadas recursivas.

PROBLEMA 7.1. Escribir una función recursiva que calcule el factorial de un número n y un programa que pida un número entero y escriba su factorial llamando a la función.

La *componente base* de la función recursiva que calcula el factorial es que `n = 0`, o incluso `n = 1` ya que en ambos caso el factorial es 1. El problema se resuelve recordando la definición de factorial:

```
n! = 1            si n = 0   o n = 1   (componente base)
n! = n *(n - 1)   si n > 1
```

```
#include <iostream>
using namespace std;
long factorial (int n);

int main()
{
  int n;
  do {
    cout << "Introduzca número n: ";
    cin >> n;
  }while (n < 0);
  cout  << " \t" << n << "!= " << factorial(n) << endl;
  return 0;
}
```

```
long factorial (int n)
{
  if (n <= 1)
    return 1;
  else
  {
    long resultado = n * factorial(n - 1);
    return resultado;
  }
}
```

7.2.1. Recursividad indirecta: funciones mutuamente recursivas

La recursividad indirecta se produce cuando una función llama a otra, que eventualmente terminará llamando de nuevo a la primera función. El programa del Problema 7.2 visualiza el alfabeto utilizando recursión mutua o indirecta.

PROBLEMA 7.2. Escribir una aplicación para mostrar por pantalla el alfabeto, utilizando recursión indirecta.

El programa principal llama a `funcionA()` con el argumento `'Z'` (la última letra del alfabeto). Ésta examina su parámetro *c*, si *c* está en orden alfabético después que `'A'`, llama a `funcionB()`, que inmediatamente invoca a `funcionA()` pasándo el carácter predecesor de *c*. Esta acción hace que `funcionA()` vuelva a examinar *c*, y nuevamente llame a `funcionB()` hasta que *c* sea igual a `'A'`. Las llamadas continúan hasta que *c* sea igual a `'A'`, en este momento, la recursión termina ejecutando la sentencia `cout << )` veintiséis veces y de esa forma visualiza el alfabeto carácter a carácter.

```
#include <iostream>
using namespace std;
void funcionA(char c);
void funcionB(char c);

int main()
{
  cout << "Alfabeto: ";
  funcionA('Z');
  cout  << endl;
  return 0;
}

void funcionA(char c)
{
  if (c > 'A')
    funcionB(c);
  cout  << c;
}

void funcionB(char c)
{
  funcionA(--c);
}
```

7.2.2. Condición de terminación de la recursión

Cuando se implementa una función recursiva será preciso considerar una condición de terminación, ya que en caso contrario la función continuaría indefinidamente llamándose a sí misma y llegaría un momento en que la pila que registra las llamadas se desbordaría. En consecuencia, se necesita establecer en toda función recursiva la *condición de parada* de las llamadas recursivas. Por ejemplo, la condición de parada de la función `factorial()(Problema 7.1)` ocurre cuando `n` es `1` o `0`, en ambos casos el factorial es `1`. Es importante que cada llamada suponga un acercamiento a la condición de parada, en `factorial()` cada llamada decrementa el valor de `n` y, por consiguiente, se acerca a la condición `n == 1`.

En el Problema 7.2 se produce recursión mutua entre `funcionA()` y `funcionB()`, la condición de parada es `c == 'A'`. La primera llamada se hace con `c = 'Z'`, cada llamada mutua decrementa `c` y, por tanto, se va acercando a la letra `'A'`.

> En un algoritmo recursivo, se entiende por *caso base* el que se resuelve sin recursión, directamente, con unas pocas sentencias elementales. El *caso base* se ejecuta cuando se alcanza la *condición de par ada* de llamadas recursiv as. P ara que funcione la recursión el progreso de las llamadas debe tender a la condición de par ada.

7.3. RECURSIÓN *VERSUS* ITERACIÓN

En las secciones anteriores se han estudiado varios algoritmos que se pueden implementar fácilmente de modo recursivo, o bien de modo iterativo. En esta sección se comparan los dos enfoques y se examinan las razones por las que el programador puede elegir un enfoque u otro según la situación específica.

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura repetitiva y la recursión utiliza una estructura de selección.* La iteración y la recursión implican ambas repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante sucesivas llamadas a la función. La iteración y recursión implican cada una un test de terminación (*condición de parada*). La iteración termina cuando la condición del bucle no se cumple, mientras que la recursión termina cuando se reconoce un caso base (se alcanza la condición de parada).

La recursión tiene muchas *desventajas*. Se invoca repetidamente al mecanismo de llamada a función y, en consecuencia, se necesita un tiempo suplementario para realizar cada llamada. Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada recursiva produce que se realice una nueva creación y copia de las variables de la función; esto puede consumir mucha memoria e incrementar el tiempo de ejecución. Por el contrario, la iteración se produce dentro de una función de modo que las operaciones suplementarias en la llamada a la función y asignación de memoria adicional son omitidas.

Entonces, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. Sin embargo, en condiciones críticas de tiempo y de memoria, es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema, la solución a elegir debe ser, normalmente, la iterativa.

Nota de ejecución

Cualquier problema que se puede resolver recursivamente, tiene al menos una solución iterativa utilizando una pila. Se elige el enfoque recursivo frente al enfoque iterativo cuando el recursivo es más natural para la resolución del problema y produce un programa más fácil de comprender y depurar. Otra razón para elegir una solución recursiva es que una solución iterativa puede no ser clara ni evidente.

Consejo de programación

Se ha de evitar utilizar recursividad en situaciones de rendimiento crítico o exigencia de altas prestaciones en tiempo y memoria, ya que las llamadas recursivas emplean tiempo y consumen memoria adicional. No es conveniente el uso de una llamada recursiva para sustituir un simple bucle.

EJEMPLO 7.3. Dado un número natural n obtener la suma de los dígitos de que consta. Presentar un algoritmo recursivo y otro iterativo.

El ejemplo ofrece una estructura clara de comparación entre la resolución de modo iterativo y de modo recursivo. Se asume que el número *n* es natural y que, por tanto, no tiene signo, la suma de los dígitos se puede expresar:

```
suma = suma(n / 10) + modulo (n, 10)   para n > 9
suma = n                              para n ≤ 9   caso base
```

Por ejemplo, si `n = 259`:

```
suma = suma(259 / 10) +  modulo (259,10)   →   2 + 5 + 9 = 16
         ↓
suma = suma(25 / 10)  +  modulo (25,10)    →   2 + 5 ↑
         ↓
suma = suma(2 / 10)   +  modulo (2,10)     →   2 ↑
```

Se puede observar que el caso base, el que se resuelve directamente, es `n ≤ 9` y a su vez es la condición de parada.

Solución recursiva

```
int sumaRecursiva(int n)
{
  if (n <= 9)
    return n;
  else
    return sumaRecursiva(n / 10) + n % 10;
}
```

Solución iterativa

La solución iterativa de hallar la suma de los dígitos del número n se construye con un bucle *mientras*, repitiendo la acumulación del resto de dividir n por 10 y actualizar n en el cociente. La condición de salida del bucle es que n sea menor o igual que 9.

```
int sumaIterativa(int n)
{
  int suma = 0;

  while (n > 9)
  {
    suma += n % 10;
    n /= 10;
  }
  return (suma + n);
}
```

7.3.1. Directrices en la toma de decisión iteración/recursión

1. Considérese una solución recursiva sólo cuando una solución iterativa *sencilla* no sea posible.
2. Utilícese una solución recursiva sólo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables considerando las limitaciones del sistema.
3. Si son posibles las dos soluciones, iterativa y recursiva, la solución recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales a las funciónes.
4. En ciertos problemas, la recursión conduce naturalmente a soluciones que son mucho más fiables de leer y comprender que su correspondiente iterativa. En estos casos los beneficios obtenidos con la claridad de la solución suelen compensar el coste extra (en tiempo y memoria) de la ejecución de un programa recursivo.

Consejo de programación

Una función recursiva que tiene la llamada recursiva como última sentencia (*recursión final*) puede transformarse fácilmente en iterativa reemplazando la llamada mediante un bucle condicional que chequea el caso base.

7.3.2. Recursión infinita

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el caso base o condición de salida.

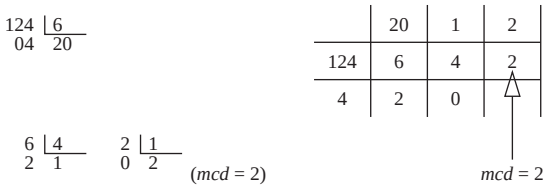
En realidad la **recursión infinita** significa que cada llamada recursiva produce otra llamada recursiva y ésta a su vez otra llamada recursiva y así para siempre. En la práctica dicha función se ejecutará hasta que la computadora agota la memoria disponible y se produce una terminación anormal del programa.

El flujo de control de una función recursiva requiere tres condiciones para una terminación normal:

- Un test para detener (o continuar) la recursión (*condición de salida* o *caso base*).
- Una llamada recursiva (para continuar la recursión).
- Un caso final para terminar la recursión.

EJEMPLO 7.4. Deducir cuál es la condición de salida de la función `mcd()` que calcula el mayor denominador común de dos números enteros `b1` y `b2` (el ***mcd***, máximo común divisor, es el entero mayor que divide a ambos números).

Supongamos dos números 6 y 124; el procedimiento clásico de obtención del ***mcd*** es la obtención de divisiones sucesivas entre ambos números (124 entre 6) si el resto no es 0, se divide el número menor (6, en el ejemplo) por el resto (4, en el ejemplo) y así sucesivamente hasta que el resto sea 0.



En el caso de 124 y 6, el ***mcd*** es 2. En consecuencia, la condición de salida es que el resto sea cero. El algoritmo del ***mcd*** entre dos números *m* y *n* es:

```
mcd(m,n) = n                                si n <= m y n divide a m
mcd(m,n) = mcd(n, m)                        si m < n
mcd(m,n) = mcd(n, resto de m divido por n)  en caso contrario.
```

Los pasos anteriores significan: el ***mcd*** es `n` si `n` es el número más pequeño y `n` divide a `m`. Si `m` es el número más pequeño, entonces la determinación del *mcd* se debe ejecutar con los argumentos transpuestos. Por último, si `n` no divide a `m`, el ***mcd*** se obtiene encontrando el `mcd` de `n` y el resto de `m` divido por `n`. La función recursiva:

```
int mcd(int m, int n)
{
  if (n <= m && m % n == 0)
    return n;
  else if (m < n)
    return mcd(n, m);
  else
    return mcd(n, m % n);
}
```

7.4. ALGORITMOS *DIVIDE* Y *VENCE*

Una de las técnicas más importantes para la resolución de muchos problemas de computadora es la denominada *"divide y vence"*. El diseño de algoritmos basados en esta técnica consiste

en transformar (dividir) un problema de tamaño *n* en problemas más pequeños, de tamaño menor que *n* pero similares al problema original. De modo que resolviendo los subproblemas y combinando las soluciones se pueda construir fácilmente una solución del problema completo (*vencerás*).

Normalmente, el proceso de división de problema en otros de tamaño menor va a dar lugar a que se llegue al *caso base*, cuya solución es inmediata. A partir de la obtención de la solución del problema para el caso base, se combinan soluciones que amplían el tamaño del problema resuelto, hasta que el problema original queda resuelto.

Por ejemplo, se plantea el problema de dibujar un segmento que está conectado por los puntos en el plano `(x1,y1)` y `(x2,y2)`. El problema puede descomponerse así: determinar el punto medio del segmento, dibujar dicho punto y *dibujar los dos segmentos mitad obtenidos al dividir el segmento original por el punto mitad*. El tamaño del problema se ha reducido a la mitad, el hecho de dibujar un segmento de longitud *n* se ha transformado en dibujar dos segmentos de tamaño *n*/2. Sobre cada segmento mitad se vuelve aplicar el mismo procedimiento, de tal forma que llega un momento en que a base de dividir el segmento se alcanza uno de longitud cercana a cero, se ha alcanzado el caso base, y simplemente se dibuja un punto. El planteamiento del problema es totalmente recursivo, cada tarea de dibujar realiza las mismas acciones, por consiguiente, la solución se puede plantear con llamadas recursivas al proceso de dibujar un segmento, cada vez con menor tamaño, exactamente la mitad.

Un algoritmo *"divide y vence"* se define de manera recursiva, de tal modo que se llama a sí mismo sobre un conjunto menor de elementos. Normalmente, se implementan con dos llamadas recursivas, cada una con un tamaño menor, generalmente la mitad. Se alcanza el *caso base* cuando el problema se resuelve directamente.

Un algoritmo *divide y vence* consta de dos partes:

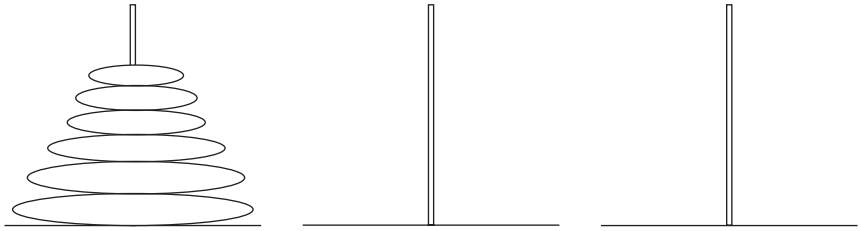
1. *Dividir recursivamente* el problema original en subproblemas cada vez más pequeños.
2. *Resolver* (*vence*) el problema dando solución a los subproblemas a partir del caso base.

En esta sección se describen una serie de ejemplos que son problemas clásicos resueltos mediante recursividad. Entre ellos se destacan el problema matemático, las Torres de Hanoi, función de búsqueda binaria, la ordenación por mezclas, etc.

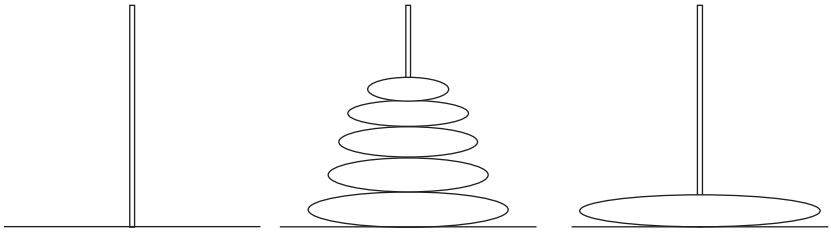
7.4.1. Torres de Hanoi

Este juego (un algoritmo clásico) tiene sus orígenes en la cultura oriental y en una leyenda sobre el Templo de Brahma cuya estructura simulaba una plataforma metálica con tres varillas y discos en su interior. El problema en cuestión suponía la existencia de tres varillas (*A*, *B*, y *C*) o postes en los que se alojaban discos (*n* discos) que se podían trasladar de una varilla a otra libremente pero con una condición: cada disco era ligeramente inferior en diámetro al que estaba justo debajo de él.

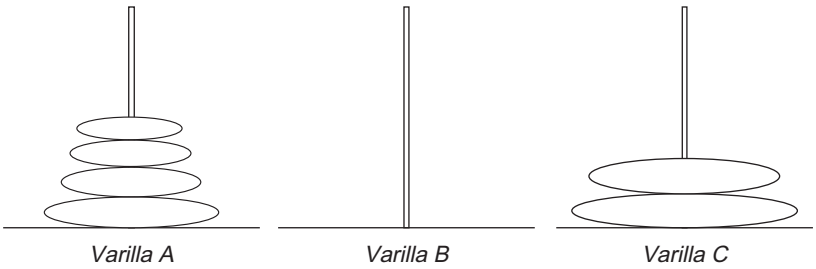
La secuencia de figuras ilustran el problema, inicialmente en la varilla *A* hay seis discos, y se desea trasladar a la varilla *C* conservando la condición de que cada disco sea ligeramente inferior en diámetro al que tiene situado debajo de él. Así, por ejemplo, se pueden cambiar



cinco discos de golpe de la varilla *A* a la varilla *B*, y el disco más grande a la varilla *C*. Ya ha habido una transformación del problema en otro de menor tamaño, se ha *divido* el problema original.



Ahora el problema se centra en pasar los cinco discos de la varilla *B* a la varilla *C*. Para lo cual se se pasan los cuatro discos superiores de la varilla *B* a la varilla *A* y, a continuación, se pasa el disco de mayor tamaño de la varilla *B* a la varilla *C*, y así sucesivamente. El proceso continúa del mismo modo, siempre *dividiendo* el problema en dos de menor tamaño, hasta que finalmente se queda un disco en la varilla *B* que es *el caso base* y a su vez la condición de parada.



La solución del problema es claramente recursiva, además con las dos partes mencionadas anteriormente: división recursiva y solución a partir del caso base.

Diseño del algoritmo

La función `hanoi()` declara las tres varillas como parámetros de tipo `char`, el orden es el siguiente:

```
varinicial, varcentral, varfinal
```

lo que implica que se están moviendo discos desde la varilla inicial a la final utilizando la varilla central como auxiliar para almacenar los discos. Si *n = 1* se tiene el caso base, se resuelve directamente moviendo el único disco desde la varilla inicial a la varilla final. El algoritmo es el siguiente:

1. Si *n* es 1:
   1.1. Mover el disco `1` de `varinicial` a `varfinal`.
2. Sino:
   2.1. Mover *n - 1* discos desde `varinicial` hasta `varcentral` utilizando `varfinal` como auxiliar.
   2.2. Mover el disco *n* desde varilla inicial `varinicial` a `varfinal`.
   2.3. Mover *n - 1* `discos` desde la varilla auxiliar `varcentral` a `varfinal` utilizando como auxiliar `varinicial`.

Es decir, si `n` es 1, se alcanza *el caso base*, la condición de salida o terminación del algoritmo. Si `n` es mayor que 1, las etapas recursivas *2.1, 2.2 y 2.3* son tres subproblemas más pequeños, uno de los cuales es la condición de salida.

Las Figuras 7.1 a 7.6 muestran el algoritmo:

Etapa 1: Mover *n-1* discos desde varilla inicial (A).

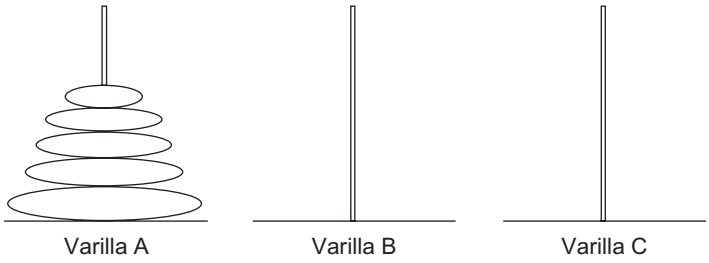


Figura 7.1. Situación inicial.

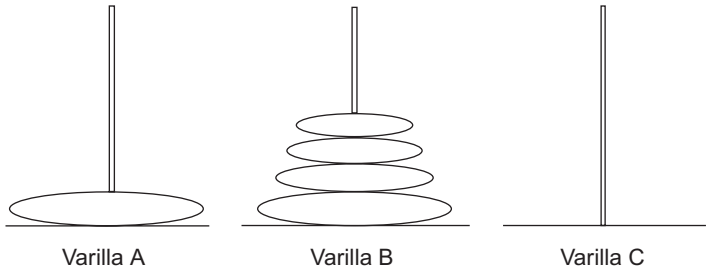


Figura 7.2. Después del movimiento.

Etapa 2: Mover un disco desde A a C

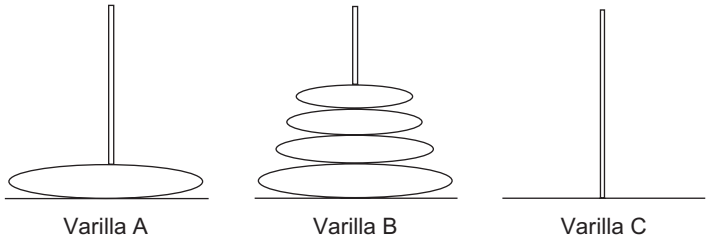


Figura 7.3. Situación de partida de etapa 2.

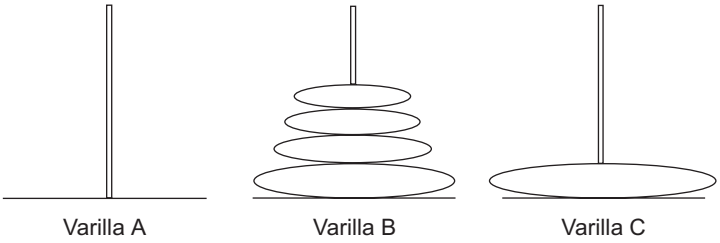


Figura 7.4. Después de la etapa 2.

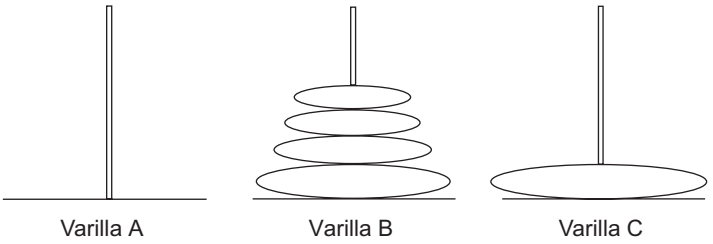


Figura 7.5. Antes de la etapa 3.

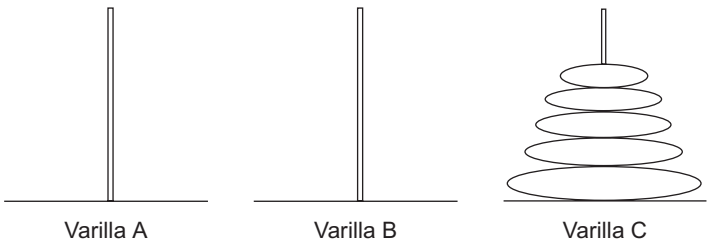


Figura 7.6. Después de la etapa 3.

La primera etapa mueve *n - 1* discos desde la varilla inicial a la varilla central utilizando la varilla final como almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada recursiva es `varinicial`, `varfinal` *y* `varcentral`.

```
hanoi(varinicial, varfinal, varcentral, n - 1);
```

La segunda etapa, simplemente mueve el disco mayor desde la varilla inicial a la varilla final:

```
cout  << "Mover disco" << n << "desde varilla" << varinicial
      << "a varilla" << varfinal;
```

La tercera etapa del algoritmo mueve *n - 1* discos desde la varilla central a la varilla final utilizando `varinicial` para almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada a la función recursiva es: **`varcentral, varinicial y varfinal.`**

```
hanoi(varcentral, varinicial, varfinal, n-1);
```

Implementación de las Torres de Hanoi

La implementación del algoritmo se apoya en los nombres de las tres varillas o alambres: `'A'`, `'B'` y `'C'` que se pasan a la función `hanoi()`. La función tiene un cuarto parámetro que es el número de discos, *n*, que intervienen. Se obtiene un listado de los movimientos que transferirá los *n* discos desde la varilla inicial, `'A'`, a la varilla final, `'C'`. La codificación:

```
void hanoi(char varinicial, char varcentral, char varfinal, int n)
{
  if (n == 1)
    cout  << "Mover disco" << n << "desde varilla"
          << varinicial << "a varilla" << varfinal;
  else
  {
    hanoi(varinicial, varfinal, varcentral, n - 1);
    cout  << "Mover disco" << n << "desde varilla"
          << varinicial << "a varilla" << varfinal;
    hanoi(varcentral, varinicial, varfinal, n - 1);
  }
}
```

Análisis del algoritmo Torres de Hanoi

Es de destacar que la función `hanoi()` resolverá el problema de las Torres de Hanoi para cualquier número de discos. Fácilmente se puede encontrar el árbol de llamadas recursivas para n = 3 discos, que en total realiza 7 ($2^3 - 1$) llamadas a `hanoi()` y escribe 7 movimientos de disco. El problema de 5 discos se resuelve con 31 ($2^5 - 1$) llamadas y 31 movimientos. Si se supone que `T(n)` es el número de movimientos para `n` discos, entonces teniendo en cuenta que la función realiza dos llamadas con `n-1` discos y mueve el disco `n`, se tiene la recurrencia:

$$T(n) = \begin{cases} 1 & si\ n = 1 \\ 2T(n-1) + 1 & si\ n > 1 \end{cases}$$

Los sucesivos valores que toma `T` según n: `1, 3, 7, 15, 31, 63 ...` $2^n - 1$

En general, el número de movimientos, requeridos para resolver el problema de *n* discos es $2^n - 1$. Por consiguiente, a medida que crece n crece exponencialmente el tiempo de ejecución de la función. Por esta razón, la ejecución de la función con un valor de *n* mayor que 10 requiere gran cantidad de prudencia para evitar desbordamientos de memoria y ralentización de tiempo.

7.4.2. Búsqueda binaria

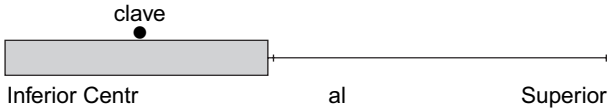
La búsqueda binaria consiste en localizar una clave especificada dentro de una lista o array ordenado de *n* elementos. El algoritmo de búsqueda binaria se puede describir recursivamente aplicando la técnica *"divide y vence"*.

Supóngase que se tiene un array ordenado `a[]` con un límite inferior y un límite superior. Dada una clave (valor buscado) la búsqueda comienza en la posición central de la lista.

`central = (inferior + superior) / 2` Comparar `A[central]` y `clave`

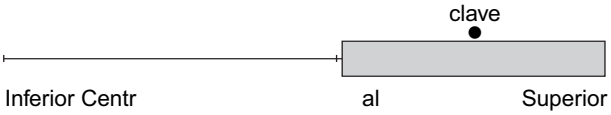
Si se encuentra la clave se ha alcanzado la condición de terminación que permite detener la búsqueda y devolver el índice central. Si no se produce la coincidencia, dado que la lista está ordenada, se centra la búsqueda en la "sublista inferior" (a la izquierda de la posición central) o en la "sublista superior" (a la derecha de la posición central); el problema de la búsqueda se ha dividido jústamente en la mitad. Cada vez el tamaño de la secuencia donde buscar se reduce a la mitad, así hasta que se encuentre el elemento, o bien la lista resultante esté vacía.

1. Si `clave < a[central]`, el valor buscado sólo puede estar en la mitad izquierda de `a[]`con elementos en el rango, `inferior` a `central - 1`.



Búsqueda en sublista izquierda
`[inferior .. central – 1]`

2. Si `clave > a[central]`, el valor buscado sólo puede estar en la mitad superior de `a[]`con elementos en el rango, `central + 1` a `superior`.



3. La búsqueda continúa en sublistas más y más pequeñas, exactamente la mitad, con dos llamada recursivas, una se corresponde con la sublista inferior y la otra con la sublista superior. El algoritmo termina, o con éxito (*aparece la clave buscada*) o sin éxito (*no aparece la clave buscada*), situación que ocurrirá cuando el límite superior de la lista sea más pequeño que el límite inferior. La condición `inferior > superior` es la condición de salida o terminación sin éxito y el algoritmo devuelve el índice -1.

Las operaciones descritas se escriben en la función `busquedaBinaria()`.

```
int busquedaBinaria(double a[], double clave, int inferior, int superior)
{
  int central;
```

```
    if (inferior > superior)      // no encontrado
      return -1;
    else
    {
      central = (inferior + superior) / 2;
      if (a[central] == clave)
        return central;
      else if (a[central] < clave)
        return busquedaBinaria(a, clave, central + 1, superior);
      else
        return busquedaBinaria(a, clave, inferior, central - 1);
    }
  }
```

Análisis del algoritmo

El *peor de los casos* que hay que contemplar en una búsqueda es que ésta no tenga éxito. El tiempo del algoritmo recursivo de búsqueda binaria depende del número de llamadas. Cada llamada reduce la lista a la mitad, así progresivamente se llega a que el tamaño de la lista es unitario, y en la siguiente llamada el algoritmo termina, el tamaño es `0`. La sucesión de tamaños (`t+1` es el número de llamadas):

$$n/2,\ n/2^2,\ n/2^3,\ \ldots,\ n/2^t = 1$$

tomando logaritmos, $t = \log n$. Por tanto, el número de llamadas es $\log n + 1$. Como cada llamada es de complejidad constante se puede afirmar que la complejidad, en término de notación `O`, es logarítmica $O(\log n)$.

7.5. ORDENACIÓN POR MEZCLAS: *MERGESORT*

La idea básica de este algoritmo de ordenación es la mezcla (*merge*) de listas ya ordenadas. El algoritmo sigue la típica estrategia *divide y vence*. Cada paso se basa en *dividir* el problema de ordenar `n` elementos en dos subproblemas más pequeños, de tamaño mitad, de tal forma que una vez ordenada cada mitad se mezclan para así resolver el problema original. Con más detalle: *se ordena la primera mitad de la lista, se ordena la segunda mitad de la lista y una vez ordenadas su mezcla da lugar a una lista de elementos ordenada. A su vez, la ordenación de la sublista mitad consiste en los mismos pasos, ordenar la primera mitad, ordenar la segunda mitad y mezclar.*

La sucesiva división de la lista en dos hace que el problema (número de elementos) cada vez sea más pequeño; así hasta que la lista tiene 1 elemento y, por tanto, en sí ordenada, es el *caso base.* A partir de dos sublistas de un número mínimo de elementos, empiezan las mezclas de pares de sublistas ordenadas, dando lugar a sublistas ordenadas del doble de elementos que la anterior, hasta alcanzar la lista completa. El Ejemplo 7.6 describe el proceso de *división* y *mezcla*.

EJEMPLO 7.6. Seguir la estrategia del algoritmo mergesort para ordenar la lista:

9 1 3 5 10 4 6

La Figura 7.7 muestra las sucesivas divisiones que origina el algoritmo. Cada división se corresponde con una llamada recursiva, por lo que a la vez queda reflejado el árbol de llamadas recursivas.

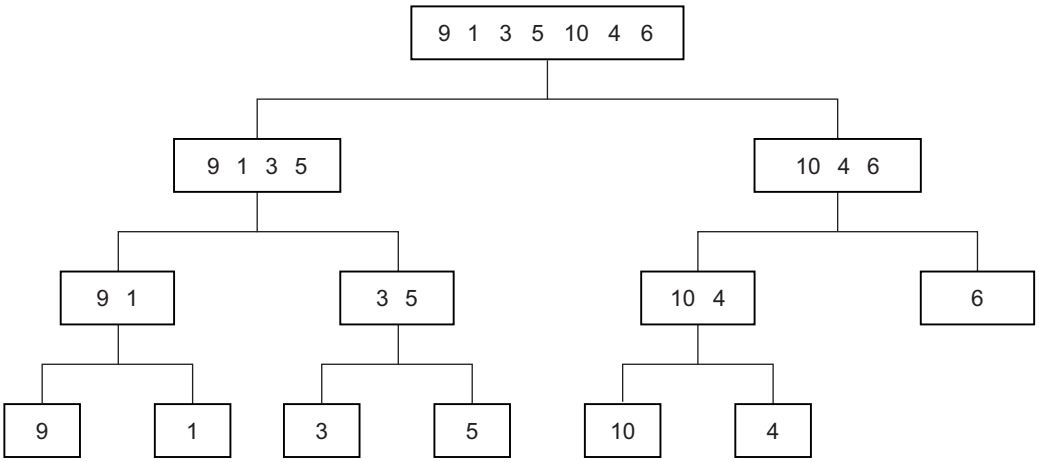


Figura 7.7. Sucesivas divisiones de una lista por algoritmo mergesort.

La Figura 7.8 muestra la mezcla de sublistas, primero de tamaño 1, hasta que el proceso se propaga a la ráiz de las llamadas recursivas y la lista queda ordenada.

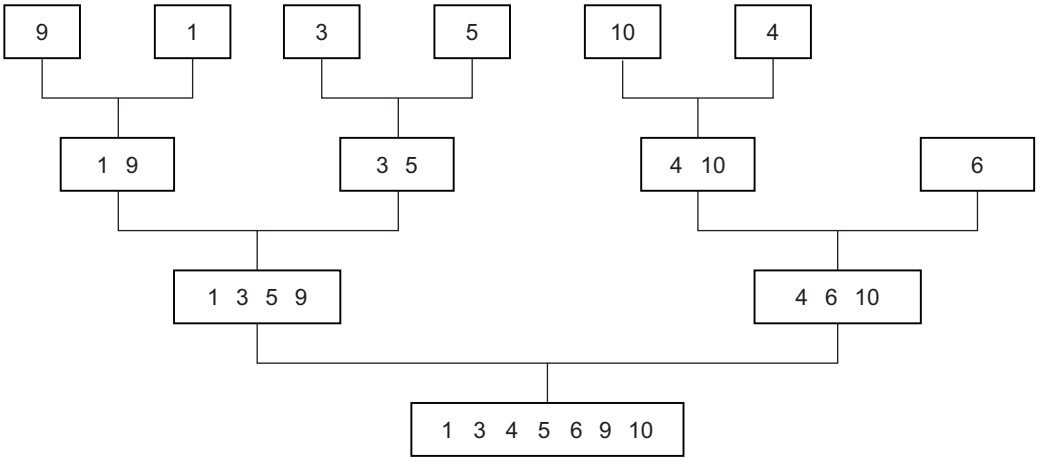


Figura 7.8. Progresivas mezclas de sublistas, de arriba abajo.

7.5.1. Algoritmo *mergesort*

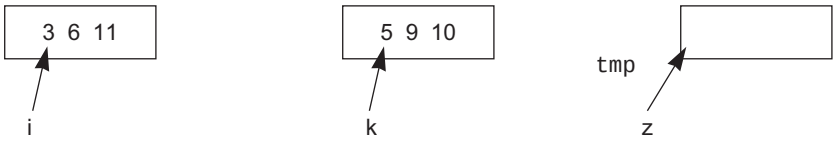
Este algoritmo de ordenación se diseña fácilmente con ayuda de las llamadas recursivas para dividir las listas en dos mitades; posteriormente se invoca a la función de mezcla de dos listas ordenadas. La delimitación de las dos listas se hace con tres índices: `primero, central` y `ultimo`. Así si se tiene una lista de 10 elementos los valores de los índices:

```
primero = 0;  ultimo = 9;  central = (primero + ultimo) / 2 = 4
```

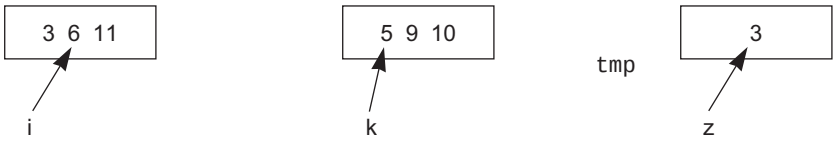
La primera sublista comprende los elementos $a_0 \ldots a_4$; y la segunda los elementos siguientes $a_{4+1} \ldots a_9$. El algoritmo para el array a:

```
mergesort(a, primero, ultimo)
  si (primero < ultimo) Entonces
    central = (primero+ultimo)/2
    mergesort(a, primero, central);      ordena primera mitad de la lista
    mergesort(a, central+1, ultimo);     ordena segunda mitad de la lista
    mezcla(a, primero, central, ultimo); fusiona las dos sublistas ordena-
                                         das, delimitadas por los extremos
  fin_si
fin
```

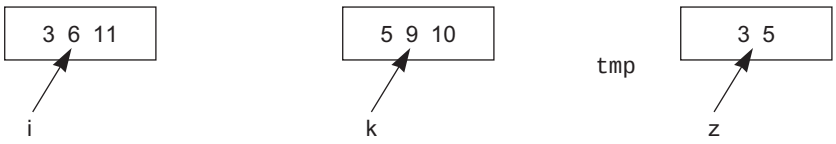
El algoritmo de mezcla utiliza un array auxiliar, `tmp[]` para realizar la fusión entre dos sublistas ordenadas, que se encuentran en el vector `a[]`, delimitadas por los índices `izda`, `medio` y `drcha`. A partir de estos índices se pueden recorrer las sublistas como se muestra en la Figura 7.9 con las variables `i, k`. En cada pasada del algoritmo de mezcla se compara `a[i]` con `a[k]`, el menor se copia en el vector auxiliar, `tmp[]`, y avanzan los índices de la sublista y del vector auxiliar.



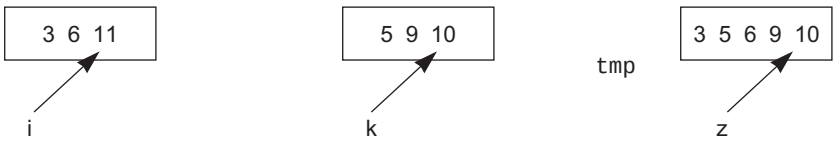
a) Punto de partida en la mezcla de dos sublistas ordenadas.



b) Primera pasada, se copia el elemento `a[i]` en `tmp [z]` y avanzan `i`, `z`.



c) Segunda pasada, se copia el elemento `a[k]` en `tmp [z]` y avanzan `k`, `z`.



d) índices y vector auxiliar después de 5 pasadas.

Figura 7.9. Mezcla de sublistas ordenadas.

El algoritmo de mezcla es de complejidad lineal, `O(n)`, debido a que se realizan tantas pasadas como número de elementos, en cada pasada se realiza una comparación y una asignación (complejidad constante, `O(1)`). El número de pasadas que realiza el algoritmo mergesort es igual a la parte entera de $\log_2 n$ y, por consiguiente, el tiempo de ejecución del algoritmo es `O(n log n)`.

Codificación

```
void mergesort(double a[], int primero, int ultimo)
{
  int central;

  if (primero < ultimo)
  {
    central = (primero + ultimo) / 2;
    mergesort(a, primero, central);
    mergesort(a, central + 1, ultimo);
    mezcla(a, primero, central, ultimo);
  }
}

void mezcla(double a[], int izda, int medio, int drcha)
{
  double* tmp;
  int i, k, z;
  tmp = new double[drcha - izda + 1];
  i = z = izda;
  k = medio + 1;
    // bucle para la mezcla, utiliza tmp[] como array auxiliar
  while (i <= medio  && k <= drcha)
  {
    if (a[i] <= a[k])
      tmp[z++] = a[i++];
    else
      tmp[z++] = a[k++];
  }
    // se mueven elementos no mezclados de sublistas
  while (i <= medio)
    tmp[z++] = a[i++];

  while (k <= drcha)
    tmp[z++] = a[k++];
    // Copia de elementos de tmp[] al array a[]
  for (i = izda; i <= drcha; i++)
    a[i] = tmp[i];
  delete tmp;
}
```

7.6. *BACKTRACKING,* ALGORITMOS DE VUELTA ATRÁS

La resolución de algunos problemas exige probar sistemáticamente todas las posibilidades que pueden existir para encontrar una solución. Los algoritmos de *vuelta atrás* utilizan la recursividad para probar cada una de las posibilidades de encontrar la solución.

Este método de resolución de problemas recurre a realizar una búsqueda exhaustiva, sistemática, de una posible solución al problema planteado. Descomponen el proceso de búsqueda o tanteo en tareas parciales. Cada *tarea parcial* realiza las mismas acciones que la tarea anterior y por eso se expresa, frecuentemente, de forma recursiva.

Por ejemplo, suponga el siguiente problema: determinar los sucesivos saltos que debe hacer un caballo, desde una posición inicial cualquiera, (con su forma típica de moverse) para que pase por todas las casillas de un tablero de ajedrez vacío. La *tarea parcial* consiste en realizar un salto válido, en cuanto a que esté dentro de las coordenadas del tablero y que no haya pasado ya por la casilla destino. En este problema todas las posibilidades son los ocho saltos que puede realizar un caballo desde una casilla dada.

El proceso general de los algoritmos de *vuelta atrás* se contempla como un método de prueba o búsqueda, que gradualmente construye tareas básicas y las inspecciona para determinar si conducen a la solución del problema. Si una tarea no conduce a la solución, se prueba con otra tarea básica. Es una prueba sistemática hasta llegar a la solución, o bien determinar que no la hay al haberse agotado todas las posibilidades de crear tareas.

Para recordar

Dos son las características principales de los algoritmos de vuelta atrás:

1. *Búsqueda exhaustiva*, con todas las posibilidades o alternativas, de soluciones (tareas) parciales que conducen a la solución del problema.
2. *Vuelta atrás*, si una solución parcial no conduce a la solución global del problema se *vuelve atrás* para probar (ensayar) con otra solución parcial.

7.6.1. Modelo de los algoritmos de vuelta atrás

El esquema general de estos algoritmos:

```
procedimiento ensayarSolucion
inicio
  <inicializar cuenta de posibles selecciones>
  repetir
    <tomar siguiente selección (tarea)>
    <determinar si selección es válida>
    si válido entonces
      <anotar selección>
      si  < problema soluciónado >  entonces
         exito = true
      si no
         ensayarSolución {llamada para realizar otra tarea}
                         {vuelta atrás, analiza si se ha alcanzado
                           la solución del problema}
        si no exito entonces
          <borrar anotación> {el bucle se encarga de probar con otra
                              selección}
        fin_si
      fin_si
    fin_si
```

```
    hasta (exito) o (< no más posibilidades >)
fin
```

Éste es el esquema general, es inmediato pensar que se necesita adaptarlo a la casuística del problema a resolver.

7.6.2. Problema del salto del caballo

En un tablero de ajedrez de `n x n` casillas, se tiene un caballo situado en la posición inicial de coordenadas (x_0,y_0). El problema consiste en encontrar, si existe, un camino que permita al caballo pasar exactamente una vez por cada una de las casillas de tablero, teniendo en cuenta los movimientos (saltos) permitidos a un caballo de ajedrez.

Éste es un ejemplo clásico de problema que se resuelve con el esquema del algoritmo de *vuelta atrás*. El problema consiste en buscar la secuencia de saltos que tiene que dar el caballo, partiendo de una casilla cualquiera, para pasar por cada una de las casillas del tablero. Se da por supuesto que el tablero está vacío, no hay figuras excepto el caballo. Lo primero que hay que tener en cuenta es que el caballo, desde una casilla, puede realizar hasta 8 movimientos; la Figura 7.10 representa estos movimientos.

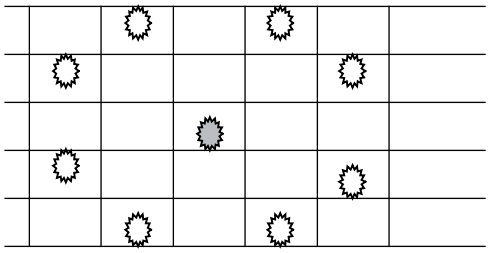


Figura 7.10. Los ocho posibles saltos del caballo.

Por consiguiente, en este problema el número de *posibles selecciones* es ocho. La tarea básica o *solución parcial* consiste en que el caballo realice un nuevo movimiento entre los ocho posibles.

Los ocho posibles movimientos de caballo se obtienen sumando a la posición actual de éste, `(x, y)`, unos desplazamientos relativos, éstos son:

```
d = {(2,1), (1,2), (-1,2), (-2,1), (-2,-1), (-1,-2), (1,-2), (2,-1)}
```

Por ejemplo, suponiendo que el caballo se encuentra en la casilla `(3,5)`, los posibles movimientos que puede realizar:

```
{(5,6), (4,7), (2,7), (1,6), (1,4), (2,3), (4,3), (5,4)}
```

No siempre será posible realizar los ocho movimientos, se debe comprobar que la casilla destino esté dentro del tablero y también que no haya pasado previamente el caballo por esa casilla. En caso de ser posible el movimiento se anota, guardando el número del salto realizado.

```
tablero[x][y] = numeroSalto    { número del Salto }
nuevaCoorX    = x + d[k][1]
nuevaCoorY    = y + d[k][2]
```

Las llamadas a la función recursiva transmiten las nuevas coordenadas y el nuevo salto a realizar.

```
saltoCaballo(nuevaCoorX, nuevaCoorY, numeroSalto + 1)
```

La condición que determina que el problema se ha resuelto está ligada con el objetivo que se persigue, y en este problema es que se haya pasado por las n^2 casillas, en definitiva que el caballo haya realizado $n^2 - 1$ (63) saltos.

¿Qué ocurre si se agotan los ocho posibles movimientos sin alcanzar la solución? Se vuelve al movimiento anterior, *vuelta atrás*, se borra la anotación para ensayar con el siguiente movimiento.

Representación del problema

Una matriz de enteros es la estructura que mejor se ajusta al tablero. Cada elemento de la matriz guarda el número de salto por el que pasa el caballo. Debido a que los arrays en C++ siempre tienen como índice inferior `0`, se ha preferido reservar una fila y columna más para el tablero y así representarlo más fielmente.

```
const int N = 8;
const int n = (N+1);
int tablero [n][n];
```

Una posición de `tablero` puede contener:

`TABLERO[X][Y]=` { `0` Por la casilla `(x,y)` no pasó el caballo.
{ `i` Por la casilla `(x,y)` pasó el caballo en el salto `i`.

Codificación del algoritmo *Salto del caballo*

```
const int N = 8;
const int n = (N+1);
int tablero[n][n];
int d[8][2] = {{2,1}, {1,2}, {-1,2}, {-2,1},{-2,-1},{-1,-2},
               {1,-2}, {2,-1}}; // desplazamientos relativos del caballo

void saltoCaballo(int i, int x, int y, bool& exito)
{
  int nx, ny;
  int k;

  exito = false;
  k = 0;   // inicializa contador de posibles (8) movimientos

  do {
    k++;
    nx = x + d[k - 1][0];
    ny = y + d[k - 1][1];
        // determina si nuevas coordenadas son aceptables
```

```
      if ((nx >= 1) && (nx <= N) &&
          (ny >= 1) && (ny <= N) &&
          (tablero[nx][ny] == 0))
      {
        tablero[nx][ny]= i;       // anota movimiento
        if (i < N * N)
        {
          saltoCaballo(i + 1, nx, ny, exito);
                // analiza si se ha completado la solución
          if (!exito)
          {                          // no se alcanzó la solución
            tablero[nx][ny] = 0;     // se borra anotación
          }
        }
        else
          exito = true;            // caballo ha cubierto el tablero
      }
    } while ((k < 8) && !exito);
}

void escribeTablero()
{
  int i,j;
  for(i = 1; i <= N; i++)
  {
    for (j = 1; j <= N; j++)
       cout  << tablero[i][j] << "  ";
    cout  << endl;
  }
}
```

Nota de programación

Los algoritmos que hacen una búsqueda e xhaustiva de la solución tienen tiempos de ejecución muy elevados. La complejidad es exponencial, por lo que puede ocurr ir que si no se ejecuta en un ordenador potente éste se b loquee.

7.6.3. Problema de la ocho reinas

El juego de colocar ocho reinas en un tablero de ajedrez sin que se ataquen entre sí es otro de los ejemplos del uso de los métodos de búsqueda exhaustiva, sistemática, de los algoritmos de *vuelta atrás*. El problema se plantea de la forma siguiente:

"*dado un tablero de ajedrez (8 × 8 casillas), hay que situar ocho reinas de forma que ninguna de ellas pueda actuar (“comer”) a cualquiera de las otras*".

En primer lugar, conviene recordar que la reina puede moverse a lo largo de la columna, fila y diagonales donde se encuentra. Sin embargo, es posible realizar una *poda* y reducir los movimientos de las reinas en el problema. Cada columna del tablero puede contener una y sólo una reina, la razón es inmediata: si por ejemplo en la columna 1 se encuentra la reina 1 y en

esta columna se sitúa otra reina entonces se *atacan* mutuamente. Entonces, de las `n x n` casillas que puede ocupar una reina, se limita su ubicación a las 8 casillas de la columna en que se encuentra. De tal forma que si se numeran las reinas de 1 a 8, entonces la reina `i` se va a situar en alguna de las casillas de la columna `i`.

Lo primero a considerar a la hora de aplicar el *algoritmo de vuelta atrás* es la *tarea básica* (*solución parcial*) que exhaustivamente se va a realizar. La propia naturaleza del problema de las 8 reinas determina que la tarea sea *tantear* si es posible ubicar la reina número *i*, para lo que hay 8 alternativas correspondientes a las 8 filas de la columna `i`. La comprobación de que una selección es válida se hace *comprobando* que en la fila selecionada y en las dos diagonales en que una reina puede atacar, no haya otra reina colocada anteriormente. Cada paso amplía la solución parcial ya que aumenta el número de reinas colocadas.

La segunda cuestión es analizar la *vuelta atrás*. ¿Qué ocurre si no se es capaz de colocar las 8 reinas?: se borra la ubicación, la fila donde se ha colocado, y se ensaya con la siguiente fila válida.

La realización de cada tarea supone ampliar el número de reinas colocadas, hasta llegar a la solución completa, o bien determinar que no es posible colocar la reina actual en las 8 posibles filas. Entonces, en la *vuelta atrás* se coloca la reina actual en otra fila válida para realizar un nuevo *tanteo*.

Representación del problema

Debido a que el objetivo del problema es encontrar la fila en que se sitúa la reina de la columna *`i`*, se define el vector entero, `reinas[]`, tal que cada elemento contiene el índice de fila donde se sitúa la reina, o bien cero. El número de reina, *`i`*, es a su vez el índice de columna dentro de la cual se puede colocar entre los ocho posibles valores de fila.

```
const int N = 8;
const int n = (N+1);
int reinas[n];
```

Verificación de la ubicación de una reina

Al colocar la reina *i* en la fila `j` hay que comprobar que no se ataque con las reinas colocadas anteriormente. Una reina *i*, ocupa la columna `i`, situada en la fila `j` ataca, o es atacada, por otra reina que esté en la misma fila, o bien por otra que esté en la misma diagonal (`i-j`), o en la diagonal (`i+j`). La función `valido()` realiza esta comprobación mediante un bucle con tantas iteraciones como reinas colocadas.

Codificación del algoritmo de las *8 reinas*

```
const int N = 8;
const int n = (N+1);
int reinas[n];

void ponerReina(int i, bool& solucion)
{
  int k;
  solucion = false;
  k = 0;               // inicializa contador de movimientos
  do {
    k++;
    reinas[i] = k;     // coloca y anota reina i en fila k
```

```
      if (valido(i))
      {
        if (i < N)
        {
          ponerReina(i + 1, solucion);
          // vuelta atrás
          if (!solucion)
            reinas[i] = 0;
        }
        else    // todas las reinas colocadas
          solucion = true;
      }
    } while(!solucion && (k < 8));
  }

  bool valido(int i)
  {
    // comprueba si la reina de la columna i es atacada por alguna
    // reina colocada anteriormente
    int r;
    bool v = true;

    for(r = 1; r <= i - 1; r++)
    {
      v = v && (reinas[r]!= reinas[i]);  // no esté en la misma fila
      v = v && ((reinas[i] - i) != (reinas[r] - r));          // diagonal 1
      v = v && ((reinas[i] + i) != (reinas[r] + r));   // diagonal 2
    }
    return v;
  }
```

7.6.4. Todas las soluciones

El algoritmo de las ocho reinas y el del salto del caballo encuentran una solución y terminan. En lugar de terminar el proceso puede continuar y encontrar todas las solucuines. Las variaciones del algoritmo son mínimas, consiste en cambiar el bucle condicional por un bucle automático que itere para cada una de los posibles alternativas, de tal forma cada vez que se encuentre una solución llamar a la función que la escriba por pantalla. El apartado siguiente aplica el algoritmo de vuelta atrás para encontrar todas las soluciones.

7.6.5. Objetos que totalizan un peso

El planteamieno es el siguiente: *se tiene un conjunto de objetos de pesos* `p1, p2, p3,...,pn;` *se quiere estudiar si existe una selección de dichos objetos que totalice exactamente un peso v que se tiene como objetivo.*

Por ejemplo, supóngase los objetos de pesos `4, 3, 6, 2, 1` y el peso objetivo `v = 12`. La selección formada por los objetos primero, tercero y el cuarto totaliza el peso objetivo, ya que la suma de sus pesos, `4 + 6 + 2 = 12`.

Algoritmo

Como la finalidad es obtener una selección de objetos, entonces la *tarea básica* (*solución parcial*) consiste en añadir un objeto nuevo, para probar si con su peso se alcanza el peso objetivo,

o bien se avanza en la dirección de alcanzar la solución. La repetición de esta tarea permitirá alcanzar la solución que en este problema es totalizar el peso objetivo.

De nuevo hay una búsqueda sistemática, exhaustiva, de la solución dando pasos hacia adelante. Si se llega a una situación en la que no se consigue el peso objetivo porque es superado, entonces se retrocede, *vuelta atrás*, para eliminar el objeto añadido y probar con otro y volver a inspeccionar si a partir de él se consigue el objetivo. El proceso de selección termina en el momento de totalizar el peso objetivo, o bien se haya ensayado con todos los objetos.

El número de posibles elecciones de objetos con los que se pueden ensayar son los `n` objetos disponibles.

Representación de los datos

El peso de los objetos se supone que son valores enteros. Entonces, la secuencia de objetos que forman el problema se representa en un array de tal forma que cada posición almacena el peso de un objeto.

Como cada tarea, cada llamada recursiva, selecciona un objeto nuevo, la `bolsa` en la se van metiendo los objetos es un objeto `Conjunto` (se supone con las operaciónes `insertar`, `quitar`, y `pertenece`).

Codificación

Al codificar este programa se utilizan todas las posiciones del array, desde la posición `0` a la última (`M-1`). Por esa razón, habrá indexaciones con `candidato - 1` para que se corresponda con la posición donde se guarda el peso del objeto.

```
const int M = 12;                       // máximo número de objetos
int n;                                  // número de objetos
int objs[M];
Conjunto bolsa;

void seleccion(int obj, int candidato, int suma, bool& encontrado)
{
  /* obj       : peso objetivo
     candidato : índice del peso a añadir
     suma      : suma parcial de pesos      */
  encontrado = false;
  while ((candidato < n) && !encontrado)
  {
    candidato++;
    if ((suma + objs[candidato - 1]) <= obj)
    {
      bolsa.insertar(candidato);          // objeto anotado
      suma += objs[candidato - 1];
      if (suma < obj)                     // ensaya con siguiente objeto
      {
        seleccion(obj, candidato, suma, encontrado);
        if (!encontrado)                  // vuelta atrás, se extrae objeto
        {
          bolsa.retirar(candidato);
          suma -= objs[candidato – 1];
        }
      }
```

```
      else
        encontrado = true;
    }
  }
}

void escribirSeleccion()
{
  cout  << "\nSelección de objetos: ";
  for (int candidato = 1; candidato <= n; candidato++)
  {
    if (bolsa.pertenece(candidato))
    {
      cout  << " Objeto " << candidato
            << " peso: " << objs[candidato - 1] << endl;
      bolsa.retirar(candidato);
    }
  }
}
```

Todas las soluciones

Para encontrar todas las combinaciones de objetos que totalizan un peso *v*, cada vez que la suma de pesos de una selección sea igual al peso *v* se escriben los objetos que la forman.

El problema se resuelve modificando la función `seleccion()`. La variable `encontrado` ya no es necesaria, el bucle tiene que iterar con todos los objetos, cada vez que `suma` sea igual al peso objetivo, `obj`, se llama a `escribirSeleccion()`. Ahora, en la *vuelta atrás* siempre se borra de la bolsa la anotación realizada, para así probar con los `n` objetos.

```
void seleccion(int obj, int candidato, int suma)
{

  while (candidato < n)
  {
    candidato++;
    if ((suma + objs[candidato - 1]) <= obj)
    {
      bolsa.insertar(candidato);       // objeto anotado
      suma += objs[candidato - 1];
      if (suma < obj)                 // ensaya con siguiente objeto
      {
        seleccion(obj, candidato, suma);
      }
      else
        escribirSeleccion();          // vuelta atrás, se extrae el objeto
      bolsa.retirar(candidato);
      suma -= objs[candidato – 1];

    }
  }
}
```

7.7. SELECCIÓN ÓPTIMA

Los problemas que se resuelven aplicando el esquema de *selección óptima* no persiguen encontrar una situación fija o un valor predeterminado, sino encontrar, del conjunto de todas las soluciones la *óptima*, según unas condiciones que establecen qué es lo óptimo. Por tanto, hay que probar con todas las posibilidades que existen de realizar una nueva tarea para encontrar de entre todas las soluciones la que cumpla una segunda condición o requisito, la *solución óptima.*

El esquema general del algoritmo para encontrar la *selección óptima*:

```
procedimiento ensayarSeleccion(objeto i)
inicio
  <inicializar cuenta de posibles selecciones>
  repetir
    <tomar siguiente selección (tarea)>
    <determinar si selección es valida>
    si válido entonces
      <anotar selección>
      si  <es solución>  entonces
        si <mejor(solución)> entonces
          <guardar selección>
        fin_si
      fin-si

      ensayarSeleccion(objeto i+1)
     <borrar anotación> {el bucle se encarga de probar con otra selección}
    fin_si
  hasta (<no más posibilidades>)
fin
```

La selección óptima implica probar con todas los elementos disponibles, para seleccionar, entre todas las configuraciónes que cumplan una primera condición, la más próxima a una segunda condición.

Nota de eficiencia

Estos algoritmos son de *complejidad exponencial* debido a que el número de llamadas recursivas crece exponencialmente. Por ello, es importante evitar llamadas improductivas; cuando se sabe que la selección actual no mejor ará la selección óptima no se realiza la llamada recursiva (*poda de las ramas en el árbol de llamadas*).

7.7.1. Programa del viajante

El ejemplo del problema del viajante es el que mejor explica la *selección óptima*:

> *"el viajante (piense en un representante de joyería) tiene que confeccionar la maleta, seleccionando entre n artículos, aquéllos cuyo valor sea máximo (lo óptimo es que la*

suma de valores sea máximo) y su peso no exceda de una cantidad, la que puede sustentar la maleta".

Por ejemplo, la maleta de seguridad para llevar joyas es capaz de almacenar 1,5 Kg y se tiene los objetos, representados por un par `(peso, valor)`: `(115gr, 100€), (90gr, 110€), (50gr, 60€), (120gr, 110€) ...` . Se pretende hacer una selección, que no sobrepase los 1,5 Kg y que la suma de *valor* sea máximo.

Para resolver el problema se generan todas las selecciones posibles con los `n` objetos disponibles, cada selección debe cumplir la condición de no superar el peso máximo prefijado; cada vez que se alcance una seleción se comprueba si es mejor que cualquiera de las anteriores, en caso positivo se guarda como actual *mejor seleción*. En definitiva, una *búsqueda exhaustiva*, un *tanteo sistemático,* con los `n` objetos del problema; cada tanteo realiza la tarea de probar si el incluir el objeto *i* va a dar lugar a una mejor selección; y también prueba si la exclusión del objeto *i* dará lugar a una *mejor selección*.

```
si solucion entonces
  si mejor(solucion) entonces
     optimo = solucion
```

Tarea básica

El objetivo del problema es que el *valor* que se pueda conseguir sea máximo, por esa razón se considera que el *valor* más alto es la suma de los valores de cada objeto; posteriormente, al excluir un objeto de la selección se ha de restar el valor que tiene.

La tarea básica en esta *búsqueda sistemática* es analizar si es adecuado incluir un objeto *i;* será adecuado si el peso acumulado más el del objeto *i* no supera al peso de la maleta. En el caso de que sea necesario excluir al objeto *i* (*vuelta atrás* de las llamadas recursivas) de la selección actual, el criterio para seguir con el proceso de selección es que el valor total todavía alcanzable, después de esta exclusión, no sea menor que el valor óptimo encontrado hasta ese momento. La inclusión de un objeto supone incrementar el peso de la selección actual en el peso del objeto. A su vez, excluir un objeto de la selección supone que el valor alcanzable por la selección tiene que ser decrementado en el valor del objeto excluido. Cada tarea realiza las mismas acciones que la tarea anterior, por ello se expresa recursivamente.

La prueba de si es mejor selección se hace ensayando con todos los objetos de que dispone, una vez que se alcance el último es cuando se determina si el valor asociado a la selección es el mejor, el óptimo.

> ***A tener en cuenta***
>
> En el proceso de *selección óptima* se prueba con la inclusión de un objeto `i` y con la exclusión de `i`. Para hacer más eficiente el algoritmo se hace una *poda* de aquellas selecciones que no van a ser mejores; por ello antes de probar con la exclusión se determina si la selección en curso puede alcanzar un mejor valor, en el caso de que no ¿para qué ir por esa rama si no se va a conseguir una mejor selección?

Representación de los datos

Se supone que el viajante tiene *n* objetos. Cada objeto tiene asociado el par: `<peso, valor>`, por ello sendos arrays almacenan los datos de los objetos. La selección actual y la óptima van

a tratarse como dos conjuntos de objetos; realmente, los conjuntos incluirán únicamente el índice del objeto.

Codificación

El valor máximo que pueden alcanzar los objetos es la suma de los valores de cada uno, está representado por la variable `totalValor`. El valor óptimo alcanzado durante el proceso está en la variable `mejorValor`.

Los parámetros de la función recursiva son los necesarios para realizar una nueva tarea: `i` número del objeto a probar su inclusión, `pt` peso de la selección actual, `va` valor máximo alcanzable por la selección actual y `mejorValor` que es el actual valor máximo.

```
const int M = 12;                              // máximo número de objetos
int n;                                         // número de objetos
int pesoObjs[M];
int valorObjs[M];
Conjunto actual, optimo;
int pesoMaximo;

void probarObjeto(int i, int pt, int va, int& mejorValor)
{
  int valExclusion;

  if (pt + pesoObjs[i - 1] <= pesoMaximo)    // objeto i se incluye
  {
    actual.insertar(i);                      //  se anota
    if (i < n)
      probarObjeto(i + 1, pt + pesoObjs[i - 1], va, mejorValor);
    else                                     // todos los objetos probados
      if (va > mejorValor)
      {                                      // es una mejor selección
        optimo = actual;
        mejorValor = va;
      }
      actual.retirar(i);                     // vuelta atrás, ensaya la
                                             //exclusión de obj i
          // exclusión del objeto i, sigue la búsqueda sistemática con i+1
      valExclusion = va – valorObjs[i - 1];  // decrementa valor del
                                                objeto
  if (valExclusion > mejorValor)  /* se puede alcanzar una mejor
                                     selección, sino poda la búsqueda */
    if (i < n)
      probarObjeto(i + 1, pt, valExclusion, mejorValor);
    else
    {
      optimo = actual;
      mejorValor = valExclusion;
    }
}

void escribirOptimo(int mejor)
{

  cout << " \tObjetos que forman la selección óptima" << endl;
```

```
  for (int j = 1; j <= n; j++)
  {
    if (optimo.pertenece(j))
    {
      cout  << " Objeto " << j  << " peso: " << pesoObjs[j - 1]
            << " , valor: " << valorObjs[j - 1] << endl;
      bolsa.retirar(candidato);
    }

  }
  cout << "\t Valor óptimo = << mejor
       << " para un peso máximo = " << pesoMaximo << endl;
}

//Llamada desde main()

  // Suma de los valores de cada objeto
  for (maxValor = i = 0; i < n; i++)
    maxValor += valorObjs[i];
  mejorValor = 0;
  probarObjeto(1, 0, maxValor, mejorValor);

  escribirOptimo(mejorValor);
```

RESUMEN

Una función se dice que es recursiva si tiene una o más sentencias que son llamadas a sí misma. La recursividad puede ser directa o indirecta, ésta ocurre cuando la función `f()` llama a `p()` y ésta a su vez llama a `f()`. La recursividad es una alternativa a la iteración en la resolución de algunos problemas matemáticos, aunque en general es preferible la implementación iterativa debido a que es más eficiente. Los aspectos más importantes a tener en cuenta en el diseño y construcción de funciónes recursivas son los siguientes:

- Un algoritmo recursivo correspondiente con una función normalmente contiene dos tipos de casos: uno o más casos que incluyen al menos una llamada recursiva y uno o más casos de terminación o parada del problema en los que éste se soluciona sin ninguna llamada recursiva sino con una sentencia simple. De otro modo, un algoritmo recursivo debe tener dos partes: una parte de terminación en la que se deja de hacer llamadas, es el caso base, y una llamada recursiva con sus propios parámetros.
- Muchos problemas tienen naturaleza recursiva y la solución más fácil es mediante una función recursiva. De igual modo aquellos problemas que no entrañen una solución recursiva se deberán seguir resolviendo mediante algoritmos iterativos.
- Las funciónes con llamadas recursivas utilizan memoria extra en las llamadas; existe un límite en las llamadas, que depende de la memoria de la computadora. En caso de superar este límite ocurre un error de *overflow* (desbordamiento).
- Cuando se codifica una función recursiva se debe comprobar siempre que tiene una condición de terminación; es decir, que no se producirá una recursión infinita. Durante el aprendizaje de la recursividad es usual que se produzca ese error.
- Para asegurarse de que el diseño de una función recursiva es correcto se deben cumplir las siguientes tres condiciones:

1. No existir recursión infinita. Una llamada recursiva puede conducir a otra llamada recursiva y ésta conducir a otra, y así sucesivamente; pero cada llamada debe de aproximarse más a la condición de terminación.
2. Para la condición de terminación, la función devuelva el valor correcto para ese caso.
3. En los casos que implican llamadas recursivas: si cada una de las funciónes devuelve un valor correcto, entonces el valor final devuelto por la función es el valor correcto.

- Una de las técnicas más utilizadas en la resolución de problemas es la denominada *"divide y vence"*. La implementación de estos algoritmos se puede realizar con funciónes recursivas.
- Los algoritmos del tipo *vuelta atrás* o *backtracking* se caracterizan por realizar una búsqueda sistemática, exhaustiva de la solución. Prueba con todas posibilidades que se tiene para realizar una tarea que vaya encaminada hacia la solución. Cada tarea se expresa por una llamada recursiva, los elementos de la tarea se apuntan (se guardan). En la *vuelta atrás*, retorno de la llamada recursiva, se determina si se alcanzó la solución y, en caso contrario, se borra la anotación para probar de nuevo con otra posibilidad.
- La selección óptima se puede considerar como una variante de los algoritmos de *vuelta atrás*, en la que se busca no sólo los elementos que cumplen una condición sino que éstos sean los mejores según el criterio que se haya establecido como *mejor*.

EJERCICIOS

7.1. Convierta la siguiente función iterativa en recursiva. La función calcula un valor aproximado de *e*, base de los logaritmos naturales, sumando la serie

1 + 1/1! + 1/2! + ... + 1/n!

hasta que los términos adicionales no afecten a la aproximación

```
double loge()
{
  double enl, delta, fact;
  int n;
  enl = fact = delta = 1.0;
  n = 1;
  do
  {
    enl += delta;
    n++;
    fact * = n:
    delta = 1.0 / fact;
  } while (enl != enl + delta);
  return enl;
}
```

7.2. Explique las razones por las cuales la siguiente función puede producir un valor incorrecto cuando se ejecute:

```
long factorial (long n)
{
  if (n == 0 || n == 1)
    return 1;
```

```
  else
    return n * factorial (--n);
}
```

7.3. ¿Cuál es la secuencia numérica generada por la función recursiva `f()`si la llamada es `f(5)`?

```
long f(int n)
{
  if (n == 0 || n == 1)
    return 1;
  else
    return 3 * f(n - 2) + 2 * f(n - 1);
}
```

7.4. Escribir una función recursiva `int vocales(char * cd)` para calcular el número de vocales de una cadena.

7.5. Proporcionar funciones recursivas que representen los siguientes conceptos:

a) El producto de dos números naturales.
b) El conjunto de permutaciones de una lista de números.

7.6. Suponer que la función matemática G está definida recursivamente de la siguiente forma:

$$G(x, y) = \begin{cases} \dfrac{1}{G(x - y + 1, y)} & si \quad x < y \\ 2x - y & si \quad x < y \end{cases}$$

siendo `x`, `y` enteros positivos. Encontrar el valor de: (*a*) G(8,6); (*b*) G(100, 10).

7.7. Escribir un programa recursivo que calcule la función de Ackermann definida de la siguiente forma:

$A(m, n) = n + 1 \quad si \quad m = 0$
$A(m, n) = A(m - 1, 1) \quad si \quad m > 0, y\ n = 0$
$A(m, n) = A(m - 1, A(m, n - 1)) \quad si \quad m > 0, y\ n > 0$

7.8. ¿Cuál es la secuencia numérica generada por la función recursiva siguiente, si la llamda es `f(8)`?

```
long f (int n)
{
  if(n == 0||n ==1)
    return 1;
  else if (n % 2 == 0)
    return 2 + f(n – 1);
  else
    return 3 + f(n – 2);
}
```

7.9. ¿Cuál es la secuencia numérica generada por la función recursiva siguiente?

```
int f(int n)
{
  if (n == 0)
    return 1;
```

```
  else if (n == 1)
    return 2;
  else
    return 2*f(n – 2) + f(n – 1);
}
```

7.10. El elemento mayor de un array entero de *n-elementos* se puede calcular recursivamente. Suponiendo que la función:

```
int max(int x, int y);
```

devuelve el mayor de dos enteros x e y. Definir la función

```
int maxarray(int a[], int n);
```

que utiliza recursión para devolver el elemento mayor de *a*

Condición de parada: `n == 1`
Incremento recursivo: `maxarray = max(max(a[0]...a[n-2]),a[n-1])`

7.11. Escribir una función recursiva,

```
int product(int v[], int b);
```

que calcule el producto de los elementos del array *v* mayores que `b`.

7.12. El Ejercicio 7.6 define recursivamente una función matemática. Escribir una función que no utilize la recursividad para encontrar valores de la función.

7.13. La resolución recursiva de las torres de hanoi ha sido realizada con dos llamadas recursivas. Volver a escribir la solución con una sola llamada recursiva.

Nota: Sustituir la última llamada por un bucle *repetir-hasta* (`repeat-until`).

7.14. Aplicar el esquema de los algoritmos *divide y vence* para que dados las coordenadas (x,y) de dos puntos en el plano, que representan los extremos de un segmento, se dibuje el segmento.

7.15. Escribir una función rerecursiva para transformar un número entero en una cadena con el signo y los dígitos de que consta: `string entroaCadena(int n);`

PROBLEMAS

7.1. La expresión matemática *C(m, n)* en el mundo de la teoría combinatoria de los números, representa el número de combinaciones de *m* elementos tomados de *n* en *n* elementos

$$C(m, n) = \frac{m!}{n!(m - n)!}$$

Escribir un programa que dé entrada a los enteros *m, n* y calcule *C (m, n)* donde *n!* es el factorial de *n*.

7.2. Un palíndromo es una palabra que se escribe exactamente igual leída en un sentido o en otro. Palabras tales como *level, deed, ala*, etc., son ejemplos de palíndromos. Aplicar un algoritmo

divide y vence para determinar si una palabra es palíndromo. Escribir una función recursiva que implemente el algoritmo. Escribir un programa que lea una cadena hasta que ésta sea un palíndromo.

7.3. Escribir un programa en la que una función recursiva liste todos los subconjuntos de `n` letras para un conjunto dado de `m` letras, por ejemplo para m = 4 y n = 2.

`[A, C, E, K]` → `[A, C], [A, E], [A, K], [C, E,], [C, K], [E, K]`

Nota: el número de subconjuntos es $C_{4,2}$.

7.4. El problema de las 8 reinas se ha resuelto en este capítulo de tal forma que en el momento de encontrar una solución se detiene la búsqueda de más soluciones. Modificar el algoritmo de tal forma que la función recursiva escriba todas las soluciones.

7.5. Escribir un programa que tenga como entrada una secuencia de números enteros positivos (mediante una variable entera). El programa debe de hallar la suma de los dígitos de cada entero y encontrar cuál es el entero cuya suma de dígitos es mayor. La suma de dígitos ha de ser con una función recursiva.

7.6. El problema de las Torres de Hanoi se ha resuelto en el Apartado 7.4.1, aplicando un algoritmo recursivo que sigue la estrategia *divide y vencerás*. Resolver el problema aplicando un esquema iterativo.

7.7. Desarrolar un programa que lea un número entero positivo $n < 10$ y calcule el desarrollo del polinomio $(x + 1)^n$. Imprimir cada potencia x^i de la forma `x**i`.

Sugerencia:

$(x + 1)^n = C_{n,n}x^n + C_{n,n-1}x^{n-1} + C_{n,n-2}x^{n-2} + ... + C_{n,2}x^2 + C_{n,1}x^1 + C_{n,0}x^0$

La relación de recurrencia de los coeficientes binomiales C_{nk} es:

C(n, 0) = 1
C(n, n) = 1
C(n, k) = C(n-1, k-1) + C(n-1, k)

Estos coeficientes constituyen el famoso *Triángulo de Pascal* y será preciso definir la función que genera el triángulo

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
...
```

7.8. Sea `A` una matriz cuadrada de `n x n` elementos, el determinante de `A` podemos definirlo de manera recursiva:

a) Si `n = 1` entonces `Deter(A) =` $a_{1,1}$.
b) Para `n > 1`, el determinante es la suma alternada de productos de los elementos de una fila o columna elegida al azar por sus menores complementarios. A su vez, los menores complementarios son los determinantes de orden `n-1` obtenidos al suprimir la fila y columna en que se encuentra el elemento.

Puede expresarse:

$$\texttt{Det(A)} = \sum_{i=1}^{n} (-1)^{i+j} * A[i,j] * \texttt{Det(Menor(A[i,j]))};$$ para cualquier columna `j`

o

$$\texttt{Det(A)} = \sum_{i=1}^{n} (-1)^{i+j} * A[i,j] * \texttt{Det(Menor(A[i,j]))};$$ para cualquier fila `i`

Se observa que la resolución del problema sigue la estrategia de los algoritmos *divide y vence*.

Escribir un programa que tenga como entrada los elementos de la matriz `A`, y tenga como salida la matriz `A` y el determinante de `A`. Eligiendo la fila 1 para calcular el determinante.

7.9. Escribir un programa que transforme números enteros en base 10 a otro en base b. Siendo la base b de 8 a 16. La transformación se ha de realizar siguiendo una estrategia recursiva.

7.10. Escribir un programa para resolver el problema de la subsecuencia creciente más larga. La entrada es una secuencia de n números $a_1, a_2, a_3, \ldots, a_n$; hay que encontrar la subsecuencia más larga $a_{i1}, a_{i2}, \ldots a_{ik}$ tal que $a_{i1} < a_{i2} < a_{i3} \ldots < a_{ik}$ y que $i1 < i2 < i3 < \ldots < ik$. El programa escribirá tal subsecuencia.

Por ejemplo, si la entrada es `3,2,7,4,5,9,6,8,1`, la subsecuencia creciente más larga tiene longitud cinco: `2,4,5,6,7`.

7.11. El sistema monetario consta de monedas de valor $p_1, p_2, p_3, \ldots, p_n$ (orden creciente) euros (o dólares). Escribir un programa que tenga como entrada el valor de las n monedas, en orden creciente, y una cantidad `x` de cambio. Calcule:

a) El número mínimo de monedas que se necesitan para dar el cambio `x`.
b) Calcule el número de formas diferentes de dar el cambio de la cantidad `x` con la p_i monedas.

Aplicar técnicas recursivas para resolver el problema.

7.12. En un tablero de ajedrez, se coloca un alfil en la posición (x_0, y_0) y un peón en la posición `(1,j)`, siendo `1 <= j <= 7`. Se pretende encontrar una ruta para el peón que llegue a la fila 8 sin ser comido por el alfil. Siendo el único movimiento permitido para el peón el de avance desde la posición `(i,j)` a la posición `(i+1,j)`. Si se encuentra que el peón está amenazado por el alfil en la posición `(i,j)`, entonces debe de retroceder a la fila 1, columna `j+1` o `j-1` `{(1,j+1), (1,j-1)}`.

Escribir un programa para resolver el supuesto problema. Hay que tener en cuenta que el alfil ataca por diagonales.

7.13. Dados n números enteros positivos encontrar una combinación de ellos que mediante sumas o restas totalicen exactamente un valor objetivo `Z`. El programa debe tener como entrada los n números y el objetivo `Z`; la salida ha de ser la combinación de números con el operador que le corresponde.

Tener en cuenta que pueden formar parte de la combinación los n números o parte de ellos.

7.14. Dados n números encontrar combinación con sumas o restas que más se aproxime a un objetivo Z. La aproximación puede ser por defecto o por exceso. La entrada son los n números y el objetivo y la salida la combinación más próxima al objetivo.

7.15. Un laberinto podemos emularlo con una matriz `n x n` en la que los pasos libres están representados por un carácter (el blanco por ejemplo) y los muros por otro carácter (el por ejemplo). Escribir un programa en el que se genere aleatoriamente un laberinto, se pide las coordenadas de entrada (la fila será la 1), las coordenadas de salida (la fila será la `n`) y encontrar todas las rutas que nos llevan de la entrada a la salida.

7.16. Realizar las modificaciones necesarias en el problema del laberinto 7.15 para encontrar la ruta más corta. Considerando ruta más corta la que pasa por un menor número de casillas.

7.17. Una región castellana está formada por n pueblos dispersos. Hay conexiones directas entre algunos de estos pueblos y entre otros no existe conexión aunque puede haber un camino. Escribir un programa que tenga como entrada la matriz que representa las conexiones directas entre pueblos, de tal forma que el elemento `M(i,j)` de la matriz sea:

$$M(i, j) = \begin{cases} 0 & \text{si no hay conexión directa entre pueblo i y pueblo j.} \\ d & \text{hay conexión entre pueblo i y pueblo j de distancia d.} \end{cases}$$

También tenga como entrada un par de pueblos `(x,y)`. El programa tiene que encontrar un camino entre ambos pueblos utilizando técnicas recursivas. La salida ha de ser la ruta que se ha de seguir para ir de `x` a `y` junto a la distancia de la ruta.

7.18. En el programa escrito en 7.17 hacer las modificaciones necesarias para encontrar todos los caminos posibles entre el par de pueblos `(x,y)`.

7.19. Un número entero sin signo, m, se dice que es *dos_tres_cinco*, si cumple las características:

- Todos los dígitos de m son distintos.
- La suma de los dígitos que ocupan posiciones pares, es igual a la suma de los dígitos que ocupan posiciones múltiplos de tres más la suma de los dígitos que ocupan posiciones múltiplos de cinco.

Implementar un programa que genere todos los números enteros de cinco o más cifras que séan *dos_tres_cinco.*

CAPÍTULO 8

Algoritmos de ordenación y búsqueda

Objetivos

Una vez que se haya leído y estudiado este capítulo usted podrá:

- Conocer los algoritmos basados en el intercambio de elementos.
- Conocer el algoritmo de ordenación por inserción.
- Conocer el algoritmo de selección.
- Distinguir entre los algoritmos de ordenación basados en el intercambio y en la inserción.
- Saber la eficiencia de los métodos básicos de ordenación.
- Conocer los métodos más eficientes de ordenación.
- Diferenciar entre búsqueda secuencial y búsqueda binaria.

Contenido

Conceptos clave

- Búsqueda binaria.
- Búsqueda secuencial.
- Complejidad cuadrática.
- Complejidad logarítmica.
- Ordenación alfabética.
- Ordenación por burbuja.
- Ordenación numérica.
- Ordenación por intercambio.
- Ordenación por inserción.
- Ordenación por selección.
- Ordenación rápida.
- Residuos.

INTRODUCCIÓN

Muchas actividades humanas requieren que a diferentes colecciones de elementos utilizados se pongan en un orden específico. Las oficinas de correo y las empresas de mensajería ordenan el correo y los paquetes por códigos postales con el objeto de conseguir una entrega eficiente; las facturas telefónicas se ordenan por la fecha de las llamadas; los anuarios o listines telefónicos se ordenan por orden alfabético de apellidos con el fin último de encontrar fácilmente el número de teléfono deseado. Los estudiantes de una clase en la universidad se ordenan por sus apellidos o por los números de expediente. Por esta circunstancia una de las tareas que realizan más frecuentemente las computadoras en el procesamiento de datos es la *ordenación*.

El estudio de diferentes métodos de ordenación es una tarea intrínsecamente interesante desde un punto de vista teórico y, naturalmente, práctico. El capítulo estudia los algoritmos y técnicas de ordenación más usuales y su implementación en C++. Además, se analizan los diferentes métodos de ordenación con el objeto de conseguir la máxima eficiencia en su uso real.

8.1. ORDENACIÓN

La **ordenación** o **clasificación** de datos (*sort* en inglés) es una operación consistente en disponer un conjunto —estructura— de datos en algún determinado orden con respecto a uno de los *campos* de los elementos del conjunto. Por ejemplo, cada elemento del conjunto de datos de una guía telefónica tiene un campo nombre, un campo dirección y un campo número de teléfono; la guía telefónica está dispuesta en orden alfabético de nombres. Los elementos numéricos se pueden ordenar en orden creciente o decreciente de acuerdo al valor numérico del elemento. En terminología de ordenación, el elemento por el cual está ordenado un conjunto de datos (o se está buscando) se denomina *clave*.

Una colección de datos *(estructura)* puede ser almacenada en memoria central o en archivos de datos externos guardados en unidades de almacenamiento magnético (discos, cintas, CD-ROM, DVD, etc.). Cuando los datos se guardan en memoria principal —un *array,* una *lista enlazada* o un *árbol*— se denomina *ordenación interna;* estos datos se almacenan exclusivamente para tratamientos internos que se utilizan para gestión masiva de datos y se guardan en arrays de una o varias dimensiones. Si los datos están almacenados en un archivo, el proceso de ordenación se llama *ordenación externa*. Este capítulo estudia los métodos de *ordenación interna.*

Una *lista está ordenada por la clave k* si la lista está en orden ascendente o descendente con respecto a esta clave. La lista está en *orden ascendente* si:

```
i < j       implica que       k[i] <= k[j]
```

y está en *orden descendente* si:

```
i > j       implica que       k[i] <= k[j]
```

para todos los elementos de la lista. Por ejemplo, para una guía telefónica, la lista está clasificada en orden ascendente por el campo clave *k*, siendo *k* el nombre del abonado (apellidos, nombre).

```
4   5   14  21  32  45                          orden ascendente
75  70  35  16  14  12                          orden descendente

Zacarias Rodriguez Martinez Lopez Garcia        orden descendente
```

Los métodos (algoritmos) de ordenación son numerosos, por ello se debe prestar especial atención en su elección. ¿Cómo se sabe cuál es el mejor algoritmo? La *eficiencia* es el factor que mide la calidad y rendimiento de un algoritmo. En el caso de la operación de ordenación, dos criterios se suelen seguir a la hora de decidir qué algoritmo —de entre los que resuelven la ordenación— es el más eficiente: 1) *tiempo menor de ejecución en computadora;* 2) *menor número de instrucciones.* Sin embargo, no siempre es fácil efectuar estas medidas: puede no disponerse de instrucciones para medida de tiempo, y las instrucciones pueden variar, dependiendo del lenguaje y del propio estilo del programador. Por esta razón, el mejor criterio para medir la eficiencia de un algoritmo es aislar una operación específica clave en la ordenación y contar el número de veces que se realiza. Así, en el caso de los algoritmos de ordenación, se utilizará como medida de su eficiencia el número de comparaciones entre elementos efectuados. El algoritmo de *ordenación A* será más eficiente que el *B,* si requiere menor número de comparaciones. En la ordenación de los elementos de un array, el número de comparaciones será *función* del número de elementos *(n)* del array. Por consiguiente, se puede expresar el número de comparaciones en términos de *n*.

Todos los métodos de este capítulo, normalmente —para comodidad del lector— se *ordena de modo ascendente* sobre listas (arrays unidimensionales). Se suelen dividir en dos grandes grupos:

- *Directos* burbuja, selección, inserción.
- *Indirectos* (avanzados) shell, ordenación rápida, ordenación por mezcla, radixsort.

En el caso de listas pequeñas, los métodos directos se muestran eficientes, sobre todo porque los algoritmos son sencillos; su uso es muy frecuente. Sin embargo, en listas grandes estos métodos se muestran ineficaces y es preciso recurrir a los métodos avanzados.

> Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*. Los métodos de ordenación se conocen como *internos* o *externos* según que los elementos a ordenar estén en la memor ia principal o en la memoria externa.

8.2. ALGORITMOS DE ORDENACIÓN BÁSICOS

Existen diferentes algoritmos de ordenación elementales o básicos cuyos detalles de implementación se pueden encontrar en diferentes libros de algoritmos. La enciclopedia de referencia es [Knuth 1973][1] y sobre todo la 2.ª edición publicada en el año 1998 [Knuth 1998][2]. Los algoritmos presentan diferencias entre ellos que los convierten en más o menos eficientes y prácticos según sea la rapidez y eficiencia demostrada por cada uno de ellos. Los algoritmos básicos de ordenación más simples y clásicos son:

- Ordenación por selección.
- Ordenación por inserción.
- Ordenación por burbuja.

---

[1] [Knuth 1973] Donald E. Knuth. *The Art of Computer Programming*. Volume 3: *Sorting and Searching*. Addison-Wesley, 1973.

[2] [Knuth 1998] Donald E. Knuth. *The Art of Computer Programming*. Volume 3: *Sorting and Searching*. Second Edition. Addison-Wesley, 1998.

Los métodos más recomendados son el de *selección* y el de *inserción*, aunque se estudiará el método de *burbuja*, por aquello de ser el más sencillo aunque a la par también es el más *ineficiente*; por esta causa no se recomienda su uso, pero sí conocer su técnica.

Con el objeto de facilitar el aprendizaje del lector y aunque no sea un método utilizado por su poca eficiencia, se describe en primer lugar el método de *ordenación por intercambio*, debido a la sencillez de su técnica y con el objetivo de que el lector no introducido en los algoritmos de ordenación pueda comprender su funcionamiento y luego asimile más eficazmente los tres algoritmos básicos ya citados y los avanzados que se estudian más adelante.

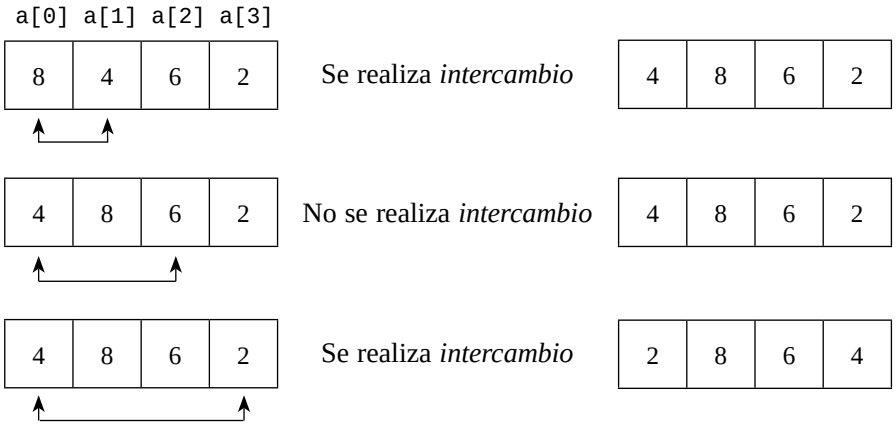
8.3. ORDENACIÓN POR INTERCAMBIO

El algoritmo se basa en la lectura sucesiva de la lista a ordenar, comparando el elemento inferior de la lista con los restantes y efectuando intercambio de posiciones cuando el orden resultante de la comparación no sea el correcto.

Los pasos que sigue el algoritmo se muestran al aplicarlo a la lista original 8, 4, 6, 2 que ha de convertirse en la lista ordenada 2, 4, 6, 8. El algoritmo efectua $n - 1$ pasadas (3 en el ejemplo), siendo n el número de elementos.

Pasada 1

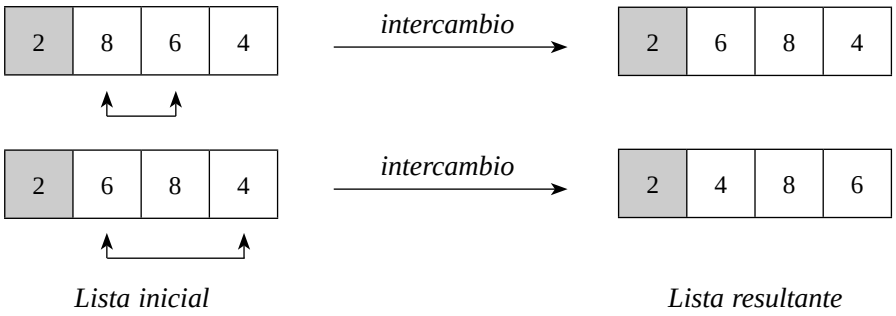
El elemento de índice `0 (a[0])` se compara con cada elemento posterior de la lista, de índices `1, 2` y `3`. Cada comparación comprueba si el elemento siguiente es más pequeño que el elemento de índice `0`, en cuyo caso se intercambian. Después de terminar todas las comparaciones, el elemento más pequeño se sitúa en el índice o.



Pasada 2

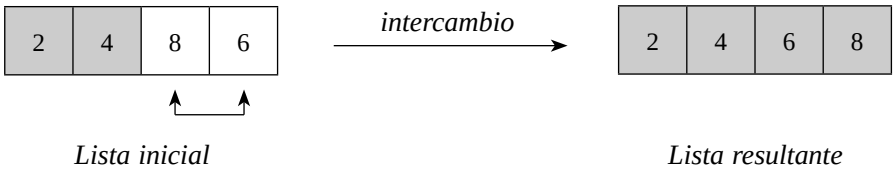
El elemento más pequeño ya está en la posición de índice `0`, ahora se considera la sublista restante 8, 6, 4. El algoritmo continúa comparando el elemento de índice `1` con los elementos posteriores de índices `2` y `3`. Por cada comparación, si el elemento mayor está en el índice `1`

se intercambian los elementos. Después de hacer todas las comparaciones, el segundo elemento más pequeño de la lista se almacena en el índice `1`.



Pasada 3

Ahora la sublista a considerar es 8, 6 ya que 2, 4 están ordenads. Una comparación única se produce entre los dos elementos de la sublista.



8.3.1. Codificación del algoritmo de ordenación por intercambio

El método `ordIntercambio()` implementa el algoritmo descrito, para ello utiliza dos bucles anidados. Separa una lista de tamaño *n*, el rango del bucle externo va de `0` a `n – 2`. Por cada índice *i*, se comparan los elementos posteriores de índices *j = i + 1*, *i + 2* ,..., *n – 1*. El intercambio *(swap)* de dos elementos `a[i], a[j]` se realiza en esta función:

```
void intercambiar(int& x, int& y)
{
  int aux = x;
  x = y;
  y = aux;
}
```

Se supone que se ordena un *array* de n enteros:

```
void ordIntercambio (int a[], int n)
{
  int i, j;

  for (i = 0; i < n - 1; i++)
    // sitúa mínimo de a[i+1]...a[n-1] en a[i]
    for (j = i + 1; j < n; j++)
```

```
        if (a[i] > a[j])
        {
          intercambiar(a[i], a[j]);
        }
}
```

8.3.2. Complejidad del algoritmo de ordenación por intercambio

El algoritmo consta de dos bucles anidados, está *dominado* por los dos bucles. De ahí, que el análisis del algoritmo en relación a la complejidad sea inmediato, siendo `n` el número de elementos, el primer bucle hace `n-1` pasadas y el segundo `n-i-1` comparaciones en cada pasada (`i` es el índice del bucle externo, `i = 0 .. n-2`). El número total de comparaciones se obtiene desarrollando la sucesión matemática formada para los distintos valores de `i`:

```
n-1, n-2, n-3, ... , 1
```

El número de comparaciones se obtiene sumando los términos de la sucesión: $\frac{(n-1)\cdot n}{2}$

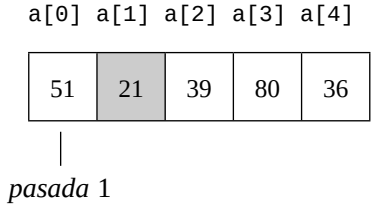
y un número similar de intercambios *en el peor de los casos*. Entonces, el número de comparaciones y de intercambios *en el peor de los casos* es $(n-1)\cdot n/2 = (n^2-n)/2$. El término dominante es n^2, por tanto la complejidad es $0(n^2)$.

8.4. ORDENACIÓN POR SELECCIÓN

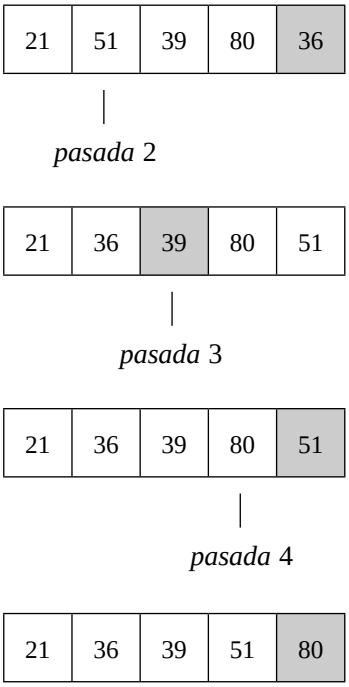
Considérese el algoritmo para ordenar un array `a[]` en orden ascendente; es decir, si el array tiene *n* elementos, se trata de ordenar los valores del array de modo que `a[0]` sea el valor más pequeño, el valor almacenado en `a[1]` el siguiente más pequeño, y así hasta `a[n-1]` que ha de contener el elemento mayor. El algoritmo de selección realiza *pasadas* que intercambian el elemento más pequeño sucesivamente, con el elemento del array que ocupa la posición igual al orden de *pasada* (hay que considerar el índice `0`).

La *pasada* inicial busca el elemento más pequeño de la lista y se intercambia con `a[0]`, primer elemento de la lista. Después de terminar esta primera pasada, el frente de la lista está ordenado y el resto de la lista `a[1], a[2] ... a[n-1]` permanece desordenada. La siguiente *pasada* busca en esta lista desordenada y *selecciona* el elemento más pequeño, que se almacena en la posición `a[1]`. De este modo los elementos `a[0]` y `a[1]` están ordenados y la sublista `a[2], a[3]...a[n-1]` desordenada. El proceso continúa hasta realizar `n-1` *pasadas*, en ese momento la lista desordenada se reduce a un elemento (el mayor de la lista) y el *array* completo ha quedado ordenado.

El siguiente ejemplo práctico ayudará a la comprensión del algoritmo:



Pasada 1*: Seleccionar* `21`
Intercambiar `21` y `a[0]`



Pasada 2: Seleccionar `36`
Intercambiar `36` y `a[1]`

Pasada 3: Seleccionar `39`
Intercambiar `39` y `a[2]`

Pasada 4: Seleccionar `51`
Intercambiar `51` y `a[3]`

Array ordenado

8.4.1. Codificación del algoritmo de *selección*

La función `ordSeleccion()` ordena un array de números reales de `n` elementos. El proceso de selección explora, en la pasada `i`, la sublista `a[i]` a `a[n-1]` y fija el índice del elemento más pequeño. Después de terminar la exploración, los elementos `a[i]` y `a[indiceMenor]` se intercambian.

```
  /*
      ordenar un array de n elementos de tipo double
      utilizando el algoritmo de ordenación por selección
  */

void ordSeleccion (double a[], int n)
{
  int indiceMenor, i, j
         // ordenar a[0]..a[n-2] y a[n-1] en cada pasada
  for (i = 0; i < n - 1; i++)
  {
     // comienzo de la exploración en índice i
     indiceMenor = i;
                    // j explora la sublista a[i+1]..a[n-1]
     for (j = i + 1; j < n; j++)
       if (a[j] < a[indiceMenor])
          indiceMenor = j;
     // sitúa el elemento mas pequeño en a[i]
     if (i != indiceMenor)
```

```
        intercambiar(a[i], a[indiceMenor]);
    }
  }

  void intercambiar(double& x, double& y)
  {
    double aux = x;
    x = y;
    y = aux;
  }
```

8.4.2. Complejidad del algoritmo de *selección*

El análisis del algoritmo, con el fin de determinar la función *tiempo de ejecución* `t(n)`, es sencillo y claro, ya que requiere un número fijo de comparaciones que sólo dependen del tamaño del *array* y no de la distribución inicial de los datos. El término *dominante* del algoritmo es el bucle externo que anida a un bucle interno. Por ello, el número de comparaciones que realiza el algoritmo es el número decreciente de iteraciones del bucle interno: `n-1, n-2, ... 2, 1` (`n` es el número de elementos). La suma de los términos de la sucesión se ha obtenido en el apartado anterior, 8.3.2, y se ha comprobado que depende de n^2. Como conclusión, la complejidad del algoritmo de selección es $0(n^2)$.

8.5. ORDENACIÓN POR INSERCIÓN

Este método de ordenación es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, que consiste en insertar un nombre en su posición correcta dentro de una lista que ya está ordenada. Así el proceso en el caso de la lista de enteros `a[] = 50, 20, 40, 80, 30`

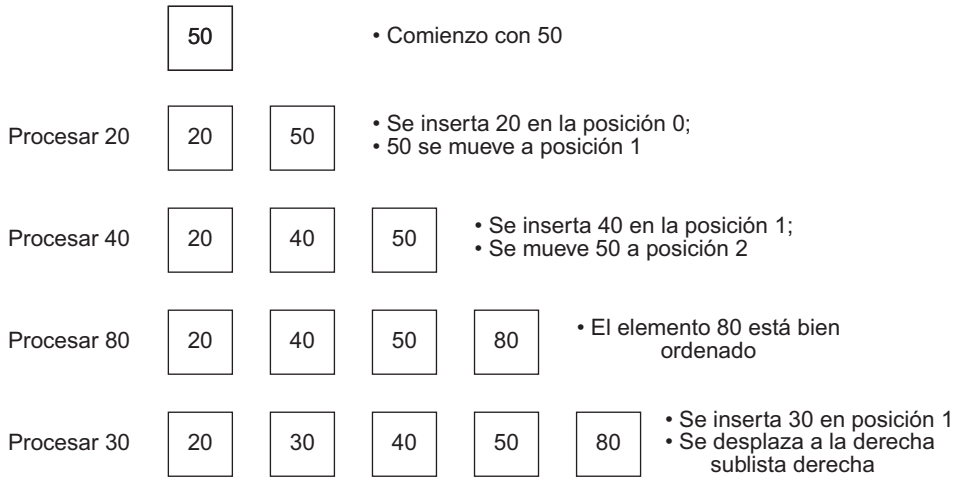


Figura 8.1. Método de ordenación por inserción

8.5.1. Algoritmo de ordenación por inserción

El algoritmo correspondiente a la ordenación por inserción contempla los siguientes pasos:

1. El primer elemento `a[0]` se considera ordenado; es decir, la lista inicial consta de un elemento.
2. Se inserta `a[1]` en la posición correcta; delante o detrás de `a[0]`, dependiendo de que sea menor o mayor.
3. Por cada iteración `i` (`desde i = 1 hasta n - 1`) se explora la sublista `a[i-1] ... a[0]` buscando la posición correcta de inserción de `a[i]`; a la vez se mueve *hacia abajo* (a la derecha en la sublista) una posición todos los elementos mayores que el elemento a insertar `a[i]`, para dejar vacía esa posición.
4. Insertar el elemento `a[i]` en la posición correcta.

8.5.2. Codificación del algoritmo de ordenación por *inserción*

La codificación del algoritmo se realiza en la función `ordInsercion()`. Los elementos del *array* son de tipo entero, en realidad puede ser cualquier tipo básico y ordinal.

```
void ordInsercion (int a[], int n)
{
  int i, j, aux;

  for (i = 1; i < n; i++)
  {
    /* indice j es para explorar la sublista a[i-1]..a[0] buscando la po-
       sicion correcta del elemento destino */
    j = i;
    aux = a[i];
      // se localiza el punto de inserción explorando hacia abajo
    while (j > 0 && aux < a[j-1])
    {
      // desplazar elementos hacia arriba para hacer espacio
      a[j] = a[j-1];
      j--;
    }
    a[j] = aux;
  }
}
```

8.5.3. Complejidad del algoritmo de *inserción*

A la hora de analizar este algoritmo se observa que el número de instrucciones que realiza depende del bucle externo que anida al bucle condicional `while`. Siendo `n` el número de elementos, el bucle externo realiza `n - 1` *pasadas*, por cada una de ellas y *en el peor de los casos* (`aux` siempre menor que `a[j-1]`), el bucle interno `while` itera un número creciente de veces que da lugar a la sucesión: `1, 2, 3, ... n-1` (para `i == n-1`). La suma de los términos de la sucesión se ha obtenido en el Apartado 8.3.2, y se ha comprobado que el término dominante es n^2. Como conclusión, la complejidad del algoritmo de inserción es $0(n^2)$.

8.6. ORDENACIÓN POR BURBUJA

El método de *ordenación por burbuja* es el más conocido y popular entre estudiantes y aprendices de programación, por su facilidad de comprender y programar; por el contrario, es el menos eficiente y por ello, normalmente, se aprende su técnica pero no suele utilizarse.

La técnica utilizada se denomina ***ordenación por burbuja*** u ***ordenación por hundimiento*** debido a que los valores más pequeños "*burbujean*" gradualmente (suben) hacia la cima o parte superior del *array* de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del *array*.

8.6.1. Algoritmo de la burbuja

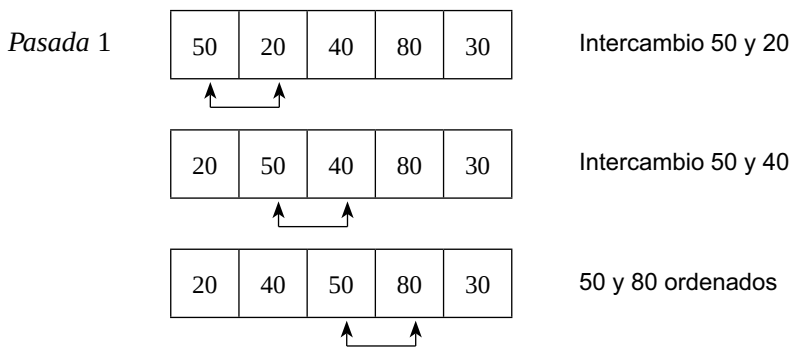
Para un *array* con *n* elementos, la ordenación por burbuja requiere hasta *n – 1* pasadas. Por cada pasada se comparan elementos adyacentes y se intercambian sus valores cuando el primer elemento es mayor que el segundo elemento. Al final de cada pasada, el elemento mayor ha "*burbujeado*" hasta la cima de la sublista actual. Por ejemplo, después que la pasada `1` está completa, la cola de la lista *a[n – 1]* está ordenada y el frente de la lista permanece desordenado. Las etapas del algoritmo son:

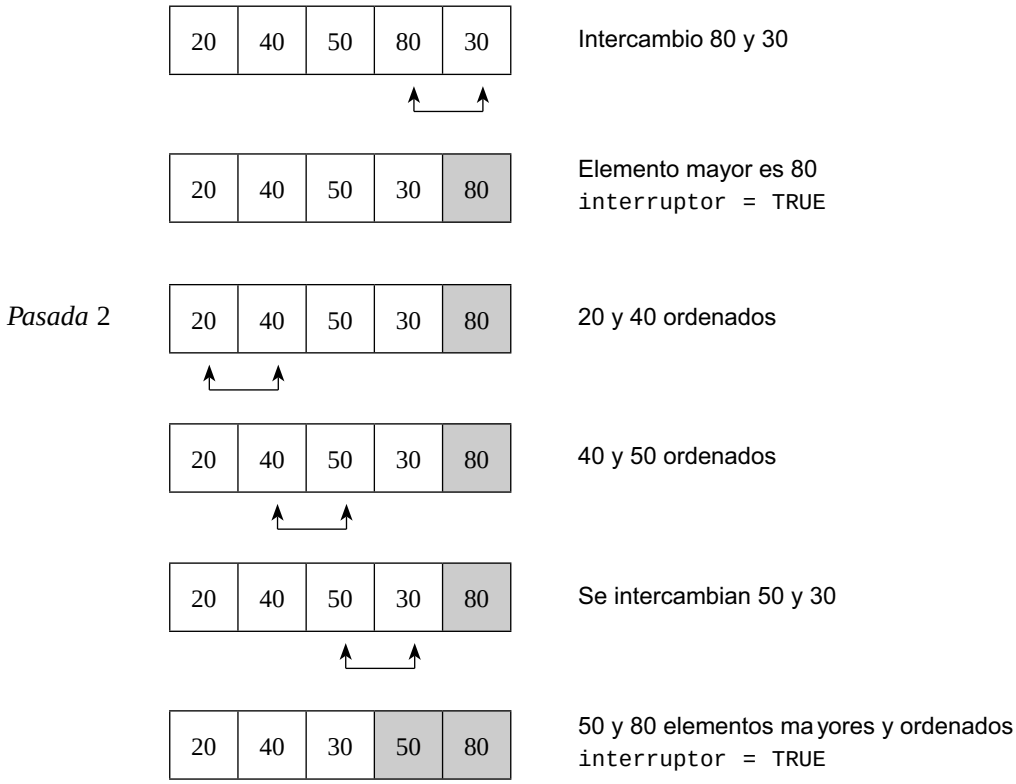
- En la pasada `1` se comparan elementos adyacentes.

  ```
  (a[0],a[1]),(a[1],a[2]),(a[2],a[3]),...(a[n-2],a[n-1])
  ```

  Se realizan *n* – 1 comparaciones, por cada pareja `(a[i],a[i+1])`, se intercambian los valores si `a[i+1] < a[i]`.
  Al final de la pasada, el elemento mayor de la lista está situado en `a[n-1]`.
- En la pasada `2` se realizan las mismas comparaciones e intercambios, terminando con el elemento de segundo mayor valor en `a[n-2]` .
- El proceso termina con la pasada *n – 1*, en la que el elemento más pequeño se almacena en `a[0]`.

El algoritmo tiene una mejora inmediata, el proceso de ordenación puede terminar en la pasada *n – 1*, o bien antes, si en un una pasada no se produce intercambio alguno entre elementos del *array* es porque ya está ordenado, entonces no es necesario más pasadas. A continuación, se ilustra el funcionamiento del algoritmo realizando las dos primeras pasadas en un array de 5 elementos; se introduce la variable `interruptor` para detectar si se ha producido intercambio en la pasada.





El algoritmo terminará cuando se termine la última pasada *(n – 1)*, o bien cuando el valor del `interruptor` sea *falso*, es decir no se haya hecho ningún intercambio.

8.6.2. Codificación del algoritmo de la burbuja

El algoritmo de ordenación de burbuja *mejorado* contempla dos bucles anidados: el *bucle externo* controla la cantidad de pasadas, el *bucle interno* controla cada pasada individualmente y cuando se produce un intercambio, cambia el valor de `interruptor` a *verdadero* `(true)`.

```
void ordBurbuja (long a[], int n)
{
  bool interruptor = true;
  int pasada, j;
    // bucle externo controla la cantidad de pasadas
  for (pasada = 0; pasada < n - 1 && interruptor; pasada++)
  {
    interruptor = false;
    for (j = 0; j < n - pasada - 1; j++)
      if (a[j] > a[j + 1])
      {
          // elementos desordenados, se intercambian
```

```
            interruptor = true;
            intercambiar(a[j], a[j + 1]);
          }
      }
  }
```

8.6.3. Análisis del algoritmo de la burbuja

¿Cuál es la eficiencia del algoritmo de ordenación de la burbuja?

La ordenación de burbuja hace una sola pasada en el caso de una lista que ya está ordenada en orden ascendente y, por tanto, su complejidad es *0(n)*. En el *caso peor* se requieren *(n – i – 1)* comparaciones y *(n – i – 1)* intercambios. La ordenación completa requiere $\frac{n(n-1)}{2}$ comparaciones y un número similar de intercambios. La complejidad para el caso peor es $0(n^2)$ comparaciones y $0(n^2)$ intercambios.

De cualquier forma, el análisis del caso general es complicado dado que alguna de las pasadas pueden no realizarse. Se podría señalar que el número medio de pasadas *k* es *0(n)* y el número total de comparaciones es $0(n^2)$. En el mejor de los casos, la ordenación por burbuja puede terminar en menos de *n – 1* pasadas pero requiere, normalmente, muchos más intercambios que la ordenación por selección y su prestación media es mucho más lenta, sobre todo cuando los arrays a ordenar son grandes.

Consejo de programación

Los algoritmos de ordenación interna: intercambio, selección, inserción y burbuja son fáciles de entender y de codificar, sin embargo poco eficientes y no recomendables para ordenar listas de muchos elementos. La complejidad de todos ellos es cuadrática, $0(n^2)$.

8.7. ORDENACIÓN SHELL

La ordenación Shell debe el nombre a su inventor, *D. L. Shell*. Se suele denominar también *ordenación por inserción con incrementos decrecientes*. Se considera que es una mejora del método de inserción directa.

En el algoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el más pequeño hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo. El algoritmo de Shell modifica los saltos contiguos por saltos de mayor tamaño y con ello consigue que la ordenación sea más rápida. Generalmente, se toma como salto inicial $n/2$ (siendo *n* el número de elementos), luego en cada iteración se reduce el salto a la mitad, hasta que el salto es de tamaño 1. El Ejemplo 8.1 muestra paso a paso el método de Shell.

EJEMPLO 8.1. Aplicar el método Shell para ordenar en orden creciente la lista: 6 1 5 2 3 4 0

El número de elementos que tiene la lista es 6, por lo que el salto inicial es 6/2 = 3. La siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondiente.

Recorrido	*Salto*	*Intercambios*	*Lista*
1	3	(6,2),(5,4),(6,0)	2 1 4 0 3 5 6
2	3	(2, 0)	0 1 4 2 3 5 6
3	3	ninguno	0 1 4 2 3 5 6
salto 3/2 = 1			
4	1	(4,2),(4,3)	0 1 2 3 4 5 6
5	1	ninguno	0 1 2 3 4 5 6

8.7.1. Algoritmo de ordenación Shell

Los pasos a seguir por el algoritmo para una lista de `n` elementos:

1. Se divide la lista original en `n/2` grupos de dos, considerando un incremento o salto entre los elementos de `n/2`.
2. Se clasifica cada grupo por separado, comparando las parejas de elementos y si no están ordenados se intercambian.
3. Se divide ahora la lista en la mitad de grupos (`n/4`), con un salto entre los elementos también mitad (`n/4`), y nuevamente se clasifica cada grupo por separado.
4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.
5. El algoritmo termina cuando el tamaño del salto es 1.

Por consiguiente, los recorridos por la lista están condicionados por el bucle,

```
salto ← n / 2
mientras (salto > 0) hacer
```

Para dividir la lista en grupos y clasificar cada grupo se anida este código:

```
desde i ← (salto + 1) hasta n hacer
  j ← i – salto
  mientras (j > 0) hacer
    k ← j + salto
    si (a[j] <= a[k]) entonces
      j ← 0
    sino
      Intercambio (a[j], a[k])
      j ← j - salto
    fin_si
  fin_mientras
fin_desde
```

Donde se observa que se comparan pares de elementos de índice `j` y `k`, separados por un *salto* de `salto` . Así, si `n = 8` el primer valor de `salto = 4`, y los índices `i = 5`, `j = 1`, `k = 6` . Los siguiente valores que toman son `i = 6`, `j = 2`, `k = 7`, y así hasta recorrer la lista.

8.7.2. Codificación del algoritmo de ordenación Shell

Al codificar el algoritmo se considera que el rango de elementos es `0 ... n-1` y, por consiguiente, se ha de desplazar una posición a la *izquierda* las variables índice respecto a lo expuesto en el algoritmo.

```
void ordenacionShell(double a[], int n)
{
  int salto, i, j, k;
  salto = n / 2;
  while (salto > 0)
  {
    for (i = salto; i < n; i++)
    {
      j = i - salto;
      while (j >= 0)
      {
        k = j + salto;
        if (a[j] <= a[k])
          j = -1;          // par de elementos ordenado
        else
        {
          intercambiar(a[j], a[j+1]);
          j -= salto;
        }
      }
    }
    salto = salto / 2;
  }
}
```

8.7.3. Análisis del algoritmo de ordenación Shell

A pesar de que el algoritmo tiene tres bucles anidados (`while-for-while`), es más eficiente que el algoritmo de inserción y que cualquiera de los algoritmos simples analizados en los apartados anteriores. El análisis del tiempo de ejecución del algoritmo *Shell* no es sencillo. Su inventor, *Shell*, recomienda que el salto inicial sea `n/2`, y continuar dividiendo el salto por la mitad hasta conseguir un salto `1`. Con esta elección se puede probar que el tiempo de ejecución es $0(n^2)$ *en el peor de los casos*, y el tiempo medio de ejecución es $0(n^{3/2})$.

Posteriormente, se han encontrado secuencias de saltos que mejoran el rendimiento del algoritmo. Así, dividiendo el salto por 2.2 en lugar de la mitad se consigue un tiempo medio de ejecución de complejidad menor de $0(n^{5/4})$.

Nota de programación

La codificación del algoritmo Shell con el salto igual al salto anterior dividido por `2.2`, puede hacer el salto igual a `0`. Si esto ocurre, se ha de codificar que el salto sea igual a `1`, en caso contrario no funcionaría el algoritmo.

```
salto = (int) salto / 2.2;
salto = (salto == 0) ? 1 : salto;
```

8.8. ORDENACIÓN RÁPIDA (*QUICKSORT*)

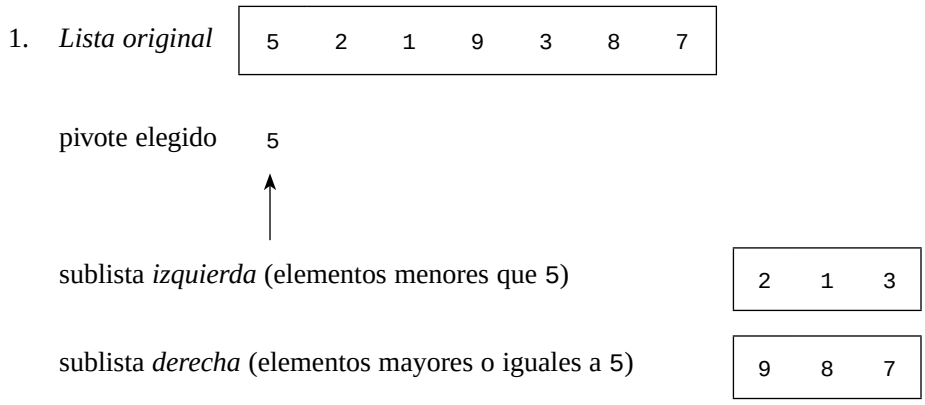
El algoritmo conocido como *quicksort* (ordenación rápida) recibe el nombre de su autor, *Tony Hoare*. El fundamento del algoritmo es simple, se basa en la división de la lista en particiones a ordenar, en definitiva aplica la técnica "*divide y vencerás*". El método es, posiblemente, el más pequeño de código, más rápido de media, más elegante y más interesante y eficiente de los algoritmos conocidos de ordenación.

El algoritmo divide los *n* elementos de la lista a ordenar en dos partes o particiones separadas por un elemento: una partición *izquierda*, un elemento *central* denominado *pivote*, y una partición *derecha*. La partición se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) son menores que todos los elementos de la segunda sublista (partición derecha). Las dos sublistas se ordenan entonces independientemente.

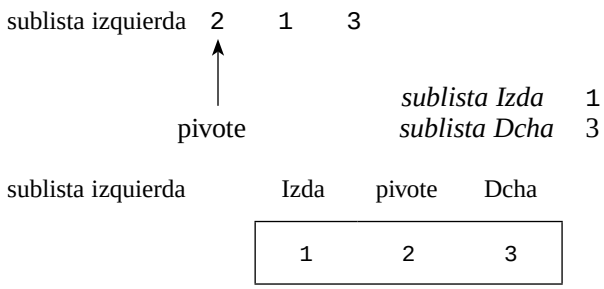
Para dividir la lista en particiones (sublistas) se elige uno de los elementos de la lista como *pivote* o *elemento de partición*. Si los elementos de la lista están en orden aleatorio, se puede elegir cualquier elemento como *pivote*, por ejemplo, el primer elemento de la lista. Si la lista tiene algún orden parcial, que se conoce, se puede tomar otra decisión para el *pivote*. Idealmente, el *pivote* se debe elegir de modo que se divida la lista por la mitad, de acuerdo al tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían *pivotes* ideales, mientras que 1 o 10 serían elecciones "pobres" de *pivotes*.

Una vez que el *pivote* ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el *pivote* y la otra todas las claves mayores o iguales que el *pivote*. Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *quicksort*. La lista final ordenada se consigue concatenando la primera sublista, el *pivote* y la segunda sublista, en ese orden, en una única lista. La primera etapa de *quicksort* es la división o *"particionado"* recursivo de la lista hasta que todas las sublistas consten de sólo un elemento.

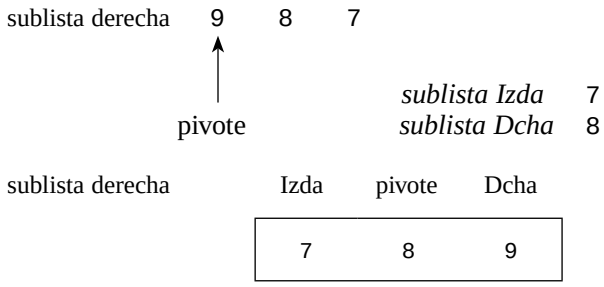
EJEMPLO 8.2. Se ordena una lista de números enteros aplicando el algor itmo *quicksort*, se elige como pivote el primer elemento de la lista.



2. El algoritmo se aplica a la sublista izquierda



3. El algoritmo se aplica a la sublista derecha



4. Lista ordenada final

Sublista izquierda *pivote* *Sublista derecha*

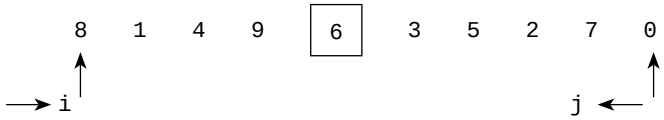
1 2 3 5 7 8 9

EJEMPLO 8.3. Se aplica el algoritmo *quicksort* para dividir una lista de números enteros en dos sublistas. El pivote es el elemento central de la lista.

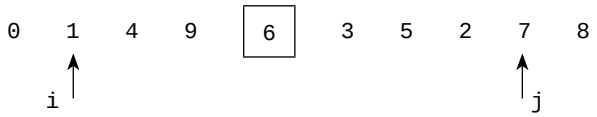
Lista original: 8 1 4 9 6 3 5 2 7 0
pivote (elemento central) 6

Una vez elegido el *pivote*, la segunda etapa requiere mover todos los elementos menores al pivote a la parte izquierda del *array* y los elementos mayores a la parte derecha. Para ello se recorre la lista de izquierda a derecha utilizando un índice `i`, que se inicializa a la posición más baja (`inferior`), buscando un elemento mayor al pivote. También se recorre el array de derecha a izquierda buscando un elemento menor. Para esto se utilizará el índice `j`, inicializado a la posición más alta (`superior`).

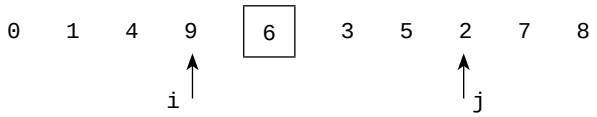
El índice `i` se detiene en el elemento `8` (mayor que el *pivote*) y el índice `j` se detiene en el elemento `0` (menor que el *pivote*).



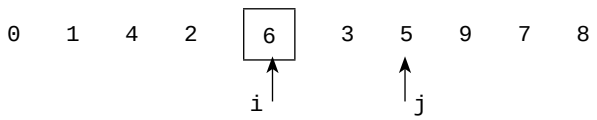
Ahora, se intercambian `a[i]` y `a[j]` para que estos dos elementos se sitúen correctamente en cada sublista; y se incrementa el índice `i`, y se decrementa `j` para seguir los intercambios.



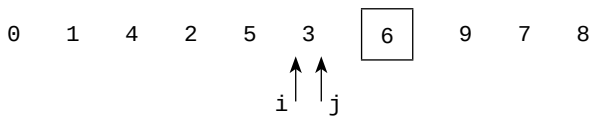
A medida que el algoritmo continúa, `i` se detiene en el elemento mayor, `9`, y `j` se detiene en el elemento menor, `2`,



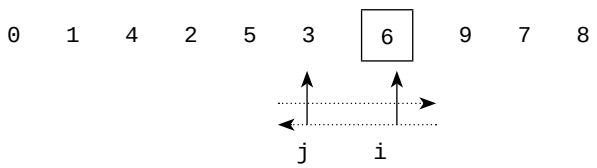
Se intercambian los elementos mientras que `i` y `j` no se crucen. En caso contrario se detiene este bucle. En el caso anterior se intercambian `9` y `2`.



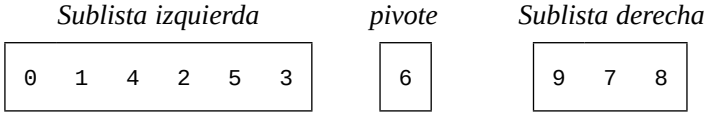
Continúa la exploración y ahora el contador `i` se detiene en el elemento `6` (que es el *pivote*) y el índice `j` se detiene en el elemento menor `5`



Los índices tienen actualmente los valores `i = 5`, `j = 5`. Continúa la exploración hasta que `i > j`, acaba con `i = 6`, `j = 5`



En esta posición los índices `i` y `j` han cruzado posiciones en el *array*, se detiene la búsqueda y no se realiza ningún intercambio ya que el elemento al que accede `j` está ya correctamente situado. Las dos sublistas ya han sido creadas, la lista original se ha dividido en dos particiones:



8.8.1. Algoritmo *Quicksort*

El primer problema a resolver en el diseño del algoritmo de *quicksort* es seleccionar el *pivote*. Aunque la posición del *pivote*, en principio puede ser cualquiera, una de las decisiones más ponderadas es aquella que considera el *pivote* como el elemento central o próximo al central de la lista. Una vez que se ha seleccionado el *pivote*, se ha de buscar el sistema para situar en la sublista izquierda todos los elementos menores que el *pivote* y en la sublista derecha todos los elementos mayores. La Figura 8.2 muestra las operaciones del algoritmo para ordenar la lista `a[]` de `n` elementos enteros.

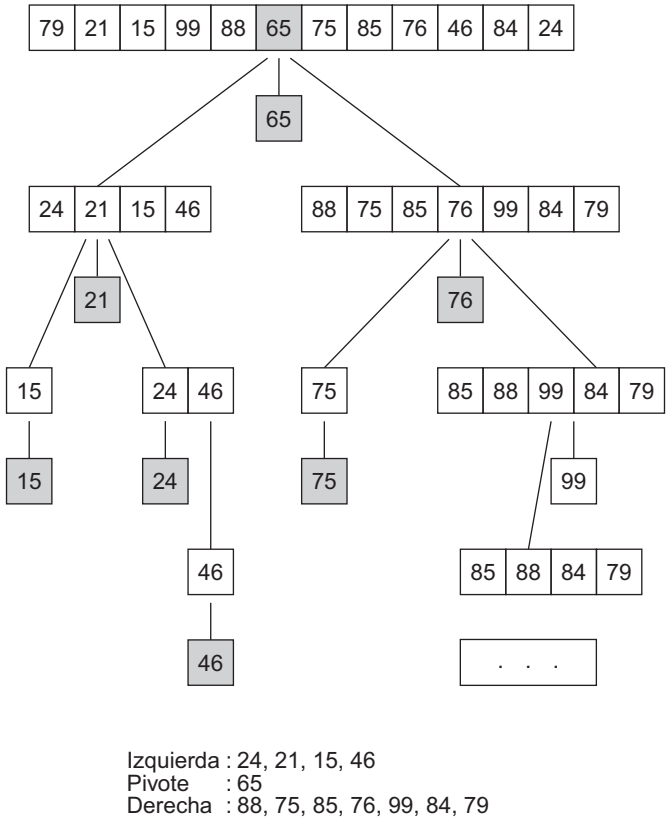


Figura 8.2. Ordenación rápida eligiendo como pivote el elemento central.

Los pasos que sigue el algoritmo *quicksort* son:

Seleccionar el elemento central de `a[]` como *pivote*.
Dividir los elementos restantes en particiones *izquierda* y *derecha*, de modo que ningún elemento de la izquierda tenga una clave mayor que el *pivote* y que ningún elemento a la derecha tenga una clave más pequeña que la del *pivote*.
Ordenar la partición izquierda utilizando *quicksort* recursivamente.
Ordenar la partición derecha utilizando *quicksort* recursivamente.
La solución es partición *izquierda* seguida por el *pivote* y la partición *derecha*.

8.8.2. Codificación del algoritmo *QuickSort*

La implementación, al igual que el algoritmo, es recursiva; la función `quicksort()` tiene como argumentos el array `a[]` y los índices que le delimitan: `primero` y `ultimo`.

```
void quicksort(double a[], int primero, int ultimo)
{

  int i, j, central;
  double pivote;

  central = (primero + ultimo) / 2;
  pivote = a[central];
  i = primero;
  j = ultimo;

  do {
     while (a[i] < pivote) i++;
     while (a[j] > pivote) j--;

     if (i <= j)
     {
        intercambiar(a[i], a[j]);
        i++;
        j--;
     }
  }while (i <= j);

   if (primero < j)
     quicksort(a, primero, j); // mismo proceso con sublista izqda
   if (i < ultimo)
     quicksort(a, i, ultimo); // mismo proceso con sublista drcha
}
```

8.8.3. Análisis del algoritmo *Quicksort*

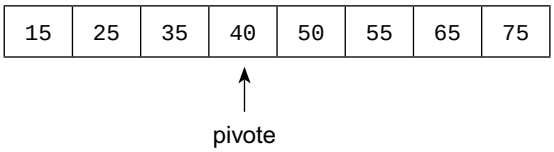
El análisis general de la eficiencia del *quicksort* es difícil. La mejor forma de determinar la complejidad del algoritmo es considerar el número de comparaciones realizadas teniendo en cuenta circunstancias ideales. Supongamos que *n* (número de elementos) es una potencia de *2*,

$n = 2^k$ ($k = log_2 n$). Además, supongamos que el *pivote* es el elemento central de cada lista, de modo que *quicksort* divide la sublista en dos sublistas aproximadamente iguales.

En el primer recorrido se realizan $n - 1$ comparaciones. Este recorrido crea dos sublistas, aproximadamente de tamaño $n/2$. En la siguiente etapa, el proceso de cada sublista requiere aproximadamente $n/2$ comparaciones. Las comparaciones totales de esta fase son $2*(n/2) = n$. A continuación, se procesan cuatro sublistas que requieren un total de $4*(n/4)$ comparaciones, etcétera. Eventualmente, el proceso de división termina después de k pasadas cuando la sublista resultante tenga tamaño 1. El número total de comparaciones es aproximadamente:

$$n + 2*(n/2) + 4*(n/4) + ... + n*(n/n) = n + n + ... + n = n * k = n * log_2 n$$

El caso ideal que se ha examinado se realiza realmente cuando la lista está ordenada en orden ascendente. En este caso el *pivote* es siempre el centro de cada sublista y el algoritmo tiene la complejidad $0(n \log n)$.



El escenario del caso peor de *quicksort* ocurre cuando el *pivote* cae consistentemente en una sublista de un elemento y deja el resto de los elementos en la segunda sublista. Esto sucede cuando el pivote es siempre el elemento más pequeño de su sublista. En el recorrido inicial, hay n comparaciones y la sublista grande contiene $n - 1$ elementos. En el siguiente recorrido, la sublista mayor requiere $n - 1$ comparaciones y produce una sublista de $n - 2$ elementos, etc. El número total de comparaciones es:

$$n + n - 1 + n - 2 + ... + 2 = (n - 1) * (n + 2)/2$$

Entonces, la complejidad en el caso peor es $0(n^2)$. En general, el algoritmo de ordenación *quicksort* tiene como complejidad media $0(n \log n)$ siendo posiblemente el algoritmo más rápido. La Tabla 8.1 muestra las complejidades de los algoritmos empleados en los métodos explicados en el libro.

Tabla 8.1. Comparación complejidad métodos de ordenación.

Método	Complejidad
Burbuja	n^2
Inserción	n^2
Selección	n^2
Montículo	$n \log n$
Mergersort	$n \log n$
Shell	$n^{3/2}$
Quicksort	$n \log n$

En conclusión, se suele recomendar que para listas pequeñas, los métodos más eficientes son: inserción y selección, y para listas grandes: *quicksort, Shell, mergesort* (Apartado 7.5), y *montículo* (se desarrolla en Capítulo 13; *Colas de Prioridades y Montículos*).

8.9. ORDENACIÓN CON *URNAS*: *BINSORT* Y *RADIXSORT*

Estos métodos de ordenación utilizan *urnas* en el proceso de ordenación. En cada recorrido se deposita en una $urna_i$ aquellos elementos del array cuya clave tienen cierta correspondencia con el índice i.

8.9.1. *Binsort* (ordenación por urnas)

Este método, también denominado *clasificación por urnas*, se propone conseguir funciones tiempo de ejecución de complejidad menor que *O(n log n)* para ordenar una lista de `n` elementos, siempre que se conozca alguna relación del campo *clave* de los elementos respecto de las urnas.

Supóngase este caso ideal: se desea ordenar un array `v[]` respecto un campo clave de tipo entero, además los valores de las claves se encuentran en el rango de `1` a `n`, sin claves duplicadas y siendo `n` el número de elementos. En estas circunstancias ideales es posible ubicar los registros ordenados en un array auxiliar `t[]` mediante este único bucle:

```
desde i ← 1 hasta n hacer
  t[v[i].clave] ← v[i];
fin_desde
```

Sencillamente, determina la posición que le corresponde al registro según el valor del campo clave. El bucle lleva un tiempo de ejecución de complejidad lineal `O(n)`.

Esta ordenación tan sencilla e intuitiva es un caso particular del método binsort. El método utiliza urnas, de tal forma que una urna contiene todos los registros con una misma clave.

El proceso consiste en examinar cada elemento, `r`, del array y situarle en la urna `i`, siendo `i` el valor del campo clave de `r`. Es previsible que sea necesario guardar más de un elemento en una misma urna por tener claves repetidas. Entonces, las urnas hay que concatenarlas, en el orden de menor índice de urna a mayor, para que el array quede ordenado (ascendentemente) respecto al campo clave.

La Figura 8.3 muestra un vector de `m` urnas. Cada urna están representada mediante una lista enlazada.

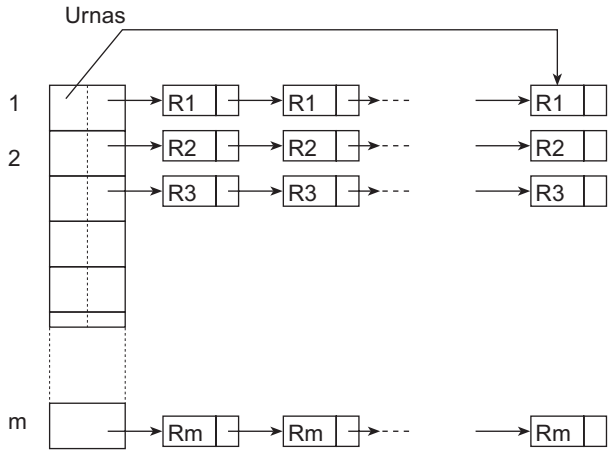


Figura 8.3. Estructura formada por m urnas.

Algoritmo *Binsort*

El algoritmo se aplica sobre elementos que se ordenan respecto una clave cuyo rango de valores es relativamente pequeño respecto al número de elementos. Si la clave es de tipo entero, en el rango `1 .. m`, son necesarias `m` urnas agrupadas en un vector. Las urnas son listas enlazadas, cada nodo de la lista contiene un elemento cuya clave se corresponde con el índice de la urna en la que se encuentra. Para una mejor comprensión del algoritmo, la clave será igual al índice de la urna. Así, en la urna `1` se sitúan los elementos cuya clave es `1`, en la urna `2` los elementos de clave `2`, y así sucesivamente en la urna *i* se sitúan los registros cuya clave sea *i*.

La primera acción del algoritmo consiste en distribuir los elementos en las diversas urnas. A continuación, se concatena las listas enlazadas para formar un única lista, que contendrá los elementos en orden creciente; por último, se recorre la lista asignando cada nodo al array. La Figura 8.4 se muestra cómo realizar la concatenación.

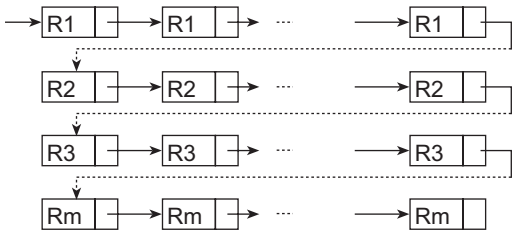


Figura 8.4. Concatenación de urnas representadas por listas enlazadas.

El algoritmo expresado en seudocódigo para un vector de *n* elementos:

```
OrdenacionBinsort(vector, n)

inicio
  CrearUrnas(Urnas);
    {Distribución de elementos en sus correspondientes urnas}
  desde j ← 1 hasta n hacer
    AñadirEnUrna(Urnas[vector[j].clave], vector[j]);
  fin_desde

    {Concatena las listas que representan a las urnas
      desde Urna_i hasta Urna_m}

  i ← 1;                          {búsqueda de primera urna no vacía}
  mientras EsVacia(Urnas[i]) hacer
    i ← i+1
  fin_mientras

  desde j ← i+l a m hacer
    EnlazarUrna(Urnas[i], Urnas[j]);
  fin_desde

  {recorre las lista(urnas) resultado de la concatenación}
  j ← 1;
  dir = <frente Urnas[i]>;
```

```
  mientras dir <> nulo hacer
    vector[j] = < elemento apuntado por dir>;
    j ← j+i;
    dir ← Sgte(dir)
  fin_mientras

fin
```

8.9.2. RadixSort *(ordenación por residuos)*

Este método de ordenación es un caso particular del algoritmo de clasificación por urnas. La manera de ordenar, manualmente, un conjunto de fichas nos da una idea intuitiva de este método de ordenación: se forman *montones* de fichas, cada uno caracterizado por tener sus componentes un mismo dígito (letra, si es ordenación alfabética) en la misma posición. Inicialmente se forman los *montones* por las unidades (dígito de menor peso); estos montones se recogen y agrupan en orden ascendente, desde el montón del dígito 0 al montón del dígito 9. Entonces, las fichas están ordenadas respecto a las unidades, a continuación, se vuelve a distribuir las fichas en montones, según el dígito de las decenas. El proceso de distribuir las fichas por montones y posterior acumulación en orden se repite tantas veces como número de dígitos tiene la ficha de mayor valor.

Suponer que las fichas están identificadas por un campo entero de tres dígitos, los pasos del algoritmo *RadixSort* para los siguientes valores:

```
345, 721, 425, 572, 836, 467, 672,194, 365, 236, 891, 746, 431, 834, 247,
529, 216, 389
```

Atendiendo al dígito de menor peso (unidades) los *montones*:

				216		
431			*365*	*746*		
891	*672*	*834*	*425*	*236*	*247*	*389*
721	*572*	*194*	*345*	*836*	*467*	*529*
1	*2*	*4*	*5*	*6*	*7*	*9*

Una vez agrupados los montones en orden ascendente la lista es la siguiente:

```
721, 891, 431, 572, 672, 194, 834, 345, 425, 365, 836, 236, 746, 216, 467,
247, 529, 389
```

Esta lista ya está ordenada respecto al dígito de menor peso, respecto a las unidades. Pues bien, ahora se vuelven a distribuir en *montones* respecto al segundo dígito (decenas):

		236					
	529	*836*	*247*				
	425	*834*	*746*	*467*	*672*		*194*
216	*721*	*431*	*345*	*365*	*572*	*389*	*891*
1	*2*	*3*	*4*	*6*	*7*	*8*	*9*

Una vez agrupados los *montones* en orden ascendente la lista es la siguiente:

```
216, 721, 425, 529, 431, 834, 836, 236, 345, 746, 247, 365, 467, 572, 672,
389, 891, 194
```

La lista fichas ya está ordenada respecto a los dos últimos dígitos, es decir, respecto a las decenas. Por último, se vuelven a distribuir en *montones* respecto al tercer dígito:

	247	389	467				891
	236	365	431	572		746	836
194	216	345	425	529	672	721	834
1	2	3	4	5	6	7	8

Se agrupan los *montones* en orden ascendente y la lista ya está ordenada:

```
194, 216, 236, 247, 345, 365, 389, 425, 431, 467, 529, 572, 672, 721, 746,
834, 836, 891
```

Algoritmo de ordenación RadixSort

La idea clave de la ordenación RadixSort es clasificar por urnas tantas veces como máximo número de dígitos (o de letras) tengan los elementos de la lista. En cada paso se realiza una distribución en las urnas y una unión de éstas en orden ascendente, desde la urna *0* a la urna 9.

Al igual que en el método de *BinSort*, las urnas se representan mediante un array de listas enlazadas. Se ha de disponer de tantas urnas como dígitos, `10`, numeradas de `0` a 9. Si la clave respecto a la que se ordena es alfabética, habrá tantas urnas como letras distintas, desde la urna que representa a la letra *a* hasta la *z*.

El algoritmo que se escribe, en primer lugar determina el número máximo de dígitos que puede tener una clave. Un bucle externo, de tantas iteraciones como el máximo de dígitos, realiza las acciones de distribuir por urnas los elementos y concatenar.

```
OrdenacionRadixsort(vector, n)

inicio
  { cálculo el número máximo de dígitos: ndig }
  ndig ← 0;
  temp ← maximaClave;
  mientras (temp > 0) hacer
    ndig ← ndig+1
    tem ← temp / 10;
  fin_mientras

  peso ← 1 { permite obtener los dígitos de menor a mayor peso}
  desde i ← 1 hasta ndig hacer
    CrearUrnas(Urnas);
    desde j ← 1 hasta n hacer
      d ← (vector[j] / peso) modulo 10;
      AñadirEnUma(Urnas[d], vector[j]);
    fin_desde

    { búsqueda de primera urna no vacía: j }
        j ← 0;
        mientras frente (Urnas i, j <> nulo) hacer
        j ← j + 1;
    desde r ← j+1 hasta M hace { M: número de urnas }
      EnlazarUma(Urnas[r], Urnas[j]);
    fin_desde
```

```
      {Se recorre la lista resultado de la concatenación}
    r ← 1;
    dir ← frente(Urnas[j]);
    mientras dir <> nulo hacer
      vecto[r] ← dir.elemento;
      r ← r+1;
      dir ← siguiente(dir)
    fin_mientras
    peso ← peso * 10;
  fin_desde
fin_ordenacion
```

8.10. BÚSQUEDA EN LISTAS: BÚSQUEDA SECUENCIAL Y BINARIA

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arrays y registros y, por ello, será necesario determinar si un array contiene un valor que coincida con un *valor clave*. El proceso de encontrar un elemento específico de un array se denomina *búsqueda*. En esta sección se examinarán dos técnicas de búsqueda: *búsqueda lineal* o *secuencial*, la más sencilla, y *búsqueda binaria* o *dicotómica*, la más eficiente.

8.10.1. Búsqueda secuencial

La búsqueda secuencial busca un elemento de una lista utilizando un valor destino llamado ***clave***. En una búsqueda secuencial (a veces llamada ***búsqueda lineal***), los elementos de una lista se exploran (se examinan) en secuencia, uno después de otro.

El algoritmo de búsqueda secuencial compara cada elemento del array con la *clave* de búsqueda. Dado que el array no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primer elemento, el último elemento o cualquier otro. De promedio, al menos el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del array. El método de búsqueda lineal funcionará bien con arrays pequeños o no ordenados.

8.10.2. Búsqueda binaria

La búsqueda secuencial se aplica a cualquier lista. Si la lista está ordenada, la *búsqueda binaria* proporciona una técnica de búsqueda mejorada. Localizar una palabra en un diccionario es un ejemplo típico de búsqueda binaria. Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra de la palabra que busca. Se puede tener suerte y acertar con la página correcta; pero, normalmente, no será así y se mueve el lector a la página anterior o posterior del libro. Por ejemplo, si la palabra comienza con "`J`" y se está en la "`L`" se mueve uno hacia atrás. El proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que la palabra no está en la lista.

Una idea similar se aplica en la búsqueda en una lista ordenada. Se sitúa el índice de búsqueda en el centro de la lista y se comprueba si nuestra clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sitúa en la mitad inferior o superior del elemento central de la lista. En general, si los datos de la lista están ordenados se puede utilizar esa información para acortar el tiempo de búsqueda.

EJEMPLO 8.4. Se desea buscar el elemento 225 en el conjunto de datos siguiente:

```
a[0]  a[1]  a[2]  a[3]  a[4]  a[5]  a[6]  a[7]
 13    44    75   100   120   275   325   510
```

El punto central de la lista es el elemento `a[3] (100)`. El valor que se busca es `225` que es mayor que `100`; por consiguiente, la búsqueda continúa en la mitad superior del conjunto de datos de la lista, es decir, en la sublista:

```
a[4]  a[5]  a[6]  a[7]
120   275   325   510
```

Ahora el elemento mitad de esta sublista es `a[5] (275)`. El valor buscado, `225`, es menor que `275` y, por consiguiente, la búsqueda continúa en la mitad inferior del conjunto de datos de la lista actual; es decir en la sublista unitaria: `a[4](120)`.

El elemento mitad de esta sublista es el propio elemento `a[4](120)` y al ser `225` mayor que `120` la búsqueda continuar en una sublista vacía. Se concluye indicando que no se ha encontrado la clave en la lista.

8.10.3. Algoritmo y codificación de la búsqueda binaria

Suponiendo que la lista está en un array delimitado por índices `bajo` y `alto`, los pasos a seguir:

1. Calcular el índice del punto central del array:

   ```
   central = (bajo + alto)/2          (división entera)
   ```

2. Comparar el valor de este elemento central con la clave:

   ```
   clave = a[central]    a[central] > clave    a[central] < clave
   ├─────┼─────┤         ├─────┼─────┤         ├─────┼─────┤
   bajo central alto     bajo central alto     bajo central alto
   ```

   Clave encontrada *Búsqueda lista inferior* *Búsqueda lista superior*

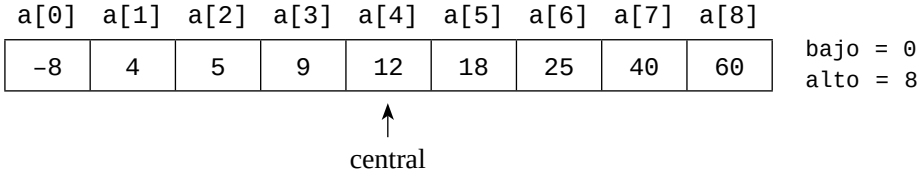
   Si `a[central] < clave`, la nueva sublista de búsqueda queda delimitada por:
   `bajo = central+1 .. alto`
   Si `a[central] > clave`, la nueva sublista de búsqueda queda delimitada por:
   `bajo .. alto central-1`

   ```
          clave                          clave
   [──────────────────]           [──────────────────]
   bajo        central-1=alto     bajo=central+1     alto
   ```

El algoritmo termina bien porque se ha encontrado la clave o porque el valor de `bajo` excede a `alto` y el algoritmo devuelve el indicador de fallo de `-1`.

EJEMPLO 8.5. Sea el array de enteros a `(-8, 4, 5, 9, 12, 18, 25, 40, 60)`, buscar la clave 40.

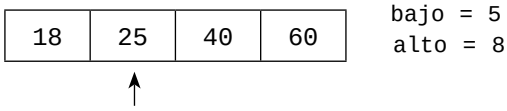
1.



$$central = \frac{bajo + alto}{2} = \frac{0 + 8}{2} = 4$$

`clave (40) > a[4] (12)`

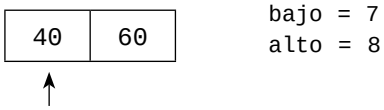
2. Buscar en sublista derecha



$$central = \frac{bajo + alto}{2} = \frac{5 + 8}{2} = 6$$ (división entera)

`clave (40) > a[6] (25)`

3. Buscar en sublista derecha



$$central = \frac{bajo + alto}{2} = \frac{7 + 8}{2} = 7$$

`clave (40) = a[7](40)` *búsqueda con éxito*

El algoritmo ha requerido 3 comparaciones frente a 8 comparaciones `(n-1)` que se hubieran realizado con la búsqueda secuencial.

Codificación

```
int busquedaBin(int a[], int n, int clave)
{
  int central, bajo, alto;
  int valorCentral;
  bajo = 0;
  alto = n - 1;
  while (bajo <= alto)
```

```
  {
    central = (bajo + alto)/2;       // índice de elemento central
    valorCentral = a[central];       // valor del índice central
    if (clave == valorCentral)
      return central;                // encontrado, devuelve posición
    else if (clave < valorCentral)
      alto = central - 1;            // ir a sublista inferior
    else
      bajo = central + 1;            // ir a sublista superior
  }
  return -1;                         //elemento no encontrado
}
```

8.10.4. Análisis de los algoritmos de búsqueda

Al igual que sucede con las operaciones de ordenación, cuando se realizan operaciones de búsqueda es preciso considerar la eficiencia (complejidad) de los algoritmos empleados en la búsqueda. El grado de eficiencia en una búsqueda será vital cuando se trata de localizar un dato en una lista o tabla en memoria de muchos elementos.

Complejidad de la búsqueda secuencial

La complejidad diferencia entre el comportamiento en el *caso peor y mejor*. El *mejor caso* se encuentra cuando aparece una coincidencia en el primer elemento de la lista y en ese caso el tiempo de ejecución es *O(1)*. El caso peor se produce cuando el elemento no está en la lista o se encuentra al final de la lista. Esto requiere buscar en todos los *n* términos, lo que implica una complejidad de *O(n)*.

Análisis de la búsqueda binaria

El *caso mejor* se presenta cuando una coincidencia se encuentra en el punto central de la lista. En este caso la complejidad es *O(1)* dado que sólo se realiza una prueba de comparación de igualdad. La complejidad del *caso peor* es *O(log2n)* que se produce cuando el elemento no está en la lista o el elemento se encuentra en la última comparación. Se puede deducir intuitivamente esta complejidad. El caso peor se produce cuando se debe continuar la búsqueda y llegar a una sublista de longitud de 1. Cada iteración que falla debe continuar disminuyendo la longitud de la sublista por un factor de 2. Si la división de sublistas requiere *m* iteraciones, la sucesión de tamaños de las sublistas hasta una sublista de longitud 1:

$$n \quad n/2 \quad n/2^2 \quad n/2^3 \quad n/2^4 \ldots n/2^m$$

siendo $n/2^m = 1$. Tomando logaritmos en base 2 en la expresión anterior:

$$n = 2^m$$
$$m = \log_2 n$$

Por esa razón, la complejidad del caso peor es $O(\log_2 n)$. Cada iteración requiere una operación de comparación:

$$\text{Total comparaciones} \approx 1 + \log_2 n$$

Comparación de la búsqueda binaria y secuencial

La comparación en tiempo entre los algoritmos de búsqueda secuencial y binaria se va haciendo espectacular a medida que crece el tamaño de la lista de elementos. Tengamos presente que en el caso de la búsqueda secuencial en el peor de los casos coincidirá el número de elementos examinados con el número de elementos de la lista tal como representa su complejidad *O(n)*.

Sin embargo, en el caso de la búsqueda binaria, tengamos presente, por ejemplo, que $2^{10} = 1024$, lo cual implica el examen de 11 posibles elementos; si se aumenta el número de elementos de una lista a 2048 y teniendo presente que $2^{11} = 2048$ implicará que el número máximo de elementos examinados en la búsqueda binaria es 12. Si se sigue este planteamiento, se puede encontrar el número *m* más pequeño para una lista de 1000000, tal que:

$2^n \geq 1.000.000$

Es decir, $2^{19} = 524.288$, $220 = 1.048.576$ y, por tanto, el número de elementos examinados (en el peor de los casos) es 21.

Tabla 8.2. Comparación de las búsquedas binaria y secuencial.

Números de elementos examinados		
Tamaño de la lista	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	1.000.000

Consejo de programación

La búsqueda secuencial se aplica para localizar una clave en un array no ordenado. Para aplicar el algoritmo de búsqueda binaria la lista, o array, debe de estar ordenado.

RESUMEN

Una de las aplicaciones más frecuentes en programación es la ordenación. Los datos se pueden ordenar en orden ascendente o en orden descendente. Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*. También se denominan *ordenación interna* y *ordenación externa* respectivamente.

Los algoritmos de ordenación exploran los datos de las listas o arrays para realizar comparaciones y, si es necesario, cambios de posición. Cada recorrido de los datos durante el proceso de ordenación se conoce como *pasada* o *iteración*.

Los algoritmos de ordenación básicos son:

- Selección.
- Inserción.
- Burbuja.

Los algoritmos de ordenación más avanzados son:

- *Shell.*
- *Heapsort (por montículos).*
- *Mergesort.*
- *Radixsort.*
- *Binsort*
- *Quicksort.*

La eficiencia de los algoritmos de burbuja, inserción y selección es $0(n^2)$. La eficiencia de los algoritmos heapsort, radixsort, mergesort y quicksort es *0(n Log n)*.

La búsqueda es el proceso de encontrar la posición de un elemento destino dentro de una lista. Existen dos métodos básicos de búsqueda en arrays: **búsqueda secuencial** y **binaria**.

La **búsqueda secuencial** se utiliza normalmente cuando el array no está ordenado. Comienza en el principio del array y busca hasta que se encuentra el dato buscado y se llega al final de la lista.

Si un array está ordenado, se puede utilizar un algoritmo más eficiente denominado **búsqueda binaria**.

La eficiencia de una búsqueda secuencial es *0(n)*. La eficiencia de una búsqueda binaria es *0(log n)*.

EJERCICIOS

8.1. ¿Cuál es la diferencia entre ordenación por intercambio y ordenación por el método de la burbuja?

8.2. Se desea eliminar todos los números duplicados de una lista o vector (array). Por ejemplo, si el array toma los valores:

```
4    7    11    4    9    5    11    7    3    5
```

ha de cambiarse a

```
4    7    11    9    5    3
```

Escribir una función que elimine los elementos duplicados de un array.

8.3. Escribir una función que elimine los elementos duplicados de un vector ordenado. ¿Cuál es la eficiencia de esta función? Compare la eficiencia con la que tiene la función del Ejercicio 8.2.

8.4. Un vector contiene los elementos mostrados a continuación. Los primeros dos elementos se han ordenado utilizando un algoritmo de inserción. ¿Cuál será el valor de los elementos del vector después de tres pasadas más del algoritmo?

3	13	8	25	45	23	98	58

8.5. Dada la siguiente lista

47	3	21	32	56	92

Después de dos pasadas de un algoritmo de ordenación, el array se ha quedado dispuesto así

3	21	47	32	56	92

¿Qué algoritmo de ordenación se está utilizando (selección, burbuja o inserción)? Justifique la respuesta.

8.6. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de ordenación *Shell* encuentre las pasadas y los intercambios que se realizan para su ordenación.

8	43	17	6	40	16	18	97	11	7

8.7. Partiendo del mismo array que en el Ejercicio 8.6, encuentre las particiones e intercambios que realiza el algoritmo de ordenación quicksort para su ordenación.

8.8. Un array de registros se quiere ordenar según el campo clave *fecha de nacimiento*. Dicho campo consta de tres subcampos: dia, mes y año de 2, 2 y 4 dígitos respectivamente. Adaptar el método de ordenación radixsort a esta ordenación.

8.9. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número 88.

8	13	17	26	44	56	88	97

Igual búsqueda pero para el número 20.

8.10. Escribir la función de ordenación correspondiente al método radixsort para poner en orden alfabético una lista de *n* nombres.

8.11. Escribir una función de búsqueda binaria aplicado a un array ordenado descendentemente.

8.12. Supongamos que se tiene una secuencia de n números que deben ser clasificados:

1. Utilizar el método de *Shell*, ¿cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si:
   - Ya está clasificado.
   - Está en orden inverso.
2. Repetir el paso 1 para el método de Quicksort.

PROBLEMAS

8.1. Un método de ordenación muy simple, pero no muy eficiente, de elementos x_1, x_2, x_3, ... x_n en orden ascendente es el siguiente:

Paso 1: Localizar el elemento más pequeño de la lista x_1 a x_n; intercambiarlo con x_1.
Paso 2: Localizar el elemento más pequeño de la lista x_2 a x_n, intercambiarlo con x_2.
Paso 3: Localizar el elemento más pequeño de la lista x_3 a x_n, intercambiarlo con x_n.

En el último paso, los dos últimos elementos se comparan e intercambian, si es necesario, y la ordenación se termina. Escribir un programa para ordenar una lista de elementos, siguiendo este método.

8.2. Dado un vector *x* de *n* elementos reales, donde *n* es impar, diseñar una función que calcule y devuelva la mediana de ese vector. La mediana es el valor tal que la mitad de los números son mayores que el valor y la otra mitad son menores. Escribir un programa que compruebe la función.

8.3. Se trata de resolver el siguiente problema escolar. Dadas las notas de los alumnos de un colegio en el primer curso de bachillerato, en las diferentes asignaturas (5, por comodidad), se trata de calcular la media de cada alumno, la media de cada asignatura, la media total de la clase y ordenar los alumnos por orden decreciente de notas medias individuales. *Nota:* utilizar como algoritmo de ordenación el método Shell.

8.4. Escribir un programa de consulta de teléfonos. Leer un conjunto de datos de 1.000 nombres y números de teléfono de un archivo que contiene los números en orden aleatorio. Las consultas han de poder realizarse por nombre y por número de teléfono.

8.5. Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria. Una lista A se rellena con 2.000 enteros aleatorios en el rango 0 .. 1.999 y, a continuación, se ordena. Una segunda lista B se rellena con 500 enteros aleatorios en el mismo rango. Los elementos de B se utilizan como claves de los algoritmos de búsqueda.

8.6. Se dispone de dos vectores, *Maestro* y *Esclavo*, del mismo tipo y número de elementos. Se deben imprimir en dos columnas adyacentes. Se ordena el vector *Maestro*, pero siempre que un elemento de *Maestro* se mueva, el elemento correspondiente de *Esclavo* debe moverse también; es decir, cualquier cosa que se haga a *Maestro[i]* debe hacerse a *Esclavo[i]*. Después

de realizar la ordenación se imprimen de nuevo los vectores. Escribir un programa que realice esta tarea. *Nota:* utilizar como algoritmo de ordenación el método Quicksort.

8.7. Cada línea de un archivo de datos contiene información sobre una compañía de informática. La línea contiene el nombre del empleado, las ventas efectuadas por el mismo y el número de años de antigüedad del empleado en la compañía. Escribir un programa que lea la información del archivo de datos y se ordene por ventas de mayor a menor. Visualizar la información ordenada.

8.8. Se desea realizar un programa que realice las siguientes tareas

a) Generar, aleatoriamente, una lista de 999 de números reales en el rango de 0 a 2.000.
b) Ordenar en modo creciente por el método de la burbuja.
c) Ordenar en modo creciente por el método *Shell*.
d) Ordenar en modo creciente por el método *Radixsort*.
e) Buscar si existe el número *x* (leído del teclado) en la lista. Búsqueda debe ser binaria.

Ampliar el programa anterior de modo que pueda obtener y visualizar en el programa principal los siguientes tiempos:

*t*1. Tiempo empleado en ordenar la lista por cada uno de los métodos.
*t*2. Tiempo que se emplearía en ordenar la lista ya ordenada.
*t*3. Tiempo empleado en ordenar la lista ordenada en orden inverso.

8.9. Construir un método que permita ordenar por fechas y de mayor a menor un vector de *n* elementos que contiene datos de contratos (*n* <= 50). Cada elemento del vector debe ser un objeto con los campos día, mes, año y número de contrato. Pueden existir diversos contratos con la misma fecha, pero no números de contrato repetidos. *Nota*: El método a utilizar para ordenar será *Radixsort*.

8.10. Escribir un programa que genere un vector de 10.000 números aleatorios de 1 a 500. Realice la ordenación del vector por dos métodos:

- Binsort
- Radixsort.

Escriba el tiempo empleado en la ordenación de cada método.

8.11. Se leen dos listas de números enteros, A y B de 100 y 60 elementos, respectivamente. Se desea resolver las siguientes tareas:

a) Ordenar aplicando el método de Quicksort cada una de las listas A y B.
b) Crear una lista C por intercalación o mezcla de las listas A y B.
c) Visualizar la lista C ordenada.

CAPÍTULO 9

Algoritmos de ordenación de archivos

Objetivos

Con el estudio de este capítulo usted podrá:

- Manejar archivos como objetos de una clase.
- Conocer la jerarquía de clases definida en el entorno de C++ para el entrada/salida.
- Procesar un archivo con organización secuencial.
- Procesar un archivo de acceso directo.
- Distinguir entre ordenación en memoria y ordenación externa.
- Conocer los algoritmos de ordenación de archivos basados en la mezcla.
- Realizar la ordenación de archivos secuénciales con mezcla múltiple.

Contenido

Conceptos clave

- Acceso secuencial.
- Archivos de texto.
- Flujos.
- Memoria externa.
- Memoria interna.
- Mezcla.
- Ordenación.
- Organización de un archivo.
- Secuencia de Fibonacci.
- Secuencia ordenada.

INTRODUCCIÓN

Los algoritmos de ordenación de arrays o vectores están limitados a una secuencia de datos relativamente pequeña, ya que los datos se guardan en memoria interna. No se pueden aplicar si la cantidad de datos a ordenar no cabe en la memoria principal de la computadora y como consecuencia están almacenados en un dispositivo de memoria externa, tal como un disquete o un disco óptico. Los archivos tienen como finalidad guardar datos de forma permanente, una vez que acaba la aplicación los datos siguen disponibles para que otra aplicación pueda recuperarlos, para su consulta o modificación. Es necesario aplicar nuevas técnicas de ordenación que se complementen con las ya estudiadas. Entre las técnicas más importantes destaca la *mezcla*. Mezclar, significa combinar dos (o más) secuencias en una sola secuencia ordenada por medio de una selección repetida entre los componentes accesibles en ese momento.

El proceso de archivos en C++ se hace mediante el concepto de **flujo** (*streams*) o canal, o también denominado secuencia. Los flujos pueden estar abiertos o cerrados, conducen los datos entre el programa y los dispositivos externos. Con las clases proporcionadas en la biblioteca iostream y fstream se puede tratar todo tipo de archivo

9.1. FLUJOS Y ARCHIVOS

Un ***fichero*** *(archivo)* de datos —o simplemente un **archivo**— es una colección de registros relacionados entre sí con aspectos en común y organizados para un propósito específico. Por ejemplo, un fichero de una clase escolar contiene un conjunto de registros de los estudiantes de esa clase.

Un archivo en una computadora es una estructura diseñada para contener datos. Los datos están organizados de tal modo que pueden ser recuperados fácilmente, actualizados o borrados y almacenados de nuevo en el archivo con todos los cambios realizados.

Según las características del soporte empleado y el modo en que se han organizado los registros, se consideran dos tipos de acceso a los registros de un archivo:

- *acceso secuencial,*
- *acceso directo.*

El *acceso secuencial* implica el acceso a un archivo según el orden de almacenamiento de sus registros, uno tras otro.

El *acceso directo* implica el acceso a un registro determinado, sin que ello implique la consulta de los registros precedentes. Este tipo de acceso sólo es posible con soportes direccionales.

En C++ la entrada/salida se produce por flujos de bytes. Entonces, un archivo en C++ es, sencillamente, una secuencia o flujo de bytes, que son la representación de los datos almacenados.

Un **flujo** (*stream*) es una abstracción que se refiere a una *corriente* de datos que fluyen entre un origen o fuente (*productor*) y un destino o sumidero (*consumidor*). Entre el origen y el destino debe existir una conexión o canal ("*pipe*") por la que circulen los datos. La apertura de un archivo supone establecer la conexión del programa con el dispositivo que contiene al archivo, por el canal que comunica el archivo con el programa van a fluir las secuencias de datos. Abrir un archivo supone crear un objeto que queda asociado con un flujo.

Existen dos formas de flujo: texto y binario. ***Flujos de texto*** se utilizan con caracteres ASCII, mientras que los ***flujos binarios*** se pueden utilizar con cualquier tipo de dato. Los si-

nónimos *extraer* u *obtener* se utilizan, generalmente, para referirse a la entrada de datos de un dispositivo e *inserción* o *colocación* (poner) cuando se refieren a la salida de datos a un dispositivo.

Todo programa C++ tiene flujos disponibles automáticamente para entrada (`cin`) y salida (`cout`), siempre que se incluya el archivo de cabecera *`iostream`*. *`cin`* y *`cout`* son objetos de tipos *`istream`* y *`ostream`* respectivamente. La lectura de caracteres desde el objeto *`cin`* del flujo de entrada estándar es equivalente a la lectura del teclado; la escritura de caracteres con *`cout`* al flujo de salida estándar es equivalente a visualizar estos caracteres en su pantalla. La Tabla 9.1 muestra los objetos de flujos estándar en la biblioteca *`iostream`*.

Tabla 9.1. Objetos estándar de flujo.

Nombre	operador	Clase de flujos	significado
`cin`	`>>`	`istream`	entrada estándar (con *búfer*)
`cout`	`<<`	`ostream`	salida estándar (con *búfer*)
`cerr`	`<<`	`ostream`	error estándar (sin *búfer*)
`clog`	`<<`	`ostream`	error estándar (sin *búfer*)

9.1.1. Las clases de flujo de E/S

La clase `istream` es para entrada de datos desde un flujo de entrada, la clase `ostream` es para salida de datos a un flujo de salida, y la clase `iostream` es para operaciones ordinarias de E/S.

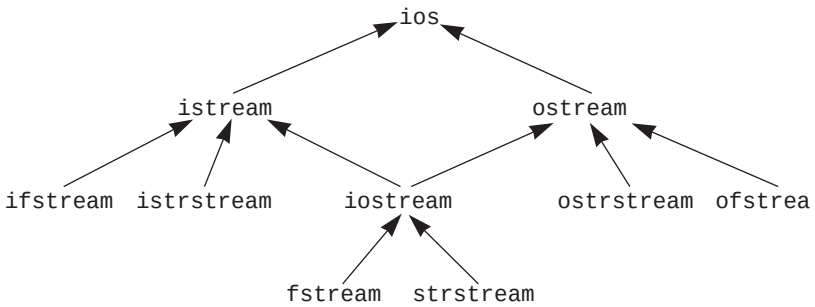


Figura 9.1. Clases derivadas de `ios`.

Las clases `istrstream`, `ostrstream` y `strstream` permiten realizar operaciones de entrada/salida con cadenas de caracteres en lugar de con archivos.

Las tres clases que incluyen la palabra "`fstream`" en su nombre se utilizan para tratamiento de archivos.

La clase `istream` permite definir un flujo de entrada y soporta métodos para entrada *formateada* y *no formateada*. El operador de extracción >>, está sobrecargado para todos los tipos de datos integrales de C++, haciendo posible operaciones de entrada de alto nivel. Su declaración es:

```
class istream: virtual public ios   // archivo iostream
{
public:
    istream& operator >> (int& i);
    istream& operator >> (double& d);
    istream& operator >> (char& c);
    istream& operator >> (char* cad);
    // ...
};
```

La clase `ostream` permite a un usuario definir un flujo de salida y soporta métodos para salidas *formateadas* y *no formateadas*. El operador de inserción `<<` se sobrecarga para todos los tipos de datos integrales. Al igual que `istream`, la clase `ostream` deriva *virtualmente* de la clase `ios` para evitar herencia repetida cuando se declara la clase `iostream`.

```
class ostream: virtual public ios   // archivo iostream
{
public:
    ostream& operator << (int i);
    ostream& operator << (double d);
    ostream& operator << (char c);
    ostream& operator << (const char* cad);
    // ...
};
```

La entrada y salida de datos realizada con los operadores de inserción y extracción pueden formatearse ajustándola a la izquierda o derecha, proporcionando una longitud mínima o máxima, precisión, etc. Normalmente, los manipuladores se utilizan en el centro de una secuencia de inserción o extracción de flujos. Casi todos los manipuladores están incluidos en el archivo de cabecera `iomanip`. La Tabla 9.2 recoge los manipuladores más importantes de flujo de entrada y salida.

Tabla 9.2. Manipuladores de flujo.

Manipulador	Descripción
`endl`	Inserta nueva línea en `ostream`.
`ende`	Inserta carácter nulo (fin de cadena).
`flush`	Limpia `ostream`.
`dec`	Activa conversión decimal.
`oct`	Activa conversión octal.
`hex`	Activa conversión hexadecimal.
`left`	Justifica salida a la izquierda del campo.
`right`	Justifica salida a la derecha del campo.
`setbase(int b)`	Establece base a `b`. Por defecto es decimal.
`setw(arg)`	En la entrada, limita la lectura a `arg` caracteres; en la salida utiliza campo `arg` como mínimo.
`setprecision(arg)`	Fija la precisión de coma flotante a `arg` sitios a la derecha del punto decimal.
`setfill(char car)`	Usa `car` para rellenar caracteres (por omisión blanco).

EJEMPLO 9.1. Uso de algunos manipuladores de flujo .

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int n = 15;
  double dato = 123.4567;
  cout << n << endl;                        // salida en base diez
  cout << oct << n << endl;                 // salida en base ocho
  cout << hex << n << endl;                 // salida en base 16
  cout << setw(8) << "hola" << endl;        // ancho de campo 8
  cout << setw(10);                         // ancho de campo 10
  cout.fill('#');                           // rellena con #
  cout << 34 << endl;      // escribe  34 en base 16, ancho 10 y relleno #
  cout<< setprecision(6)<< dato << endl;    // 6 dígitos de precisión
  return 0;
}
```

9.1.2. Archivos de cabecera

Existen dos archivos de cabecera importantes para clases de flujos de E/S. El archivo de cabecera `<iostream>` declara las clases `istream`, `ostream` e `iostream` para las operaciones de E/S de flujos de entrada y salida estándar. Declara también los objetos `cout`, `cin`, `cerr` y `clog` que se utilizan en la mayoría de los programas C++.

El archivo de cabecera <fstream> declara las clases `ifstream`, `ofstream` y `fstream` para operaciones de E/S a archivos de disco.

9.2. ENTRADAS/SALIDAS POR ARCHIVOS: CLASES `IFSTREAM` Y `OFSTREAM`

Las tres clases siguientes permiten efectuar entradas/salidas en archivos:

- `ifstream`, clase derivada de `istream`; se utiliza para gestionar la lectura de un archivo. Cuando se crea un objeto `ifstream` y se especifica el nombre del archivo, se abre él mismo.
- `ofstream`, clase derivada de `ostream`; gestiona la escritura en un archivo. Los objetos `ofstream` se utilizan para hacer operaciones de salida de archivos. Se declara un objeto `ofstream` cuando se va a escribir un archivo. Si se proporciona un nombre de archivo cuando se declara un objeto `ofstream`, se abre el archivo. Se puede especificar que el archivo se cree en modo binario o en modo texto. Si un objeto de `ofstream` está ya declarado, se puede utilizar la función miembro `open()` para abrir el archivo. Por otra parte, se dispone de la función miembro `close()`, que sirve para cerrar el archivo.
- `fstream`, clase derivada de `iostream`; permite leer y escribir en un archivo. Los objetos `fstream` se utilizan cuando se desea simultanear operaciones de lectura y escritura en el mismo archivo.

Las definiciones de estas clases se encuentran en el archivo de cabecera `fstream`. Es necesario incluir los archivos de cabecera `<iostream>` y `<fstream>` para el proceso de archivos en C++.

9.2.1. Apertura de un archivo

Para comenzar a procesar un archivo la primera operación que hay que realizar es abrir el archivo. La apertura del archivo supone conectar el archivo externo con el programa, e indicar cómo va a ser tratado el archivo: binario, texto.

Las clases `ifstream` y `ofstream` disponen de sendos constructores para abrir el archivo asociado. Por ejemplo:

```
ofstream ft("Cartas.txt", ios::out);
ifstream rf("Cartas.txt", ios::in);
```

Los constructores de estas clases tienen el siguiente formato:

```
ifstream();
ifstream(const char* nombre, int modo = ios::in,
                                        int prot = filebuf::openprot);
ofstream();
ofstream(const char* nombre, int modo = ios::out,
                                        int prot = filebuf::openprot);
```

Además, en ambas clases está el método `open()` para abrir un archivo:

```
void open(const char *nombre, int modo);
```

`nombre:` Contiene el identificador externo del archivo.
`modo:` Contiene el modo en que se va a tratar el archivo. Puede ser una combinación de indicadores del tipo enumerado `open_mode` de la clase `ios` separados por el operador |.

El segundo argumento de `open()` indica el modo de tratar el archivo. Fundamentalmente, se establece si el archivo es para leer, para escribir o para añadir; y si es de texto o binario. Los modos básicos se expresan en la Tabla 9.2. Se puede comprobar el resultado de la operación `open()` con el método `good()`. Por ejemplo:

```
ofstream mifichero;
mifichero.open("Laminas.cat", ios::out);
if (! mifichero.good()) ...
```

También, evaluando directamente la instancia de la clase stream:

```
ifstream fichero ("Cartas.txt", ios::in | ios::binary)
if (!fichero)
  throw "Error al abrir el archivo de lectura";
```

Tabla 9.3. Modos de apertura de un archivo.

Modo	Significado
`ios::in`	Abre para lectura (valor por defecto para `ifstream`). Se sitúa al principio.
`ios::out`	Abre para crear nuevo archivo (valor por defecto para `ofstream`). Se sitúa al principio y pierde los datos.
`ios::app`	Abre para añadir al final.
`ios::trunc`	Crea un archivo para escribir/leer (si ya existe se pierden los datos).
`ios::nocreate`	Si el archivo no está creado falla la apertura.
`ios::noreplace`	Si el archivo no se abre para añadir falla la apertura.
`ios::binary`	Archivo binario.

Al terminar la ejecución del programa podrá ocurrir que haya datos en el buffer de entrada/salida, si no se volcasen en el archivo quedaría este sin las últimas actualizaciones. Los destructores de las clases "`fstream`" cierran el archivo asociado. También, la función miembro `close()` cierra el archivo asociado a un objeto de clase *stream*. El prototipo es: `void close()`.

9.2.2. Funciones de lectura y escritura en archivos

Por el mecanismo de herencia de clases, el operador de flujo << se puede usar con flujos de tipo `ofstream`, al igual que se hace con `cout`, cuando se trabaje en modo texto (no binario). De igual forma, el operador de flujo >> puede ser usado con flujos de tipo `ifstream` como se hace con `cin` cuando se trabaje en modo texto.

El método de salida `put()` sirve para insertar un carácter en cualquier dispositivo de salida. Su prototipo es `ostream& put(char c);`

Las funciones de entrada `get()` (lee un carácter) y `getline()` (lee una cadena) funcionan de igual forma que para el caso de la lectura del objeto `cin`.

La función `eof()` verifica si se ha alcanzado el *final del archivo*, devuelve un valor nulo si no es así. El prototipo de la función es el siguiente: `int eof();`

EJEMPLO 9.2. Tratamiento de archivos de secuencia. El programa solicita al usuario que teclee el texto que se va a almacenar en un archivo; posteriormente se lee el archivo y lo visualiza.

```
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream fichero;
  char nombre[81];
  char c;
```

```
    cout << "Nombre del archivo: ";
    cin >> nombre;
    fichero.open(nombre, ios::out);
    if (! fichero.good())
    {
      cout << "Error al abrir el archivo" << endl;
      exit(-1);
    }
    cout << " Escriba texto: control+z para terminar\n";
    while(cin.get(c))
      fichero.put(c);
    fichero.close();

    cout  << " Lectura y escritura del archivo anterior\n";
    ifstream fichero1(nombre, ios::in);

    while(fichero1.get(c))
      cout.put(c);
    fichero1.close();
    return 0;
  }
```

9.2.3. `write()` y `read()`

`write()` se utiliza para grabar un *buffer* de datos en el archivo. Escribe el número de bytes (caracteres), indicado en el segundo parámetro, desde de cualquier tipo de dato suministrado por el primero. El prototipo de la función es:

```
ostream & write(const char *buffer, int  tamaño);
```

`read()` lee el número de bytes indicado en el parámetro *tamaño* dentro del *buffer* de cualquier tipo de dato suministrado por el primer parámetro. El prototipo de la función es:

```
ostream & read(char * buffer, int  tamaño);
```

La extracción de caracteres continúa hasta que se hayan leído `tamaño` caracteres o se encuentre el fin de fichero.

Estos dos métodos son muy usados para el proceso de archivos en modo binario. También, para guardar y recuperar estructuras, registros y objetos en un archivo; en estos casos se utiliza el operador genérico de conversión `reinterpret_cast`. Su sintaxis es:

```
reinterpret_cast< T > (variable)
```

El operador convierte el tipo de dato de la `variable` al tipo especificado por `T` (normalmente `char*`). Por ejemplo:

```
class Racional { ...} unNumero;
reinterpret_cast< char* > (unNumero);
```

EJEMPLO 9.3. Tratamiento de un archivo binario. Se crea un archivo binario para almacenar los datos del pulsómetro de un atleta.

La clase *Pulsom* guarda la información de una observación: hora y pulsaciones. Se supone que la función *lectura()* genera un objeto *Pulsom* con los datos correctamente asignados. Una vez escrito en esl archivo los datos, el programa visualiza el contenido completo del archivo

```
#include <iostream>
#include <fstream>
using namespace std;

class Pulsom
{
  int hora;
  int pulsaciones;
  ...
};

fstream f;
f.open("ENTRENAM.DAT", ios::out | ios:: binary);
if (!f.good())
{
  cout <<"Error de apertura ";
  exit(1);
}
// lectura de 30 datos del pulsómetro
Pulsom pulsacion;
cout <<"\n--- Datos pulsómetro ---" << endl;

for (int m = 0; m < 30; m++)
{
  pulsacion = lectura()
  f.write ( reinterpret_cast<char *>(&pulsacion), sizeof(Pulsom));
}
f.close();

cout << « Lectura del archivo \n»;
f.open(«ENTRENAM.DAT», ios::in | ios:: binary);
//lectura de los distintos datos del pulsómetro de f
for (int m = 0; m < 30; m++)
{
  f.read( reinterpret_cast<char *>(& pulsacion), sizeof(Pulsom));
  cout <<" Hora  : " << pulsacion.Hora() << endl;
  cout << "Pulsaciones : " << pulsacion.numPuls() << endl;
}
```

9.2.4. Funciones de posicionamiento

Las funciones `seekg()` y `seekp()`permiten tratar un archivo en C++ como un array que es una estructura de datos de acceso aleatorio. `seekg()` está definido en la clase `istream` y, por tanto, heredado en `ifstream`, sitúa *el puntero* del archivo de entrada en una posición aleatoria;

`seekp()`está definido en `ostream` y lo hereda `ofstream`, sitúa *el pointer* del archivo de salida en una posición aleatoria. Los prototipos son:

```
istream & seekg (long desplazamiento);
istream & seekg (long desplazamiento, seek_dir origen);
ostream & seekp (long desplazamiento);
ostream & seekp (long desplazamiento, seek_dir origen);
```

El primer formato de ambas funciones cambia la posición de modo absoluto, y el segundo lo hace modo relativo, dependiendo de tres valores.

El primer parámetro indica el desplazamiento, absoluto o relativo dependiendo del formato. El segundo parámetro es el origen del desplazamiento. Este origen puede tomar tres valores que se encuentran en la clase `ios`:

```
class ios
{
  enum seek_dir
  {
    beg = 0;   // Cuenta desde el inicio del archivo.
    cur = 1;   // Cuenta desde la posición actual del archivo.
    end = 2;   // Cuenta desde el final del archivo.
  }
```

La posición actual del cursor de un archivo se puede obtener llamando a la función `tellg()` de la clase `istream`, o `tellp()` de la clase `ostream`. Sus prototipos son:

```
long int tellg();
long int tellp();
```

La función `gcount()` devuelve el número de caracteres o bytes que se han leído en la última lectura realizada en un flujo de entrada. Su formato es:

```
int gcount();
```

EJEMPLO 9.4. Proceso de archivos binarios con las funciones read y wr ite.

El usuario introduce el nombre de un archivo que se procesa como un archivo binario, y se copia en otro archivo.

```
#include <iostream>
#include <fstream>
using namespace std;

char nom[81], buffer[81];
ifstream f;      // fichero de entrada
ofstream g;      // fichero de salida

cout << "Nombre del archivo de entrada: ";
cin >> nom;
f.open(nom, ios::binary);
if(!f.good())
```

```
{
  cout <<"El archivo " << nom << " no existe \n";
  exit (1);
}
// Crear o sobreescribir el fichero g en binario

cout << "Nombre del archivo de salida: ";
cin >> nom;
g.open(nom, ios::binary);
if(!g.good())
{
  cout <<"El archivo " << nom << " no se puede crear\n";
  exit (1);
}
do
{
  f.read(buffer, 81);                // lectura de f
  g.write(buffer, f.gcount());       // escritura de los bytes leídos
} while(f.gcount(> 0);               // repetir hasta que no se leen datos
f.close();
g.close();
```

9.3. ORDENACIÓN DE UN ARCHIVO. MÉTODOS DE ORDENACIÓN EXTERNA

Para ordenar secuencias grandes de elementos, que posiblemente no pueden almacenarse en memoria interna, se aplican los *algoritmos de ordenación externa*. La ordenación externa está ligada con los archivos y los dispositivos en que se encuentran, al leer el archivo para realizar la ordenación el tiempo de lectura de los registros es notablemente mayor que el tiempo que se tarda en realizar las operaciones de ordenación.

Un archivo está formado por una secuencia de `n` elementos. Un elemento es un número entero, un registro, ..., un objeto. Los elementos, *R(i)*, pueden ser comparables, o no; en general, un elemento es comparable si dispone de una clave *K(i)*, mediante la que se puede hacer comparaciones con otro elemento. El archivo está ordenado respecto a la clave si:

$$\forall\ i < j \Rightarrow K(i) < K(j)$$

En la ordenación interna los elementos están en todo momento en memoria principal, salvo cuando termina el proceso que se vuelven a almacenar en el dispositivo externo. La memoria principal es limitada, por contra, el número de registros u objetos de los que puede constar una archivo externo es muy elevado.

La ordenación de los registros (objetos) de una archivo mediante archivos auxiliares se denomina **ordenación externa**. Los distintos algoritmos de ordenación externa utilizan el esquema general de **separación** en tramos y **fusión** o mezcla. Por separación se entiende la distribución de secuencias de registros ordenados en varios archivos; por fusión se entiende la mezcla de dos o más secuencias ordenadas en una única secuencia ordenada. Variaciones de este esquema general dan lugar a diferentes algoritmos de ordenación externa.

Ejecución

El tiempo de un algoritmo de ordenación de registros de un archivo, *ordenación externa*, depende notablemente del dispositivo de almacenamiento. Los algoritmos repiten consecutivamente una fase *separación* en tramos y otra de *mezcla* que da lugar a tramos ordenados cada vez mas largos. Se considera únicamente el acceso secuencial a los registros (objetos).

9.4. MEZCLA DIRECTA

Éste es el método más simple de ordenación externa, consiste en repetir el esquema de separación en secuencias ordenadas de registros y su mezcla, para originar secuencias ordenadas de doble longitud. Se opera con el archivo original y dos archivos auxiliares. El proceso consiste:

1. Separar los registros individuales del archivo original *O* en dos archivos *F1* y *F2*.
2. Mezclar los archivos *F1* y *F2* combinando registros aislados (según sus claves) y formando pares ordenados que son escritos en el archivo *O*.
3. Separar pares de registros del archivo original O en los archivos *F1* y *F2*.
4. Mezclar *F1* y *F2* combinando pares de registros y formando cuádruplos ordenados que son escritos en el archivo *O*.
5. Se repiten los pasos de separación y mezcla, combinando cuádruplos para formar óctuplos ordenados. En cada paso de separación y mezcla se duplica el tamaño de las subsecuencias mezcladas, así hasta que la longitud de la subsecuencia sea la que tiene el archivo y en ese momento el archivo original *O* está ordenado.

El método de ordenación externa *mezcla directa* en el paso *i* obtiene secuencias ordenadas de longitud 2^i. Termina cuando la longitud de la secuencia es igual al número de registros del archivo.

EJEMPLO 9.5. Realizar un seguimiento del algoritmo de ordenación externa mezcla directa. Se supone un archivo formado por registros que tienen un campo clave de tipo entero; las claves son las siguientes:

```
34  23  12  59  73  44  8  19  28  51
```

Se realizan los pasos del algoritmo de mezcla directa para ordenar la secuencia. Se considera el archivo `O` como el original, `F1` y `F2` archivos auxiliares.

Pasada 1

Separación:

```
F1: 34  12  73  8  28
F2: 23  59  44  19  51
```

Mezcla formando duplos ordenados:

```
O: 23  34  12  59  44  73  8  19  28  51
```

Pasada 2

Separación de duplos:

```
F1: 23  34  44  73  28  51
F2: 12  59  8  19
```

Mezcla formando cuádruplos ordenados:

```
O: 12  23  34  59  8  19  44  73  28  51
```

Pasada 3

Separación de cuádruplos:

```
F1: 12  23  34  59  28  51
F2: 8  19  44  73
```

Mezcla formando óctuplos ordenados:

```
O: 8  12  19  23  34  44  59  73  28  51
```

Pasada 4

Separación de óctuplos:

```
F1: 8  12  19  23  34  44  59  73
F2: 28  51
```

Mezcla con la que ya se obtiene el archivo ordenado:

```
O: 8  12  19  23  28  34  44  51  59  73
```

En el Ejemplo 9.5 han sido necesarias cuatro pasadas, cada pasada constituye una fase de separación y otra de mezcla.

Después de `i` pasadas se tiene el archivo *O* con subsecuencias ordenadas de longitud 2^i, si $2^i \geq n$, siendo `n` el número de registros, el archivo estará ordenado. El número de pasadas que realiza el algoritmo se obtiene tomando logaritmos, $i \geq Log_2 n$, `Log n` pasadas serán suficientes. Cada pasada escribe el total de registros, por ello el número total de movimientos es `O(nLog n)`.

Como el tiempo de las comparaciones, realizadas en la fase de fusión, es insignificante respecto a las operaciones de movimiento de registros en los archivos externos, no resulta interesante analizar el número de comparaciones.

9.4.1. Codificación del algoritmo de mezcla directa

Las funciones `distribuir()` y `mezclar()`implementan las dos partes fundamentales del algoritmo. La primera *separa* secuencias de registros del archivo original en los dos archivos auxiliares. La segunda *mezcla* secuencias de los dos archivos auxiliares, y la secuencia resultante se escribe en el archivo original. Cada pasada realiza una llamada a `distribuir()` y `mezclar()`, obteniendo una secuencia del doble de longitud de registros ordenados. El algoritmo termina cuando la longitud de la secuencia iguala al número de registros.

La función `distribuir()` utiliza la función auxiliar, `subsecuencia()`, para escribir una secuencia de un número especificado de registros en un archivo auxiliar. Esta función se llama alternativamente con el archivo `F1` y `F2`, de esa forma se consigue distribuir el archivo en secuencias de longitud fija. La codificación asume que los registros son números enteros perfectamente formateados.

```
#include <iostream>
#include <fstream>

using namespace std;
typedef const charx nomArchi;

nomArch fichero = "origen";
nomArch fileAux1 = "auxiliar1";
nomArch fileAux2 = "auxiliar2";
    // función que controla el algoritmo de ordenación
void mezclaDirecta(nomArch f)
{
  int longSec;
  int numReg;
  nomArch f1= "auxiliar1";
  nomArch f2= "auxiliar2";
  numReg = cuentaReg(f);     // determina número de registros
  longSec = 1;

  while (longSec < numReg)
  {
    distribuir(f, f1, f2, longSec, numReg);
    mezclar(f1, f2, f, longSec, numReg);
  }
}
    // función que separa secuencias ordenadas de registros
void distribuir(nomArch nf, nomArch nf1, nomArch nf2, int lonSec, int
numReg)
{
  int numSec, resto, i;

  numSec = numReg /(2*lonSec);
  resto =  numReg %(2*lonSec);

  ifstream f(nf);
  ofstream f1(nf1);
  ofstream f2(nf2);

  if (f.bad()||f1.bad()||f2.bad())
    throw " Error en el proceso de separación ";

  for (i = 1; i <= numSec; i++)
  {
    subSecuencia(f, f1, lonSec);
    subSecuencia(f, f2, lonSec);
  }
  /*
    Se procesa el resto de registros del archivo
  */
  if (resto > lonSec)
    resto -= lonSec;
```

```
  else
  {
    lonSec = resto;
    resto = 0;
  }

  subSecuencia(f, f1, lonSec);
  subSecuencia(f, f2, resto);

  f1.close(); f2.close(); f.close();
}
  // lee de f una secuencia y la escribe en t
 void subSecuencia(ifstream f, ofstream t, int longSec)
 {
   for (int j = 1; j <= longSec; j++)
   {
      int reg;
      f >> reg;   // lee (extrae) del archivo asociado a f
      t << reg <<" ";   //  escribe en archivo asociato a t
   }
 }
      // mezcla pares de secuencias ordenadas
void mezclar(nomArch nf1, nomArch nf2, nomArch nf, int& lonSec, int numReg)

{
  int numSec, resto, i, j, k;
  int reg1, reg2;

  numSec = numReg /(2 * lonSec);   // número de subsecuencias
  resto =  numReg %(2 * lonSec);

  ofstream f(nf);
  ifstream f1(nf1);
  ifstream f2(nf2);

  f1 >> reg1;
  f2 >> reg2;

  for (int s = 1; s <= numSec + 1; s++)
  {
    int n1, n2;
    n1 = n2 = lonsec;
    if (s == numSec+1)
    {    // proceso de los registros de la subsecuencia incompleta
      if (resto > lonSec)
        n2 = resto - lonSec;
      else
      {
        n1 = resto;
        n2 = 0;
      }
    }

    i = j = 1;
    while (i <= n1 && j <= n2)
    {
      int actual;
```

```
        if (reg1 < reg2)
        {
          actual = r1;
          f1 >> reg1;
          i++;
        }
        else
        {
          actual = r2;
          f2 >> reg2;
          j++;
        }
        f << actual <<" ";
      }
      /*
        Los registros no procesados se escriben directamente
      */
      for (k = i; k <= n1; k++)
      {
        f << reg1 <<" ";
        f1 >> reg1;
      }

      for (k = j; k <= n2; k++)
      {
        f << reg2 <<" ";
        f2 >> reg2;
      }
    }

    lonSec *= 2;
    f.close(); f1.close();f2.close();
  }
    // función para determinar el número de registros
  int cuentaReg(nomArch nf)
  {
    int k = 0;
    int registro;
    ifstream f(nf);
    while (!f.eof())
    {
      f >> registro;
      k++;
    }
    f.close();
    return k;
  }
```

9.5. FUSIÓN NATURAL

El método de fusión natural mejora el tiempo de ejecución de la mezcla directa. Introduce una pequeña variación en el algoritmo de ordenación, respecto a la longitud de las secuencias de registros. La longitud de las secuencias en la mezcla directa es fija, son múltiplos de dos: `1,`

2, 4, 8, 16, ... de tal forma que el número de *pasadas* a realizar es fijo, dependiente del número de registros. La longitud de las secuencias en el algoritmo de fusión natural no es fija, es la máxima posible en cuanto a que considera una secuencia una sucesión de elementos ordenados. Estas secuencias también se mezclan y dan lugar a otra secuencia ordenada.

El método de ordenación e xterna *fusión natural,* en todo momento distr ibuye secuencias ordenadas (tramos) lo mas largas posibles y mezcla secuencias ordenadas lo más largas posibles.

9.5.1. Algoritmo de la fusión natural

La característica fundamental de este método es la mezcla de secuencias ordenadas máximas, o simplemente tramo máximo. Una secuencia ordenada a_i ... a_j es tal que:

$a_k \leq a_{k+1}$ para k = i ... j–1
$a_{i-1} > a_i$
$a_j > a_{j+1}$

Por ejemplo, en esta lista de claves enteras: 4 9 11 5 8 12 9 17 18 21 26 18 los tramos máximo que se encuentran: 4 9 11; **5** 8 12; **9** 17 18 21 26; **18**. La ruptura de un tramo ocurre cuando la clave actual es menor que la clave anterior.

El método de fusión natural funde tramos máximos en lugar de secuencias de tamaño fijo y predeterminado. Esto hace que se optimice el número de pasadas a realizar. Los tramos tienen la propiedad de que si se tiene dos listas de *n* tramos cada una y se mezclan, se produce una lista de *n* tramos como consecuencia, el número total de tramos disminuye, al menos se divide por dos, en cada pasada del algoritmo *fusión natural*.

EJEMPLO 9.6. Un archiv o está f ormado por registros que tienen un campo cla ve de tipo entero, suponiendo que éstas son las siguientes:

17 31 5 59 33 41 43 67 11 23 29 47 3 7 71 2 19 57 37 61

Se realizan los pasos que sigue el algoritmo de fusión natural para ordenar la secuencia. El archivo O es el original, los archivos auxiliares son F1 y F2.

Los tramos máximo de la lista de claves se separan por el carácter ´:

17 31´ 5 59´ 33 41 43 67´ 11 23 29 47´ 3 7 71´ 2 19 57´ 37 61

Pasada 1

Separación:

F1: 17 31 33 41 43 67´ 3 7 71´ 37 61
F2: 5 59´ 11 23 29 47´ 2 19 57

Se puede observar que de manera natural, al distribuirse por tramos, la secuencia 17 31 se ha expandido junto a 33 41 43 67 y han formado una única secuencia ordenada.

Mezcla o fusión de tramos:

```
O: 5 17 31 33 41 43 59 67´ 3 7 11 23 29 47 71´ 2 19 37 57 61
```

Pasada 2

Separación:

```
F1: 5 17 31 33 41 43 59 67´ 2 19 37 57 61
F2: 3 7 11 23 29 47 71
```

Mezcla o fusión de tramos:

```
O: 3 5 7 11 17 23 29 31 33 41 43 47 59 67 71´ 2 19 37 57 61
```

Pasada 3

Separación:

```
F1: 3 5 7 11 17 23 29 31 33 41 43 47 59 67 71
F2: 2 19 37 57 61
```

Mezcla o fusión de tramos máximos:

```
O: 2 3 5 7 11 17 19 23 29 31 33 37 41 43 47 57 59 61 67 71
```

La lista ya está ordenada, la longitud del tramo máximo es igual al número de registros del archivo. Han sido necesarias tres pasadas, cada pasada constituye una fase de separación y otra de mezcla o fusión.

El algoritmo, al igual que en el método mezcla directa, descompone las acciones que realiza en dos rutinas: `separarNatural()` y `mezclaNatural()`. La primera *separa* tramos máximos del archivo original en los dos archivos auxiliares. La segunda *mezcla* tramos máximos de los dos archivos auxiliares y la escribe en el archivo original. El algoritmo termina cuando hay un único tramo, entonces el archivo está ordenado.

```
Ordenación MezclaNatural

inicio
  repetir
    separarNatural(f, f1, f2)
    mezclaNatural(f, f1, f2, numeroTramos)
  hasta numerosTramos = 1
fin
```

9.6. MEZCLA EQUILIBRADA MÚLTIPLE

La eficiencia de los métodos de ordenación externa es directamente proporcional al número de pasadas. Para aumentar la eficiencia hay que reducir el número de pasadas, de esa forma se reducen el número de operaciones de entrada/salida en dispositivos externos. El algoritmo *fu-*

sión natural es más eficiente que el de mezcla directa ya que, en general, reduce el número de pasadas. Ambos algoritmos tienen en común las dos fases: *separación y mezcla*; y también que utilizan dos archivos auxiliares.

Otra forma de reducir el número de *pasadas* es incrementar el número de archivos auxiliares. Supóngase que se tiene `w` tramos distribuidos equitativamente en `m` archivos, la primera *pasada* mezcla los `w` tramos y da lugar a `w/m` tramos. En la siguiente *pasada*, la mezcla de los `w/m` tramos da lugar a w/m^2 tramos, en la siguiente se reducen a w/m^3, después de `i` *pasadas* quedarán w/m^i tramos. En definitiva, se ha realizado una mezcla de de *m-uples* tramos, una mezcla múltiple.

Para determinar la complejidad del algoritmo mediante mezcla de *m-uples* tramos, se supone que, *en el peor de los casos*, un archivo de `n` registros tiene `n` tramos iniciales; entonces, el número de pasadas necesarias para la ordenación completa es $Log_m\ n$, como cada pasada realiza `n` operaciones de entrada/salida con los registros, la eficiencia es $O(nLog_m\ n)$. La mejora obtenida, en cuanto a la disminución de las pasadas necesarias para la ordenación, hace que los movimientos o transferencias de cada registro sea $Log_2 m$ veces menos .

La sucesión del número de tramos, suponiendo tanto tramos iniciales como registros:

$$n,\ n/m,\ n/m^2,\quad n/m^3\ \ldots\ n/m^t = 1$$

tomando logaritmos en base `m` se calcula el número de tramos `t`:

$$n = m^t\ ;\ Log_m\ n = Log_m\ m^t \Rightarrow t = Log_m\ n$$

Nota de programación

El algoritmo mezcla equilibrada múltiple emplea un número par de archivos para ordenar `n` registros. La utilización de un número elevado de archivos no todas las aplicaciones puedan soportarlo.

9.6.1. Algoritmo de mezcla equilibrada múltiple

La mezcla equilibrada múltiple utiliza `m` archivos auxiliares, de los que `m/2` son de entrada y `m/2` de salida. Inicialmente, se distribuyen los tramos del archivo origen en los `m/2` archivos auxiliares. A partir de esta distribución, se repiten los procesos de mezcla reduciendo a la mitad el número de tramos, hasta que queda un único tramo. La principal característica del algoritmo es que el proceso de mezcla se realiza en una sola fase, en lugar de las dos fases (*separación, fusión*) de los algoritmos mezcla directa y fusión natural. Los pasos que sigue el algoritmo:

1. Distribuir registros del archivo original por tramos en los `m/2` primeros archivos auxiliares. A continuación, éstos se consideran archivos de entrada.
2. Mezclar tramos de los `m/2` archivos de entrada y escribirlos consecutivamente en los `m/2` archivos de salida.
3. Cambiar la finalidad de los archivos, los de entrada pasan a ser de salida y viceversa. Repetir a partir del segundo paso hasta que quede un único tramo, entonces la secuencia está ordenada.

9.6.2. Archivos auxiliares para realizar la mezcla equilibrada múltiple

El principal cambio, en cuanto a las variables que se utilizan, está en que los flujos correspondientes a los archivos auxiliares se agrupan en un array. La constante `N` representa el número de archivos auxiliares (flujos); la constante `N2` es el número de archivos de entrada, la mitad de `N`, también es el índice inicial de los flujos de salida.

```
const int N = 6;
const int N2 = N/2;
fstream f [N];
```

La variable `f[]` representa a los archivos auxiliares, alternativamente, la primera mitad y la segunda irán cambiando su cometido, entrada o salida.

9.6.3. Cambio de finalidad de un archivo: *entrada* ↔ *salida*

La forma de cambiar la finalidad de los archivos (*entrada* ↔ *salida*) se hace mediante una tabla de correspondencia entre índices de archivo. De tal forma que en vez de acceder a un archivo por el índice del array se accede por la tabla, la cual cambia alternativamente los índices de los archivos y de esa forma pasan de ser archivo de entrada a ser de salida (*flujos de entrada, flujos de salida*).

`int c[N];` *Tabla de índices de archivo.*

`c[i] = i  ∀i ∈ 0 .. N - 1.` *Índices iniciales.*

Como consecuencia los archivos de entrada son,

```
f[c[0]], f[c[1]],..., f[c[N2-1]];
```

y los ficheros de salida son la otra mitad,

```
f[c[N2]], f[c[N2+1]], … f[c[N-1]]
```

Para realizar el cambio de archivo de entrada por el de salida se intercambian los valores de las dos mitades de la tabla de correspondencia:

```
c[0] ↔ c[N2]
c[1] ↔ c[N2+1]
. . .
c[N2] ↔ c[N-1]
```

En definitiva, con la tabla `c[]` siempre se accede de igual forma a los archivos, lo que cambia son los índices de `c[]`.

Al mezclar tramos de los archivos de entrada no se alcanza el fin de tramo en todos los archivos al mismo tiempo. Un tramo termina cuando es *fin de archivo*, o bien la siguiente clave es menor que la actual, en cualquier caso el archivo que le corresponde ha de quedar inactivo. Para despreocuparse de si el archivo está activo o no, se utiliza otro array cuyas posiciones indican si el archivo correspondiente al índice está activo. Como en algún momento del proceso de mezcla no todos los archivos de entrada están activos, se utiliza, sólo para archivos de entrada, la tabla `cd[]`que tiene los índices de los archivos de entrada activos.

9.6.4. Control del número de tramos

El primer paso del algoritmo realiza la distribución de los tramos del archivo original en los archivos de entrada, a la vez determina el número de tramos del archivo. Es importante conocer el número de tramos ya que cuando queda sólo uno el archivo está ordenado y éste será el archivo `f[c[0]]`.

La ejecución del algoritmo de ordenación va reduciendo el número de tramos, llega un momento en el que el número de tramos a mezclar es menor que el número de archivos de entrada. La variable `k1` controla el número de archivos de entrada, cuando el número de tramos, `t`, es mayor o igual que la mitad de los archivos, el valor de `k1` es justamente la mitad; cuando `t` es menor que dicha mitad, `k1` será igual a `t` y, por último, cuando `t` es `1`, `k1` también es `1` y el archivo `f[c[0]]` está ya ordenado.

9.6.5. Codificación

El programa asume, por simplicidad, que el archivo que se ordena está formado por registros de tipo entero. El proceso de mezcla de un tramo de cada uno de los `m/2` archivos (*flujos*) de entrada, comienza leyendo de cada archivo un registro (clave de tipo entero) y guardándolo en `rs[]`. La mezcla selecciona repetidamente el registro menor, con una llamada a la función `minimo();` el proceso de selección tiene en cuenta que los registros, que se encuentran en el array `rs[],` se correspondan con archivos activos. Cada vez que es seleccionado un registro, se lee el siguiente registro del mismo archivo de entrada, a no ser que haya acabado el tramo; así hasta que termina la mezcla de todos los tramos. La función `fintramos()` determina si se ha llegado al final de todos los tramos involucrados en la mezcla.

La codificación completa se encuentra en la página web del libro. Únicamente se escribe la función `mezclaEqMple` que controla el proceso de ordenación.

```
  // número de archivos utilizados
const int N = 6;
const int N2 = N/2;

const int MAXT = 32767;
  // flujos utilizados
fstream* f [N];
ifstream forigen; // flujo para asociar al archivo original
  // nombre de los archivos utilizados y del original
const char* nomf[N] = {"ar1", "ar2", "ar3", "ar4","ar5","ar6"};
const char* nomOrigen = "fileorg";
  //
typedef int Registro;
Registro rs[N2];
int c[N], cd[N];
bool actvs[N];
 // función que realiza el proceso de ordenación
void mezclaEqMple()
{
   int i, j, k, k1, t;
   Registro anterior;
      // distribución inicial de los tramos del archivo origen
   t = distribuir();
```

```
for (i = 0; i < N; i++)
{
   c[i] = i;
}
 // bucle externo, termina cuando numero de tramos, t, es 1
do {
   k1 = (t < N2) ? t : N2;
   k = k1;
   for (i = 0; i < k1; i++)
   {
      f[c[i]] = new fstream( nomf[c[i]],ios::in);
      if (f[c[i]] -> bad())
        throw " Error al abrir archivo en modo lectura";
        cd[i] = c[i];
   }
   t = 0;  // Número de tramos mezclados
   j = N2; // índice del archivo de salida
   for (i = j; i < N; i++)
   {
      f[c[i]] = new fstream( nomf[c[i]],ios::out);
      if (f[c[i]] -> bad())
        throw " Error al abrir archivo en modo salida";
   }
   leerItems(rs, k1, cd);
   while (k1 > 0)
   {
     t++;  // mezcla de un nuevo tramo
     for (i = 0; i < k1; i++)  actvs[i] = true;
     while (!finDeTramos(actvs, k))
     {
        int indiceMin;
        indiceMin = minimo(rs, actvs, k);
        (*f[c[j]]) << rs[indiceMin] << " " ;
        anterior = rs[indiceMin];
        (*f[cd[indiceMin]]) >> rs[indiceMin];
        if (f[cd[indiceMin]]-> eof())  // archivo inactivo
        {
           k1--;
           actvs[indiceMin] = false;
           f[cd[indiceMin]]->close();   // archivo inactivo
           cd[indiceMin] = cd[k1];      // cambia k1
           rs[indiceMin] = rs[k1];      // por el de indiceMin
           actvs[indiceMin] = actvs[k1];
           actvs[k1] =false;    // no se accede a la posición k1
        }
        else
        {
           if (anterior > rs[indiceMin])   // fin de tramo
              actvs[indiceMin] = false;
        }
     }  // fin de while
     j = (j < N - 1 ) ? j + 1 : N2 ; // siguiente archivo salida
   }  // fin de while (k1>0)
     // cierre de archivos

   for (i = N2; i < N; i++)
```

```
         f[c[i]]->close();
      /*
        Cambio en la finalidad de los archivos, se tiene
        en cuenta el índice 0
      */
      for (i = 0; i < N2; i++)
      {
         int a;
         a         = c[i];
         c[i]      = c[i + N2];
         c[i + N2] = a;
      }
    } while (t > 1); // fin del bucle do - while
}

  //    Esta función reparte los tramos del archivo origen en los
  //    ficheros de salida.
int distribuir() { ... }
 // lee nf registros de los archivos de índice cd
void leerItems(Registro r[], int nf, int cd[]) { ... }
 // devuelve índice del menor registro activo
int minimo(Registro r[], bool activo[], int n) { ... }
 // devuelve true si los n archivos no están activos
bool finDeTramos(bool activo[], int n) { ... }
```

9.7. MÉTODO POLIFÁSICO DE ORDENACIÓN EXTERNA

La mezcla equilibrada múltiple puede mejorarse consiguiendo ordenar `m` tramos con sólo `m+1` archivos, para ello hay que abandonar la idea rígida de *pasada* en la que la finalidad de los archivos de entrada no cambia hasta que se leen todos los archivos.

El *método polifásico* utiliza `m` archivos auxiliares para ordenar `n` registros de un archivo. La característica que marca la diferencia de este método respecto a los otros es que, continuamente, se consideran `m-1` archivos de entrada desde los que se mezclan registros, y `1` archivo de salida. En el momento que uno de los archivos de entrada alcanza su final hay un cambio de cometido, pasa a ser considerado como archivo de salida, el archivo que en ese momento era de salida pasa a ser de entrada y la mezcla de tramos continúa. La sucesión de pasadas continúa hasta alcanzar el archivo ordenado.

Cabe recordar la propiedad base de todos los métodos de mezcla: *"la mezcla de k tramos de los archivos de entrada se transforma en k tramos en el archivo de salida"*.

A recordar

La mezcla polifásica se caracteriza por realizar una mezcla continua de tramos. De tal forma que si se utilizan `m` archivos, en un momento dado uno de ellos es archivo de salida y los otros `m-1` archivos son de entrada. Durante el proceso, cuando se alcanza el *registro de fin de fichero* en un archivo de entrada, éste pasa a ser de salida, el anterior archivo de salida pasa a ser de entrada y la mezcla continúa. La dificultad del método es que el número de tramos iniciales debe pertenecer a una sucesión de números que depende de `m`.

9.7.1. Mezcla polifásica con m = 3 archivos

A continuación, se muestra un ejemplo que parte de un archivo original de 55 tramos. El número de archivos auxiliares es m = 3; entonces, en todo momento 2 archivos son de entrada y un tercero de salida.

Inicialmente, el método distribuye los tramos, de manera no uniforme, en los archivos F1, F2. Suponiendo que se dispone de 55 tramos, se sitúan 34 en F1 y 21 en F2; el archivo F3 es, inicialmente, de salida. Empieza la mezcla, y 21 tramos de F1 se fusionan con los 21 tramos de F2 dando lugar a 21 tramos en F3. En este instante F1 tiene 13 tramos, en F2 se ha alcanzado el *fin de archivo* y en F3 hay 21 tramos; F2 pasa a ser archivo de salida y la mezcla continúa entre F1 y F3. Ahora se mezclan 13 tramos de F1 con 13 tramos de F3, dando lugar a 13 tramos que se escriben en F2; ahora se alcanza *el fin de archivo* de F1. En el archivo F2 hay 13 tramos y en el F3 quedan 8 tramos, continúa la mezcla con F1 como archivo de salida. En la nueva pasada se mezclan 8 tramos que se escriben en F1, quedan 5 tramos en F2 y en F3 ninguno ya que se ha alcanzado *el fin de archivo*. El proceso sigue hasta que queda un único tramo y el archivo ha quedado ordenado. La Figura 9.2 muestra las sucesivas pasadas y los tramos de cada archivo hasta que termina la ordenación.

Es evidente que se ha partido de una distribución óptima. Además, si se escribe la sucesión de tramos mezclados a partir de la distribución inicial: 21, 13, 8, 5, 3, 2, 1, 1; es justamente la sucesión de los números de *fibonacci*:

$f_{i+1} = f_i + f_{i-1}\ \forall i \geq 1\ ,,\ f_1 = 1,\ f_0 = 0$

Entonces, si el número de tramos iniciales f_k es tal que es un número de fibonacci, la mejor forma de hacer la distribución inicial es según la sucesión de fibonacci, f_{k-1} y f_{k-2}. Ahora bien, el archivo origen no siempre dispone de un número de tramos perteneciente a la sucesión de fibonacci, en esos casos se recurre a escribir tramos ficticios para conseguir un número de la secuencia de fibonacci.

Tramos iniciales		Despu s de cada pasada							
		F1+F2	F1+F3	F2+F3	F1+F2	F1+F3	F2+F3	F1+F2	F1+F3
F1	34	13	0	8	3	0	2	1	0
F2	21	0	13	5	0	3	1	0	1
F3	0	21	8	0	5	2	0	1	0

Figura 9.2. Tramos en cada archivo después de cada pasada en la mezcla polifásica con 3 archiv os.

9.7.2. Mezcla polifásica con m = 4 archivos

Se puede extender el proceso de mezcla polifásica a un número mayor de archivos. Por ejemplo, si se parte de un archivo con 31 tramos y se utilizan m = 4 archivos para la mezcla polifásica, la distribución inicial y los tramos de cada archivo se muestran en la Figura 9.3.

La Figura 9.4 muestra en forma tabular la distribución perfecta de los tramos en los archivos para cada pasada. Es necesario conocer, previamente, el número de pasadas que se van a

Tramos iniciales		Despu s de cada pasada				
		F1+F2 +F3	F1+F2 +F4	F1+F3 +F4	F2+F3 +F4	F1+F2 +F3
F1	13	6	2	0	1	0
F2	11	4	0	2	1	0
F3	7	0	4	2	1	0
F4	0	7	3	1	0	1

Figura 9.3. Tramos en cada archivo en la mezcla polifásica con 4 archivos.

realizar; el primer valor del índice *L* es el número de pasadas, y va decrementando hasta tomar el valor *0*.

De la Figura 9.4 se deducen ciertas relaciones entre t_1 , t_2, t_3 en un nivel:

$$t_3^{L+1)} = t_1^{L)}$$
$$t_2^{L+1)} = t_1^{L)} + t_3^{L)}$$
$$t_1^{L+1)} = t_1^{L)} + t_2^{L)}$$
$$\forall\ L > 0 \quad y\ t_1^{0)} = 1\ ,\ t_2^{0)} = 0,\ t_3^{0)} = 0$$

Estas relaciones permiten encontrar la distribución inicial ideal cuando se quiera ordenar un archivo de un número arbitrario de tramos. Además, haciendo el cambio de variable de f_i por $t_1^{L)}$ se tiene la sucesión de los números de *fibonacci* de orden 2:

$$f_{i+1} = f_i + f_{i-1} + f_{i-2}\ \forall i \geq 2\ ,,\ f_2 = 1,\ f_1 = 0,\ f_0 = 0$$

L	$t_1^{L)}$	$t_2^{L)}$	$t_3^{L)}$	
5	13	11	7	→ *(Total 31 tramos, situación inicial)*
4	7	6	4	
3	4	3	2	
2	2	2	1	
1	1	1	1	
0	1	0	0	

Figura 9.4. Tramos que intervienen en cada pasada para 31 tramos.

Definición

La sucesión de *números de fibonacci de orden k* tiene la expresión:

$$f_{i+1} = f_i + f_{i-1} + ... + f_{i-k+1}\ \forall i \geq k-1 \quad \text{tal que,}$$
$$f_{k-1} = 1,\ f_{k-2} = 0\ ...,\ f_0 = 0$$

9.7.3. Distribución inicial de tramos

Por inducción, se obtiene las fórmulas recurrentes que permiten conocer el número de tramos para cada pasada `L`, en el supuesto de utilizar `m` archivos, son las siguientes:

$$t_{m-1}^{L+1)} = t_1^{L)}$$
$$t_{m-2}^{L+1)} = t_1^{L)} + t_{m-1}^{L)}$$
$$\ldots$$
$$t_2^{L+1)} = t_1^{L)} + t_3^{L)}$$
$$t_1^{L+1)} = t_1^{L)} + t_2^{L)}$$
$$\forall\ L > 0 \text{ y } t_1^{0)} = 1\ ,\ t_i^{0)} = 0 \quad \forall i \le m-1$$

Con estas relaciones puede conocerse de antemano el número de tramos necesarios para aplicar el método polifásico con `m` archivos. Es evidente, que no siempre el número de tramos iniciales del archivo a ordenar va a coincidir con ese número ideal, ¿qué hacer ? Sencillamente, simular los tramos necesarios para completar la distribución perfecta con tramos vacíos o tramos ficticios.

¿Cómo tratar los tramos ficticios? La selección de un tramo ficticio de un archivo i consiste en, simplemente, ignorar el archivo y, por consiguiente, desechar dicho archivo de la mezcla del tramo correspondiente. En el supuesto de que el tramo sea ficticio para los `m-1` archivos de entrada, no habrá que hacer ninguna operación, salvo considerar un tramos ficticio en el archivo de salida. Los distribución inicial ha de repartir los tramos ficticios lo más uniformemente posible en los `m-1` archivos.

Para tener en cuenta estas consideraciones relativas a los tramos, se utilizan dos arrays, `a[]` y `d[]`. El primero, `a[]`, contiene los números de tramos que ha de tener cada archivo de entrada en una pasada dada; el segundo, `d[]`, guarda el número de tramos ficticios que tiene cada archivo.

El proceso se inicia de *"abajo a arriba"*; por ejemplo, aplicando la tabla de la Figura 9.4, se empieza asignando al array `a[]` el numero de tramos correspondientes con la última mezcla, siempre `(1,1,... ,1)`. A la vez, al array de tramos ficticios `d[]` también `(1,1,... ,1)`, de tal forma que cada vez que se copie un tramo, del archivo origen, en el archivo `i`, se decrementa `d[i]`, y así con cada uno de los `m-1` archivos.

Si no se ha terminado el archivo original, se determina de nuevo el número de tramos del siguiente nivel y los tramos que hay que añadir a cada archivo para alcanzar ese segundo nivel, según las relaciones recurrentes.

EJEMPLO 9.7. Se desea ordenar un archivo que dispone de 28 tramos, se van a utilizar `m = 4` archivos auxiliares. Encontrar la distribución inicial de tramos en los `m-1` archivos.

La distribución se realiza consecutivamente, de nivel a nivel, según la Figura 9.5. El primer nivel consta de tres tramos, `(1,1,1)`, se distribuyen `1+1+1 = 3` tramos. El segundo nivel consta de `5` tramos, `(2,2,1)`, como ya se distribuyeron `3` tramos se añaden `2` nuevos tramos, `(2,2,1)-(1,1,1) = (1,1,0)`. Los tramos necesarios para alcanzar el tercer nivel son `9 = (4,3,2)`, entonces se debe añadir `(4,3,2)-(2,2,1) = (2,1,1)`, se han distribuido `4` nuevos tramos.

Sucesivamente, se calcula el número de tramos de cada nuevo nivel y los tramos que hay que añadir para obtener esa cantidad, esos tramos serán, inicialmente, los tramos ficticios. A medida que se reparten tramos se decrementa el correspondiente elemento del array de tramos ficticios `d[]`, así hasta alcanzar todos los tramos de los que consta el archivo. Al final en

`a[]` quedan los tramos del último nivel y en `d[]` los tramos que son ficticios. La Figura 9.5 muestra los distintos valores que toma `a[]` y `d[]` en cada nivel, hasta completar los 28 tramos.

En el supuesto planteado, 28 tramos, una vez completado el cuarto nivel se han repartido `17` tramos. Se pasa al siguiente nivel, ahora `a[] = (13,11,7),` los tramos que se incorporan son `(13,11,7)-(7,6,4) = (6,5,3);` los tramos ficticios siempre se inicializan al número de tramos que se añaden, en este nivel `d(6,5,3)`. Restan únicamente `28-17=11` tramos, éstos se distribuyen uniformemente en los archivos, al finalizar la distribución el array de tramos ficticios queda `d[] = (2,1,0)` que se deben tener en cuenta durante la fase de mezcla.

T	ramos iniciales	Tramos, nivel, añadir			
		Número de nivel			
		2	3	4	5
a[]	(1,1,1)	(2,2,1)	(4,3,2)	(7,6,4)	**(13,11,7)**
añadir		(1,1,0)	(2,1,1)	(3,3,2)	(6,5,3)
	(1,1,1)	(1,1,0)	(2,1,1)	(3,3,2)	(6,5,3)
d[]	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	**(2,1,0)**

Figura 9.5. Distribución inicial con 28 tramos y m = 4 archivos.

9.7.4. Mezcla polifásica *versus* mezcla múltiple

Las principales diferencias de la ordenación polifásica respecto a la mezcla equilibrada múltiple:

1. En cada pasada hay un solo archivo destino (salida), en vez de los `m/2` que necesita la mezcla equilibrada múltiple.
2. La finalidad de los archivos (*entrada, salida*) no cambia en cada pasada, sólo uno de ellos pasa de *entrada* a *salida*, según la rotación de los índices de los archivos. La mezcla múltiple intercambia `m/2` archivos de entrada con `m/2` archivos de salida.
3. El número de archivos de *entrada* varía dependiendo del número de tramo en proceso. Éste se determina en el momento de empezar el proceso de mezcla de un tramo, a partir del contador `d[i]` de tramos ficticios para cada archivo `i`. Puede ocurrir que `d[i] > 0` para todos los valores de `i, i = 1..m - 1`, esto significa que hay que mezclar `m-1` tramos ficticios dando lugar a un tramo ficticio en el archivo destino, a la vez se incrementará el elemento `d[] correspondiente al` archivo de salida.
4. Ahora el criterio de terminación de una fase viene dado por el número de tramos a ser mezclados de cada archivo. Puede ocurrir que se alcance el último registro del archivo `m-1` y sea preciso más mezclas que utilicen tramos ficticios de ese archivo. En la fase de distribución inicial fueron calculados los números de *fibonacci* de cada nivel de

forma progresiva, hasta alcanzar el nivel en el que se agotó el archivo de entrada; aplicando fórmulas recurrentes, a partir de los números (de tramos) del último nivel se obtienen los números del nivel anterior:

RESUMEN

La ordenación de archivos se denomina ordenación externa pues los registros no se encuentran en arrays (memoria interna), sino en dispositivos de almacenamiento masivo, como son los cartuchos, *cd*s, *dvd*s, discos duros...; se requiere algoritmos apropiados. Una manera trivial de realizar la ordenación de un archivo secuencial consiste en copiar los registros a otro archivo de acceso directo, o bien secuencial indexado, usando como clave el campo por el que se desea ordenar.

Si se desea realizar la ordenación de archivos, utilizando solamente como estructura de almacenamiento auxiliar otros archivos secuenciales de formato similar al que se desea ordenar, hay que trabajar usando el esquema de *separación* y *mezcla*.

En el caso del algoritmo de *mezcla simple* se opera con tres archivos análogos: el original y dos archivos auxiliares. El proceso consiste en recorrer el archivo original y copiar secuencias de sucesivos registros en, alternativamente, cada uno de los archivos auxiliares. A continuación se mezclan las secuencias de los archivos y se copia la secuencia resultante en el archivo original. El proceso continúa, de tal forma que en cada pasada la longitud de la secuencia es el doble de la longitud de la pasada anterior. Todo empieza con secuencias de longitud `1`, y termina cuando se alcanza una secuencia de longitud igual al número de registros.

El algoritmo de *mezcla natural* también trabaja con tres archivos, se diferencia de la mezcla directa en que distribuye secuencias ordenadas, en vez de secuencias de longitud fija.

Se estudian métodos avanzados de ordenación externa, como la *mezcla equilibrada múltiple* y la *mezcla polifásica*. El primero utiliza un número par de archivos auxiliares, la mitad de ellos son archivos de entrada y la otra mitad archivos de salida. El segundo, mucho más complejo, se caracteriza por realizar la distribución inicial de tramos según las secuencias de números de *fibonacci*; después realiza una *mezcla continuada* hasta obtener un único tramo ordenado.

La primera parte del capítulo revisa la jerarquía de clases para procesar archivos. C++ dispone de un conjunto de clases organizadas jerárquicamente para tratar cualquier tipo flujo de entrada o de salida de datos. Es necesario incluir los archivos de cabecera `<iostream>` y `<fstream>`.

EJERCICIOS

9.1. Escribir las sentencias necesarias para abrir un archivo de caracteres, cuyo nombre y acceso se introduce por teclado, en modo lectura; en el caso de que el resultado de la operación sea erróneo, abrir el archivo en modo escritura.

9.2. Un archivo contiene enteros positivos y negativos. Escribir una función para leer el archivo y determinar el número de enteros negativos.

9.3. Escribir una función para copiar un archivo. La función tendrá dos argumentos de tipo cadena, el primero es el archivo original y el segundo el archivo destino.

9.4. Una aplicación instancia objetos de las clases `NumeroComplejo` y `NumeroRacional`. La primera tiene dos variables instancia de tipo `float`, `parteReal` y `parteImaginaria`. La se-

gunda clase tiene definidas tres variables, *numerador* y *denominador* de tipo `int` y *frac* de tipo `double`. Escribir un programa de tal forma que los objetos se guarden en un archivo y que posteriormente se pueda recuperar.

9.5. El archivo F, almacena registros con un campo clave de tipo entero. La secuencia de claves que se encuentra en el archivo es la siguiente:

```
14  27  33  5  8  11  23  44  22  31  46  7  8  11  1  99  23  40  6
11  14  17
```

Aplicar el algoritmo de mezcla directa, realizar la ordenación del archivo y determinar el número de pasadas necesarias.

9.6. Con el mismo archivo que el del Ejercicio 9.5, aplicar el algoritmo de mezcla natural para ordenar el archivo. Comparar el número de pasadas con las obtenidas en el Ejercicio 9.5.

9.7. Un archivo secuencial F contiene registros y quiere ser ordenado utilizando `4` archivos auxiliares. La ordenación se desea hacer respecto a un campo de tipo entero, con estos valores:

```
22  11  3  4  11 55  2  98  11  21  4  3  8  12  41 21 42  58  26  19
11 59  37  28  61  72  47
```

Aplicar el algoritmo de mezcla equilibrada múltiple y obtener el número de pasadas necesarias para su ordenación.

9.8. Con el mismo archivo que en el Ejercicio 9.7, y también con `m = 4` archivos auxiliares aplicar el algoritmo de mezcla polifásica. Comparar el número de pasadas realizadas con ambos métodos.

PROBLEMAS

9.1. Escribir un programa que compare dos archivos de texto (caracteres). El programa ha de mostrar las diferencias entre el primer archivo y el segundo, precedidas del número de línea y de columna.

9.2. Un atleta utiliza un pulsómetro para sus entrenamientos. El pulsómetro almacena las pulsaciones cada `15` segundos, durante un tiempo máximo de `2` horas. Escribir un programa para almacenar en un archivo los datos del pulsómetro del atleta, de tal forma que el primer registro contenga la fecha, hora y tiempo en minutos de entrenamiento, a continuación los datos del pulsómetro por parejas: tiempo, pulsaciones.

9.3. Las pruebas de acceso a la universidad Uni1, constan de `4` apartados, cada uno de los cuales se puntúa de `1` a `25` puntos. Escribir un programa para almacenar en un archivo los resultados de las pruebas realizadas, de tal forma que se escriban objetos con los siguientes datos: nombre del alumno, puntuaciones de cada apartado y la puntuación total.

9.4. Dado el archivo de puntuaciones generado en el Problema 9.3, escribir un programa para ordenar el archivo utilizando el algoritmo de ordenación externa *mezcla natural*.

9.5. Se dispone del archivo ordenado de puntuaciones de la universidad Uni1 del Problema 9.4, y del archivo de la universidad Uni2 que consta de objetos con los mismos datos y también está ordenado. Escribir un programa que mezcle ordenadamente los dos archivos en un tercero.

9.6. Se tiene guardado en un archivo a los pobladores de la comarca de Pinilla. Los datos de cada persona son los siguientes: primer apellido (campo clave), segundo apellido (campo secundario), edad, años de estancia y estado civil. Escribir un programa para ordenar el archivo por el algoritmo de *mezcla equilibrada múltiple*. Utilizar 6 archivos auxiliares.

9.7. Una farmacia quiere mantener su stock de medicamentos en una archivo. De cada producto interesa guardar el código, precio y descripción. Escribir un programa que genere el archivo pedido, almacenándose los objetos de manera secuencial.

9.8. Escribir un programa para ordenar el archivo que se ha generado en el Problema 9.6. Utilizar el algoritmo de ordenación *mezcla polifásica*, con m = 5 archivos auxiliares.

9.9. Implementar un método de ordenación externa con dos archivos auxiliares. La separación inicial del archivo en tramos sigue la siguiente estrategia: se leen 20 elementos del archivo en un array y se ordenan con el método de ordenación interna *quicksort*. A continuación, se escribe el elemento menor del array en el archivo auxiliar y se lee el siguiente elemento del archivo origen. Si el elemento leído es mayor que el elemento escrito (forma parte del tramo actual), entonces se inserta en orden en el subarray ordenado; en caso contrario se añade en las posiciones libres del array. Debido a que los elementos del array se extraen por la *cabeza* quedan posiciones libres por el extremo contrario. El tramo termina en el momento que el subarray ordenado queda vacío. Para formar el siguiente tramo se empieza ordenando el array (recordar que los elementos que no formaban parte del tramo se iban añadiendo al array) y después el proceso continúa de la misma forma: escribir el elemento menor en otro archivo auxiliar y leer elemento del archivo origen... Una vez realizada la distribución, la fase de mezcla es igual que en los algoritmos de *mezcla directa* o *natural*.

Capítulo 10
Listas

Objetivos

Con el estudio de este capítulo usted podrá:

- Distinguir entre una estructura secuencial y una estructura enlazada.
- Definir el tipo abstracto de datos *Lista*.
- Conocer las operaciones básicas de la listas enlazadas.
- Implementar una lista enlazada para un tipo de elemento.
- Aplicar la estructura Lista para almacenar datos en el desarrollo de aplicaciones.
- Definir una lista doblemente enlazada.
- Definir una lista circular.
- Implementar listas enlazadas ordenadas.
- Realizar una lista genérica.

Contenido

Conceptos clave

- Enlace.
- Estructura enlazada.
- Lista circular.
- Lista doble.
- Lista doblemente enlazada.
- Lista genérica.
- Lista simplemente enlazada.
- Nodo.
- Recorrer una lista
- Tipo abstracto de dato.

INTRODUCCIÓN

En este capítulo se comienza el estudio de las estructuras de datos dinámicas. Al contrario que las estructuras de datos estáticas (*arrays* —listas, vectores y tablas— y *estructuras*) en las que su tamaño en memoria se establece durante la compilación y permanece inalterable durante la ejecución del programa, las estructuras de datos dinámicas crecen y se contraen a medida que se ejecuta el programa.

La estructura de datos que se estudiará en este capítulo es la **lista enlazada** (ligada o encadenada, "*linked list*") que es una colección de elementos (denominados nodos) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un "enlace" o "referencia". En el capítulo se desarrollan algoritmos para insertar, buscar y borrar elementos en las listas enlazadas. De igual modo, se muestra el tipo abstracto de datos (*TAD*) que representa a las listas enlazadas.

10.1. FUNDAMENTOS TEÓRICOS DE LISTAS ENLAZADAS

Las estructuras lineales de elementos homogéneos (listas, tablas, vectores) implementadas con *arrays* necesitan fijar por adelantado el espacio a ocupar en memoria, de modo que cuando se desea añadir un nuevo elemento, que rebase el tamaño prefijado, no será posible sin que se produzca un error en tiempo de ejecución. Esto hace ineficiente el uso de los arrays en algunas aplicaciones.

Gracias a la asignación dinámica de memoria, se pueden implementar listas de modo que la memoria física utilizada se corresponda exactamente con el número de elementos de la tabla. Para ello se recurre a los punteros (*apuntadores*) y *variables apuntadas* que se crean en tiempo de ejecución.

Una **lista enlazada** es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un "enlace". La idea básica consiste en construir una lista cuyos elementos, llamados **nodos**, se componen de dos partes (*campos)*: la primera parte contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado *Dato, TipoElemento*, *Info*, etc.), y la segunda parte es un *enlace* que apunta al siguiente nodo de la lista.

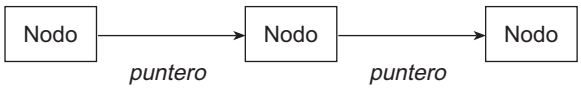
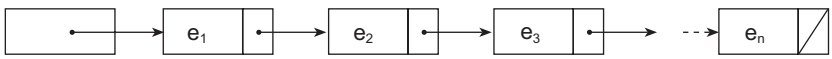


Figura 10.1. Lista enlazada (representación simple).

La representación gráfica más extendida es aquella que utiliza una caja con dos secciones en su interior. En la primera sección se encuentra el elemento o valor del dato y en la segunda sección el enlace, representado mediante una flecha que sale de la caja y apunta al siguiente nodo.



e_1, e_2, ...e_n son valores del tipo `TipoElemento`

Figura 10.2. Lista enlazada (representación gráfica típica).

Definición

Una **lista enlazada** consta de un número de nodos con dos componentes (*campos*), un enlace al siguiente nodo de la lista y un v alor, que puede ser de cualquier tipo

Los enlaces se representan por flechas para facilitar la comprensión de la conexión entre dos nodos; ello indica que el enlace tiene la dirección en memoria del siguiente nodo. Los enlaces también sitúan los nodos en una secuencia. La Figura 10.2 muestra una lista cuyos nodos forman una secuencia desde el primer elemento (e_1) al último elemento (e_n). El último nodo ha de representarse de forma diferente, para significar que este nodo no se enlaza a ningún otro. La Figura 10.3 muestra diferentes representaciones gráficas que se utilizan para dibujar el campo enlace del último nodo.

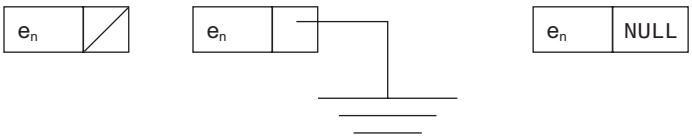


Figura 10.3. Diferentes representaciones gráficas del nodo último.

10.1.1. Clasificación de las listas enlazadas

Las listas se pueden dividir en cuatro categorías:

- *Listas simplemente enlazadas*. Cada nodo (elemento) contiene un único enlace que conecta ese nodo al nodo siguiente o nodo sucesor. La lista es eficiente en recorridos directos ("adelante").
- *Listas doblemente enlazadas*. Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo ("adelante") como en recorrido inverso ("atrás").
- *Lista circular simplemente enlazada*. Una lista simplemente enlazada en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular ("en anillo").
- *Lista circular doblemente enlazada*. Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular (en anillo) tanto en dirección directa ("adelante") como inversa ("atrás").

La implementación de cada uno de los cuatro tipos de estructuras de listas se puede desarrollar utilizando punteros.

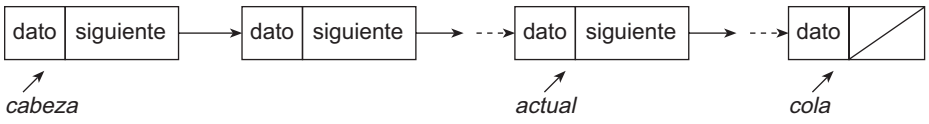


Figura 10.4. Representación gráfica de una lista enlazada.

El primer nodo, **frente**, de una lista es el nodo apuntado por **cabeza**. La lista encadena nodos juntos desde el frente al final (**cola**) de la lista. El final se identifica como el nodo cuyo campo enlace tiene el valor `NULL`. La lista se recorre desde el primero al último nodo; en cualquier punto del recorrido la posición *actual* se referencia por el puntero `actual`. Una lista vacía, es decir, que no contiene nodos se representa con el puntero `cabeza` a nulo.

10.2. TIPO ABSTRACTO DE DATOS *Lista*

Una lista almacena información del mismo tipo, con la característica de que puede contener un número indeterminado de elementos, y que estos elementos mantienen un orden explícito. Este ordenamiento explícito se manifiesta en que, en sí mismo, cada elemento contiene la dirección del siguiente elemento. Cada elemento es un nodo de la lista.

Una lista es una estructura de datos dinámica. El número de nodos puede variar rápidamente en un proceso. Aumentando los nodos por inserciones, o bien, disminuyendo por eliminación de nodos.

Las inserciones se pueden realizar por cualquier punto de la lista. Por la cabeza (inicio), por el final (cola), a partir o antes de un nodo determinado de la lista. Las eliminaciones también se pueden realizar en cualquier punto de la lista; además se eliminan nodos dependiendo del campo de información o dato que se desea suprimir de la lista.

10.2.1. Especificación formal del *TAD Lista*

Matemáticamente, una lista es una secuencia de cero o más elementos de un determinado tipo.

`(a1, a2, a3, ... , an)` donde `n >= 0`, *si* `n = 0` *la lista es vacía*.

Los elementos de la lista tienen la propiedad de que sus elementos están ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que a_i precede a a_{i+1} para `i = 1 ..., n-1`; y que a_i sucede a a_{i-1} para `i = 2 ... n`.

Para formalizar el *Tipo Abstracto de Dato Lista* a partir de la noción matemática, se define un conjunto de operaciones básicas con objetos de tipo `Lista`. Las operaciones:

`∀ L ∈ Lista,` `∀ x ∈ Dato,` `∀ p ∈ puntero`

`Listavacia(L)`	Inicializa la lista `L` como lista vacía.
`Esvacia(L)`	Determina si la lista `L` está vacía.
`Insertar(L,x,p)`	Inserta en la lista `L` un nodo con el campo dato `x`, delante del nodo de dirección `p`.
`Localizar(L,x)`	Devuelve la posición/dirección donde está el campo de información `x`.
`Suprimir(L,x)`	Elimina de la lista el nodo que contiene el dato `x`.
`Anterior(L,p)`	Devuelve la posición/dirección del nodo anterior a `p`.
`Primero(L)`	Retorna la posición/dirección del primer nodo de la lista `L`.
`Anula(L)`	Vacía la lista `L`.

Estas operaciones son las básicas para manejar listas. En realidad, la decisión de qué operaciones son las básicas depende de las características de la aplicación que se va a realizar con los datos de la lista. También dependerá del tipo de representación elegido para las listas. Así, para añadir nuevos nodos a una lista se implementan, además de `insertar()`, versiones de ésta como:

`inserPrimero(L,x)`	Inserta un nodo con el dato `x` como primer nodo de la lista `L`.
`inserFinal(L,x)`	Inserta un nodo con el dato `x` como último nodo de la lista `L`.

Otra operación típica de toda estructura enlazada de datos es *recorrer*. Consiste en visitar cada uno de los datos o nodos de que consta. En las listas enlazadas, normalmente se realiza desde el nodo *cabeza* al último nodo o *cola* de la lista.

10.3. OPERACIONES DE LISTAS ENLAZADAS

La implementación del *TAD* `Lista` requiere, en primer lugar, declarar la clase `Nodo` en la cual se *encierra* el dato (entero, real, doble, carácter, o referencias a objetos) y el enlace. Además, la clase `Lista` con las operaciones (*creación, inserción,* ...) y el atributo *cabeza* de la lista.

10.3.1. Clase *Nodo*

Una lista enlazada se compone de una serie de nodos enlazados mediante punteros. La clase `Nodo` declara las dos partes en que se divide: `dato` y `enlace`. Además, el constructor y la interfaz básica; por ejemplo, para el caso de una lista enlazada de números enteros:

```
typedef int Dato;
        // archivo de cabecera Nodo.h
#ifndef _NODO_H
#define _NODO_H
class Nodo
{
protected:
    Dato dato;
    Nodo* enlace;
public:
  Nodo(Dato t)
  {
    dato = t;
    enlace = 0;                 // 0 es el puntero NULL en C++
  }
  Nodo(Dato p, Nodo* n)         // crea el nodo y lo enlaza a n
  {
    dato = p;
    enlace = n;
  }

  Dato datoNodo() const
```

```
  {
    return dato;
  }

  Nodo* enlaceNodo() const
  {
    return enlace;
  }

  void ponerEnlace(Nodo* sgte)
  {
    enlace = sgte;
  }
};
#endif
```

Dado que los datos que se puede incluir en una lista pueden ser de cualquier tipo (entero, real, caracter o cualquier objeto), puede declararse un nodo genérico y, en consecuencia, una lista genérica con las plantillas de C++:

```
template <class T> class NodoGenerico
{
protected:
    T dato;
    NodoGenerico <T>* enlace;

public:
    NodoGenerico (T t)
    {
      dato = t;
      enlace = 0;
    }
    NodoGenerico (T p, NodoGenerico<T>* n)
    {
      dato = p;
      enlace = n;
    }

    T datoNodo() const
    {
       return dato;
    }

    NodoGenerico<T>* enlaceNodo() const
    {
       return enlace;
    }

    void ponerEnlace(NodoGenerico<T>* sgte)
    {
      enlace = sgte;
    }
};
```

EJEMPLO 10.1. Se declara la clase `Punto` para representar un punto en el plano , con su coordenada x e y. También, se declara la clase `Nodo` con el campo dato de tipo `Punto`.

```
// archivo de cabecera punto.h
class Punto
{
  double x, y;

  public:
    Punto(double x = 0.0, double y = 0.0)
    {
      this → x = x;
      this → y = y;
    }
};
```

La declaración de la clase `Nodo` consiste, sencillamente, en asociar el nuevo tipo de dato, el resto no cambia.

```
typedef Punto Dato;
#include "Nodo.h"
```

10.3.2. Acceso a la lista: *cabecera* y *cola*

Cuando se construye y utiliza una lista enlazada en una aplicación, el acceso a la lista se hace mediante uno, o más, *punteros* a los nodos. Normalmente, se accede a partir del primer nodo de la lista, llamado **cabeza** o **cabecera** de la lista. En ocasiones, se mantiene también un apuntador al último nodo de la lista enlazada, llamado **cola** de la lista.

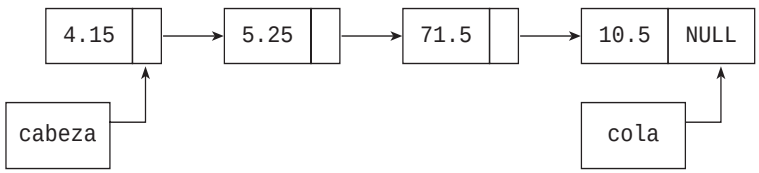


Figura 10.5. Declaraciones de tipos en lista enlazada.

Los apuntadores, `cabeza` y `cola`, se declararan como variables puntero a `Nodo`:

```
Nodo* cabeza;
Nodo* cola;
```

La Figura 10.5 muestra una lista a la que se accede con el puntero `cabeza`; cada nodo está enlazado con el siguiente nodo. El último nodo, cola o final de la lista, no se enlaza con otro nodo, entonces su campo de enlace contiene *nulo* (*0* o `NULL` indistintamente). Normalmente `NULL` se utiliza en dos situaciones:

- Campo *enlace* del último nodo (*final o cola*) de una lista enlazada.
- Como valor `cabeza`, para una lista enlazada que no tiene nodos, es decir una **lista vacía** (`cabeza = NULL`).

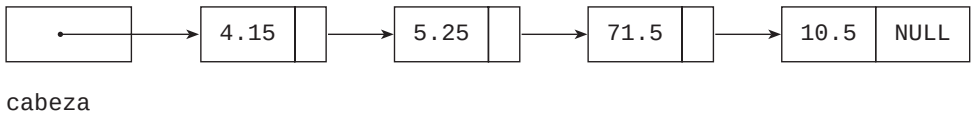


Figura 10.6. Acceso a una lista con puntero cabeza.

Nota de programación

Las variables de acceso a una lista, `cabeza` y `cola`, se inicializan a `NULL` cuando comienza la construcción de la lista.

Error de programación

Uno de los errores típicos en el tr atamiento de punteros consiste en escr ibir la expresión `p → miembro` cuando el valor del puntero `p` es `NULL`.

10.3.3. Clase `Lista`: construcción de una lista

La creación de una lista enlazada implica la definición de, al menos, la clases `Nodo` y `Lista`. La clase `Lista` contiene el puntero de acceso a la lista enlazada, de nombre primero, que apunta al nodo cabeza; también se puede declarar un puntero al nodo cola, como no se necesita para implementar las operaciones no se ha declarado.

Las funciones de la clase `Lista` implementan las operaciones de una lista enlazada: *inserción, búsqueda* .... El constructor inicializa `primero` a `NULL`, (*lista vacía*). Además, `crearLista()` construye iterativamente el primer elemento (`primero`) y los elementos sucesivos de una lista enlazada.

El Ejemplo 10.2 declara una lista para un tipo particular de dato: `int`. Lo más interesante del ejemplo es la codificación, paso a paso, de la función `crearLista()`.

EJEMPLO 10.2. Crear una lista enlazada de elementos que almacenen datos de tipo entero.

La declaración de la clase `Nodo` se encuentra en el archivo de cabecera `Nodo.h`, (*consultar* Apartado 10.3.1).

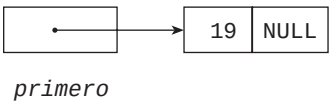
```
typedef int Dato;
#include "Nodo.h"
```

```
class Lista
{
protected:
    Nodo* primero;
  public:
    Lista()
     {
       primero = NULL;
    }
       void crearLista();
    //...
```

La referencia `primero` (también se puede llamar `cabeza`) se ha inicializado en el constructor a un valor *nulo*, es decir, a *lista vacía*.

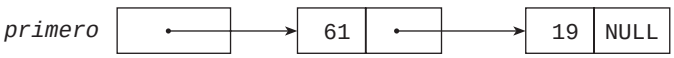
A continuación, y antes de escribir su código, se muestra el comportamiento de la función `crearLista()`. En primer lugar, se crea un nodo con un valor y su dirección se asigna a `primero`:

```
primero = new Nodo(19);
```



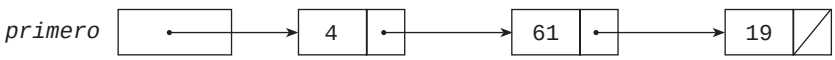
Ahora se desea añadir un nuevo elemento con el valor `61`, y situarlo en el primer lugar de la lista. Se utiliza el constructor de `Nodo` que enlaza con otro nodo ya creado:

```
primero = new Nodo(61,primero);
```



Por último, para obtener una lista compuesta de `4, 61, 19` se habría de ejecutar:

```
primero = new Nodo(4,primero);
```



A continuación, se escribe `CrearLista()` que codifica las acciones descritas anteriormente. Los valores se leen del teclado, termina con el valor clave `-1`.

```
void Lista::crearLista()
{
  int x;
  primero = 0;
  cout << "Termina con -1" << endl;
  do {
    cin >> x;
    if (x != -1)
```

```
        {
          primero = new Nodo(x,primero);
        }
      }while (x != -1);
   }
```

10.4. INSERCIÓN EN UNA LISTA

El nuevo elemento que se desea incorporar a una lista se puede insertar de distintas formas, según la posición o punto de inserción. Éste puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final o cola de la lista (elemento último).
- Antes de un elemento especificado, o bien.
- Después de un elemento especificado.

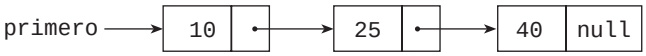
10.4.1. Insertar en la cabeza de la lista

La posición más facil y, a la vez, más eficiente donde insertar un nuevo elemento en una lista es por la *cabeza*. El proceso de inserción se resume en este algoritmo:

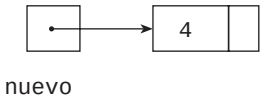
1. Crear un nodo e inicializar el campo dato al nuevo elemento. La dirección del nodo creado se asigna a `nuevo`.
2. Hacer que el campo `enlace` del nodo creado apunte a la *cabeza* (`primero`) de la lista.
3. Hacer que `primero` apunte al nodo que se ha creado.

El Ejemplo 10.3 inserta un elemento por la *cabeza* de una lista siguiendo los pasos del algoritmo y se escribe el código.

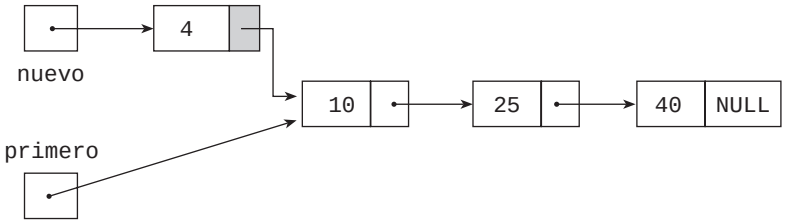
EJEMPLO 10.3. Una lista enlazada contiene tres elementos , 10, 25 y 40. Insertar un nuevo elemento, 4, en cabeza de la lista.



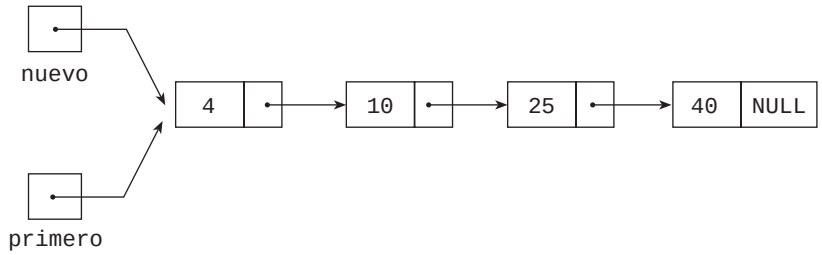
Paso 1



Paso 2



Paso 3



En este momento, la función termina su ejecución, la variable local `nuevo` desaparece y sólo permanece el puntero `primero` al inicio de la lista.

El código fuente de `insertarCabezaLista`:

```
void Lista::insertarCabezaLista(Dato entrada)
{
  Nodo* nuevo ;
  nuevo = new Nodo(entrada);
  nuevo -> ponerEnlace(primero);       // enlaza nuevo con primero
  primero = nuevo;     // mueve primero y apunta al nuevo nodo
}
```

Caso particular

`insertarCabezaLista` también actúa correctamente si se trata de añadir un primer nodo a una lista vacía. En este caso, `primero` apunta a `NULL` y termina apuntando al nuevo nodo de la lista enlazada.

EJERCICIO 10.1. Programa para crear una lista de números aleatorios. Inserta los nuevos nodos por la cabeza de la lista. Una vez creada la lista, se ha de recorrer para mostrar los números pares.

Se crea una lista enlazada de números enteros, para ello se utilizan las clases `Nodo` y `Lista` según las declaraciones de los Apartado 10.3 y 10.4. En esta última se añade la función `visualizar()` que recorre la lista escribiendo el campo dato de cada nodo. Desde `main()` se

crea un objeto `Lista`, se llama iterativamente a la función miembro `insertarCabezaLista`, y, por último, se llama a `visualizar()` para mostrar los elementos.

```
// archivo de cabecera Lista.h e implementación de visualizar()

#include <iostream >
using namespace std;
typedef int Dato;
#include "Nodo.h"

class Lista
{
protected:
    Nodo* primero;
public:
    Lista();
    void crearLista();
    void insertarCabezaLista(Dato entrada);
    void visualizar();
};

void Lista::visualizar()
{
  Nodo* n;
  int k = 0;
  n = primero;
  while (n != NULL)
  {
    char c;
    k++; c = (k%15 != 0 ? ‘ ‘ : ‘\n’);
    cout << n -> datoNodo() << c;
    n = n -> enlaceNodo();
  }
}

// archivo con la función main
#include <iostream >
using namespace std;
typedef int Dato;
#include "Nodo.h"
#include "Lista.h"

int main()
{
  Dato d;
  Lista lista;     // crea lista vacía
  cout << "Elementos de la lista, termina con -1 " << endl;
  do {
    cin >> d;
    lista.insertarCabezaLista(d);
  } while (d != -1);
  // recorre la lista para escribir sus elementos
  cout << "\t Elementos de la lista generados al azar" << endl;
  lista.visualizar();
  return 0;
}
```

10.4.2. Inserción al final de la lista

La inserción al final de la lista es menos eficiente debido a que, normalmente, no se tiene un puntero al último nodo y entonces se ha de seguir la traza desde la cabeza de la lista hasta el último nodo y, a continuación, realizar la inserción. Una vez que la variable `ultimo` apunta al final de la lista, el enlace con el nuevo nodo es sencillo:

```
ultimo -> ponerEnlace(new Nodo(entrada));
```

El campo `enlace` del último nodo queda apuntando al nodo creado y así se enlaza, como nodo final, a la lista.

A continuación, se codifica la función *public* `insertarUltimo()` junto a la función que recorre la lista y devuelve el puntero al último nodo.

```
void Lista::insertarUltimo(Dato entrada)
{
  Nodo* ultimo = this -> ultimo();
  ultimo -> ponerEnlace(new Nodo(entrada));
}

Nodo* Lista::ultimo()
{
  Nodo* p = primero;
  if (p == NULL ) throw "Error, lista vacía";
  while (p -> enlaceNodo()!= NULL) p = p -> enlaceNodo();
  return p;
}
```

10.4.3. Insertar entre dos nodos de la lista

La inserción de nodo no siempre se realiza al principio o al final de la lista, puede hacerse entre dos nodos cualesquiera. Por ejemplo, en la lista de la Figura 10.7 se quiere insertar el elemento `75` entre los nodos con los datos `25` y `40`.

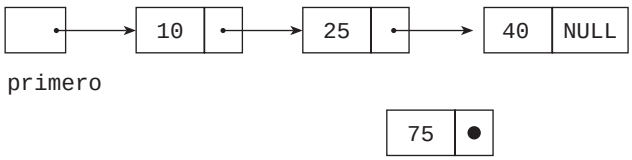


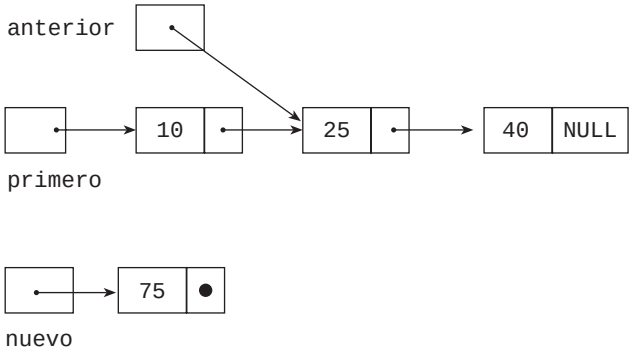
Figura 10.7. Inserción entre dos nodos.

El algoritmo para la operación de insertar entre dos nodos (`n1, n2`) requiere las siguientes etapas:

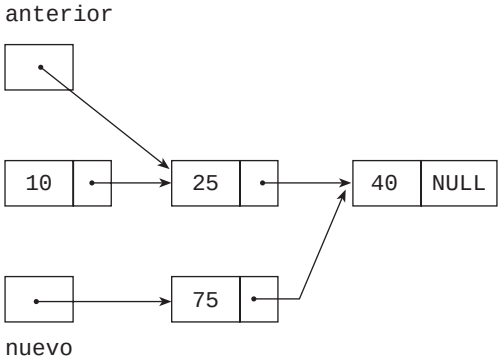
1. Crear un nodo con el elemento y el campo `enlace` a `NULL`.
2. Poner campo `enlace` del nuevo nodo apuntando a `n2`, ya que el nodo creado se ubicará justo antes de `n2`.
3. Si el puntero `anterior` tiene la dirección del nodo `n1`, entonces poner su atributo `enlace` apuntando al nodo creado.

A continuación se muestra gráficamente las etapas del algoritmo relativas a la inserción de `75` entre `25` (`n1`) y `40` (`n2`).

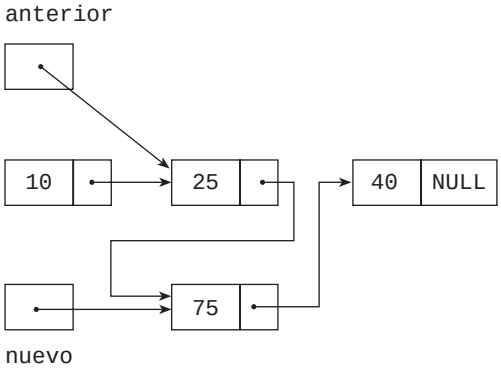
Etapas 1



Etapa 2



Etapa 3



La operación es una función miembro de la clase *Lista*:

```
void Lista::insertarLista(Nodo* anterior, Dato entrada)
{
  Nodo* nuevo;

  nuevo = new Nodo(entrada);
  nuevo -> ponerEnlace(anterior -> enlaceNodo());
  anterior -> ponerEnlace(nuevo);
}
```

Antes de llamar a `insertarLista()`, es necesario buscar la dirección del nodo `n1`, esto es, del nodo a partir del cual se enlazará el nodo que se va a crear.

Otra versión de la función tiene como argumentos el dato a partir del cual se realiza el enlace, y el dato del nuevo nodo: `insertarLista(Dato testigo, Dato entrada)`. El algoritmo de este versión, primero busca el nodo con el dato *testigo* a partir del cual se inserta, y, a continuación, realiza los mismos enlaces que en la anterior función.

10. 5. BÚSQUEDA EN LISTAS ENLAZADAS

La operación *búsqueda* de un elemento en una lista enlazada recorre la lista hasta encontrar el nodo con el elemento. El algoritmo que se utiliza, una vez encontrado el nodo, devuelve el puntero al nodo (en caso negativo, devuelva `NULL`). Otro planteamiento consiste en devolver `true` si encuentra el nodo y `false` si no está en la lista

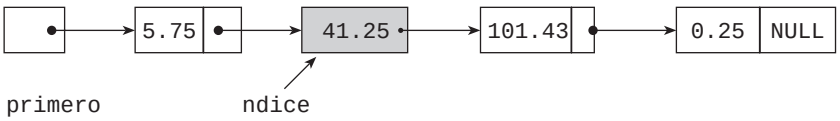


Figura 10.8. Búsqueda en una lista.

La función `buscarLista` de la clase `Lista` utiliza el puntero `indice` para recorrer la lista, nodo a nodo. Primero, se inicializa `indice` al nodo *cabeza* (`primero`), a continuación se compara el dato del nodo apuntado por `indice` con el elemento buscado, si coincide la búsqueda termina, en caso contrario `indice` avanza al siguiente nodo. La búsqueda termina cuando se encuentra el nodo, o bien cuando se ha terminado de recorrer la lista y entonces `indice` toma el valor `NULL`.

```
Nodo* Lista::buscarLista(Dato destino)
{
  Nodo* indice;

  for (indice = primero; indice!= NULL; indice = indice->enlaceNodo())
    if (destino == indice -> datoNodo())
      return indice;
  return NULL;
}
```

EJEMPLO 10.4. Se escribe una función alternativa a la búsqueda del nodo que contiene un campo dato. Ahora también se devuelve un puntero a nodo, pero con el criterio de que ocupa una posición, pasada como argumento, en una lista enlazada.

La función es un miembro *público* de la clase `Lista`, por consiguiente, tiene acceso a sus miembros y dado que se busca por posición en la lista, se considera posición *1* la del nodo cabeza (`primero`); posición *2* al siguiente nodo, y así sucesivamente.

El algoritmo de búsqueda comienza inicializando `indice` al nodo cabeza de la lista. En cada iteración del bucle se mueve `indice` un nodo hacia adelante. El bucle termina cuando se alcanza la posición deseada, o bien si `indice` apunta a `NULL` como consecuencia de que la posición solicitada es mayor que el número de nodos de la lista.

```
Nodo* Lista::buscarPosicion(int posicion)
{
  Nodo* indice;
  int i;

  if (0 >= posicion)             // posición ha de ser mayor que 0
    return NULL;
  indice = primero;
  for (i = 1;(i < posicion) && (indice != NULL); i++)
    indice = indice -> enlaceNodo();
  return indice;
}
```

10.6. BORRADO DE UN NODO

Eliminar un nodo de una lista enlazada supone enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y liberar la memoria que ocupa. El algoritmo se enfoca para eliminar un nodo que contiene un dato, sigue estos pasos:

1. Búsqueda del nodo que contiene el dato. Se ha de obtener la dirección del nodo a eliminar y la dirección del anterior.
2. El `enlace` del nodo anterior que apunte al nodo siguiente al que se elimina.
3. Si el nodo a eliminar es el *cabeza* de la lista (`primero`), se modifica `primero` para que tenga la dirección del siguiente nodo.
4. Por último, la memoria ocupada por el nodo se libera.

Naturalmente, `eliminar()` es una función miembro y *pública* de la clase `Lista`, recibe el dato del nodo que se quiere borrar. Desarrolla su propio bucle de búsqueda con el fin de disponer de la dirección del nodo anterior.

```
void Lista::eliminar(Dato entrada)
{
  Nodo *actual, *anterior;
  bool encontrado;

  actual = primero;
  anterior = NULL;
```

```
  encontrado = false;
                              // búsqueda del nodo y del anterior
  while ((actual != NULL) && !encontrado)
  {
    encontrado = (actual -> datoNodo() == entrada);
    if (!encontrado)
    {
      anterior = actual;
      actual = actual -> enlaceNodo();
    }
  }
                               // enlace del nodo anterior con el siguiente
  if (actual != NULL)
  {
                            // distingue entre cabecera y resto de la lista
    if (actual == primero)
    {
      primero = actual -> enlaceNodo();
    }
    else
    {
      anterior -> ponerEnlace(actual -> enlaceNodo());
    }
    delete actual;
  }
}
```

10.7. LISTA ORDENADA

Los elementos de una lista tienen la propiedad de estar ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que n_i precede a n_{i+1} para `i = 1 ... n-1`; y que n_i sucede a n_{i-1} para `i = 2 ...` n. Ahora bien, tambien es posible mantener una lista enlazada ordenada según el dato asociado a cada nodo. La Figura 10.9 muestra una lista enlazada de números reales, ordenada de forma creciente.

La forma de insertar un elemento en una lista ordenada siempre realiza la operación de tal forma que la lista resultante mantiene la propiedad de ordenación. Para lo cual, en primer lugar, determina la posición de inserción y, a continuación, ajusta los enlaces.

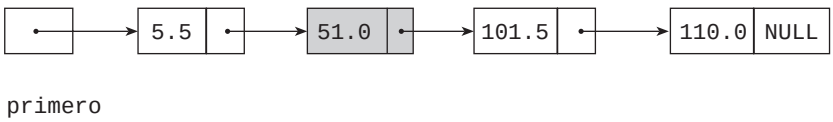


Figura 10.9. Lista ordenada.

Por ejemplo, para insertar el dato `104` en la lista de la Figura 10.9 es necesario recorrer la lista hasta el nodo con `110.0`, que es el nodo inmediatamente mayor. El puntero `indice` se queda con la dirección del nodo anterior, a partir del cual se realizan los enlaces con el nuevo nodo.

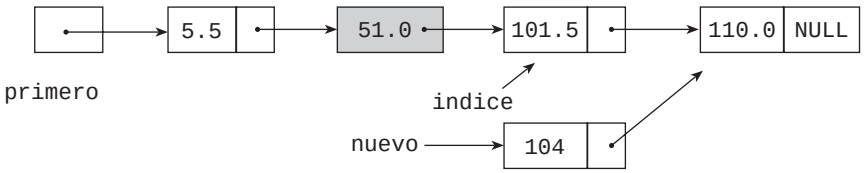


Figura 10.10. Inserción en una lista ordenada.

10.7.1. Clase `ListaOrdenada`

Una lista enlazada ordenada *es una* lista enlazada a la que se añade la propiedad ordenación de sus datos. Ésa es la razón que aconseja declarar la clase `ListaEnlazada` *derivada* de la clase `Lista`, por consiguiente, heredará las propiedades de `Lista`. Las funciones `eliminar()` y `buscarLista()` se deben redefinir para que los bucles de búsqueda aprovechen el hecho de que los datos están ordenados.

La función `insertaOrden()` crea la lista ordenada; el punto de partida es una lista vacía, a la que se añaden nuevos elementos, de tal forma que en todo momento los elementos están ordenados en orden creciente. La inserción del primer nodo de la lista consiste, sencillamente, en crear el nodo y asignar su dirección a la cabeza de la lista. El segundo elemento se ha de insertar antes o después del primero, dependiendo de que sea menor o mayor. El tipo de los datos de una lista ordenada han de ser ordinal, para que se pueda aplicar los operadores `==`, `<`, `>`. A continuación, se escribe el código que implementa la función.

```
void ListaOrdenada::insertaOrden(Dato entrada)
{
  Nodo* nuevo ;
  nuevo = new Nodo(entrada);
  if (primero == NULL)       // lista vacía
    primero = nuevo;
  else if (entrada < primero -> datoNodo())
  {
      nuevo -> ponerEnlace(primero);
      primero = nuevo;
  }
  else          // búsqueda del nodo anterior al de inserción
  {
    Nodo *anterior, *p;
    anterior = p = primero;

    while ((p->enlaceNodo()!= NULL) && (entrada > p->datoNodo()))
    {
        anterior = p;
        p = p -> enlaceNodo();
    }

    if (entrada > p->datoNodo()) // se inserta después del último
    anterior = p;
    // se procede al enlace del nuevo nodo
    nuevo -> ponerEnlace(anterior-> enlaceNodo());
```

```
      anterior -> ponerEnlace(nuevo);
    }
  }
```

10.8. LISTA DOBLEMENTE ENLAZADA

Hasta ahora el recorrido de una lista se ha realizado en sentido directo (*adelante*). Existen numerosas aplicaciones en las que es conveniente poder acceder a los nodos de una lista en cualquier orden, tanto hacia adelante como hacia atrás. Desde un nodo de una **lista doblemente enlazada** se puede avanzar al siguiente, o bien retroceder al nodo anterior. Cada nodo de una lista doble tiene tres campos, el dato y dos punteros, uno apunta al siguiente nodo de la lista y el otro al nodo anterior. La Figura 10.11 muestra una lista doblemente enlazada y un nodo de dicha lista.

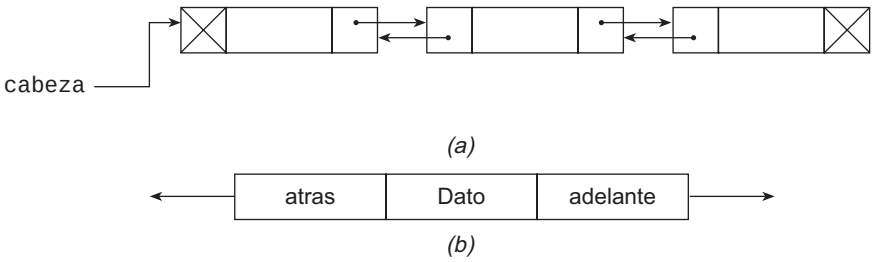


Figura 10.11. Lista doblemente enlazada. *(a)* Lista con tres nodos; *(b)* nodo.

Las operaciones de una *Lista Doble* son similares a las de una *Lista: insertar, eliminar, buscar, recorrer...* La operación de insertar un nuevo nodo en la lista debe realizar ajustes de los dos punteros. La Figura 10.12 muestra los movimientos de punteros para insertar un nodo, como se observa intervienen cuatro enlaces.

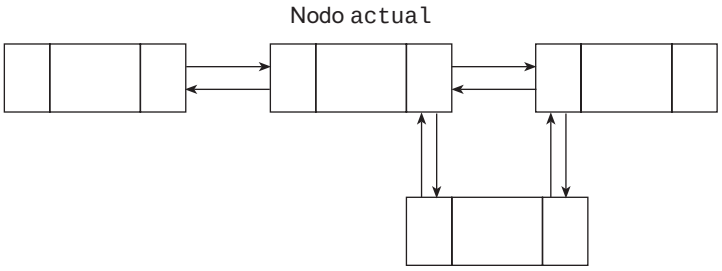


Figura 10.12. Inserción de un nodo en una lista dob lemente enlazada.

La operación de eliminar un nodo de la lista doble necesita enlazar, mutuamente, el nodo anterior y el nodo siguiente del que se borra, como se observa en la Figura 10.13.

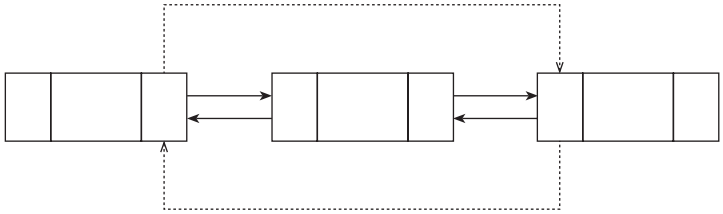


Figura 10.13. Eliminación de un nodo en una lista doblemente enlazada.

10.8.1. Nodo de una lista doblemente enlazada

La clase `NodoDoble` agrupa los componentes del nodo de una lista doble y las operaciones de la *interfaz*.

```
// archivo de cabecera NodoDoble.h
class NodoDoble
{
protected:
    Dato dato;
    NodoDoble* adelante;
    NodoDoble* atras;

public:
    NodoDoble(Dato t)
    {
      dato = t;
      adelante = atras = NULL;
    }

    Dato datoNodo() const { return dato; }

    NodoDoble* adelanteNodo() const { return adelante; }
    NodoDoble* atrasNodo() const { return atras; }

    void ponerAdelante(NodoDoble* a) { adelante = a; }
    void ponerAtras(NodoDoble* a) { atras = a; }
};
```

10.8.2. Insertar un nodo en una lista doblemente enlazada

La clase `ListaDoble` encapsula las operaciones básicas de las listas doblemente enlazadas. La clase dispone del puntero variable `cabeza` para acceder a la lista, apunta al primer nodo. El constructor de la clase inicializa la lista vacía.

Se puede añadir nodos a la lista de distintas formas, según la posición donde se inserte. La posición de inserción puede ser:

- En *cabeza* de la lista.
- Al *final* de la lista.

- Antes de un elemento especificado.
- Después de un elemento especificado.

Insertar por la cabeza

El proceso sigue estos pasos:

1. Crear un nodo con el nuevo elemento.
2. Hacer que el campo `adelante` del nuevo nodo apunte a la *cabeza* (primer nodo) de la lista original, y que el campo `atras` del nodo *cabeza* apunte al nuevo nodo.
3. Hacer que `cabeza` apunte al nodo creado.

A continuación, se escribe la función miembro de la clase `ListaDoble`, que implementa la operación.

```
void ListaDoble::insertarCabezaLista(Dato entrada)
{
  NodoDoble* nuevo;

  nuevo = new NodoDoble (entrada);
  nuevo -> ponerAdelante(cabeza);

  if (cabeza != NULL )
    cabeza -> ponerAtras(nuevo);
  cabeza = nuevo;
}
```

Insertar después de un nodo

El algoritmo de la operación que inserta un nodo después de otro, `n`, requiere las siguientes etapas:

1. Crear un nodo, `nuevo`, con el elemento.
2. Poner el enlace `adelante` del nodo creado apuntando al nodo siguiente de `n`. El enlace `atras` del nodo siguiente a `n` (si `n` no es el último nodo) tiene que apuntar a `nuevo`.
3. Hacer que el enlace `adelante` del nodo `n` apunte al nuevo nodo. A su vez, el enlace `atras` del nuevo nodo debe de apuntar a `n`.

La función `insertaDespues()` implementa el algoritmo, naturalmente es miembro de la clase `ListaDoble`. El primer argumento, `anterior`, representa un puntero al nodo `n` a partir del cual se enlaza el nuevo. El segundo argumento, `entrada`, es el dato que se añade a la lista.

```
void ListaDoble::insertaDespues(NodoDoble* anterior, Dato entrada)
{
  NodoDoble* nuevo;

  nuevo = new NodoDoble(entrada);
  nuevo -> ponerAdelante(anterior -> adelanteNodo());
  if (anterior -> adelanteNodo() != NULL)
    anterior -> adelanteNodo() -> ponerAtras(nuevo);
  anterior-> ponerAdelante(nuevo);
  nuevo -> ponerAtras(anterior);
}
```

10.8.3. Eliminar un nodo de una lista doblemente enlazada

Quitar un nodo de una lista doble supone ajustar los enlaces de dos nodos, el nodo *anterior* con el nodo *siguiente* al que se desea eliminar. El puntero `adelante` del nodo anterior debe apuntar al nodo *siguiente*, y el puntero `atras` del nodo *siguiente* debe apuntar al nodo *anterior*.

El algoritmo es similar al del borrado para una lista simple, más simple, ya que ahora la dirección del nodo *anterior* se encuentra en el campo `atras` del nodo a borrar. Los pasos a seguir:

1. Búsqueda del nodo que contiene el dato.
2. El puntero `adelante` del nodo anterior tiene que apuntar al puntero `adelante` del nodo a eliminar (si no es el nodo *cabecera*).
3. El puntero `atras` del nodo siguiente a borrar tiene que apuntar a donde apunta el puntero `atras` del nodo a eliminar (si no es el último nodo).
4. Si el nodo que se elimina es el primero, se modifica `cabeza` para que tenga la dirección del nodo `siguiente`.
5. La memoria ocupada por el nodo es liberada.

La implementación del algoritmo es una función miembro de la clase `ListaDoble`.

```
void ListaDoble::eliminar (Dato entrada)
{
  NodoDoble* actual;
  bool encontrado = false;

  actual = cabeza;
  // Bucle de búsqueda
  while ((actual != NULL) && (!encontrado))
  {
    encontrado = (actual -> datoNodo() == entrada);
    if (!encontrado)
      actual = actual -> adelanteNodo();
  }
  // Enlace de nodo anterior con el siguiente
  if (actual != NULL)
  {
    //distingue entre nodo cabecera o resto de la lista
    if (actual == cabeza)
    {
      cabeza = actual -> adelanteNodo();
      if (actual -> adelanteNodo() != NULL)
        actual -> adelanteNodo() -> ponerAtras(NULL);
    }
    else if (actual -> adelanteNodo() != NULL) // No es el último
    {
      actual->atrasNodo()->ponerAdelante(actual->adelanteNodo());
      actual->adelanteNodo()->ponerAtras(actual->atrasNodo());
    }
    else      // último nodo
      actual->atrasNodo()->ponerAdelante(NULL);
  }
}
```

EJERCICIO 10.2. Se crea una lista doblemente enlazada con números enteros, del 1 al 999 generados aleatoriamente. Una vez creada la lista se eliminan los nodos que estén fuera de un rango de valores leídos desde el teclado.

En el Apartado 10.8 se han declarado las clases `NodoDoble` y `ListaDoble` necesarias para realizar este ejercicio. Se añade la función `visualizar()` para recorrer la lista doble y mostrar por pantalla los datos de los nodos. Además, se declara la clase `IteradorLista`, para *visitar* cada nodo de cualquier lista doble (de enteros); en esta ocasión se utiliza el mecanismo friend para que `IteradorLista` pueda acceder a los miembros protegidos de `ListaDoble`. El constructor de `IteradorLista` asocia la lista a recorrer con el objeto iterador. En la clase iterador se implementa la función `siguiente()`, cada llamada a `siguiente()`devuelve el puntero al nodo actual de la lista y avanza al siguiente nodo. Una vez visitados todos los nodos de la lista devuelve `NULL`.

La función `main()` genera los números aleatorios, que se insertan en la lista doble. A continuación, se pide el rango de elementos a eliminar; con el objeto iterador se obtienen los nodos y aquéllos fuera de rango se borran de la lista.

```
// archivo con la clase NodoDoble
#include "NodoDoble.h"

// declaración friend en la clase ListaDoble
class IteradorLista;
class ListaDoble
{
  friend class IteradorLista;
    // ...
};
void ListaDoble::visualizar()
{
  NodoDoble* n;
  int k = 0;
  n = cabeza;
  while (n != NULL)
  {
    char c;
    k++; c = (k % 15 != 0 ? ' ' : '\n');
    cout << n -> datoNodo() << c;
    n = n -> adelanteNodo();
  }
}
// archivo con la clase IteradorLista
class IteradorLista
{
protected:
    NodoDoble* actual;
public:
    IteradorLista(ListaDoble& ld)
    {
      actual = ld.cabeza;
    }
    NodoDoble* siguiente()
    {
```

```
      NodoDoble* a;
      a = actual;
      if (actual != NULL)
      {
        actual = actual -> adelanteNodo();
      }
      return a;
    }
  };

/*
   función main(). Crea el objeto lista doble e inserta
   datos enteros generados aleatoriamente.
   Crea objeto iterador de lista, para recorrer sus elementos y
   aquellos fuera de rango se eliminan. El rango se lee del teclado.
*/

#include <stdlib.h>
#include <time.h>
#define N 999
#define randomize (srand(time(NULL)))
#define random(num) (rand()%(num))
#include <iostream>
#include "NodoDoble.h"
#include "ListaDoble.h"
#include "IteradorLista.h"
using namespace std;
typedef int Dato;

int main()
{
  int d, x1, x2, m;

  ListaDoble listaDb;
  cout << "Número de elementos de la lista: ";
  cin >> m;
  for (int j = 1; j <= m; j++)
  {
    d = random(N) + 1;
    listaDb.insertarCabezaLista(d);
  }

  cout << "Elementos de la lista original" << endl;
  listaDb.visualizar();
  // rango de valores
  cout << "\nRango que va a contener la lista: " << endl;
  cin >> x1 >> x2;
  // recorre la lista con el iterador
  IteradorLista *iterador;
  iterador = new IteradorLista(listaDb);
  NodoDoble* a;

  a = iterador -> siguiente();
  while (a != NULL)
```

```
  {
    int w;
    w = a -> datoNodo();
    if (!(w >= x1 && w <= x2))
      // fuera de rango
      listaDb.eliminar(w);
    a = iterador -> siguiente();
  }

  // muestra los elementos de la lista
  cout << "Elementos actuales de la lista" << endl;
  listaDb.visualizar();
  return 0;
}
```

10.9. LISTAS CIRCULARES

En las listas lineales simples o en las dobles siempre hay un primer nodo (*cabeza*) y un último nodo (*cola*). Una lista circular, por propia naturaleza, no tiene ni principio ni fin. Sin embargo, resulta útil establecer un nodo de acceso a la lista y, a partir de él, al resto de sus nodos.

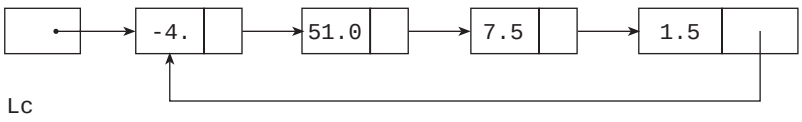


Figura 10.14. Lista circular.

Las operaciones que se realizan sobre una lista circular son similares a las operaciones sobre listas lineales, teniendo en cuenta que no hay *primero* ni `último` nodo, aunque sí un nodo de acceso. Son las siguientes:

- *Inicialización*.
- *Inserción de elementos*.
- *Eliminación de elementos*.
- *Búsqueda de elementos*.
- *Recorrido de la lista circular*.
- *Verificación de lista vacía*.

Nota de programación

Una lista circular es un tipo abstr acto de datos formado por elementos de cualquier tipo y unas operaciones características. En C++ se implementa con la clase *ListaCircular*.

10.9.1. Implementación de la clase ListaCircular

La construcción de una lista circular se puede hacer con enlace simple o enlace doble entre sus nodos. A continuación se implementa utilizando un enlace simple.

La clase `ListaCircular` dispone del puntero de acceso a la lista, junto a las funciones que implementan las operaciones.

La creacción de un nodo varía respecto al de las listas no circulares, el campo `enlace`, en vez de inicializarse a `NULL`, se inicializa para que apunte a sí mismo, de tal forma que es una lista circular de un solo nodo. La funcionalidad (la *interfaz*) de la clase `NodoCircular` es la misma que la de un `Nodo` de una lista enlazada.

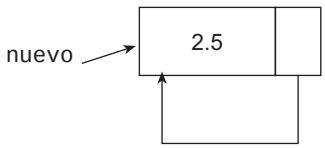


Figura 10.15. Creacción de un nodo en lista circular.

```
// archivo de cabecera NodoCircular.h
typedef int Dato;
class NodoCircular
{
private:
    Dato dato;
    NodoCircular* enlace;
public:
    NodoCircular (Dato entrada)
    {
      dato = entrada;
      enlace = this;    // se apunta a sí mismo
    }
      // ...
};
```

La clase `ListaCircular`:

```
class ListaCircular
{
  private:
    NodoCircular* lc;
  public:
    ListaCircular()
    {
      lc = NULL;
    }
    void insertar(Dato entrada);
    voideliminar(Dato entrada);
    void recorrer();
    void borrarLista();
    NodoCircular* buscar(Dato entrada);
};
```

10.9.2. Insertar un elemento

El algoritmo empleado para añadir o insertar un elemento en una lista circular varía dependiendo de la posición en que se desea insertar. La implementación realizada considera que `lc` tiene la dirección del último nodo, e inserta un nodo en la posición anterior a `lc`.

```
void ListaCircular::insertar(Dato entrada)
{
  NodoCircular* nuevo;
  nuevo = new NodoCircular(entrada);
  if (lc != NULL)     // lista circular no vacía
  {
    nuevo -> ponerEnlace(lc -> enlaceNodo());
    lc -> ponerEnlace(nuevo);
  }
  lc = nuevo;
}
```

10.9.3. Eliminar un elemento

Eliminar un nodo de una lista circular sigue los mismos pasos que los dados para eliminar un nodo en una lista lineal. Hay que enlazar el nodo anterior con el siguiente al que se desea eliminar y liberar la memoria que ocupa. El algoritmo es el siguiente:

1. Búsqueda del nodo.
2. Enlace del nodo anterior con el siguiente.
3. En caso de que el nodo a eliminar sea el de acceso de a la lista, *lc*, se modifica *lc* para que tenga la dirección del nodo anterior.
4. Por último, liberar la memoria ocupada por el nodo.

La implementación debe de tener en cuenta que la lista circular conste de un solo nodo, ya que al eliminarlo la lista se queda vacía. La condición `lc == lc → enlaceNodo()` determina si la lista consta de un solo nodo.

La función recorre la lista buscando el nodo con el dato a eliminar, utiliza un puntero al nodo *anterior* para que cuando encuentre el nodo se enlace con el *siguiente*. Se accede al dato del nodo con la sentencia: `actual→ enlaceNodo()→ datoNodo()`, de tal forma que si coincide con el dato a eliminar, en `actual` está la dirección el nodo anterior. Después del bucle se vuelve a preguntar por el campo dato, ya que no se comparó el nodo de acceso a la lista y el bucle puede terminar sin encontrar el nodo.

```
void ListaCircular:: eliminar (Dato entrada)
{
  NodoCircular* actual;
  bool encontrado = false;

  actual = lc;
  while ((actual -> enlaceNodo() != lc) && (!encontrado))
  {
    encontrado = (actual->enlaceNodo()->datoNodo() == entrada);
    if (!encontrado)
```

```
    {
      actual = actual -> enlaceNodo();
    }
  }
  encontrado = (actual->enlaceNodo()->datoNodo() == entrada);
  // Enlace de nodo anterior con el siguiente
  if (encontrado)
  {
    NodoCircular* p;
    p = actual -> enlaceNodo();     // Nodo a eliminar
    if (lc == lc -> enlaceNodo())   // Lista de un nodo
     lc = NULL;
    else
    {
      if (p == lc)
          lc = actual; // el nuevo acceso a la lista es el anterior
      actual -> ponerEnlace(p -> enlaceNodo());
    }
    delete p;
  }
}
```

10.9.4. Recorrer una lista circular

Una operación común de todas las estructuras enlazadas es recorrer o visitar todos los nodos de la estructura. En una lista circular el recorrido puede empezar en cualquier nodo, a partir de uno dado procesa cada nodo hasta alcanzar el nodo de partida. La función, miembro de la clase `ListaCircular`, inicia el recorrido en el nodo siguiente al de acceso a la lista, `lc`, y termina cuando alcanza de nuevo al nodo `lc`. El proceso que se realiza con cada nodo consiste en escribir su contenido.

```
void ListaCircular:: recorrer()
{
  NodoCircular* p;
  if (lc != NULL)
  {
    p = lc -> enlaceNodo();  // siguiente nodo al de acceso
    do {
      cout << "\t" << p -> datoNodo();
      p = p -> enlaceNodo();
    }while(p != lc -> enlaceNodo());
  }
  else
     cout << "\t Lista Circular vacía." << endl;
}
```

EJERCICIO 10.3. Crear una lista circular con palabras leídas del teclado. El programa debe presentar estas opciones:

- *Mostrar las cadenas que forman la lista.*
- *Borrar una palabra dada.*
- *Al terminar la ejecución, recorrer la lista eliminando los nodos.*

El atributo dato del nodo de la lista es de tipo `string`. Las cadenas se leen del teclado con la función `getline()`, cada cadena leída se inserta en la lista circular, llamando a la función `insertar()` de la clase `ListaCircular`.

La función `eliminar()` busca el nodo que tiene una palabra y le retira de la lista. La comparación de cadenas, tipo `string`, se puede realizar con el operador == (está sobrecargado). Con el fin de recorrer la lista circular liberando cada nodo, se implementa `borrarLista()` de la clase `ListaCircular`.

```
void ListaCircular::borrarLista()
{
  NodoCircular* p;
  if (lc != NULL)
  {
    p = lc;
    do {
      NodoCircular* t;
      t = p;
      p = p -> enlaceNodo();
      delete t;       // no es estrictamente necesario
     }while(p != lc);
  }
  else
    cout << "\n\t Lista vacía." << endl;
  lc = NULL;
}

/*
   función main(): escribe un sencillo menu para
   elegir operaciones con la lista circular.
*/

#include <iostream>
using namespace std;
typedef string Dato;
#include "NodoCircular.h"
#include "ListaCircular.h"

int main()
  {
  ListaCircular listaCp;
  int opc;
  char palabra[81];

  cout << "\n Entrada de Nombres. Termina con FIN\n";
  do
  {
  cin.getline(palabra,80);
  if (strcmp(palabra,"FIN") != 0)
  listaCp.insertar(palabra);
  }while (strcmp(palabra,"FIN")!=0);

  cout << "\t\tLista circular de palabras" << endl;
  listaCp.recorrer();
```

```
      cout << "\n\t Opciones para manejar la lista" << endl;
      do {
        cout << "1. Eliminar una palabra.\n";
        cout << "2. Mostrar la lista completa.\n";
        cout << "3. Salir y eliminar la lista.\n";
        do {
          cin >> opc;
        }while (opc < 1 || opc > 3);

        switch (opc) {
        case 1: cout << "Palabra a eliminar: ";
                cin.ignore();
                cin.getline(palabra,80);
                cin.ignore();
                listaCp.eliminar(palabra);
                break;
        case 2: cout << "Palabras en la Lista:\n";
                listaCp.recorrer(); cout << endl;
                break;
        case 3: cout << "Eliminación de la lista." << endl;
                listaCp.borrarLista();
       }
    }while (opc != 3);
    return 0;
  }
```

10.10. LISTAS ENLAZADAS GENÉRICAS

La definición de una lista está muy ligada al tipo de datos de sus elementos; así, se han puesto ejemplos en los que el tipo es `int`, otros, en los que el tipo es `double`, otros, `string`. C++ dispone del mecanismo `template` para declarar clases y funciones con independencia del tipo de dato de al menos un elemento. Entonces, el tipo de los elementos de una lista gerérica será *un tipo genérico T*, que será conocido en el momento de crear la lista.

```
template <class T> class ListaGenerica
template <class T> class NodoGenerico

ListaGenerica<double> lista1;   // lista de número reales
ListaGenerica<string> lista2;   // lista de cadenas
```

10.10.1. Declaración de la clase `ListaGenérica`

Las operaciones del tipo *lista genérica* son las especificadas en el Apartado 10.2. En el Apartado 10.3 se delaró la clase `NodoGenerico`, ahora se declara y se implementa la clase `ListaGenerica`.

```
// archivo ListaGenerica.h

template <class T> class ListaGenerica
{
```

```
protected:
  NodoGenerico<T>* primero;

public:
  ListaGenerica(){ primero = NULL;}
  NodoGenerico<T>* leerPrimero() const { return primero;}

  void insertarCabezaLista(T entrada);
  void insertarUltimo(T entrada);
  void insertarLista(NodoGenerico<T>* anterior, T entrada);
  NodoGenerico<T>* ultimo();
  void eliminar(T entrada);
  NodoGenerico<T>* buscarLista(T destino);
 };
```

A continuación, la implementación de las funciones miembro, que también son funciones genéricas.

```
// inserción por la cabeza de la lista
template <class T>
void ListaGenerica<T>::insertarCabezaLista(T entrada)
{
  NodoGenerico<T>* nuevo;
  nuevo = new NodoGenerico<T>(entrada);
  nuevo -> ponerEnlace(primero);// enlaza nuevo con primero
  primero = nuevo; // mueve primero y apunta al nuevo nodo
}
// inserción por la cola de la lista
template <class T>
void ListaGenerica<T>::insertarUltimo(T entrada)
{
  NodoGenerico<T>* ultimo = this -> ultimo();
  ultimo -> ponerEnlace(new NodoGenerico<T>(entrada));
}
// recorre hasta el último nodo la lista
template <class T>
NodoGenerico<T>* ListaGenerica<T>::ultimo()
{
  NodoGenerico<T>* p = primero;
  if (p == NULL ) throw "Error, lista vacía";
  while (p -> enlaceNodo()!= NULL) p = p -> enlaceNodo();
  return p;
}
// inserción entre dos nodos de la lista
template <class T> void
ListaGenerica<T>::insertarLista(NodoGenerico<T>* ant, T entrada)
{
  NodoGenerico<T>* nuevo = new NodoGenerico<T>(entrada);
  nuevo -> ponerEnlace(ant -> enlaceNodo());
  ant -> ponerEnlace(nuevo);
}
// búsqueda, si el elemento correspondiente a T es una clase
// debe redefinir el operador de comparación ==
template <class T>
NodoGenerico<T>* ListaGenerica<T>::buscarLista(T destino)
```

```
{
  NodoGenerico<T>* indice;
  for (indice = primero; indice!= NULL; indice = indice->enlaceNodo())
    if (destino == indice -> datoNodo())
      return indice;
  return NULL;
}
// borra el primer nodo encontrado con dato
template <class T>
void ListaGenerica<T>:: eliminar(T entrada)
{
  NodoGenerico<T> *actual, *anterior;
  bool encontrado;
  actual = primero;
  anterior = NULL;
  encontrado = false;
// búsqueda del nodo y del anterior
  while ((actual != NULL) && !encontrado)
  {
    encontrado = (actual -> datoNodo() == entrada);
    if (!encontrado)
    {
      anterior = actual;
      actual = actual -> enlaceNodo();
    }
  }
  // enlace del nodo anterior con el siguiente
  if (actual != NULL)
  {
    // Distingue entre cabecera y resto de la lista
    if (acual == primero)
    {
      primero = actual -> enlaceNodo();
    }
    else
      anterior -> ponerEnlace(actual -> enlaceNodo());
    delete actual;
    }
}
```

10.10.2. Iterador de `ListaGenerica`

Un objeto `Iterador` se diseña para recorrer los elementos de un contenedor. Un iterador de una lista enlazada accede a cada uno de sus nodos de la lista, hasta alcanzar el último elemento. El constructor del objeto iterador inicializa el puntero `actual` al primer elemento de la estructura; la función `siguiente()` devuelve el elemento `actual` y hace que éste quede apuntando al siguiente elemento. Si no hay siguiente, devuelve `NULL`.

La clase `ListaIterador` implementa el iterador de una lista enlazada genérica y, en consecuencia, también será clase genérica.

```
template <class T> class ListaIterador
{
```

```
private:
  NodoGenerico<T> *prm, *actual;
public:
  ListaIterador(const ListaGenerica<T>& list)
  {
    prm = actual = list.leerPrimero();
  }

  NodoGenerico<T> *siguiente()
  {
    NodoGenerico<T> * s;
    if (actual != NULL)
    {
      s = actual;
      actual = actual -> enlaceNodo();
    }
    return actual;
  }
  void iniciaIterador()     // pone en actual la cabeza de la lista
  {
    actual = prm;
  }
}
```

RESUMEN

Una **lista enlazada** es una estructura de datos dinámica, que se crea vacía y crece o decrece en tiempo de ejecución. Los componentes de una lista están ordenados por sus campos de enlace en vez de ordenados físicamente como están en un array. El final de la lista se señala mediante una constante o puntero especial llamado `NULL`. La principal ventaja de una lista enlazada sobre un array radica en el tamaño dinámico de la lista, ajustándose al número de elementos. Por contra, la desventaja de la lista frente a un array está en el acceso a los elementos, para un array el acceso es directo, a partir del índice, para la lista el acceso a un elemento se realiza mediante el campo enlace entre nodos.

Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único, a menos que sea el último, en cuyo caso no se enlaza con ningún otro nodo.

Cuando se inserta un elemento en una lista enlazada, se deben considerar cuatro casos: añadir a una lista vacía, añadir al principio de la lista, añadir en el interior y añadir al final de la lista.

Para borrar un elemento, primero hay que buscar el nodo que lo contiene y considerar dos casos: borrar el primer nodo y borrar cualquier otro nodo de la lista.

El recorrido de una lista enlazada significa pasar por cada nodo (*visitar*) y procesarlo. El proceso de cada nodo puede consistir en escribir su contenido, modificar el campo dato, ...

Una **lista doblemente enlazada** es aquélla en la que cada nodo tiene una referencia a su sucesor y otra a su predecesor. Las listas doblemente enlazadas se pueden recorrer en ambos sentidos. Las operaciones básicas son inserción, borrado y recorrer la lista; similares a las listas simples.

Una **lista enlazada circularmente** por propia naturaleza no tiene primero ni último nodo. Las listas circulares pueden ser de enlace simple o doble.

Una **lista enlazada genérica** tiene como tipo de dato el *tipo genérico T*, es decir, el tipo concreto se especificará en el momento de crear el objeto lista. La construcción de una *lista genérica* se realiza con las plantillas (`template`), mediante dos clases genéricas: `NodoGenerico` y `ListaGenerica`.

EJERCICIOS

10.1. Escribir un función, en la clase `Lista`, que devuelva cierto si la lista está vacía.

10.2. Añadir a la clase `ListaDoble` un función que devuelva el número de nodos de una lista doble.

10.3. En una lista enlazada de números enteros se desea añadir un nodo entre dos nodos consecutivos cuyos datos tienen distinto signo. El nuevo nodo debe ser la diferencia en valor absoluto de los dos nodos.

10.4. A la clase *Lista* añadir la función `eliminaraPosicion()` que retire el nodo que ocupa la posición i, siendo 0 la posición del nodo cabecera.

10.5. Escribir un función que tenga como argumento el puntero al primer nodo de una lista enlazada, y cree una lista doblemente enlazada con los mismos atributos dato pero en orden inverso.

10.6. Se tiene una lista simplemente enlazada de números reales. Escribir una función para obtener una lista doble ordenada respecto al atributo dato, con los valores reales de la lista simple.

10.7. Escribir una función para crear una lista doblemente enlazada de palabras introducidas por teclado. El acceso a la lista debe ser el nodo que está en la posición intermedia.

10.8. La clase `ListaCircular` dispone de las funciones que implementan las operaciones de una lista circular de palabras. Escribir un función miembro que cuente el número de veces que una palabra dada se encuentra en la lista.

10.9. Escribir una función que devuelva el mayor entero de una lista enlazada de números enteros.

10.10. Se tiene una lista de simple enlace, el campo dato son objetos *Alumno* con las variables: *nombre, edad, sexo*. Escribir una función para transformar la lista de tal forma que si el primer nodo es de un alumno de sexo masculino el siguiente sea de sexo femenino, así alternativamente, siempre que sea posible, masculino y femenino.

10.11. Supóngase una lista circular de cadenas ordenada alfabéticamente. El puntero de acceso a la lista tiene la dirección del nodo alfabéticamente mayor. Escribir un función para añadir una nueva palabra, en el orden que le corresponda, a la lista.

10.12. Dada la lista del Ejercicio 10.11 escribir un función que elimine una palabra dada.

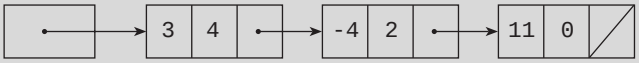
PROBLEMAS

10.1. Escribir un programa que realicen las siguientes tareas:

- Crear una lista enlazada de números enteros positivos al azar. Insertar por el último nodo.
- Recorrer la lista para mostrar los elementos por pantalla.
- Eliminar todos los nodos que superen un valor dado.

10.2. Se tiene un archivo de texto de palabras separadas por un blanco o el carácter fin de línea. Escribir un programa para formar una lista enlazada con las palabras del archivo. Una vez formada la lista, añadir nuevas palabras o borrar alguna de ellas. Al finalizar el programa escribir las palabras de la lista en el archivo.

10.3. Un polinomio se puede representar como una lista enlazada. El primer nodo representa el primer término del polinomio, el segundo nodo al segundo término del polinomio y así sucesivamente. Cada nodo tiene como campo dato el coeficiente del término y su exponente. Por ejemplo, $3x^4 - 4x^2 + 11$ se representa:



Escribir un programa que dé entrada a polinomios en x, los represente en una lista enlazada simple. A continuación, obtenga valores del polinomio para valores de `x = 0.0, 0.5, 1.0, 1.5, … , 5.0`

10.4. Teniendo en cuenta la representación de un polinomio propuesta en el Problema 10.3, hacer los cambios necesarios para que la lista enlazada sea circular. La referencia de acceso debe tener la dirección del último término del polinomio, el cuál apuntará al primer término.

10.5. Según la representación de un polinomio propuesta en el Problema 10.4, escribir un programa para realizar las siguientes operaciones:

- Obtener la lista circular suma de dos polinomios.
- Obtener el polinomio derivada.
- Obtener una lista circular que sea el producto de dos polinomios.

10.6. Escribir un programa para obtener una lista doblemente enlazada con los caracteres de una cadena leída desde el teclado. Cada nodo de la lista tendrá un carácter. Una vez que se haya creado la lista, ordenarla alfabéticamente y escribirla por pantalla.

10.7. Un conjunto es una secuencia de elementos todos del mismo sin duplicados. Escribir un programa para representar un conjunto de enteros con una lista enlazada. El programa debe contemplar las operaciones:

- Cardinal del conjunto.
- Pertenencia de un elemento al conjunto.
- Añadir un elemento al conjunto.
- Escribir en pantalla los elementos del conjunto.

10.8. Con la representación propuesta en el Problema 10.7, añadir las operaciones básicas de conjuntos:

- Unión de dos conjuntos.
- Intersección de dos conjuntos.
- Diferencia de dos conjuntos.
- Inclusión de un conjunto en otro.

10.9. Escribir un programa en el que dados dos archivos *F1*, *F2* formados por palabras separadas por un blanco o fin de línea, se creen dos conjuntos con las palabras de *F1* y *F2* respectivamente. Posteriormente, encontrar las palabras comunes y mostrarlas por pantalla.

10.10. Utilizar una lista doblemente enlazada para controlar una lista de pasajeros de una línea aérea. El programa principal debe ser controlado por menú y permitir al usuario visualizar los datos de un pasajero determinado, insertar un nodo (siempre por el final), eliminar un pasajero de la lista. A la lista se accede por dos variables, una referencia al primer nodo y la otra al último nodo.

10.11. Para representar un entero largo, de más de 30 dígitos, utilizar una lista circular cuyos nodos tienen como atributo dato un dígito del entero largo. Escribir un programa cuya entrada sea dos enteros largos y se obtenga su suma.

10.12. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa que represente un vector disperso con listas enlazadas. Los nodos son los elementos del vector distintos de cero. Cada nodo contendrá el valor del elemento y su índice (posición del vector). El programa ha de realizar las operaciones: sumar dos vectores de igual dimensión y hallar el producto escalar.

Capítulo 11
Pilas

Objetivos

Con el estudio de este capítulo usted podrá:

- Especificar el tipo abstracto de datos Pila.
- Conocer aplicaciones de las Pilas en la programación.
- Definir e implementar la clase Pila.
- Conocer las diferentes formas de escribir una expresión.
- Evaluar una expresión algebraica.

Contenido

Conceptos clave

- Concepto de tipo abstracto de datos.
- Concepto de una pila.
- Expresiones y sus tipos.
- Listas enlazadas.
- Notación de una expresión.
- Prioridad.

INTRODUCCIÓN

En este capítulo se estudian en detalle la estructura de datos *Pila* utilizada frecuentemente en la resolución de algoritmos. La *Pila* es una estructura de datos que almacena y recupera sus elementos atendiendo a un estricto orden. Las pilas se conocen también como estructuras **LIFO** (*Last-in, First-Out, último en entrar primero en salir*). El desarrollo de las pilas como tipos abstractos de datos es el motivo central de este capítulo.

Las pilas se utilizan en compiladores, sistemas operativos y programas de aplicaciones. Una aplicación interesante es la evaluación de expresiones algebraicas mediante pilas.

11.1. CONCEPTO DE PILA

Una **pila** (*stack*) es una colección ordenada de elementos a los que sólo se puede acceder por un único lugar o extremo. Los elementos de la pila se añaden o quitan (borran) de la misma sólo por su parte superior, la **cima** de la pila. Éste es el caso de una pila de platos, una pila de libros, etc.

Definición

Una pila es una estructura de datos de entradas ordenadas tales que sólo se pueden introducir y eliminar por un extremo, llamado cima.

Cuando se dice que la pila está ordenada, se quiere decir que hay un elemento al que se puede acceder primero (el que está encima de la pila), otro elemento al que se puede acceder en segundo lugar (justo el elemento que está debajo de la cima), un tercero, etc. No se requiere que las entradas se puedan comparar utilizando el operador "*menor que*" y pueden ser de cualquier tipo.

Las entradas de la pila deben ser eliminadas en el orden inverso al que se situaron en la misma. Por ejemplo, se puede crear una pila de libros, situando primero un diccionario, encima de él una enciclopedia y encima de ambos una novela de modo que la pila tendrá la novela en la parte superior.

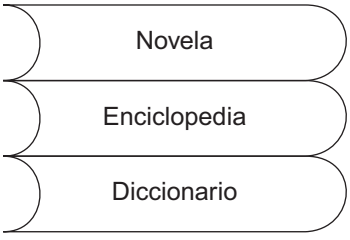


Figura 11.1. Pila de libros.

Cuando se quitan los libros de la pila, primero debe quitarse la novela, luego la enciclopedia y por último el diccionario.

Debido a su propiedad específica "*último en entrar, primero en salir*" se conoce a las pilas como estructura de datos **LIFO** (*last-in, first-out*).

Las operaciones usuales en la pila son *Insertar* y *Quitar*. La operación **`Insertar`** (`push`) añade un elemento en la cima de la pila y la operación **`Quitar`** (pop) elimina o saca un elemento de la pila. La Figura 11.2 muestra una secuencia de operaciones *Insertar* y *Quitar*.

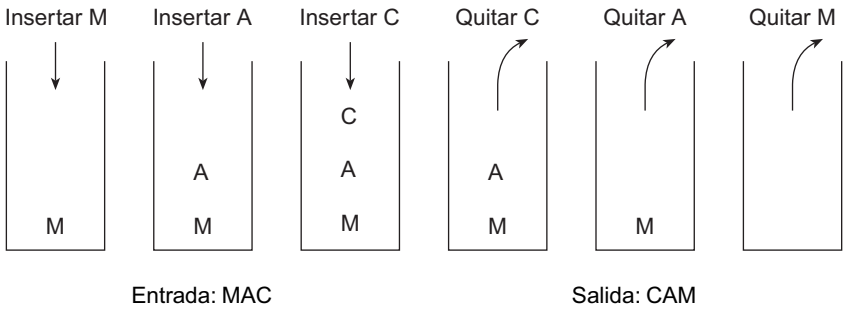


Figura 11.2. Insertar y quitar elementos de la pila.

La operación `insertar` (*push*) sitúa un elemento en la cima de la pila y `quitar` (*pop*) elimina o extrae el elemento cima de la pila.

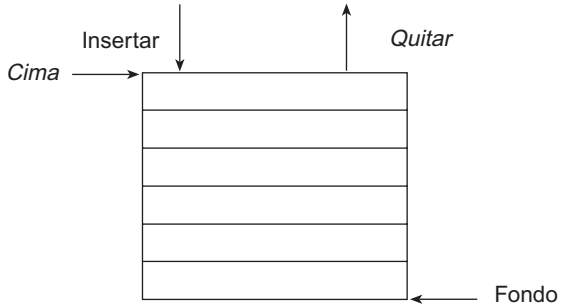


Figura 11.3. Operaciones básicas de una pila.

La pila se puede implementar guardando los elementos en un array, en cuyo caso su dimensión o longitud es fija. Otra forma de implementación consiste en construir una lista enlazada, cada elemento de la pila forma un nodo de la lista; la lista crece o decrece según se añaden o se extraen, respectivamente, elementos de la pila; ésta es una representación dinámica y no existe limitación en su tamaño excepto la memoria del ordenador.

Una pila puede estar *vacía* o *llena* (en la representación con un array, si se ha llegado al último elemento). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error, una *excepción*, debido a que esa operación es imposible; esta situación se denomina **desbordamiento negativo** (*underflow*). Por el contrario, si un programa intenta poner un elemento en una pila *llena* se produce un error, una *excepción*, de **desbordamiento** (*overflow*) o *rebosamiento*. Para evitar estas situaciones se diseñan funciones, que comprueban si la pila está llena o vacía.

11.1.1. Especificaciones de una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes.

Tipo de dato	Dato que se almacena en la pila
Operaciones	
CrearPila	Inicia la pila.
Insertar (push)	Pone un dato en la pila.
Quitar (pop)	Retira (saca) un dato de la pila.
Pilavacía	Comprobar si la pila no tiene elementos.
Pilallena	Comprobar si la pila está llena de elementos.
Limpiar pila	Quita todos sus elementos y dejar la pila vacía.
CimaPila	Obtiene el elemento cima de la pila.
Tamaño de la pila	Número de elementos máximo que puede contener la pila.

La operación *Pilallena* sólo se implementa cuando se utiliza un array para almacenar los elementos. Una pila puede crecer indefinidamente si se implementa con una estructura dinámica.

11.2. TIPO DE DATO PILA IMPLEMENTADO CON ARRAYS

La implementación con un array es *estática* porque el array es de tamaño fijo. La clase *Pila*, con esta representación, además del array, utiliza la variable `cima` para apuntar (índice) al último elemento colocado en la pila. Es necesario controlar el tamaño de la pila para que no exceda al número de elementos del array, y la condición *Pilallena* será significativa para el diseño.

El método usual de introducir elementos en la pila es definir el *fondo* en la posición `0` del array y sin ningún elemento en su interior, es decir, definir una *pila vacía*; a continuación, se van introduciendo elementos en el array de modo que el primer elemento se introduce en una pila vacía y en la posición `0`, el segundo elemento en la posición `1`, el siguiente en la posición `2` y así sucesivamente. Con estas operaciones el índice que apunta a la cima de la pila va incrementando en `1` cada vez que se añade un nuevo elemento. Los algoritmos de insertar (*push*) y quitar (*pop*) datos de la pila son::

Insertar (*push*)

1. Verificar si la pila no está llena.
2. Incrementar en 1 el apuntador (cima) de la pila.
3. Almacenar el elemento en la posición del apuntador de la pila.

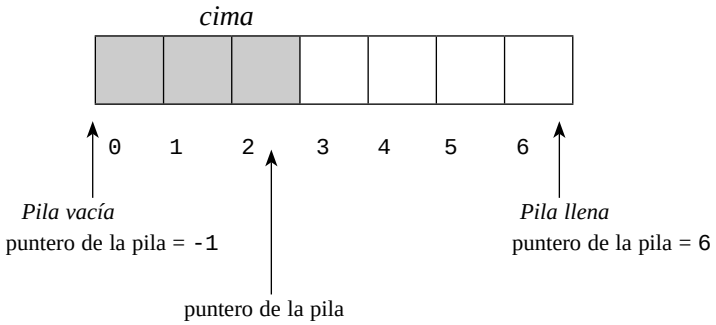
Quitar (*pop*)

1. Si la pila no está vacía.
2. Leer el elemento de la posición del apuntador de la pila.
3. Decrementar en 1 el apuntador de la pila.

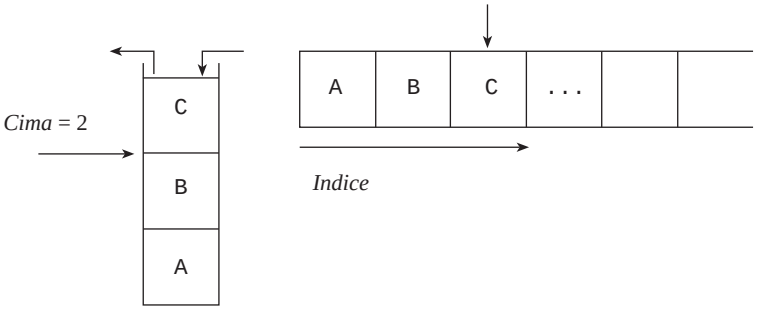
El rango de elementos que puede tener una pila varía de `0` a `TAMPILA-1`, en el supuesto de que el array se defina de tamaño `TAMPILA` elementos. De modo que *en una pila llena* el

apuntador (índice del array) de la pila tiene el valor `TAMPILA-1,` y *en una pila vacía* tendrá `-1` (el valor `0` es el índice del primer elemento).

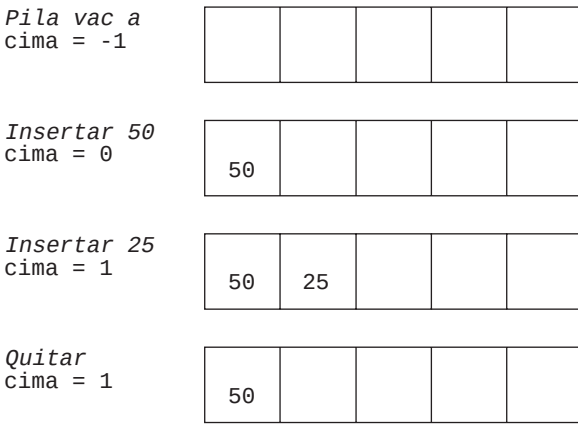
EJEMPLO 11.1. Una pila de 7 elementos se puede representar g ráficamente así:



Si se almacenan los datos `A, B, C`, ... en la pila se puede representar gráficamente de alguna de estas formas



A continuación, se muestra la imagen de una pila según diferentes operaciones realizadas.



11.2.1. Especificación de la clase `Pila`

La declaración de un tipo abstracto incluye la representación de los datos y la definición de las operaciones. En el *TAD* `Pila` los datos pueden ser de cualquier tipo y las operaciones las ya citadas en 11.1.1.

1. Datos de la pila (`TipoDato` es cualquier tipo de dato primitivo o tipo clase).
2. `crearPila` inicializa una pila. Crear una pila sin elementos, por tanto, vacía.
3. Verificar que la pila no está llena antes de `insertar` o poner ("*push*") un elemento en la pila; verificar que una pila no está vacía antes de `quitar` o sacar ("*pop*") un elemento de la pila. Si estas precondiciones no se cumplen se debe visualizar un mensaje de error (una *excepción*) y el programa debe terminar.
4. `pilaVacia` devuelve *verdadero* si la pila está vacía y *falso* en caso contrario.
5. `pilaLlena` devuelve *verdadero* si la pila está llena y *falso* en caso contrario. Estas dos últimas operaciones se utilizan para verificar las precondiciones de *insertar* y *quitar*.
6. `limpiarPila` vacía la pila, dejándola sin elementos y disponible para otras tareas.
7. `cimaPila`, devuelve el valor situado en la cima de la pila, pero no se decrementa el puntero de la pila, ya que la pila queda intacta.

Declaración de la clase `Pila`

```
typedef tipo TipoDeDato;  // tipo de los elementos de la pila
// archivo de cabecera pilalineal.h
const int TAMPILA = 49;
class PilaLineal
{
  private:
    int cima;
    TipoDeDato listaPila[TAMPILA];
  public:
    PilaLineal()
    {
      cima = -1;  // condición de pila vacía
    }
    // operaciones que modifican la pila
    void insertar(TipoDeDato elemento);
    TipoDeDato quitar();
    void limpiarPila();
    // operación de acceso a la pila
    TipoDeDato cimaPila();
    // verificación estado de la pila
    bool pilaVacia();
    bool pilaLlena();
};
```

La declaración realizada está ligada al tipo de los elementos de la pila. Para alcanzar la máxima abstracción, se declara la clase genérica *PilaLineal* de tal forma que el tipo de dato de los elementos se especifica al crear el objeto pila.

EJEMPLO 11.2. Escribir un programa que cree una pila de enteros. Se realicen operaciones de añadir datos a la pila, quitar ...

Se supone implementada la clase pila con el tipo primitivo `int`. El programa crea una pila de números enteros, inserta en la pila elementos leídos del teclado (hasta leer la clave `-1`), a continuación, extrae los elementos de la pila hasta que se vacía. En pantalla deben escribirse los números leídos en orden inverso por la naturaleza de la pila. El bloque de sentencias se encierra en un bloque `try` para tratar errores de desbordamiento de la pila.

```
#include <iostream>
using namespace std;
typedef int TipoDeDato;
#include "pilalineal.h"

int main()
{
  PilaLineal pila;      // crea pila vacía
  TipoDeDato x;
  const TipoDeDato CLAVE = -1;

  cout << "Teclea elemento de la pila(termina con -1)" << endl;
  try {
    do {
      cin >> x;
      pila.insertar(x);
    }while (x != CLAVE);

    // proceso de la pila
    cout << "Elementos de la Pila: " ;
    while (!pila.pilaVacia())
    {
      x = pila.quitar();
      cout << x << " ";
    }
  }
  catch (const char * error)
  {
    cout << "Excepción: " << error;
  }
  return 0;
}
```

11.2.2. Implementación de las operaciones sobre pilas

Las funciones de la clase `Pila` son sencillas de implementar, teniendo en cuenta la característica principal de esta estructura: *inserciones y borrados se realizan por el mismo extremo, la cima de la pila*.

La operación de `insertar` un elemento en la pila, incrementa el apuntador `cima` y asigna el nuevo elemento a la lista. Cualquier intento de añadir un elemento en una pila llena genera una excepción o error debido al "*Desbordamiento de la pila*".

```
void PilaLineal::insertar(TipoDeDato elemento)
{
```

```
  if (pilaLlena())
  {
    throw "Desbordamiento pila";
  }
  //incrementar puntero cima y copia elemento
  cima++;
  listaPila[cima] = elemento;
}
```

La operación `quitar` elimina un elemento de la pila copiando, en primer lugar, el valor de la cima de la pila en una variable local, `aux`, y a continuación, decrementa el puntero de la pila. `quitar()` devuelve la variable `aux`, es decir, el elemento eliminado por la operación. Si se intenta eliminar o borrar un elemento en una pila vacía se produce error, se lanza una excepción.

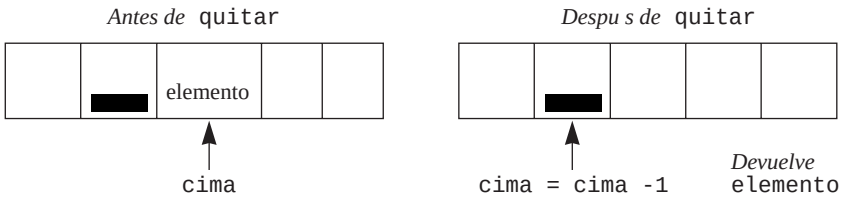


Figura 11. 4. Extraer elemento cima.

```
TipoDeDato PilaLineal::quitar()
{
  TipoDeDato aux;
  if (pilaVacia())
  {
    throw "Pila vacía, no se puede extraer.";
  }
    // guarda elemento de la cima
    aux = listaPila[cima];
    // decrementar cima y devolver elemento
    cima--;
    return aux;
}
```

La operación `cimaPila` devuelve el elemento que se encuentra en la cima de la pila, no se modifica la pila, únicamente accede al elemento.

```
TipoDeDato PilaLineal::cimaPila()
{
  if (pilaVacia())
  {
    throw "Pila vacía, no hay elementos.";
  }
  return listaPila[cima];
}
```

11.2.3. Operaciones de verificación del estado de la pila

Se debe proteger la integridad de la pila, para lo cual el tipo `Pila` ha de proporcionar operaciones que comprueben el estado de la pila: *pila vacía* o *pila llena*. Asimismo, se ha

de definir una operación, `LimpiarPila`, que restaure la condición inicial de la pila, cima igual a -1.

La función `pilaVacia` comprueba si la cima de la pila es -1. En cuyo caso la pila está vacía y devuelve *verdadero*.

```
bool PilaLineal::pilaVacia()
{
  return cima == -1;
}
```

La función `pilaLlena` comprueba si la cima es `TAMPILA-1`; en cuyo caso la pila está llena y devuelve *verdadero*.

```
bool PilaLineal::pilaLlena()
{
  return cima == TAMPILA - 1;
}
```

Por último, `limpiarPila()` pone la cima de la pila a su valor inicial.

```
void PilaLineal::limpiarPila()
{
  cima = -1;
}
```

EJERCICIO 11.1. Escribir un programa que utilice una `Pila` para comprobar si una determinada frase/palabra (cadena de caracteres) es un palíndromo. Nota: una palabra o frase es un palíndromo cuando la lectura directa e indirecta de la misma tiene igual valor: **alila**, es un palíndromo; **cara (arac)** no es un palíndromo.

La palabra se lee con la función `gets()` y se almacena en un `string`; cada carácter de la palabra se pone en una pila de caracteres. Una vez leída la palabra y construida la pila, se compara el primer carácter del `string` con el carácter que se extrae de la pila, si son iguales sigue la comparación con el siguiente carácter del `string` y de la pila; así sucesivamente hasta que la pila se queda vacía o hay un carácter no coincidente.

Al guardar los caracteres de la palabra en la pila se garantiza que las comparaciones de caracteres se realizan en orden inverso: primero con último … .

No es necesario volver a implementar las operaciones de la clases `Pila`, simplemente se cambia el tipo de dato de los elementos, en esta ocasión `char`.

```
#include <iostream>
using namespace std;
#include <string.h>
typedef char TipoDeDato;
#include "pilalineal.h"

int main()
{
  PilaLineal pilaChar;   // crea pila vacía
  TipoDeDato ch;
  bool esPal;
  char pal[81];
```

```
  cout << "Teclea la palabra verificar si es palíndromo: ";
  gets(pal);
  for (int i = 0; i < strlen(pal); )
  {
    char c;
    c = pal[i++];
    pilaChar.insertar(c);
  }
  // se comprueba si es palíndromo

  esPal = true;
  for (int j = 0; esPal && !pilaChar.pilaVacia(); )
  {
    char c;
    c = pilaChar.quitar();
    esPal = pal[j++] == c;
  }
  pilaChar.limpiarPila();
  if (esPal)
    cout << "La palabra " << pal << " es un palíndromo \n";
  else
    cout << "La palabra " << pal << " no es un palíndromo \n";
  return 0;
}
```

EJEMPLO 11.3. Llenar una pila con números leídos del teclado. A continuación vaciar la pila de tal forma que se muestren los valores positivos.

En este ejemplo el tipo de los elementos de la pila es `double`. El número de elementos que tendrá la pila se solicita al usuario; en un bucle *for* se lee el elemento y se inserta en la pila. Para vaciar la pila se diseña un bucle, *hasta pila vacía*; cada elemento que se extrae se escribe si es positivo.

```
#include <iostream>
using namespace std;
typedef double TipoDeDato;
#include "pilalineal.h"

int main()
{
  PilaLineal pila;
  int x;

  cout << "Teclea número de elementos: ";
  cin >> x;
  for (int j = 1; j <= x; j++)
  {
    double d;
    cin >> d;
    pila.insertar(d);
  }
  // vaciado de la pila
```

```
  cout << "Elementos de la Pila: ";
  while (!pila.pilaVacia())
  {
    double d;
    d = pila.quitar();
    if (d > 0.0)
      cout << d << " ";
  }
  return 0;
}
```

11.3. PILA GENÉRICA CON LISTAS ENLAZADA

La realización dinámica de una pila utilizando una lista enlazada almacena cada elemento de la pila como un nodo de la lista. Como las operaciones de *insertar* y *extraer* en el *TAD Pila* se realizan por el mismo extremo (cima de la pila), las acciones correspondientes con la lista se realizarán siempre por el mismo extremo de la lista.

Esta realización tiene la ventaja de que el tamaño se ajusta exactamente al número de elementos de la pila. Sin embargo, para cada elemento es necesaria más memoria para guardar el campo de enlace entre nodos consecutivos. La Figura 11.5 muestra la imagen de una pila implementada con una lista enlazada.

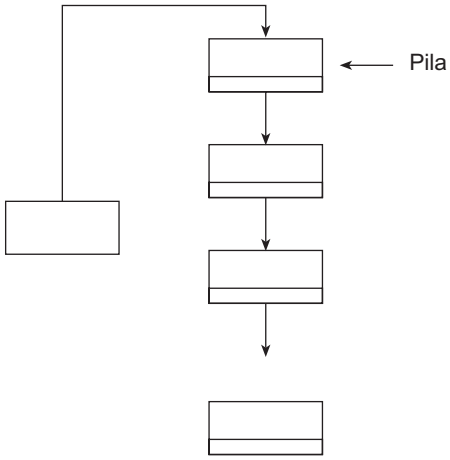


Figura 11.5. Representación de una pila con una lista enlazada.

Nota

Una pila realizada con una lista enlazada crece y decrece dinámicamente . En tiempo de ejecución, se reserva memoria según se ponen elementos en la pila y se libera memoria según se extraen elementos de la pila.

11.3.1. Clase *PilaGenerica* y *NodoPila*

La estructura que tiene la pila implementada con una lista enlaza es muy similar a la expuesta en listas enlazadas. Los elementos de la pila son los nodos de la lista, con un atributo para guardar el elemento y otro de enlace. Las operaciones del tipo pila implementada con listas son, naturalmente, las mismas que si la pila se implementa con arrays, salvo la operación que controla si la pila está llena, *pilaLlena*, que ahora no tiene significado ya que la listas enlazadas crecen indefinidamente, con el único límite de la memoria.

El tipo de dato de elemento se corresponde con el tipo de los elementos de la `Pila`, para que no dependa de un tipo concreto; para que sea genérico, se diseña una *pila genérica* utilizando las *plantillas* (`template`) de C++. La clase `NodoPila` representa un nodo de la lista enlazada, tiene dos atributos: elemento, guarda el elemento de la pila y siguiente, contiene la dirección del siguiente nodo de la lista. En esta implementación, `NodoPila` es una clase interna de `PilaGenerica`.

```
// archivo PilaGenerica.h

template <class T>
class PilaGenerica
{
  class NodoPila
  {

    public:
      NodoPila* siguiente;
      T elemento;
      NodoPila(T x)
      {
        elemento = x;
        siguiente = NULL;
      }
  };
  NodoPila* cima;

  public:
    PilaGenerica ()
    {
      cima = NULL;
    }
    void insertar(T elemento);
    T quitar();
    T cimaPila(); const
    bool pilaVacia(); const
    void limpiarPila();
    ~PilaGenerica()
    {
      limpiarPila();
    }
};
```

11.3.2. Implementación de las operaciones del *TAD Pila* con listas enlazadas

El constructor de `Pila` inicializa a ésta como pila vacía `(cima == NULL)`, realmente, a la condición de *lista vacía*. Las operaciones `insertar`, `quitar` y `cimaPila` acceden a la lista directamente con el puntero cima (apunta al último nodo apilado). Entonces, como no necesitan recorren los nodos de la lista, no dependen del número de nodos, la eficiencia de cada operación es constante, `O(1)`.

La codificación que a continuación se escribe, es para una pila de elemento de cualquier tipo. Es preciso recordar que al crear una instancia de pila es cuando se informa del tipo concreto de sus elementos, por ejemplo `PilaGenerica<char> pila`.

Verificación del estado de la pila

```
template <class T>
bool PilaGenerica<T>::pilaVacia() const
{
   return cima == NULL;
}
```

Poner un elemento en la pila

Crea un nuevo nodo con el elemento que se pone en la pila y se enlaza por la cima.

```
template <class T>
void PilaGenerica<T>::insertar(T elemento)
{
  NodoPila* nuevo;
  nuevo = new NodoPila(elemento);
  nuevo -> siguiente = cima;
  cima = nuevo;
}
```

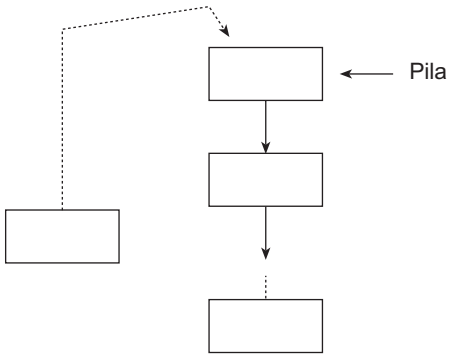


Figura 11.6. Apilar un elemento.

Eliminación del elemento cima

Retorna el elemento cima y lo quita de la pila, disminuye el tamaño de la pila.

```
template <class T>
T PilaGenerica<T>::quitar()
{
  if (pilaVacia())
    throw "Pila vacía, no se puede extraer.";
  T aux = cima -> elemento;
  cima = cima -> siguiente;
  return aux;
}
```

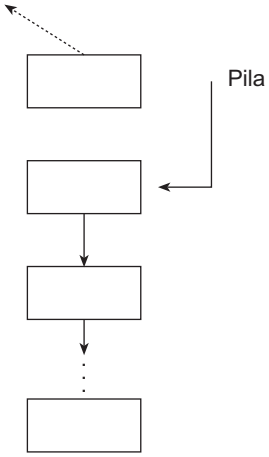


Figura 11.7. Quita la cima de la pila.

Obtención del elemento cima de la pila

```
template <class T>
T PilaGenerica<T>:: cimaPila(); const
{
  if (pilaVacia())
    throw "Pila vacía";
  return cima -> elemento;
}
```

Vaciado de la pila

Libera todos los nodos de que consta la pila. Recorre los `n` nodos de la lista enlazada, es una operación de complejidad lineal, `O(n)`.

```
template <class T>
void PilaGenerica<T>:: limpiarPila()
```

```
{
  NodoPila* n;
  while(!pilaVacia())
  {
    n = cima;
    cima = cima -> siguiente;
    delete n;
  }
}
```

11.4. EVALUACIÓN DE EXPRESIONES ARITMÉTICAS CON PILAS

Una *expresión aritmética* está formada por operandos y operadores. La expresión `x * y - (a + b)` consta de los operadores `*, -, +` y de los operandos `x, y, a, b`. Los operandos pueden ser valores constantes, variables o, incluso, otra expresión. Los operadores son los símbolos conocidos de las operaciones matemáticas.

La evaluación de una expresión aritmética da lugar a un valor numérico, se realiza sustituyendo los operandos que son variables por valores concretos y ejecutando las operaciones aritméticas representadas por los operadores. Si los operandos de la expresión anterior toman los valores: `x = 5, y = 2, a = 3, b = 4` el resultado de la evaluación es:

```
5 * 2 – (3 + 4) = 5 * 2 – 7 = 10 – 7 = 3
```

La forma habitual de escribir expresiones matemáticas sitúa el operador entre sus dos operandos. La expresión anterior está escrita de esa forma, recibe el nombre de *notación infija*. Esta forma de escribir las expresiones exige, en algunas ocasiones, el uso de paréntesis para *encerrar* subexpresiones con mayor prioridad.

Los operadores, como es sabido, tienen distintos niveles de precedencia o prioridad a la hora de su evaluación. A continuación, se recuerda estos niveles de prioridad en orden de mayor a menor:

```
Paréntesis              : ( )
Potencia                : ^
Multiplicación/división : *, /
Suma/Resta              : +, -
```

Normalmente, en una expresión hay operadores con la misma prioridad, a igualdad de precedencia, los operadores se evalúan de izquierda a derecha *(asociatividad)*, excepto la potencia que es de derecha a izquierda.

11.4.1. Notación *prefija* y notación *postfija* de una expresiones aritmética

Las operaciones aritméticas escritas en *notación infija* en muchas ocasiones necesitan usar paréntesis para indicar el orden de evaluación. Las expresiones

```
r = a * b / (a + c)                    g = a * b / a + c
```

son distintas al no poner paréntesis en la expresión `g`. Igual ocurre con estas otras:

```
r = (a - b) ^ c + d                    g = a - b ^ c + d
```

Existen otras formas de escribir expresiones aritméticas, que se diferencian por la ubicación del operador respecto de los operandos. La notación en la que el operador se coloca delante de los dos operandos, *notación prefija*, se conoce también como *notación polaca* por el matemático polaco que la propuso. En el Ejemplo 11.4 se escriben expresiones en notación *prefija* o notación polaca.

EJEMPLO 11.4. Dadas las expresiones: `a * b / (a + c); a * b / a + c; (a-b)^c+d`. Escribir las expresiones equivalentes en notación prefija.

Paso a paso, se escribe la transformación de cada expresión algebraica en la expresión equivalente en notación polaca.

```
a * b / (a + c) (infija) → a * b / + ac → * ab / + ac → / *ab + ac (polaca)
a * b / a + c (infija) → * ab / a + c → / * aba + c → + / *abac (polaca)
(a - b) ^ c + d (infija) → -ab ^ c + d → ^ -abc + d → +^-abcd (polaca)
```

Nota

La propiedad fundamental de la notación polaca es que el orden de ejecución de las operaciones está determinado por las posiciones de los operadores y los operandos en la expresión. No son necesarios los paréntesis al escribir la expresión en notación polaca, como se observa en el Ejemplo 11.4.

Notación postfija

Hay más formas de escribir las expresiones. La notación *postfija* o *polaca inversa* coloca el operador a continuación de sus dos operandos.

EJEMPLO 11.5. Dadas las expresiones: `a*b/(a+c); a*b/a+c; (a-b)^c+d`. Escribir las expresiones equivalentes en notación postfija.

Paso a paso se transforma cada subexpresión en notación polaca inversa.

```
a*b/(a+c) (infija) → a*b/ac+ → ab*/ac+ → ab*ac+/    (polaca inversa)
a*b/a+c (infija) → ab*/a+c → ab*a/+c → ab*a/c+      (polaca inversa)
(a-b)^c+d (infija) → ab-^c+d → ab-c^+d → ab-c^d+    (polaca inversa)
```

Recordar

Las diferentes formas de escribir una misma expresión algebraica dependen de la ubicación de los operadores respecto a los operandos. Es importante tener en cuenta que tanto en la notación prefija como en la postfija no son necesarios los paréntesis para cambiar el orden de evaluación.

11.4.2. Evaluación de una expresión aritmética

La evaluación de una expresión aritmética escrita de *manera habitual*, en *notación infija*, se realiza en dos pasos principales:

1.º Transformar la expresión de notación infija a postfija.
2.º Evaluar la expresión en notación postfija.

El *TAD Pila* es fundamental en los algoritmos que se aplican a cada uno de los pasos. El orden que fija la estructura pila asegura que el *último en entrar es el primero en salir*, y de esa forma el algoritmo de transformación a *postfija* sitúa los operadores después de sus operandos, con la prioridad o precedencia que le corresponde. Una vez que se tiene la expresión en notación *postfija*, se utiliza otra pila, de elementos numéricos, para guardar los valores de los operandos, y de las operaciones parciales con el fin de obtener el valor numérico de la expresión.

11.4.3. Transformación de una expresión infija a *postfija*

Se parte de una expresión en *notación infija* que tiene operandos, operadores y puede tener paréntesis. Los operandos se representan con letras, los operadores son éstos:

```
^ (potenciación), *, /, +, - .
```

La transformación se realiza utilizando una pila para guardar operadores y los paréntesis izquierdos. La expresión aritmética se lee del teclado y se procesa carácter a carácter. Los operandos pasan directamente a formar parte de la expresión en *postfija* la cual se guarda en un array. Un operador se mete en la pila si se cumple que:

- La pila esta vacía, o,
- El operador tiene mayor prioridad que el operador cima de la pila, o bien,
- El operador tiene igual prioridad que el operador cima de la pila y se trata de la máxima prioridad.

Si la prioridad es menor o igual que la de *cima pila*, se saca el elemento cima de la pila, se pone en la expresión en *postfija* y se vuelve a hacer la comparación con el nuevo elemento cima.

El paréntesis izquierdo siempre se mete en la pila; ya en la pila se les considera de mínima prioridad para que todo operador que se encuentra dentro del paréntesis entre en la pila. Cuando se lee un paréntesis derecho se sacan todos los operadores de la pila y pasan a la expresión *postfija*, hasta llegar a un paréntesis izquierdo que se elimina ya que los paréntesis no forman parte de la expresión *postfija*. El algoritmo termina cuando no hay más items de la expresión origen y la pila está vacía.

Por ejemplo, dada la expresión `a*(b+c-(d/e^f)-g)-h` escrita en *notación infija*, a continuación, se va a ir formando, paso a paso, la expresión equivalente en postfija.

Expresión en postfija

`a`	Operando `a` pasa a la expresión *postfija*; operador `*` a la pila.
`ab`	Operador `(` pasa a la pila; operando `b` a la expresión.

`abc` Operador + pasa a la pila; operando a la expresión.

En este punto, el estado de la pila:

El siguiente carácter de la expresión, `-`, tiene igual prioridad que el operador de la cima (+), da lugar:

`abc+`

`abc+d` El operador `(`se mete en la pila; el operando `d` a la expresión.

`abc+de` El operador `/` pasa a la pila; el operando e a la expresión.

`abc+def` El operador `^` pasa a la pila; el operando `f` a la expresión.
El siguiente item,`)` (paréntesis derecho), produce que se vacié la pila hasta un `(` .La pila, en este momento, dispone de estos operadores:

`abc+def^/` El algoritmo saca operadores de la pila hasta un '(' y da lugar a la pila:

`abc+def^/-` El operador – pasa a la pila y, a su vez, se extrae `-`; el siguiente carácter, el operando `g` pasa a la expresión.

`abc+def^/-g` El siguiente carácter es `)`, por lo que son extraídos de la pila los operadores hasta un `(`, la pila queda de la siguiente forma:

`abc+def^/-g-` El siguiente carácter es el operador `-`, hace que se saque de la pila el operador * y se meta en la pila el operador –.

`abc+def^/-g-*` Por último, el operando h pasa directamente a la expresión.

`abc+def^/-g-*h` Fin de entrada, se vacía la pila pasando los operadores a la expresión:
`abc+def^/-g-*h-`

El seguimiento realizado pone de manifiesto la importancia de considerar al paréntesis izquierdo un operador de mínima prioridad dentro de la pila, para que los operadores, dentro de un paréntesis, se metan en la pila y después extraerlos cuando se trata el paréntesis derecho. También tiene un comportamiento distinto el operador de potenciación dentro y fuera de la pila, debido a que tienen asociatividad de derecha a izquierda. Las prioridades se fijan en la Tabla 11.1.

Tabla 11.1. Tabla de prioridades de los operadores considerados.

Operador	Prioridad dentro pila	Prioridad fuera pila
^	3	4
*, /	2	2
+, -	1	1
(	0	5

Observe, que el paréntesis derecho no se considera ya que éste provoca sacar operadores de la pila hasta el paréntesis izquierdo.

Algoritmo de paso de notación *infija* a *postfija*

Los pasos a seguir para transformar una expresión algebraica de *notación infija* a *postfija*:

1. Obtener caracteres de la expresión y repetir los pasos 2 al 4 para cada carácter.
2. Si es un operando, pasarlo a la expresión postfija.
3. Si es operador:
   3.1. Si la pila está vacía, meterlo en la pila. Repetir a partir de 1.
   3.2. Si la pila no está vacía:
   Si prioridad del operador es mayor que prioridad del operador cima, meterlo en la pila y repetir a partir de 1.
   Si prioridad del operador es menor o igual que prioridad del operador cima, sacar operador cima de la pila y ponerlo en la expresión postfija, volver a 3.
4. Si es paréntesis derecho:
   4.1. Sacar operador cima y ponerlo en la expresión postfija.
   4.2. Si el nuevo operador cima es paréntesis izquierdo, suprimir elemento cima.
   4.3. Si cima no es paréntesis izquierdo, volver a 4.1.
   4.4. Volver a partir de 1.
5. Si quedan elementos en la pila pasarlos a la expresión postfija.
6. Fin del algoritmo.

Codificación del algoritmo de transformación a *postfija*

Se necesita crear una pila de caracteres para guardar los operadores. Se utiliza el diseño de Pila genérica del Apartado 11.3. La expresión original se lee del teclado en una cadena de suficiente tamaño. También se declara una estructura para representar un elemento de la ex-

presión, con un campo carácter para el operando o el operador, y el otro campo para indicar si es operador u operando.

```
struct Elemento
{
  char c;
  bool oprdor;
};
```

La función `postFija()` implementa los pasos del algoritmo de transformación, recibe como argumento una cadena con la expesión, crea una pila y en un bucle de tantas iteraciones como caracteres realiza las acciones del algoritmo. La función define el array `Elemento* expresión`, a la que se pasan los elementos que forman la expresión en postfija. Una vez que termina la transformación, la función devuelve la expresión en *notación postfija* y el número de elementos de que consta agrupados en la siguiente estructura:

```
struct Expresion
{
  Elemento* expr;
  int n;
};
```

El archivo `TiposExpresio.h` contiene el tipo `Elemento` y el tipo `Expresion`.

En la función `prdadDentro()`se fija la prioridad de un operador dentro de la pila, y en `prdadFuera()`la prioridad de un operador fuera de la pila.

```
//archivo postFija.cpp

#include <cstdlib>
#include <string.h>
#include <ctype.h>
#include "PilaGenerica.h"
#include "TiposExpresio.h"

Expresion postFija(const char* expOrg);
int prdadFuera (char op);
int prdadDentro (char op);
bool valido(const char* expresion);
bool operando(char c);

Expresion postFija(const char* expOrg)
{
  PilaGenerica<char> pila;
  Elemento* expsion;
  bool desapila;
  int n = -1;   //contador de expresión en postfija

  if (! valido(expOrg))  // verifica los caracteres de la expresión
     throw "Carácter no válido en una expresión";

  expsion = new Elemento[strlen(expOrg)];
  for (int i = 0; i < strlen(expOrg); i++)
  {
    char ch, opeCima;
    ch = toupper(expOrg[i]);    // operandos en mayúsculas
```

```
    if (operando(ch))           // análisis del elemento
    {
      expsion[++n].c = ch;
      expsion[n].oprdor = false;
    }
    else if (ch != ')')         // es un operador
    {
      desapila = true;
      while (desapila)
      {
        opeCima = ' ';
        if (!pila.pilaVacia())
          opeCima = pila.cimaPila();
          if (pila.pilaVacia() ||
            (prdadFuera(ch) > prdadDentro(opeCima)))
          {
            pila.insertar(ch);
            desapila = false;
          }
          else if (prdadFuera(ch) <= prdadDentro(opeCima))
          {
            expsion[++n].c = pila.quitar();
            expsion[n].oprdor = true;
          }
        }
    }
    else                        // es un ')'
    {
      opeCima = pila.quitar();
      do{
        expsion[++n].c = opeCima;
        expsion[n].oprdor = true;
        opeCima = pila.quitar();
      }while (opeCima != '(');
    }
  }
    /*
       se vuelca los operadores que quedan en la pila y se pasan a la ex-
       presión.
    */
  while (!pila.pilaVacia())
  {
    expsion[++n].c = pila.quitar();
    expsion[n].oprdor = true;
  }
  Expresion post;
  post.expr = expsion;
  post.n = n;
  return post;   // expresión en postfija
}
// prioridad del operador dentro de la pila

int prdadDentro(char op)
{
  int pdp;
  switch (op)
```

```
  {
    case '^':  pdp = 3;
               break;
    case '*': case '/':
               pdp = 2;
               break;
    case '+': case '-':
               pdp = 1;
               break;
    case '(':  pdp = 0;
  }
  return pdp;
}
// prioridad del operador en la expresión infija
int prdadFuera(char op)
{
  int pfp;
  switch (op)
  {
    case '^':  pfp = 4;
               break;
    case '*': case '/':
               pfp = 2;
               break;
    case '+': case '-':
               pfp = 1;
               break;
    case '(':  pfp = 5;
  }
  return pfp;
}

//analiza cada carácter de la expresión
bool valido(const char* expresion)
{
  bool sw = true;
  for (int i = 0; (i < strlen(expresion))&& sw; i++)
  {
    char c;
    c = expresion[i];
    sw = sw && (
                (c >= 'A' && c <= 'Z') ||
                (c >= 'a' && c <= 'z') ||
                (c == '^' || c == '/' || c== '*' ||
                  c == '+' || c == '-' || c== '\n' ||
                  c == '(' || c == ')' )
               );
  }
  return sw;
}

bool operando(char c)
{

//determina si c es un operando
  return (c >= 'A' && c <= 'Z');
}
```

11.4.4. Evaluación de la expresión en notación *postfija*

Una vez que se tiene la expresión en *notación postfija* se evalúa la expresión. Evaluar significa obtener un resultado de la expresión para valores particulares de los operandos. De nuevo, el algoritmo de evaluación utiliza una pila, en esta ocasión de operandos, es decir, de números reales.

Al describir el algoritmo `expsion` es el array con la la expresión postfija. El número de elementos es la longitud, `n`, de la cadena.

1. Examinar `expsion` elemento a elemento: repetir los pasos 2 y 3 para cada elemento.
2. Si el elemento es un operando, meterlo en la pila.
3. Si el elemento es un operador, se simboliza con `&`, entonces:
   - Sacar los dos elementos superiores de la pila, se denominarán `b` y `a` respectivamente.
   - Evaluar `a & b`, el resultado es `z = a & b`.
   - El resultado `z`, meterlo en la pila. Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento cima de la pila.
5. Fin del algoritmo.

Codificación de la evaluación de expresión en postfija

La función `double evalua()` implementa el algoritmo, recibe la expresión en postfija y el array con el valor de cada operando. La pila de números reales, utilizada por el algoritmo, se instancia de la clase genérica `PilaGenerica` para elementos de tipo `double`.

La función `valorOprdos()` da entrada a los valores de los operandos. Estos valores se guardan en un array, cada posición del array se corresponde con una letra, que a su vez es un operando.

```
// archivo evaluaExpresion.h

#include "pilagenerica.h"
#include "TiposExpresio.h"
#include <math.h>

double evalua(Expresion postFija, double v[]);
void valorOprdos(Expresion ep, double v[]);

double evalua(Expresion postFija, double v[])
{
  PilaGenerica<double> pila;
  double valor, a, b;

  for (int i = 0; i <= postFija.n; i++)
  {
    char op;

    if (postFija.expr[i].oprdor)     // es un operador
    {
      op = postFija.expr[i].c;
```

```
        b = pila.quitar();
        a = pila.quitar();
        switch (op)
        {
          case '^': valor = pow(a,b);
                    break;
          case '*': valor = a * b;
                    break;
          case '/': if (b != 0.0)
                      valor = a / b;
                    else
                      throw "División por cero.";
                    break;
          case '+': valor = a + b;
                    break;
          case '-': valor = a – b;
        }
        pila.insertar(valor);
      }
      else               // es un operando
      {
        int indice;
        op = postFija.expr[i].c;
        indice = op - 'A';       // posición en array de valores
        pila.insertar(v[indice]);
      }
    }
    return pila.quitar();        // resultado de la expresión
  }

  // asignan valores numéricos a los operandos
  void valorOprdos(Expresion ep, double v[])
  {
    char ch;
    for (int i = 0; i <= ep.n; i++)
    {
      char op;
      op = ep.expr[i].c;
      if (! ep.expr[i].oprdor)   // es un operando
      {
        int indice;
        double d;
        indice = op - 'A';
        cout << op << " = ";
        cin >> v[indice];
      }
    }
  }
```

Programa

La función `main()` controla las etapas principales del algoritmo: petición de la expresión algebraica, llamada a la función que transforma a notación postfija y, por último, evaluar la expresión para unos valores concretos de los operandos.

```
#include <iostream>
using namespace std;
#include "TiposExpresio.h"

Expresion postFija(const char* expOrg);
double evalua(Expresion postFija, double v[]);
void valorOprdos(Expresion ep, double v[]);

int main()
{
  double v[26];
  double resultado;
  char expresion[81];
  Expresion ex;

  cout << "\nExpresión aritmética: ";
  cin.getline(expresion, 80);
  // Conversión de infija a postfija
  expr = postFija(expresion);
  cout << "\nExpresión en postfija: ";
  for (int i = 0; i <= ex.n; i++)
  {
    cout << ex.expr[i];
  }
  // Evaluación de la expresión
  valorOprdos(ex, v);       // valor de operandos
  resultado = evalua(ex, v);
  cout << "Resultado = " << resultado;
  return 0;
}
```

RESUMEN

Una *pila* es una estructura de datos tipo **LIFO** (*last in first out*, último en entrar primero en salir) en la que los datos (todos del mismo tipo) se añaden y eliminan por el mismo extremo, denominado *cima* de la pila. Se definen las siguientes operaciones básicas sobre pilas: `crear`, `insertar`, `cima`, `quitar`, `pilaVacia`, `pilaLlena` y `limpiarPila`.

`insertar`, añade un elemento en la cima de la pila. Debe de haber espacio en la pila.
`cima`, devuelve el elemento que está en la cima, sin extraerlo.
`quitar`, extrae de la pila el elemento cima de la pila.
`pilaVacia`, determina si el estado de la pila es vacía.
`pilaLlena`, determina si existe espacio en la pila para añadir un nuevo elemento.
`limpiarPila`, el espacio asignado a la pila se libera, queda disponible.

Las aplicaciones de las pilas en la programación son numerosas, entre las que está la evaluación de expresiones aritméticas. Primero, se transforma la expresión a notación postfija, a continuación se evalúa.

Las expresiones en notación polaca, postfija o prefija, tienen la característica de que no necesitan paréntesis.

EJERCICIOS

11.1. ¿ Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es `int`:

```
Pila p;
int x = 4, y;
p.insertar(x);
cout << "\n " << p.cimaPila();
y = p.quitar();
p.insertar(32);
p.insertar(p.quitar());
do {
   cout << "\n " << p.quitar();
}while (!p.pilaVacia());
```

11.2. Con las operaciones implementadas en la clase `PilaGenerica` escribir la función `mostrarPila()` que muestre en pantalla los elementos de una pila de cadenas.

11.3. Utilizando una pila de caracteres, transformar la siguiente expresión a su equivalente expresión en postfija.

```
(x-y)/(z+w) – (z+y)^x
```

11.4. Obtener una secuencia de 10 números reales, guardarlos en un array y ponerlos en una pila. Imprimir la secuencia original y, a continuación, imprimir la pila extrayendo los elementos.

11.5. Transformar la expresión algebraica del Ejercicio 11.3 en su equivalente expresión en *notación prefija*.

11.6. Dada la expresión algebraica `r = x * y - (z + w)/(z + y) ^ x`, transformar la expresión a *notación postfija* y, a continuación, evaluar la expresión para los valores: `x = 2, y = -1, z = 3, w = 1`. Obtener el resultado de la evaluación siguiendo los pasos del algoritmo descrito en el Apartado 11.4.3.

11.7. Se tiene una lista enlazada a la cual se accede por el primer nodo. Escribir una función que visualice los nodos de la lista en orden inverso, desde el último nodo al primero; como estructura auxiliar utilizar una pila y sus operaciones.

11.8. La implementación del *TAD* `Pila` con arrays establece un tamaño máximo de la pila que se controla con la función `pilaLlena()`. Modificar la función de tal forma que cuando se llene la pila se amplíe el tamaño del array a justamente el doble de la capacidad actual.

PROBLEMAS

11.1. Escribir una función, `copiarPila()`, que copie el contenido de una pila en otra. La función tendrá dos argumentos de tipo pila, uno la pila fuente y otro la pila destino. Utilizar las operaciones definidas sobre el *TAD* `Pila`.

11.2. Escribir una función para determinar si una secuencia de caracteres de entrada es de la forma:

```
X & Y
```

donde `X` una cadena de caracteres e `Y` la cadena inversa. El carácter `&` es el separador.

11.3. Escribir un programa que haciendo uso de una `Pila`, procese cada uno de los caracteres de una expresión que viene dada en una línea. La finalidad es verificar el equilibrio de paréntesis, llaves y corchetes. Por ejemplo, la siguiente expresión tiene un número de paréntesis equilibrado:

```
((a+b)*5) - 7
```

y esta otra expresión le falta un corchete: `2*[(a+b)/2.5+x -7*y`

11.4. Escribir un programa en el que se manejen un total de n = 5 pilas: P_1, P_2, P_3, P_4 y P_5. La entrada de datos será pares de enteros `(i,j)` tal que `1 ≤ abs(i) ≤ n`. De tal forma que el criterio de selección de pila:

- Si i es positivo, debe de insertarse el elemento j en la pila P_i .
- Si i es negativo, debe de eliminarse el elemento j de la pila P_i .
- Si i es cero, fin del proceso de entrada.

Los datos de entrada se introducen por teclado. Cuando termina el proceso el programa debe escribir el contenido de las n pilas en pantalla.

11.5. Modificar el Programa 11.4 para que la entrada sean triplas de números enteros `(i,j,k)`, donde `i`, `j` tienen el mismo significado que en 11.4, y `k` es un número entero que puede tomar los valores `–1`, `0` con este significado:

- `-1`, hay que borrar todos los elementos de la pila.
- `0`, el proceso es el indicado en 11.4 con `i` y `j`.

11.6. Se quiere determinar frases que son palíndromo. Para lo cual se ha de seguir la siguiente estrategia: considerar cada línea de una frase; añadir cada carácter de la frase a una pila y a la vez a lista enlazada circular por el final; extraer carácter a carácter, simultáneamente de la pila y de la lista circular el considerado *primero* y su comparación determina si es palíndromo o no. Escribir un programa que lea líneas de y determine si son palíndromo.

11.7. La función de Ackerman, definida de la siguiente forma:

A(m, n) = n + 1	*si m = 0*
A(m, n) = A (m – 1, 1)	*si n = 0*
A(m, n) = A(m – 1, A(m, n – 1))	*si m > 0, y n > 0*

Se obseva que la definición es recursiva y, por consiguiente, la implementación recursiva es inmediata de codificar. Como alternativa, escribir una función que evalúe la función Ackermannç iterativamente utilizando el *TAD Pila*.

Capítulo 12
Colas

Objetivos

Con el estudio de este capítulo usted podrá:

- Especificar el tipo abstracto de datos *Cola*.
- Encontrar las diferencias fundamentales entre *Pilas* y *Colas.*
- Definir una clase *Cola* con arrays.
- Definir una clase *Cola* con listas enlazadas.
- Aplicar el tipo abstracto *Cola* para la resolución de problemas.

Contenido

Conceptos clave

- Array *circular*.
- Cola de objetos.
- Lista enlazada.
- Lista FIFO.
- Prioridad.

INTRODUCCIÓN

En este capítulo se estudia el tipo abstracto de datos *Cola*, estructura muy utilizada en la vida cotidiana, y también para resolver problemas en programación. Esta estructura, al igual que las pilas, almacena y recupera sus elementos atendiendo a un estricto orden. Las colas se conocen como estructuras **FIFO** (*First-in*, *First-out*, primero en entrar - primero en salir), debido a la forma y orden de inserción y de extracción de elementos de la cola. Las colas tienen numerosas aplicaciones en el mundo de la computación: colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades.

12.1. CONCEPTO DE COLA

Una **cola** es una estructura de datos que almacena elementos en una lista y el acceso a los datos se hace por uno de los dos extremos de la lista. (Figura 12.1). Un elemento se inserta en la cola (parte *final*) de la lista y se suprime o elimina por el frente (parte inicial, *frente*) de la lista. Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia.

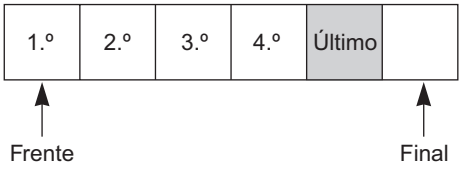


Figura 12.1. Una cola.

Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacenan y, por consiguiente, una cola es una estructura de tipo **FIFO** *(first-in/firs-out, primero en entrar/primero en salir* o bien *primero en llegar/primero en ser servido).* El servicio de atención a clientes en un almacén es un típico ejemplo de cola. La acción de gestión de memoria intermedia *(buffering)* de trabajos o tareas de impresora en un distribuidor de impresoras *(spooler)* es otro ejemplo de cola[1]. Dado que la impresión es una tarea (un trabajo) que requiere más tiempo que el proceso de la transmisión real de los datos desde la computadora a la impresora, se organiza una cola de trabajos de modo que los trabajos se imprimen en el mismo orden en que se recibieron por la impresora. Este sistema tiene el gran inconveniente de que si su trabajo personal consta de una única página para imprimir y delante de su petición de impresión existe otra petición para imprimir un informe de 300 páginas, deberá esperar a la impresión de esas 300 páginas antes de que se imprima su página.

Definición

Una cola es una estr uctura de datos cuy os elementos mantienen un cier to orden, tal que sólo se pueden añadir elementos por un extremo, **final** de la cola, y eliminar o extraer por el otro extremo, llamado **frente**.

[1] Recordemos que este caso sucede en sistemas multiusuario donde hay varios terminales y sólo una impresora de servicio. Los trabajos se "encolan" en la cola de impresión.

Las operaciones usuales en las colas son `Insertar` y `Quitar`. La operación `Insertar` añade un elemento por el extremo `final` de la cola, y la operación `Quitar` elimina o extrae un elemento por el extremo opuesto, el frente o primero de la cola. La organización de elementos en forma de cola asegura que *el primero en entrar es el primero en salir*. En la Figura 12.2 se realizan las operaciones básicas sobre colas, insertar y retirar elementos.

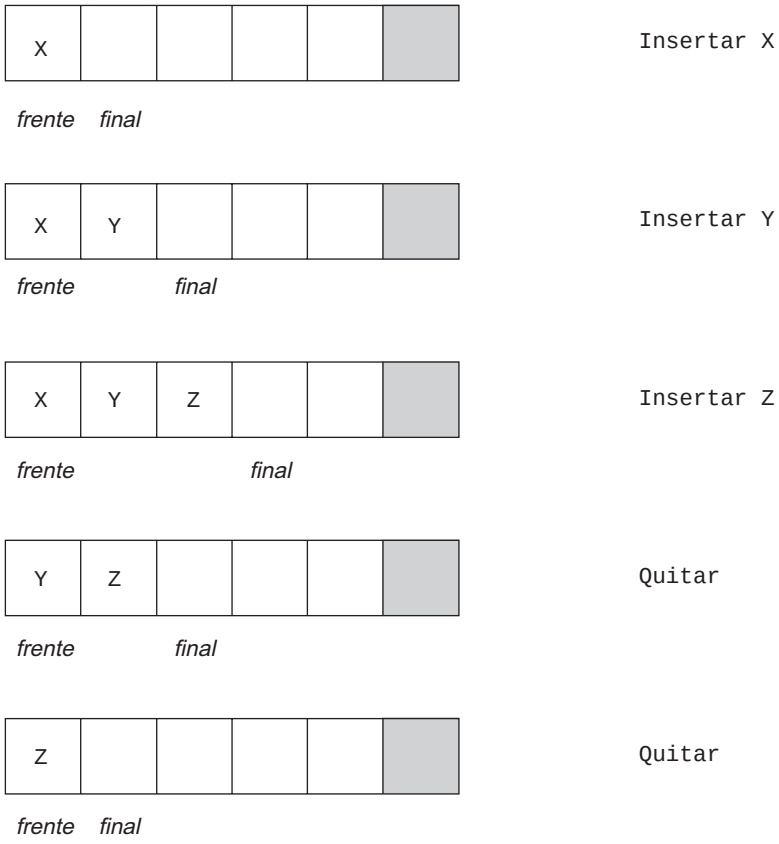


Figura 12.2. Oper aciones `Insertar` y `Quitar` en una `Cola`.

12.1.1. Especificaciones del tipo abstracto de datos `Cola`

Las operaciones que definen la estructura de una cola son las siguientes:

Tipo de dato	Elemento que se almacena en la cola.
Operaciones	
CrearCola	Inicia la cola como vacía.
Insertar	Añade un elemento por el final de la cola.
Quitar	Retira (extrae) el elemento frente de la cola.

Cola vacía	Comprobar si la cola no tiene elementos.
Cola llena	Comprobar si la cola está llena de elementos.
Frente	Obtiene el elemento frente o primero de la cola.
Tamaño de la cola	Número de elementos máximo que puede contener la cola.

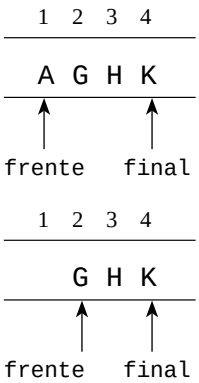
Desde el punto de vista de estructura de datos, una cola es similar a una pila, en cuanto que los datos se almacenan de modo lineal y el acceso a los datos sólo está permitido en los extremos de la cola.

La forma que los lenguajes tienen para representar el *TAD Cola* depende de donde se almacenen los elementos, en un array, en una estructura dinámica como puede ser una lista enlazada. La utilización de arrays tiene el problema de que la *cola* no puede crecer indefinidamente, está limitada por el tamaño del array, como contrapartida el acceso a los extremos es muy eficiente. Utilizar una lista dinámica permite que el número de nodos se ajuste al de elementos de la cola, por el contrario cada nodo necesita memoria extra para el enlace y también está el limite de memoria de la pila del computador.

12.2. COLAS IMPLEMENTADAS CON ARRAYS

Al igual que las pilas, las colas se implementan utilizando una estructura estática (arrays), o una estructura dinámica (listas enlazadas). La implementación estática se realiza declarando un array para almacenar los elementos, y dos marcadores o apuntadores para mantener las posiciones `frente` y `final` de la cola; es decir, un marcador apuntando a la posición de la *cabeza* de la cola y el otro al primer espacio vacío que sigue al *final* de la cola. Cuando un elemento se añade a la cola, se verifica si el marcador `final` apunta a una posición válida, entonces se asigna el elemento en esa posición y se incrementa el marcador `final` en 1. Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador (puntero) de cabeza y éste se incrementa en 1.

La operación de poner un elemento inserta por el extremo *final*. La primera asignación se realiza en la posición `final = 0`, cada vez que se añade un nuevo elemento se incrementa `final` en 1 y se asigna el elemento. La extracción de un elemento se hace por el extremo contrario, `frente`, cada vez que se extrae un elemento avanza `frente` una posición. La Figura 12.3 muestra el avance del puntero `frente` al extraer un elemento.



posición de `frente` y `final` después de extraer.

Figura 12.3. Una cola representada en un array.

El avance lineal de `frente` y `final` tiene un grave problema, deja *huecos* por la *izquierda del array*. Llegando a ocurrir que `final` alcance el índice más alto del array, no pudiéndose añadir nuevos elementos y, sin embargo, haya posiciones libres a la izquierda de `frente`.

Una alternativa que evita el problema de dejar *huecos*, consiste en mantener fijo el `frente` de la cola al comienzo del array, y mover todos los elementos de la cola una posición cada vez que se retira un elemento de la cola. Otra alternativa, mucho más eficiente, es considerar el array como una estructura *circular*.

12.2.1. Clase Cola

Los elementos de una cola pueden ser de cualquier tipo de dato: entero, cadena, objetos ...; por esa razón, se abstrae el tipo con la sentencia `typedef`, para que pueda sustituirse por cualquier tipo simple, posteriormente se implementa una *Cola Genérica* con *plantillas* (`template`).

La clase `ColaLineal` declara un array (`listaCola`) cuyo tamaño se determina por la constante `MAXTAMQ`. Las variables `frente` y `final` son los apuntadores a cabecera y cola, respectivamente. El constructor de la clase inicializa la estructura, de tal forma que se parte de una cola vacía.

Las operaciones básicas del tipo abstracto de datos cola: `insertar`, `quitar`, `colaVacia`, `colaLlena`, *y* `frente` se implementan en la clase. `insertar` toma un elemento y lo añade por el final. `quitar` suprime y devuelve el elemento *cabeza* de la cola. La operación `frente` devuelve el elemento que está en la primera posición (`frente`) de la cola, sin eliminar el elemento.

La operación de control, `colaVacia` comprueba si la cola tiene elementos, esta comprobación es necesaria antes de eliminar un elemento; `colaLlena` comprueba si se pueden añadir nuevos elementos, esta comprobación se realiza antes de insertar un nuevo miembro. Si las precondiciones para `insertar` y `quitar` se violan, el programa debe generar una excepción o error.

```
// archivo de cabecera ColaLineal.h

typedef tipo TipoDeDato;        // tipo ha de ser conocido
const int MAXTAMQ = 39;

class ColaLineal
{
protected:
     int frente;
     int final;
     TipoDeDato listaCola[MAXTAMQ];
public:
     ColaLineal()
     {
       frente = 0;
       final = -1;
     }
     // operaciones de modificación de la cola
     void insertar(const TipoDeDato& elemento)
```

```
        {
          if (!colaLlena())
          {
            listaCola[++final] = elemento;
          }
          else
            throw "Overflow en la cola";
        }
        TipoDeDato quitar()
        {
          if (!colaVacia())
          {
            return listaCola[frente++];
          }
          else
            throw "Cola vacia ";
        }
        void borrarCola()
        {
          frente = 0;
          final = -1;
        }
          // acceso a la cola
        TipoDeDato frenteCola()
        {
          if (!colaVacia())
          {
            return listaCola[frente];
          }
          else
            throw "Cola vacia ";
        }
          // métodos de verificación del estado de la cola
        bool colaVacia()
        {
          return frente > final;
        }
        bool colaLlena()
        {
          return final == MAXTAMQ - 1;
        }
};
```

Esta implementación de una cola es notablemente ineficiente, se puede alcanzar la condición de cola llena habiendo posiciones del array sin ocupar. Esto es debido a que al realizar la operación *quitar* avanza el `frente`, y, por consiguiente, las posiciones anteriores quedan desocupadas, no accesibles. Una solución a este problema consiste en que al retirar un elemento, `frente` no se incremente y se desplace el resto de elementos una posición a la izquierda.

Recodar

La realización de una cola con un arr ay lineal es notab lemente ineficiente, se puede alcanzar la condición de cola llena habiendo elementos del arr ay sin ocupar.

12.3. COLA CON UN ARRAY CIRCULAR

La alternativa, sugerida en la operación *quitar* un elemento, de desplazar los restantes elementos del array de modo que la *cabeza* de la cola vuelva al principio del array, es costosa, en términos de tiempo de computadora, especialmente si los datos almacenados en el array son estructuras de datos grandes.

La forma más eficiente de almacenar una cola en un array es modelar éste de tal forma que se una el extremo final con el extremo cabeza. Tal array se denomina *array circular* y permite que la totalidad de sus posiciones se utilicen para almacenar elementos de la cola sin necesidad de desplazar elementos. La Figura 12.4 muestra un *array circular* de *n* elementos.

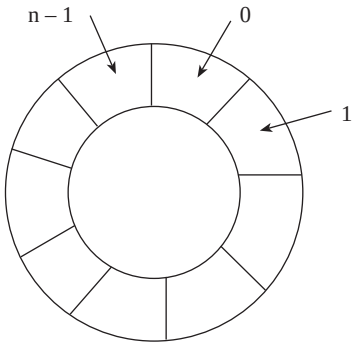


Figura 12.4. Un array circular.

El array se almacena de modo natural en la memoria, como un bloque lineal de *n* elementos. Se necesitan dos marcadores (apuntadores) *frente* y *final* para indicar, respectivamente, la posición del elemento *cabeza* y la posición donde se almacenó el último elemento puesto en la cola.

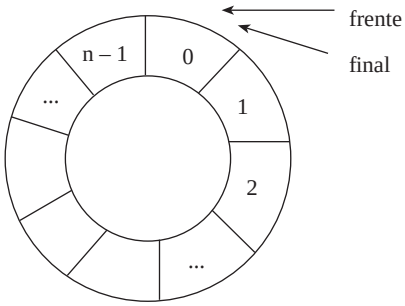


Figura 12.5. Una cola vacía.

El apuntador `frente` siempre contiene la posición del primer elemento de la cola y avanza en el sentido de las agujas del reloj; `final` contiene la posición donde se puso el último elemento, también avanza en el sentido del reloj (circularmente a la derecha). La implementación del movimiento circular se realiza segun la *teoría de los restos,* de tal forma que se generen índices de `0` a `MAXTAMQ - 1`:

Mover `final` adelante = `(final + 1) % MAXTAMQ`
Mover `frente` adelante = `(frente + 1) % MAXTAMQ`

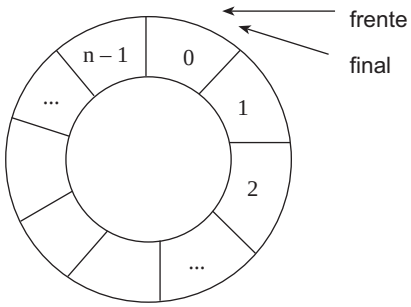


Figura 12.6. Una cola que contiene un elemento.

La implementación de la gestión de colas con un *array circular* ha de incluir las operaciones básicas del *TAD Cola*, en decir, las siguientes tareas básicas:

- Creación de una cola vacía, de tal forma que `final` apunte a una posición inmediatamente anterior a `frente`:

```
frente = 0; final = MAXTAMQ - 1.
```

- Comprobar si una cola está vacía:

```
frente == siguiente(final)
```

- Comprobar si una cola está llena. Para diferenciar la condición *cola llena* de *cola vacía* se sacrifica una posición del array , entonces la capacidad real de la cola será ser `MAXTAMQ-1`. La condición de cola llena:

```
frente == siguiente(siguiente(final))
```

- Poner un elemento a la cola: si la cola no está llena, fijar `final` a la siguiente posición: `final = (final + 1) % MAXTAMQ` y asignar el elemento.
- Retirar un elemento de la cola: si la cola no está vacía, quitarlo de la posición `frente` y establecer `frente` a la siguiente posición: `(frente + 1) % MAXTAMQ.`
- Obtener el elemento primero de la cola, si la cola no está vacía, sin suprimirlo de la cola.

12.3.1. Clase `Cola` con array circular

La representación de los elementos de la cola no cambia, un array lineal y dos índices: `listaCola[], frente, final.` Las operaciones tienen la misma interfaz que `ColaLineal`, entonces para aprovechar la potencia de la orientación a objetos, la nueva clase deriva de

`ColaLineal` y *redefine* las funciones de la *interfaz*. Además, se escribe la función auxiliar `siguiente()` para obtener la *siguiente* posición de una dada, aplicando la *teoría de los restos*.

A continuación, se codifica los métodos que implementan las operaciones del *TAD* `Cola`.

```
// archivo ColaCircular.h

#include "ColaLineal.h"
class ColaCircular : public ColaLineal
{

protected:
    int siguiente(int r)
    {
      return (r+1) % MAXTAMQ;
    }
    //Constructor, inicializa a cola vacía
public:
    ColaCircular()
    {
      frente = 0;
      final = MAXTAMQ-1;
    }
    // operaciones de modificación de la cola
    void insertar(const TipoDeDato& elemento);
    TipoDeDato quitar();
    void borrarCola();
    // acceso a la cola
    TipoDeDato frenteCola();
    // métodos de verificación del estado de la cola
    bool colaVacia();
    bool colaLlena();
};
```

Implementación

```
void ColaCircular :: insertar(const TipoDeDato& elemento)
{
  if (!colaLlena())
  {
    final = siguiente(final);
    listaCola[final] = elemento;
  }
  else
    throw "Overflow en la cola";
}

TipoDeDato ColaCircular :: quitar()
{
  if (!colaVacia())
  {
    TipoDeDato tm = listaCola[frente];
```

```
    frente = siguiente(frente);
    return tm;
  }
  else
    throw "Cola vacia ";
}

void ColaCircular :: borrarCola()
{
  frente = 0;
  final = MAXTAMQ-1;
}

TipoDeDato ColaCircular :: frenteCola()
{
  if (!colaVacia())
  {
    return listaCola[frente];
  }
  else
    throw "Cola vacia ";
}

bool ColaCircular :: colaVacia()
{
  return frente == siguiente(final);
}

bool ColaCircular :: colaLlena()
{
  return frente == siguiente(siguiente(final));
}
```

EJEMPLO 12.1. Se desea decidir si un número leído del dispositiv o estándar de entrada es capicúa.

El algoritmo para determinar si un número es capicúa utiliza conjuntamente una *Cola* y una *Pila*. El número se lee del teclado en forma de cadena de dígitos. La cadena se procesa carácter a carácter, es decir, dígito a dígito (un dígito es un carácter del '0' al '9'). Cada dígito se pone en la cola y a la vez en la pila. Una vez que se termina de leer los dígitos y de ponerlos en la cola y en la pila, comienza la comprobación: se extraen consecutivamente elementos de la cola y de la pila, y se comparan por igualdad, de producirse alguna no coincidencia entre dígitos es que el número no es capicúa y entonces se vacían las estructuras. El número es capicúa si el proceso de comprobación termina habiendo coincidido todos los dígitos en orden inverso, lo cual equivale a que la pila y la cola terminen vacías.

¿Por qué utilizar una pila y una cola?, sencillamente para asegurar que se procesan los dígitos en orden inverso; en la pila el *último en entrar es el primero en salir*, en la cola el *primero en entrar es el primero en salir*.

La pila que se implementa es de tipo `PilaGenérica` y la cola de la clase `ColaCircular` implementada con un *array circular*.

```
#include <iostream>
using namespace std;
```

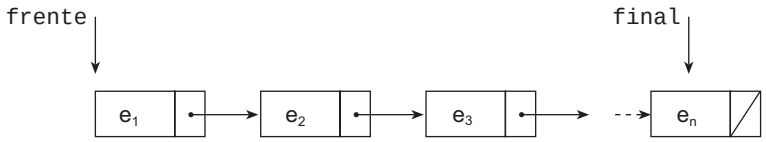
```
#include <string.h>
#include "PilaGenerica.h"
typedef char TipoDeDato;
#include "ColaCircular.h"

bool valido(const char* numero);

int main()
{
  bool capicua;
  char numero[81];
  PilaGenerica<char> pila;
  ColaCircular q;

  capicua = false;
  while (!capicua)
  {
    do {
      cout << "\nTeclea el número: ";
      cin.getline(numero,80);
    }while (!valido(numero)); // todos los caracteres dígitos
    // pone en la cola y en la pila cada dígito
    for (int i = 0; i < strlen(numero); i++)
    {
      char c;
      c = numero[i];
      q.insertar(c);
      pila.insertar(c);
    }
    // se retira de la cola y la pila para comparar
    do {
      char d;
      d = q.quitar();
      capicua = d == pila.quitar();    //compara por igualdad
    } while (capicua && !q.colaVacia());

    if (capicua)
      cout << numero << " es capicúa " << endl;
    else
    {
      cout << numero << " no es capicúa, ";
      cout << " intente con otro. ";
     // se vacía la cola y la pila
      q.borrarCola();
      pila.limpiarPila();
    }
  }
  return 0;
}

    // verifica que cada carácter es dígito
bool valido(const char* numero)
{
  bool sw = true;
  int i = -1;
```

```
    while (sw && (i < strlen(numero)))
    {
      char c;
      c = numero[++i];
      sw = (c >= '0' && c <= '9');
    }
    return sw;
  }
```

12.4. COLA GENÉRICA CON UNA LISTA ENLAZADA

La implementación del *TAD* `Cola` con independencia del tipo de dato de los elementos se consigue utilizando las *plantillas* (`template`) de C++. Además, se va a utilizar la estructura dinámica lista enlazada para que, en todo momento, el número de elementos de la cola se ajuste al número de nodos de la lista, de tal forma que pueda crecer o disminuir sin problemas de espacio.

La implementación del *TAD* `Cola` con una lista enlazada utiliza dos punteros de acceso a la lista: `frente` y `final`. Son los extremos por donde salen y por donde se ponen, respectivamente, los elementos de la cola.



e_1, e_2, ...en son valores del tipo *TipoDeDato*

Figura 12.9. Cola con lista enlazada (representación gráfica típica).

El puntero `frente` apunta al primer elemento de la lista y, por tanto, de la cola (el primero en ser retirado), `final` apunta al último elemento de la lista y también de la cola.

La lista enlazada crece y decrece según las necesidades, según se incorporen elementos o se retiren, y por esta razón en esta implementación no se considera la función de control de `Colallena`.

12.4.1. Clase genérica `Cola`

La implementación de una *cola genérica* se realiza con dos clases: clase `ColaGenerica` y clase `Nodo` que será una clase anidada. El `Nodo` representa al elemento y al enlace con el siguiente nodo; al crear un `Nodo` se asigna el elemento y el enlace se pone `NULL`.

La clase `ColaGenerica` define las variables de acceso: `frente` y `final`, y las operaciones básicas del *TAD* `Cola`. Su constructor inicializa `frente` y `final` a `NULL`, es decir, a la condición *cola vacía*.

```
  // archivo ColaGenerica.h
template <class T>
class ColaGenerica
```

```
{
protected:
    class NodoCola
    {
    public:
         NodoCola* siguiente;
         T elemento;
         NodoCola (T x)
         {
            elemento = x;
            siguiente = NULL;
         }
    };
    NodoCola* frente;
    NodoCola* final;

public:
    ColaGenerica()
    {
      frente = final = NULL;
    }
    void insertar(T elemento);
    T quitar();
    void borrarCola();
    T frenteCola()const;
    bool colaVacia() const;
    ~ColaGenerica()
    {
      borrarCola ();
    }
};
```

12.4.2. Implementación de las operaciones de cola genérica

Las operaciones acceder directamente a la lista; `insertar` crea un nodo y lo enlaza por el `final`, `quitar` devuelve el dato del nodo `frente` y lo borra de la lista, `frenteCola` accede al `frente` para obtener el elemento.

Añadir un elemento a la cola

```
template <class T>
void ColaGenerica<T> :: insertar(T elemento)
{
  NodoCola* nuevo;
  nuevo = new NodoCola (elemento);
  if (colaVacia())
  {
    frente = nuevo;
  }
  else
  {
    final -> siguiente = nuevo;
  }
  final = nuevo;
}
```

Sacar de la cola

Devuelve el elemento frente y lo quita de la cola, disminuye el tamaño de la cola.

```
template <class T>
T ColaGenerica<T> :: quitar()
{
  if (colaVacia())
    throw "Cola vacía, no se puede extraer.";
  T aux = frente -> elemento;
  NodoCola* a = frente;
  frente = frente -> siguiente;
  delete a;
  return aux;
}
```

Elemento frente de la cola

```
template <class T>
T ColaGenerica<T> :: frenteCola()const
{
  if (colaVacia())
    throw "Cola vacía";
  return frente -> elemento;
}
```

Vaciado de la cola

Elimina todos los elementos de la cola. Recorre la lista desde `frente` a `final`, es una operación de complejidad lineal, `O(n)`.

```
template <class T>
void ColaGenerica<T> :: borrarCola()
{
  for (;frente != NULL;)
  {
      NodoCola* a;
      a = frente;
      frente = frente -> siguiente;
      delete   a;
  }
  final = NULL;
}
```

Verificación del estado de la cola

```
template <class T>
bool ColaGenerica<T> :: colaVacia() const
{
  return frente == NULL;
}
```

EJERCICIO 12.1. Se quiere generar números de la suerte aplicando una variación al llamado "problema de José". El punto de partida es una lista de n números, esta lista se va reduciendo siguiendo el siguiente algoritmo:

1. *Se genera un número aleatorio* `n1`.
2. *Si* `n1 <= n` *se quitan de la lista los números que ocupan las posiciones* `1, 1 + n1, 1 + 2 * n1, ...; n` *toma el valor del número de elementos que quedan en la lista.*
3. *Se vuelve al paso 1.*
4. *Si* `n1 > n` *fin del algoritmo, los números de la suerte son los que quedan en la lista.*

El problema se va a resolver utilizando una *cola*. En primer lugar, se genera la lista de `n` números aleatorios que se almacenan en la cola. A continuación, se siguen los pasos del algoritmo, en cada pasada se eliminan los elementos de la cola que están en las posiciones *`(múltiplos de n1) + 1`*. Estas posiciones, denominadas `i`, se pueden expresar matemáticamente:

```
i modulo n1 == 1
```

Una vez que termina el algoritmo, los números de la suerte son los que han quedado en la cola, entonces se retiran de la cola y se escriben.

Se utiliza una *cola genérica*, implementada con listas enlazadas. Al crear la cola se pasa el argumento `int` que, en este ejercicio, es el tipo de dato de los elementos.

```
#include <iostream>
using namespace std;
#include <time.h>
#include "ColaGenerica.h"

const int N = 99;
#define randomize (srand(time(NULL)))
#define random(num) (rand()%(num))

template <class T>
void mostrarCola(ColaGenerica<T>& q);

int main()
{
  int n, n1, n2, i;
  ColaGenerica<int> q;

  randomize;
  // número inicial de elementos de la lista
  n = 11 + random(N);
  // se generan n números aleatorios
  for (int i = 1; i <= n; i++)
  {
    q.insertar(random(N * 3));
  }
```

```
  // se genera aleatoriamente el intervalo n1
  n1 = 1 + random(11);
  // se retiran de la cola elementos a distancia n1
  while (n1 <= n)
  {
    int nt;
    n2 = 0; // contador de elementos que quedan
    for (i = 1; i <= n; i++)
    {
      nt = q.quitar();
      if (i % n1 == 1)
      {
        cout << "\n Se quita " << nt << endl;
      }
      else
      {
        q.insertar(nt); // se vuelve a meter en la cola
        n2++;
      }
    }
    n = n2;
    n1 = 1 + random(11);
  }
  cout << "\n\t Los números de la suerte: ";
  mostrarCola(q);
  return 0;
}

template <class T>
void mostrarCola(ColaGenerica<T>& q)
{
  while (! q.colaVacia())
  {
    int v;
    v = q.quitar();
    cout << v << " ";
  }
  cout << endl;
}
```

12.5. BICOLAS: COLAS DE DOBLE ENTRADA

Una *bicola* o *cola de doble entrada* es un conjunto *ordenado* de elementos al que se puede *añadir* o *quitar* elementos desde cualquier extremo del mismo. El acceso a la bicola está permitido desde cualquier extremo, por lo que se considera que es una *cola bidireccional*. La estructura *bicola* es una extensión del *TAD Cola*.

Los dos extremos de una bicola se identifican con los apuntadores `frente` y `final` (mismos nombres que en una cola). Las operaciones básicas que definen una bicola son una ampliación de la operaciones de una cola :

CrearBicola : inicializa una bicola sin elementos.
BicolaVacia : devuelve `true` si la bicola no tiene elementos.

PonerFrente : añade un elemento por extremo frente.
PonerFinal : añade un elemento por extremo final.
QuitarFrente : devuelve el elemento `frente` y lo retira de la bicola.
QuitarFinal : devuelve el elemento `final` y lo retira de la bicola.
Frente : devuelve el elemento `frente` de la bicola.
Final : devuelve el elemento `final` de la bicola.

Al tipo de datos bicola se puede poner restricciones respecto a la entrada o a la salida de elementos. Una *bicola con restricción de entrada* es aquella que sólo permite inserciones por uno de los dos extremos, pero que permite retirar elementos por los dos extremos. Una *bicola con restricción de salida* es aquella que permite inserciones por los dos extremos, pero sólo permite retirar elementos por un extremo.

La representación de una bicola puede ser con un array, con un *array circular*, o bien con *listas enlazadas*. Siempre se debe disponer de dos *marcadores* o variables índice (*apuntadores*) que se correspondan con los extremos, *frente* y *final*, de la estructura.

12.5.1. Bicola genérica con listas enlazadas

La implementación del *TAD* `Bicola` con una lista enlazada se caracteriza por ajustarse al número de elementos; es una implementación dinámica, *crece* o *decrece* según lo requiera la ejecución del programa que utiliza la bicola. Como los elementos de una bicola, y en general de cualquier *estructura contenedora*, pueden ser de cualquier tipo, se declara la clase genérica `Bicola`. Además, la clase va a heredar de `ColaGenerica` ya que es una extensión de ésta.

```
template <class T> class BicolaGenerica : public ColaGenerica<T>
```

De esta forma, `BicolaGenerica` dispone de todas las funciones y atributos de la clase `ColaGenerica`. Entonces, sólo es necesario codificar las operaciones de `Bicola` que no están implementadas en `ColaGenerica`.

```
// archivo BicolaGenerica.h
#include "ColaGenerica.h"

template <class T>
class BicolaGenerica : public ColaGenerica<T>
{
public:
  void ponerFinal(T elemento);
  void ponerFrente(T elemento);
  T quitarFrente();
  T quitarFinal();
  T frenteBicola()const;
  T finalBicola() const;
  bool bicolaVacia();
  void borrarBicola();
  int numElemsBicola() const; // cuenta los elementos de la bicola
};
```

12.5.2. Implementación de las operaciones de `BicolaGenerica`

Las funciones: `ponerFinal(), quitarFrente(), bicolaVacia(), frenteBicola()` son identicas a las funciones de la clase `ColaGenerica insertar(), quitar(), colaVacia() y frenteCola()` respectivamente, y como por el mecanismo de la derivación de clases se han heredado, su implementación consiste en una simple llamada a la correspondiente función heredada.

Añadir un elemento a la bicola

Añadir por el extremo final de `Bicola`.

```
template <class T>
void BicolaGenerica<T> :: ponerFinal(T elemento)
{
  insertar(elemento); // heredado de ColaGenerica
}
```

Añadir por el extremo `frente` de `Bicola`.

```
template <class T>
void BicolaGenerica<T> :: ponerFrente(T elemento)
{
  NodoCola* nuevo;

  nuevo = new NodoCola(elemento);
  if (bicolaVacia())
  {
    final = nuevo;
  }
  nuevo -> siguiente = frente;
  frente = nuevo;
}
```

Sacar un elemento de la bicola

Devuelve el elemento `frente` y lo quita de la *Bicola*, disminuye su tamaño.

```
template <class T>
T BicolaGenerica<T> :: quitarFrente()
{
  return quitar();  // método heredado de ColaLista
}
```

Devuelve el elemento `final` y lo quita de la *Bicola*, disminuye su tamaño. Es necesario recorrer la lista para situarse en el nodo anterior a `final`, y después enlazar.

```
template <class T>
T BicolaGenerica<T> :: quitarFinal()
{
  T aux;
  if (! bicolaVacia())
```

```
  {
    if (frente == final)   // Bicola dispone de un solo nodo
    {
      aux = quitar();
    }
    else
    {
      NodoCola* a = frente;
      while (a -> siguiente != final)
         a = a -> siguiente;
      aux = final -> elemento;
      final = a;
      delete (a -> siguiente);
    }
  }
  else
    throw "Eliminar de una bicola vacía";
  return aux;
}
```

Acceso a los extremos de la bicola

Elemento frente

```
template <class T>
T BicolaGenerica<T>:: frenteBicola()const
{
  return frenteCola();   // heredado de ColaGenerica
}
```

Elemento final

```
template <class T>
T BicolaGenerica<T>:: finalBicola() const
{
  if (bicolaVacia())
  {
    throw "Error: bicola vacía";
  }
  return (final -> elemento);
}
```

Vaciado de la bicola

```
template <class T>
void BicolaGenerica<T> :: borrarBicola()
{
  borrarCola();        // heredado de ColaGenerica
}
```

Verificación del estado y número de elementos de la bicola

```
template <class T>
bool BicolaGenerica<T> :: bicolaVacia() const
{
  return colaVacia();  // heredado de ColaGenerica
}
```

La siguiente operación recorre la estructura, de `frente` a `final`, para contar el número de elementos de que consta.

```
template <class T>
int BicolaGenerica<T> :: numElemsBicola() const
{
  int n = 0;
  NodoCola* a = frente;
  if (!bicolaVacia())
  {
    n = 1;
    while (a != final)
    {
      n++;
      a = a -> siguiente;
    }
  }
  return n;
}
```

EJERCICIO 12.2. La salida a pista de las a vionetas de un aeródromo está organizada en forma de fila(línea), con una capacidad máxima de apar atos en esper a de 16 a vionetas. Las avionetas llegan por el e xtremo izquierdo (final) y salen por el e xtremo derecho (frente). Un piloto puede decidir retir arse de la fila por r azones técnicas, en ese caso todas las a vionetas que siguen han de ser quitadas de la fila, retir ar el apar ato y las a vionetas desplazadas colocarlas de n uevo en el mismo orden relativ o en que estaban. La salida de una a vioneta de la fila supone que las demás son mo vidas hacia adelante , de tal f orma que los espacios libres del estacionamiento estén en la par te izquierda (final).

La aplicación para emular este estacionamiento tiene como entrada un carácter que indica una acción sobre la avioneta, y la matrícula de la avioneta. La acción puede ser llegada (E), salida (S) de la avioneta que ocupa la primera posición y retirada (T) de una avioneta de la fila. En la llegada puede ocurrir que el estacionamiento esté lleno, si esto ocurre la avioneta espera hasta que se quede una plaza libre.

El estacionamiento va a estar representado por una *bicola*, (realmente debería ser una *bicola* de salida restringida). ¿Por qué esta elección?, la salida siempre se hace por el mismo extremo, sin embargo la entrada se puede hacer por los dos extremos, y así se contempla dos acciones: *llegada* de una avioneta nueva; y *entrada de una avioneta que ha sido movida* para que salga una intermedia.

Las avionetas que se mueven para poder retirar del estacionamiento una intermedia, se disponen en una *pila*, así la ultima en entrar será la primera en añadirse al extremo salida del estacionamiento (*bicola*) y seguir en el mismo orden relativo.

Las avionetas se representan mediante una cadena para almacenar, simplemente, el número de matrícula. Entonces, los elementos de la pila y de la *bicola* son de tipo *cadena* (`string`).

Se utiliza la implementación de `PilaGenerica`, en esta ocasión el tipo de dato de los elementos es `string`. La bicola utilizada es de tipo `BicolaGenerica`, también el tipo de dato de los elementos es `string`.

La función `main()` gestiona las operaciones indicadas en ejercicio. La resolución del problema no toma acción cuando una avioneta no puede incorporarse a la fila por estar llena. El

lector puede añadir el código necesario para que, utilizando una cola, las avionetas que no pueden entrar en la fila se guarden en la cola, y cada vez que salga una avioneta se añada otra desde la cola.

En la función `retirar()` se simula el hecho de que una avioneta, que se encuentra en cualquier posición de la *bicola*, decide salir de la fila; la función retira de la fila las avionetas por el *frente*, a la vez que las guarda en una pila, hasta que encuentra la avioneta a retirar. A continuación, se insertan en la fila, por el *frente*, las avionetas de la pila, así quedan en el mismo orden que estaban anteriormente. La constante `maxAvtaFila (16)` guarda el número máximo de avionetas que pueden estar en la fila esperando la salida.

```
#include <iostream>
using namespace std;
#include <string>
#include <ctype.h>
#include "BicolaGenerica.h"
#include "PilaGenerica.h"

const int maxAvtaFila = 16;
template <class T>
bool retirar(BicolaGenerica<T>& fila, string avioneta);

int main()
{
  string avta;
  char ch;
  BicolaGenerica<string> fila;
  bool esta, mas = true;

  while (mas)
  {
    char bf[81];
    cout << "Entrada: acción(E/S/T)matrícula."
         << " Para terminar la simulación: X." << endl;
    do {
      cin >> ch; ch = toupper(ch); cin.ignore(2,'\n');
    } while(ch != 'E' && ch != 'S' && ch != 'T' && ch != 'X');
    if (ch == 'S')              // sale de la fila una avioneta
    {
      if (!fila.bicolaVacia())
      {
        avta = fila.quitarFrente();
        cout << "Salida de la avioneta: " << avta <<endl;
      }
    }
    else if (ch == 'E') // llega a la fila una avioneta
    {
      if (fila.numElemsBicola() < maxAvtaFila)
      {
        cout << " Matricula avioneta: " <<endl;
        cin.getline(bf,80);
        avta = bf;
        fila.ponerFinal(avta);
      }
    }
```

```
      else if (ch == 'T')  // avioneta abandona la fila
      {
        cout << " Matricula avioneta: " << endl;
        cin.getline(bf,80);
        avta = bf;
        esta = retirar(fila, avta);
        if (!esta)
           cout << "¡¡ avioneta no encontrada !!" <<endl;
      }
      mas = !(ch == 'X');
    }
    return 0;
  }

  template <class T>
  bool retirar(BicolaGenerica<T>& fila, string avioneta)
  {
    bool encontrada = false;
    PilaGenerica<string> pila;
    while (!encontrada && !fila.bicolaVacia())
    {
      string avta;
      avta = fila.quitarFrente();
      if (avioneta == avta) // sobrecarga del operator ==
         { encontrada = true;
           cout << "Avioneta" <<avta << "retirada" <<endl;
         }
      else pila.insertar (avta);
    }
    while (!pila.pilaVacia())
        fila.ponerFrente (pila.quitar());
    return encontrada;
  }
```

RESUMEN

Una cola es una lista lineal en la que los datos se insertan por un extremo (`final`) y se extraen por el otro extremo (`frente`). Es una estructura *FIFO* (*first in first out*, primero en entrar primero en salir).

Las operaciones básicas que se aplican sobre colas: `crearCola, colaVacia, colaLlena, insertar, frente` y `quitar`.

`crearCola`, inicializa a una cola sin elementos. Es la primera operación a realizar con una cola. La operación queda implementada en el constructor de la clase `Cola`.

`colaVacia`, determina si una cola tiene o no elementos. Devuelve `true` si no tiene elementos.

`colaLlena`, determina si no se pueden almacenar más elementos en una cola. Se aplica esta operación cuando se utiliza un array para guardar los elementos de la cola.

`insertar`, añade un nuevo elemento a la cola, siempre por el extremo `final`.

`frente`, devuelve el elemento que está, justamente en el extremo `frente` de la cola, sin extraerlo.

`quitar`, extrae el elemento frente de la cola y lo elimina.

La implementación del *TAD Cola*, en C++, se realiza con arrays, o bien con listas enlazadas. La implementación con un array lineal es muy ineficiente; se ha de considerar el array como una estructura circular y aplicar la teoría de los restos para avanzar el *frente* y el *final* de la cola.

La realización de una cola con listas enlazadas permite que el tamaño de la estructura se ajuste al número de elementos. La cola puede crecer indefinidamente, con el único tope de la memoria libre.

Las colas se utilizan en numerosos modelos de sistemas del mundo real: cola de impresión en un servidor de impresoras, programas de simulación, colas de prioridades en organización de viajes.

Las *bicolas* son *colas dobles* en el sentido de que las operaciones básicas *insertar* y *retirar* elementos se realizan por los dos extremos. A veces se ponen restricciones de entrada o de salida por algún extremo. Una bicola es, realmente, una extensión de una cola. La implementación natural del *TAD Bicola* es con una clase derivada de la clase `Cola`.

EJERCICIOS

12.1. Considerar una cola de nombres representada por una array circular con 6 posiciones, el campo frente con el valor: `frente = 2`. Y los elementos de la cola: `Mar, Sella, Centurión`.
Escribir los elementos de la cola y los campos `frente` y `final` según se realizan estas operaciones:

- Añadir `Gloria` y `Generosa` a la cola.
- Eliminar de la cola.
- Añadir `Positivo`.
- Añadir `Horche`.
- Eliminar todos los elementos de la cola.

12.2. A la clase que representa una cola implementada con un array circular y dos variables `frente` y `final`, se le añade una variable más que guarda el número de elementos de la cola. Escribir de nuevo las funciones de manejo de colas considerando este campo contador.

12.3. Suponer que para representar una `bicola` se ha elegido una lista doblemente enlazada, y que los extremos de la lista se denominan `frente` y `final`. Escribir la clase *Bicola* con esta representación de los datos y las operaciones del *TAD Bicola*.

12.4. Supóngase que se tiene la clase `Cola` que implementa las operaciones del *TAD Cola*. Escribir una función para crear *un clon* (una copia) de una cola determinada. Las operaciones que se han de utilizar serán únicamente las del *TAD Cola*.

12.5. Considere una *bicola* de caracteres, representada en un array circular. El array consta de 9 posiciones. Los extremos actuales y los elementos de la bicola:

```
frente = 5 final = 7   bicola: A,C,E
```

Escribir los extremos y los elementos de la bicola según se realizan estas operaciones:

- Añadir los elementos `F` y `K` por el `final` de la bicola.
- Añadir los elementos `R`, `W` y `V` por el `frente` de la bicola.

- Añadir el elemento `M` por el `final` de la bicola.
- Eliminar dos caracteres por el `frente`.
- Añadir los elementos `K` y `L` por el `final` de la bicola.
- Añadir el elemento `S` por el `frente` de la bicola.

12.6. Se tiene una pila de enteros positivos. Con las operaciones básicas de pilas y colas escribir un fragmento de código para poner todos los elementos de la pila que sean pares en la cola.

12.7. Implementar el *TAD Cola* utilizando una lista enlazada circular. Por conveniencia, establecer el acceso a la lista, `lc`, por el último nodo (elemento) insertado, y considerar al nodo siguiente de `lc` el primero o el que más tarde se insertó.

PROBLEMAS

12.1. Con un archivo de texto se quieren realizar las siguientes acciones: formar una lista de colas, de tal forma que cada nodo de la lista esté en la dirección de una cola que tiene todas las palabras del archivo que empiezan por una misma letra. Visualizar las palabras del archivo, empezando por la cola que contiene las palabras que comienzan por *a*, a continuación las de la letra *b*, y así sucesivamente.

12.2. Se tiene un archivo de texto del cual se quiere determinar las frases que son palíndromo. Para lo cual se ha de seguir la siguiente estrategia:

- Considerar cada línea del texto una frase.
- Añadir cada carácter de la frase a una pila y a la vez a una cola.
- Extraer carácter a carácter, y simultáneamente de la pila y de la cola. Su comparación determina si es palíndromo o no.

Escribir un programa que lea cada línea del archivo y determine si es palíndromo.

12.3. Escribir un programa en el que se generen 100 números aleatorios en el rango `-25 .. +25` y se guarden en una cola implementada mediante un array circular. Una vez creada la cola, el usuario puede pedir que se forme otra cola con los números negativos que tiene la cola original.

12.4. Escribir una función que tenga como argumentos dos colas del mismo tipo. Devuelva cierto si las dos colas son idénticas.

12.5. Un pequeño supermercado dispone en la salida de tres cajas de pago. En el local hay 25 carritos de compra. Escribir un programa que simule el funcionamiento del supermercado, siguiendo las siguientes reglas:

- Si cuando llega un cliente no hay ningún carrito disponible, espera a que lo haya.
- Ningún cliente se impacienta y abandona el supermercado sin pasar por alguna de las colas de las cajas.
- Cuando un cliente finaliza su compra, se coloca en la cola de la caja que hay menos gente, y no se cambia de cola.
- En el momento en el cual un cliente paga en la caja, el carro de la compra que tiene queda disponible.

Representar la lista de carritos de la compra y las cajas de salida mediante colas.

12.6. En un archivo F están almacenados números enteros arbitrariamente grandes. La disposición es tal que hay un número entero por cada línea de F. Escribir un programa que muestre por pantalla la suma de todos los números enteros. Al resolver el problema habrá que tener en cuenta que al ser enteros grandes no pueden almacenarse en variables numéricas.
Utilizar dos pilas para guardar los dos primeros números enteros, almacenándose dígito a dígito. Al extraer los elementos de la pila salen en orden inverso y, por tanto, de menos peso a mayor peso, se suman dígito con dígito y el resultado se guarda en una cola, también dígito a dígito. A partir de este primer paso, se obtiene el siguiente número del archivo, se guarda en una pila y a continuación se suma dígito a dígito con el número que se encuentra en la cola; el resultado se guarda en otra cola. El proceso se repite, nuevo número del archivo se mete en la pila, que se suma con el número actual de la cola.

12.7. Una empresa de reparto de propaganda contrata a sus trabajadores por días. Cada repartidor puede trabajar varios días continuados o alternos. Los datos de los repartidores se almacenan en una lista enlazada. El programa a desarrollar contempla los siguientes puntos:

- Crear una cola que guarde el número de la seguridad social de cada repartidor y la entidad anunciada en la propaganda para un único día de trabajo.
- Actualizar la lista citada anteriormente (que ya existe con contenido) a partir de los datos de la cola.

La información de la lista es la siguiente: número de seguridad social, nombre y total de días trabajados. Además, está ordenada por el número de la seguridad social. Si el trabajador no está incluido en la lista debe añadirse a la misma de tal manera que siga ordenada.

12.8. El supermercado "Esperanza" quiere simular los tiempos de atención al cliente a la hora de pasar por la caja, los supuestos de que se parte para la simulación son los siguientes:

- Los clientes forman una única fila. Si alguna caja está libre, el primer cliente de la fila es atendido. En el caso de que haya más de un caja libre, la elección del número de caja por parte del cliente es aleatoria.
- El número de cajas de que se dispone para atención a los clientes es de tres, salvo que haya mas de 20 personas esperando en la fila, entonces se habilita una cuarta caja, ésta se cierra cuando no quedan clientes esperando. El tiempo de atención de cada una de las caja está distribuido uniformemente, la caja 1 entre 1.5 y 2.5 minutos; la caja 2 entre 2 y 5 minutos, la caja 3 entre 2 y 4 minutos. La caja 4, cuando está abierta tiene un tiempo de atención entre 2 y 4.5 minutos.
- Los clientes llegan a la salida en intervalos de tiempo distribuidos uniformemente, con un tiempo medio de 1 minuto.

El programa de simulación se pide hacerlo para 7 horas de trabajo. Se desea obtener una estadística con los siguientes datos:

- Clientes atendidos durante la simulación.
- Tamaño medio de la fila de clientes.
- Tamaño máximo de la fila de clientes.
- Tiempo máximo de espera de los clientes.
- Tiempo en que está abierto la cuarta caja.

12.9. La institución académica de *La Alcarria* dispone de 15 computadores conectados a Internet. Se desea hacer una simulación de la utilización de los computadores por los alumnos. Para

ello se supone que la frecuencia de llegada de un alumno es de 18 minutos las 2 dos primeras horas y de 15 minutos el resto del día. El tiempo de utilización del computador es un valor aleatorio, entre 30 y 55 minutos.
El programa debe tener como salida líneas en las que se refleja la llegada de un alumno, la hora en que llega y el tiempo de la conexión. En el supuesto de que llegue un alumno y no existan computadores libres el alumno no espera, se mostrará el correspondiente aviso. En una cola de prioridad se tiene que guardar los distintos "eventos" que se producen, de tal forma que el programa avance de evento a evento. Suponer que la duración de la simulación es de las 10 de la mañana a las 8 de la tarde.

12.10. La entrada a una sala de arte que ha inaugurado una gran exposición sobre la evolución del arte rural, se realiza por tres torniquetes. Las personas que quieren ver la exposición forman un única fila y llegan de acuerdo a una distribución exponencial, con un tiempo medio entre llegadas de 2 minutos. Una persona que llega a la fila y ve más de 10 personas esperando se va con una probabilidad del 20 por 100, aumentando en 10 puntos por cada 15 personas más que haya esperando, hasta un tope del 50 por 100. El tiempo medio que tarda una persona en pasar es de 1 minuto (compra de la entrada y revisión de los bolsos). Además, cada visitante emplea en recorrer la exposición entre 15 y 25 minutos distribuido uniformemente. La sala sólo admite, como máximo, 50 personas. Simular el sistema durante un período de 6 horas para determinar:

- Número de personas que llegan a la sala y número de personas que entran.
- Tiempo medio que debe esperar una persona para entrar en la sala.

CAPÍTULO 13

Colas de prioridades y montículos

Objetivos

Con el estudio de este capítulo usted podrá:

- Conocer el tipo de datos *Colas de prioridades*.
- Describir las operaciones fundamentales de las colas de prioridades.
- Implementar colas de prioridades con listas enlazadas.
- Conocer la estructura de datos montículo binario.
- Implementar las operaciones básicas de los montículos con *complejidad logarítmica*.
- Aplicar un montículo para realizar un ordenación.
- Conocer los montículos binomiales.

Contenido

Conceptos clave

- Árbol binario.
- Cola.
- Complejidad.
- Listas enlazadas.
- Matrícula binomial.
- Montículo.
- Nodo *padre*, nodo *hijo*.
- Ordenación.
- Prioridad.

INTRODUCCIÓN

En este capítulo se estudia, pormenorizadamente, la estructura de datos *colas de prioridades*, estructura que se utiliza para planificar tareas en aplicaciones informáticas donde la prioridad de la tarea se corresponde con la clave de ordenación. Las *colas de prioridades* también se aplican en los sistemas de simulación donde los sucesos se deben procesar en orden cronológico. El común denominador de estas aplicaciones es que siempre se selecciona el *elemento mínimo* una y otra vez hasta que no quedan más elementos. Tradicionalmente, se ha designado la prioridad más alta con una clave menor, así los elementos de prioridad 1 tienen más prioridad que los elementos de prioridad 3.

Se utilizan estructuras lineales (listas enlazadas, vectores) para implementar las colas de prioridades; también, una estructura de multi colas. El montículo binario, simplemente montículo, es la estructura más apropiada para implementar eficientemente las colas de prioridades.

La *idea intuitiva* de un montículo es la que mejor explica esta estructura de datos. En la *parte más alta* del montículo se encuentra el *elemento más pequeño*; los elementos descendientes de uno dado son mayores en la estructura montículo. La estructura montículo garantiza que el tiempo de las operaciones básicas insertar y eliminar mínimo, sean de complejidad logarítmica.

13.1. COLAS DE PRIORIDADES

El término cola sugiere la forma en que ciertos objetos esperan la utilización de un determinado servicio. Por otro lado, el término *prioridad* sugiere que el servicio no se proporciona únicamente aplicando el concepto "*primero en llegar primero en ser atendido*" sino que cada objeto tiene asociado una prioridad basada en un criterio objetivo. La cola de prioridad es una estructura ordenada que se utiliza para guardar elementos en un orden establecido. El orden para extraer un elemento de la estructura sigue estas reglas:

- Se elige la cola no vacía que se corresponde con la mayor prioridad.
- En la cola de mayor prioridad los elementos se procesan según el orden de llegada: *primero en entrar primero en salir*.

Las Colas de prioridad son muy importantes en la implementación de algoritmos ávidos y en la simulación de eventos. Un ejemplo de organización formando colas de prioridades es el sistema de tiempo compartido, necesario para mantener una serie de procesos que esperan ser ejecutados por el procesador; cada proceso lleva asociado una prioridad, de tal manera que se ejecuta siempre el proceso de mayor prioridad, y dentro de la misma prioridad, aquel proceso que primero llega. Otro posible ejemplo que se realiza con una *cola de prioridad* es la simulación de sucesos que ocurren en un tiempo discreto, como es la atención de una fila de clientes en un sistema de n ventanillas de despacho de billetes de transporte.

13.1.1. Declaración del *TAD cola de prioridad*

La principal característica de una cola de prioridades consiste en que repetidamente se selecciona del conjunto de elementos el de clave máxima, o prioridad máxima, y dentro de esta prioridad máxima en el orden en que llegan. Normalmente, se insertan nuevos elementos en la

estructura de datos, a la vez que se procesan otros, colocándose en la estructura de acuerdo con su prioridad, y dentro de la misma prioridad el último; esto lleva a especificar las operaciones que se detallan a continuación.

Tipo de dato: *Proceso* que tiene asociado una *prioridad*.

Operaciones:

`CrearColaPrioridad`	*Inicia la estructura con las prioridades sin elementos.*
`InserEnPrioridad`	*Añade un elemento a la cola según prioridad.*
`ElementoMin`	*Devuelve el elemento de la cola con la prioridad más alta.*
`QuitarMin`	*Devuelve y retira el elemento de la cola con prioridad más alta.*
`ColaVacía`	*Comprobar si una determinada cola no tiene elementos.*
`ColaPrioridadVacía`	*Comprueba si todas las colas de la estructura están vacías.*

A tener en cuenta

En programación siempre se ha seguido la convención de considerar un intervalo de máxima a mínima prioridad con valores clave en orden inverso; por ejemplo, asignar a la máxima prioridad la clave 0, a la siguiente la clave 1, y así sucesivamente de tal forma que la clave mínima sea n.

13.1.2. Implementación

Una forma simple de implementar una *cola de prioridades* es utilizar un array de tareas ordenado de acuerdo con su prioridad, y a igualdad de prioridades añadir la tarea la última dentro de la misma prioridad. Otra implementación puede ser usar una lista enlazada y ordenada según la prioridad, con la mismas características que el array de tareas. Otra alternativa consiste en emplear un array o tabla de tantos elementos como prioridades estén previstas en la estructura; cada elemento de la tabla guarda a los objetos con la misma prioridad en el orden en que van llegando. Por último, se puede utilizar la estructura del montículo para guardar los componentes. La característica del montículo permite recuperar el elemento de máxima prioridad con una simple operación (caso de montículo máximo).

Los elementos de una *cola de prioridades* son objetos con la propiedad de *ordenación*, de tal forma que se pueda realizar comparaciones. Esto equivale a que dispongan de un atributo, de tipo ordinal, representando la prioridad del objeto.

13.2. VECTOR DE PRIORIDADES

La forma más simple de implementar una *cola de prioridades* es con un array de tareas ordenado crecientemente por el número de prioridad de cada una de ellas, teniendo en cuenta que: cada vez que se añada una nueva tarea a la *cola de prioridades* hay que desplazar todas las tareas de prioridad mayor hacia la zona más alta del array; y cada vez que se elimine una tarea de la cola de prioridades hay que desplazar todas las tareas que queden una posición hacia la zona más baja del array.

13.2.1. Implementación

Se define el array de Tareas de tamaño variable, inicializado con el constructor de la `Colade-Prioridad`. Se asume que las prioridades varían en cualquier rango entero. Se declarará la Estructura `Tarea`, con los campos `nombre` y `prioridad`, para representar los elementos del objeto *cola de prioridades*:

```
#define Ndatosmax 160 // para crear cola de prioridad de 160 tareas
    //como máximo en caso de que no se especifique cuántas se tiene
typedef struct
{
    int prioridad;
    char nombre[51];
}Tarea;
```

Declaración de la clase Cola de Prioridad

La clase ColaPrioridad tiene los siguientes tributos:

- `PcolaPrioridad`. Puntero a `Tarea` encargado de almacenar las distintas Tareas de la *Cola de Prioridad*. Es el array almacén de tareas.
- **`Ndatos.`** Variable entera que representan el número de datos máximo que podrá almacenar el array.
- `NdatosActual`. Variable entera que almacena el número de datos que contiene en cada momento la `Cola de Prioridad`. Siempre varía entre `0` y `Ndatos`.

```
Cola de prioridad creada con 16 elementos
Ndatosmax = 16
NdatosActual = 0
```

0	1	2	3	4	5	6	7	8	9

```
InserEnPrioridad   la tarea B con prioridad 5
InserEnPrioridad   la tarea C con prioridad 2
InserEnPrioridad   la tarea A con prioridad 4
InserEnPrioridad   la tarea D con prioridad 2
InserEnPrioridad   la tarea G con prioridad 4


Ndatosmax = 16
NdatosActual = 5
```

0	1	2	3	4	5	6	7	8	9
C	D	A	G	B					
2	2	4	4	5					

```
class ColaPrioridad
{
    protected:
      Tarea *PColaPrioridad;
      int Ndatos, NdatosActual;
    public:
      ColaPrioridad();                        // constructor por defecto
      ColaPrioridad(int n);                   //constructor sobrecargado
                         //con número máximo de posiciones para el array
      bool EsVaciaColaPrioridad();              // Decide si está vacía
      void InserEnPrioridad(Tarea e);    //añade a la cola de prioridad
      void QuitarMin();          //borra el elemento de menor prioridad
      Tarea ElementoMin();     //retorna el elemento de menor prioridad
      ~ColaPrioridad(){}                       // destructor por defecto
    private:
      Tarea&operator[](int i); //función oculta. Sobrecarga operador []
};
```

El constructor por defecto es el responsable de establecer el número máximo de Tareas que puede almacenar la cola de prioridad, mediante una asignación dinámica de memoria al puntero a Tareas PColaPrioridad.

```
ColaPrioridad::ColaPrioridad()
{
  Ndatos = Ndatosmax;
  PColaPrioridad = new Tarea[Ndatos];
  NdatosActual = 0;
}
```

El constructor sobrecargado de `ColaPrioridad` permite declarar una *Cola de Prioridad* que puede almacenar un número variable de Tareas, decidido por el usuario de la *Cola de Prioridad*.

```
ColaPrioridad::ColaPrioridad(int n): Ndatos(n)
{
   PColaPrioridad = new Tarea[Ndatos];
   NdatosActual = 0;
}
```

La *Cola de Prioridad* usa el operador privado sobrecargado `[]` para acceder a cada una de las posiciones del array que almacena las tareas.

```
Tarea & ColaPrioridad::operator[](int i)
{
   return PColaPrioridad[i];
}
```

13.2.2. Insertar

La operación que añade una nueva `Tarea`, a la *cola de prioridades* es `InserEnPrioridad`. La `Tarea` se inserta en el array en la posición que le corresponde. La función miembro informa, además, mediante una excepción de un posible error en caso de que la *Cola de prioridad* esté llena.

Ndatosmax = 16
NdatosActual = 5

0	1	2	3	4	5	6	7	8	9
C	D	A	G	B					
2	2	4	4	5					

```
void ColaPrioridad::InserEnPrioridad(Tarea e)
{
   int i = 0;
   bool enc = false;
   while (!enc && i < NdatosActual)  // búsqueda de la posición donde se
                     //almacenará la tarea en caso de que haya capacidad
     if (PColaPrioridad[i].prioridad <= e.prioridad)
        i++;
     else
        enc = true;
   if( NdatosActual < Ndatos)
   {
               // hay que insertar la Tarea en posicón i desplazar datos
     for (int j = NdatosActual; j > i; j--)
        PColaPrioridad[j] = PColaPrioridad[j-1]; // se mueven en el array
     PColaPrioridad[i] = e;
     NdatosActual++;
     }
   else
     throw "Cola de prioridad llena";
}
```

InserEnPrioridad la tarea F con prioridad 2
Ndatosmax = 16
NdatosActual = 6

0	1	2	3	4	5	6	7	8	9
C	D	F	A	G	B				
2	2	2	4	4	5				

La *complejidad* de Insertar un elemento en la *Cola de Prioridad* viene dada por los dos bucles necesarios para su codificación. El primero de ellos busca en el array la posición `i` donde se debe alacena la `Tarea`. El segundo bucle desplaza tareas hacia la zona alta del array para dejar libre la posición `i`. El tiempo de ejecución es, por tanto, lineal. Esta c*omplejidad* puede bajarse a tiempo mitad si se utiliza la implementación de listas enlazadas ordenadas, ya que no necesita desplazar la Tareas, o si se usa la implementación con montículos, en cuyo caso la complejidad es logarítmica. Veánse implementaciones realizadas en los Apartados 13.3 y 13.5.

13.2.3. Elemento de máxima prioridad

La operaciones `ElementoMin` y `QuitarMin` retorna o elimina respectivamente el elemento de mayor prioridad que siempre se encuentra en la posición `0` del array.

La función miembro `ElementoMin` retorna el elemento que se encuentra en la posición 0 en caso de que exista.

```
Tarea ColaPrioridad::ElementoMin()
{
     if(NdatosActual > 0)
        return PColaPrioridad[0];
     else
        throw " cola de Prioridad Vacía";
}
```

El método `QuitarMin`, es semejante a la función miembro `ElementoMin`, excepto que en este caso se borra el primer elemento del array y se desplazan las Tareas que quedan en la *Cola de Prioridad* una posición en el array.

```
void ColaPrioridad::QuitarMin()
{
   if (NdatosActual > 0) // si no es vacía
   {
     for (int i = 1; i < NdatosActual; i++)
       PColaPrioridad[i-1] = PColaPrioridad[i];
       NdatosActual--;
   }
   else
     throw "Cola de prioridad Vacía";
}
```

`QuitarMin; Ndatosmax = 16`
`NdatosActual= 5`

0	1	2	3	4	5	6	7	8	9
D	F	A	G	B					
2	4	4	4	5					

La complejidad de `ElementoMin` es constante, pero la de `QuitarMin` es lineal, ya que necesita un bucle para desplazar las tareas. Esta complejidad lineal de `QuitarMin` puede reducirse a tiempo constante, si se implementa el array como circular, ya que en este caso no es necesario que todas las tareas de la *Cola de Prioridad* se desplacen una posición en el array. Otra forma de conseguir que baje a tiempo constante es usar las implantaciones de Lista enlazada ordenada o montículo que se desarrollan posteriormente. Véanse las implementaciones realizadas en los Apartados 13.3 y 13.4.

13.2.4. Cola de prioridad vacía

El método Es`VaciaColaPrioridad`, decide si la *Cola de Prioridad* está vacía comprobando el número de datos actuales que hay almacenados.

```
bool ColaPrioridad::EsVaciaColaPrioridad()
{
return (NdatosActual == 0);
}
```

La complejidad de este método constante.

13.3. LISTA DE PRIORIDADES ENLAZADA Y ORDENADA

Una forma más sencilla de implementar una *cola de prioridades* es mediante una *Lista Enlazada de objetos* (*Tareas*) ordenada respecto a la prioridad del objeto. La lista se organiza de tal forma que un elemento *x* precede a *y* si:

1. *Prioridad(x)* es mayor que *Prioridad(y)*.
2. Ambos tienen la misma prioridad, pero *x* se añadió antes que *y*.

Con esta organización, siempre el primer elemento de la lista es el elemento de la cola de prioridades de máxima prioridad, el último elemento de la lista es el de menor prioridad y, por consiguiente, el último a procesar. De esta forma, se garantiza que el acceso y borrado del primer elemento sea en tiempo constante. No ocurre lo mismo con la inserción que es de tiempo lineal.

13.3.1. Implementación

Se declara la clase `Nodo` que contiene como atributos `e` de tipo `Telemento` y `Sig` como puntero a la clase `Nodo`, con su correspondiente constructor destructor y las funciones miembros encargadas de poner y obtener cada uno de sus atributos.

```
class Nodo
{
    protected:
        Tarea e;
        Nodo *Sig;
    public:
        Nodo (Tarea x){ e = x; Sig = NULL;};   // constructor
        Tarea ObtenerE(){ return e;}           // Obtener elemento
        void PonerE(Tarea x){ e = x;}          // poner elemento
        Nodo * ObtenerSig(){ return Sig;}      // Obtener siguiente
        void PonerSig( Nodo *p){ Sig = p;}     // poner siguiente
        ~Nodo(){}                              // destructor
};
```

La clase `ColaPrioridad` representa la *cola de prioridades* mediante una lista enlazada ordenada por prioridades que es una clase agregada de la clase `Nodo`. El número máximo de elementos que puede almacenar así como el de prioridades es en teoría ilimitado.

```
class ColaPrioridad
{
    protected:
      Nodo * CP;
```

```
    public:
      ColaPrioridad();          // constructor
      bool EsVaciaColaPrioridad();
      void InserEnPrioridad(Tarea e);
      void QuitarMin();
      Tarea ElementoMin();
      ~ColaPrioridad() // destructor por defecto
};
```

El constructor por defecto crea la *Cola de Prioridad* inicializando el atributo `CP` Puntero a `Nodo` **`NULL`**. Este constructor tiene tiempo de ejecución constante.

```
ColaPrioridad::ColaPrioridad()
{
   CP = NULL;
}
```

Por su parte el destructor de la *Cola de Prioridad*, destruye toda la lista enlazada de tareas, recorriéndola desde la primera posición de la lista hasta la última, por lo que su tiempo de ejecución es lineal.

```
ColaPrioridad:: ~ColaPrioridad()
{
    Nodo * Aux;
      while(CP)           // mientras queden datos en la lista
      {
          Aux = CP;
          CP = CP->ObtenerSig();         //avanzar en la lista
          Aux->PonerSig(NULL);
          delete Aux;                          //liberar Nodo
      }
}
```

13.3.2. Insertar

La nueva `Tarea` de la *Cola de Prioridades* se añade en la posición que le corresponde, teniendo en cuenta su prioridad, por lo que es necesario recorrer la lista enlazada ordenada hasta encontrar la posición donde insertar la `Tarea`. El bucle de búsqueda en la lista enlazada ordenada debe avanzar si la prioridad de la Tarea almacenada en el `Nodo` que se encuentra en la posición actual `pos` es menor o igual que la prioridad de la `Tarea` a insertar. Una vez que el bucle ha terminado hay que insertar la nueva `Tarea`, teniendo en cuenta que la inserción en una lista enlazada puede realizarse en el principio de la lista o en el centro final.

```
void ColaPrioridad::InserEnPrioridad(Tarea e)
{
   Nodo* Pos = CP, *Ant = NULL, *Nuevo = new Nodo(e);
   bool enc = false;
   while (!enc && Pos)  // mientras queden datos y no lo haya encontrado
     if (Pos->ObtenerE().prioridad <= e.prioridad)
     {          // avanzar en la búsqueda de la posición donde insertar
         Ant = Pos;
         Pos = Pos->ObtenerSig();
```

```
        }
        else
            enc = true;
                              // hay que insertar la Tarea e entre Ant y Pos
        // esta inserción puede ser en el pricipio de la lista o no principio
        if(Ant) // no es el primero
          Ant->PonerSig(Nuevo);
        else
          CP = Nuevo;
        Nuevo->PonerSig(Pos);
}
```

El *coste*, en el peor de los casos, de la búsqueda en la lista enlazada ordenada crecientemente es lineal, en el pero de los casos, por lo que el coste de la operación es lineal (*complejidad lineal O(n)*). Esta complejidad puede bajarse a tiempo logarítmico si se usa la implementación de montículos realizada en 13.5.

13.3.3. Elemento de máxima prioridad

Con esta representación de una *Cola de Prioridades* el elemento de máxima prioridad siempre es el primero. La implementación de las operaciones **`ElementoMin`** y **`QuitarMin`** es inmediata, ya que siempre se encuentra en el primer `Nodo` de la lista.

```
Tarea ColaPrioridad::ElementoMin()
{
    if(! CP)
     return CP->ObtenerE();
    else
        throw " cola de Prioridad Vacía";
}

void ColaPrioridad::QuitarMin()
{
     Nodo *Aux;
     if (!CP)                  // si no es vacía borrar el primer Nodo
     {
        Nodo *Aux = CP;
        CP = Aux->ObtenerSig(); //Avanza la Cola de Prioridad
        Aux-> PonerSig(NULL);
        delete Aux;
     }
     else
       throw «Cola de prioridad Vacía»;
}
```

La *complejidad* de la operación **`ElementoMin`** y **`QuitarMin`** es constante.

13.3.3. Cola de prioridad vacía

Se comprueba la condición *es vacía la Cola de Prioridad* con el atributo **`CP`** de la lista enlazada ordenada. La *Cola de Prioridad* estará vacía si apunta a `NULL`. El tiempo de ejecución de la operación es constante.

```
bool ColaPrioridad::EsVaciaColaPrioridad()
{
    return (!CP);
}
```

13.4. ARRAY O TABLA DE PRIORIDADES

Quizá la forma más intuitiva de implementar una *Cola de Prioridades* es con una tabla o array de listas, cada elemento de la tabla se organiza a la manera de una cola (*primero en entrar primero en salir*) y representa a los objetos con la misma prioridad. La Figura 13.1 muestra una tabla en la que cada elemento se corresponde con una prioridad y de la que emerge una cola.

La operación **`inserEnPrioridad`** añade una nueva tarea T de prioridad m a la estructura, siguiendo estos pasos:

1. Buscar la prioridad `m` en la tabla.
2. Si existe, poner el elemento `T` al final de la cola `m`.
3. Si la prioridad `m` está fuera del rango de prioridades generar un error.

La operación `elementoMin` devuelve el elemento `frente` de la cola no vacía que tiene la máxima prioridad. La operación `quitarMin` también devuelve el elemento de máxima prioridad y, además, lo extrae de la cola.

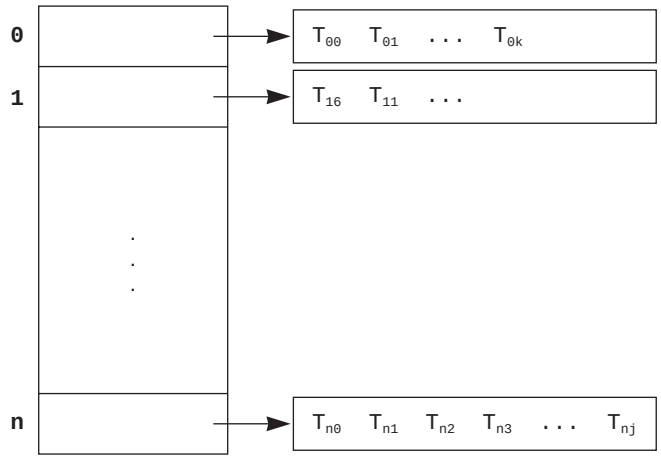


Figura 13.1. Cola de prioridades, de 0 a n prioridades.

13.4.1. Implementación

La tabla se define como un array de tamaño el número de prioridades. Se asume que estas prioridades varían en un rango de `0` al máximo de prioridades previsto. Los elementos de la tabla son colas (*first in, first out*), a su vez implementada con una lista circular. Por ello, se incluye el archivo `colacircular.h` para poder aplicar las operaciones y hacer uso de los tipos de datos ya declarados.

Se declarará la Estructura *Tarea*, con los campos nombre y prioridad, para representar los elementos del objeto *cola de prioridades*:

```
#define N 12
typedef struct
{
      int prioridad;
      char nombre[51];
}Tarea;
```

Se define `Telemento` como un sinónimo de la estructura `Tarea`

```
typedef Tarea Telemento;
```

Se usa la `Nodo` implementada en 13.3. Se declara e implementa la clase `Cola` mediante una lista circular de acuerdo a las especificaciones realizadas en el Capítulo 9. El archivo `cola-circular.h`, contiene la implementación:

```
class Cola
{
   protected:
      Nodo *C ;
   public:
      Cola() { C = NULL; }         //constructor de la Cola
      bool EsvaciaC(){ return !C;} //decide si está vacía
      Cola (const Cola &p2);       // constructor de copia
      void AnadeC( Tarea e);       //añade un elemento a la cola
      Tarea PrimeroC();            //retorna el primer elemento de la cola
      void BorrarC();              //borra el primer elemento de la cola
      ~Cola();                     //destructor
};

Cola::Cola (const Cola &p2)
{// realiza una copia del objeto que se le pasa como parámetro //en el ac-
tual
   Tarea e;
   Nodo* a = p2.C;
   if (a != NULL)
   {
      a = a->ObtenerSig();
      while ( a != p2.C)
      {
         AnadeC ( a -> ObtenerE());
         a = a->ObtenerSig();
      }
   AnadeC ( a -> ObtenerE());   //falta por añadir el último
   }
}

void Cola::AnadeC(Tarea e)
{
   Nodo *aux;
   aux = new Nodo(e);
   if(C)
```

```
   {
      aux-> PonerSig(C-> ObtenerSig());
      C-> PonerSig(aux);
   }
   else
      aux-> PonerSig(aux);
   C = aux;
}

Tarea Cola::PrimeroC()
{
   if (C)
      return (C->ObtenerSig())-> ObtenerE();
      else
   throw "Cola vacía no tiene primero";
}

void Cola::BorrarC()
{
   Nodo *paux;
   if(C)
   {
      paux=C->ObtenerSig();
      if (C==paux)
         C = NULL;
      else
         C->PonerSig(paux->ObtenerSig());
      delete paux;
   }
   else
      throw " Cola vacía no se puede Borrar";
}

Cola:: ~Cola()
{
   Nodo *paux;
   while(C != 0)
   {
      paux = C;
      C = C->ObtenerSig();
      C->PonerSig(NULL);
      delete paux;
   }
}
```

Declaración de la clase Cola de Prioridad

La clase `ColaPrioridad` que representa tiene como atributos: el array `CP` de colas de tamaño máximo declarado anteriormente (`N`); y el atributo `NumPrioridades` que almacena el número de prioridades de cada objeto `ColaPrioridad`. Este tamaño máximo podría eliminarse si se utilizara una lista enlazada ordenada por prioridades que almacenara en cada nodo una `Cola` de Tareas, pero en todo caso la implementación de las funciones `EsVaciaColaPrioridad`, `QuitarMin`, `ElementoMin` sería de tiempo lineal al igual que en la implementación

que se presenta en este apartado. A las funciones anteriores habría que añadir, además, tiempo lineal a la función `InserEnPrioridad`, ya que requeriría la búsqueda de la prioridad en una lista enlazada ordenada.

```
class ColaPrioridad
{
   protected:
      Cola CP[N+1];
      int NumPrioridades;
   public:
      ColaPrioridad();                          // constructor por defecto
      ColaPrioridad(int Prioridad);             //constructor sobrecargado
                                                //con número de prioridades
      bool EsVaciaColaPrioridad();                 // Decide si está vacía
      void InserEnPrioridad(Tarea e);     //añade a la cola de prioridad
      void QuitarMin();            //borra el elemento de menor prioridad
      Tarea ElementoMin();      //retorna el elemento de menor prioridad
      ~ColaPrioridad(){}                         // destructor por defecto
   private:
      int Encontrar();                           //función miembro oculta
};
```

El constructor por defecto es el responsable de establecer el número de prioridades máximo y definir el array:

```
ColaPrioridad::ColaPrioridad()
{
   NumPrioridades = N + 1;
   for(int i = 0; i < NumPrioridades; i++)
      CP[i] = Cola();                  // llamada al constructor de Cola
}
```

Se asume que la máxima prioridad es `0` y la mínima `N`. También, hay una correspondencia biunívoca entre el índice de la tabla y el ordinal de la prioridad. La complejidad del constructor es en este caso lineal.

El constructor sobrecargado es el responsable de establecer el número de prioridades de un objeto con un Número de prioridades máximo menor o igual que `N`, lanzando una excepción en el caso de que la prioridad que reciba como parámetro no cumpla la especificación de estar en el rango `0` a `N`.

```
ColaPrioridad::ColaPrioridad(int Prioridades)
{
   if (0 <= Prioridades && Prioridades <= N )
   {
      NumPrioridades = Prioridades + 1;
      for( int i = 0; i < NumPrioridades; i++)
         CP[i] = Cola();               // llamada al constructor de Cola
   }
   else
      throw "Tarea con prioridad fuera de rango";
}
```

13.4.2. Insertar

La operación que añade una nueva `Tarea`, o bien un nuevo elemento, a la *Cola de Prioridades* es `InserEnPrioridad`. La `Tarea` se inserta en la `Cola` almacenada en `CP[prioridad]` siendo `prioridad` la asociada la que tiene asociada la `Tarea`. La función miembro informa, además, mediante una excepción de un posible error en la prioridad de la tarea.

```
void ColaPrioridad::InserEnPrioridad(Tarea e)
{
   int prioridad = e.prioridad;
   if (0 <= prioridad && prioridad <= NumPrioridades)
      CP[prioridad].AnadeC(e);
   else
      throw "Tarea con prioridad fuera de rango";
}
```

La *complejidad* de Insertar un elemento en la *Cola de Prioridad* es la requerida por la operación de insertar en la `Cola` almacenada en `CP[p]`, por consiguiente, la operación tiene *complejidad constante* (tiempo constante) ya que la función miembro de la clase `Cola` `AnadeC`, está implementada en tiempo constante.

13.4.3. Elemento de máxima prioridad

Las operaciones **`ElementoMin`** y **`QuitarMin`** necesitan buscar, en primer lugar, el elemento de máxima prioridad usando la función miembro oculta `Encontrar`, que retorna el índice del atributo arrray `CP` que contiene la `Cola` con mayor prioridad, en caso de que exista y `-1` en otro caso. El método `Encontrar` busca mediante un algoritmo voraz la primera posición de `CP` que contiene una `Cola` no vacía ya que el convenio establecido es "la máxima prioridad se corresponde con `0` y la menor con `NumPrioridades`".

```
int ColaPrioridad::Encontrar()
{
   int Indice = -1;              / inicialización del índice de búsqueda
   bool Encontrado = false;                        // bandera de búsqueda
   while (Indice < NumPrioridades && !Encontrado)
   {          // algoritmo voraz de búsqueda de la primera cola no vacía
      Indice++;         //como se inicializa a -1 se incrementa antes de
                                                                 //comparar
      if (! CP[Indice].EsvaciaC())
         Encontrado = true;                        ruptura de la búsqueda
   }
   if ( Encontrado)
      return Indice;
   else
      return -1;                        //la Cola de prioridad está vacía
}
```

La *complejidad* de `Encontrar` es en este caso lineal en el máximo número de prioridades en el peor de los casos, ya que `EsvaciaC` es un método de *complejidad* constante de la clase `Cola`.

La función miembro `ElementoMin` realiza la llamada al método `Encontrar` y si éste retorna una `prioridad` distinta de `-1` entonces la `Tarea` que se busca es el primero de la `Cola` de la posición `prioridad` correspondiente.

```
Tarea ColaPrioridad::ElementoMin()
{
   int prioridad = Encontrar();
   if (prioridad != -1)
      return CP[prioridad].PrimeroC();
   else
      throw " cola de Prioridad Vacía";
}
```

El método `QuitarMin`, es semejante a la función miembro `ElementoMin`, excepto que en este caso se borra del frente de la `Cola` de mayor prioridad.

```
void ColaPrioridad::QuitarMin()
{
   int Prioridad = Encontrar();
   if (Prioridad != -1)
      CP[Prioridad].BorrarC();
   else
      throw "Cola de prioridad Vacía";
```

La complejidad de las operaciones `BorrarC` y `PrimeroC` de una `Cola` es constante, y el proceso de búsqueda de la cola de máxima prioridad es lineal por lo que la complejidad de **`ElementoMin`** y **`QuitarMin`** es lineal.

13.4.4. Cola de prioridad vacía

El método `EsVaciaColaPrioridad`, decide si la *Cola de Prioridad* está vacía realizando una llamada a la función miembro oculta `Encontrar`, decidiendo que está vacía en el caso de que el valor que retorne es -1

```
bool ColaPrioridad::EsVaciaColaPrioridad()
{
   return (Encontrar() == -1);
}
```

La complejidad de este método es lineal en el número máximo de prioridades, ya que realiza una llamada a un método de tiempo de ejecución lineal.

13.5. MONTÍCULOS

El montículo binario, o simplemente montículo, es una estructura abstracta de datos que almacena la información de tal forma que pueden recuperarse en orden. Por ello, se utiliza para implementar estructuras de datos en las que el orden del proceso es esencial como, por ejemplo, las *colas de prioridades*; también para uno de los algoritmos de ordenación más eficientes de compleidad `O(nlog(n))`: *ordenación por montículos* o *HeapSort*. Además de la propiedad del

orden, los montículos se caracterizan por la organización de los datos en un array representando un árbol binario completo[1].

La idea intuitiva de montículo[2] mínimo (máximo) es muy conocida: *una agrupación piramidal de elementos en la que para cualquier nivel el peso de éstos es menor (mayor) que la de los elementos adjuntos del nivel inferior y, por consiguiente, en la parte más alta se encuentra el elemento más pequeño (más grande).*

13.5.1. Definición de montículo

Un montículo mínimo (máximo) binario *de tamaño n se define como un árbol binario casi completo de n nodos, tal que el contenido de cada nodo es menor o igual (mayor o igual) que el contenido de su hijos.*

La definición formal de la estructura montículo está relacionada con los árboles binarios. Así, en la Figura 13.2, se puede observar un árbol binario con todos los niveles completos, excepto el último que puede no estar completo y tener los nodos situados lo más a la izquierda posible, que forma un montículo binario.

Sin embargo, el árbol binario de la Figura 13.3 no es completo, ya que en el último nivel no se han colocado los nodos de izquierda a derecha. Tampoco cumple la propiedad de ordenación de los montículos, el nodo con clave 21 es mayor que uno de sus hijos, con clave 19.

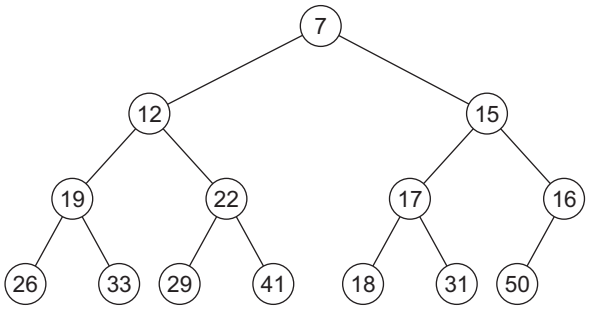


Figura 13.2. Montículo de elementos enteros.

Definición

Un *árbol binario completo* es un árbol con todos los niveles llenos, con la excepción del último nivel que se llena de izquierda a derecha.Tienen la propiedad importante de que la altura de un árbol binario completo de n nodos es [log n].

[1] Un árbol binario casi completo es un árbol binario completamente lleno, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha. El estudio de los árboles binarios se realiza en el Capítulo 16; se adelanta su concepto, por coherencia con la temática del capítulo.

[2] Los montículos pueden ser mínimos o máximos. Ambas estructuras son similares. La única diferencia radica en que en los montículos mínimos la información de un nodo que se encuentra en un nivel cualquiera es menor o igual que la información de los nodos que son hijos suyos (se encuentran en el siguiente nivel). El montículo máximo cumple la condición: " la información de un nodo es mayor o igual que la información almacenada en sus hijos". Se estudian en este capítulo los montículos mínimos.

Los montículos tienen dos propiedades fundamentales: la propiedad de organizarse como un árbol binario y la propiedad de la ordenación. La primera, asegura complejidades en las operaciones con cotas logarítmicas, y además una fácil representación en una estructura tipo vector o array.

La propiedad de ordenación: la clave de cualquier nodo es inferior o igual a la de sus hijos (si tiene hijos). Por consiguiente, el nodo con menor clave debe ser el nodo raíz del árbol, siempre el nodo menor es el raíz, por ello la operación de *buscar mínimo* es directa, se puede implementar con *complejidad constante*, *O(1)*.

Las operaciones definidas con la estructura montículo puede violar alguna de sus propiedades, si esto ocurre será necesario restablecer la condición de montículo.

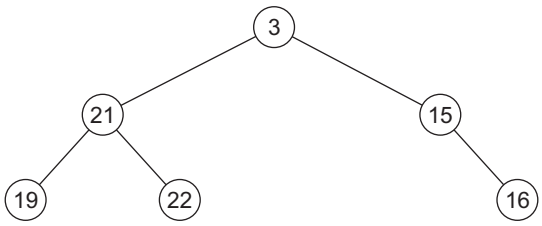


Figura 13.3. Árbol binario no completo.

A tener en cuenta

En un montículo la clave que se encuentra en la raíz es la menor de todos los elementos, sin embargo no existe un orden entre las claves de los elementos que se encuentran en el mismo nivel.

13.5.2. Representación de un montículo

La característica esencial de un árbol binario completo, tener los niveles internos llenos, permite guardar sus claves secuencialmente mediante un vector o array. La raíz del árbol se sitúa en la posición[3] 0, sus hijos en las posiciones 1 y 2; los hijos de éstos en las posiciones 3, 4, 5 y 6 respectivamente, y así sucesivamente. El montículo de la Figura 13.4 consta de 8 nodos que han sido numerados según las posiciones que ocuparán en un almacenamiento secuencial.

Esta disposición natural de las claves de un árbol completo (las posiciones se corresponde con los niveles del árbol, para un nivel de izquierda a derecha) en un array es muy útil ya que permite acceder desde un nodo al nodo padre, al hijo izquierdo y derecho (si los tiene). Un nodo que esté en la posición `i` su nodo padre ocupa la posición `[i/2]`, el nodo hijo izquierdo se ubica en la posición `2*i+1` y el nodo hijo derecho en `(2*i+1)+1`. La representación secuencial del montículo binario de la Figura 13.4.

15	39	18	41	72	37	19	56

[3] Se considera que el índice inferior de un array es 0, si fuera 1 habría que desplazar las posiciones.

A tener en cuenta

La forma secuencial de un montículo de n elementos implica que si `2*i+1` ≥ n entonces `i` no tiene hijo izquierdo (tampoco hijo derecho), y si `(2*i+1)+1` ≥ n entonces `i` no tiene hijo derecho.

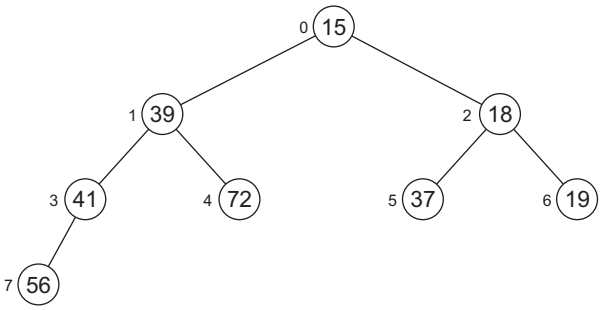


Figura 13.4. Montículo de 8 elementos numerando sus posiciones secuenciales.

13.5.3. Propiedad de ordenación: condición de montículo

La ordenación parcial de los nodos de un montículo se conoce como *condición de montículo*; permite encontrar directamente al elemento mínimo, éste siempre se almacena en la primera posición del array, `v[0]`. Como consecuencia, el montículo es una estructura idónea para representar una *cola de prioridades (siempre que no se necesite además que en el caso de igualdad de prioridades se procesen en el orden de llegada como es el caso de múltiples aplicaciones).*

Definición

La *condición de montículo* establece que cada nodo debe ser menor o igual que los nodos hijos. Esta condición debe satisfacerla todos los nodos del montículo, en definitiva, cada nodo será menor que sus descendientes.

Utilizando la representación secuencial de los montículos la propiedad de ordenación parcial de las claves se expresa:

```
v[i] ≤ v[2*i+1]                          ∀i = 0 .. n/2
v[i] ≤ v[2*i+2]
```

El montículo de la Figura 13.2 se representa secuencialmente:

7	12	15	19	22	17	16	26	33	29	41	18	31	50

Cualquier nodo cumple la condición; así `v[4] == 22`, tiene como hijos a `v[9]` y `v[16]`, cuyos respectivos valores son `29` y `41`, menores que `22`.

13.5.4. Operaciones en un montículo

Se consideran las operaciones básicas `Innsertar`, `BuscarMinimo`, `EliminarMinimo` (además de `CrearMontículo` y `EsVacio`); las que modifican el montículo originan, posiblemente, una violación de la *condición de montículo* por lo que son ncesarias operaciones auxiliares para recomponer el montículo; éstas recorren la estructura, de arriba abajo (de la raíz al último nivel), o bien en orden inverso.

Así, el montículo de la Figura 13.4 consta de `8` claves, la inserción de una nueva clave, `32` por ejemplo, se realiza en la posición `8`, `v[8] == 32`:

15	39	18	41	72	37	19	56	**32**

La condición de ordenación es violada, la clave de la posición `3` es mayor que la clave del hijo derecho, que acaba de ser insertado, `v[3] > v[8]`.

Declaración de la clase Montículo

La clase `Monticulo` tiene como atributos protegidos el array `v` que almacena la información, así como `NdatosMax, NdatosActual` que contienen el número de datos máximo que se puede contener el montículo y el número de datos actuales respectivamente. En la sección pública se encuentran las funciones miembro `Insertar`, `BuscarMinimo`, y `EliminarMínimo` `Esvacio` del TDA `Monticulo`, así como el destructor por defecto y dos constructores que permiten construir montículos con un número de datos máximo o bien a medida del usuario del `Montículo`. En la sección privada se encuentran funciones miembro auxiliares de la clase Montículo.

```
#define MAXDATOS 1600
typedef int Clave;
class Monticulo
{
   protected:
      Clave *v;
      int NdatosMax, NdatosActual;
   public:
      Monticulo();            // constructor por defecto
      Monticulo(int n);       //constructor sobrecargado
      void Insertar(Clave e); //añade al montículo
      Clave BuscarMinimo();   // retorna el elemento menor del montículo
      Clave EliminarMinimo(); //elimina el primer elemento
      ~Monticulo(){}          // destructor por defecto
      bool Esvacio();         // está vacío
   private:
   Clave & Monticulo::operator[](int i);
      int Padre (int i);      // padre de i
      int HijoIzq(int i);     // hijo izquierdo de i
      int HijoDer(int i);     // hijo derecho de i
      void flotar(int i);
      bool Lleno();           // está lleno
      void criba (int raiz);
};
```

Los dos constructores de la clase establece una capacidad de almacenamiento e inicializan los atributos que son los índices del `Montículo`. El constructor por defecto reserva memoria para un número de datos `MAXDATOS`, previamente declarado.

```
Monticulo::Monticulo()
{
   NdatosMax = MAXDATOS;
   v = new Clave[NdatosMax];
   NdatosActual = 0;
}
```

El constructor con parámetro `n` reserva memoria para el vector a medida del usuario.

```
Monticulo::Monticulo( int n)
{
   NdatosMax = n;
   v = new Clave[NdatosMax];
   NdatosActual = 0;
}
```

13.5.5. `insertar`

La operación añade una clave al montículo, incrementa su tamaño (`NdatosActual`) y en el *hueco* se coloca la clave. De esta forma, no se viola la estructura del montículo; ahora bien, también es necesario comprobar la *condición de ordenación.* En el caso de que al situar la nueva clave no se viole la *condición de ordenación*, termina la operación. Por ejemplo, si de nuevo se considera el montículo[4] de la Figura 13.4 consta de `8` claves, la inserción de la clave 50, se realiza en la posición `8`, `v[8] == 50`, y termina la operación:

15	39	18	41	72	37	19	56	**50**

Para verificar la *condición de ordenación*, se ha de comprobar que la clave del nodo *padre*, sea menor o igual que la nueva clave insertada; en caso negativo el *padre* (es decir, *padre* > *clave*) baja a la posición de la clave (*hueco)* y, naturalmente, ésta sube. Entonces, el proceso se repite comparando la clave del nodo padre actual con la nueva clave, y si la clave almacenada en *padre* es mayor que *clave* se produce otro intercambio. En definitiva, la clave *sube por el árbol* hasta encontrar la posición que le corresponde. La búsqueda de la posición de inserción termina cuando la clave del *padre* es menor o igual que la nueva clave (no se viola la *condición de ordenación*), o bien se alcanza la raíz.

Al montículo binario de la Figura 13.5 se quiere añadir la clave `23`; en primer lugar se hace un *hueco* (posición `8` del array) en el árbol.

El nodo padre, `55`, es mayor que `23`, entonces viola la *condición de ordenación*. Baja la clave `55` al *hueco* y sube `23` al siguiente nivel del árbol, en busca de la posición de inserción. A continuación, se compara la clave `42` (nuevo nodo padre) con `23`, como es mayor, baja `42` al *hueco* dejado anteriormente por `55`, y de nuevo sube `23` al siguiente nivel del árbol. El proceso continúa, se compara `26` con `23` y al ser mayor `26` baja al *hueco*. Se ha alcanzado la raíz

[4] Con el fin de facilitar la comprensión, los nodos de los montículos son valores enteros.

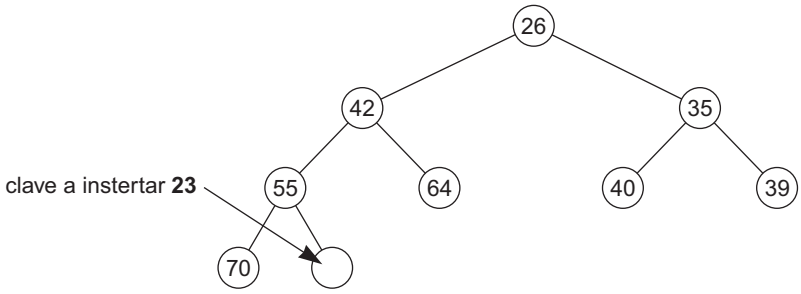


Figura 13.5. Montículo al que se añade la clave 23.

del árbol, el proceso termina con 23 como nueva raíz del montículo. La Figura 13.6 muestra las comparaciones que se realizan hasta encontrar la posición donde insertar la clave 23.

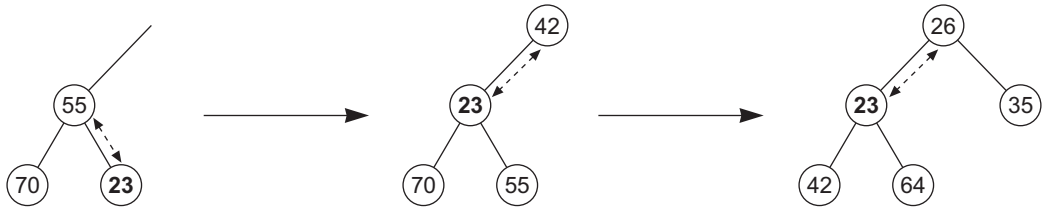


Figura 13.6. Reconstrucción del montículo moviendo hacia arriba la nueva clave.

El número máximo de pasos que se realizan al insertar una clave en el montículo ocurre cuando la clave es el nuevo valor mínimo. Éste coincide con el número de niveles del árbol (para un montículo de n nodos, [log n]); entonces el tiempo necesario, en el peor de los casos, para realizar la inserción es O([log n]).

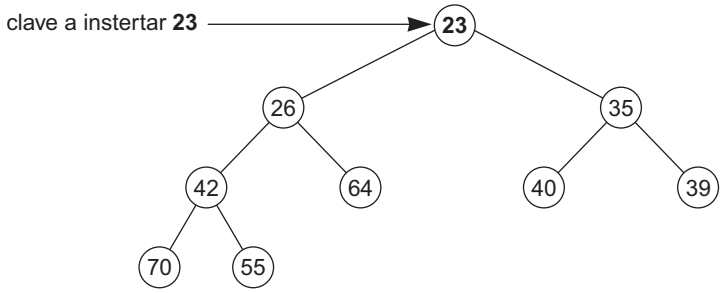


Figura 13.7. Montículo después de insertar 23.

A recordar

La inserción de una clave en el montículo se realiza en la siguiente posición libre del array y, a continuación, se hace que *flote hacia arriba*, hasta encontrar la posición adecuada en el árbol. La complejidad de la operación es logarítmica, O([log n]).

Implementación

Primero se codifican los métodos privados que devuelven las posiciones de nodo *padre, hijo izquierdo y hijo derecho*:

```
int Monticulo::Padre (int i)
{
   return ( i - 1) / 2;
}

int Monticulo::HijoIzq(int i)
{
   return (2 * i + 1);
}

int Monticulo::HijoDer(int i)
{
   return (2 * i + 1) + 1;
}
```

El método `flotar()`realiza el *movimiento hacia arriba*, en busca de la posición de inserción. Tiene como argumento el índice, `i`, correspondiente al *hueco* donde, inicialmente, se encuentra la nueva clave. La comparación entre dicha clave y la del nodo padre, se realizan muy eficientemente, con acceso directo al *padre*. El bucle de búsqueda que realiza el proceso termina cuando la clave del nodo padre es menor o igual `(v[padre(i)] <= nuevaClave)`, o bien se *ha subido* hasta la raíz `(i == 0)`.

```
void Monticulo::flotar(int i)
{
   Clave Nueva = v[i];                  // se almacena la clave a flotar
   while ((i > 0) && (v[Padre(i)] > Nueva))
   {
      v[i] = v[Padre(i)];               // baja el padre al hueco
      i = Padre(i);                     // sube un nivel en el árbol
   }
   v[i] = Nueva;                        // sitúa la clave en su posición
}
```

La operación busca en un árbol completo por lo que su tiempo es O([`log n`]).

El método `Insertar()` comprueba que no esté lleno el montículo mediante el método oculto `Lleno`, asigna la clave en la primera posición libre (*hueco*), llama a `flotar()`para que restablezca la *condición de ordenación* del montículo e incrementa el contador de número de claves almacenadas en el montículo.

Un montículo está lleno si el numero de datos actual coincide con el número máximo de datos reservados

```
bool Monticulo::Lleno()
{
   return (NdatosMax == NdatosActual);
}

void Monticulo::Insertar(Clave c)
{
```

```
   if (!Lleno())
   {
      v[NdatosActual] = c;
      flotar(NdatosActual);
      NdatosActual++;
   }
   else
   throw " Montículo lleno";
}
```

Esta operación llama al método `flotar` que es de complejidad logarítmica, por lo que la operación es de complejidad O([log n]).

13.5.6. Operación `buscar mínimo`

La propiedad de ordenación parcial de los montículos asegura que el elemento mínimo se encuentra en la raíz del árbol, que en el almacenamiento secuencial se corresponde con la primera posición del vector, `v[0]`.

```
Clave Monticulo:: BuscarMinimo()
{
   if (Esvacio())
      throw "Acceso a montículo vacío";
   return v[0];
}
```

La operación accede directamente al elemento, es de complejidad constante *O(1)*.
El método `Esvacio()`que verifica si el montículo se ha quedado vacío.

```
bool Monticulo::Esvacio()
{
   return NdatosActual==0;
}
```

13.5.7. `eliminar mínimo`

La operación implica eliminar la clave que se encuentra en la raíz y, en consecuencia, reducir el número de elementos. Ahora bien, la estructura resultante debe seguir siendo un montículo, manteniendo la ordenación parcial de las claves y la forma estructural.

El elemento mínimo, `v[0]`, se extrae del array, quedando un *hueco* en esa posición. El número de elementos disminuye y para que la estructura siga siendo un montículo, el último elemento (hoja del árbol más a la derecha) pasa a ser el raíz. La Figura 13.8 muestra este primer paso en la eliminación del elemento mínimo.

Falta por analizar si la ordenación parcial, *condición de ordenación*, se mantiene. Como los últimos nodos son mayores que los del anterior nivel, es necesario restablecer la *condición de ordenación*, para lo cual se procede a la inversa que en la inserción: se deja *hundir la clave por el camino de claves mínimas*.

El algoritmo compara el raíz con el menor de sus hijos, si la raíz es mayor se intercambia con el menor de los hijos; así en el montículo de la Figura 13.8*c*), la clave `41` se intercambia

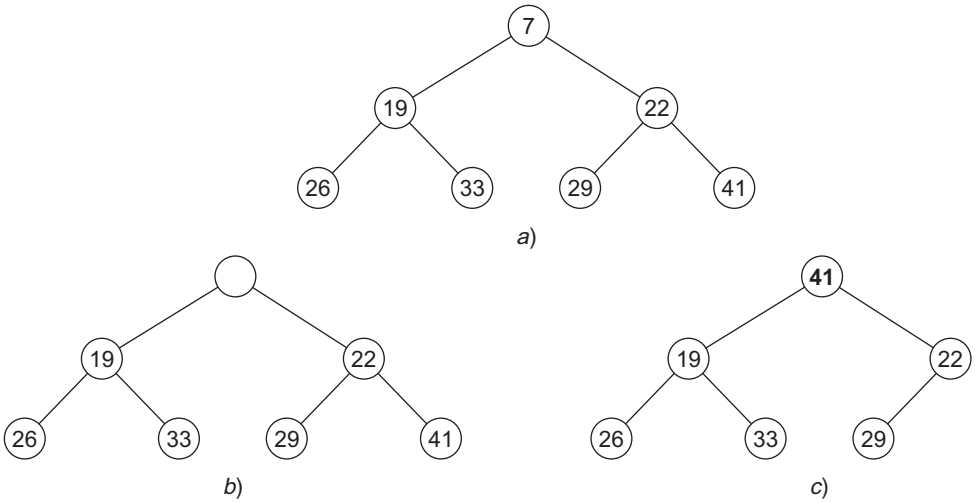


Figura 13.8. *a*) Montículo al que se elimina el mínimo (`7`); *b*) *hueco* dejado, *c*) sube el último elemento (`41`).

con `19`. Esto puede hacer que de nuevo se viole la *condición de ordenación* del montículo, pero en el siguiente nivel. Entonces se procede de igual manera, se compara la clave con el menor de sus hijos y si ésta es mayor se intercambia con el menor. En el ejemplo de la Figura 13.8*c*), el menor de los hijos actuales de `41` es `26`, y se intercambian. Se observa que la clave que inicialmente subió a la raíz baja por *el camino de las claves mínimas*; el proceso continúa hasta que no se viole la condición de montículo, o bien llegue al último nivel.

La Figura 13.9 muestra los intercambios que se realizan en el montículo 13.8c hasta situar la clave `41` en la posición adecuada, en este caso como nodo hoja.

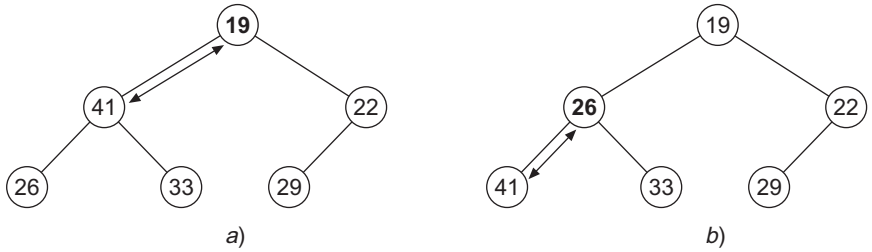


Figura 13.9. Proceso que restablece la *condición de ordenación* parcial del montículo. *a*) Intercambio de `41` con `19`; *b*) intercambio de `41` con `26`.

A recordar

La función hundir la clave situada en la raíz hasta encontrar la posición en la que no se viole la condición de montículo, realiza un máximo de intercambios igual al número de niveles menos uno, por lo que la complejidad que tiene es *logarítmica*, O([`log` n]).

Implementación

El método `criba()` implementa el algoritmo, se pasa como argumentos el índice del elemento (`raiz`) que se ha de dejar hundir.

```
void Monticulo::criba (int raiz)
{

   bool esMonticulo;
   int hijoi = HijoIzq(raiz), hijod= HijoDer(raiz);
   int hijom = hijoi;              // el candidato a hijo menor es hijoi
   esMonticulo = false;
   while ((hijom <= NdatosActual-1) && !esMonticulo)
   {
                                   // determina el índice del hijo menor
      if (hijom < (NdatosActual - 1))              // único descendiente
         if (v[hijom] < v[hijod])
               hijom = hijod;
                               // compara raiz con el menor de los hijos
         if (v[hijom] < v[raiz])
         {
            Clave c = v[raiz];
            v[raiz] = v[hijom];
            v[hijom] = c;
            raiz = hijom; // continua rama de claves mínimas
         }
         else
         esMonticulo = true;
   }
}
```

El método `eliminarMinimo()` realiza la operación: extrae la clave raíz y llama a `criba()` para restablecer la *condición de ordenación.*

```
Clave Monticulo::EliminarMinimo()
{
   if (Esvacio())
      throw "Acceso a montículo vacío";
   else
   {
      Clave menor;
      menor = v[0];
      v[0] = v[NdatosActual - 1];
      NdatosActual--;
      criba(0);
      return menor;
   }
}
```

13.6. ORDENACIÓN POR MONTÍCULOS (*HEAPSORT*)

Una de las aplicaciones que se ha dado a la estructura del montículo es la de ordenar los n elementos de un vector. Haciendo uso de las operaciones definidas anteriormente se puede diseñar un algoritmo de ordenación; los pasos a seguir:

1. Crear el montículo vacío `v[]`.
2. Insertar cada uno de los n elementos en el montículo. Se utiliza la operación `insertar`.
3. Extraer cada uno de los elementos, llamando a la operación `eliminarMinimo`, y asignarlos al array auxiliar `w[]`.

Un problema de este algoritmo es la necesidad de un segundo vector en el que almacenar los elementos que se extraen. Con el fin de realizar la ordenación en el mismo vector, sin necesidad de memoria adicional, se sigue una estrategia conocida como algoritmo de *ordenación HeapSort*, o simplemente *ordenación por montículo*.

La operación `EliminarMinimo` del montículo retira el elemento menor y disminuye el número de elementos. El algoritmo *HeapSort* evita el uso de un segundo vector utilizando, en cada pasada, la actual última posición para guardar el elemento eliminado. La Figura 13.10*b*) muestra el montículo después de eliminar la clave mínima y ser guardada en la última posición del array; la operación ha reconstruido el montículo, como consecuencia el nuevo mínimo se corresponde con la clave `22`. En la Figura 13.16*c*) se ha aplicado de nuevo la operación, el mínimo se guarda en la actual última posición, posición `4`; la operación reconstruye el montículo. con cuatro pasadas más el array queda ordenado, aunque en orden decreciente.

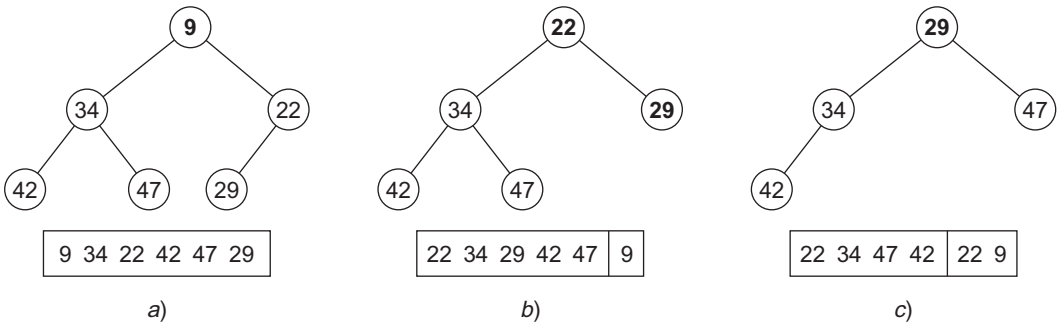


Figura 13.10. *a*) Montículo de 6 elementos. *b*) Montículo después de eliminar mínimo. *c*) Después de volver aplicar eliminar mínimo.

Los algoritmos de ordenación interna siempre ordenan en orden creciente. Para que el algoritmo del montículo lo haga así, simplemente hay que usar el *montículo maximal*. Éste cambia la *condición de ordenación*, de tal forma que la clave del nodo padre sea mayor o igual que la de sus hijos.

A tener en cuenta

El método de ordenación por montículos ordena *descendentemente* ya que siempre intercambia el elemento menor, `v[0]` con el último elemento del montículo actual. Para una ordenación ascendente se considera *un montículo maximal*, simplemente se invierte el sentido de las desigualdades (cambiar < por > y vice versa).

13.6.1. Algoritmo

El algoritmo empieza con un vector de n elementos que no cumple la condición de montículo. Lo primero que hace el algoritmo es construir el montículo de partida. Para ello, considera el montículo como una estructura recursiva. Los nodos del montículo, del último nivel del árbol, se pueden considerar que cada uno es un submontículo de 1 nodo. Subiendo un nivel en el árbol, cada nodo es la raíz de un árbol que cumple la *condición del montículo*, excepto, quizá, en la raíz (su rama izquierda y derecha cumplen la condición ya que se está *construyendo de abajo a arriba*); entonces, al aplicar el método `criba()` (reconstruye el montículo *hundiendo* la raíz) se asegura que cumple la *condición de ordenación* y ya es submontículo. El algoritmo va subiendo de nivel en nivel, construyendo tantos submontículos como nodos tiene el nivel, hasta llegar al primer nivel en el que sólo hay un nodo que es la raíz del montículo completo.

La Figura 13.11*a* se corresponde con el array inicial en la forma de árbol completo. En la Figura 13.11*b* todos los subárboles del penúltimo nivel son *montículos maximales*, se puede observar los intercambios realizados para que cumplan la propiedad.

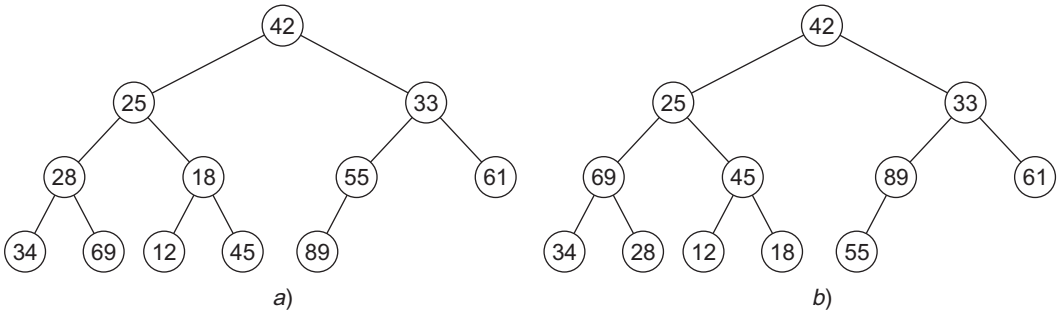


Figura 13.11. *a*) Montículo inicial. *b*) Montículo después de construir submontículos maximales en penúltimo nivel.

En la Figura 13.12*a*) se ha subido un nivel, se ha construido dos submontículos maximales dejando hundir la correspondiente clave raíz (`25` y `33` respectivamente). Para terminar, la Figura 13.12*b*) muestra el montículo maximal completo; esta última reconstrucción ha hundido la clave `42` y ha subido la raíz la clave `89` que es el valor mayor.

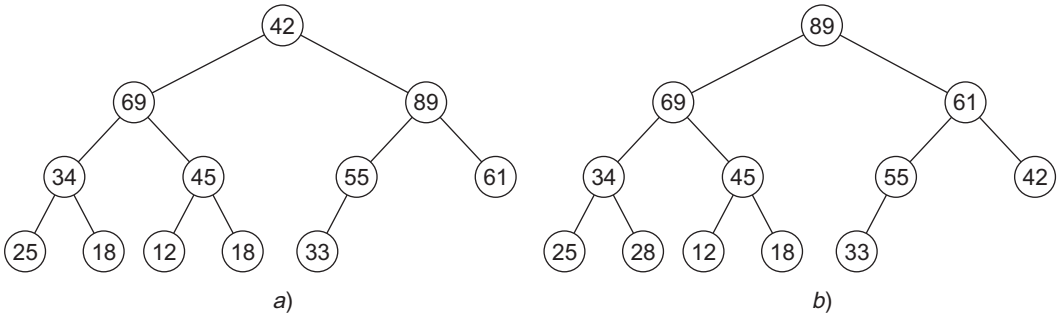


Figura 13.12. *a*) Reconstrucción de montículo en segundo nivel. *b*) Montículo maximal reconstruido de abajo a arriba.

Según esto, los pasos que sigue el algoritmo de ordenación por montículos o *HeapSort* para un array de `n` elementos (`0 .. n-1`) son:

1. Construir un montículo inicial con todos los elementos del vector:

   `v[0], v[1] , ..., v[n-1]`

2. Intercambiar los valores de `v[0]` y `v[n-1]`.
3. Reconstruir el montículo con los elementos `v[0], v[1], ...v[n-2]`.
4. Intercambiar los valores de `v[0]` y `v[n-2]`.
5. Reconstruir el montículo con los elementos `v[0], v[1], ...v[n-3]`.

Es un proceso iterativo que partiendo de un montículo inicial, repite *intercambiar* los extremos, decrementar en `1` la posición del extremo superior y reconstruir el montículo del nuevo vector. En seudocódigo:

```
OrdenacionMonticulo(Tvector v, Entero n)

inicio
   <Construir monticulo inicial (v, 0, n)>
   desde k ← n-1 hasta 1 hacer
      intercambiar (v[0], v[k])
      reconstruir monticulo (v, 0, k - 1)
   fin_desde
fin OrdenacionMonticulo
```

El problema de construir el montículo inicial y el de reconstruir se resuelve con la función (método) `criba()` que constituye la base del algoritmo.

13.6.2. Codificación

Se considera un *montículo maximal* para que de esa forma los elementos queden en orden ascendente. La función `criba1()`, trabaja como si fuera con un montículo maximal, pero tiene como argumentos: el array `v` que se ordena, el índice de la `raíz` o primer elemento, así como el índice el `último` elelemento.

```
void criba1( Clave v[], int raiz, int ultimo)
{
   bool esMonticulo;
   int hijo = 2 * raiz + 1;
   esMonticulo = false;
   while (hijo <= ultimo && !esMonticulo)
   {                              // determina el índice del hijo mayor
      if (hijo < ultimo )         // único descendiente
         if (v[hijo] < v[hijo + 1])
            hijo++;
                                 // compara raiz con el mayor de los hijos
      if (v[hijo] > v[raiz])
      {
         Clave c = v[raiz];
         v[raiz] = v[hijo];
         v[hijo] = c;
```

```
            raiz = hijo;          // continúa por la rama de claves mínimas
         }
         else
         esMonticulo = true;
         hijo = 2*raiz+1;
      }
   }
```

Ordenación

Para construir el montículo inicial se llama a `criba1()` pasando, sucesivamente, como segundo argumento, la raíz de los subárboles desde el penúltimo nivel del árbol hasta el raíz (posición `0`). En definitiva:

`Criba1(v, j, n-1)` para todo `j = (n-1)/2, n/2 - 1, ..., 0.`

El bucle que construye el montículo:

```
for (j = (n - 1) / 2; j >= 0; j--)
criba1(v, j, n-1);
```

En `criba1()` reside todo el trabajo de la realización del algoritmo de ordenación por montículos. Por último, la codificación del método de ordenación:

```
void ordenacionMonticulo(Clave v[], int n)
{ // el vector tiene n datos en las posiciones 0,1,…n-1
   int j;
   for (j = (n - 1) / 2; j >= 0; j--)
      criba1(v, j, n-1 );
   for (j = n - 1; j >= 1; j--)
   {
      Clave c;
      c = v[0];
      v[0] = v[j];
      v[j] = c;
      criba1(v, 0, j-1 );
   }
}
```

13.6.3. Análisis del algoritmo de ordenación por montículos

El tiempo del algoritmo de ordenación *HeapSort* depende de la complejidad del método `criba2()`. Para determinar ésta, supóngase que el árbol binario que representa al montículo está completo, y que el número de niveles de que consta es `k`. Además, el nivel que se corresponde con el raíz es el cero, por lo que los niveles de que constará de `0` a `k-1 (0, 1, 2, 3, ... k-1)`.

El número de elementos de cada nivel es potencia de 2: nivel `0`, $2^0 = 1$; nivel `1`, $2^1 = 2$, $2^2 = 4$, $2^3 = 4$, ... 2^{k-1}.

Entonces, el total de elementos = $1 + 2 + 2^2 + 2^3 + \ldots + 2^{k-1}$

Si *n* es el número de elementos a ordenar, aplicando la fórmula de la suma de los términos de una progresión geométrica de razón 2 se ha de cumplir:

$$n = \frac{2 * 2^{k-1} - 1}{2 - 1} = 2^k - 1$$

tomando logaritmos se obtiene $k = log_2\ n + 1$.

`criba1()` realiza, en el peor de los casos, tantas iteraciones como niveles tiene el árbol menos 1, hasta que llega al penúltimo nivel ($log_2 n$). Por ello, se puede concluir que la complejidad es *O(log n)*, *complejidad logarítmica*.

La ordenación consta de dos bucles, el primero:

```
for (j = (n –1) / 2; j >= 0; j--)
```

construye el montículo inicial y se ejecuta *n/2* veces, por lo que su complejidad:

O(n/2 · log n)

El segundo bucle: `for (j = n - 1; j >= 1; j--)` se ejecuta *n-1* veces y, por consiguiente, su complejidad:

O((n-1) · log n)

Se puede concluir que la eficiencia del método de ordenación:

O(n/2 · log n) + O((n-1) · log n)

por las propiedades de la notación O, se puede afirmar que la complejidad del método de ordenación *HeapSort*:

O (n · log n)

Este método de ordenación es de los más eficientes, tiene la misma complejidad que el método *mergeSort* o el *quicksort* en el caso medio

13.7. COLA DE PRIORIDADES EN UN MONTÍCULO

La estructura montículo es la más eficiente para implementar una cola de prioridades, siempre que se permita que a igualdad de prioridades no sea necesario salir en el orden en que entraron. A continuación, se escribe su implementación utilizando las operaciones básicas del montículo.

```
typedef Clave Tarea;          //la Tarea es la clave del montículo
class ColaPrioridad
{
   protected:
      Monticulo Cp;
   public:
      ColaPrioridad() {Monticulo Cp; }             // constructor
      ColaPrioridad(int n){Monticulo Cp(n);}
      bool EsVaciaColaPrioridad(){return Cp.Esvacio();}
      void InserEnPrioridad(Tarea e){Cp.Insertar(e);}
      void QuitarMin(){Cp.EliminarMinimo();}
```

```
        Tarea ElementoMin(){ return Cp.BuscarMinimo();}
        ~ColaPrioridad(){}                // destructor por defecto
};
```

13.8. MONTÍCULOS BINOMIALES

Los montículos binarios son estructuras de datos que contienen n elementos permiten: buscar el menor elemento en tiempo constante `O(1)`, eliminar el menor elemento o insertar un nuevo elemento en tiempo `O(log(n))`, pero necesitan tiempo lineal `O(n)` para fusionar dos montículos que tengan entre los dos *n* elementos. Los montículos binomiales pemiten realizar la fusión de dos de estos montículos en tiempo `O(log(n))`, manteniendo el mismo tiempo `O(log(n))` para el borrado del menor elemento, y tiempo amortizada constante `O(1)` para la inserción.

Se definen los árboles binomiales como árboles generales que cumplen la siguiente condición: el *i*-ésimo arbol binomial B_i con `i` mayor o igual que cero tiene un nodo raíz así como `j` hijos. Cada uno de estos `i` hijos es, a su vez, un árbol binomial B_i. La Figura 13.13 muestra los 5 primeros montículos binomiales. Hay que observar que cada montículos binomial B_i contiene exactamente 2^i nodos.

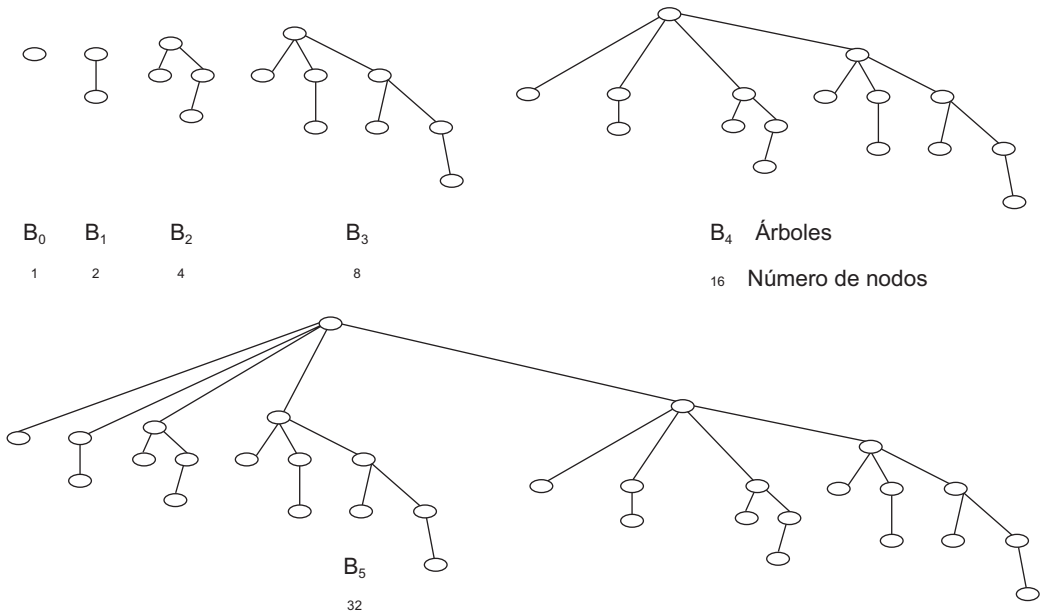


Figura 13.13. Árboles binomiales. Cada árbol binomial contiene exactamente una potencia de dos elementos.

Un montículo binomial es una colección de árboles binomiales. Cada árbol binomial debe tener diferente número de elementos y, además, poseer la propiedad del montículo: el valor almacenado en cada nodo debe ser menor o igual que el de sus hijos. En los montículos binomiales es conveniente almacenar la información de cada una de las raíces en una lista doblemente enlazada, para que la inserción y borrado de una raíz resulte sencilla (véase la Figura 13.14).

La búsqueda del elemento más pequeño de una cola binomial puede hacerse en tiempo logarítmico. Basta con recorrer la primera lista doblemente enlazada que une las raíces de los

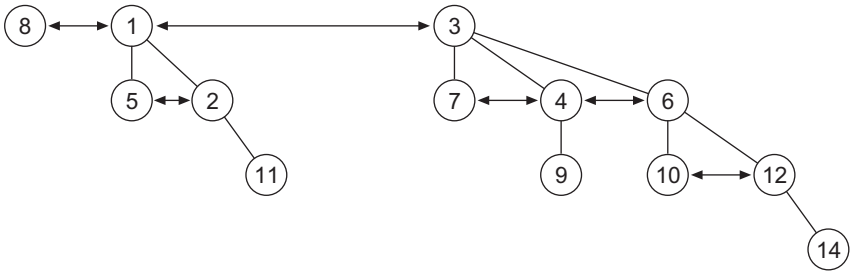


Figura 13.14. Representación de una cola binomial como un bosque en el que cada una de las raíces es una lista doblemente enlazada.

árboles binomales. Esta operación puede hacerse en tiempo `O(log(n))` y que cola binomial tienen como máximo tiene `log(n)` árboles binomiales.

La fusión de dos colas binomiales puede realizarse de una manea sencilla uniendo entre sí los árboles de la misma altura, según el siguiente algoritmo que se explica con un ejemplo parecido al de la suma en binario.

Para fusionar dos colas binomales M1 y M2, se determina, en primer lugar, los árboles B_0 de M1 y M2, si no hay ninguno se avanza. Si hay sólo uno, se incluye en M3. Si hay dos B_0 se unen en un árbol B_1 dejando como raíz la clave menor. Se determinan ahora los posibles árboles B_1 que hay tanto en M1 como en M2 como en M3. Si sólo hay uno se deja en M3. Si hay dos se forma un árbol B_2 cuya raíz es la más pequeña de las raíces agregándolo a M3. Si hay tres se deja un árbol B_1 en M3 y se forma con los otros dos árboles un B_2 que es añadido a M3 y así sucesivamente. Esta operación de árboles es similar a la de la suma en binario.

```
  0 1 1 0 1 1 0
  1 0 1 0 1 1 1
---------------
1 0 0 0 1 1 0 1
```

Por ejemplo, en la Figura 13.15 se representa la suma de dos montículos binomiales M1 y M2 representados en orden inverso.

B_3	B_2	B_1	B_0
	1	1	0
	1	1	1
1	1	0	1

Se pone el primer B_0 en M3. Se fusionan dos B_1 dando lugar a un B_2. Se deja un B_2 en M3 y se fusionan dos B_2 para dar lugar a un B_3 que se deja en M3 (véase la Figura 13.15).

De esta forma se tiene que la fusión de montículos binomiales necesita tiempo `O(log(n)`.

Teniendo en cuenta que añadir un elemento puede considerarse como una fusión de dos montículos binomiales, uno de ellos con un solo elemento se tiene que la inserción es un caso particular de la fusión y por tanto de su misma complejidad, si bien en tiempo medio puede demostrarse que es de orden `O(1)`.

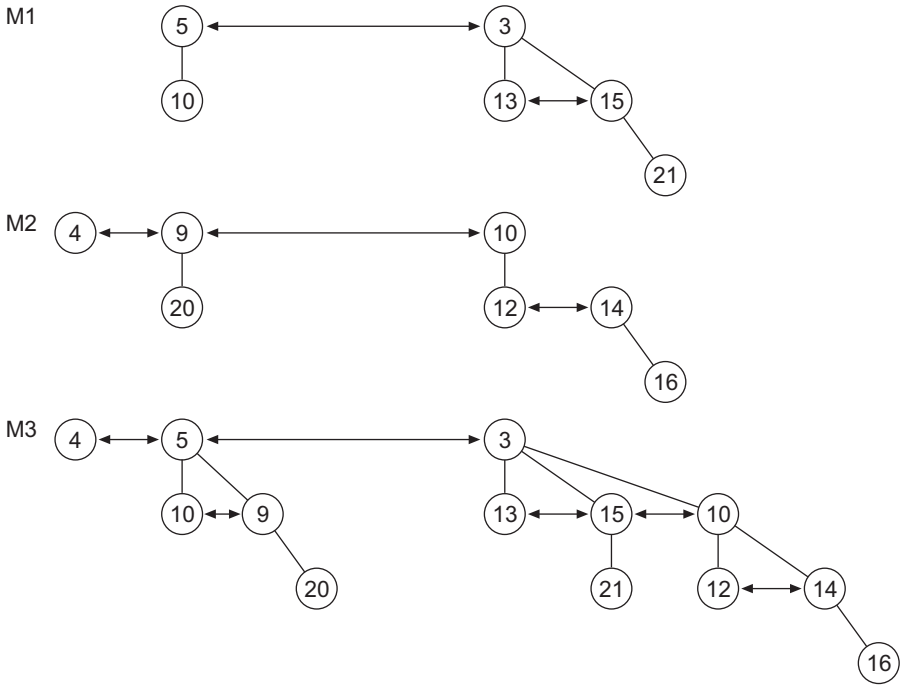


Figura 13.15. Fusión de montículos binomiales.

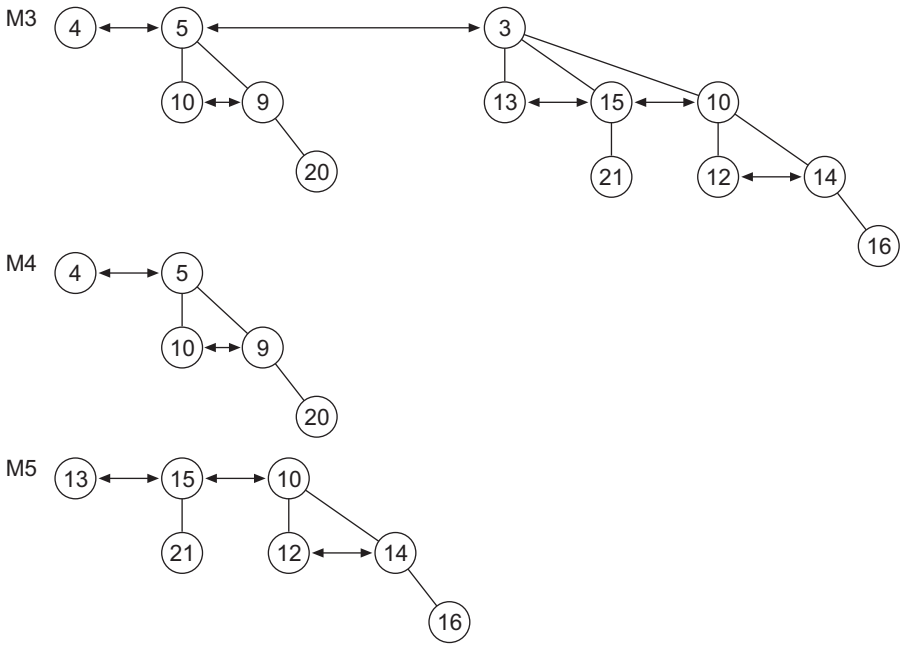


Figura 13.16. Eliminación del mínimo. Montículos del borrado.

La operación de eliminación del mínimo, comienza con la búsqueda de la clave menor en la lista doblemente enlazada que enlaza las raíces de los árboles binomiales que forman el montículo binomial. Una vez encontrada se forman dos montículos binomiales. El primero de ellos contiene todos los árboles binomiales en los que no se encontra la clave mínima y el otro contiene los árboles binomiales resultantes de la eliminación de la raíz del árbol binomial que contiene la clave mínima. Posteriormente, se procede a la fusión de ambos montículos. El ejemplo de las Figuras 13.16 y 13.17 ilustra el borrado de la clave mínima del montículo binomial M3. La Figura 13.16 presenta los dos montículos binomiales M3 y M4 que surgen cuando se elimina la clave 3. La Figura 13.17 muestra el resultado de la fusión de M4 y M5.

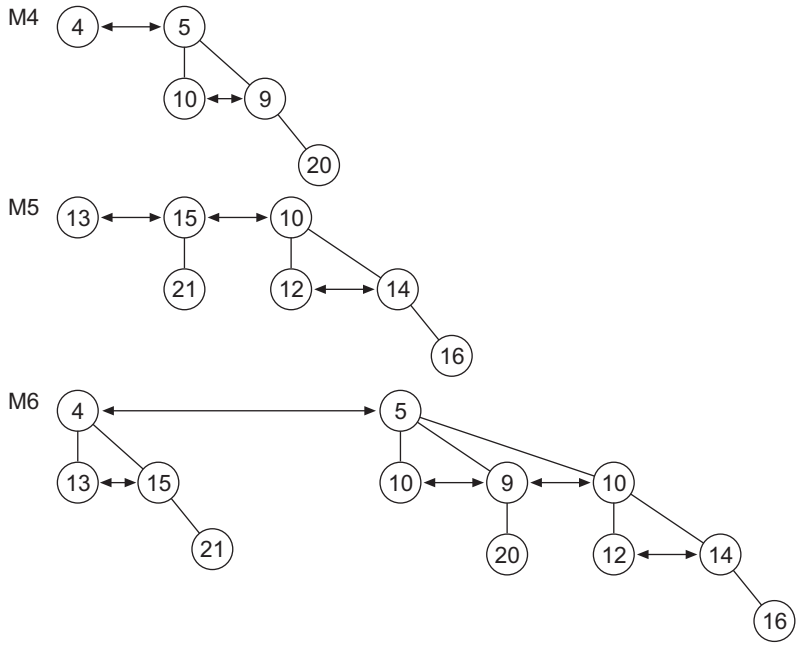


Figura 13.17. Fusión de dos montículos.

RESUMEN

Las *colas de prioridades* son estructuras de datos que tienen la propiedad de que siempre se accede al *mínimo elemento*. Una cola de prioridad puede asemejarse a una estructura de n colas, tantas como prioridades se quieran establecer en la estructura. La prioridad asignada a cada elemento es la clave de ordenación parcial de los elemento que forman parte de una *cola de prioridades*. Las operaciones básicas que tiene esta estructura son: *InserEnPrioridad*, *elemento mínimo*, *quitar mínimo*. La primera añade un elemento a la estructura en el orden que le corresponde según la prioridad y a igual prioridad el último. Las otras dos operaciones acceden al elemento mínimo, en cuanto a la clave, que en realidad es el que tiene mayor prioridad.

En el capítulo se han representado las colas de prioridades con una estructura multienlazada formada por un array de tantas colas como prioridades. A su vez, cada cola implementada con una lista enlazada. También con un `Vector` ordenado respecto a la prioridad de cada elemento.

Tradicionalmente, las colas de prioridades se han implementado con una estructura de árbol binario, llamada *montículo binario*. La definición que se ha dado a esta estructura se corresponde con los montículos minimales los cuales tienen en la raíz el elemento mínimo, también existe la estructura de montículo maximal que en vez de tener el mínimo en la parte más alta tienen el máximo. Las operaciones definidas sobre los montículo: *insertar*, *buscar mínimo* y *eliminar mínimo* se han implementado representado el montículo en un array (índice `0` primera posición), de tal forma que siendo `i` el índice de un nodo, `2*i+1` y `(2*i+1)+1` son los índices del nodo hijo izquierdo y derecho respectivamente.

Como aplicación del montículo maximal, el capítulo desarrolla el algoritmo de ordenación *HeapSort*, también llamado *ordenación por montículos*. Se analiza la complejidad del algoritmo, siendo ésta `O(n*log)`, por tanto, está entre los algoritmos de ordenación más eficientes y además fácil de implementar.

En este capítulo se han introducido los montículos binomiales como estructuras de datos avanzadas para el tratamiento eficiente de las *colas de prioridades*.

EJERCICIOS

13.1. Describir un algoritmo para implementar la operación `cambiar()` que sustituya la clave del elemento *k* en un montículo.

13.2. Demostrar la certeza de la siguiente afirmación: en un árbol binario completo de n claves, hay exactamente n/2 hojas.

13.3. Dibujar un montículo binario maximal a partir de un montículo vacío, al realizar las siguientes operaciones: `insertar(16)`, `insertar(50)`, `insertar(20)`, `insertar(60)`, `eliminarMinimo()`, `insertar(70)` y, por último, `insertar(30)`.

13.4. Considerar el árbol completo de la Figura 13.18; suponiendo que se guarda secuencialmente en el array `v[]`, encontrar el montículo que se forma llamando `insertar()` de la estructura montículo, transmitiendo en cada llamada la clave `v[k]`, para valores `k = 0, 1, ... n-1`; *n* es el número de nodos.

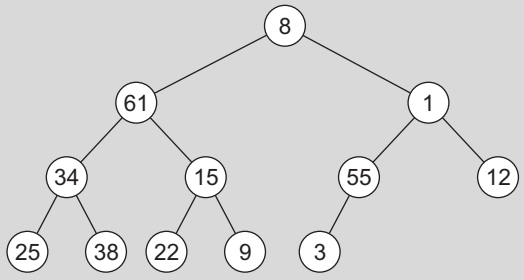


Figura 13.18. Árbol binario completo de 12 nodos.

13.5. Mostrar el resultado del algoritmo de ordenación por montículos sobre la entrada: `18`, `11`, `22`, `33`, `11`, `34`, `44`, `2`, `8`, `11`.

13.6. Diseñar un algoritmo para que dados dos montículo binarios se mezclen formando un único montículo. ¿Qué complejidad tiene el algoritmo diseñado?

13.7. Suponer que se quiere añadir la operación `eliminar(k)`, con el objetivo de quitar del montículo el elemento que se encuentra en la posición *k*. Diseñar un algoritmo que realice la operación.

13.8. En un montículo minimal diseñar un algoritmo que encuentre el elemento con mayor clave. ¿Qué complejidad tiene el algoritmo diseñado?

13.9. Hacer un seguimiento de inserción de las claves 1, 2, 3, 4 ... 15 en un montículo binomial.

PROBLEMAS

13.1. Escribir un programa que compare la eficiencia de los métodos de ordenación *HeapSort* y *Quicksort*, para:

a) 1.600 elementos ordenados ascendentemente.
b) 1.600 elementos ordenados descendentemente.
c) 1.600 elementos aleatorios.

13.2. Implemente el algoritmo escrito en el Ejercicio 13.2 para la operación *cambiar* un elemento que ocupa la posición *k*.

13.3. La universidad de La Alcarria dispone de 15 ordenadores conectados a Internet. Se quiere hacer una simulación de la utilización de los ordenadores por los alumnos. Para ello se supone que la frecuencia de llegada de un alumno es de 18 minutos las dos primeras horas, y de 15 minutos el resto del día. El tiempo de utilización del ordenador es un valor aleatorio, entre 30 y 55 minutos. El programa debe tener como salida líneas en las que se refleja la llegada de un alumno, la hora en que llega y el tiempo de la conexión. En el supuesto de que llegue un alumno y no haya ordenadores libres el alumno no espera, se mostrará el correspondiente aviso. En una cola de prioridad se tiene que guardar los distintos "eventos" que se producen, de tal forma que el programa avance de evento a evento. Suponer que la duración de la simulación es de las 14 de la mañana a las 8 de la tarde.

13.4. La entrada a una sala de arte que ha inaugurado una gran exposición sobre la evolución del arte rural, se realiza por tres torniquetes. Las personas que quieren ver la exposición forman una única fila y llegan de acuerdo a una distribución exponencial, con un tiempo medio entre llegadas de 2 minutos. Una persona que llega a la fila y ve esperando a más de 16 personas, se va con una probabilidad del 20 por 100, aumentando ésta en 16 puntos por cada 15 personas más que haya esperando, hasta un tope del 50 por 100. El tiempo medio que tarda una persona en pasar es de 1 minuto (compra de la entrada y revisión de los bolsos). Además, cada visitante emplea en recorrer la exposición entre 15 y 25 minutos distribuido uniformemente.

La sala sólo admite, como máximo, 50 personas. Simular el sistema durante un período de 6 horas para determinar:

- Número de personas que llegan a la sala y número de personas que entran.
- Tiempo medio que debe esperar una persona para entrar en la sala.

Utilizar para simulación el *TAD Cola de prioridad*.

13.5. Escribir una función que busque todos los nodos menores que algún valor v en un montículo binario. Estudiar su complejidad.

13.6. Escribir un TDA que permita el tratamiento de montículos binomiales y que tengan las operaciones de fusionar montículos, borrar el mínimo, e insertar en la cola binomial.

13.7. Suponer que se añade a la estructura de los montículos binomiales la posibilidad de contener dos árboles binomiales de la misma altura como máximo. Mostrar cómo serían en ese caso las operaciones de tratamiento del montículo binomial y estudiar su complejidad.

CAPÍTULO 14

Tablas de dispersión, funciones hash

Objetivos

Con el estudio de este capítulo usted podrá:

- Distinguir entre una tabla lineal y una tabla hash.
- Conocer las aplicaciones que tienen las tablas hash.
- Manejar funciones matemáticas sencillas con las que obtener direcciones según una clave.
- Conocer diversos métodos de resolución de colisiones.
- Realizar en C++ una tabla hash con resolución de colisiones.

Contenido

Conceptos clave

- Acceso aleatorio.
- Colisión.
- Dispersión.
- Diccionario.
- Exploración.
- Factor de carga.
- Hueco.
- Tabla.

INTRODUCCIÓN

Las tablas de datos permiten el acceso directo a un elemento de una secuencia, indicando la posición que ocupan. Un *diccionario* es también una secuencia de elementos, pero éstos son realmente pares formados por un identificador y un número entero. Las tablas de dispersión se suelen utilizar para implementar cualquier tipo de *diccionario*, por ejemplo, la tabla de símbolos de un compilador necesaria para generar el código ejecutable. La potencia de las tablas hash o dispersas radica en la búsqueda de elementos, conociendo el campo clave se puede obtener directamente la posición que ocupa y, por consiguiente, la información asociada a dicha clave. Sin embargo, no permiten algoritmos eficientes para acceder a todos los elementos de la tabla, en su recorrido. El estudio de tablas hash acarrea el estudio de funciones hash o dispersión, que mediante expresiones matemáticas permiten obtener direcciones según una clave que es el argumento de la función.

En el capítulo se estudian diversas funciones hash y cómo resolver el problema de que para dos o más claves se obtenga una misma dirección, lo que se conoce como colisión

14.1. TABLAS DE DISPERSIÓN

Las tablas de dispersión o, simplemente, tablas hash son estructuras de datos que se usan en aplicaciones que manejan una secuencia de elementos. De tal forma que cada elemento tiene asociado un valor *clave*, que es un número entero positivo perteneciente a un rango de valores, relativamente pequeño. En estas organizaciones cada uno de los elementos ha de tener una clave que identifica de manera unívoca al elemento. Por ejemplo, el campo *número de matrícula* del conjunto de alumnos puede considerarse un campo clave para organizar la información relativa al alumnado de la Universidad; el número de matrícula es único, hay una relación biunívoca, uno a uno, entre el campo y el registro alumno. Puede suponerse que no existen, simultáneamente, dos registros con el mismo número de matrícula.

A tener en cuenta

Un *Diccionario* es un tipo abstracto de datos en el que los elementos tienen asociado una clave única en el conjunto de los números enteros positivos. De tal forma que para cualquier par de elementos distintos, sus claves son también distintas. Con las tablas de dispersión se implementa eficientemente el tipo abstracto de datos *Diccionario*.

14.1.1. Definición de una tabla de dispersión

Las tablas de dispersión son estructuras de datos que tienen como finalidad realizar las operaciones fundamentales de búsqueda y eliminación de un registro en un tiempo de ejecución constante (complejidad constante *O(1)*). La organización ideal de una tabla es de tal forma que el campo clave de los elementos se corresponda directamente con el índice de la tabla. Si, por ejemplo, una compañía tiene `300` empleados, cada uno identificado con un número de nómina de `0` a `999`. La forma de organizar la tabla es con un array de `1000` registros:

```
Empleado * tabla = new Empleado[1000];
```

El elemento `tabla[i]` almacena al empleado cuya nómina es `i`. Con esta organización la búsqueda de un empleado se realizado directamente, con un único acceso, debido a que el número de nómina es la posición en la tabla. La eficiencia se puede expresar como tiempo constante, `O(1)`.

Sin embargo, muchas posiciones de la tabla están vacías (se corresponden con números de nómina que no existen) y eso que el rango del campo clave es relativamente pequeño. Imagínese que los número de nómina fueran de 5 dígitos, en este caso, las posiciones vacías estarán en clara desproporción, la memoria ocupada por la tabla queda desaprovechada. Pero enseguida se puede plantear una solución: tomar los tres primeros dígitos del número de nómina, campo clave, como índice del array o tabla de registros, entonces se ha hecho una transformación del campo clave en un entero de 3 dígitos:

```
h(número de nómina) → indice
```

Se puede concluir que el primer problema que plantea esta organización: ¿cómo evitar que el array o vector utilizado esté en una proporción adecuada al número de registros? Las funciones de transformación de claves, funciones hash, permiten que el rango posible de índices estén en proporción al número real de registros.

Una función que transforma números grandes en otros más pequeños se conoce como *funciones de dispersión* o *funciones hash*.

A recordar

Una tabla de dispersión consta de un array (vector), donde se almacenan los registros o elementos, y de una función hash que transforma el campo clave elegido en el rango entero del array.

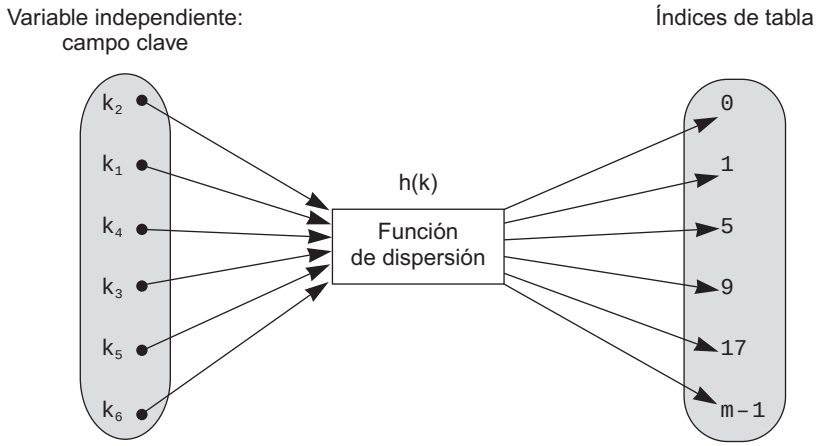


Figura 14.1. Función de dispersión.

14.1.2. Operaciones de una tabla de dispersión

Las tablas dispersas implementan eficientemente los tipos de datos denominados *Diccionarios*. Un diccionario asocia una *clave* con un *valor*. Por ejemplo, el diccionario de inglés asocia una palabra con la traducción en inglés. La operación *búsqueda* de una palabra se realiza frecuentemente; dada una palabra, se localiza su posición, y se obtiene la traducción correspondiente (el *valor asociado*).

La primera operación relativa a una tabla dispersa consiste en su creacción, esto es, dar de alta elementos. La operación de *insertar* añade un elemento, en la posición que le corresponde según la clave del elemento. De igual manera, también pueden darse de baja elementos, la operación *eliminar* extrae un elemento de la tabla que se encuentra en la posición que le corresponde según su clave.

En las tablas dispersas no se utiliza directamente la clave para indexar, el índice se calcula con una función matemática, *función hash h(x)*, que tiene como argumento la clave del elemento y devuelve una dirección o índice en el rango de la tabla. Según esto, considerando *h(x)* la función hash, se puede especificar las operaciones del tipo *Tabla dispersa*:

Buscar(Tabla T, clave x)

```
devuelve el elemento de la tabla T[h(x)]
```

Insertar(Tabla T, elemento k)

```
añade el elemento k, T[h(clave(k)] ← k
```

Eliminar (Tabla T, clave x)

```
retira de la tabla el elemento con clave x, T[h(x)] ← LIBRE
```

La ventaja de utilizar tablas de dispersión radica en la eficiencia de estas operaciones. Si la función hash es de complejidad constante, la complejidad de cada una de estas tres operaciones también es constante, `O(1)`. Un problema potencial de la función de dispersión es el de las colisiones, éstos es, dadas dos claves distintas x_i, x_j se obtenga la misma dirección o índice: $h(x_i) = h(x_j)$. La operación de *insertar* tiene que incorporar el proceso de *resolución de colisiones*, no pueden estar dos elementos en la misma posición. De igual forma, los procesos de *búsqueda* y *eliminación* también quedan afectados por la resolución de colisiones.

Las tablas dispersas se diseñan considerando el problema de las colisiones. Siempre se reservan más posiciones de memoria, `m`, que elementos previstos a almacenar, `n`. Cuanta más posiciones haya, menor es el riesgo de colisiones, pero más *huecos* libres quedan (memoria desaprovechada).

El parámetro que mide la proporción entre el número de elementos almacenados, `n`, en una tabla dispersa y el tamaño de la tabla, `m`, se denomina factor de carga, $\lambda = n/m$. *Se recomienda* elegir m de tal forma que el factor de carga, $\lambda \leq 0.8$.

14.2. FUNCIONES DE DISPERSIÓN

Una *función de dispersión* convierte el dato considerado campo clave (tipo entero o cadena de caracteres) en un índice dentro del rango de definición del array o vector que almacena los elementos, de tal forma que sea adecuado para indexar el array.

La idea que subyace es utilizar la clave de un elemento para determinar su dirección o posición en un almacenamiento secuencial, pero sin desperdiciar mucho espacio, para lo que se realiza una transformación, mediante una *función hash*, del conjunto `K` de claves sobre el conjunto `L` de direcciones de memoria.

```
h: K → L
x → h(x)
```

Ésta es la función de direccionamiento *hash* o función de dispersión. Si *x* es una clave, entonces *h(x)* se denomina direccionamiento *hash* de la clave *x*, además, es el índice de la tabla donde se guardará el registro con esa clave. Así, si la tabla tiene un tamaño de `tamTabla = = 199`, la *función hash* que se elija tiene generar índices en el rango `0 ... tamTabla-1`. Si la clave es un entero, por ejemplo el número de serie de un artículo (hasta 6 dígitos) y se dispone de una tabla de `tamTabla` elementos, la función de direccionamiento tiene que transformar valores pertenecientes al rango `0 ... 999999`, en valores pertenecientes al subrango `0 ... tamTabla-1`. La clave elegida también puede ser una cadena de caracteres, en ese caso se hace una transformación previa a valor entero. La función *hash* más simple utiliza el operador *módulo* (*resto entero*):

```
x % tamTabla = genera un número entero de 0 a tamTabla-1
```

Es necesario valorar el hecho que la *función hash*, `h(x)`, no genere valores distintos, es posible (según la función elegida) que dos claves diferentes `c1` y `c2` den la misma dirección, `h(c1) = h(c2)`. Entonces se produce el fenómeno de la *colisión*, se debe usar algún método para resolverla. Por tanto, el estudio de *direccionamiento hash* implica dos hechos: elección de *funciones hash* y *resolución de colisiones*.

Existe un número considerable de *funciones hash*. Dos criterios deben considerarse a la hora de seleccionar una función. En primer lugar, la función, `h(x)`, sea fácil de evaluar (dependerá del campo clave) y que su tiempo de ejecución sea mínimo, de complejidad constante, `O(1)`. En segundo lugar, `h(x)` debe distribuir uniformemente las direcciones sobre el conjunto `L`, de forma que se minimice el número de *colisiones*. Nunca existirá una garantía plena de que no haya colisiones, y más sin conocer de antemano las claves y las direcciones. La experiencia enseña que siempre habrá que preparar la *resolución de colisiones* para cuando éstas se produzcan.

Algunas de las *funciones hash* de cálculo más fácil y rápido se exponen en las secciones siguientes. Para todas ellas se sigue el criterio de considerar a `x` una clave cualquiera, `m` el tamaño de la tabla y, por tanto, los índices de la tabla varían de `0` a `m-1`; y por último, el número de elementos es `n`.

14.2.1. Aritmética modular

Una *función de dispersión* que utiliza la *aritmética modular* genera valores dispersos calculando el resto de la división entera entre la clave `x` y el tamaño de la tabla `m`.

```
h(x)= x modulo m
```

Normalmente, la clave asociada con un elemento es de tipo entero. A veces, los valores de la clave no son enteros, entonces, previamente, hay que transformar la clave a un valor entero.

Por ejemplo, si la clave es una cadena de caracteres, se puede transformar considerando el valor ASCII de cada carácter como si fuera un dígito entero de base 128.

La operación *resto(módulo)* genera un número entre 0 y m-1 cuando el segundo operando es m. Por tanto, esta *función de dispersión* proporciona valores enteros dentro del rango 0 ... m-1.

Con el fin de que esta función disperse lo más uniformemente posible, es necesario tener ciertas precauciones con la elección del tamaño de la tabla, m. Así, no es recomendable escoger el valor de m múltiplo de 2, ni tampoco múltiplo de 10. Si $m = 2^j$ entonces la distribución de h(x) se basa únicamente en los j dígitos menos significativos; y si $m = 10^j$ ocurre lo mismo. En estos casos, la distribución de las claves se basa sólo en una parte, los j dígitos menos significativos, de la información que suministra la clave y esto sesgará la dispersión hacia ciertos valores o índices de la tabla. Si por ejemplo, el tamaño elegido para la tabla es $m = 100\ (10^2)$, entonces h(128) = h(228) =h(628) = 28.

Las elecciones recomendadas del tamaño de la tabla, m, son números primos mayores, aunque cercanos, al número de elementos, n, que tiene previsto que almacene la tabla.

EJEMPLO 14.1. Considerar una aplicación en la que se debe almacenar n = 900 registros . El campo clave elegido es el número de identificación.Elegir el tamaño de la tabla de dispersión y calcular la posición que ocupa los elementos cuy o número de identificación es:

```
245643   245981   257135
```

Una buena elección de m, en este supuesto, es 997 al ser un número primo próximo y tener como factor de carga (n/m) aproximadamente 0.8 cuando se hayan guardado todos los elementos.

Teniendo en cuenta el valor de m, se aplica la función hash de aritmética modular y se obtienen estas direcciones:

```
h(245643)= 245643 modulo 997 = 381
h(245981)= 245981 modulo 997 = 719
h(257135)= 257135 modulo 997 = 906
```

14.2.2. Plegamiento

La técnica del plegamiento se utiliza cuando el valor entero del campo clave elegido es demasiado grande, pudiendo ocurrir que no pueda ser almacenado en memoria. Consiste en partir la clave x en varias partes $x_1, x_2, x_3 \ldots x_n$, la combinación de las partes de un modo conveniente (a menudo sumando las partes) da como resultado la dirección del registro. Cada parte x_i, con a lo sumo la excepción de la última, tiene el mismo número de dígitos que la dirección especificada.

La función hash se define:

```
h(x) = x1 + x2 + x3 + ... + xr
```

La operación que se realiza para el cálculo de la función *hash* desprecia los dígitos más significativos obtenidos del acarreo.

A tener en cuenta

La técnica de plegar la clave de dispersión a menudo se utiliza para transformar una clave muy *grande* en otra más *pequeña*, y a continuación aplicar la función hash de aritmética modular.

EJEMPLO 14.2. Los registros de pasajeros de un tren de largo recorrido se identifican por un campo de seis dígitos, que se va a utilizar como clave para crear una tabla dispersa de m = 1.000 posiciones (rango de 0 a 999). La función de dispersión utiliza la técnica de plegamiento de tal forma que parte a la clave, 6 dígitos, en dos grupos de tres y tres dígitos y, a continuación, se suman los valores de cada grupo.

Aplicar esta función a los registros:

```
245643     245981      257135
```

La primera clave, 245643, al dividirla en dos grupos de 3 dígitos se obtiene: 245 y 643. La dirección obtenida:

```
h(245643) = 245 + 643 = 888
```

Para las otras claves:

```
h(245981) = 245 + 981 = 1226 → 226 (se ignora el acarreo 1)
h(257135) = 257 + 135 = 392
```

A veces, se determina la inversa de las partes pares, x_2, x_4 ... antes de sumarlas, con el fin de conseguir mayor dispersión. La inversa en el sentido de invertir el orden de los dígitos. Así, con las mismas claves se obtienen las direcciones:

```
h(245643) = 245 + 346 = 591
h(245981) = 245 + 189 = 434
h(257135) = 257 + 531 = 788
```

14.2.3. Mitad del cuadrado

Esta técnica de obtener direcciones dispersas consiste, en primer lugar, en calcular el cuadrado de la clave x y, a continuación, extraer del resultado, x^2, los dígitos que se encuentran en ciertas posiciones. El número de dígitos a extraer depende del rango de dispersión que se quiere obtener. Así, si el rango es de 0 .. 999 se extraen tres dígitos, siempre, los que están en las mismas posiciones.

Un problema potencial, al calcular x^2, es que sea demasiado grande y *exceda* el máximo entero. Es importante, al aplicar este método de dispersión, utilizar siempre las mismas posiciones de extracción para todas las claves.

EJEMPLO 14.3. Aplicar el método de *Mitad del Cuadrado* a los mismos registros del Ejemplo 14.2.

Una vez elevado al cuadrado el valor de la clave, se eligen los dígitos que se encuentran en las posiciones 4, 5 y 6 por la derecha. El valor de esa secuencia es la dirección obtenida al aplicar este método de dispersión.

Para `245643, h(245643)= 483;` *paso a paso*:
`245643` → 245643^2 → `60340483449` → (dígitos 4, 5 y 6 por la derecha) `483`

Para `245981, h(245981)= 652;` *paso a paso*:
`245981` → 245981^2 → `60506652361` → (dígitos 4, 5 y 6 por la derecha) `652`

Para `257135, h(257135)= 408;` *paso a paso*:
`257135` → 257135^2 → `66118408225` → (dígitos 4, 5 y 6 por la derecha) `408`

14.2.4. Método de la multiplicación

La dispersión de una clave, utilizando el *método de la multiplicación*, genera direcciones en tres pasos. Primero, multiplica la clave `x` por una constante real, `R`, comprendida entre `0` y `1` (`0. < R < 1.0`). En segundo lugar, determina la parte decimal, `d`, del número obtenido en la multiplicación, `R*x`. Y por último, multiplica el tamaño de la tabla, `m`, por `d` y al truncarse el resultado se obtiene un número entero en el rango `0 .. m - 1` que será la dirección dispersa.

```
1. R * x
2. d = R * x – ParteEntera(R * x)
3. h(x) = ParteEntera(m * d)
```

Una elección de la constante `R` es la inversa de la razón áurea, `R = 0.6180334` = $\frac{2}{1+\sqrt{5}}$.

Por ejemplo, la clave `x = 245981, m = 1000`, la dirección que se obtiene:

```
1. R * x:     0.6180334 * 245981 → 152024.4738
2. d:         152024.4738 – ParteEntera(152024.4738) → 0.4738
3. h(245981): 1000 * 0.4738 → ParteEntera (473.8) → 473
```

Este método de obtener dirección dispersa tiene dos características importantes. La primera, dos claves con los dígitos permutados no tienen mayor probabilidad de generar una colisión (igual dirección) que dos claves cualesquiera. La segunda característica, dos valores de claves numéricamente muy próximas, generan direcciones dispersas que pueden estar muy separadas.

Considérese, por ejemplo, la clave `x = 245982` para el mismo valor de `R = 0.6180334` y `m = 1000`, la dispersión:

```
1. R * x:     0.6180334 * 245982 → 152025.0918
2. d:         152025.0918 – ParteEntera(152025.0918) → 0.0918
3. h(245982): 1000 * 0.0918 → ParteEntera (91.8) → 91
```

La dispersión obtenida para `245891, h(245981) = 473` muy alejada de `91`.

EJERCICIO 14.1. Los registros que representan a los objetos de una perfumería se van a guardar en una tabla dispersa de m = 1.024 posiciones. El campo clave es una cadena de caracteres, cuya máxima longitud es 10. Se decide aplicar el método de la multiplicación como función de dispersión. Con este supuesto codificar la función de dispersión y mostrar 10 direcciones dispersas.

Para generar la dispersión de las claves, primero se transforma la cadena, que es el campo clave, en un valor entero. Una vez hecha la transformación, se aplica el método de la multiplicación.

La transformación de la cadena se realiza considerando que es una secuencia de valores numéricos en base 27. Así, por ejemplo, la cadena `'RIO'` se transforma en:

$$\texttt{'R'}*27^2 + \texttt{'I'}*27^1 + \texttt{'O'}*27^0$$

El valor entero de cada carácter es su ordinal en el código ASCII. El tipo de dato `char` ocupa dos bytes de memoria, se representa como un valor entero que es, precisamente, el ordinal. Para obtener el valor entero de un carácter, simplemente se hace una conversión a `int`; por ejemplo, `(int)'a'` es el valor entero 97.

La transformación da lugar a valores que sobrepasan el máximo entero (incluso con enteros largos), generando números negativos. Esto no genera ningún tipo de problema, sencillamente se cambia de signo.

Para generar las 10 direcciones dispersas y probar la función, el programa tiene como entrada cada una de las 10 cadenas; se invoca a la función de dispersión y se escribe la dirección.

```
  /*
    función hash: método de la multiplicación
    Las claves son cadenas de caracteres, primero se transforma
    a valor entero. A continuación se aplica el método de
    multiplicación
  */

#include <cstdlib>
#include <iostream>
#include <math.h> // incluye la función floor de conversión decimal

using namespace std;
const int m = 1024;
const float R = 0.618034;

long transformaClave(const char* clave);
int dispersion(long x);

int main(int argc, char *argv[])
{
    char clave[11];
    long valor;
    int k;

    for (k = 1; k <= 10; k++)
    {
       cout << "\nClave a dispersar: ";
```

```
            cin >> clave;
            valor = transformaClave(clave);
            valor = dispersion(valor);
            cout << "Dispersion de clave " << clave << "--> "
                 << valor << endl;
        }
        return 0;
    }

    long transformaClave(const char* clave)
    {
        int j;
        long d;

        d = 0;
        for (j = 0; j < strlen(clave); j++)
        {
            d = d * 27 + clave[j];
        }
            //En caso de valor grande no se puede almacenar en memoria
            //y genera un numero negativo
        if (d < 0) d = -d;
        return d;
    }

    int dispersion(long x)
    {
      double t;
      int v;
      t = R * x - floor(R * x);     // parte decimal
      v = (int) (m * t);
      return v;
}
```

14.3. COLISIONES Y RESOLUCIÓN DE COLISIONES

La función de dispersión, `h(x)`, elegida puede generar la misma posición al aplicarla a las claves de dos o más registros diferentes; esto es, obtener la misma posición de la tabla en la que ubicar dos registros. Si ocurre, entonces se produce una **colisión** que hay que resolver para que los registros ocupen diferentes posiciones.

Una función hash ideal, `h(x)`, debe generar direcciones distintas para dos claves distintas. No siempre es así, no siempre proporciona direcciones distintas; en ocasiones ocurre que dadas dos claves diferentes x_1, $x_2 \Rightarrow h(x_1) = h(x_2)$. Este hecho es conocido como colisión, es evidente que el diseño una tabla dispersa debe proporcionar métodos de *resolución de colisiones*.

A tener en cuenta

Hay dos cuestiones que se deben considerar a la hora de diseñar una tabla hash. Primero, seleccionar una buena función hash, que *disperse lo más uniformemente*. Segundo, seleccionar un *método para resolver colisiones*.

Considérese, por ejemplo, una comunidad de `88` vecinos, cada uno con muchos campos de información relevantes: nombre, edad, ..., de los que se elige como campo clave el *DNI*. El tamaño de la tabla va a ser `101` (número primo mayor que `88`), por consiguiente, `101` posibles direcciones. Aplicando la función hash del *módulo*, la claves `123445678`, `123445880` proporcionan las direcciones:

```
h(123445678) = 123445678 modulo 101 = 44
h(123445880) = 123445880 modulo 101 = 44
```

En el caso de dos claves distintas, a las que se aplica la función *hash* del módulo, se obtienen dos direcciones iguales, se dice que las claves han colisionado.

Es importante tener en cuenta que la *resolución de colisiones*, en una tabla dispersa, afecta directamente a la eficiencia de las operaciones básicas sobre la tabla: *insertar*, *buscar* y *eliminar*.

Se consideran dos modelos para resolver colisiones: *exploración de direcciones o direccionamiento abierto* y *direccionamiento enlazado*. En las siguientes secciones se muestran los diversos métodos de resolución.

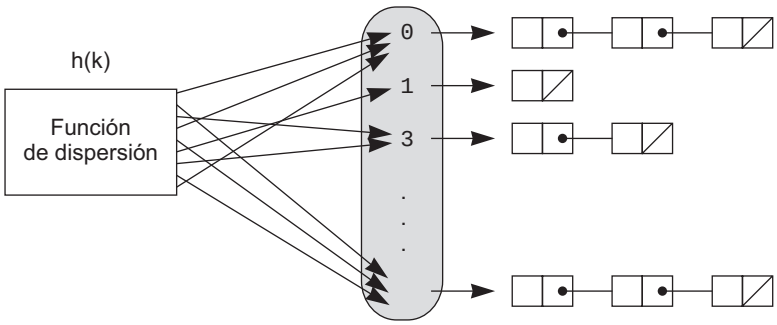


Figura 14.2. Resolución de colisiones por dispersión abier ta (enlazada).

14.4. EXPLORACIÓN DE DIRECCIONES. DIRECCIONAMIENTO ABIERTO

Los diversos métodos de exploración se utilizan cuando todos los elementos, colisionados o no, se almacenan en la misma tabla. Las colisiones se resuelven explorando consecutivamente en una secuencia de direcciones, hasta que se encuentra una posición *libre* (un *hueco*), en la operación de insertar, o se encuentra el elemento buscado en las operaciones *buscar* y *eliminar*.

Es importante, al diseñar una tabla dispersa basada la resolución de colisiones en la *exploración de una secuencia*, inicializar todas las posiciones de la tabla a un valor que indique vacío, por ejemplo `NULL`, o cualquier parámetro que indique posición vacía. Al insertar un elemento, si se produce una colisión, la secuencia de exploración termina cuando se encuentra una dirección de la secuencia vacía.

Al buscar un elemento, se obtiene la dirección dispersa según su clave, a partir de esa dirección es posible que haya que explorar la secuencia de posiciones hasta encontrar la clave

buscada. En la operación de eliminar, una vez encontrada la clave, se indica con un parámetro el estatus de borrado. Dependiendo de la aplicación una posición eliminada puede utilizarse, posteriormente para una inserción.

A tener en cuenta

La secuencia de posiciones (índices) a la que da lugar un método de exploración tiene que ser el mismo, independiente de la operación que realiza la tabla dispersa. Es importante marcar con un parámetro que una posición de la tab la está vacía.

14.4.1. Exploración lineal

Es la forma más primaria y simple de resolver una colisión entre claves al aplicar una función de dispersión. Supóngase que se tiene un elemento de clave `x`, la dirección que devuelve la función `h(x) = p,` si esta posición ya está ocupada por otro elemento se ha producido una colisión. La forma de resolver está colisión, con *exploración lineal*, consiste en buscar la primera posición disponible que siga a `p`. La secuencia de exploración que se genera es lineal: `p, p+1, p+2, ... m-1, 0, 1, ...` y así consecutivamente hasta encontrar una posición vacía. La tabla se ha de considerar *circular*, de tal forma que considerando `m-1` la última posición, la siguiente es la posición `0`.

EJEMPLO 14.4. Se tiene 9 elementos cuyas claves simbólicas son: x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 y x_9. Para cada uno la función de dispersión gener a las direcciones:

```
Elemento: x1 x2 x3 x4 x5 x6 x7 x8 x9
h(x) :    5  8  11 9  5  7  8  6  14
```

Entonces las posiciones de almacenamiento en la tabla, aplicando exploración lineal:

```
Elemento: x1 x2 x3 x4 x5 x6 x7 x8 x9
Posición: 5  8  11 9  6  7  10 12 14
```

La primera colisión se produce al almacenar el elemento x_5, según la función hash le corresponde la posición 5, que está ocupada. La siguiente posición esta libre y a ella se asigna el elemento. Al elemento x_8 le corresponde la dirección dispersa 6, sin embargo, esa posición ya está ocupada por x_5 debido a una colisión previa, la siguiente posición libre es la 12, dirección que le corresponde al resolver la colisión con el método de exploración lineal.

Las operaciones de *buscar* y *eliminar* un elemento en una tabla dispersa con exploración lineal sigue los mismos pasos que la operación de *insertar* (Ejemplo 14.4). La búsqueda comienza en la posición que devuelve la función hash, `h(clave)`. Si la clave, en esa posición, es la clave de búsqueda la operación ha tenido éxito; en caso contrario, la exploración continúa linealmente en las siguientes posiciones, hasta encontrar la clave, o bien decidir que no existe en la tabla, lo que ocurre si una posición está vacía. Por ejemplo, para buscar el

elemento x_8 en la tabla del ejemplo anterior, se examinan las posiciones `6, 7, 8, 9, 10` y `12`.

La exploración lineal es muy sencilla de implementar, tiene un inconveniente importante, el agrupamiento de elementos en la tabla, sobre todo, cuando el factor de ocupación se acerca a `0.5`, el agrupamiento de elementos en posiciones adyacentes o consecutivas es notable.

Análisis de la exploración lineal

La eficiencia de una función hash, junto al método de resolución de colisiones, supone analizar la operación de *insertar* y la operación de *buscar* (*eliminar* supone una búsqueda). El factor de carga de la tabla, λ, es determinante a la hora de determinar la eficiencia de las operaciones.

La eficiencia de la inserción se suele medir como *el número medio de posiciones examinadas*. En la exploración lineal esta magnitud se aproxima a:

$$\frac{1}{2} * \left(1 + \frac{1}{(1-\lambda)^2}\right)$$

Al analizar la operación de búsqueda de un elemento se tiene en cuenta que tenga éxito, o bien que sea una búsqueda sin éxito (que no se encuentre). Por esa razón, la eficiencia de la operación buscar se expresa mediante dos parámetros:

`S` (λ) = número medio de comparaciones para una búsqueda con éxito.
`U` (λ) = número medio de comparaciones para una búsqueda sin éxito.

Se puede demostrar que para la exploración lineal estos parámetros:

$$S(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right) \qquad \text{y} \qquad U(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

A tener en cuenta

La exploración lineal es fácil de implementar en cualquier lenguaje de prog ramación. Tiene como principal inconveniente, cuando el factor de carga supera el 50 por 100, el agrupamiento de elementos en posiciones contiguas . Situar los elementos en posiciones contiguas aumenta el tiempo medio de la oper acíon búsqueda

14.4.2. Exploración cuadrática

La resolución de colisiones con la exploración lineal provoca que se agrupen los elementos de la tabla según se va acercando el factor de carga a `0.5`. Una alternativa para evitar la agrupación es la *exploración cuadrática*. Suponiendo que un elemento con clave `x` le corresponde la dirección `p`, y que la posición de la tabla indexada por `p` está *ocupada*, el método de exploración o prueba cuadrática busca en las direcciones `p, p+1, p+4, p+9, ... p+i²`, considerando a la tabla como un array circular. El nombre de *cuadrática*, a esta forma de explorar, se debe al desplazamiento relativo, `i²` para valores de `i= 1, 2, 3, ....`

EJEMPLO 14.5. Se dispone de 9 elementos cuy as claves simbólicas son: `x1, x2, x3, x4, x5, x6, x7, x8 y x9`. El tamaño de la tabla donde se guardan es `17` (número primo), rango de 0 a 16. Para cada uno de los elementos la función de dispersión gener a las direcciones:

```
Elemento:  x1 x2 x3 x4 x5 x6 x7 x8 x9
h(x):      5  8  10 8  5  11 6  7  7
```

Las posiciones de almacenamiento en la tabla, aplicando la exploración cuadrática:

```
Elemento:  x1 x2 x3 x4 x5 x6 x7 x8 x9
Posición:  5  8  10 9  6  11 7  16 15
```

La primera colisión se produce al almacenar el elemento `x4`. Según la función hash le corresponde la posición `8`, que está ocupada. Si se considera la secuencia de exploración cuadrática: `8, 8+1, 8+4,...`, como la posición `9` está libre en ella se asigna el elemento.

El elemento `x5` le corresponde la dirección dispersa `5`, sin embargo, hay una colisión con el elemento `x1`; la secuencia de exploración para esta clave, `5, 5+1, 5+4,...`, como la posición `6` está libre, se asigna el elemento.

La siguiente colisión se da con `x7`, le corresponde la dirección dispersa `6`, sin embargo esa posición está ya ocupada debido a una colisión previa; la secuencia de exploración para esta clave, `6, 6+1, 6+4,...`, la posición `7` está libre y se asigna el elemento.

Al elemento `x8` le corresponde la dirección `7` ya ocupada. La secuencia de exploración, `7, 7+1, 7+4, 7+9,...`, resulta que la posición `16` está libre, en ella se asigna el elemento.

Por último, el elemento `x9` le corresponde la dirección dispersa `7`, ya asignada. La secuencia de exploración cuadrática en la que se prueba hasta encontrar una posición libre: `7, 7+1, 7+4, 7+9, 7+16(6), 7+25 ...`, resulta que ahora hay que tratar la secuencia en forma circular al estar ocupadas las cuatro primeras posiciones. El siguiente valor de la secuencia es `23`, se corresponde con `6` (*teoría de restos*), también está ocupado. A continuación se prueba la dirección `32`, que se corresponde con `15` por la *teoría de restos*, al estar libre es la posición donde se asigna el elemento.

Análisis de la exploración cuadrática

Este método de resolver colisiones reduce el agrupamiento que produce la exploración lineal. Sin embargo, si no se elige convenientemente el tamaño de la tabla no se puede asegurar que se prueben todas las posiciones de la tabla.

Se puede demostrar que si el tamaño de la tabla es un número primo y el factor de carga no alcanza el `50` por `100`, todas las *pruebas* que se realicen con la secuencia $p+i^2$ se hace sobre posiciones de la tabla distintas y siempre se podrá insertar.

14.4.3. Doble dirección dispersa

Este método de resolución de colisiones utiliza una segunda función hash. Se tiene una función hash principal, `h(x)`, y otra función secundaria, `h'(x)`. El primer intento de insertar, o buscar,

un nuevo elemento inspecciona la posición `h(x) = p`, si hay una colisión se obtiene un segundo desplazamiento con otra función *hash*, `h'(x) = p'`. Entonces la secuencia de exploración: `p, p+p', p+2p', p+3p' ...` se inspecciona hasta encontrar una posición libre, para insertar, o bien, buscar un elemento.

14.4.4. Inconvenientes del direccionamiento abierto

Si se trabaja con un factor de carga alto los resultados obtenidos por el direccionamiento abierto tienden a degenerar debido al aumento de la probabilidad de que una clave sea direccionada a una posición que ya esté ocupada por otra clave. Otro inconveniente, es que se entremezclan las claves que han colisionado con las que no lo han hecho, por lo que se dificulta el proceso de borrado de las claves en la tabla de dispersión. El problema está en que si una posición está ocupada, provocará una colisión cada vez que se inserta un elemento en ella. Todas las búsquedas posteriores de elementos que colisionen pasan por esta posición para seguir la secuencia de exploración, hasta que se encuentre la clave. De esta forma, si se borra la posición hay que tener en cuenta que ya estuvo ocupada para poder seguir la búsqueda en un proceso de exploración posterior. Es decir, se trata de marcar la casilla como borrada para poder seguir el proceso de búsqueda. Hay que tener en cuenta en este caso, que la inserción de una nueva clave también debe modificarse, ya que no solamente puede insertarse en posiciones libres, sino también en posiciones que hayan sido borradas.

14.5. REALIZACIÓN DE UNA TABLA DISPERSA CON DIRECCIONAMIENTO ABIERTO

A continuación, se implementa una tabla dispersa para almacenar un conjunto de elementos, se supone que son las *Casas Rurales* de la comarca de La Mancha. La resolución de colisiones se realiza formando una secuencia de posiciones aplicando el método de *exploración cuadrática*.

Los elementos de la tabla son objetos con los siguientes atributos: *población*, *dirección*, *numHabitacion*, *precio* por día y *código* de identificación. El código, normalmente 5 caracteres, tiene una relación biunívoca con la Casa Rural, por ello se elige como atributo clave. Los campos *población* y *dirección*, son de tipo cadena; y *numHabitacion*, *precio* de tipo `int` y `double` respectivamente.

La clase *TablaDispersa* consta de un array de referencias a los objetos *CasaRural*. El tamaño de la tabla está en función del número de *Casas* conocido. Este tamaño debe ser decidido por el programador, en el momento que se llame al construtor de la clase `TablaDispersa` (Por ejemplo, si la tabla dispersa almacena aproximadamente `50` objetos de `CasaRural`, el tamaño elegido n debe ser el número primo `101`). Cada posición de la tabla contiene `NULL`, o bien la dirección de un objeto *Casa Rural.* Se ha tomado la decisión de que los elementos dados de baja permanezcan en la tabla (para no perder información histórica), por ello se añade el atributo, *esAlta*, que si está activo (*true*) indica que es un *alta*, en caso contrario (*false*) se dio de *baja*. El número de elementos que hay en la tabla, incluyendo las bajas, se almacena en la variable *numElementos*; además se añade la variable *factorCarga*, de tal forma que cuando se alcance el `0.5` se pueda generar un *aviso*.

14.5.1. Declaración de la clase `TablaDispersa`

Se declara la estructura `TipoCasa` que almacena la información correspondienete a una casa. En primer lugar, se declara la clase `CasaRural`. Esta clase ofrece de modo público el constructor por defecto `CasaRural` y el constructor sobrecargado `CasaRural (TipoCasa c)`, que construye un objeto de la clase almacenando una casa `c` de `TipoCasa`. Este método usa la función miembro privada `asigna` que inicializa los atributos de la casa que recibe como parámetro en la estructura `c`. Se implementan las funciones `OesAlta`, `PesAlta(bool sw)` encargados de retornar el atributo `esAlta` y de cambiar el atributo `esAlta` respectivamente. La función `elCodigo`, obtiene el atributo `código`, y `visualiza` muestra todos los atributos de la clase `CasaRural`.

```
struct TipoCasa
{
    char codigo[5];
    char poblacion[31];
    char direccion[51];
    int habitaciones;
    double precio;
};

class CasaRural
{
protected:
    char codigo[5];
    char poblacion[31];
    char direccion[51];
    int habitaciones;
    double precio;
    bool esAlta;
    public:
    CasaRural(){} // constructor por defecto
    CasaRural (TipoCasa c) {esAlta = true; asigna(c);}
    ~CasaRural(){ esAlta = false;}
    char* elCodigo(){return codigo;}
    bool OesAlta(){return esAlta;}
    void PesAlta(bool sw){ esAlta = sw;}
    void muestra(){
        cout << "\n Casa Rural " << codigo << endl;
        cout << "Población: " << poblacion << endl;
        cout << "Dirección: " << direccion << endl;
        cout << "Precio por día: " << precio << endl;
    }

    private:
      void asigna(TipoCasa c){
            strcpy(poblacion, c.poblacion);
            strcpy(direccion, c.direccion);
            strcpy(codigo, c.codigo);
            habitaciones = c. habitaciones;
            precio = c.precio;}
};
```

La declaración de la clase `TablaDispersa` no implementa las funciones con la operaciones; se hace en los siguientes apartados. Esta clase que contiene los atributos `final, numElementos, factorCarga` y un array `tabla` de tamaño variable de punteros a objetos de la clase `CasaRural`.

```
class TablaDispersa
{
protected:
   int final;
   int numElementos;
   double factorCarga;
   CasaRural **tabla;
public:
   TablaDispersa(int n);            // apartado 14.5.2
   int direccion(char *clave);      // apartado 14.5.3
   void insertar(CasaRural r);      // apartado 14.5.4
   CasaRural* buscar(char* clave);  // apartado 14.5.5
   void eliminar(char *clave);      // apartado 14.5.6
private:
   long transformaCadena(char * c); // apartado 14.5.3
};
```

14.5.2. Inicialización de la tabla dispersa

La creación de un objeto supone una llamada al constructor. En esta implementación, simplemente se crea el objeto array con el tamaño especificado n que se recibe como parámetro, y se inicializa a `NULL` cada posición. Los atributos `numElementos` y `factorCarga` se inicializan a `0`. El atributo `final` se inicializa al tamaño especificado `n`.

```
TablaDispersa::TablaDispersa(int n)
{
  *tabla = new (CasaRural)[n];
  final = n;
  for(int j = 0; j < final ; j++)
    tabla[j] = NULL;
  numElementos = 0;
  factorCarga = 0.0;
}
```

14.5.3. Posición de un elemento

Al añadir un nuevo elemento a la tabla, o bien al buscar el elemento, es necesario determinar la posición en la tabla. El método *aritmética* modular es el utilizado para obtener la dirección dispersa, a partir de la cual se forma una secuencia de exploración cuadrática en la que se busca la primera posición libre (posición a `NULL`). Al ser la clave de *dispersión* de tipo cadena, primero se convierte a un valor entero. El método privado `transforma()` realiza la conversión, de igual forma que en el Ejercicio 14.1.

Se prueba secuencialmente cada posición que determina el método de exploración cuadrática, hasta encontrar una posición vacía (`NULL`), o bien, una posición con la misma clave, que

puede haber sido dada de baja previamente. La función `direccion()` gestiona estas acciones, devuelve la posición o índice de la tabla.

```
int TablaDispersa::direccion(char *clave)
{
    int i = 0;
    long p, d;
    d = transformaCadena(clave);
      // aplica aritmética modular para obtener dirección base
    p = d % final;
      // bucle de exploración
    while (tabla[p]!= NULL && strcmp(tabla[p]->elCodigo(),clave)!= 0)
    {
      i++;
      p = p + i * i;
      p = p % final;            // considera el array como circular
    }
    return (int)p;
  }

long TablaDispersa::transformaCadena(char* c)
{
    long d = 0;
    for (int j = 0; j < strlen(c); j++)
    {
       d = d * 27 + (int)c[j];
    }
    if (d < 0) d = -d;
    return d;
}
```

14.5.4. Insertar un elemento en la tabla

Para incorporar un nuevo elemento a la tabla, primero se busca la posición que debe ocupar, la función `direccion()` devuelve la posición buscada. La operación no considera el hecho de que en esa posición haya un elemento, si es así, se *sobreescribe*. El número de entradas de la tabla se incrementa y se hace un nuevo cálculo del factor de carga. La única acción que se hace con el factor de carga es escribir un mensaje de aviso, si éste supera el `50` por `100`.

```
void TablaDispersa::insertar(CasaRural r)
{
    int posicion;

    posicion = direccion(r.elCodigo());
    tabla[posicion] = new CasaRural(r);
    numElementos++;
    factorCarga = (double)numElementos/final;
    if (factorCarga > 0.5)
      cout<< "\n!! Factor de carga supera el 50 por 100.!!"
            << " Conviene aumentar el tamaño." << endl;
}
```

14.5.5. Búsqueda de un elemento

La operación busca un elemento en la tabla a partir de la dirección dispersa correspondiente a la clave. La función `buscar()` devuelve un puntero al elemento, si se encuentra en la tabla; de no encontrarse, o bien esté dado de baja, devuelve `NULL`.

```
CasaRural* TablaDispersa::buscar(char* clave)
{
    CasaRural *pr;
    int posicion;

    posicion = direccion(clave);
    pr = tabla[posicion];
    if (pr != NULL)
      if (!pr->OesAlta()) pr = NULL;
    return pr;
}
```

14.5.6. Dar de baja un elemento

Para dar de baja a un elemento se siguen los mismos pasos que la operación de búsqueda. Primero se determina la posición del elemento en la tabla, llamando a `direccion()`. A continuación, si en la posición hay un elemento, simplemente se desactiva, se pone a `false` el atributo `esAlta`.

```
void TablaDispersa::eliminar(char *clave)
{
    int posicion;

    posicion = direccion(clave);
    if (tabla[posicion] != NULL)
       tabla[posicion] -> PesAlta(false);
}
```

14.6. DIRECCIONAMIENTO ENLAZADO

En la sección anterior se ha desarrollado una estrategia para la resolución de colisiones que, de una manera o de otra, forma una secuencia de posiciones a explorar. Una alternativa a la secuencia de exploración es el direccionamiento (*hashing*) enlazado. Se basa en utilizar listas enlazadas (cadenas de elementos), de tal forma que en cada lista se colocan los elementos que tienen la misma dirección hash. Todos los elementos que *colisionan*: `h(x1) = h(x2) = = h(x3) ...` van a estar ubicados en la misma lista enlazada.

En la Figura 14.3 se puede ver gráficamente la estructura de datos básica para este método de dispersión. La idea fundamental es la siguiente: si se ha elegido una función hash que genera direcciones en el rango `0` a `m-1`, se debe crear una tabla de `m` posiciones, indexada de `0` a `m-1`; cada posición de la tabla contendrá la dirección de acceso a su correspondiente lista enlazada.

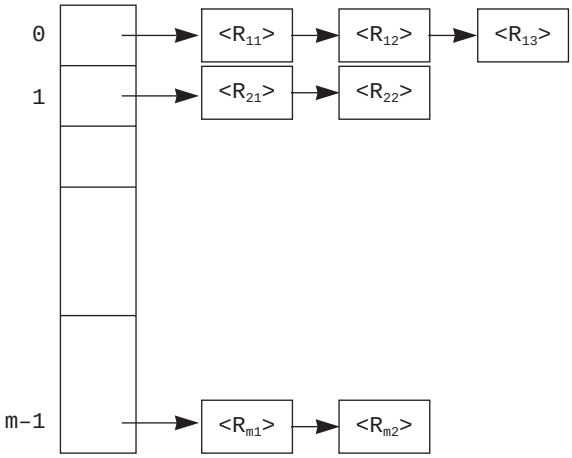


Figura 14.3. Tabla de dispersión enlazada con rango `0..m-1` direcciones.

La estructura de datos que se emplea para la implementación de una tabla dispersa con direccionamiento enlazado es un array, o un vector, de listas enlazadas. Cada posición `i` del vector es una referencia a la lista enlazada cuyos nodos son elementos de la tabla dispersa cuyo direccionamiento hash, obtenido con la función hash elegida, es `i`. Ahora, los elementos de la tabla dispersa se agrupan en listas enlazadas, por consiguiente, es necesario que dispongan de un campo adicional para poder enlazar con el *siguiente* elemento. La declaración de clase *Elemento*:

```
class Elemento
{
  //
    < atributos según los datos del elemento a representar >
  //

   Elemento * sgte;
};
```

La declaración de la *Tabla* se hace con un *array* de punteros a `Elemento` que son listas enlazadas.

```
class TablaDispersa
{
protected:
   int final;
   int numElementos;
   Elemento ** tabla;
public:
   TablaDispersa(int n);
   void insertar(TipoSocio s);
   Elemento* buscar(int codigo);
   void eliminar(int codigo);
```

```
private:
      // operaciones para obtener índices dispersos
      int dispersion(long x);
};
```

14.6.1. Operaciones de la tabla dispersa enlazada

Las operaciones fundamentales que se realizan con tablas dispersas, *insertar*, *buscar* y *eliminar*, se convierten, una vez obtenido el índice de la tabla con la función hash, en operaciones sobre listas enlazadas.

Para añadir un elemento a la tabla cuya clave es `x`, se inserta en la lista de índice `h(x)`. Puede ocurrir que la lista esté vacía, no se almacenó elemento con esa dirección dispersa, o bien que ya tenga elementos. En cualquier caso, la inserción en la lista se hace como primer elemento, de esa forma el tiempo de ejecución es de complejidad constante; entonces, se puede afirmar que la complejidad de la operación *inserción* es `O(1)`.

Ahora, un elemento de la tabla puede eliminarse *físicamente*, no es necesario marcar el elemento como *no activo*, como ocurre con las *secuencias de exploración*. Los pasos a seguir, primero se obtiene el índice de la tabla, `h(x)`, a continuación, se aplica la operación *eliminar* un nodo de una lista enlazada.

La búsqueda de un elemento sigue los mismos pasos, se obtiene el índice de la tabla mediante la función hash, a continuación, se aplica la operación *búsqueda* en listas enlazadas.

14.6.2. Análisis del direccionamiento enlazado

El planteamiento de *hashing encadenado* permite acceder directamente a la lista enlazada en la que se busca un elemento. La búsqueda es una operación lineal, en el peor de los casos recorre todos los nodos de la lista; a pesar de ello, si las listas tienen un número suficientemente pequeño el tiempo de ejecución será reducido.

Considerando el factor de carga, λ, la longitud media de cada lista enlazada (número de nodos) es, precisamente, λ. Esto permite concluir que el número medio de nodos visitados en una búsqueda sin éxito es λ, y el promedio de nodos visitados en una búsqueda con éxito es `1 + ½ λ`.

El factor de carga en estas tablas, normalmente es `1.0`, es decir, el rango de índices del vector coincide con el número de elementos que se espera sean insertados. A medida que aumenta el factor de carga la eficiencia de la búsqueda de un elemento en la lista disminuye. Sin embargo, no es conveniente que disminuya mucho el factor de carga, debido a que cuando éste es menor que `1.0` aumenta la memoria no utilizada.

> ***A tener en cuenta***
>
> El principal inconveniente del direccionamiento con encadenamiento es el espacio adicional de cada elemento, necesario para enlazar un nodo de la lista con el siguiente.

14.7. REALIZACIÓN DE UNA TABLA DISPERSA ENCADENADA

El desarrollo de las operaciones de una tabla dispersa encadenada se basa en las operaciones del TAD Lista. Los pasos que se siguen:

1. Declarar los tipos de datos, ajustándose a los elementos de la aplicación.
2. Definir la función hash con la que se van a obtener los índices de la tabla.
3. Inicializar la estructura, como un vector o array de listas vacías.
4. Definir la operación *insertar* un elemento. La inserción llama a la función que añade un nodo como primer elemento de la lista.
5. Definir la operación de *eliminar* un elemento. Esta operación busca el elemento dando su clave, antes de llamar a la función que borra un nodo de una lista enlazada, se pide que el usuario confirme la acción de quitar el nodo.
6. Definir la función de búsqueda en la tabla de un elemento. Para realizar la operación se pide la clave del elemento a buscar, si la búsqueda tiene éxito devuelve la dirección del nodo, en caso contrario `NULL`.

El desarrollo de las operaciones de la tabla dispersa se realiza en el contexto de una aplicación. Los elementos de la tabla son, a título de ejemplo, socios del club de montaña *La Parra*. Cada socio se identifica con estos atributos: *número de socio, nombre completo, edad, sexo, fecha de alta*. Se elige como clave el número de socio, que es un entero en el rango de `101` a `1999`. Actualmente, el club tiene `94` socios, entonces la tabla dispersa se diseña con el tamaño de `97` posiciones (no hay perspectivas de mucho crecimiento). El factor de carga, con este supuesto, estará en valores próximos a uno.

La función hash utiliza el método de la multiplicación, genera valores dispersos entre `0` y `96`. La declaración de la estructura `Fecha` es la siguiente:

```
struct Fecha
{
    int dia;
    int mes;
    int anno;
};
```

La declaración de la clase `TipoSocio`, es:

```
class TipoSocio
{
protected:
    int codigo;
    char * nombre;
    int edad;
    Fecha f;
public:
    TipoSocio(){};
    TipoSocio( int c, char *nom, int ed, Fecha fech)
        { codigo = c; nombre = nom; edad = ed; f = fech; }
    int Ocodigo(){return codigo;}
    void Pcodigo( int c) { codigo = c;}
    void Pedad( int e) { edad = e;}
    int Oedad(){ return edad;}
```

```
    void Pnombre(char * nom){ nombre = nom;}
    char * Onombre(){ return nombre;}
    void PFecha(Fecha fech){ f = fech;}
    Fecha OFecha(){ return f;}
};
```

La declaración de la clase `Elemento`:

```
class Elemento
{
protected:

    TipoSocio socio;
    Elemento *sgte;
public:
    Elemento(TipoSocio e)
    {
      socio = e;
      sgte = NULL;
    };
    Elemento(){};
    Elemento* Osgte(){ return sgte;}
    void Psgte( Elemento *sig){ sgte = sig;}
    TipoSocio Osocio(){ return socio;}
    void Psocio(TipoSocio s) { socio = s;}
};
```

Función hash, método de multiplicación

En el Apartado 14.2.4 se explica este método de generar índices dispersos aplicando el método de multiplicación. El rango de la dispersión depende del tamaño de la tabla, `M` o el valor de `final` en este caso. La clase declara el método `dispersion()`:

```
int TablaDispersa::dispersion(long x)
{
    double t, R = 0.618034;
    int v;
    t = R * x - floor(R * x);   // parte decimal
    v = (int) (final * t);
    return v;
}
```

Inicializar la tabla dispersa

La operación inicial que se hace con la tabla, antes de dar de alta elementos, consiste en establecer cada posición del array a la condición de lista vacía; la forma de asegurar que esta operación se realiza es incluyéndola en el constructor de `TablaDispersa`. La implementación reserva memoria con un bucle de tantas iteraciones como tamaño de la tabla, en el que se asigna `NULL` a cada posición de la tabla. Se fija el `numElementos` a 0.

```
TablaDispersa:: TablaDispersa(int n)   // constructor
{
    *tabla = new (Elemento *)[n];
    final = n;
    for (int k = 0; k < final; k++)
      tabla[k] = NULL;
    numElementos = 0;
}
```

14.7.1. Dar de alta en elemento en una tabla dispersa encadenada

Para dar de alta un elemento en la tabla, primero se determina el número de lista enlazada que le corresponde, según el índice que devuelve la función de dispersión. A continuación, se inserta como primer nodo de la lista. El modo de inserción elegido es el más eficiente, accede directamente a la posición de inserción, no necesita recorrer la lista, por ello la complejidad de la operación es `O(1)`.

La operación, en el contexto de almacenar socios, recibe a un objeto `Socio`; crea un objeto `Elemento` (nodo de la lista) con el `Socio` que es el que se inserta.

```
void TablaDispersa::insertar(TipoSocio s)
{
   int posicion;
   Elemento *nuevo, *p;
   posicion = dispersion(s.Ocodigo());
   p = buscar(s.Ocodigo());
   if(!p)// el código no se encuentra en la tabla
   {
      nuevo = new Elemento(s);
      nuevo->Psgte(tabla[posicion]);
      tabla[posicion] = nuevo;
      numElementos++;
   }
   else
      throw "repetición de socio";
}
```

14.7.2. Eliminar un elemento de la tabla dispersa encadenada

La supresión de un elemento se hace dando como entrada la clave del elemento, en el supuesto el número de socio. El argumento que recibe la operación, en el contexto que se ha supuesto para implementar la tabla, es el código del socio a dar de baja.

Con la función de dispersión se obtiene el número de lista donde se encuentra, para hacer una búsqueda secuencial dentro de la lista. Una vez encontrado el elemento, se muestra los campos y se pide *conformidad*. La retirada del nodo se hace enlazando el nodo anterior con el nodo siguiente, por ello la búsqueda mantiene en la variable `anterior` la dirección del nodo anterior al nodo actual.

La función que establece la conformidad con la eliminación del nodo no se escribe, simplemente mostrará el elemento y pedirá que se pulse una letra para indicar la acción a tomar.

El tiempo de ejecución de esta operación depende de la longitud de la lista enlazada en la que se busca el elemento; la longitud media de cada lista enlazada (número de nodos) es el factor de carga (consultar Apartado 14.6.2).

```
bool conforme ( TipoSocio cod){ }

void TablaDispersa::eliminar(int codigo)
{
    int posicion;
    posicion = dispersion(codigo);
    if (tabla[posicion] != NULL)// no está vacía
    {
        Elemento* anterior, *actual;
        anterior = NULL;
        actual = tabla[posicion];
        while ((actual->Osgte() != NULL)
              && actual->Osocio().Ocodigo() != codigo)
        {
          anterior = actual;
          actual = actual->Osgte();
        }
        if (actual->Osocio().Ocodigo() != codigo)
          cout << "No se encuentra en la tabla el socio \n";
        else if (conforme (actual->Osocio())){  //se retira el nodo
          if (anterior == NULL) // primer nodo
             tabla[posicion] = actual->Osgte();
          else
             anterior->Psgte(actual->Osgte());
          delete actual;
          numElementos--;
          {
    }
}
```

14.7.3. Buscar un elemento de la tabla dispersa encadenada

El algoritmo de búsqueda de un elemento es similar a la búsqueda que realiza la operación *eliminar* un elemento. El método `buscar()` devuelve la dirección del nodo que contiene la clave de búsqueda, en el contexto de tabla de socios se corresponde con el código de socio. Si no se encuentra en la lista enlazada devuelve `NULL`.

```
Elemento* TablaDispersa::buscar(int codigo)
{
    Elemento *p = NULL;
    int posicion;

    posicion = dispersion(codigo);

    if (tabla[posicion] != NULL)
    {
      p = tabla[posicion];
```

```
        while ((p->Osgte() != NULL)
              && p->Osocio().Ocodigo() != codigo)
          p = p->Osgte();
        if (p->Osocio().Ocodigo() != codigo)
          p = NULL;
      }
      return p;
  }
```

RESUMEN

Las tablas de dispersión son estructuras de datos para las que la complejidad de las operaciones básicas *insertar*, *eliminar* y *buscar* es constante. Cuando se utilizan tablas de dispersión los elementos que se guardan tienen que estar identificados por un campo clave, de tal forma que dos elementos distintos tengan claves distintas. La correspondencia entre un elemento y el índice de la tabla se hace con las funciones de transformación de claves, funciones hash, que convierten la clave de un elemento en un índice o posición de la tabla.

Dos criterios se deben seguir al elegir una *función hash*, `h(x)`. En primer lugar, la complejidad de la función `h(x)` sea constante y fácil de evaluar. En segundo lugar, `h(x)` debe de distribuir uniformemente las direcciones sobre el conjunto de índices posibles, de forma que se minimice el número de *colisiones*. El fenómeno de la *colisión* se produce si dadas dos claves diferentes `c1` y `c2`, la función hash devuelve la misma dirección, `h(c1) = h(c2)`. Aun siendo muy buena la distribución que realice `h(x)`, siempre existe la posibilidad de que colisionen dos claves, por ello el estudio del *direccionamiento disperso* se divide en dos partes: búsqueda de *funciones hash* y *resolución de colisiones*. Hay muchas funciones hash, la más popular es la *aritmética modular* que aplica la *teoría de restos* para obtener valores dispersos dentro del rango determinado por el tamaño de la tabla, que debe elegirse como un número primo.

Se consideran dos modelos para resolver colisiones: *exploración* de direcciones y *hashing* enlazado. Las tablas hash con exploración resuelven las colisiones examinando una secuencia de posiciones de la tabla a partir de la posición inicial, determinada por `h(x)`. El método *exploración lineal* examina secuencialmente las claves hasta encontrar una posición vacía (caso de insertar); tiene el problema de la agrupación de elementos en posiciones contiguas, que afecta negativamente a la eficiencia de las operaciones. El método *exploración cuadrática* examina las posiciones de la tabla `p, p+1, p+2`^{2}`, ..., p+i`^{2} considerando la tabla como un array circular. Para asegurar la eficiencia de la exploración cuadrática, el tamaño de la tabla debe elegirse como un número primo y el *factor de carga* no superar el 50 por 100.

Las tablas dispersas enlazadas se basan en utilizar listas (cadenas), de tal forma que en cada lista se colocan los elementos que tienen la misma dirección hash. Las operaciones *insertar*, *buscar*, *eliminar*, primero determinan el índice de la lista que le corresponde con la función hash, a continuación, aplican la operación correspondiente del *TAD* `Lista`. El factor de carga en las tablas enlazadas se aconseja que sea próximo a 1, en el caso de crezca mucho puede empeorar la eficiencia de la búsqueda al crecer el número de nodos de las listas.

Una de las aplicaciones de las tablas hash está en los compiladores de lenguajes de programación. La tabla donde se guardan los identificadores del programa, *tabla de símbolos*, se implementa como una tabla hash. La clave es el *símbolo* que se transforma en un valor entero para aplicar alguna de las funciones hash.

EJERCICIOS

14.1. Se tiene una aplicación en la que se espera que se manejen `50` elementos. ¿Cuál es el tamaño apropiado de una tabla hash que los almacene?

14.2. Una tabla hash se ha implementado con exploración lineal, considerando que actualmente el factor de carga es `0.30`. ¿Cuál es el número esperado de posiciones de la tabla que se prueban en una búsqueda de una clave, con éxito y sin éxito?

14.3. Para la secuencia de claves: `29, 41, 22, 31, 50, 19, 42, 38`; una tabla hash de tamaño `12`, y la función hash *aritmética modular*, mostrar las posiciones de almacenamiento suponiendo la exploración lineal.

14.4. Para la misma secuencia de claves del Ejercicio 14.3, igual tamaño de la tabla y la misma función hash, mostrar la secuencia de almacenamiento con la exploración cuadrática.

14.5. Se va a utilizar como función hash el método *mitad del cuadrado*. Si el tamaño de la tabla es de `100`, encontrar los índices que le corresponden a las claves: `2134, 5231, 2212, 1011`.

14.6. Para resolución de colisiones se utiliza el método de doble función hash. Siendo la primera función el método *aritmética modular* y la segunda el método de la *multiplicación*. Encontrar las posiciones que ocupan los elementos con claves: `14, 31, 62, 26, 39, 44, 45, 22, 15, 16` en una tabla de tamaño `17`.

14.7. Escribir el método `estaVacia()` para determinar que una tabla dispersa con exploración no tiene ningún elemento activo.

14.8. Escribir el método `estaVacia()` para determinar que una tabla dispersa enlazada no tiene elementos asignados.

14.9. Una tabla de dispersión con exploración tiene asignada n elementos. Demostrar que si el factor de carga es λ, y la función hash distribuye uniformemente las claves, entonces se presentan $(n - 1) * \lambda/2$ pruebas de posiciones de la tabla cuando se quiere insertar un elemento con una clave previamente insertada.

14.10. En el hashing enlazado la operación de insertar un elemento se ha realizado como primer elemento de la lista. Si la inserción en la lista enlazada se hace de tal forma que esté ordenada respecto la clave. ¿Qué ventajas y desventajas tiene respecto a la eficiencia de las operaciones *insertar, buscar y eliminar*?

14.11. Considérese esta otra estrategia para resolver colisiones en una tabla: en un vector auxiliar, *vector de desbordamiento*, se sitúan los elementos que colisionan con una posición ya ocupada, en orden relativo al campo clave. ¿Qué ventajas y desventajas pueden encontrarse en cuanto a la eficiencia de las operaciones de la tabla?

14.12. Emplear una tabla de dispersión con exploración cuadrática para almacenar 1.000 cadenas de caracteres. El máximo valor que se quiere alcanzar del factor de carga es 0.6, suponiendo que las cadenas tienen un máximo de 12 caracteres calcular:

a) El tamaño de la tabla.
b) La memoria total que necesita la tabla.

PROBLEMAS

14.1. Escribir la función `posicion()`, que tenga como entrada la clave de un elemento que representa los coches de alquiler de una compañía, devuelva una dirección dispersa en el rango `0 .. 99`. Utilizar el método *mitad del cuadrado*, siendo la clave el número de matrícula pero sólo considerando los caracteres de orden par.

14.2. Escribir la función `plegado()`, para dispersar en el rango de 0 a 997 considerando como clave el número de la seguridad social. Utilizar el método de plegamiento.

14.3. Se desea almacenar en un archivo los atletas participantes en un *cross* popular. Se ha establecido un máximo de participantes, `250`. Los datos de cada atleta: *Nombre, Apellido, Edad, Sexo, Fecha de nacimiento y Categoría (Junior, Promesa, Senior, Veterano)*. Supóngase que el conjunto de direcciones del que se dispone es de `400`, en el rango de `0` a `399`. Se elige como campo clave el *Apellido* del atleta. La función hash a utilizar es la de *aritmética modular*, antes de invocar a la función hash es necesario transformar la cadena a valor numérico (considerar los caracteres de orden impar, con un máximo de 5). Las colisiones se pueden resolver con el método de *exploración (prueba) lineal*.

Las operaciones que contempla el ejercicio son las de dar de alta un nuevo registro; modificar datos de un registro; eliminar un registro y búsqueda de un atleta.

14.4. Las palabras de un archivo de texto se quieren mantener en una tabla hash con *exploración cuadrática* para poder hacer consultas rápidas. Los elementos de la tabla son la cadena con la palabra y el número de línea en el que aparece, sólo se consideran las 10 primeras ocurrencias de una palabra. Escribir un programa que lea el archivo, según se vaya capturando una palabra, y su número de línea, se insertará en la tabla dispersa. Téngase en cuenta que una palabra puede estar ya en la tabla, si es así se añade el número de línea.

14.5. Para comparar los resultados teóricos de la eficiencia de las tablas de dispersión, escribir un programa que inserte 1.000 enteros generados aleatoriamente en una tabla dispersa con *exploración lineal*. Cada vez que se inserte un nuevo elemento contabilizar el número de posiciones de la tabla que se exploran. Calcular el número medio de posiciones exploradas para los factores de carga: 0.1, 0.2, ... 0.9.

14.6. Escribir un programa para comparar los resultados teóricos de la eficiencia de las tablas de dispersión con *exploración cuadrática*. Seguir las pauta marcadas en el Problema 14.5.

14.7. Determinar la lista enlazada más larga (mayor número de nodos) cuando se utiliza una tabla hash enlazada para guardar 1.000 números enteros generados aleatoriamente. Escribir un programa que realice la tarea propuesta, considerar que el factor de carga es 1.

14.8. La realización de una tabla dispersa con exploración puede hacerse de tal manera que el tamaño de la tabla se establezca dinámicamente. Con los vectores dinámicos se puede hacer una *reasignación* de tal forma que si el factor de carga alcanza un determinado valor, se amplíe el tamaño en una cantidad determinada y se vuelvan a asignar los elementos, pero calculando un nuevo índice disperso ya que el tamaño de la tabla ha aumentado. Se pide escribir una función que implemente este tipo de *reasignación*.

14.9. Para una tabla dispersa con exploración escribir una función que determine el numero de elementos dados de baja.

14.10. Partiendo de la función escrita en el Problema 14.9, añadir el código necesario para cuando el número de elementos eliminados supere el `10` por `100` se produzca una *reasignación* de tal forma que el factor de carga supere el `40 por 100`.

14.11. En una tabla de dispersión enlazada la eficiencia de la búsqueda disminuye según aumenta la longitud media de las listas enlazadas. Se quiere realizar una *reasignación* cuando la longitud media supere un factor determinado. La tabla dispersa, inicialmente es de tamaño `M`, la función de dispersión es aritmética modular: `h(x) = x modulo m`. Cada ampliación incrementa el tamaño de la tabla en `1`, así, en la primera el número de posiciones será `M+1`, en el rango de `0` a `M`; por esa razón, se crea la lista en la posición `M`, y los elementos que se encuentran en la lista 0 se dispersan utilizando la función h_1`(x) = x modulo 2* M`. La segunda ampliación supone que el tamaño de la tabla sea `M` +2, en el rango de `0` a `M +1`, entonces se crea la lista de índice `M` +1, y los elementos que se encuentran en la lista 1 se dispersan utilizando la función h_2`(x) = x modulo 2 * (M + 1)`. Así sucesivamente, se va ampliando la estructura que almacena los elementos de una tabla dispersa enlazada.

Escribir las funciones que implementan la operación *insertar* siguiendo la estrategia indicada.

14.12. Escribir un programa que utilice la tabla hash enlazada descrita en el Problema 14.11 para guardar 1.000 números enteros generados aleatoriamente. Inicialmente, el tamaño de la tabla es 50, cada vez que la longitud media de las cadenas (listas enlazadas) sea 4.5 se debe ampliar la tabla dispersa según el método descrito en el Problema 14.11.

CAPÍTULO 15

Biblioteca estándar de plantillas (STL)

Objetivos

Con el estudio de este capítulo usted podrá:

- Conocer el concepto de biblioteca de plantillas.
- Generalizar el concepto de apuntador mediante los iteradores.
- Aplicar los iteradores a los contenedores estándar.
- Conocer las funciones miembro más importantes de los contenedores.
- Implementar listas, vectores, pilas, colas, dobles colas, colas de prioridad mediante las STL.

Contenido

Conceptos clave

- *Adaptadores*.
- Algoritmos.
- Biblioteca STL
- Clases contenedoras (*contenedores*).
- Interador.

INTRODUCCIÓN

En el mes de abril de 1995, los comités ANSI e ISO C++ publicaron su primer documento oficial, el Committe Draft (CD), para estudio y revisión de la comunidad de desarrollo informática internacional. Las grandes diferencias entre el *"ARM C++"* y el *"Standard C++"* no residen en el lenguaje, sino en las características disponibles en la biblioteca. La biblioteca *C++ Standard*, conocida como *Standard Template Library (STL)*, es una biblioteca que incluye clases contenedoras y algoritmos basados en plantillas. **STL** es una biblioteca muy importante para el desarrollo de programas profesionales dado que facilita considerablemente el diseño y construcción de aplicaciones basadas en estructuras de datos tales como vectores, listas, conjuntos y algoritmos.

15.1. BIBLIOTECA STL: CONCEPTOS CLAVE

La biblioteca de plantillas estándar (**STL**) es la biblioteca estándar de C++ que proporciona programación genérica para muchas estructuras de datos y algoritmos. Los componentes fundamentales de STL se dividen en las siguientes categorías importantes:

- Clases contenedoras (o simplemente, *contenedores*).
- *Iteradores*, que permiten hacer recorridos a través de los contenedores.
- Algoritmos.
- Objetos función (*funciones*), que encapsulan funciones en objetos para ser utilizados por otros componentes.
- *Adaptadores*, que adaptan componentes para proporcionar un interfaz diferente y adecuado para otros componentes.

En realidad, desde un punto de vista práctico, STL incluye clases contenedoras de plantilla para almacenar objetos, clases iteradoras de plantilla para acceder a objetos en el interior de los contenedores y algoritmos genéricos que manipulan objetos a través de iteradores. Así, la Figura 15.1 muestra un diagrama más detallado de los componentes de STL.

La parte inferior de la Figura 15.1 es una descripción más detallada de los tres componentes de la parte superior. En ambas partes de la figura, los iteradores se muestran en el centro, proporcionando un conector "compatible" y conectable entre contenedores y algoritmos. Por ejemplo, el contenedor `lista` es compatible y conectable con iteradores `bidireccionales`. Las implementaciones STL pueden conseguirse lo más eficiente posible. Así, por ejemplo, `sort()` genérico requiere iteradores de acceso aleatorio para eficiencia de implementación. El algoritmo `sort()`, sin embargo, excluye los contenedores `lista` (*lista*), `multiconjunto` (*multiset*), `conjunto` (*set*), `multimapa` (*multimap*) y `mapa` (*map*) (que son compatibles y conectables con iteradores bidireccionales). Todos estos contenedores (excepto *lista*) mantienen el orden en cualquier punto. El contenedor lista proporciona su propia función miembro `sort()`.

Al igual que sucede en cualquier programa C++, se accede a la biblioteca STL de igual modo que se accede a la biblioteca estándar clásica, es decir, mediante una sentencia `#include`. Los interfaces a los componentes STL están contenidos en un conjunto de archivos de cabecera. Cada archivo de cabecera representa un conjunto de clases que encapsulan ciertas estructuras de datos y los algoritmos asociados, iteradores, etc.

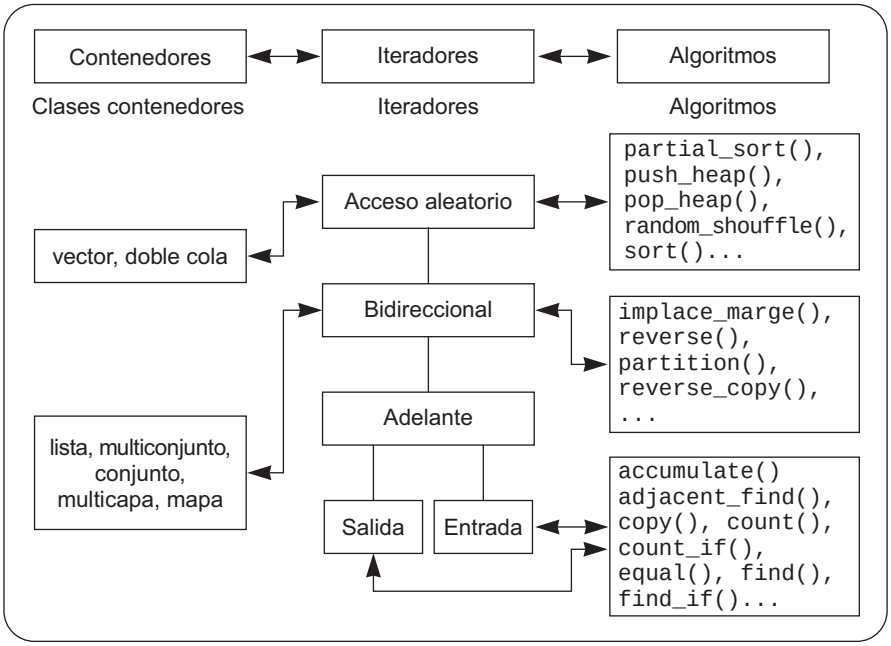


Figura 15.1.

Con el fin de utilizar componentes STL en sus programas, se debe utilizar la ya citada directiva del preprocesador `#include` para incluir uno o más archivos de cabecera. Las normas de la organización de archivos de cabecera son, esencialmente, las siguientes:

1. La clase contenedora de STL, denominada `c,` reside en `<c>`; por ejemplo, vector está en `<vector>`, `list` está en `<list>`, etc.
2. Los adaptadores de contenedores `(stack, queue y priority_queue)` están en `<stack>`.
3. Todos los algoritmos genéricos STL están en `<algo>`.
4. Las clases iteradoras de flujo STL y los adaptadores de iteradores están en `<iterator>`.
5. Los objetos función y adaptadores de función están en `<function>`.

El estándar ANSI/ISO C + + no utiliza la extensión .h de la biblioteca C++, excepto aquellos que están también en cabeceras de la biblioteca C. Otras diferencias de la organización, descritas anteriormente, son las siguientes:

1. Tanto `set` como `multiset` están en `<set>`, y `map` y `multimap` están en `<map>`.
2. El adaptador `stack` (pila) está en `<stack>`, y los adaptadores `queue` (cola) y `priority_queue` (cola de prioridad) están en `<queue>`.
3. Todos los algoritmos genéricos están en `<algorithm>`, excepto los algoritmos numéricos generalizados que están en `<numeric>`.
4. Los objetos función y los adaptadores de función STL están en `<functional>`.

Tabla 15.1. Archivos cabecera más notables de la biblioteca STL

Archivo cabecera	Contenido
`algorithm, algo`	Algoritmos de plantilla.
`deque`	El contenedor *deque.*
`function, functional`	Los objetos función de la plantilla.
`iterator`	Iteradores.
`list`	El contenedor lista.
`map`	Los contenedores mapa y multimapa.
`memory`	Asignadores de gestión de memoria.
`queue`	Los contenedores cola y cola de prioridades.
`set`	Los contenedores conjunto y multiconjunto.
`stack`	El contenedor pila.
`string`	La cadena ANSI mediante plantillas.
`utility`	Los operadores relacionales mediante plantillas.
`vector`	El contenedor vector.

EJEMPLO 15.1. Estructura de una pila. Introduce los datos 45, 110, y 51 y visualiza 51, 110, 45.

```
#include <cstdlib>
#include <iostream>
#include <stack> // STL pila
using namespace std;
int main()
{
   stack<int> s;                              // pila de enteros
   s.push (45); s.push (110); s.push (51);  // añade a la pila
   while (!s.empty())          // mientras no esté vacía la pila hacer
   {
     cout << s.top () << endl;              // escribir la cumbre
     s.pop();                               // borrar elemento de la pila
   }
    return 0;
}
```

15.2. CLASES CONTENEDORAS

El componente principal de STL es la familia de clases contenedoras. *Contenedores* son objetos genéricos que se utilizan para almacenar otros objetos. Ejemplos de contenedores son: listas, vectores, etc.

Un contenedor es un modo de almacenar datos en memoria. Las estructuras de datos o contenedores, almacenan piezas de información, o elementos, de un modo tal que permiten acceso apropiado. Las diferentes estructuras de datos tienen características y prestaciones adecuadas para insertar, borrar, desplazar los elementos. Es importante que el lector se familiarice con las estructuras de datos disponibles de modo que pueda elegir la más adecuada a la tarea a realizar.

Las clases contenedoras (también denominadas *clases colección*) proporcionan normalmente servicios tales como inserción, borrado, búsqueda, ordenación, pruebas o comparación de un elemento con un miembro de la clase, y similares. Los arrays (vectores, lista, matrices, "arreglos"), pilas, colas y listas enlazadas son ejemplos de clases contenedoras. La especificación STL define los contenedores de la forma siguiente:

> "Contenedores son objetos que almacenan otros objetos . Controlan la asignación y liberación de estos objetos a tr avés de constr uctores, destr uctores, operaciones de insertar y borrar."

Todos los contenedores de STL son plantillas, de modo que puede utilizarlas, almacenar cualquier tipo de datos, desde tipos incorporados como `int` y `double` a sus propias clases. Obsérvese, que se deben almacenar elementos del mismo tipo en cualquier contenedor dado. Es decir, no se pueden almacenar elementos `int` y `double` en la misma cola. Sin embargo, se pueden crear para operar con los dos tipos de datos, dos colas independientes.

> "Los contenedores de STL son homogéneos: per miten elementos de solo un tipo de datos en cada contenedor".

La mayoría de los programadores han estudiado, o están familiarizados, con estructuras de datos clásicas tales como arrays, listas enlazadas y árboles. La STL ha tomado algunas de las estructuras de datos más útiles y ha creado implementaciones eficientes y fáciles de utilizar. La declaración de un contenedor en un programa define el tipo de objeto que contiene.

La biblioteca STL proporciona servicios para diez categorías básicas de contenedores:

- `vector<T>`: array (lista) secuencial.
- `deque<T>`: cola de doble entrada, secuencial, por ejemplo, LIFO.
- `list<T>`: lista doblemente enlazada.
- `set<clave, comparar>`: conjunto de elementos a los que se accede con la clave de búsqueda asociativa.
- `multiset<clave, comparar>`: igual que `set`, pero puede contar múltiples copias del mismo elemento.
- `map<clave, T, comparar>`: colección de correspondencias (mapas) de 1 : 1 entre clave y objeto.
- `multimap<clave, T, comparar>`: colección de *1:N* correspondencias, con claves múltiples.
- `stack<T, contenedor <T>>`: adaptador para utilizar un contenedor como una pila.
- `queue<T,Contenedor<T>>`: cola; solo FIFO.
- `priority_queue<contenedor<T>`, comparar>: cola FIFO ordenada.

15.2.1. Tipos de contenedores

STL incluye tres categorías de contenedores:

1. *Secuencial*, contiene los datos organizados de modo lineal, como un array (lista, arreglo) o una lista enlazada.

`vector<T>`	Vector
`deque<T>`	Doble cola
`list<T>`	Lista

2. *Asociativo*, contiene los datos organizados basándose en la consulta o búsqueda mediante una clave.

`set<c1ave, comparar>`	Conjunto
`multiset<clave, comparar>`	Multiconjunto
`map<c1ave, T, comparar>`	Mapa
`multimap<clave, T, comparar>`	Multimapa

3. *Secuencial adaptativo*, que proporciona un interfaz "adaptado" a un contenedor existente, por ejemplo, una cola con prioridad.s

`stack<Container <T>>`	Pila
`queue<Container <T>>`	Cola
`priority_queue<Container <T>, comparar>`	Cola de prioridad

15.2.2. Contenedores secuenciales

El contenedor agrupa a todos sus miembros como una sucesión lineal y, en consecuencia, son adecuadas para accesos directos y secuenciales. Los contenedores secuenciales contienen datos organizados como un array o lista enlazada. Es fácil recorrer la lista y acceder a miembros dato secuencialmente. Si se accede a un objeto en un punto dado en un contenedor de secuencia, es sencillo moverse al elemento siguiente o al anterior de la lista. Los tres contenedores de secuencia son:

- `vector<T>`. *Vector*, es un array estándar que puede cambiar el tamaño de modo transparente a medida que se necesite.
- `deque<T>`. *Doble cola*, es una estructura de datos cola que inserta y elimina objetos por cualquiera de los extremos.
- `list <T>`. *Lista*, es una lista doblemente enlazada.

Un ejemplo de un programa corto escrito con un contenedor de secuencia es:

```
#include <iostream>
#include <vector>
using namespace std;

int main ()
{
    vector <char *> v;
    v.insert(v.end(), "Mazarambroz");
    v.insert(v.end(), "Ajofrin");
    v.insert(v.end(), "Sonseca");
    v.insert(v.end(), "Chueca");
```

```
    for (int i = 0; i < 4; i++)
       cout << v[i] << " ";
    return 0;
}
```

Este programa visualiza `Mazarambroz Ajofrin Sonseca Chueca`.

Función miembro	Propósito
Comunes a todos los contenedores STL	
`default constructor`	Constructor por omisión.
`copy const, operador=()`	Constructor de copia y operador de asignación.
`==, !=, <, >, <=, >=`	Operadores de comparación.
`bool empty() const`	Devuelve true si contenedor vacío.
`size_tipe size() const`	Devuelve número de elementos del contenedor.
`size_tipe max_size()const`	Devuelve número máximo de elementos.
Comunes a `vector, list, deque, set, multiset, map, multimap`	
`begin()`	Devuelve un iterador que apunta al primer elemento del contenedor.
`end()`	Itera después del último elemento del contenedor.
`rbegin()`	Iterador inverso al último elemento del contenedor.
`rend()`	Iterador inverso antes del primer elemento del contenedor.
`insert()`	Familia de funciones de inserción en distintas posiciones.
`erase(iterador i)`	Familia de funciones de borrado.
Comunes sólo a `vector, list, deque`	
`front()`	Primer elemento.
`back()`	Último elemento.
`push_back (const T& valor)`	Inserta por detrás.
`pop_back()`	Elimina por detrás.

15.2.3. Contenedores asociativos

Los contenedores asociativos almacenan objetos basados en un valor clave; los objetos se pueden recuperar rápidamente mediante el mismo valor de clave. Los contenedores asociativos almacenan sus miembros en forma de árbol indexado, por lo que son denominados también contenedores asociativos ordenados, y resultan adecuados para accesos aleatorios mediante claves.

STL define cuatro tipos diferentes de contenedores para implementar contenedores asociativos:

- `set<c1ave>`
- `multiset<clave>`
- `map<clave,T>`
- `multimap<clave,T>`

Estos cuatro tipos de contenedor se implementan normalmente como árboles binarios o como tablas de dispersión (*hash*) con recorridos rápidos entre nodos y hojas. Un ejemplo de uso de un contenedor asociativo incluye la localización de un objeto `Persona` asociado con un número de la Seguridad Social como clave.

Los cuatro contenedores diferentes se crean variando dos características. La diferencia entre conjuntos y mapas es que un mapa consta de una colección de objetos datos que pueden ser referenciados por una clave, mientras que un conjunto es simplemente una colección ordenada de claves. La diferencia entre implementaciones singular y múltiples de conjuntos y mapas es que las versiones singulares `set<c1ave> y map<clave, T>` sólo contienen una única copia de una clave dada; `multiset<clave> y multimap<clave, T>` permiten copias múltiples de una única clave.

Un ejemplo de cómo se pueden utilizar arrays asociativos en un programa es:

```
#include <iostream>
#include <map>
using namespace std;
int main ()
{
    map <string, int, less<string> > m;
    m["Mckoy"] = 1;
    m["Carrigan"] = 3;
    m["Mackena"] = 5;
    m["Mortimer"] = 7;
    cout << "Valor de Mortimer = " << m["Mortimer"] << endl;
    return 0;
}
```

Los contenedores asociativos son útiles siempre que se necesita mantener una estructura de datos con objetos ordenados. Es fácil mantener una lista de datos de empleados, números de teléfono, ciudades, etc., utilizando contenedores asociativos.

Tabla 15.3. Funciones miembro comunes a contenedores asociativos.

Función miembro	Propósito
Comunes a todos los contenedores STL	
`default constructor`	Constructor por omisión.
`copy constr, operator=()`	Constructor de copia y operador de asignación.
`==, !=, <, >, <=, >=`	Operadores de comparación.
`empty()`	Devuelve `true` si contenedor vacío.
`size()`	Devuelve número de elementos del contenedor.
`max_size()`	Devuelve número máximo de elementos.
Comunes a `set, multiset, map, multimap`	
`begin()`	Devuelve un iterador al primer elemento de la lista.
`end()`	Devuelve un iterador que apunta después del último contenedor.
`rbegin()`	Devuelve un iterador inverso al último elemento del contenedor.
`rend()`	Devuelve un iterador antes del primer elemento del contenedor.
`insert()`	Inserta un valor en una posición.
`erase()`	Elimina elementos de un rango.
`key_comp()`	Devuelve clave comparación.
`value_comp()`	Devuelve objeto comparación.
`find()`	Busca elemento basado en clave.
`lower_bound()`	Busca primera posición a insertar.
`upper_bound()`	Busca última posición a insertar.
`count()`	Devuelve número de elementos que coincide con la clave.

15.2.4. Adaptadores de Contenedores

La biblioteca STL introduce un nuevo tipo de componente denominado **adaptador** *(adaptador).* Los adaptadores se definen en la especificación de STL como "una clase plantilla que proporciona correspondencias de interfaces". STL tiene tres adaptadores de contenedor.

- `stack <Container>`
- `queue <Container>`
- `priority_queue <Container>`

El adaptador *pila* (`stack <Contenedor>`) implementa una simple pila LIFO con operaciones de *poner* y *quitar.* El adaptador *cola* (`queue <Contenedor>`) implementa la estructura cola (FIFO). Por último, *cola_ de_prioridad* (`priority_queue <Contenedor>`) implementa una cola cuyos elementos se eliminan en orden de prioridad. Estos adaptadores crean nuevos contenedores a partir de tipos contenedor existentes. Esto significa que se puede crear una pila (`stack <Contenedor>`) a partir de un tipo `vector<T>`, `deque<T>` o `list<T>`. De modo similar, las colas y las colas de prioridad se pueden crear a partir de otros contenedores de secuencia.

Tabla 15.4. Funciones miembro comunes a adaptadores STL.

Función miembro	Propósito
`push()`	Añadir elemento.
`pop()`	Quitar (borrar) elemento.

El uso de un adaptador de contenedor se muestra en el siguiente ejemplo. Este programa utiliza las operaciones miembro `push()` *(meter),* `pop()` *(sacar)* y `top()` (*cima*).

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

int main()
{
    stack <float,vector <float> > s;
    s.push (23.5 ); s.push ( 14.3); s.push (20.15);
    while (!s.empty())
    {
      cout << s.top () << endl;
      s.pop ();
    }
    return 0;
}
```

15.3. ITERADORES

Un **iterador** es un puntero inteligente; apunta a objetos de un contenedor. Los iteradores se utilizan como marcadores de índice para proporcionar la posición de un elemento en una estructura de datos. Un iterador puede avanzar o retroceder con operaciones de incremento o

decremento; es decir, sirve para recorrer un contenedor, un array en "C" o un *iostream* en C + +. La Figura 15.2 muestra un ejemplo del uso de iteradores. El contenedor `deque<T>` (al igual que otros contenedores STL) tiene una función miembro `begin()` y `end()`. Cada una de estas funciones devuelve un iterador que apunta al principio y al final del objeto `deque`.

Un iterador, en esencia, es un objeto que devuelve el siguiente elemento de una colección de clase o de una clase contenedora (o ejecuta alguna acción sobre el siguiente elemento de una colección). Los iteradores se escriben normalmente como amigas de las clases a las que iteran; esto permite a los iteradores tener acceso directo a los datos privados de las clases a las que iteran. Al igual que un libro puede ser compartido por varias personas y puede tener varias marcas de lectura a la vez, una clase contenedora puede tener varios iteradores funcionando a la vez. Cada iterador mantiene su propia información de"posición".

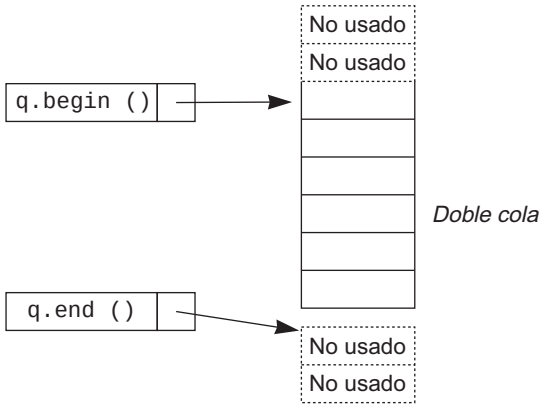


Figura 15.2. Iterador de una doble cola.

15.3.1. Categorías de los iteradores

STL define cinco categorías principales de iteradores: entrada, salida, avance bidireccional y acceso aleatorio.

Tabla 15.5. Tipos de iteradores.

Iterador	Funcionalidad
entrada	Puede leer un elemento cada vez, sólo en dirección directa (adelante).
salida	Puede escribir un elemento cada vez, sólo en dirección directa (adelante).
adelante	Combina características de iteradores de entrada y salida.
bidireccional	Igual que *adelante*, más la capacidad de moverse hacia atrás.
acceso aleatorio	Igual que bidireccional, más la capacidad de saltar una distancia arbitraria.

Estas categorías de iteradores se caracterizan por las siguientes propiedades:

- Los iteradores de entrada permiten algoritmos para avanzar el iterador y proporcionar acceso de *sólo lectura* al valor.

- Los iteradores de salida permiten algoritmos para avanzar el iterador y proporciona acceso de *sólo escritura* al valor.
- Iteradores de avance (adelante, *forward*) combinan el acceso de lectura y escritura, pero sólo en una dirección.
- Iteradores bidireccionales permiten a los algoritmos recorrer la secuencia en ambas direcciones, adelante y atrás.
- Iteradores de acceso aleatorio permiten saltos y aritmética de punteros.

El acceso a los interfaces de iterador están disponibles en el archivo de cabecera *<iterator>*. El formato de un iterador de STL es:

```
contenedor <TIPO>::iterator iter; //definición de iterador
```

Algunos ejemplos de uso de iteradores son:

```
list<int>::iterator iter;          //iterator a list<int>
vector<int>::iterator y;           //iterador a vector<int>
list<string>::reverse_iterator r;  //iterador inverso
map<int, string>::iterator m;      //iterador a map<int, string>
deque<char>::const_iterator c;     //const iterator para deque<char
```

La Tabla 15.6 lista las cabeceras más frecuentes que se han de incluir para las diferentes estructuras de datos contenedores.

Tabla 15.6. Archivos de cabecera y clases contenedoras.

#include	Clases contenedoras
<bitset>	bitset
<deque>	deque
<1ist>	list
<map>	map, multimap
<queue>	queue, priority_queue
<set>	set, multiset
<stack>	stack
<vector>	vector, vector<bool>

15.3.2. Comportamiento de los iteradores

Los interfaces proporcionados por las diferentes categorías de iteradores son sencillos. Suponiendo que `i` y `j` son iteradores, y `n` es un entero, la lista de todas las funciones de iterador dispuestas del orden más general al orden más específico.

Comunes a todos los tipos de iteradotes

`++i` Avanza un elemento y devuelve una referencia a `i`.
`i++` Avanza un elemento y devuelve el valor anterior de `i`.

Entrada

`*i` Devuelve una referencia de sólo lectura al elemento en la posición actual de `i`.
`i==j` Devuelve verdadero si `i` y `j` están ambas posicionadas en el mismo elemento.

Salida

`*i` Devuelve una referencia de escritura al elemento en la posición actual de `i`.
`i=j` Fija la posición de `i` a la misma que la de `j`.

Bidireccional

`--i` Retroceder un elemento y devolver el nuevo valor de `i`.
`i--` Retroceder un elemento y devolver el valor anterior de `i`.

Acceso aleatorio

`i+=n` Avanza `n` posiciones y devuelve una referencia a `i`.
`i-=n` Retrocede n posiciones y devuelve una referencia a `i`.
`i+n` Devuelve un iterador que está posicionado `n` elementos delante de su posición actual.
`i-n` Devuelve un iterador que está posicionado `n` elementos detrás de su posición actual.
`i[n]` Devuelve una referencia al n-ésimo elemento de su posición actual.

Obsérvese, que de acuerdo a esta jerarquía de interfaces, un puntero se considera un iterador de acceso aleatorio.

15.3.4. Iteradores definidos en cada contenedor.

Un iterador es un tipo de dato que permite el recorrido y la búsqueda de elementos en los contenedores. El concepto resulta ser una especie de punteros que señalan a los diversos miembros del contenedor (punteros genéricos que como tales no existen en el lenguaje). Aunque los punteros ordinarios son iteradores, es más frecuente utilizar iteradores generados por una clase contenedora. Estos iteradores llamarán a funciones contenedoras tales como `begin` y `end` para obtener iteradores de una colección específica de datos.

```
#include <cstdlib>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> VectorEnt(100);
    VectorEnt[10] = 30;
    vector<int>::iterator IterEnt;
    IterEnt = find(VectorEnt.begin(), VectorEnt.end(), 30);
    if (IterEnt != VectorEnt.end())
       cout << "El vector tiene el valor " << *IterEnt << endl;
```

```
    else
       cout << "El vector no contiene 30" << endl;
    return 0;
}
```

El programa define un vector de los `100` valores enteros que utilizan la plantilla de clase contenedora estándar `<vector>`. Las sentencias

```
vector<int>::iterator IterEnt;
IterEnt = find(VectorEnt.begin(), VectorEnt.end(), 30);
```

Construyen un iterator y llaman a `find`. El iterador `IterEnt` se define utilizando el tipo `iterator` de la plantilla clase contenedora `vector<int>`. Esto permite a `IterEnt` para acceder a los datos del contenedor. El programa pasa a `find` otros dos iteradores proporcionados por las funciones `begin` y `end` del contenedor. La función `begin` devuelve un iterador al primer elemento del contenedor. La función `end` devuelve un valor después del final.

15.4. CONTENEDORES ESTÁNDAR

Las clases contenedoras de STL ofrecen una amplia gama de soluciones para aplicaciones. La Tabla 15.7 lista las 10 plantillas de clases contenedoras de la biblioteca estándar dispuestas en orden de complejidad.

Tabla 15.7. Plantillas de clase contenedoras estándar.

Plantilla de clases	Descripción
`vector`	Secuencia lineal, similar a un array C ++.
`list`	Lista doblemente enlazada.
`deque`	Cola con doble extremo.
`set`	*Array* asociativo de claves únicas.
`bitset`	Una forma especial de conjunto capaces de contener valores binarios se denomina un conjunto de bit.
`multiset`	*Array* asociativo de claves posiblemente duplicadas.
`map`	*Array* asociativo de claves y valores únicos.
`multimap`	*Array* asociativo de claves y valores posiblemente duplicados.
`stack`	Estructura de datos LIFO (último-en-entrar, primero-en-salir).
`queue`	Estructura de datos FIFO (primero-en-entrar, primero-en-salir).
`priority_queue`	Vector o cola ordenada por sucesos críticos.

Las siguientes reglas se han de seguir con los contenedores:

- Un contenedor puede almacenar cualquier tipo de datos, incluyendo objetos tales como enteros (`int` y `double`), cadenas (*strings*), estructuras, objetos de clases y punteros. Sin embargo, un contenedor no puede almacenar referencias.
- Los contenedores automáticamente asignan y liberan memoria a medida que se necesitan para almacenamiento de objetos.

- Un vector puede crecer para acomodar nuevos datos, pero no puede disminuir de tamaño. Otros contenedores tales como listas, dobles colas y conjuntos pueden crecer y decrecer a medida que se necesite.

15.5. VECTOR

Un vector es similar a un array inteligente que mantiene información del tamaño y puede crecer para hacer espacio para almacenar más datos. Los vectores proporcionan acceso aleatorio a sus datos a través del operador de subíndice ([ ]). Proporciona también un número de funciones miembro útiles que se pueden llamar. Los vectores se destacan por proporcionar acceso aleatorio inmediato a todos los elementos dato. Los elementos de un vector se almacenan físicamente de forma contigua, por lo que las inserciones en el centro de un vector originan que otros elementos se muevan y su tiempo de ejecución es proporcional al número de elementos. Las inserciones al final de un vector son eficientes, a menos que la inserción requiera expandir el vector. Se dice también que cuando un vector se expande, sus datos deben ser copiados a otro bloque de memoria. Por consiguiente, las expansiones del vector pueden requerir temporalmente más de dos veces la cantidad de memoria ocupada. Si su programa hace muchas inserciones, se puede desear utilizar una lista u otra estructura de datos eficiente en memoria.

15.5.1. Declaración de vectores

La declaración más común de un objeto de tipo vector se realiza de la siguiente manera: *vector<tipo> objeto; o vector<tipo> objeto(numero)*;

Como por ejemplo

```
vector<int> vecent1 // declara un vector de enteros vacío
vector<string> vstring1(10); // declara un vector de 10 string
vector<double> vdouble(100); // vector de 100 reales dobles
vector<double> vdouble(100,-10) // vector inicializado a -10
```

15.5.2. Funciones miembro y operadores más comunes en los vectores

Alguna de las funciones miembro más utilizadas con su descripción es:

Función	Prototipo	Descripción
empty	`bool empty() const.`	Devuelve `true` si el número de elementos es cero y `false` en caso contrario.
push_back	`void push_back(const T& x).`	Añade un elemento `x` de tipo `T` al final del vector.
begin	`iterator begin().`	Devuelve un iterador que referencia el comienzo del vector.

end	`iterator end().`	Devuelve un iterador que referencia la posición siguiente al final del vector.
erase	`void erase(iterator p, iterator u).`	Borra los elementos del vector que estén situados entre los iteradores p y u.
size	`size_type size() const.`	Devuelve el número de elementos almacenados en el vector.
capacity	`size_type capacity() const.`	Devuelve el número de elementos con que se ha creado el vector.

La Tabla 15.8 muestra todas las funciones miembro que actúan sobre un vector.

Tabla 15.8. Funciones miembro que actúan sobre vector

operador ==	operador !=	operador >	operador <
operador <=	operador >=	operador =	operador []
begin	end	rbegin	rend
empty	size	max_size	front
back	push_back	pop_back	swap
insert	erase	capacity	reverse

EJEMPLO 15.2. Uso de la STL vector cuando se necesita un vector de cadenas de caracteres.

En el ejemplo se *instancia* el `vector <T>` con el tipo de una cadena de caracteres y se insertan cuatro cadenas al final del array utilizando el iterador de insertar objetos simples. Obsérvese, que se ha utilizado el operador de índice `operator [ ] ()` para moverse a través del array de cadenas. En el ejemplo se muestra cómo se pueden utilizar subíndices para asignar y referirse a datos en contenedores vectores:

```
#include <cstdlib>
#include <iostream>
#include <vector>
using namespace std;
const int MaxStr = 4;
int main()
{
    vector <char*> vec;                // crea un vector vacío
    vec.insert(vec.end(), "Mazarambroz");
    vec.insert(vec.end(), "Ajofrin");
    vec.insert(vec.end(), "Sonseca");
    vec.insert(vec.end(), "Toledo");
    for (int i = 0; i < MaxStr; ++i)   // recorre el vector y lo muestra
       cout << " " << vec[i] << endl;
  return 0;
}
```

El programa anterior muestra en pantalla:

Mazarambroz
Ajofrín
Sonseca
Toledo

EJEMPLO 15.3. Uso de algunas de las funciones miembro de la STL v ector.

El programa crea un vector dinámico al que se le añaden datos siempre por el final. En caso de que se termine la capacidad del vector se duplica para poder almacenar nuevos datos.

```
# include <iostream>
# include <vector>
using namespace std;

int main()
{
    vector<int> venteros(4);
    int dato;
    int contador = 0;
    cout << "los datos que introduzca se añaden al final del vector\n";
    do {
      cout << " introduzca dato -1 fin ";
      cin >> dato;
      if (dato != -1) {
        if (contador < venteros.size())
          venteros[contador] = dato;   // añade dato al final
        else {
         if (venteros.size() > = venteros.capacity())
             // Se duplica el vector
            venteros.reserve(2*venteros.capacity());
         venteros.push_back(dato); // añade dato al final
        }
        contador++;
      }
    } while (dato != -1);
    cout << "Vector: leido\n ";

    //se visualiza el vector con un iterador constante de la clase vector
    for (vector<int>::const_iterator cont = venteros.begin();
          cont != venteros.end(); cont++)
       cout << *cont << " ";
    return 0;
}
```

15.6. LISTA

La clase **`list`** (lista) es una lista doblemente enlazada que proporciona una estructura de datos eficiente para insertar y quitar elementos de cualquier parte de la lista. No existen punteros `"anterior"` y `"siguiente"` para acceder a la estructura de la lista. La navegación a través de una lista se realiza de modo similar al contenedor `vector`. Se proporcionan iteradores para

acceder a sus elementos, pero los iteradores de `list` son bidireccionales. Su principal ventaja es la eficiencia de las operaciones de inserción y borrado, independientemente de la posición de la lista en la que se realicen. Como aspectos negativos podemos destacar dos: no podemos acceder a sus elementos de forma aleatoria, y el costo de las operaciones básicas es sensiblemente mayor que con los otros tipos de contenedores.

15.6.1. Declaración de una lista

Algunos medios para construir contenedores lista:

```
#include <list>
...
list<int> Listaent1;
list<int> Listaent2(100);
list<double> Listaent3(50, 2.718281);
```

La primera sentencia crea una lista vacía que puede almacenar valores enteros (`int`). La segunda sentencia crea una lista con 100 valores `int` que no están inicializados. La tercera sentencia construye una lista de 50 valores `double` que se inicializan a `2.718281`.

15.6.2. Funciones miembro y operadores más comunes en las listas

Algunas de las funciones miembro más utilizadas son las siguientes:

Función	Prototipo	Descripción
size	`size_type size() const.`	Devuelve el número de elementos almacenados actualmente en la lista.
empty	`bool empty() const.`	Devuelve `true` si la lista está vacía y `false` en otro caso.
begin	`iterator begin().`	Devuelve un iterador que referencia el primer elemento de la lista.
end	`iterator end().`	Devuelve un iterador que referencia la posición siguiente al último elemento en la lista.
push_back	`void push_back(const T& e).`	Añade el elemento e al final de la lista.
push_front	`void push_front(const T& e).`	Añade el elemento e al principio de la lista.
front	`T& front(); const T& front() const.`	Obtiene una referencia al primer elemento de la lista. Requiere que la lista esté no vacía.
back	`T& back(); const T& back() const.`	Obtiene una referencia al último elemento de la lista. Requiere que la lista esté no vacía.
pop_front	`void pop_front().`	Borra el primer elemento de la lista, requiere que la lista esté no vacía.
pop_back	`void pop_back().`	Borra el último elemento de la lista, requiere que la lista esté no vacía.

remove	`void remove (const T& e).`	Borra todos los elementos de la lista que sean iguales al valor e. El operador de igualdad (==) debe estar definido para el tipo T almacenado en la lista.
sort	`void sort().`	Ordena la lista de elementos en orden ascendente. El operador < debe estar definido para el tipo de elemento almacenado en la lista.
reverse	`void reverse().`	Invierte el orden de los elementos de la lista.
clear	`void clear().`	Borra todos los elementos de la lista.
insert	`iterator insert(iterator position, const T& e).`	Inserta un elemento e en la lista en la posición especificada por el iterador. El valor devuelto es un iterador que especifica la posición donde se ha insertado.
erase	`iterator erase(iterator position); iterator erase(iterator primero, iterator ultimo)`	Borra un elemento o todos los elementos de un rango de una lista. En el caso de un rango, esta operación borra elementos de la posición a la que apunta el primer iterador, hasta (pero sin incluirlo), la posición del segundo. Devuelve la posición del elemento siguiente al último borrado.

La Tabla 15.9 muestra las funciones miembro que actúan sobre listas.

Tabla 15.9 . Tabla Funciones miembro de un lista

`operador ==`	`operador !=`	`operador>`	`operador<`
`operador <=`	`operador >=`	`operador=`	`Operador*`
`operador ->`	`splice`	`begin`	`end`
`rbegin`	`rend`	`empty`	`size`
`max_size`	`front`	`back`	`push_front`
`push_back`	`pop_front`	`pop_back`	`swap`
`insert`	`erase`	`remove`	`Unique`
`merge`	`reverse`	`sort`	`clear`

EJEMPLO 15.4. Uso de alguna función miembro de la STL lista.

Se muestra cómo añadir elementos al final de una lista y dos formas distintas de visualizar elementos de la lista

```
#include <cstdlib>
#include <iostream>
#include <list>          // clase lista
using namespace std;
```

```
int main(int argc, char *argv[])
{
   list <int> l;                 // lista de enteros
   list<int>::const_iterator i; // iterador de la lista
   int valor;
                                 // Añadir valores al final de la lista
   l.push_back(2); l.push_back(4);
   l.push_back(5); l.push_back(1);
   l.push_back(20);
   for (int j = 10; j < 15; j++)
      l.insert(l.end(), j);     // añade j al final de la lista.

   cout << " datos almacenados en la lisa " << endl;
       // visualización de la lista
   for (i = l.begin(); i != l.end(); i++)
      cout << *i << " ";
   cout << endl;
       // nueva visualización de la lista
   cout << "Valores de la lista:" << endl;
   while (l.size() > 0)    // mientras haya elementos en la lista
   {
     valor = l.front();    // Obtener el valor del inicio de la lista
     cout << valor << " ";
     l.pop_front();        // Eliminar el elemento
   }
   return 0;
}
```

15.7. *DEQUE* (DOBLE COLA)

El contenedor **deque** *(double-ended queue,* doble cola) es parecido a la estructura típica doble cola. **deque** combina las características de un vector con una lista. En este sentido, una *deque* es una lista en la cual están permitidas las operaciones de indexado. Las *deques* ofrecen ventaja en algoritmos que requieren acceso en la parte frontal y trasera de listas. Una *deque* clásica funciona como una pila mezclada con una cola, en la que se pueden poner (`push_front`) y quitar (`pop_front`) elementos en cualquier extremo de una lista. El `deque` estándar permite también inserciones en cualquier parte del contenedor, aunque las inserciones centrales no son tan eficientes como en los extremos.

La memoria se asigna de modo diferente en los vectores y en las colas. Un vector ocupa siempre una región contigua de la memoria, de modo que si crece puede necesitar moverse a una nueva posición donde se pueda acomodar. Una cola, por el contrario, se puede almacenar en áreas de memoria no contiguas, es decir, se puede segmentar. La función miembro `capacity()` devuelve el número máximo de elementos `deque` que puede almacenar sin necesidad de movimiento; mientras que esta función no tiene aplicación en las dobles colas, por la simple razón de que éstas no necesitan desplazarse.

15.7.1. Declaración de *deque*

Algunas declaraciones usuales de dobles colas:

```
#include <deque>
#include <string>
```

```
...
deque <string> d1;
deque <double> d2 (50)
deque <int> d3 (15, -5)
```

La primera sentencia crea una *deque* vacía de objetos cadena. La segunda crea una lista de 50 elementos de tipo double. La tercera crea una *deque* con una quincena de valores enteros inicializados a –5.

15.7.2. Uso de *deque*

Las funciones `push_front` y `push_back` se utilizan para insertar elementos en *deques*. Por ejemplo, estas sentencias construyen una *deque* de cadena denominada `animales` e inserta cuatro objetos cadena:

```
deque <string> animales;
animales.push_front("perro");
animales.push_front("gato");
animales.push_front("pato");
animales.push_front("elefante");
```

La lista de resultados es `perro, gato, pato` y `elefante.` Para visualizar estos valores y eliminarlos de la *deque* utilizando la función `pop_front:`

```
while (!animales.empty())
{
   cout << animales.front() << endl;
   animales.pop_front();
}
```

Se pueden cambiar `front` por `back` y `pop_front` por `pop_back` para visualizar y eliminar elementos en orden inverso. Las funciones de `deque` más usuales son: `assign`, `insert`, `erase`, `size` y `swap`, similares a las funciones correspondientes de las plantillas `vector` y `list`.

El contenedor es similar a un vector, excepto que soporta algoritmos eficientes para insertar y eliminar elementos en cualquier extremo de la cola. Esta propiedad puede ser útil como una estructura de datos **FIFO** en la que se pueden insertar en la cabeza y eliminar de la cola.

EJEMPLO 15.5. Muestra el uso de una deque . De modo usual, se accede a los elementos de una deque sólo en la cabeza y en la cola de la lista, pero el contenedor estándar per mite también el indexado, como se muestra a continuación.

```
#include <cstdlib>
#include <iostream>
#include <deque>
using namespace std;

int main(int argc, char *argv[])
{
    deque<int> deq;
```

```
    deq.push_back(5);       //añadir después del final
    deq.push_back(10);
    deq.push_back(17);
    deq.push_front(3);      //insertar al principio
    for (int i = 0; i < deq.size(); i++)
       cout << "deq ["<< i<<"]="<< deq[i] << endl;

    cout << endl;

    deq.pop_front();        //borrar primer elemento
    deq[2] = 25;            //reemplazar último elemento
    for (int i = 0; i < deq.size(); i++)
       cout << "deq ["<< i <<"] =" << deq [i] << endl;
    return 0;
}
```

Al ejecutar el programa se obtiene la salida:

```
deq[0] = 3
deq[1] = 5
deq[2] = 10
deq[3] = 17
deq[0] = 5
deq[1] = 10
deq[2] = 25
```

Tabla 15.10. Funciones miembro de una deque

operador ==	operador !=	operador >	operador <
operador <=	operador >=	operador =	operador[]
operador +=	operator -=	begin	end
rbegin	rend	empty	size
max_size	front	back	push_front
push_back	pop_front	pop_back	swap
insert	erase		

Este contenedor está especializado en la inserción y eliminación de elementos en el frente y en el final y, por consiguiente, es ideal para crear colas **FIFO**. Un listado de las funciones miembro de las plantillas es:

```
void push_front(const T& x);
void push_back(const T& x);
void pop_front();
void pop_back();
iterator insert(iterator posicion, const T&, x);
void erase(iterator posicion);
void erase(iterator primero, iterator ultimo);
```

15.8. CONTENEDORES ASOCIATIVOS SET Y MULTISET

Un conjunto (*set*) es una colección de valores únicos. Un multiconjunto (*multiset*) es una colección de, posiblemente, valores no únicos. La principal característica de los contenedores asociativos es su capacidad para almacenar objetos basados en un valor clave y la recuperación eficientes de objetos vía las claves. La especificación de STL incluye cuatro contenedores asociativos:

- `set<clave, comparar>`: colección ordenada de claves. Este no permite claves duplicadas y es un candidato ideal para un grupo de registros ordenados único.
- `map<clave, T, comparar>`: colección de objetos que pueden ser referenciados por un valor clave. Este es un candidato ideal para una búsqueda de elementos no secuenciales.
- `multiset<clave, comparar>`: igual que `set<clave, comparar>`, pero permite claves duplicadas.
- `multimap <clave, T, comparar>`: igual que `map<clave, T, comparar>`, pero permite valores duplicados.

La diferencia entre un conjunto y un mapa, es que un conjunto es una colección ordenada de claves, mientras que un mapa es una colección de objetos que se pueden referenciar por sus claves asociadas. Un mapa almacena el objeto asociado con el valor clave; un conjunto sólo almacena la clave. Los conjuntos y multiconjuntos se comportan como bolsas que pueden contener y recuperar rápidamente cualquier tipo de valores. Los programas pueden determinar rápidamente si un valor está en un conjunto o cuántos de esos valores están en un multiconjunto.

Sin embargo, los conjuntos y multiconjuntos no son buenos para operaciones lineales o de acceso aleatorio. No se puede ordenar un conjunto o multiconjunto, y existen pocas funciones miembro definidas para estas plantillas.

15.8.1. Declaraciones de conjunto

Los conjuntos son más complejos de declarar que otros contenedores. Además, para especificar el tipo de datos a almacenar en un conjunto, debe definir también un objeto de función para mantener valores de conjunto en orden. Por esta razón, es mejor normalmente utilizar una sentencia `typedef` que define los parámetros del conjunto. Por ejemplo, esta sentencia:

```
typedef set<int, 1ess<int>> Tset;
```

define un tipo de dato conjunto denominado `Tset` que puede almacenar valores `int` y que utiliza el objeto función `1ess<int>` para mantener esos valores en orden.

La declaración normal de un conjunto se hace de la siguiente manera:

```
set <tipo> nombre;
```

15.8.2. Uso de conjuntos

La manipulación del conjunto es mediante **iteradores**, la forma de declarar un iterador es la siguiente:

```
set <tipo>::iterator nombre;
```

Los métodos más comunes que se pueden usar sobre los conjuntos son los siguientes:

Función	Prototipo	Descripción
empty	`bool empty() const`	Devuelve `true` si el número de elementos es cero y `false` en caso contrario.
begin	`iterator begin()`	Devuelve un iterador que referencia el comienzo del conjunto.
end	`iterator end()`	Devuelve un iterador que referencia la posición siguiente al final del conjunto.
erase	`void erase(tipo x)`	Borra el elemento `x` del conjunto.
erase	`void erase(iterator p)`	Borra el elemento que ocupa posición `p` del conjunto.
insert	`void insert(tipo x)`	Añade `x` al conjunto.
clear	`void clear()`	Borra todos los elementos del conjunto.
find	`iterator find(tipo x)`	Busca `x` en el conjunto, si lo encuentra retorna un iterador que apunta a él, en caso contrario retorna un iterador que apunta al final del conjunto.

La biblioteca `algorithm` proporciona algunas funciones para trabajar con conjuntos. Algunas de ellas requieren usar `insert` que se encuentra en `iterator`:

`includes` Comprueba si un conjunto está contenido en otro. `includes(C1. begin(), C1.end(), C2.begin(),C2.end())` Decide si el conjunto `C1` está contenido en `C2`.
`set_union` Une dos conjuntos añadiendo los elementos de la unión en un tercero. `set_union(C1.begin(),C1.end(), C2.begin(),C2.end(),inserter(C3,C3.begin())).` Une el conjunto `C1` con `C2` añadiendo el resultado a `C3`.
`set_intersection` Contruye la intersección de dos conjuntos en un tercero. `set_intersection(C1.begin(),C1.end(),C2.begin(),C2.end(),inserter(C3,C3.begin())).` Añade a `C3` la intersección del conjunto `C1` con `C2`.

EJEMPLO 15.6. Muestra el resultado de la unión e intersección de dos conjuntos de enteros previamente inicializados.

```
#include <cstdlib>
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

int main(int argc, char *argv[])
```

```
{
  int n, i, j;
  set <int> c1, c2, c3;  // se crean los conjuntos vacíos c1, c2 y c3
  set <int>::iterator iterador;
  for (i = 1; i <= 4; i++)  // se añaden datos al conjunto c1
     c1.insert(i);
  for (i = 3; i <= 7; i++)  // se añaden datos al conjunto C2
     c2.insert(i);
    // la union del conjunto c1 y c2 se almacena en el conjunto c3
  set_union(c1.begin(), c1.end(), c2.begin(), c2.end(),
            inserter(c3,c3.begin()));
    // Se visualiza el resultado
 for (iterador = c3.begin(); iterador != c3.end(); iterador++)
     cout << *(iterador) << " ";
 cout << endl;
 c3.clear();      // se limpia el conjunto c3
 set_intersection(c1.begin(), c1.end(), c2.begin(), c2.end(),
                  inserter(c3, c3.begin()));
   // se visualza la intersección
 for (iterador = c3.begin(); iterador != c3.end(); iterador++)
   cout << *(iterador) << " ";
 cout << endl;
 return 0;
}
```

La ejecución del programa anterior muestra en pantalla:

```
1 2 3 4 5 6 7
3 4
```

que son los resultados de la unión e intersección de los conjuntos `C1` y `C2`.

15.8.3. Uso de mapas y multimapas

Los contenedores de mapas se pueden visualizar como arrays que se indexan por claves por algún tipo arbitrario clave en lugar de con enteros 0, 1, 2 ... al estilo tradicional. Son contenedores asociativos ordenados que proporcionan una recuperación rápida de información de algún tipo `T` basado en claves de un tipo independiente, con las claves almacenadas de modo único. Los multimapas hacen lo mismo, pero permiten claves duplicadas; sus relaciones con los mapas son similares a las existentes entre multiconjuntos y conjuntos.

Al igual que conjuntos y multiconjuntos, los mapas y multimapas tienen iteradores bidireccionales. La inserción en mapas y multimapas se puede realizar con funciones miembro `insert` que tienen los mismos interfaces que conjuntos y multiconjuntos. Los constructores tienen también el mismo formato que los conjuntos, y también consiguen que la construcción de un mapa de un rango elimina copias con claves duplicadas y un multimapa las mantiene.

EJEMPLO 15.7. El siguiente programa utiliza un multimapa para asociar dos valores diferentes con la letra `'x'`. La función `find()` se utiliza para localizar el primer par cuya clave sea igual a `'x'` y, a continuación, se visualizan todas las parejas desde este punto hasta que se alcanza el final.

```
#include <cstdlib>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <map>
using namespace std;

int main(int argc, char *argv[])
{
  typedef multimap <char, int, less<char> > mmap;
  mmap m;
  m.insert (mmap::value_type('x',10));
  cout << "contar('x')=" << m.count ('x') << endl;
  m.insert (mmap::value_type('x',20));
  cout << "contar('x')=" << m.count('x') << endl;
  m.insert (mmap::value_type('z',50));
  mmap::iterator i = m.find('x');
  while (i != m.end ())
  {
    cout <<(*i).first << "=" << (*i).second << endl;
    i++;
  }
  int count = m.erase('x');
  cout << "Eliminados" << count << "elementos" << endl;
  return 0;
}
```

Al ejecutar el programa se obtiene

```
contar ('x')=0
contar ('x')=1
contar ('x')=2
x=1
x=2
z=50
Eliminados 2 elementos
```

15.9. CONTENEDORES ADAPTADORES

Los adaptadores son componentes STL que se pueden utilizar para cambiar la interfaz de otro componente. Se definen como clases plantilla que tienen un tipo componente como parámetro. STL proporciona adaptadores para producir contenedores `stack` (*pila*), `queue` (*cola*) y `priority_queue` (*cola_de_prioridad*).

15.9.1. `stack` (pila)

Un `stack` (pila) es un adaptador que permite utilizar cualquier contenido que soporta funciones tales como meter y quitar (`push()` y `pop()`, implementadas como `push_back()` y `pop_back( )` en estructuras tipo pila (primero en entrar, último en salir).

Como los adaptadores no soportan iteración, una pila no tiene iteradores asociados. Una pila se implementa como una clase con funciones *inline* que convierte los mensajes `push()` y `pop()` en mensajes `push_back()` y `pop_back()`. El contenedor adaptador `stack` se puede aplicar a un `vector, list` (*lista*) o `deque` (*doble cola*):

- `stack <vector<T>>` es una pila de tipo `T` con una implementación de vector.
- `stack<list<T>>` es una pila de tipo `T` con una implementación de lista.
- `stack<deque<T>>` es una pila de tipo `T` con una implementación de doble cola.

Se puede crear una pila con una declaración de la forma:

```
stack<T, C<T> > p
```

Donde C puede ser cualquier contenedor que soporte las operaciones `empty` (*vacía*), `size` (*tamaño*), `push_back` (*meter*), `pop_back` (*sacar*). Si la declaración es de la forma `stack <T> p` por defecto el contenedor es una `deque`.

EJEMPLO 15.8. La pila siguiente usa una doble cola de cadenas de caracteres como estructura de datos fundamental

```
#include <cstdlib>
#include <iostream>
#include <deque>
#include <stack>
#include <string>
using namespace std;

int main(int argc, char *argv[])
{
  stack<string, deque<string> >p;
  p.push (" pila ");
  p.push (" una ");
  p.push (" de ");
  p.push (" prueba ");
  p.push (" una ");
  p.push (" es ");
  p.push (" Esto ");
  while (!p.empty())
  {
     cout << p.top() ;
     p.pop();
  }
  return 0;
}
```

El resultado de ejecutar el programa anterior es:

```
Esto es una prueba de una pila
```

15.9.2. Queue (Cola)

Una cola es un adaptador que permite utilizar cualquier contenedor que soporte operaciones `push_back()` y `pop_front()` como una estructura de datos tipo cola (primero en entrar, primero en salir). Dado que los adaptadores no soportan iteración, una cola no tiene un iterador asociado. Las operaciones de la cola están soportadas por `list` (*lista*) y `deque` (*doble cola*) y pueden construir colas a partir de estos contenedores:

- `queue< T, list<T> >` es una cola de tipo `T` con una implementación de `list`.
- `queue<T, deque<T> >` es una cola de tipo `T` con una implementación `deque`.

Las operaciones proporcionadas por `queue` son:

`empty()`, decide si la cola está vacía.
`size()`, retorna el número de elementos de la cola.
`front()`, recupera el primer elemento de la cola.
`push()`, implementa la operación "añadir un elemento a la cola".
`pop()`, implementa la operación "eliminar el primer elemento del frente de la cola".

EJEMPLO 15.9. El siguiente programa muestra una cola implementada con una lista.

```
#include <cstdlib>
#include <iostream>
#include <list>
#include <queue>
#include <string>
using namespace std;

int main(int argc, char *argv[])
{
  queue<int,list<int> > q;
  q.push (51); q.push (125); q.push (81) ;
  while (!q.empty())
  {
    cout << q.front() << endl;
    q.pop();
  }
  return 0;
}
```

La ejecución del programa produce la siguiente salida

```
51
125
81
```

15.9.3. priority_queue: Cola de prioridad

Una cola de prioridad es un adaptador que permite implementar un tipo de lista en la que el elemento inmediatamente disponible para recuperación es el mayor de los que están en secuen-

cia en un orden preestablecido. La cola de prioridad no introduce ninguna función nueva, pero su implementación de `pop()` actúa de modo diferente a la pila y a la cola. STL proporciona una cola de prioridad, implementada usando un *Max Heap* (montón max) esta función usa el operador < para determinar la prioridad de los elementos. Para usarla se debe incluir la biblioteca `queue`.

EJEMPLO 15.10. Una cola de prioridad sirve para ordenar un conjunto de números enteros decrecientemente.

```
#include <cstdlib>
#include <iostream>
#include <queue>
using namespace std;
int main(int argc, char *argv[])
{
  priority_queue<int> q;
  q.push (2); q.push (1); q.push (3); q.push(12); q.push(4);
  while (!q.empty())
  {
    cout << q.top() << endl;
    q.pop();
  }
  return 0;
}
```

La salida del programa anterior es:

```
12
4
3
2
1
```

RESUMEN

La biblioteca de plantillas estándar (**STL**) es la biblioteca estándar de C++ que proporciona programación genérica para muchas estructuras de datos y algoritmos. Consta de plantillas de clases contenedoras, iteradores, funciones y adaptadores para acceder y procesar los elementos de los contenedores.

La componente principal de STL es la familia de clases contenedoras, que son objetos genéricos que almacenan otros objetos.

Un iterador es un puntero inteligente que apunta a objetos de un contenedor. Los iteradores se utilizan como marcadores de índice para proporcionar la posición de un elemento en una estructura de datos; una generalización de los punteros.

La clase vector es un contenedor de las STL basado en vectores con una gran variedad de operadores y cuya capacidad puede incrementarse a lo largo de la ejecución de un programa.

La clase list es un contenedor de las STL que proporciona una estructura de datos para insertar y eliminar elementos en cualquier posición, implementado con una lista doblemente enlazada.

La clase `deque` implementa una doble cola que combina las propiedades de una lista y de un vector, permitiendo la inserción y borrado así como la indexación de elementos. Las deques ofrecen ventajas para aquellos algoritmos que requieran inserción y borrado de elementos por ambos extremos de la estructura de datos.

Los contenedores `stack` y `queue` son adaptadores que envuelven otros contenedores.

BIBLIOGRAFÍA RECOMENDADA

Aho V.; Hopcroft, J., y Ullman, J.: *Estructuras de datos y algoritmos*. Addison Wesley, 1983.

Garrido, A., y Fernández, J.: *Abstracción y estructuras de datos en C++*. Delta, 2006.

Joyanes, L., y Zahonero, L.: *Algoritmos y estructuras de datos. Una perspectiva en C*. McGraw-Hill, 2004.

Joyanes, L.; Sánchez, L.; Zahonero, I., y Fernández, M.: *Estructuras de datos en C*. Schaum, 2005.

Weis, Mark Allen: *Estructuras de datos y algoritmos*. Addison Wesley, 1992.

Wirth Niklaus: *Algoritmos + Estructuras de datos = programas*, 1986.

EJERCICIOS

15.1. Usar el contenedor `stack` para leer una frase del teclado que represente una expresión y decida si está bien parentizada (tiene igual número de paréntesis abiertos que cerrados).

15.2. Usar los contenedores `stack` y `queue` para decidir si una frase leída del teclado es palíndroma. Una frase es palíndroma si se lee de igual forma en ambos sentidos, después de haber eliminado los blancos. Por ejemplo: dábale arroz a la zorra el abad.

15.3. Usar las STL contenedores `vector`, y `stack` para obtener una secuencia de 10 números reales, guardarlos en un array y ponerlos en una pila. Imprimir la secuencia original y, a continuación, imprimir la pila extrayendo los elementos.

15.4. Usar las STL `list` y `stack` para leer un total de 20 elementos almacenarlos en el orden en que se leen en una lista y visualizar los nodos de la lista en orden inverso, desde el último nodo al primero; como estructura auxiliar utilizar una pila y sus operaciones.

15.5. Se tiene una pila de enteros positivos. Con las operaciones básicas de pilas y colas escribir un fragmento de código para poner todos los elementos de la pila que son par en la cola. Usar las STL `stack` y `queue`.

15.6. Escribir una función que devuelva el mayor entero de una lista doblemente enlazada de números enteros implentada con STL

CAPÍTULO 16

Árboles. Árboles binarios y árboles ordenados

Objetivos

Con el estudio de este capítulo usted podrá:

- Estructurar datos en orden jerárquico.
- Conocer la terminología básica relativa a árboles.
- Distinguir los diferentes tipos de árboles binarios.
- Recorrer un árbol binario de tres formas diferentes.
- Reconocer la naturaleza recursiva de las operaciones con árboles.
- Representar un árbol binario con una estructura enlazada.
- Evaluar una expresión algebraica utilizando un árbol binario.
- Construir un árbol binario ordenado (de búsqueda).

Contenido

Conceptos clave

- Árbol.
- Árbol binario.
- Árbol binario de búsqueda.
- *Enorden*.
- Hoja.
- Jerarquía.
- *Postorden*.
- *Preorden*.
- Raiz.
- Recorrido.
- Subárbol.

INTRODUCCIÓN

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras *no lineales*, al contrario que los arrays y las listas enlazadas que constituyen *estructuras lineales*. La estructura de datos árbol generaliza las estructuras lineales vistas en capítulos anteriores.

Los árboles se utilizan para representar fórmulas algebraicas, para organizar objetos en orden de tal forma que las búsquedas son muy eficientes, y en aplicaciones diversas tales como inteligencia artificial o algoritmos de cifrado. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. Además de las aplicaciones citadas, los árboles se utilizan en diseño de compiladores, proceso de texto y algoritmos de búsqueda.

En el capítulo se estudiará el concepto de árbol general y los tipos de árboles más usuales, binario y binario de búsqueda o árbol ordenado. Asimismo, se estudiarán algunas aplicaciones típicas del diseño y construcción de árboles.

16.1. ÁRBOLES GENERALES Y TERMINOLOGÍA

Intuitivamente el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas. El árbol genealógico es el ejemplo típico más representativo del concepto de árbol general. La Figura 16.1 representa un ejemplo de árbol general, gráficamente puede verse cómo un árbol invertido, la raíz en la parte más alta de la que salen ramas que llegan a las hojas, que están en la parte baja.

Un **árbol** consta de un conjunto finito de elementos, denominados **nodos** y un conjunto finito de líneas dirigidas, denominadas **ramas**, que conectan los nodos. El número de ramas asociado con un nodo es el **grado** del nodo.

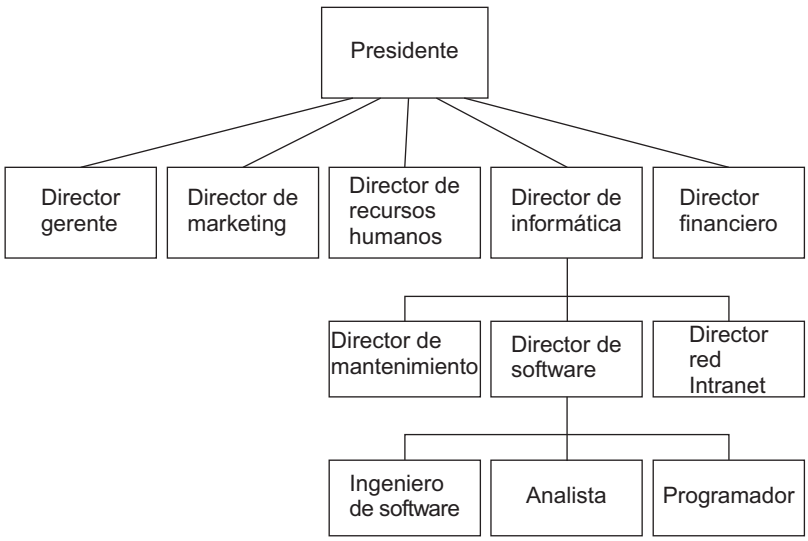


Figura 16.1. Estructura jerárquica tipo árbol.

Definición 1:

Un árbol consta de un conjunto finito de elementos, llamados nodos y un conjunto finito de líneas dirigidas, llamadas ramas, que conectan los nodos.

Definición 2:

Un árbol es:

1. La estructura de datos vacía.
2. O bien un conjunto de uno o más nodos tales que:
   2.1. Hay un nodo diseñado especialmente llamado raíz.
   2.2. Los nodos restantes se dividen en $n \geq 0$ conjuntos disjuntos, `T1...Tn`, tal que cada uno de estos conjuntos es un árbol. A `T1...Tn` se les denomina subárboles del raíz.

Si un árbol no está vacío, entonces el primer nodo se llama **raíz**. Obsérvese en la definición 2 que el árbol ha sido definido de modo recursivo ya que los subárboles se definen como árboles. La Figura 16.2 muestra un árbol.

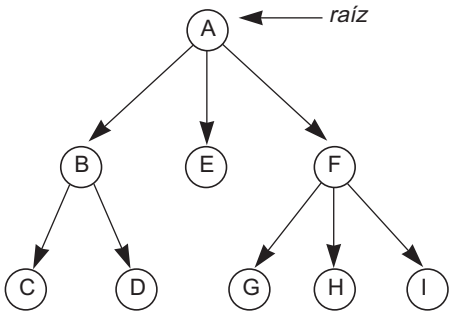


Figura 16.2. Árbol.

16.1.1. Terminología

Además del raíz, existen muchos términos utilizados en la descripción de los atributos de un árbol. En la Figura 16.3, el nodo `A` es el raíz. Utilizando el concepto de árboles genealógicos, un nodo puede ser considerado como **padre** si tiene nodos sucesores.

Estos nodos sucesores se llaman **hijos**. Por ejemplo, el nodo `B` es el padre de los hijos `E` y `F`. El padre de `H` es el nodo `D`. Un árbol puede representar diversas generaciones en la familia. Los hijos de un nodo y los hijos de estos hijos se llaman **descendientes** y el padre y abuelos de un nodo son sus **ascendientes**. Por ejemplo, los nodos `E`, `F`, `I` y `J` son descendientes de `B`. Cada nodo no raíz tiene un único padre y cada padre tiene cero o más nodos hijos. Dos o más nodos con el mismo padre se llaman **hermanos**. Un nodo sin hijos, tales como `E`, `I`, `J`, `G` y `H` se llaman nodo **hoja**.

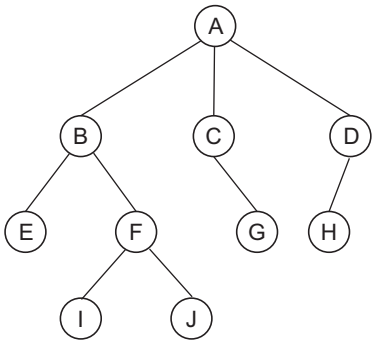


Figura 16.3. Árbol general.

El **nivel** de un nodo es su distancia al nodo raíz. El raíz tiene una distancia cero de sí misma, por ello se dice que el raíz está en el nivel 0. Los hijos del raíz están en el nivel 1, sus hijos están en el nivel 2 y así sucesivamente. Una cosa importante que se aprecia entre los niveles de nodos es la relación entre niveles y *hermanos*. Los hermanos están siempre al mismo nivel, pero no todos los nodos de un mismo nivel son necesariamente hermanos. Por ejemplo, en el nivel 2 (Figura 16.4), `C` y `D` son hermanos, al igual que lo son `G`, `H`, e `I`, pero `D` y `G` no son hermanos ya que ellos tienen diferentes padres.

Un **camino** es una secuencia de nodos en los que cada nodo es adyacente al siguiente. Cada nodo del árbol puede ser alcanzado (se llega a él) siguiendo un único camino que comienza en el raíz. En la Figura 16.4, el camino desde el raíz a la hoja `I`, se representa por `AFI`. Incluye dos ramas distintas `AF` y `FI`.

La **altura** o **profundidad** de un árbol es el nivel de la hoja del camino más largo desde la raíz más uno. Por definición[1] la altura de un árbol vacío es `0`. La Figura 16.4 contiene nodos en tres niveles: `0, 1 y 2`. Su altura es `3`.

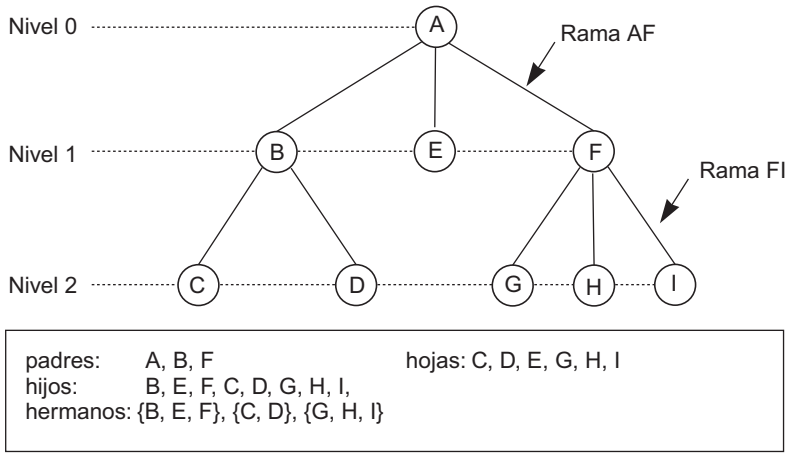


Figura 16.4. Terminología de árboles.

[1] También se suele definir la **profundidad** de un árbol como el nivel máximo de cada nodo. En consecuencia, la profundidad del nodo raíz es 0, la de su hijo 1, etc. Las dos terminologías son aceptadas.

Definición

El nivel de un nodo es su distancia desde el nodo raíz. La altura de un árbol es el nivel de la hoja del camino más largo desde el raíz más uno.

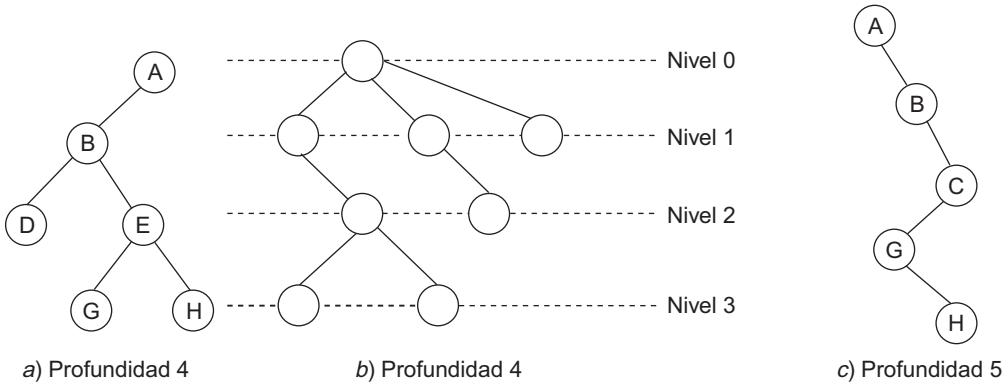


Figura 16.5. Árboles de profundidades diferentes.

Un árbol se divide en subárboles. Un **subárbol** es cualquier estructura conectada por debajo del raíz. Cada nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos los descendientes del nodo. El primer nodo de un subárbol se conoce como el raíz del subárbol y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subárboles. En la Figura 16.4, `BCD` es un subárbol al igual que `E` y `FGHI`. Obsérvese, que por esta definición, un nodo simple es un subárbol. Por consiguiente, el subárbol `B` se puede dividir en subárboles `C` y `D` mientras que el subárbol `F` contiene los subárboles `G`, `H` e `I`. Se dice que `G`, `H`, `I`, `C` y `D` son subárboles sin descendientes.

Definición recursiva

El concepto de subárbol conduce a una definición recursiva de un árbol. Un árbol es un conjunto de nodos que:

1. O bien es vacío, o bien,
2. Tiene un nodo determinado, llamado raíz, del que jerárquicamente descienden cero o más subárboles, que son también árboles.

Resumen de definiciones

- El primer nodo de un árbol, normalmente dibujado en la posición superior, se denomina **raíz** del árbol.
- Las flechas que conectan un nodo a otro se llaman **arcos** o **ramas**.

- Los **nodos terminales**, esto es, nodos de los cuales no se deduce ningún nodo, se denominan **hojas**.
- Los nodos que no son hojas se denominan **nodos internos**.
- En un árbol una rama va de un nodo n_1 a un nodo n_2, se dice que n_1 es el padre de n_2 y que n_2 es un **hijo** de n_1.
- n_1 se llama **ascendiente** de n_2 si n_1 es el padre de n_2 o si n_1 es el padre de un ascendiente de n_2.
- n_2 se llama **descendiente** de n_1 si n_1 es un ascendiente de n_2.
- Un **camino** de n_1 a n_2 es una secuencia de arcos contiguos que van de n_1 a n_2.
- La **longitud** de un camino es el número de arcos que contiene, o de forma equivalente, el número de nodos del camino menos uno.
- El **nivel** de un nodo es la longitud del camino que lo conecta al nodo raíz.
- La **profundidad** o **altura** de un árbol es la longitud del camino más largo que conecta el raíz a una hoja más uno.
- Un **subárbol** de un árbol es un subconjunto de nodos del árbol, conectados por ramas del propio árbol, esto es, a su vez un árbol.
- Sea S un subárbol de un árbol A: si para cada nodo *n* de SA, SA contiene también todos los descendientes de *n* en A. SA se llama un **subárbol completo** de A.
- Un árbol está **equilibrado** cuando, dado un número máximo *k* de hijos de cada nodo y la **altura del árbol** *h*, cada nodo de nivel $l < h - 2$ tiene exactamente *k* hijos. El árbol está equilibrado perfectamente entre cada nodo de nivel $l < h$ tiene exactamente *k* hijos.

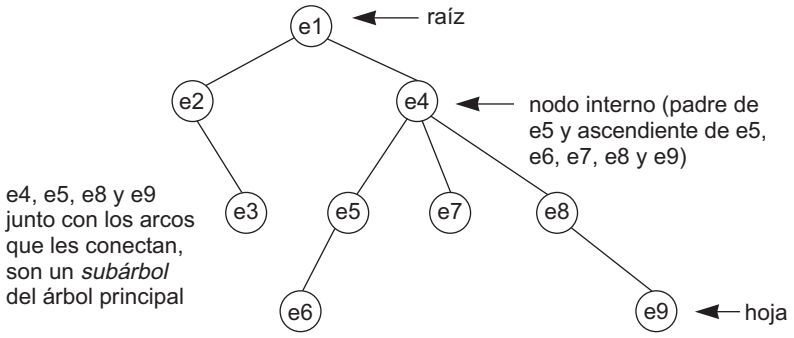


Figura 16.6. Un árbol y sus nodos.

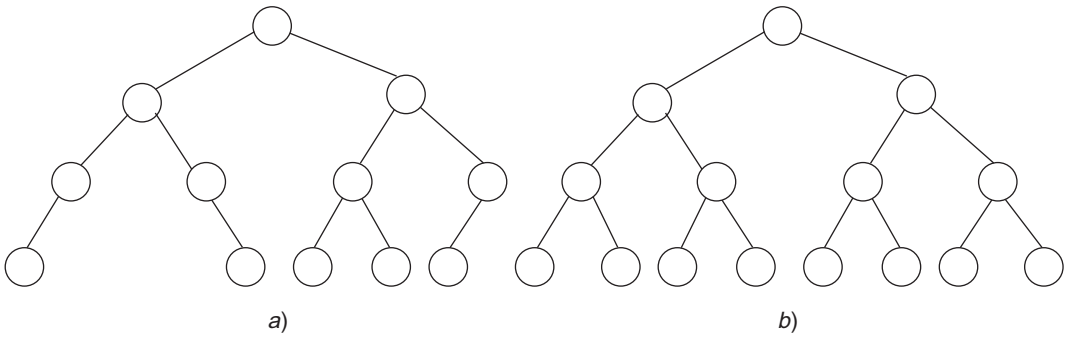


Figura 16.7. *a*) Un árbol equilibrado; *b*) Un árbol perfectamente equilibrado.

16.1.2. Representación gráfica de un árbol

Las formas más frecuentes de representar en papel un árbol son como árbol invertido y como una lista.

Representación como árbol invertido

Es el diagrama o carta de organización utilizada hasta ahora en las diferentes figuras. El nodo raíz se encuentra en la parte más alta de una jerarquía de la que descienden ramas que van a parar a los nodos hijos, y así sucesivamente. La Figura 16.8 presenta esta representación para una descomposición de una computadora.

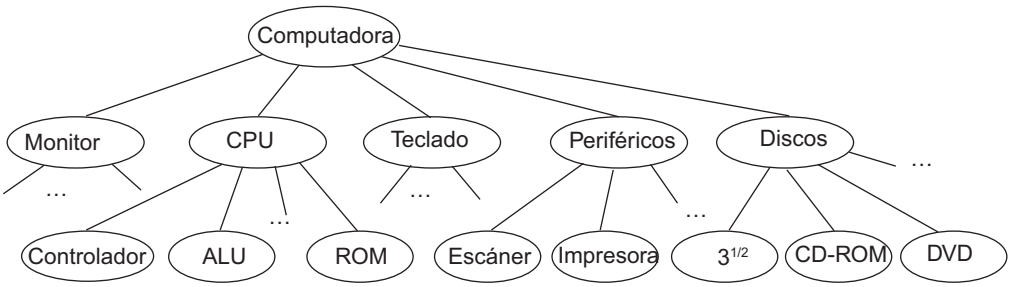


Figura 16.8. Árbol general (computadora).

Representación de lista

Otro formato utilizado para representar árboles es la lista entre paréntesis. Esta es la notación utilizada con expresiones algebraicas. En esta representación, cada paréntesis abierto indica el comienzo de un nuevo nivel; cada paréntesis cerrado completa un nivel y se mueve hacia arriba un nivel en el árbol. La notación en paréntesis correspondiente al árbol de la Figura 16.2: `A(B (C, D), E, F (G, H, I))`

EJEMPLO 16.1. Representar el árbol general de la Figura 16.9 en forma de lista.

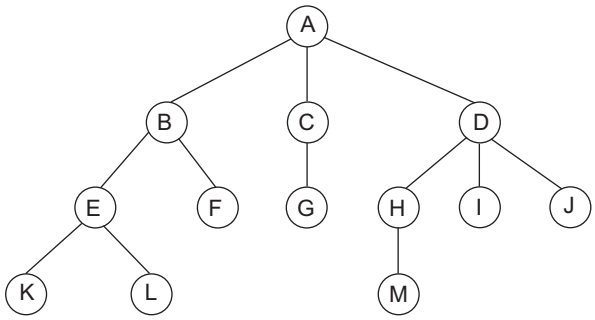
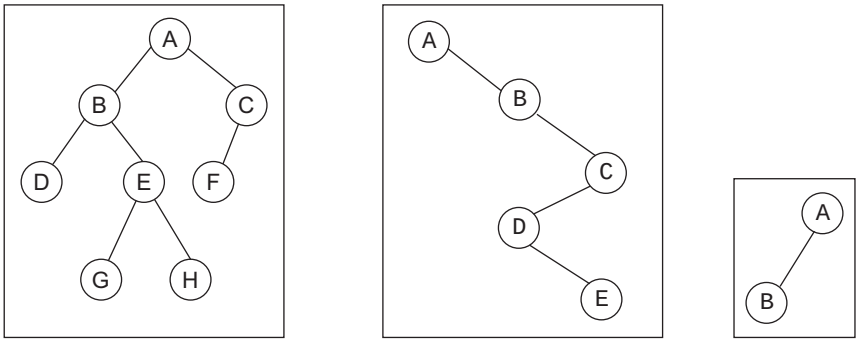


Figura 16.9. Árbol general.

La solución: `A ( B ( E ( K, L), F ), C ( G ), D ( H ( M ), I, J ) ) )`

16.2. ÁRBOLES BINARIOS

Un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos subárboles. En un árbol binario, cada nodo puede tener, cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como *hijo izquierdo* y el nodo de la derecha como *hijo derecho*.



a) *b*) *c*)

Figura 16.10. Árboles binarios.

Nota

Un árbol binario no puede tener más de dos subárboles. Un nodo no puede tener más de dos hijos.

Un árbol binario es una estructura recursiva. Cada nodo es el raíz de su propio subárbol y tiene hijos, que son raíces de árboles llamados los subárboles derecho e izquierdo del nodo, respectivamente. Un árbol binario se divide en tres subconjuntos disjuntos:

`{R}`	Nodo raíz
`{I1, I2, ...In}`	Subárbol izquierdo de `R`
`{D1, D2, ...Dn}`	Subárbol derecho de `R`

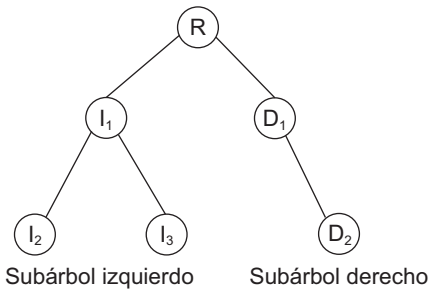


Figura 16.11. Árbol binario.

En cualquier nivel *n*, un árbol binario puede contener de `1` a 2^n nodos. El número de nodos por nivel contribuye a la densidad del árbol.

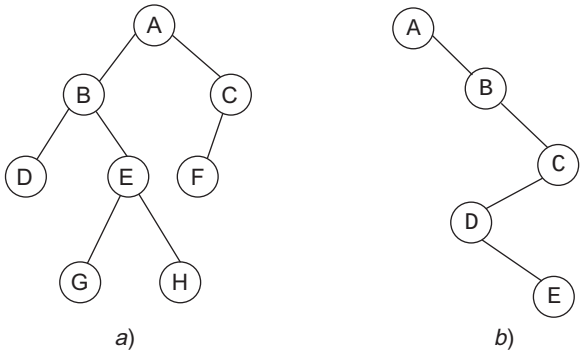


Figura 16.12. Árboles binarios: *a*) profundidad 4; *b*) profundidad 5.

En la Figura 16.12*a*) el árbol `A` contiene `8` nodos en una profundidad de `4`, mientras que el árbol 16.12*b*) contiene `5` nodos y una profundidad `5`.

16.2.1. Equilibrio

La distancia de un nodo al raíz determina la eficiencia con la que puede ser localizado. Por ejemplo, dado cualquier nodo de un árbol, a sus hijos se puede acceder siguiendo sólo un camino de bifurcación o de ramas, el que conduce al nodo deseado. De modo similar, los nodos a nivel 2 de un árbol sólo pueden ser accedidos siguiendo sólo dos ramas del árbol.

La característica anterior nos conduce a una característica muy importante de un árbol binario, su **balance** o **equilibrio**. Para determinar si un árbol está equilibrado, se calcula su factor de equilibrio. El **factor de equilibrio** de un árbol binario es la diferencia en altura entre los subárboles derecho e izquierdo. Si la altura del subárbol izquierdo es h_I y la altura del subárbol derecho como h_D, entonces el factor de equilibrio del árbol `B` se determina por la siguiente fórmula: $B = h_D - h_I$

Utilizando esta fórmula, el equilibrio del nodo raíz los árboles de la Figura 16.12 son *a* `1` y *b* `4`.

Un árbol está **perfectamente equilibrado** si su equilibrio o balance es *cero* y sus subárboles son también perfectamente equilibrados. Dado que esta definición ocurre raramente se aplica una definición alternativa. Un árbol binario está equilibrado si la altura de sus subárboles difiere en no más de uno y sus subárboles son también equilibrados; por consiguiente, el factor de equilibrio de cada nodo puede tomar los valores: `-1, 0, +1.`

16.2.2. Árboles binarios completos

Un árbol binario **completo** de profundidad *n* (0 a *n* – 1 niveles) es un árbol en el que para cada nivel, del *0* al nivel *n* – 2, tiene un conjunto lleno de nodos y todos los nodos hoja a nivel *n* – 1 ocupan las posiciones más a la izquierda del árbol.

Un árbol binario completo que contiene 2^n nodos a nivel *n* es un **árbol lleno**. Un árbol lleno es un árbol binario que tiene el máximo número de entradas para su altura. Esto sucede cuando el último nivel está lleno. La Figura 16.13 muestra un árbol binario completo; el árbol de la Figura 16.14*b*) se corresponde con uno lleno.

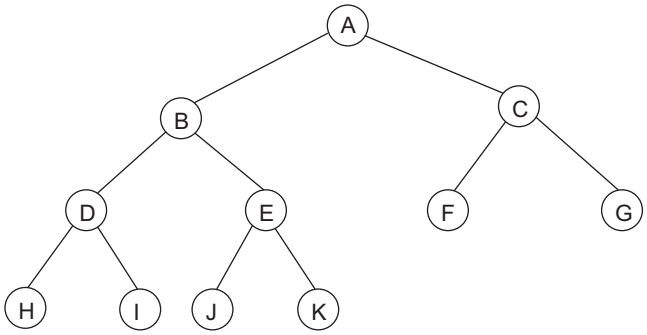


Figura 16.13. Árbol completo (profundidad 4).

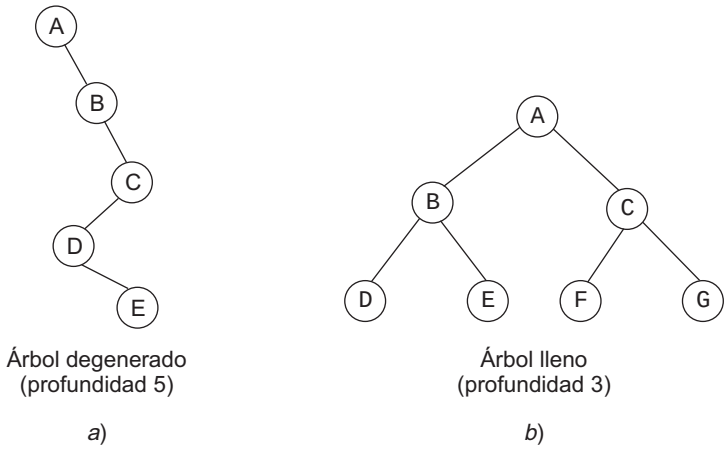


Figura 16.14. Clasificación de árboles binarios: *a*) degenerado; *b*) lleno.

El último caso de árbol es un tipo especial denominado **árbol degenerado** en el que hay un solo nodo hoja (`E`) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

Los árboles binarios completos y llenos de profundidad `k+1` proporcionan algunos datos matemáticos de interés. En cada caso, existe un nodo (2^0) al nivel `0` (raíz), dos nodos (2^1) a nivel `1`, cuatro nodos (2^2) a nivel `2`, etc. A través de los primeros *k-1* niveles se puede demostrar, considerando la suma de los términos de una progresión geométrica de razón `2`, que hay 2^k - `1` nodos.

$$1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$$

A nivel *k*, el número de nodos adicionados para un árbol completo está en el rango de un mínimo de 1 a un máximo de 2^k (lleno). Con un árbol lleno, el número de nodos es:

$$1 + 2 + 4 + \ldots + 2^{k-1} + 2^k = 2^{k+1} - 1$$

El número de nodos, n, en un árbol binario completo de profundidad k+1 (0 a k niveles) cumple la inigualdad:

$$2^k \leq n \leq 2^{k+1} - 1 < 2^{k+1}$$

Aplicando logaritmos a la ecuación con desigualdad anterior

$$k \leq \log_2 (n) < k + 1$$

se deduce que la altura o profundidad de un árbol binario completo de n nodos es:

$$h = |\text{Log}_2 n| + 1$$ (*parte entera* de $\text{Log}_2 n$ *más* 1)

Por ejemplo, un árbol lleno de profundidad 4 (*niveles 0 a 3*) tiene $2^4 - 1 = 15$ nodos.

EJEMPLO 16.2. Calcular la profundidad máxima y mínima de un árbol con 5 nodos .

La profundidad máxima de un árbol con 5 nodos es 5, se corresponde con un árbol degenerado.

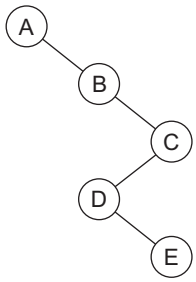


Figura 16.15. Árbol degenerado de raíz A.

La profundidad mínima *h* (número de niveles más uno) de un árbol con 5 nodos, aplicando la inecuación del número de nodos de un árbol binario completo:

$$k \leq log_2 (5) < k + 1$$

como $\log_2(5) = 2.32$, la profundidad h = 3.

EJEMPLO 16.3. Suponiendo que se tiene n = 10.000 elementos que van a ser los nodos de un árbol binario completo. Determinar la profundidad del árbol.

En el árbol binario completo con n nodos, la profundidad del árbol es el valor entero de $\log_2 n + 1$, que es a su vez, la distancia del camino más largo desde el raíz a un nodo más uno.

$$\text{Profundidad} = \text{int} (\log_2 10000) + 1 = \text{int} (13.28) + 1 = 14$$

16.2.3. *TAD* Árbol binario

La estructura de árbol binario constituye un *tipo abstracto de datos*; las operaciones básicas que definen el *TAD árbol binario* son las siguientes.

Tipo de dato	Dato que se almacena en los nodos del árbol.
Operaciones	
CrearÁrbol	Inicia el árbol como vacío.
Construir	Crea un árbol con un elemento raíz y dos ramas, izquierda y derecha que son a su vez árboles.
EsVacio	Comprueba si el árbol no tiene nodos.
Raiz	Devuelve el nodo raíz.
Izquierdo	Obtiene la rama o subárbol izquierdo de un árbol dado.
Derecho	Obtiene la rama o subárbol derecho de un árbol dado.
Borrar	Elimina del árbol el nodo con un elemento determinado.
Pertenece	Determina si un elemento se encuentra en el árbol.

16.2.4. Operaciones en árboles binarios

Algunas de las operaciones típicas que se realizan en árboles binarios son éstas:

- Determinar su altura.
- Determinar su número de elementos.
- Hacer una copia.
- Visualizar el árbol binario en pantalla o en impresora.
- Determinar si dos árboles binarios son idénticos.
- Borrar (eliminar el árbol).
- Si es un árbol de expresión, evaluar la expresión.

Todas estas operaciones se pueden realizar recorriendo el árbol binario de un modo sistemático. El recorrido es la operación de visita al árbol, o lo que es lo mismo, la visita a cada nodo del árbol una vez y sólo una. La visita de un árbol es necesaria en muchas ocasiones, por ejemplo, si se desea imprimir la información contenida en cada nodo. Existen diferentes formas de visitar o recorrer un árbol que se estudiarán más adelante.

16.3. ESTRUCTURA DE UN ÁRBOL BINARIO

Un árbol binario se construye con nodos. Cada nodo debe contener el campo *dato* (datos a almacenar) y dos campos de enlace (*apuntador*), uno al subárbol izquierdo (**izquierdo**, **izdo**) y otro al subárbol derecho (**derecho**, **dcho**). El valor `NULL` indica un árbol o un subárbol vacío.

La Figura 16.16 muestra la representación enlazada de dos árboles binarios de raíz `A`. El primero, es un árbol degenerado a la izquierda; el segundo, es un árbol binario completo de profundidad `4`.

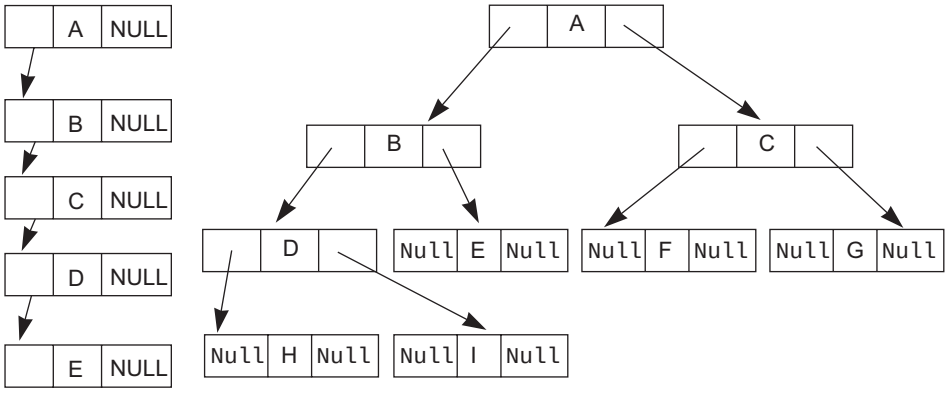


Figura 16.16. Representación enlazada de dos árboles binarios.

Se puede observar que los nodos de un árbol binario que son hojas se caracterizan por tener sus dos campos de enlace a `NULL`.

16.3.1. Representación de un nodo

La clase `Nodo` agrupa a todos los atributos de que consta: `dato`, `izdo` (rama izquierda) y `dcho` (rama derecha). Además, dispone de dos costructores, el primero inicializa el campo `dato` a un valor y los enlaces a `NULL`, en definitiva, se inicializa como hoja. El segundo, inicializa `dato` a un valor y las ramas a dos subárboles. Se incluyen, además, las funciones miembro de acceso y modificación de cada uno de sus atributos.

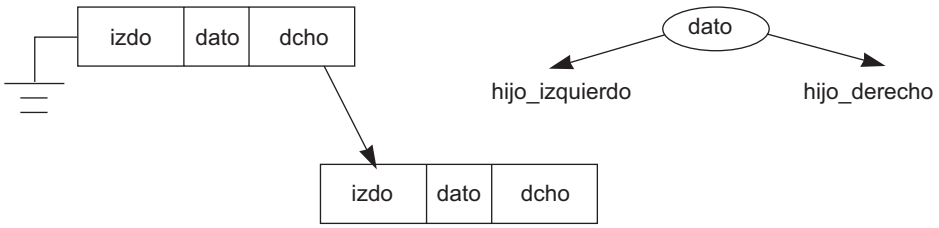


Figura 16.17. Representación gráfica de los campos de un nodo.

```
typedef int Tipoelemento;

class Nodo
{
protected:
    Tipoelemento dato;
    Nodo *izdo;
    Nodo *dcho;
```

```
public:

  Nodo(Tipoelemento valor)
  {
     dato = valor;
     izdo = dcho = NULL;
  }

  Nodo(Tipoelemento valor, Nodo* ramaIzdo, Nodo* ramaDcho)
  {
     dato = valor;
     izdo = ramaIzdo;
     dcho = ramaDcho;
  }
  // operaciones de acceso
  Tipoelemento valorNodo(){ return dato; }
  Nodo* subárbolIzdo(){ return izdo; }
  Nodo* subárbolDcho(){ return dcho; }

  // operaciones de modificación
  void nuevoValor(Tipoelemento d){ dato = d; }
  void ramaIzdo(Nodo* n){ izdo = n; }
  void ramaDcho(Nodo* n){ dcho = n; }
};
```

16.3.2. Creación de un árbol binario

A partir del nodo raíz de un árbol se puede acceder a los demás nodos del árbol, por ello se mantiene la referencia al raíz del árbol. La rama izquierda y derecha son a su vez árboles binarios que tienen su raíz, y así recursivamente hasta llegar a las hojas del árbol. La clase `ArbolBinario` tiene el atributo `raiz`, como un puntero a la clase `Nodo` dos constructores que inicializan `raiz` a `NULL` o a un puntero a un `Nodo` respectivamente. Se añaden, además, los métodos `esVacio()` que decide si un árbol binario está vacío `raizArbol(), hijoIzdo(), hijoDcho()` para implementar las operaciones de obtener el nodo raíz el hijo izquierdo y derecho respectivamente en caso de que lo tenga. El método `nuevoArbol()`crea un puntero a un `Nodo` con el atributo dato, rama izquierda y derecha pasadas en los argumentos. Se implementan, además, las funciones miembro `Praiz( Nodo *r)`y `Oraiz()` encargadas de poner el atributo raíz al puntero a `Nodo r` y obtener un puntero al nodo raíz.

```
class Arbolbinario
{
protected:
       Nodo *raiz;

public:

   ArbolBinario()
   {
     raiz = NULL;
   }
```

```
    ArbolBinario(Nodo *r)
    {
      raiz = r;
    }

    void Praiz( Nodo *r)
    {
      raiz = r;
    }

    Nodo * Oraiz()
    {
      return raiz;
    }

    Nodo raizArbol()
    {
      if(raiz)
         return *raiz;
      else
         throw " arbol vacio";
    }

    bool esVacio()
    {
      return raiz == NULL;
    }

    Nodo * hijoIzdo()
    {
      if(raiz)
         return raiz->subArbolIzdo();
      else
         throw " arbol vacio";
    }

    Nodo * hijoDcho()
    {
      if(raiz)
         return raiz->subArbolDcho();
      else
         throw " arbol vacio";
    }

    Nodo* nuevoArbol(Nodo* ramaIzqda, Tipoelemento dato, Nodo* ramaDrcha)
    {
        return new Nodo(ramaIzqda, dato, ramaDrcha);
    }
};
```

Así, para crear el árbol binario de la Figura 16.18, se puede utilizar el siguente segmento de código:

```
ArbolBinario a1,a2,a3, a4,a;
Nodo * n1,*n2,*n3, *n4;
```

```
n1 = a1.nuevoArbol(NULL,"Maria",NULL);
n2 = a2.nuevoArbol(NULL,"Rodrigo",NULL);
n3 = a3.nuevoArbol(n1,"Esperanza",n2);

n1 = a1.nuevoArbol(NULL,"Anyora",NULL);
n2 = a2.nuevoArbol(NULL,"Abel",NULL);
n4 = a4.nuevoArbol(n1,"M Jesus",n2);
n1 = a1.nuevoArbol(n3,"Esperanza",n4);
a.Praiz(n1);
```

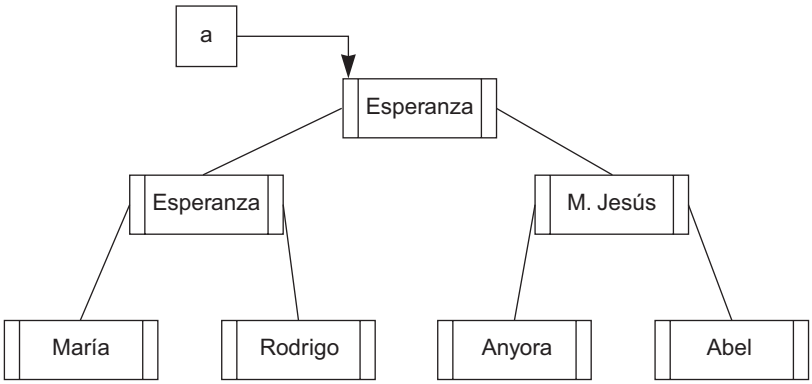


Figura 16.18. Árbol binario de cadenas.

16.4. ÁRBOL DE EXPRESIÓN

Una aplicación muy importante de los árboles binarios son los *árboles de expresiones*. Una **expresión** es una secuencia de *tokens* (componentes de léxicos que siguen unas reglas establecidas). Un *token* puede ser un operando, o bien un operador.

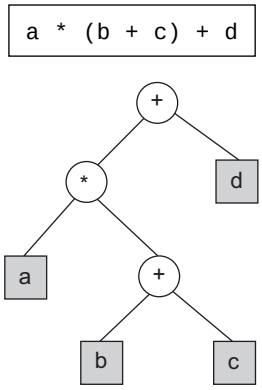


Figura 16.19. Una expresión infija y su árbol de expresión.

La Figura 16.19 representa la expresión *infija* `a * (b + c) + d` junto a su árbol de expresión. El nombre de *infija* es debido a que los operadores se sitúan *entre* los operandos.

Un **árbol de expresión** es un árbol binario con las siguientes propiedades:

1. Cada hoja es un operando.
2. Los nodos raíz y nodos internos son operadores.
3. Los subárboles son subexpresiones cuyo nodo raíz es un operador.

Los árboles binarios se utilizan para representar expresiones en memoria; esencialmente, en compiladores de lenguajes de programación. Se observa que los paréntesis de la expresión no aparecen en el árbol, pero están implicados en su forma, y esto resulta muy interesante para la evaluación de la expresión.

Si se supone que todos los operadores tienen dos operandos, se puede representar una expresión mediante un árbol binario cuya raíz contiene un operador y cuyos subárboles izquierdo y derecho, son los operandos izquierdo y derecho respectivamente. Cada operando puede ser una letra (`x, y, a, b` etc.) o una subexpresión representada como un subárbol. La Figura 16.20 muestra un árbol cuya raíz es el operador `*`, su subárbol izquierdo representa la subexpresión `(x + y)` y su subárbol derecho representa la subexpresión `(a - b)`. El nodo raíz del subárbol izquierdo contiene el operador (+) de la subexpresión izquierda y el nodo raíz del subárbol derecho contiene el operador (-) de la subexpresión derecha. Todos los operandos letras se almacenan en nodos hojas.

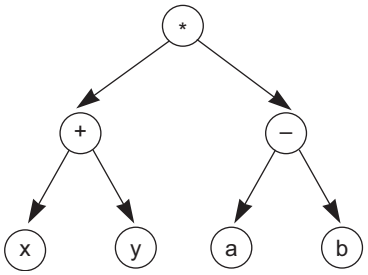


Figura 16.20. Árbol de expresión `(x + y)*(a - b)`.

Utilizando el razonamiento anterior, la expresión `(x * (y - z)) + (a - b)` con paréntesis alrededor de las subexpresiones, forma el árbol binario de la Figura 16.21.

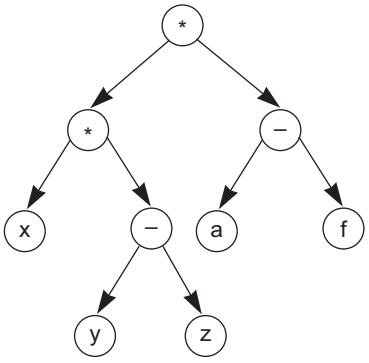
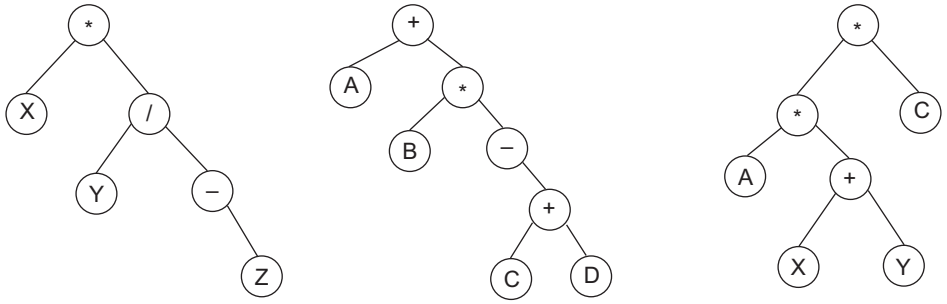


Figura 16.21. Árbol de expresión `(x * (y - z))+(a - b)`.

EJEMPLO 16.4. Deducir las expresiones que representan los árboles binarios de la Figura 16.22.



a) *b*) *c*)

Figura 16.22. Árboles de expresión.

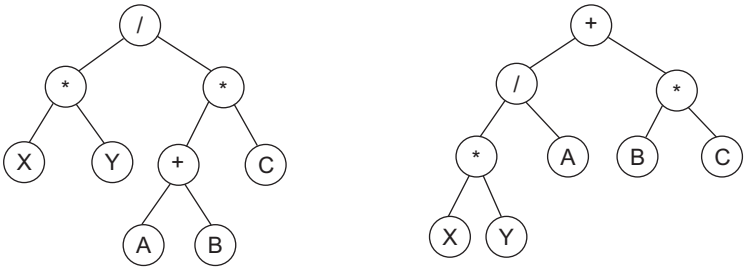
Soluciones

a. `(X * Y /(-Z))`
b. `(A + B * (-(C + D)))`
c. `(A * (X + Y))* C)`

EJEMPLO 16.5. Dibujar la representación en árbol binar io de cada una de las siguientes expresiones:

a. `X * Y /((A + B)* C)`
b. `X * Y /A + B * C`

Soluciones



a) *b*)

Figura 16.23. Árboles de expresión.

16.4.1. Reglas para la construcción de árboles de expresiones

Los árboles de expresiones se utilizan en las computadoras para evaluar expresiones usadas en programas. El algoritmo más sencillo para construir un árbol de expresión es aquél que lee una

expresión completa que contiene paréntesis en la misma. Una expresión con paréntesis es aquélla en que:

1. La prioridad se determina sólo por paréntesis.
2. La expresión completa se sitúa entre paréntesis.

A fin de ver la prioridad en las expresiones, considérese la expresión:

```
a * c + e / g - (b + d)
```

Los operadores con prioridad más alta son `*` y `/`; es decir:

```
(a * c) + (e / g) - (b + d)
```

Los operadores que siguen en orden de prioridad son `+` y `-`, que se evalúan de izquierda a derecha. Por consiguiente, se puede escribir:

```
((a * c) + (e / g)) - (b + d)
```

Por último, la expresión completa entre paréntesis será:

```
(((a * c) + (e / g)) - (b + d))
```

EJEMPLO 16.6. Determinar las expresiones correspondientes de los árboles de expresión de la Figura 16.24.

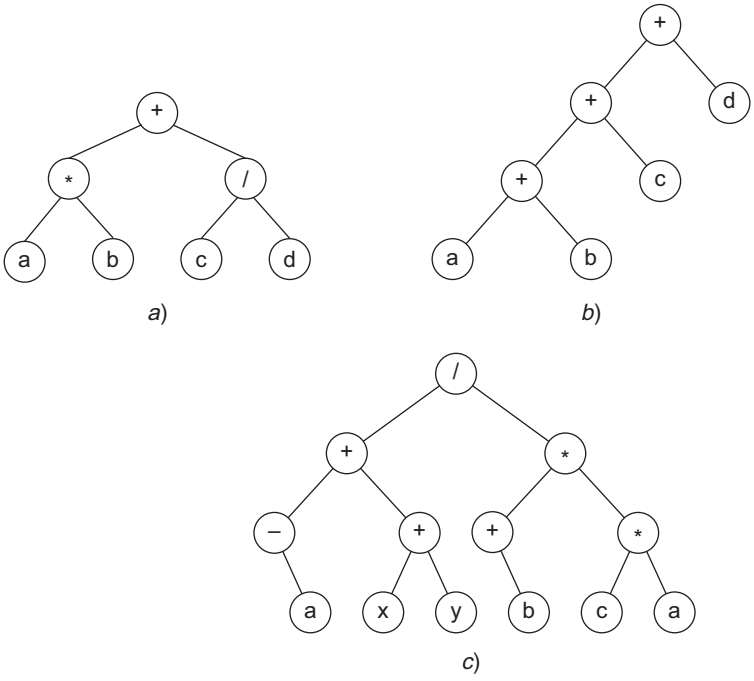


Figura 16.24. Árboles de expresión.

Las soluciones correspondientes son:

a. ((a * b) + (c / d))
b. (((a + b) + c) + d)
c. (((-a) + (x + y)) / ((+b) * (c * a)))

16.5. RECORRIDO DE UN ÁRBOL

Para visualizar o consultar los datos almacenados en un árbol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. Al contrario que las listas enlazadas, los árboles binarios no tienen realmente un primer valor, un segundo valor, tercer valor, etc. Se puede afirmar que el nodo raíz viene el primero, pero ¿quién viene a continuación? Existen diferentes métodos de recorrido de árbol ya que la mayoría de las aplicaciones binarias son bastante sensibles al orden en el que se visitan los nodos, de forma que será preciso elegir cuidadosamente el tipo de recorrido.

El **recorrido de un árbol binario** requiere que cada nodo del árbol sea procesado (visitado) una vez y sólo una en una secuencia predeterminada. Existen dos enfoques generales para la secuencia de recorrido, profundidad y anchura.

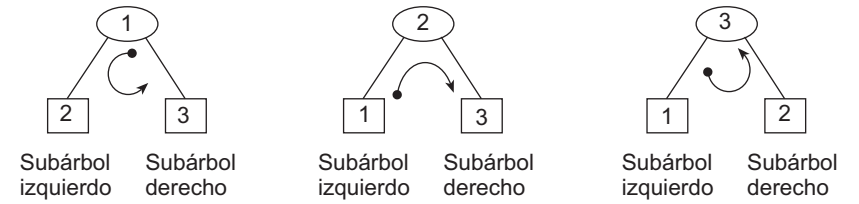
En el **recorrido en profundidad**, el proceso exige un camino desde el raíz a través de un hijo, al descendiente más lejano del primer hijo antes de proseguir a un segundo hijo. En otras palabras, en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo.

En el **recorrido en anchura**, el proceso se realiza horizontalmente desde el raíz a todos sus hijos, a continuación a los hijos de sus hijos y así sucesivamente hasta que todos los nodos han sido procesados. En el recorrido en anchura, cada nivel se procesa totalmente antes de que comience el siguiente nivel.

Definición

El **recorrido** de un árbol supone visitar cada nodo sólo una v ez.

Dado un árbol binario que consta de raíz, un subárbol izquierdo y un subárbol derecho se pueden definir tres tipos de secuencia de recorrido en profundidad. Estos recorridos estándar se muestran en la Figura 16.25.



a) **Recorrido *preorden*** *b*) **Recorrido *enorden*** *c*) **Recorrido *postorden***

Figura 16.25. Recorridos de árboles binarios.

La designación tradicional de los recorridos utiliza un nombre para el nodo raíz (**N**), para el subárbol izquierdo (**I**) y para el subárbol derecho (**D**).

16.5.1. Recorrido *preorden*

El recorrido *preorden*[2] (**NID**) conlleva los siguientes pasos, en los que el nodo raíz va antes que los subárboles:

1. Visitar el nodo raíz (**N**).
2. Recorrer el subárbol izquierdo (**I**) en preorden.
3. Recorrer el subárbol derecho (**D**) en preorden.

Dado las características recursivas de los árboles, el algoritmo de recorrido tiene naturaleza recursiva. Primero, se procesa la raíz, a continuación el subárbol izquierdo y a continuación el subárbol derecho. Para procesar el subárbol izquierdo se siguen los mismos pasos: raíz, subárbol izquierdo y subárbol derecho (proceso recursivo). Luego se hace lo mismo con el subárbol derecho.

> ***Regla***
>
> En el recorrido preorden, el raíz se procesa antes que los subárboles izquierdo y derecho.

Si utilizamos el recorrido *preorden* del árbol de la Figura 16.26 se visita primero el raíz (nodo `A`); a continuación, se visita el subárbol izquierdo de `A`, que consta de los nodos `B`, `D` y `E`. Dado que el subárbol es a su vez un árbol, se visitan los nodos utilizando el mismo orden (`NID`). Por consiguiente, se visita primero el nodo `B`, después `D` (izquierdo) y por último `E` (derecho).

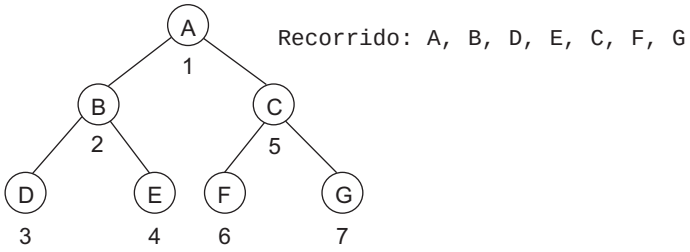


Figura 16.26. Recorrido *preorden* de un árbol binario.

A continuación, se visita subárbol derecho de `A`, que es un árbol que contiene los nodos `C`, `F` y `G`. De nuevo, siguiendo el mismo orden (`NID`), se visita primero el nodo `C`, a continuación `F` (izquierdo) y por último `G` (derecho). En consecuencia, el orden del recorrido *preorden* del árbol de la Figura 16.26 es `A-B-D-E-C-F-G`.

[2] El nombre *preorden*, viene del prefijo latino *pre* que significa "ir antes".

16.5.2. Recorrido *enorden*

El recorrido **enorden** (*inorder*) procesa primero el subárbol izquierdo, después el raíz y a continuación el subárbol derecho. El significado de *en* (*in*) es que la raíz se procesa entre los subárboles.

Si el árbol no está vacío, el método implica los siguientes pasos:

1. Recorrer el subárbol izquierdo (**I**) en enorden.
2. Visitar el nodo raíz (**N**).
3. Recorrer el subárbol derecho (**D**) en enorden.

En el árbol de la Figura 16.27, los nodos se han numerado en el orden en que son visitados durante el recorrido *enorden*. El primer subárbol recorrido es el subárbol izquierdo del nodo raíz (árbol cuyo nodo contiene la letra `B`). Este subárbol es, a su vez, otro árbol con el nodo `B` como raíz, por lo que siguiendo el orden `IND`, se visita primero `D`, a continuación `B` (nodo raíz) y por último `E` (derecha). Después, se visita el nodo raíz, `A`. Por último, se visita el subárbol derecho de `A`, siguiendo el orden `IND` se visita primero `F`, después `C` (nodo raíz) y por último `G`. Por consiguiente, el orden del recorrido enorden del árbol de la Figura 16.27 es `D-B-E-A-F-C-G`.

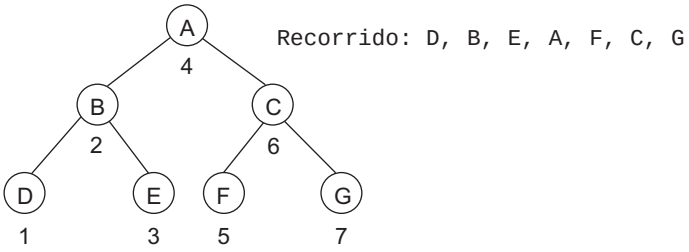


Figura 16.27. Recorrido *enorden* de un árbol binario.

16.5.3. Recorrido *postorden*

El recorrido *postorden* (**IDN**) procesa el nodo raíz (*post*) después de que los subárboles izquierdo y derecho se han procesado. Se comienza situándose en la hoja más a la izquierda y se procesa. A continuación se procesa su subárbol derecho. Por último, se procesa el nodo raíz. Las etapas del algoritmo, si el árbol no está vacío, son:

1. Recorrer el subárbol izquierdo (**I**) en postorden.
2. Recorrer el subárbol derecho (**D**) en postorden.
3. Visitar el nodo raíz (**N**).

Si se utiliza el recorrido *postorden* del árbol de la Figura 16.27 se visita primero el subárbol izquierdo de `A`. Este subárbol consta de los nodos `B`, `D` y `E`, y siguiendo el orden `IDN`, se visitará primero `D` (izquierdo), luego `E` (derecho) y, por último, `B` (nodo). A continuación, se visita el subárbol derecho de `A` que consta de los nodos `C`, `F` y `G`. Siguiendo el orden `IDN` para este árbol, se visita primero `F` (izquierdo), después `G` (derecho) y, por último, `C` (nodo). Finalmente se visita el nodo raíz `A`. Resumiendo, el orden del recorrido *postorden* del árbol de la Figura 16.27 es `D-E-B-F-G-C-A`.

EJERCICIO 16.1. Deducir el orden de los elementos en cada uno de los tres recorr idos fundamentales de los árboles binar ios siguientes.

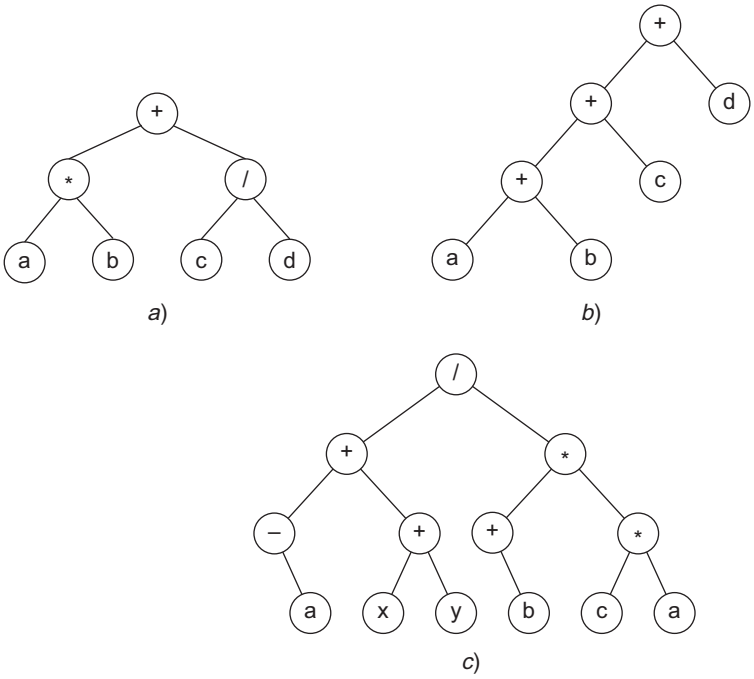


Figura 16.28. Árboles de expresión.

Los elementos de los árboles binarios listados en preorden, enorden y postorden.

	Árbol a	***Árbol b***	***Árbol c***
preorden	`+*ab/cd`	`+++abcd`	`/+-a+xy*+b*cd`
enorden	`a*b+c/d`	`a+b+c+d`	`-a+x+y/+b*c*d`
postorden	`ab*cd/+`	`ab+c+d+`	`a-xy++b+cd**/`

16.5.4. Implementación

Teniendo en cuenta que los recorridos en profundidad de un árbol binario se han definido recursivamente, las funciones que lo implementan es natural que tengan naturaleza recursiva. Prácticamente, todo consiste en trasladar la definición a la codificación. Las funciones se han de declarar en la clase `ArbolBinario` de modo público. Estas funciones `preorden()`, `inorden()`, `postorden()` deben invocar a las funciones privados de la misma clase `ArbolBinario` que tienen como argumento el atributo `raiz` que es un puntero a la clase `Nodo`. El *caso*

base, para detener la recursión, de estas funciones miembro privadas es que la raíz del árbol esté vacía (`raiz == NULL`).

```
class ArbolBinario
{

protected:

  //

public:

  // recorrido en preorden

  void preorden()
  {
    preorden(raiz);
  }

  // recorrido en ineorden

  void inorden()
  {
    inorden(raiz);
  }

  // recorrido en postorden

  void postorden()
  {
    postorden(raiz);
  }

private:

      // Recosrrido de un árbol binario en preorden
  void preorden(Nodo *r)
  {
      if (r != NULL)
      {
        r->visitar();
        preorden (r->subarbolIzdo());
        preorden (r->subarbolDcho());
      }
  };

      // Recorrido de un árbol binario en inorden

      void inorden(Nodo *r)
  {
      if (r != NULL)
      {
          inorden (r->subarbolIzdo());
          r->visitar();
          inorden (r->subarbolDcho());
      }
  }
```

```
  // Recorrido de un árbol binario en postorden
  void postorden(Nodo *r)
  {
    if (r != NULL)
    {
      postorden (r->subarbolIzdo());
      postorden (r->subarbolDcho());
      r->visitar();
    }
  }

  // Recorrido de un árbol binario en preorden
  void preorden(Nodo *r)
  {
    if (r != NULL)
    {
        r->visitar();
        preorden (r->subarbolIzdo());
        preorden (r->subarbolDcho());
    }
  }
}
```

Nota de programación

La visita al nodo se representa mediante la llamada al método de `Nodo, visitar()`. ¿Qué hacer en el método?, depende de la aplicación que se esté realizando . Si simplemente se quiere listar los nodos, puede emplearse la siguiente sentencia:

```
void visitar()
{
   cout << dato << endl;
}
```

EJEMPLO 16.7. Dado un árbol binario, eliminar cada uno de los nodos de que consta.

La función `vaciar()` (clase `ArbolBinario`) utiliza un recorrido *postorden*, de tal forma que el hecho de *visitar* el nodo se convierte en liberar la memoria del nodo con el operador `delete`. Este recorrido asegura la liberación de la memoria ocupada por un nodo después de hacerlo a su rama izquierda y derecha.

La función `vaciar(Nodo*r)` es una función miembro privada de la clase `ArbolBinario`. La codificación es:

```
void ArbolBinario::vaciar(Nodo *r)
{
    if (r != NULL)
    {
      vaciar(r->subarbolIzdo());
      vaciar(r->subarbolDcho());
      cout << "\tNodo borrado. ";
      r = NULL;
    }
}
```

La codificación de `vaciar()` de la clase `ÁrbolBinario` es:

```
void ArbolBinario::vaciar()
{
    vaciar(raiz);
}
```

16.6. IMPLEMENTACIÓN DE OPERACIONES

Se implementan propiedades y operaciones de árboles binarios. Ya que un árbol binario es un tipo de dato definido recursivamente, las funciones que implementan las operaciones tienen naturaleza recursiva, aunque siempre es posible la implementación iterativa. Las funciones que se escriben son de la clase `ArbolBinario`, tienen como argumento, normalmente, el nodo raíz del árbol o subárbol actual.

Altura de un árbol binario

El caso más sencillo de cálculo de la altura es cuando el árbol está vacío, en cuyo caso la altura es `0`. Si el árbol no está vacío, cada subárbol debe tener su propia altura, por lo que se necesita evaluar cada una por separado. Las variables `alturaIz`, `alturaDr` almacenan las alturas de los subárboles izquierdo y derecho respectivamente.

El método de cálculo de la altura de los subárboles utiliza llamadas recursivas a `altura()` con referencias a los respectivos subárboles como parámetros de la misma. Devuelve la altura del subárbol más alto más `1` (la misma del raíz).

```
int altura(Nodo *raiz)
{
  if (raiz == NULL)
    return 0 ;
  else
  {
    int alturaIz = altura (raiz->subarbolIzdo());
    int alturaDr = altura (raiz->subarbolDcho());
    if (alturaIz > alturaDr)
      return alturaIz + 1;
    else
      return alturaDr + 1;
  }
}
```

La función tiene complejidad lineal, *O(n)*, para un árbol de *n* nodos.

Árbol binario lleno

Un árbol binario lleno tiene el máximo número de nodos; esto equivale a que cumpla la propiedad de que todo nodo, excepto si es hoja, tiene dos descendientes. También, un árbol está lleno si para todo nodo, la altura de su rama izquierda es igual que la altura de su rama derecha. Con recursividad la condición es fácl de determinar:

```
bool arbolLleno(Nodo *raiz)
{
  if (raiz == NULL)
```

```
        return true;
    else
    if (altura(raiz->subarbolIzdo())!=
        altura(raiz->subarbolDcho()))
        return false;

    return arbolLleno(raiz->subarbolIzdo()) &&
           arbolLleno(raiz->subarbolDcho());
}
```

La función tiene *peor* tiempo de ejecución que `altura()`. Cada llamada recursiva a `arbolLleno()` implica llamar a `altura()` desde el siguiente nivel del árbol. La complejidad es *O(n log n)*, para un árbol lleno de `n` nodos.

Número de nodos

El número de nodos es `1` (nodo raíz) más el número de nodos del subárbol izquierdo y derecho. El número de nodos de un árbol vacío es `0`, que será *el caso base* de la recursividad.

```
int numNodos(Nodo *raiz)
{
  if (raiz == NULL)
       return 0;
    else
      return 1 + numNodos(raiz->subarbolIzdo()) +
                 numNodos(raiz->subarbolDcho());
}
```

El método accede a cada nodo del árbol, por ello tiene complejidad lineal, *O(n)*.

Copia de un árbol binario

El planteamiento es sencillo, se empieza creando una copia del nodo raíz, y se enlaza con la copia de su rama izquierda y derecha respectivamente. La función que implementa la operación, de la clase `ArbolBinario`, tiene como parámetro un puntero a la clase `Nodo` y devuelve la referencia al nodo raíz. Además, se escribe la función `copiaArbol(ArbolBinario &a)` de la clase `Arbolbinario` que tiene como parámetro un árbol binario por referencia que es donde se copia el objeto actual.

```
void ArbolBinario::copiaArbol(ArbolBinario &a)
{
     a.Praiz(copiaArbol(this->raiz));
}

Nodo* ArbolBinario::copiaArbol(Nodo* raiz)
{
    Nodo *raizCopia;
    if (raiz == NULL)
      raizCopia = NULL;
    else
    {
      Nodo* izdoCopia, *dchoCopia;
      izdoCopia = copiaArbol(raiz->subarbolIzdo());
```

```
        dchoCopia = copiaArbol(raiz->subarbolDcho());
        raizCopia = new Nodo( izdoCopia,raiz->valorNodo(), dchoCopia);
      }
      return raizCopia;
  }
```

Al igual que la función `altura()`, o `numNodos()`, se accede a cada nodo del árbol, por ello es de complejidad lineal, *O(n)*.

16.6.1. Evaluación de un árbol de expresión

En la Sección 16.5 se han construido diversos árboles binarios para representar expresiones algebraicas. La evaluación consiste en, una vez dados valores numéricos a los operandos, obtener el valor resultante de la expresión.

El algoritmo evalúa expresiones con operadores algebraicos binarios: `+`, `-`, `*`, `/` y *potenciación* (`^`). Se basa en el recorrido del árbol de la expresión en *postorden*. De esta forma se evalúa primer operando (*rama izquierda*), segundo operando (*rama derecha*) y, según el operador (nodo raíz), se obtiene el resultado.

A tener en cuenta

Una cuestión importante a considerar es que los operandos siempre son nodos hoja en el árbol binario de expresiones.

Los operandos se representan mediante letras mayúsculas. Por ello, el vector `operandos[]` tiene tantos elementos como letras mayúsculas tiene el código `ASCII`, de tal forma que la posición `0` se corresponde con `'A'`, y así sucesivamente hasta la `'Z'`. El valor numérico de un operando se encuentra en la correspondiente posición del vector.

```
double operandos[26];
```

Por ejemplo, si los operandos son `A`, `D` y `G`; sus respectivos valores se guardan en las posiciones: `operandos[0]`, `operandos[3]` y `operandos[6]`, respectivamente.

El método `evaluar()` tiene como entrada la raíz del árbol y el array con los valores de los operandos. Devuelve el resultado de la evaluación.

```
double evaluar(ArbolBinario a, double operandos[])
{
  double x, y;
  char ch ;
  Nodo *raiz;
  raiz = a.Oraiz();
  if (raiz != NULL)      // no está vacío
  {
    ch = raiz->valorNodo();
    if (ch >= 'A' && ch <= 'Z')
        return operandos[ch - 'A'];
    else
    {
        x = evaluar(raiz->subarbolIzdo(), operandos);
```

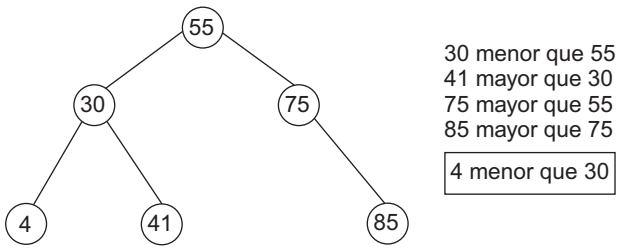
```
        y = evaluar(raiz->subarbolDcho(), operandos);
        switch (ch) {
          case '+': return x + y;
                    break;
          case '-': return x - y;
                    break;
          case '*': return x * y;
                    break;
          case '/': if (y != 0)
                        return x/y;
                    else
                        throw "Error: división por 0";
                    break;
          case '^': return pow(x, y);
        }
      }
    }
}
```

16.7. ÁRBOL BINARIO DE BÚSQUEDA

Los árboles estudiados hasta ahora no tienen un orden definido; sin embargo, los árboles binarios ordenados tienen sentido. Estos árboles se denominan árboles binarios de búsqueda, debido a que se pueden buscar en ellos un término utilizando un algoritmo de búsqueda binaria similar al empleado en arrays.

Un **árbol binario de búsqueda** es aquel que dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que sus propios datos. El árbol binario del Ejemplo 16.8 es de búsqueda.

EJEMPLO 16.8. Árbol binario de búsqueda para nodos con el campo de datos de tipo int.

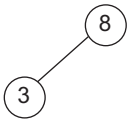


16.7.1. Creación de un árbol binario de búsqueda

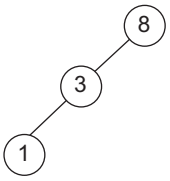
Se desea almacenar los números `8 3 1 20 10 5 4` en un árbol binario de búsqueda, siguiendo la regla: "dado un nodo en el árbol todos los datos a su izquierda deben ser menores que el dato del nodo actual, mientras que todos los datos a la derecha deben ser mayores que

el del nodo actual". Inicialmente, el árbol está vacío y se desea insertar el `8`. La única elección es almacenar el `8` en el raíz:

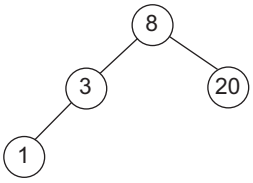
A continuación viene el `3`. Ya que `3` es menor que `8`, el `3` debe ir en el subárbol izquierdo.



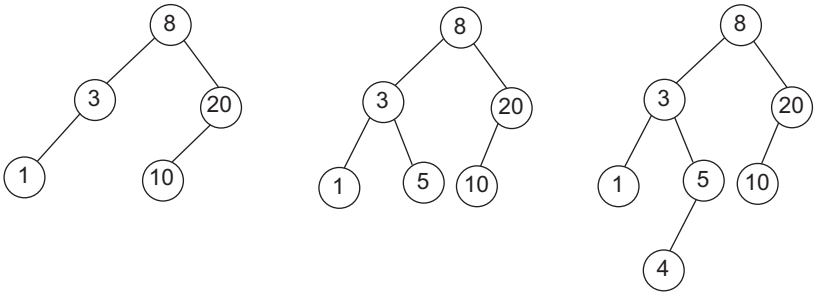
A continuación se ha de insertar `1` que es menor que `8` y que `3`, por consiguiente, irá a la izquierda y debajo de `3`.



El siguiente número es `20`, mayor que `8`, lo que implica debe ir a la derecha de `8`.



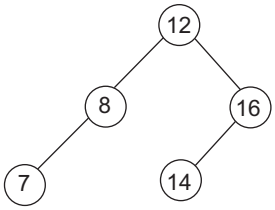
Cada nuevo elemento se inserta como una *hoja* del árbol. Los restantes elementos se pueden situar fácilmente.



Una propiedad de los árboles binarios de búsqueda es que no son únicos para los mismos datos.

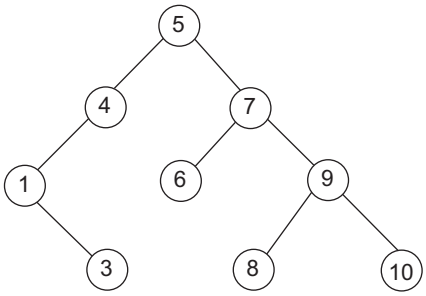
EJEMPLO 16.10. Construir un árbol binario para almacenar los datos 12, 8, 7, 16 y 14.

Solución



EJEMPLO 16.11. Construir un árbol binario de búsqueda que corresponda a un recorrido enorden cuyos elementos son: 1, 3, 4, 5, 6, 7, 8, 9 y 10.

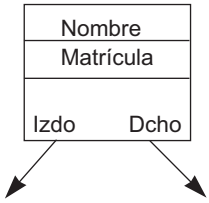
Solución



16.7.2. Nodo de un árbol binario de búsqueda

Un nodo de un árbol binario de búsqueda no difiere en nada de los nodos de un árbol binario, tiene un atributo de datos y dos enlaces a los subárboles izquierdo y derecho respectivamente. Un árbol de búsqueda se puede utilizar cuando se necesita que la información se encuentre rápidamente. Un ejemplo de árbol binario de búsqueda es el que cada nodo contiene información relativa a un estudiante. Cada nodo almacena el nombre del estudiante, y el número de matrícula en su universidad (dato entero), que puede ser el utilizado para ordenar.

Declaración de tipos




```
class Nodo {
  protected:
```

```
            int nummat;
            char nombre[30];
            Nodo *izdo, *dcho;

    public:
    //...
};

class ArbolBinario
{
    protected:
        Nodo *raiz;
    public:
    //...
};
```

16.8. OPERACIONES EN ÁRBOLES BINARIOS DE BÚSQUEDA

Los árboles binarios de búsqueda, al igual que los árboles binarios, tienen naturaleza recursiva y, en consecuencia, las operaciones sobre los árboles son recursivas, si bien siempre se tiene la opción de realizarlas de forma iterativa. Estas operaciones son:

- *Búsqueda* de un nodo. Devuelve la referencia al nodo del árbol, o `NULL`.
- *Inserción* de un nodo. Crea un nodo con su dato asociado y lo añade, en orden, al árbol.
- *Borrado* de un nodo. Busca el nodo del árbol que contiene un dato y lo quita del árbol. El árbol debe seguir siendo de búsqueda.
- *Recorrido* de un árbol. Los mismos recorridos de un árbol binario *preorden*, *inorden* y *postorden*.

16.8.1. Búsqueda

La búsqueda de un nodo comienza en el nodo raíz y sigue estos pasos:

1. La clave buscada se compara con la clave del nodo raíz.
2. Si las claves son iguales, la búsqueda se detiene.
3. Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el subárbol derecha. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda con el subárbol izquierdo.

> ***A recordar***
>
> En los árboles binarios ordenados la búsqueda de una clave da lugar a un *camino de búsqueda,* de tal forma que *baja* por la rama izquierda si la clave buscada es menor que la clave del raíz, *baja* por la rama derecha si la clave es mayor.

Implementación

Si se desea encontrar un nodo en el árbol que contenga una información determinada. La función pública `buscar()` de la Clase `ArbolBinario` realiza una llamada la función privada `buscar()` de la clase `ArbolBinario` que tiene dos parámetros, un puntero a un `Nodo` y el dato que se busca. Como resultado, la función devuelve un puntero al nodo en el que se almacena la información; en el caso de que la información no se encuentre se devuelve el valor `NULL`. El algoritmo de búsqueda es el siguiente:

1. Si el nodo raíz contiene el dato buscado, la tarea es fácil: el resultado es, simplemente, su referencia y termina el algoritmo.
2. Si el árbol no está vacío, el subárbol específico por donde proseguir depende de que el dato requerido sea menor o mayor que el dato del raíz.
3. El algoritmo termina si el árbol está vacío, en cuyo caso devuelve `NULL`.

Código de la función pública `buscar()`de la clase `ArbolBinario`

```
Nodo* ArbolBinario::buscar(Tipoelemento buscado)
{
   return buscar(raiz, buscado);
}
```

Código de la función pública `buscar()`de la clase `ArbolBinario`

```
Nodo* arbolBinario::buscar(Nodo* raizSub, Tipoelemento buscado)
{
   if (raizSub == NULL)
      return NULL;
   else if (buscado == raizSub->valorNodo())
      return raizSub;
   else if (buscado < raizSub->valorNodo())
      return buscar(raizSub->subarbolIzdo(), buscado);
   else
      return buscar (raizSub->subarbolDcho(), buscado);
}
```

El Ejemplo 16.12 implementa la operación de búsqueda con un esquema iterativo.

EJEMPLO 16.12. Aplicar el algoritmo de búsqueda de un nodo en un árbol binario ordenado para implementar la operación buscar iterativamente.

La función privada de la clase `ArbolBinario`, se codifica con un bucle cuya condición de parada es que se encuentre el nodo, o bien que el *camino de búsqueda* haya finalizado (subárbol vacío, `NULL`). La función *mueve* por el árbol el puntero a la clase `Nodo raizSub`, de tal forma que baja por la rama izquierda o derecha según la clave sea menor o mayor que la clave del nodo actual.

```
Nodo* ArbolBinario:: buscarIterativo (Tipoelemento buscado)
 {
  Tipoelemento dato;
  bool encontrado = false;
```

```
    Nodo* raizSub = raiz;
    dato = buscado;
    while (!encontrado && raizSub != NULL)
    {
      if (dato == raizSub->valorNodo())
        encontrado = true;
      else if (dato < raizSub->valorNodo())
        raizSub = raizSub->subarbolIzdo();
      else
        raizSub = raizSub->subarbolDcho();
    }
    return raizSub;
  }
```

16.8.2. Insertar un nodo

Para añadir un nodo al árbol se sigue el camino de búsqueda, y al final del camimo se enlaza el nuevo nodo, por consiguiente, siempre se inserta como hoja del árbol. El árbol que resulta después de insertar sigue siendo siempre de búsqueda.

En esencia, el algoritmo de inserción se apoya en la búsqueda de un elemento, de modo que si se encuentra el elemento buscado, no es necesario hacer nada (o bien se usa una estructura de datos auxiliar para almacenar la información); en caso contrario, se inserta el nuevo elemento justo en el lugar donde ha acabado la búsqueda (es decir, en el lugar donde habría estado en el caso de existir).

Por ejemplo, al árbol de la Figura 16.29 se le va a añadir el nodo `8`. El proceso describe un *camino de búsqueda* que comienza en el raíz `25`; el nodo `8` debe estar en el subárbol izquierdo de `25` (`8 < 25`). El nodo `10` es el raíz del subárbol actual, el nodo `8` debe estar en el subárbol izquierdo (`8 < 10`), que está actualmente vacío y, por tanto, ha terminado el *camino de búsqueda*. El nodo `8` se enlaza como hijo izquierdo del nodo `10`.

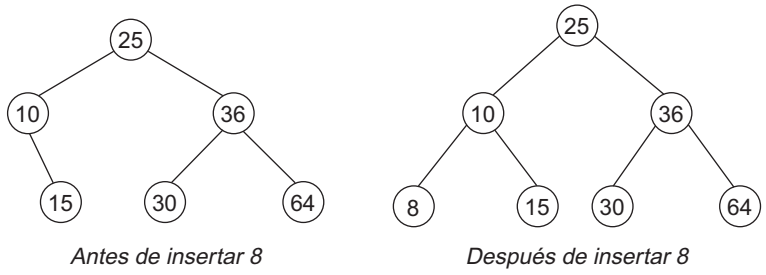
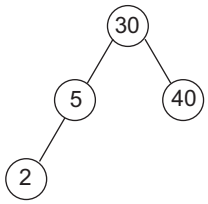


Figura 16. 29. Inserción en un árbol binario de búsqueda.

A recordar

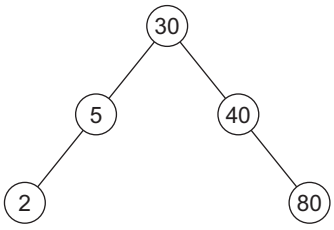
La inserción de un nuevo nodo en un árbol de búsqueda siempre se hace como nodo hoja. Para ello se *baja* por el árbol según el *camino de búsqueda.*

EJEMPLO 16.13. Insertar un elemento con cla ve 80 en el árbol binar io de búsqueda siguiente:

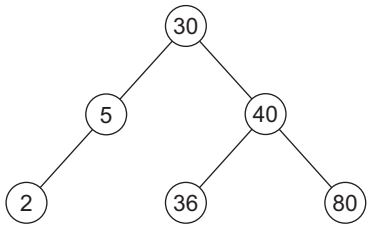


A continuación, insertar un elemento con clave 36 en el árbol binario de búsqueda resultante.

Solución



a) Inserción de 80



b) Inserción de 36

Implementación

La función `insertar()` de la clase `ArbolBinario` es el interfaz de la operación, llama la función recursiva que realiza la operación y devuelve la raíz del nuevo árbol. A esta función interna se le pasa la raíz actual, a partir de ella describe el *camino de búsqueda* y, al final, se enlaza. En un árbol binario de búsqueda no hay nodos duplicados, por ello si se encuentra un nodo igual que el que se desea insertar se lanza un excepción (o una nueva función que inserte en una estructura de datos auxiliar del propio nodo).

```
void ArbolBinario::insertar (Tipoelemento valor)
{
  raiz = insertar(raiz, valor);
}
Nodo* ArbolBinario::insertar(Nodo* raizSub, Tipoelemento dato)
{
  if (raizSub == NULL)
    raizSub = new Nodo(dato);
  else if (dato < raizSub->valorNodo())
  {
    Nodo *iz;
    iz = insertar(raizSub->subarbolIzdo(), dato);
    raizSub->ramaIzdo(iz);
  }
  else if (dato > raizSub->valorNodo())
```

```
    {
      Nodo *dr;
      dr = insertar(raizSub->subarbolDcho(), dato);
      raizSub->ramaDcho(dr);
    }
    else
      throw "Nodo duplicado"; // tratamiento de repetición
    return raizSub;
  }
```

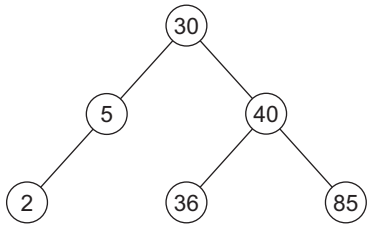
16.8.3. Eliminar un nodo

La operación de *borrado* de un nodo es también una extensión de la operación de búsqueda, si bien más compleja que la inserción, debido a que el nodo a suprimir puede ser cualquiera y la operación debe mantener la estructura de árbol binario de búsqueda después de quitar el nodo. Los pasos a seguir son:

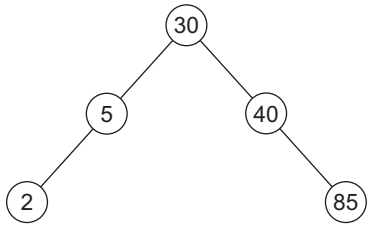
1. Buscar en el árbol la posición de "nodo a eliminar".
2. Si el nodo a suprimir tiene menos de dos hijos, reajustar los enlaces de su antecesor.
3. Si el nodo tiene dos hijos (rama izquierda y derecha), es necesario subir a la posición que éste ocupa el dato más próximo de sus subárboles (el inmediatamente superior o el inmediatamente inferior) con el fin de mantener la estructura árbol binario de búsqueda.

Los Ejemplos 16.14 y 16.15 muestran estas dos circunstancias, el primero elimina un nodo sin descendientes, el segundo elimina un nodo que, a su vez, es el raíz de un árbol con dos ramas no vacías.

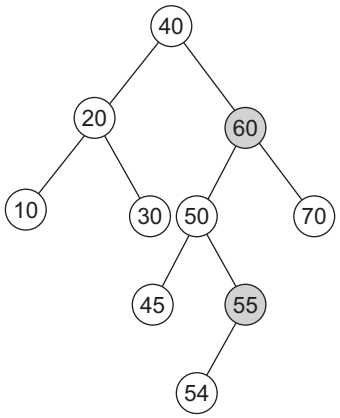
EJEMPLO 16.14. Suprimir el elemento de clave 36 del siguiente árbol binario de búsqueda:



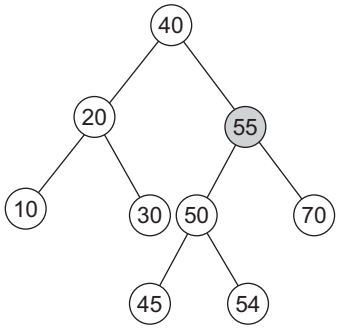
El nodo del árbol donde se encuentra la clave 36 es una hoja, por ello simplemente se reajustan los enlaces del nodo precedente en el camino de búsqueda. El árbol resultante:



EJEMPLO 16.15. Borrar el elemento de clave 60 del siguiente árbol:



Se reemplaza `60` por el elemento mayor (`55`) en su subárbol izquierdo, o por el elemento más pequeño (`70`) en su subárbol derecho. Si se opta por reemplazar por el mayor del subárbol izquierdo, se mueve el `55` al raíz del subárbol y se reajusta el árbol.

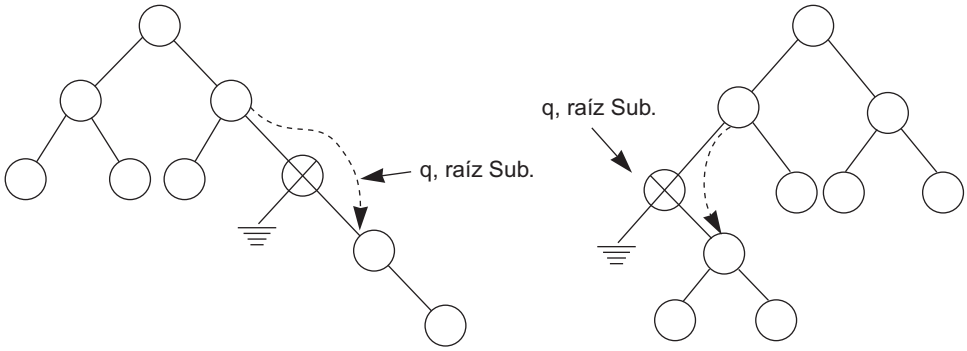


Implementación

La función `eliminar()` de la clase `ArbolBinario` es el interfaz de la operación, se le pasa el elemento que se va a buscar en el árbol para retirar su nodo; llama al método sobrecargado, privado, `eliminar()` con la raíz del árbol y el elemento.

La función lo primero que hace es buscar el nodo, siguiendo el *camino de búsqueda*. Una vez encontrado, se presentan dos casos claramente diferenciados. El primero, si el nodo a eliminar es una hoja o tiene un único descendiente, resulta una tarea fácil, ya que lo único que hay que hacer es asignar al enlace del nodo *padre* (según el *camino de búsqueda*) el descendiente del nodo a eliminar. El segundo caso, que el nodo tenga las dos ramas no vacías; esto exige, para mantener la estructura de árbol de búsqueda, reemplazar el dato del nodo por la *mayor de las claves menores* en el subárbol (otra posible alternativa: reemplazar el dato del nodo por la *menor de las claves mayores*). Como las claves menores están en la rama izquierda, se *baja* al primer nodo de la rama izquierda, y se continúa *bajando* por las ramas derecha (claves mayores) hasta alcanzar el nodo que no tiene rama derecha. Éste es el mayor de los menores, cuyo dato debe reemplazar al del nodo a eliminar. Lo que se hace es copiar el valor del dato y enlazar su padre con el hijo izquierdo. La función `reemplazar()` realiza la tarea descrita.

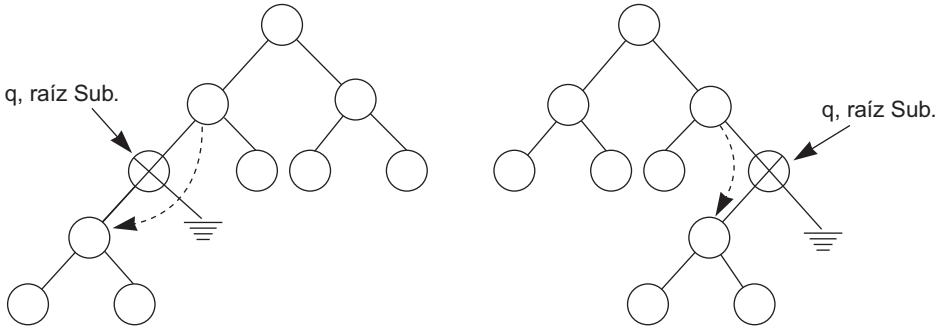
```
void ArbolBinario::eliminar (Tipoelemento valor)
{
  raiz = eliminar(raiz, valor);
}
Nodo* ArbolBinario::eliminar (Nodo *raizSub, Tipoelemento dato)
{
  if (raizSub == NULL)
    throw "No se ha encontrado el nodo con la clave";
  else if (dato < raizSub->valorNodo())
  {
    Nodo* iz;
    iz = eliminar(raizSub->subarbolIzdo(), dato);
    raizSub->ramaIzdo(iz);
  }
  else if (dato > raizSub->valorNodo())
  {
    Nodo *dr;
    dr = eliminar(raizSub->subarbolDcho(), dato);
    raizSub->ramaDcho(dr);
  }
  else     // Nodo encontrado
  {
    Nodo *q;
    q = raizSub;     // nodo a quitar del árbol
    if (q->subarbolIzdo() == NULL)
      raizSub = q->subarbolDcho();      // figura 16.30
    else if (q->subarbolDcho() == NULL)
      raizSub = q->subarbolIzdo();      // figura 16.31
    else
    {     // tiene rama izquierda y derecha
      q = reemplazar(q);      //figura 16.32
    }
    q = NULL;
  }
  return raizSub;
}
Nodo* ArbolBinario::reemplazar(Nodo* act)
{
  Nodo *a, *p;
  p = act;
  a = act->subarbolIzdo();     // rama de nodos menores
  while (a->subarbolDcho() != NULL)
  {
    p = a;
    a = a->subarbolDcho();
  }
  // copia en act el valor del nodo apuntado por a
  act->nuevoValor(a->valorNodo());
  if (p == act) // a es el hijo izquierdo de act
    p->ramaIzdo(a->subarbolIzdo()); // enlaza subarbol izquierdo. Fig. 16.32b
  else
    p->ramaDcho(a->subarbolIzdo()); // se enlaza subarbol derecho. Fig. 16.32a
     return a;
}
```



a) Es un hijo derecho

b) Es hijo derecho

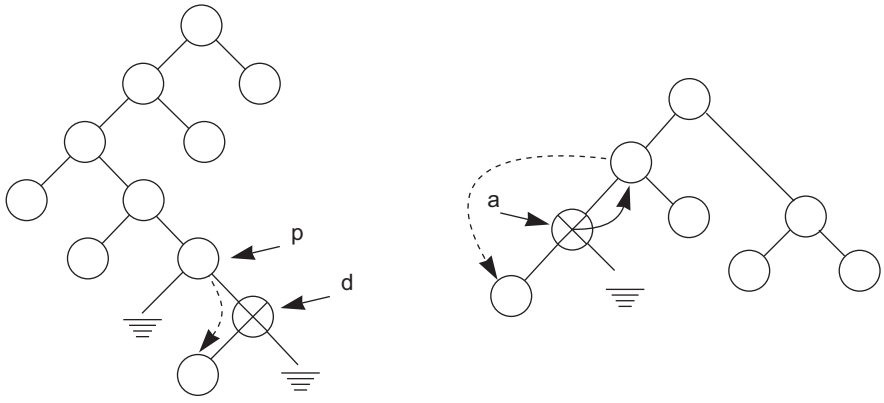
Figura 16.30. Eliminación de un nodo sin hijo izquierdo.



a) Es un hijo izquierdo

b) Es un hijo derecho

Figura 16.31. Eliminación de un nodo sin hijo derecho.



a) Se enlaza subárbol derecho

b) Se enlaza subárbol izquierdo

Figura 16.32. Eliminación de un nodo con dos hijos.

16.9. DISEÑO RECURSIVO DE UN ÁRBOL DE BÚSQUEDA

Los nodos utilizados en los árboles binarios y en los de búsqueda tienen dos enlaces, uno al nodo descendiente izquierdo y otro al derecho. Con una visión recursiva (ajustada a la propia definición de árbol) los dos enlaces pueden declararse a sendos subárboles, uno al subárbol izquierdo y el otro al subárbol derecho. La clase `ArbolBusqueda` tiene sólo un atributo `raiz` que es un puntero a la clase `Nodo`, por lo que para declararla, necesita que previamente se tenga un aviso de que la clase `Nodo` va a existir tal y como indica el siguiente código.

```
class Nodo;
class ÁrbolBusqueda
{
protected:
       Nodo* raiz;
public:
    ArbolBusqueda()
    {
      raiz = NULL;
    }
    ArbolBusqueda( Nodo *r)
    {
      raiz = r;
    }
    bool arbolVacio ()
    {
      return raiz == NULL;
    }
    void Praiz( Nodo* r){ raiz = r;}
    Nodo * Oraiz() { return raiz;}
};
```

La clase `Nodo` tiene tres atributos el dato para almacenar la información y dos atributos **arbolIzdo**, **arbolDcho** de la clase `ArbolBinario` previamente declarada.

```
typedef int Tipoelemento;

class Nodo
{
protected:
   Tipoelemento dato;
   ArbolBusqueda arbolIzdo;
   ArbolBusqueda arbolDcho;

public:
  Nodo(Tipoelemento valor)
  {
    dato = valor;
    arbolIzdo = ArbolBusqueda();// subarbol izquierdo vacío
    arbolDcho = ArbolBusqueda();// subarbol derecho vacío
  }

  Nodo(Tipoelemento valor, ArbolBusqueda ramaIzdo,
                          ArbolBusqueda ramaDcho)
```

```
  {
    dato = valor ;
    arbolIzdo = ramaIzdo;
    arbolDcho = ramaDcho;
  }
    // operaciones de acceso
  Tipoelemento valorNodo(){ return dato; }
  ArbolBusqueda subarbolIzdo(){ return arbolIzdo; }
  ArbolBusqueda subarbolDcho(){ return arbolDcho; }
  void nuevoValor(Tipoelemento d){ dato = d; }
  void ramaIzdo(ArbolBusqueda n){ arbolIzdo = n; }
  void ramaDcho(ArbolBusqueda n){ arbolIzdo = n; }
};
```

16.9.1. Implementación de las operaciones

La *búsqueda*, *inserción* y *borrado* bajan por el árbol, dando lugar al *camino de búsqueda*, con llamadas recursivas, siguiendo los detalles comentados en el Apartado 16.8 de este mismo capítulo.

Búsqueda

```
Nodo* ArbolBusqueda::buscar(Tipoelemento buscado)
{
  if (raiz == NULL)
    return NULL;
  else if (buscado == raiz->valorNodo())
    return raiz;
  else if (buscado < raiz->valorNodo())
    return raiz->subarbolIzdo().buscar(buscado);
  else
    return raiz->subarbolDcho().buscar(buscado);
}
```

Inserción

```
void ArbolBusqueda::insertar(Tipoelemento dato)
{
  if (!raiz)
    raiz = new Nodo(dato);
  else if (dato < raiz->valorNodo())
    raiz->subarbolIzdo().insertar(dato);//inserta por la izquierda
  else if (dato > raiz->valorNodo())
    raiz->subarbolDcho().insertar(dato); // inserta por la derecha
  else
    throw "Nodo duplicado";
}
```

Borrado

```
void ArbolBusqueda::eliminar (Tipoelemento dato)
{
  if (arbolVacio())
```

```
      throw "No se ha encontrado el nodo con la clave";
    else if (dato < raiz->valorNodo())
      raiz->subarbolIzdo().eliminar(dato);
    else if (dato > raiz->valorNodo())
      raiz->subarbolDcho().eliminar(dato);
    else    // nodo encontrado
    {
      Nodo *q;
      q = raiz;     // nodo a quitar del árbol
      if (q->subarbolIzdo().arbolVacio())
        raiz = q->subarbolDcho().raiz;
      else if (q->subarbolDcho().arbolVacio())
        raiz = q->subarbolIzdo().raiz;
      else
      {     // tiene rama izquierda y derecha
        q = reemplazar(q);
      }
      q->ramaIzdo(NULL);
      q->ramaDcho(NULL);
      q = NULL;
    }
  }

  Nodo* ArbolBusqueda:: reemplazar(Nodo* act)
  {
    Nodo* p;
    ArbolBusqueda a;
    p = act;
    a = act->subarbolIzdo();     // árbol con nodos menores
    while (a.raiz->subarbolDcho().arbolVacio())
    {
      p = a.raiz;
      a = a.raiz->subarbolDcho();
    }
    act->nuevoValor(a.raiz->valorNodo());
    if (p == act)
      p->ramaIzdo(a.raiz->subarbolIzdo());
    else
      p->ramaDcho(a.raiz->subarbolIzdo());
    return a.raiz;
  }
```

RESUMEN

En este capítulo se introdujo y desarrolló la estructura de datos dinámica árbol. Esta estructura, muy potente, se puede utilizar en una gran variedad de aplicaciones de programación.

La estructura árbol más utilizada normalmente es el *árbol binario*. Un **árbol binario** es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho. En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre, pero sí cada elemento restante tiene un padre.

La altura de un árbol binario es el número de ramas entre el raíz y la hoja más lejana, más 1. Si el árbol A es vacío, la altura es `0`. *El nivel* o *profundidad* de un elemento es un concepto similar al de altura.

Un árbol binario no vacío está *equilibrado totalmente* si sus subárboles izquierdo y derecho tienen la misma altura y ambos son o bien vacíos o totalmente equilibrados.

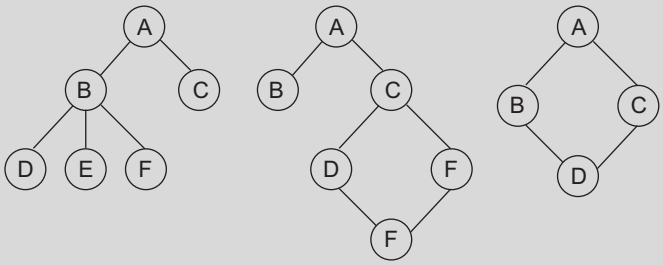
Los árboles binarios presentan dos tipos característicos: *árboles binarios de búsqueda* y *árboles binarios de expresiones*. Los árboles binarios de búsqueda se utilizan, fundamentalmente, para mantener una colección ordenada de datos y los árboles binarios de expresiones para almacenar expresiones.

BIBLIOGRAFÍA RECOMENDADA

Aho V.; Hopcroft, J., y Ullman, J.: *Estructuras de datos y algoritmos*. Addison Wesley, 1983.
Garrido, A., y Fernández, J.: *Abstracción y estructuras de datos en C++*. Delta, 2006.
Joyanes, L., y Zahonero, L.: *Algoritmos y estructuras de datos. Una perspectiva en C*. McGraw-Hill, 2004.
Joyanes, L.; Sánchez, L.; Zahonero, I., y Fernández, M.: *Estructuras de datos en C*. Schaum, 2005.
Weis, Mark Allen: *Estructuras de datos y algoritmos*. Addison Wesley, 1992.
Wirth Niklaus: *Algoritmos + Estructuras de datos = programas*. 1986.

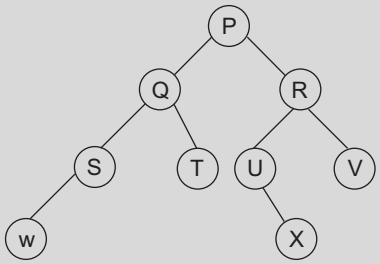
EJERCICIOS

16.1. Explicar porqué cada una de las siguientes estructuras no es un árbol binario.



16.2. Considérese el árbol siguiente:

a) ¿Cuál es su altura?
b) ¿Está el árbol equilibrado? ¿Porqué?
c) Listar todos los nodos hoja.
d) ¿Cuál es el predecesor inmediato (padre) del nodo `U`.
e) Listar los hijos del nodo `R`.
f) Listar los sucesores del nodo `R`.



16.3. Para cada una de las siguientes listas de letras.

a) Dibujar el árbol binario de búsqueda que se construye cuando las letras se insertan en el orden dado.
b) Realizar recorridos enorden, preorden y postorden del árbol y mostrar la secuencia de letras que resultan en cada caso.

(i) `M, Y, T, E, R` **(iii)** `R, E, M, Y, T`
(ii) `T, Y, M, E, R` **(iv)** `C, O, R, N, F, L, A, K, E, S`

16.4. En el árbol del Ejercicio 16.2, recorrer cada árbol utilizando los órdenes siguientes: NDI, DNI, DIN.

16.5. Dibujar los árboles binarios que representan las siguientes expresiones:

a) `(A+B)/(C-D)`
b) `A+B+C/D`
c) `A-(B-(C-D)/(E+F))`
d) `(A+B)*((C+D)/(E+F))`
e) `(A-B)/((C*D)-(E/F))`

16.6. El recorrido preorden de un cierto árbol binario produce

```
ADFGHKLPQRWZ
```

y en recorrido *enorden* produce

```
GFHKDLAWRQPZ
```

Dibujar el árbol binario.

16.7. Escribir una función recursiva que cuente las hojas de un árbol binario.

16.8. Escribir una función que determine el número de nodos que se encuentran en el nivel n de un árbol binario.

16.9. Escribir una función que tome un árbol como entrada y devuelva el número de hijos del árbol.

16.10. Escribir una función *booleana* a la que se le pase un referencia a un árbol binario y devuelva verdadero (*true*) si el árbol es completo y falso (*false*) en caso contrario.

16.11. Se dispone de un árbol binario de elementos de tipo entero. Escribir métodos que calculen:

a) La suma de sus elementos.
b) La suma de sus elementos que son múltiplos de 3.

16.12. Diseñar una función iterativa que encuentre el número de nodos hoja en un árbol binario.

16.13. En un árbol de búsqueda cuyo campo clave es de tipo entero, escribir una función que devuelva el número de nodos cuya clave se encuentra en el rango [*x1*, *x2*].

16.14. Diseñar una función que visite los nodos del árbol por niveles; primero el nivel 0, después los nodos del nivel 1, nivel 2 y así hasta el último nivel.

PROBLEMAS

16.1. Se dispone de un archivo de texto en el que cada línea contiene la siguiente información

Columnas	1-20	Nombre
	21-31	Número de la Seguridad Social
	32-78	Dirección

Escribir un programa que lea cada registro de datos y lo inserte en un árbol, de modo que cuando el árbol se recorra en *inorden*, los números de la seguridad social se ordenen en orden ascendente.

16.2. Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferentes contenidas en el texto, así como su frecuencia de aparición. Hacer uso de la estructura árbol binario de búsqueda, cada nodo del árbol que tenga una palabra y su frecuencia.

16.3. Escribir un programa que procese un árbol binario cuyos nodos contengan caracteres y a partir del siguiente menú de opciones:

`I` (seguido de un carácter)	: Insertar un carácter
`B` (seguido de un carácter)	: Buscar un carácter
`RE`	: Recorrido en orden
`RP`	: Recorrido en preorden
`RT`	: Recorrido postorden
`SA`	: Salir

16.4. Escribir una función boleana `identicos()` que permita decir si dos árboles binarios son iguales.

16.5. Construir una función en la clase `ArbolBinarioBusqueda` que encuentre el nodo máximo.

16.6. Construir una función recursiva para escribir todos los nodos de un árbol binario de búsqueda cuyo campo clave sea mayor que un valor dado (el campo clave es de tipo entero).

16.7. Escribir una función que determine la altura de un nodo. Escribir un programa que cree un árbol binario con números generados aleatoriamente y muestre por pantalla:

- La altura de cada nodo del árbol.
- La diferencia de altura entre rama izquierda y derecha de cada nodo.

16.8. Diseñar funciones no recursivas que listen los nodos de un árbol en inorden, preorden y postorden.

16.9. Dados dos árboles binarios `A` y `B` se dice que son *parecidos* si el árbol `A` puede ser transformado en el árbol B intercambiando los hijos izquierdo y derecho (de alguno de sus nodos). Escribir una función que determine dos árboles son *parecidos.*

16.10. Dado un árbol binario de búsqueda construir su árbol espejo. Árbol espejo es el que se construye a partir de uno dado, convirtiendo el subárbol izquierdo en subárbol derecho y viceversa.

16.11. Un árbol binario de búsqueda puede implementarse con un array. La representación no enlazada correspondiente consiste en que para cualquier nodo del árbol almacenado en la posición *i* del array, su hijo izquierdo se encuentra en la posición *2*i* y su hijo derecho en la posición *2*i + 1*. Diseñar a partir de esta representación las funciones con las operaciones correspondientes para gestionar interactivamente un árbol de números enteros.

16.12. Dado un árbol binario de búsqueda diseñe una función que liste los nodos del árbol ordenados descendentemente.

CAPÍTULO 17

Árboles de búsqueda equilibrados. Árboles B

Objetivos

Con el estudio de este capítulo usted podrá:

- Conocer la eficiencia de un árbol de búsqueda.
- Construir un árbol binario equilibrado conociendo el número de claves.
- Construir un árbol binario de búsqueda equilibrado.
- Describir los diversos tipos de movimientos que se hacen cuando se desequilibra un árbol.
- Diseñar y declarar la clase ArbolEquilibrado.
- Conocer las características de los árboles B.
- Utilizar la estructura de árbol B para organizar búsquedas eficientes en bases de datos.
- Implementar la operación de búsqueda de una clave en un árbol B.
- Conocer la estrategia que sigue el proceso de inserción de una clave en un árbol B.
- Implementar las operaciones del TAD árbol B en C++.

Contenido

Conceptos clave

- Altura de un árbol.
- Árbol de búsqueda.
- Árboles B.
- Camino de búsqueda.
- Complejidad logarítmica.
- Equilibrio.
- Factor de equilibrio.
- Hoja.
- Rotaciones.

INTRODUCCIÓN

En el Capítulo 16 se introdujo el concepto de árbol binario. Se utiliza un árbol binario de búsqueda para almacenar datos organizados jerárquicamente. Sin embargo, en muchas ocasiones, las inserciones y eliminaciones de elementos en el árbol no ocurren en un orden predecible; es decir, los datos no están organizados jerárquicamente.

En este capítulo se estudian tipos de árboles adicionales: los árboles equilibrados o árboles AVL, como también se les conoce, y los árbolesB.

El concepto de árbol equilibrado así como los algoritmos de manipulación son el motivo central de este capítulo. Los métodos que describen este tipo de árboles fueron descritos en 1962 por los matemáticos rusos G. M. Adelson - Velskii y E. M. Landis.

Los árboles B se utilizan para la creación de bases de datos. Así, una forma de implementar los índices de una base de datos relacional es a través de un árbol B.

Otra aplicación dada a los árboles B es la gestión del sistema de archivos de los sistemas operativos, con el fin de aumentar la eficacia en la búsqueda de archivos por los subdirectorios.

También se conocen aplicaciones de los árboles B en sistemas de comprensión de datos. Bastantes algoritmos de comprensión utilizan árboles B para la búsqueda por claves de datos comprimidos.

17.1. EFICIENCIA DE LA BÚSQUEDA EN UN ÁRBOL ORDENADO

La eficiencia de una búsqueda en un árbol binario ordenado varía entre `O(n)` y `O(log(n)),` dependiendo de la estructura que presente el árbol.

Si los elementos son añadidos en el árbol mediante el algoritmos de inserción expuesto en el capítulo anterior, la estructura resultante del árbol dependerá del orden en que sean añadidos. Así, si todos los elementos se insertan en orden creciente o decreciente, el árbol va a tener todas la ramas izquierda o derecha, respectivamente, vacías. Entonces, la búsqueda en dicho árbol será totalmente secuencial.

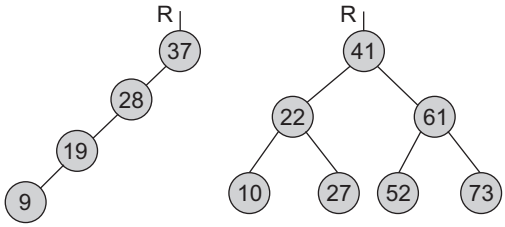


Figura 17.1. Árbol degenerado y equilibrado de búsqueda.

Sin embargo, si la mitad de los elementos insertados después de otro con clave `K` tienen claves menores de `K` y la otra mitad claves mayores de `K,` se obtiene un árbol equilibrado (también llamado balanceado), en el cual las comparaciones para obtener un elemento son como máximo $Log_2(n)$, para un árbol de n nodos.

En los árboles de búsqueda el número promedio de comparaciones que debe de realizarse para las operaciones de inserción, eliminación y búsqueda varía entre $Log_2(n)$, para el mejor de los casos, y `n` para el *peor de los casos*. Para optimizar los tiempos de búsqueda en los árboles ordenados surgen los árboles casi equilibrados, en los que la complejidad de la búsqueda es logarítmica, `O(Log (n))`.

17.2. ÁRBOL BINARIO EQUILIBRADO, ÁRBOLES AVL

Un árbol totalmente equilibrado se caracteriza por que la altura de la rama izquierda es igual que la altura de la rama derecha para cada uno de los nodos del árbol. Es un árbol ideal, no siempre se puede conseguir que el árbol esté totalmente balanceado.

La estructura de datos de árbol equilibrado que se utiliza es la árbol **AVL**. El nombre es en honor de Adelson-Velskii-Landis que fueron los primeros científicos en estudiar las propiedades de esta estructura de datos. Son árboles ordenados o de búsqueda que, además, cumplen la condición de balanceo para cada uno de los nodos.

Definición

Un árbol equilibrado o *árbol AVL* es un árbol binario de búsqueda en el que las alturas de los subárboles izquierdo y derecho de cualquier nodo difieren como máximo en 1.

La Figura 17.2 muestra dos árboles de búsqueda, el de la izquierda está equilibrado. El de la derecha es el resultado de insertar la clave 2 en el anterior, según el algoritmo de inserción en árboles de búsqueda. La inserción provoca que se *viole* la condición de equilibrio en el nodo raíz del árbol.

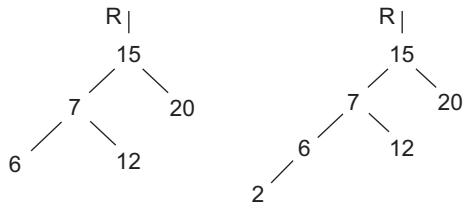


Figura 17.2. Dos árboles de búsqueda, el de la izquierda equilibr ado, el otro no.

La condición de equilibrio de cada nodo implica una restricción en las alturas de los subárboles de un árbol AVL. Si v_r es la raíz de cualquier subárbol de un árbol equilibrado, y `h` la altura de la rama izquierda entonces la altura de la rama derecha puede tomar los valores: `h-1`, `h`, `h+1`. Esto aconseja asociar a cada nodo el parámetro denominado *factor de equilibrio o balance de un nodo*. Se define como la altura del subárbol derecho menos la altura del subárbol izquierdo correspondiente. El factor de equilibrio de cada nodo en un árbol equilibrado puede tomar los valores: `1`, `-1` o `0`. La Figura 17.3 muestra un árbol balanceado con el factor de equilibrio de cada nodo.

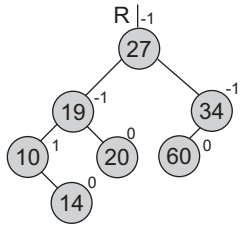


Figura 17.3. Árbol equilibrado con el factor de equilibrio de cada nodo.

A tener en cuenta

La altura o profundidad de un árbol binario es el nivel máximo de sus hojas mas uno. La altura de un árbol nulo se considera cero.

17.2.1. Altura de un árbol equilibrado, árbol AVL

No resulta fácil determinar la altura promedio de un árbol AVL, por lo que se determina la altura en el *peor de los casos*, es decir, la altura máxima que puede tener un árbol equilibrado con un número de nodos n. La altura es un parámetro importante ya que coincide con el número de iteraciones que se realizan para *bajar* desde el nodo raíz al nivel más profundo de las hojas. La eficiencia de los algoritmos de búsqueda, inserción y borrado depende de la altura del árbol AVL.

Para calcular la altura máxima de un árbol AVL de n nodos se parte del siguiente razonamiento: *¿cuál es el número mínimo de nodos que puede tener un árbol binario para que se conside reequilibrado con una altura h?* Si ese árbol es A_h, tendrá dos subárboles izquierdo y derecho respectivamente, A_i y A_d cuya altura difiera en 1, supongamos que tienen de altura h-1 y h-2 respectivamente. Al considerar que A_h es el árbol de menor número de nodos de altura h, entonces A_i y A_d también son árboles AVL de menor número de nodos, pero de altura h-1 y h-2 y son designados como A_{h-1} y A_{h-2}. Este razonamiento se sigue extendiendo a cada subárbol y se obtienes árboles equilibrados como los de la Figura 17.4.

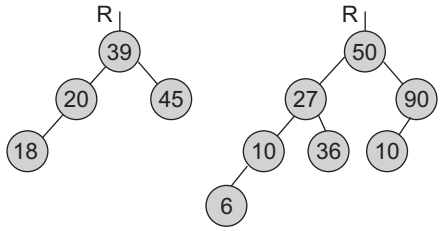


Figura 17.4. Árboles de Fibonacci.

La construcción de árboles binarios equilibrados siguiendo esta estrategia puede representarse matemáticamente:

$$A_h = A_{h-1} + A_{h-2}$$

La expresión matemática expuesta tiene una gran similitud con la ley de recurrencia que permite encontrar números de Fibonacci, $a_n = a_{n-1} + a_{n-2}$. Por esa razón, a los árboles equilibrados construidos con esta ley de formación se les conoce como *árboles de Fibonacci*.

El objetivo que se persigue es encontrar el número de nodos, n, que hace la altura máxima. El número de nodos de un árbol es la suma de los nodos de su rama izquierda, rama derecha más uno (la raíz). Si es ese número es N_h se puede escribir:

$$N_h = N_{h-1} + N_{h-2} + 1$$

donde $N_0 = 1$, $N_1 = 2$, $N_2 = 4$, $N_3 = 7$, y así sucesivamente. Se observa que los números $N_h + 1$ cumplen la definición de los números de Fibonacci. El estudio matemático de la función generadora de los números de Fibonacci permite encontrar esta relación:

$$N_h + 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^h$$

Tomando logaritmos se encuentra la altura h en función del número de nodos, N_h:

$h \approx 1.44 \log (N_h)$

Como conclusión, el árbol equilibrado de n nodos menos denso tiene como altura 1.44log n. Como n es el número de nodos en el *peor de los casos* de árbol AVL de altura h, se puede afirmar que la complejidad de una búsqueda es O(Log n).

A tener en cuenta

La altura de un árbol binario perfectamente equilibrado de n nodos es log n. Las operaciones que se aplican a los árboles AVL no requieren más del 44 por cien de tiempo (en el caso más desfavorable) que si se aplican a un árbol perfectamente equilibrado.

EJERCICIO 17.1. Se tienen n claves que se van a organizar jerárquicamente formando un árbol equilibrado. Escribir un programa para formar el árbol AVL siguiendo la estrategia de los árboles de Fibonacci descrita en la sección anterior.

La formación de un árbol balanceado de n nodos se parece mucho a la secuencia de los números de Fibonacci:

```
a(n) = a(n-2) + a(n-1)
```

Un *árbol de Fibonacci* (árbol equilibrado) puede definirse:

1. Un árbol vacío es el árbol de Fibonacci de altura 0.
2. Un nodo único es un árbol de Fibonacci de altura 1.
3. Si A_{h-1} y A_{h-2} son árboles de Fibonacci de alturas h-1 y h-2, entonces

   $A_h = < A_{h-1}, x, A_{h-2} >$ es árbol de Fibonacci de altura h.

El número de nodos, A_h , viene dado por la sencilla relación recurrente:

$N_0 = 0$
$N_1 = 1$
$N_h = N_{h-1} + 1 + N_{h-2}$

Para conseguir un árbol AVL con un número dado, n, de nodos de mínima altura hay que distribuir equitativamente los nodos a la izquierda y a la derecha de un nodo dado. En definitiva, es seguir la relación de recurrencia anterior, que expresada recursivamente:

1. Crear nodo raíz.
2. Generar el subárbol izquierdo con $n_i = n/2$ nodos del nodo raíz utilizando la misma estrategia.

3. Generar el subárbol derecho con $n_d = n - n_i - 1$ nodos del nodo raíz utilizando la misma estrategia.

En este ejercicio, el árbol no va a ser de búsqueda. Simplemente, un árbol binario de números enteros que son leídos del teclado pero que es de mínima altura. El árbol será de búsqueda, si los números que se leen están ordenados crecientemente.

El método público `ArbolFibonacci()` de la clase `Arbolbinario` realiza una llamada al método privado `arbolFibonacci()` de la clase `Arbolbinario` que retorna un puntero a la clase `Nodo`.

```
void ArbolBinario::ArbolFibonacci(int n)
{
     raiz = arbolFibonacci(n);
}

Nodo* ArbolBinario::arbolFibonacci(int n)
{
       int nodosIz, nodosDr;
       int clave;
       Nodo *nuevoRaiz;

       if (n == 0)
         return NULL;
       else
       {
         nodosIz = n / 2;
         nodosDr = n - nodosIz - 1;
         // nodo raíz con árbol izquierdo y derecho de Fibonacci
         cin >> clave;
         nuevoRaiz = new Nodo(arbolFibonacci(nodosIz), clave,
                              arbolFibonacci(nodosDr));
       }
       return nuevoRaiz;
}
```

El método público `dibujarArbol()` de la clase `ArbolBinario`, se encarga de representar en el dispositivo estándar de salida la información almacenada en cada uno de los nodos del árbol. Usa un parámetro `h` para formatear la salida.

```
void ArbolBinario::dibujarArbol(Nodo *r, int h)
 {
      /*
        escribe las claves del arbol de fibonacci; h establece
        una separación entre nodos
      */
       int i;
       if (r != NULL)
       {
           dibujarArbol(r->subarbolIzdo(), h + 1);
           for (i = 1; i <= h; i++)
                cout << " ";
           cout <<r->valorNodo() << endl;
          dibujarArbol(r->subarbolDcho(), h + 1);
       }
}
```

El programa principal realiza las llamadas correspondientes.

```
int main()
{
   ArbolBinario arbolFib;
    int n;
    do {
       cout << "Número de nodos del árbol: ";
       cin >> n;
   } while (n <= 0);
   arbolFib.ArbolFibonacci(n);
   cout << "Árbol de Fibonacci de mínima altura:\n";
   arbolFib.dibujarArbol(arbolFib.Oraiz(), 1);
   return 0;
}
```

17.3. INSERCIÓN EN ÁRBOLES DE BÚSQUEDA EQUILIBRADOS: ROTACIONES

Los árboles equilibrados, árboles AVL, son árboles de búsqueda y, por consiguiente, para añadir un elemento se ha de seguir el mismo proceso que en los árboles de búsqueda. Se compara la nueva clave con la clave del raíz, continúa por la rama izquierda o derecha según sea menor o mayor (describe el *camino de búsqueda*), termina insertándose como nodo hoja. Esta operación, como ha quedado demostrado al determinar la altura en el peor de los casos, tiene una complejidad logarítmica. Sin embargo, la nueva inserción puede hacer que aumente la altura de una rama, de manera que cambie el factor de equilibrio del nodo raíz de dicha rama. Este hecho hace necesario que el algoritmo de inserción, *regrese* por el *camino de búsqueda* actualizando el factor de equilibrio de los nodos. La Figura 17.5 muestra un árbol equilibrado y el mismo árbol justo después de la inserción de una nueva clave que provoca que *rompa* la condición de balanceo.

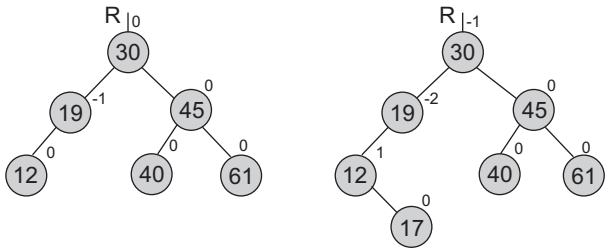


Figura 17.5. Árbol equilibrado; el mismo después de insertar la clave 17.

A recordar

Una inserción de una nueva clave, o un borrado, puede destruir el criterio de equilibrio de varios nodos del árbol. Se debe recuperar la condición de equilibrio del árbol antes de dar por finalizada la operación para que el árbol siga siendo equilibrado.

La estructura del nodo en un árbol equilibrado es una *extensión* de la declarada para un árbol binario. Para determinar si el árbol está equilibrado debe de manejarse información

relativa al balanceo o factor de equilibrio de cada nodo. Por esta razón se añade al nodo un campo más: el factor de equilibrio (`fe`). Este atributo puede tomar los valores: `-1`, `0`, `+1`.

La clase `NodoAvl` es similar a la clase `Nodo` de los árboles binarios de búsqueda. Solamente se le añade un nuevo atributo entero `fe`, a los que ya disponía la clase `Nodo`. Además, se añaden las funciones miembro encargadas de obtener el valor del nuevo atributo coma la de modificar el valor del nuevo atributo, así como los correspondientes métotos

```
class NodoAvl
{
protected:
   Tipoelemento dato;
    NodoAvl *izdo;
    NodoAvl *dcho;
    int fe;
public:
    NodoAvl(Tipoelemento valor)
    {
      dato = valor;
      izdo = dcho = NULL;
      fe = 0;
    }
    NodoAvl(Tipoelemento valor, int vfe)
    {
      dato = valor;
      izdo = dcho = NULL;
      fe = vfe;
    }
     NodoAvl(NodoAvl* ramaIzdo,Tipoelemento valor, NodoAvl* ramaDcho)
     {
       dato = valor;
       izdo = ramaIzdo;
       dcho = ramaDcho;
       fe = 0;
    }
     NodoAvl(NodoAvl* ramaIzdo, int vfe, Tipoelemento valor,
             NodoAvl* ramaDcho)
     {
       dato = valor;
       izdo = ramaIzdo;
       dcho = ramaDcho;
       fe = vfe;
    }
    // operaciones de acceso
    Tipoelemento valorNodo(){ return dato; }
    NodoAvl* subarbolIzdo(){ return izdo; }
    NodoAvl* subarbolDcho(){ return dcho; }
    void nuevoValor(Tipoelemento d){ dato = d; }
    void ramaIzdo(NodoAvl* n){ izdo = n; }
    void ramaDcho(NodoAvl* n){ dcho = n; }
    void visitar(){ cout << dato << endl;}
    void Pfe(int vfe) { fe = vfe;}
    int Ofe(){ return fe;}
};
```

Las operaciones a realizar con un árbol de búsqueda equilibrado son las mismas que con un árbol de búsqueda. Debido a los cambios del nodo, se declara la clase `ArbolAvl` sin relacionarla con `ArbolBinarioBusqueda`.

```
class ArbolAvl
{
  NodoAvl* raiz;
  ArbolAvl()
  {
    raiz = NULL;
  }
  ArbolAvl(NodoAvl * r)
  {
    raiz = r;
  }
  NodoAvl* Oraiz ()
  {
    return raiz;
  }
  void Praiz( NodoAvl *r)
  {
    raiz = r;
  }
  ...
};
```

17.3.1. Proceso de inserción de un nuevo nodo

Inicialmente, se aplica el algoritmo de inserción en un árbol de búsqueda, éste sigue el *camino de búsqueda* hasta llegar al fondo del árbol y se enlaza como nodo *hoja* y con *factor de equilibrio* `0`. Pero el proceso no puede terminar, es necesario recorrer el *camino de búsqueda* en sentido contrario, hacia la raíz, para actualizar el campo adicional *factor de equilibrio*. Después de una inserción sólo los nodos que se encuentran en el camino de búsqueda pueden haber cambiado el factor de equilibrio.

La actualización del *factor de equilibrio* (`fe`) puede hacer que éste *mejore*. Esto ocurre cuando un nodo está descompensado a la izquierda y se inserta el nuevo nodo en la rama izquierda, al crecer en altura dicha rama el `fe` se hace `0`, se ha *mejorado* el equilibrio. La Figura 17.6 muestra el árbol *a*) en el que el nodo `90` tiene `fe = 1`; en el árbol *b*), después de insertar el nodo con clave `60`, el nodo `90` tiene `fe = 0`.

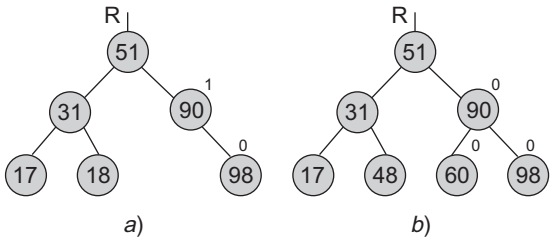


Figura 17.6. Mejora en la condición de equilibrio al insertar un nuevo nodo con clave 60.

La actualización del `fe` de un nodo del *camino de búsqueda* que, originalmente, tiene las ramas izquierda y derecha de la misma altura (`hRi = hRd`), no va a causar romper el criterio de equilibrio.

Al actualizar el nodo cuyas ramas izquierda y derecha del árbol tienen altura diferente, `|hRi-hRd|=1`, si se inserta el nodo en la rama más alta rompe el criterio de equilibrio del árbol, la diferencia de altura pasa a ser `2` y es necesario reestructurarlo.

Hay cuatro casos que se deben tener en cuenta al reestructurar un nodo `A`, según dónde se haya hecho la inserción:

1. Inserción en el subárbol izquierdo de la rama izquierda de `A`.
2. Inserción en el subárbol derecho de la rama izquierda de `A`.
3. Inserción en el subárbol derecho de la rama derecha de `A`.
4. Inserción en el subárbol izquierdo de la rama derecha de `A`.

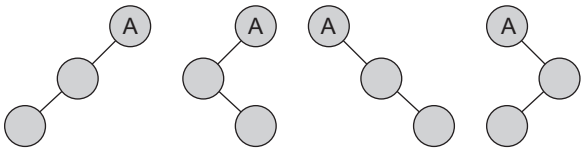


Figura 17.7. Cuatro tipos de reestructuraciones del equilibrio de un nodo.

El primer y tercer caso (*izquierda-izquierda, derecha-derecha*) se resuelven con una *rotación simple*. El segundo y cuarto caso (*izquierda-derecha, derecha-izquierda*) se resuelven con una *rotación doble*.

La rotación simple implica a dos nodos, el nodo `A` (nodo con `|fe| = 2`) y el descendiente izquierdo o derecho según el caso. En la rotación doble están implicados tres nodos, el nodo `A`, nodo descendiente izquierdo y el descendiente derecho de éste; o bien el caso simétrico, nodo `A`, descendiente derecho y el descendiente izquierdo de éste.

A recordar

Una reestructuración de los nodos implicados en la violación de *criterio de equilibrio*, ya sea una rotación simple o doble, hace que se recupere el equilibrio en todo el árbol, no siendo necesario seguir analizando los nodos del *camino de búsqueda*.

El proceso de *regresar* por el *camino de búsqueda* termina cuando se llega a la raíz del árbol, o cuando se realiza la reestructuración en un nodo del mismo. Una vez realizada una reestructuración no es necesario determinar el *factor de equilibrio* de los restantes nodos, debido a que dicho factor queda como el que tenía antes de la inserción, ya que la reestructuración hace que no aumente la altura.

17.3.2. Rotación simple

La rotación simple resuelve la *violación del equilibrio* de un nodo *izquierda-izquierda*, simétrica a la *derecha-derecha*. El árbol de la Figura 17.8*a*) tiene el nodo `C` con factor de equi-

librio `-1`; el árbol de la Figura 17.8*b*) es el resultado de insertar el nodo `A`. Resulta que ha crecido la altura de la rama izquierda, es un desequilibrio *izquierda-izquierda* que se resuelve con una rotación simple, rotación `II`. La Figura 17.9 es el árbol resultante de la rotación, el nodo `B` se ha convertido en la raíz, el nodo `C` su rama derecha y el nodo `A` continúa como rama izquierda. Con estos movimientos el árbol sigue siendo de búsqueda y se equilibra.

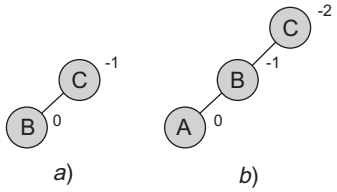


Figura 17.8. Árbol binario AVL y árbol después de inser tar nueva clave por la izquierda.

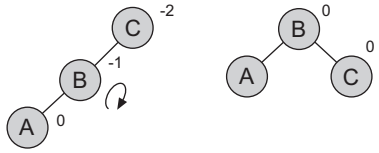


Figura 17.9. Árbol binario después de rotación simple `II`.

La Figura 17.10 muestra el otro caso de violación de la condición de equilibrio que se resuelve con una rotación simple. Inicialmente, el nodo `A` tiene como factor de equilibrio `+1`, al insertar el nodo `C` el factor de equilibrio de `A` pasa a ser `+2`, se ha insertado por la derecha y, por consiguiente, ha crecido la altura de la rama derecha. La Figura 17.11 muestra la resolución de este desequilibrio, una rotación simple que se puede denominar rotación `DD`. En el árbol resultante de la rotación, el nodo `B` se ha convertido en la raíz, el nodo `A` es su rama izquierda y el nodo `C` continúa como rama derecha. Con estos movimientos el árbol sigue siendo de búsqueda y queda equilibrado.

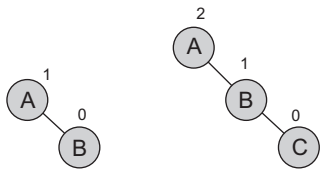


Figura 17.10. Árbol binario AVL y árbol después de inser tar nueva clave por la derecha.

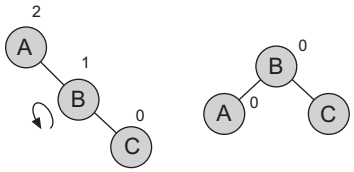


Figura 17.11. Árbol binario después de rotación simple DD.

17.3.3. Movimiento de enlaces en la rotación simple

Los cambios descritos en la rotación simple afectan a dos nodos, el tercero no se modifica, es necesario sólo una rotación. Para la rotación simple a la *izquierda*, rotación `II`, los ajustes necesarios de los enlaces, suponiendo `n` la referencia al nodo problema y `n1` la referencia al nodo de su rama izquierda:

```
n->izdo = n1->dcho;
n1->dcho = n;
n = n1;
```

Una vez realizada la rotación, los factores de equilibrio de los nodos que intervienen siempre es `0`, los subárboles izquierdo y derecho tienen la misma altura. Incluso, la altura del subárbol implicado es la misma después de la inserción que antes. La Figura 17.12 muestra estos movimientos de los enlaces.

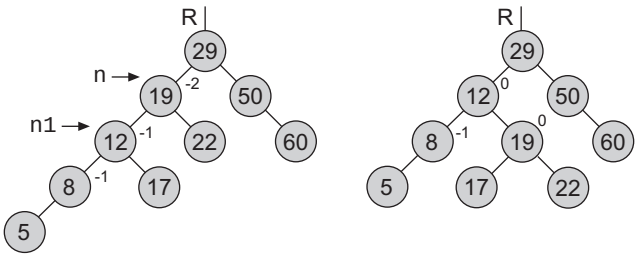


Figura 17.12. Rotación simple a izquierda en un árbol después de inser tar la clave 5.

Si la rotación simple es a *derecha*, rotación `DD`, los cambios en los enlaces del nodo `n` (con factor de equilibrio `+2`) y del nodo de su rama derecha, `n1`:

```
n->dcho = n1->izdo;
n1->izdo = n;
n = n1;
```

Realizada la rotación, los factores de equilibrio de los nodos que intervienen es 0. Se puede observar que estos ajustes son los simétricos a los realizados en la rotación `II`.

17.3.4. Rotación doble

Con la rotación simple no es posible resolver todos los casos de violación del criterio de equilibrio. El árbol de búsqueda de la Figura 17.13*a*) está desequilibrado, con factores de equilibrio `+2, -1` y `0`. La Figura 17.13*b*) aplica la rotación simple, rotación `DD`. La única solución consiste en *subir* el nodo `40` como raíz del subárbol, como rama izquierda situar al nodo `30` y como rama derecha el nodo `60`, la altura del subárbol resultante es la misma que antes de insertar. Se ha realizado una rotación doble, *derecha-izquierda*, en la que intervienen los tres nodos.

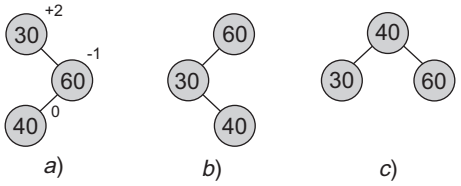


Figura 17.13. *a*) Árbol después de insertar clave 40. *b*) Rotación D *c*) Rotación doble para equilibrar.

La Figura 17.14*a*) muestra un árbol binario de búsqueda después de insertar la clave `60`. Al *volver* por el *camino de búsqueda* para actualizar los factores de equilibrio, el nodo `75` pasa a tener `fe = -1` (se ha insertado por su izquierda), el nodo `50` pasa a tener `fe = +1` y `el` nodo `80` tendrá como `fe = -2`. Es el caso simétrico al descrito en la Figura 17.13, se reestablece el equilibrio con una rotación doble, simétrica con respecto a la anterior como se muestra en la Figura 17.14*b*).

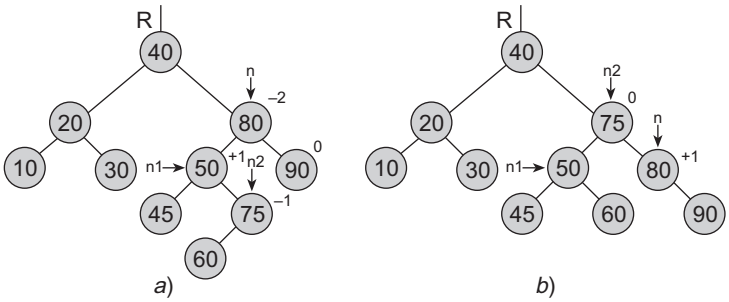


Figura 17.14. *a*) Árbol después de insertar clave 60. *b*) Rotación doble, izquierda derecha.

Recuerde

La rotación doble resuelve dos casos simétricos, se pueden denominar rotación `ID` y rotación `DI`. En la rotación doble hay que mover los enlaces de tres nodos , el nodo *padre*, el *descendiente* y el *descendiente del descendiente* por la rama contraria.

17.3.5. Movimiento de enlaces en la rotación doble

Los cambios descritos en la rotación doble afectan a tres nodos, el nodo problema `n`, el descendiente por la rama desequilibrada `n1`, y el descendiente de `n1` (por la izquierda o la derecha, según el tipo de rotación doble) apuntado por `n2`. En los dos casos simétricos de rotación doble, rotación *izquierda-derecha* (rotación `ID`) y rotación *derecha-izquierda* (rotación `DI`), el nodo `n2` pasa a ser la raíz del nuevo subárbol.

Los movimientos de los enlace para realizar la rotación `ID`:

```
n1->dcho = n2->izdo;
n2->izdo = n1;
n->izdo = n2->dcho;
n2->dcho = n;
n = n2;
```

Los factores de equilibrio de los nodos implicados en la rotación `ID` depende del factor de equilibrio, antes de la inserción, del nodo apuntado por `n2`, según esta tabla:

Si	`n2->fe = -1`	`n2->fe = 0`	`n2->fe = 1`
	`n->fe = 1`	0	0
	`n1->fe = 0`	0	-1
	`n2->fe = 0`	0	0

Los movimientos de los enlaces para realizar la rotación `DI` (observar la simetría en los movimientos de los punteros):

```
n1->izdo = n2->dcho;
n2->dcho = n1;
n->dcho = n2->izdo;
n2->izdo = n;
n = n2 ;
```

Los factores de equilibrio de los nodos implicados en la rotación `DI` también dependen del factor de equilibrio previo del nodo `n2`, según la tabla:

Si	`n2->fe = -1`	`n2->fe = 0`	`n2->fe = 1`
	`n->fe = 0`	0	-1
	`n1->fe = 1`	0	0
	`n2->fe = 0`	0	0

A recordar

La complejidad del algoritmo de inserción de una clave en un árbol de búsqueda AVL es la suma de la complejidad para bajar al nivel de las hojas (O(log n)) más la complejidad en el peor de los casos de la vuelta por el camino de búsqueda, para actualizar el factor de equilibrio de los nodos que es O(log n), más la complejidad de los movimientos de los enlaces en la rotación, que tiene complejidad constante. En definitiva, la complejidad de la inserción es O(log n), complejidad logarítmica.

17.4. IMPLEMENTACIÓN DE LA INSERCIÓN CON BALANCEO Y ROTACIONES

La realización de la fase de inserción es igual que la escrita para los árboles de búsqueda. Ahora se añade la fase de actualización de los factores de equilibrio; una vez insertado, se activa un *flag* para indicar que ha crecido en altura, de tal forma que al *regresar* por el *camino de búsqueda* calcule los nuevos factores de equilibrio de los nodos que forman el camino. Cuando la inserción se ha realizado por la rama izquierda del nodo, la altura crece por la izquierda y, por tanto, disminuye en `1` el factor de equilibrio; si se hace por la rama derecha, el factor de equilibrio aumenta en `1`. El proceso termina si la altura del subárbol no aumenta. También termina si se produce un desequilibrio, ya que cualquier rotación tiene la propiedad de que la altura del subárbol resultante es la misma que antes de la inserción.

A continuación, se escriben los métodos privados (miembros privados de la clase `ArbolAvl`) que implementan los cuatro tipos de rotaciones y de la operación de inserción. Todos devuelven la referencia al nodo raíz del subárbol implicado en la rotación.

Rotaciones

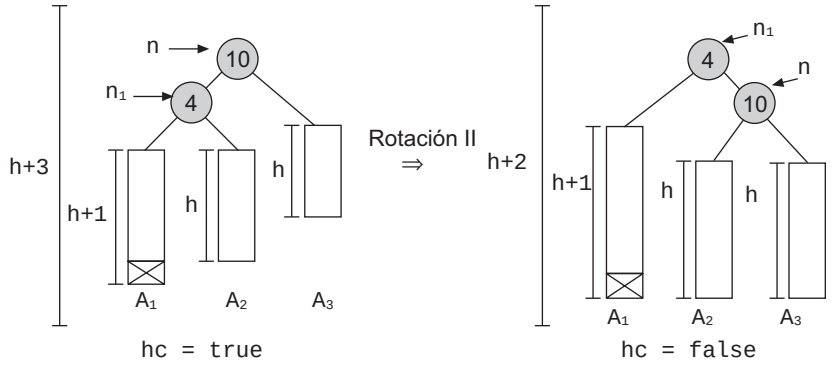


Figura 17.15.[1] Rotación II. Esquema gráfico. Árbol deja de aumentar en altura.

```
NodoAvl* ArbolAvl::rotacionII(NodoAvl* n, NodoAvl* n1)
{     //Figura 17.5
    n->ramaIzdo(n1->subarbolDcho());
    n1->ramaDcho(n);
      // actualización de los factores de equilibrio
    if (n1->Ofe() == -1) // la condición es true en la inserción
    {
      n->Pfe(0);
      n1->Pfe(0);
    }
```

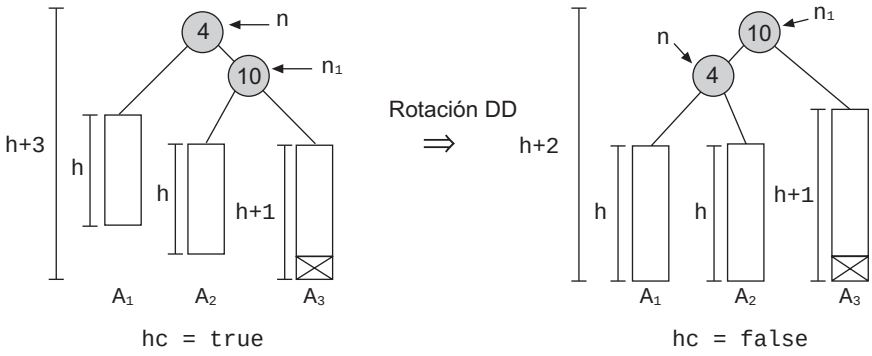


Figura 17.16. Rotación DD. Esquema gráfico. Árbol deja de aumentar en altura.

```
    else
    {
      n->Pfe(-1);
```

[1,2] En [KRUSE 94] el lector podrá encontrar una excelente referencia para profundizar en temas avanzados de árboles AVL, B y *tries* (árboles de búsqueda lexicográficos. Los códigos de las funciones, procedimientos y programas están escritos) en lenguaje Pascal, pero sus conceptos teóricos y prácticos sirven para la implantación en cualquier lenguaje.

```
      n1->Pfe(1);
    }
    return n1;
}

NodoAvl* ArbolAvl::rotacionDD(NodoAvl* n, NodoAvl* n1)
{   //Figura 17.16
    n->ramaDcho(n1->subarbolIzdo());
    n1->ramaIzdo(n);
                // actualización de los factores de equilibrio
    if (n1->Ofe() == +1) // la condición es true en la inserción
    {
      n->Pfe(0);
      n1->Pfe(0);
    }
    else
    {
      n->Pfe(+1);
      n1->Pfe(-1);
    }
    return n1;
}
```

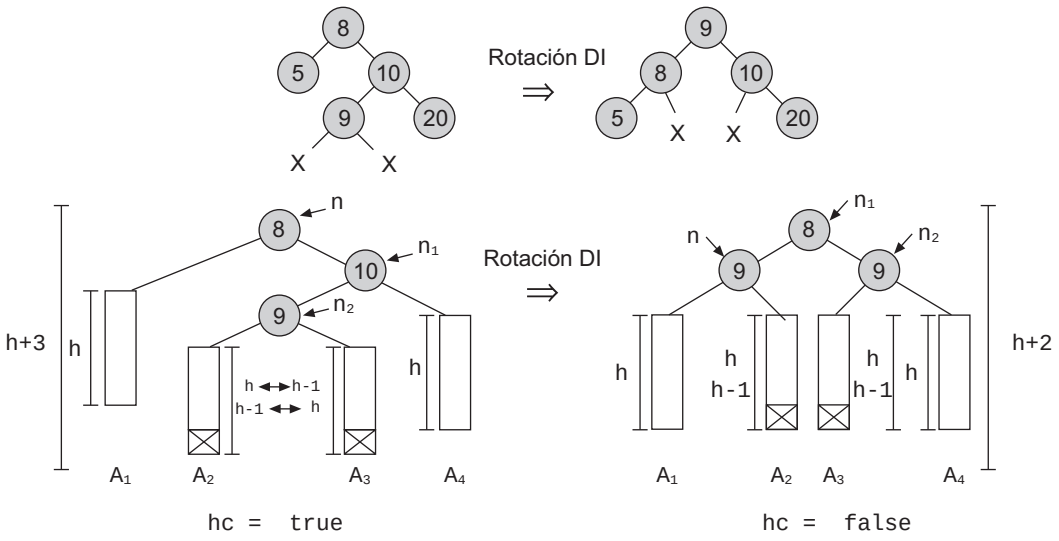


Figura 17.17. Rotación DI. Esquema gráfico. Árbol deja de aumentar en altura[3,4].

```
NodoAvl* ArbolAvl::rotacionDI(NodoAvl* n, NodoAvl* n1)
{//Figura 17.17
   NodoAvl* n2;

    n2 = n1->subarbolIzdo();
    n->ramaDcho(n2->subarbolIzdo());
```

[3] Si A_2 tiene altura h, A_3 tiene altura h–1; si A_2 tiene altura *h–1*, A_3 tiene altura h. No puede darse en las inserciones que A_2 tenga altura h y A_3 tenga altura *h*, ya que en este caso no hay desequilibrio en el nodo n_2.

[4] Ibid, [KRUSE 84].

```
    n2->ramaIzdo(n);
    n1->ramaIzdo(n2->subarbolDcho());
    n2->ramaDcho(n1);
            // actualización de los factores de equilibrio
    if (n2->Ofe() == +1)
      n->Pfe(-1);
    else
      n->Pfe(0);

    if (n2->Ofe() == -1)
      n1->Pfe(+1);
    else
      n1->Pfe(0);
    n2->Pfe(0);
    return n2;
}
```

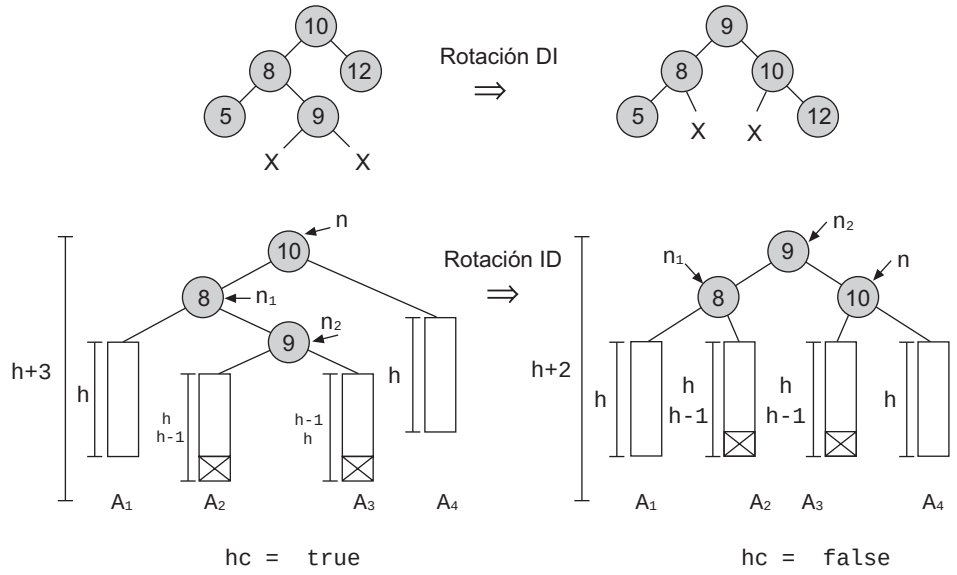


Figura 17.18. Rotación ID. Esquema gráfico. Árbol deja de aumentar en altura[5].

```
NodoAvl* ArbolAvl::rotacionID(NodoAvl* n, NodoAvl* n1)
{     //Figura 17.18
   NodoAvl* n2;
   n2 = n1->subarbolDcho();
   n->ramaIzdo(n2->subarbolDcho());
   n2->ramaDcho(n);
   n1->ramaDcho(n2->subarbolIzdo());
   n2->ramaIzdo(n1);
   // actualización de los factores de equilibrio
   if (n2->Ofe() == +1)
     n1->Pfe(-1);
```

[5] Ibid, [KRUSE 94].

```
    else
      n1->Pfe(0);
    if (n2->Ofe() == -1)
      n->Pfe(1);
    else
      n->Pfe(0);
    n2->Pfe(0);
    return n2;
}
```

Inserción con balanceo

El método público `insertarAvl()` es el interfaz de la operación, llamada a la función interna privada, recursiva, que realiza la operación y devuelve la raíz del nuevo árbol.

```
NodoAvl* ArbolAvl::insertarAvl(NodoAvl* raiz, Tipoelemento dt, bool &hc)
{
    NodoAvl *n1;

    if (raiz == NULL)
    {
      raiz = new NodoAvl(dt);
      hc = true;
    }
     else if (dt < raiz->valorNodo())
    {
       NodoAvl *iz;
       iz = insertarAvl(raiz->subarbolIzdo(), dt, hc);
       raiz->ramaIzdo(iz);
       // regreso por los nodos del camino de búsqueda
       if (hc)            // siempre se comprueba si creció en altura
       {
         // decrementa el fe por aumentar la altura de rama izquierda
         switch (raiz->Ofe())
         {
             case 1:         // tenía +1 y creció su izquierda
                raiz->Pfe(0);
                hc = false;  // árbol deja de crecer
                break;
             case 0:         // tenía 0 y creció su izquierda
                raiz->Pfe(-1);  // árbol sigue creciendo
                break;
             case -1:        // aplicar rotación a la izquierda
                n1 = raiz->subarbolIzdo();
                if (n1->Ofe() == -1)
                  raiz = rotacionII(raiz, n1);
                else
                  raiz = rotacionID(raiz, n1);
                hc = false;  // árbol deja de crecer en ambas rotaciones
         }
       }
    }
    else if (dt > raiz->valorNodo())
    {
      NodoAvl *dr;
```

```
        dr = insertarAvl(raiz->subarbolDcho(), dt, hc);
        raiz->ramaDcho(dr);
          // regreso por los nodos del camino de búsqueda
        if (hc)                // siempre se comprueba si creció en altura
        {
         // incrementa el fe por aumentar la altura de rama izquierda
         switch (raiz->Ofe())
         {
           case 1:             // aplicar rotación a la derecha
             n1 = raiz->subarbolDcho();
             if (n1->Ofe() == +1)
               raiz = rotacionDD(raiz, n1);
             else
               raiz = rotacionDI(raiz,n1);
             hc = false; // árbol deja de crecer en ambas rotaciones
             break;
           case                0: // tenía 0 y creció su derecha
             raiz->Pfe(+1);    // árbol sigue creciendo
             break;
           case -1:            // tenía -1 y creció su derecha
             raiz->Pfe(0);
             hc = false;       // árbol deja de crecer
         }
        }
      }
      else
        throw "No puede haber claves repetidas " ;
      return raiz;
}
```

EJERCICIO 17.2. Se quiere formar un árbol binario de búsqueda equilibrado de altura 5. El campo dato de cada nodo que sea una referencia a un objeto que guarda un número entero, que será la clave de búsqueda. Una vez formado el árbol, mostrar las claves en orden creciente y el número de nodos de que consta el árbol.

La formación del árbol equilibrado se puede hacer con repetidas llamadas a la función miembro `insertarAvl()`, de la clase `ArbolAvl` que realiza las llamadas, cuando es necesario, a las *rotaciones* implementadas como métodos privados de la clase. La condición para terminar la formación del árbol está expuesta en el enunciado: la altura del árbol sea igual a 5. Por ello, se escribe la función miembro publica `altura()` de la clase `ArbolAvl` para determinar dicho parámetro. También se escribe la función miembro pública `visualizar()`, que es un recorrido en inorden, para mostrar las claves en orden creciente y a la vez contar los nodos visitados. Se escribe, además, la función miembro que encuentra el número de nodos de una árbol, así como un programa principal que realiza las correspondientes llamadas

```
int mayor (int x, int y)
{  // calcula el mayor de los números que recibe como parámetro
  return (x > y ? x : y);
}
int ArbolAvl::altura(NodoAvl* r)
{ // método público que decide la altura de un árbol apuntado por r
   if (r != NULL)
     return mayor(altura(r->subarbolIzdo()),
```

```
      altura(r->subarbolDcho())) + 1;
    else
      return false;
}
int ArbolAvl::cuantos()
{ // método público que calcula el número de nodos de un árbol
  return cuantos(raiz);
}
int ArbolAvl::cuantos (NodoAvl* r)
{  // método privado que calcula el número de nodos apuntado por r.
   if (r)
   {
      int cuantosIzquierda, cuantosDerecha;
      cuantosIzquierda = cuantos(r -> subarbolIzdo());
      cuantosDerecha = cuantos(r -> subarbolDcho());
      return cuantosIzquierda + cuantosDerecha + 1;
   }
   else
      return 0;
}
void ArbolAvl::dibujarArbol(NodoAvl *r, int h)
 {
 // escribe las claves del árbol estableciendo separación entre nodos
      int i;
      if (r != NULL)
      {
        dibujarArbol(r->subarbolIzdo(), h + 1);
        for (i = 1; i <= h; i++)
        cout << " ";
        cout << r->valorNodo() << endl;
        dibujarArbol(r->subarbolDcho(), h + 1);
      }
 }

int main
{
   ArbolAvl a; const int TOPE = 999;
   int numNodos;
   randomize();
   while (a.altura(a.Oraiz()) < 5)
   {
     a.insertarAvl(random(TOPE)+1);
   }
   numNodos = a.cuantos();
   cout << "\n Número de nodos: " << numNodos << endl;
   a.dibujarArbol(a.Oraiz(), 1);
   return 0;
}
```

17.5. DEFINICIÓN DE UN ÁRBOL B

Cuando se tiene un conjunto de datos masivo, por ejemplo el conjunto `1.000.000` de clientes de un banco, los registros no pueden estar en memoria principal, y se ubican en memoria auxiliar, normalmente en disco. Los accesos a disco son *críticos*, consumen recursos y necesitan

notablemente más tiempo que las instrucciones en memoria, se necesita reducir al mínimo el número de accesos a disco. Para conseguir esto se emplean árboles de búsqueda *m-arios*, que ya no tienen dos ramas como los binarios, sino que pueden tener hasta *m* ramas o subárboles descendientes, además las claves se organizan a *la manera de los árboles de búsqueda*; el objetivo es que la altura del árbol sea lo suficientemente pequeña ya que el numero de iteraciones, y, por tanto, de acceso a disco, de la operación de búsqueda depende directamente de la altura. Un tipo particular de estos árboles son los árboles *B*, también los denominados *B*+ y *B** que proceden de pequeñas modificaciones del anterior.

Los árboles *B* son árboles *m*-arios, cada nodo tiene como máximo *m* ramas, no tienen subárboles vacíos y siempre están *perfectamente equilibrados*. Cada nodo (en árboles *B* se acostumbra a denominar *página*) es una unidad a la que se accede en bloque. Las estructura de datos que representa un árbol *B* de orden *m* tiene las siguientes características:

- Todas las páginas hoja están en el mismo nivel.
- Todos las páginas internas, menos la raíz, tienen a lo sumo *m* ramas (no vacías) y como mínimo *m/2* ramas.
- El número de claves en cada página interna es uno menos que el número de sus ramas, y estas claves dividen las de las ramas a manera de un árbol de búsqueda.
- La raíz tiene como máximo *m* ramas, puede llegar a tener hasta 2 y ninguna si el árbol consta de la raíz solamente.

Los árboles *B* más utilizados son los de orden 5, un orden mayor aumenta considerablemente la complejidad de los algoritmos de inserción y de borrado, un orden menor disminuye la eficacia de la localización de claves. Según la definición dada, el máximo número de claves de una página o nodo es 4 y el máximo de ramas o subárboles es 5.

La Figura 17.19 muestra un árbol *B* de orden 5, las claves de búsqueda son valores enteros. El árbol tiene 3 niveles, todas las páginas contienen 2, 3 o 4 claves. La raíz es la excepción, sólo tiene 1 clave. Todas las páginas, que son *hoja* del árbol, están en el nivel más bajo de éste, en la figura el nivel 3. Las claves mantienen una ordenación de izquierda a derecha dentro de cada página. Estas claves dividen a los nodos descendientes a *la manera de un árbol de búsqueda*, claves de nodo izquierdo menores, claves de nodo derecho mayores. Esta organización supone una extensión natural de los árboles binarios de búsqueda. La forma de localizar una clave en el árbol *B* consiste en seguir un camino de búsqueda, que se determina de igual manera que en los árboles binarios, con una adaptación a la característica de que cada nodo tiene *m–1* claves.

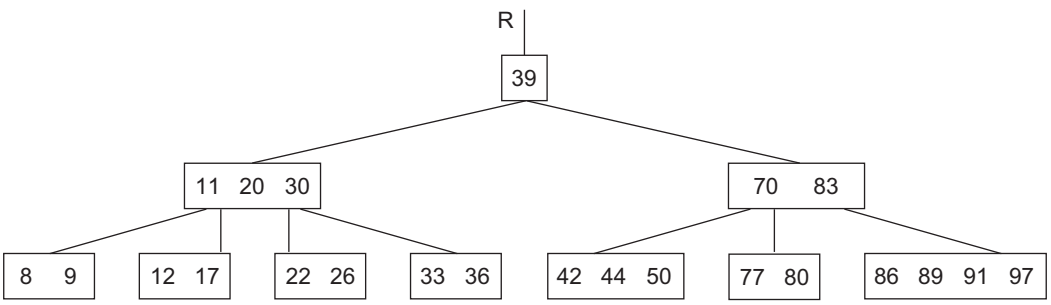


Figura 17.19. Árbol B de orden 5 de claves enteras.

17.6. *TAD ÁRBOL B* Y REPRESENTACIÓN

A la forma de organizar claves en la estructura árbol *B*, se le asocian una serie de operaciones básicas para poder buscar, añadir y eliminar claves. En definitiva, un árbol *B* es un tipo abstracto de datos con las siguientes operaciones básicas:

Crear	Inicializa el árbol *B* como árbol vacío.
Buscar	Dada una clave realiza la búsqueda de la clave. Devuelve la dirección del nodo y la posición en éste.
Insertar	Añade una nueva clave al árbol *B*. El árbol resultante sigue teniendo las características de árbol *B*.
Eliminar	Borra una clave del árbol B. El árbol resultante mantiene las características de árbol *B*.

17.6.1. Representación de una página

Los elementos de un árbol B son los nodos o *páginas* del árbol. Una *página* guarda las claves y las direcciones de las ramas de *páginas descendientes*. De manera natural surge la utilización de dos arrays dinámicos, y un atributo adicional `cuenta` que en todo momento contenga el número de claves de la *página*. El atributo privado `m` contiene siempre el numero de páginas descendientes máximo que puede contener la página. La clase `Pagina` declara los dos arrays dinámicos de claves y punteros a páginas con visibilidad *protegida*, para que las operaciones tengan restricciones de acceso. El constructor crea los arrays, ajustados al orden del árbol que recibe como parámetro. Además, el método `nodoLLeno()`, devuelve *verdadero* si el número de claves es `m-1` (máximo de claves de una página). El método `nodoSemiVacio()`, devuelve *verdadero* si el número de claves es menor que `m/2` (mínimo de claves que puede haber en una página). Se incluyen, además, los métodos para obtener y poner las claves y ramas de cada uno de los atributos de la página. Por su parte las funciones miembro `Ocuenta()` y `Pcuenta()`, se encargan de permitir el acceso al atributo `cuenta` y la modificación del propio atributo.

```
typedef int tipoClave;
class Pagina;
typedef Pagina * PPagina;
class Pagina
{
protected:
    tipoClave *claves; // puntero array de claves variables
    PPagina *ramas;    // puntero array de punteros a páginas variable
    int cuenta;        // número de claves que almacena la página
private:
    int m;      //máximo número de claves que puede almacenar la página
public:
    // crea una página vacía de un cierto orden dado
    Pagina (int orden)
    {
       m = orden;
       claves = new tipoClave[orden];
       ramas = new PPagina[orden];
       for (int k = 0; k <= orden; k++)
       ramas[k] = NULL;
     }
```

```
     // decide si un nodo está lleno
     bool nodoLLeno()
     {
       return (cuenta == m - 1);
     }
     // decide si una página tiene menos de la mitad de claves
     bool nodoSemiVacio()
     {
        return (cuenta < m / 2);
     }
      // obtener la clave que ocupa la posición i en el array de claves
     tipoClave Oclave(int i){ return claves[i];}
      // cambiar la clave que ocupa la posición i en el array de claves
     void Pclave(int i, tipoClave clave){ claves[i] = clave;}
      // obtener la rama que ocupa la posición i en el array de ramas
     Pagina* Orama(int i){ return ramas[i];}
      // cambiar la rama que ocupa la posición i en el array de ramas
     void Prama(int i, Pagina * p) { ramas[i] = p;}
      // obtener el valor de cuenta
     int Ocuenta(){ return cuenta;}
      // cambiar el valor de cuenta
     void Pcuenta( int valor) { cuenta = valor;}
};
```

Nota de programación

En un *árbol B de orden m*, el número de ramas de una página *no hoja* es uno más que el de sus claves. Por ello, las ramas se indexan de 0 a `m-1`, y las claves de `1` a `m-1`. El subárbol de `ramas[0]` se corresponde con las cla ves descendientes menores que `claves[1]`.

17.6.2. Tipo abstracto árbol B, clase ÁrbolB

El interfaz de la clase *ÁrbolB* se corresponde con las operaciones fundamentales del tipo abstracto: *buscar, insertar, eliminar*. Los métodos internos de la clase son los encargados del desarrollo de las operaciones del *tipo abstracto*. La clase tiene el atributo *orden* y la referencia al raíz del árbol como un puntero a la clase `Pagina`. Los constructores de la clase establece el orden del árbol e inicializa la raíz a `NULL` (*árbol vacío*). Se incluyen, además, los métodos públicos encargados de acceder y modificar el orden del árbol y la raíz del árbol.

```
class ArbolB
{
protected:
    int orden;
    Pagina *raiz;
public:
    ArbolB()
    {
        orden = 0;
        raiz = NULL;
    };
    ArbolB(int m)
```

```
    {
        orden = m;
        raiz = NULL;
    }
    bool arbolBvacio()
    {
        return raiz == NULL;
    }
    Pagina * Oraiz(){ return raiz;}
    void Praiz( Pagina * r){ raiz = r;}
    int Oorden(){ return orden;}
    void Porden(int ord){ orden = ord;}
    void crear() { orden = 0; raiz = NULL;}
    Pagina* buscar(tipoClave cl, int &n);
    void insertar(tipoClave cl);
    void eliminar(tipoClave cl);
};
```

17.7. FORMACIÓN DE UN ÁRBOL B

Las claves que se añaden a un árbol B siempre se insertan a partir de un nodo hoja, como ocurre en los árboles binarios. Además, al estar un árbol B perfectamente equilibrado, todas la hojas de un árbol B se encuentren en el mismo nivel, esto impone el comportamiento característico de los árboles B: *crecen "hacia arriba", crecen en la raíz*. Los pasos a seguir para añadir un nueva clave en un árbol B:

- Búsqueda de la clave a insertar en el árbol. Se sigue el *camino de búsqueda* que determina las claves de los nodos.
- En el caso de que la clave no esté en el árbol, la búsqueda termina en un nodo *hoja*. Entonces, empieza el proceso de inserción de la clave.
- De no estar lleno el nodo hoja, la inserción es posible en ese nodo y termina la inserción.
- Si está llena la *hoja*, la inserción en ella no es posible. Ahora se pone de manifiesto el comportamiento característico de los árboles B, se divide el nodo en dos en el mismo nivel del árbol, excepto la clave *mediana* que no se incluye en ninguno de los dos nodos, sino que *sube* en el árbol por el *camino de búsqueda* para, a su vez, repetir el proceso de inserción en el nodo *antecedente*. Por esto ***un árbol B crece hacia arriba***; esta *ascensión* de la clave *mediana* puede propagarse y llegar al nodo raíz, entonces éste se divide en dos nodos y la clave enviada hacia arriba se convierte en una nueva raíz. Ésta es la forma que tiene el *árbol B* de crecer en altura.

17.7.1. Creación de un árbol B de orden 5

A continuación se va a seguir los pasos de formación de un *árbol B* de orden `5`, para facilitar la comprensión las claves son números enteros. Al ser de orden `5` el número máximo de claves de un nodo es `4` y el máximo de ramas `5`.

Supóngase que las claves que se insertan:

```
6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27
```

- Con las cuatro primeras se completa el primer nodo, en orden creciente a la manera de árbol de búsqueda. La Figura 17.20(*a*) muestra este nodo.

- La clave siguiente, 8,encuentra el nodo ya lleno. La clave *mediana* de las cinco claves es 6. El nodo lleno se divide en dos, la clave mediana *"sube"* y como no hay nodo antecedente se crea otro nodo con la clave *mediana*, es la nueva raíz del árbol. El árbol ha crecido en altura como puede observarse en la Figura 17.20(*b*).

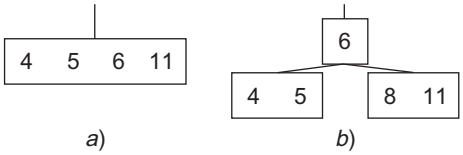


Figura 17.20. Formación de un árbol B de orden 5.

- Las siguientes claves 9, 12 se insertan en el nodo derecha de la raíz, según el criterio de búsqueda, ambas claves son mayores que 6. La Figura 17.21*a*) muestra el árbol después de esta inserción.
- La clave 21 *baja* por el *camino de búsqueda*, rama derecha del nodo raíz. El nodo hoja que le corresponde está lleno; por consiguiente, el nodo se parte en dos y la clave mediana, 11, asciende por el *camino de búsqueda* para ser insertada en el nodo antecedente, en este caso la raíz. El proceso se repite, ahora si hay *hueco* para que la clave 11 sea insertada. La Figura 17.21*b*) muestra el árbol resultante después de la creación del nuevo nodo.

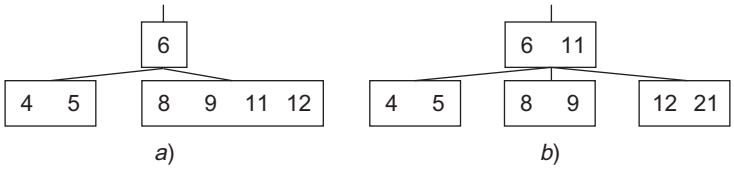


Figura 17.21. Inserción de nuevas claves en un árbol B de orden 5.

- Las siguientes claves 14, 10, 19 se insertan en los nodos *hoja* que les corresponde, según el *camino de búsqueda*.
- Al insertar la siguiente clave, 28, el *camino de búsqueda* determina que se inserte en un nodo que está lleno (nodo derecho del último nivel). De nuevo se produce el proceso de división del nodo y ascensión, por el *camino de búsqueda*, de la clave *mediana* que ahora es 19. El antecedente, nodo raíz, tiene dos claves y, por consiguiente, se inserta 19.
- Las claves que vienen a continuación, 3, 17, 32, 15 son insertadas en los nodos *hoja* que determina el *camino de búsqueda*. Cada uno de estos nodos tiene un número de claves menor que el máximo, m-1, por ello cada clave se inserta directamente en la hoja, en la posición que le corresponde según su valor. La Figura 17.22 representa al árbol después de estas inserciones.

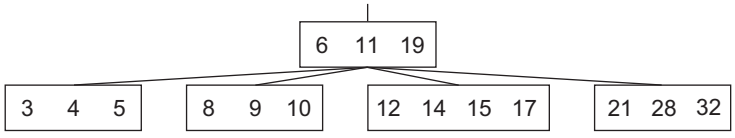


Figura 17.22. Representación de árbol B de orden 5.

- Ahora hay que añadir la clave `16`; *"baja"* por el *camino de búsqueda* (rama derecha de clave `11`). El nodo donde va a ser insertado está lleno, se produce la división del nodo y la clave mediana, `15`, *"sube"* por el *camino de búsqueda* para que sea añadida en el nodo antecedente. Como éste no está lleno, se inserta la clave `15`; el nodo raíz ha quedado completo (Figura 17.23).

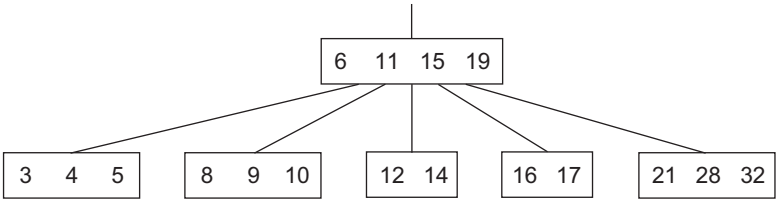


Figura 17.23. Árbol B de orden 5 después de una división de nodo .

- Al insertar la clave `26` se completa el nodo más a la derecha del último nivel. Por último, la clave `27` *"baja"* por el mismo *camino de búsqueda*, el nodo *hoja* que le corresponde está completo, entonces se divide en dos y *"sube"* la clave *mediana*, `27`, al nodo *padre*. Ocurre que este también está completo, se repite el proceso de división en dos nodos, y *"sube"* la nueva clave *mediana*, `15`. Como el nodo dividido es el raíz, con esta clave *mediana* se forma un nuevo nodo raíz: el árbol ha crecido en altura. El *árbol crece hacia arriba*, hacia la raíz.

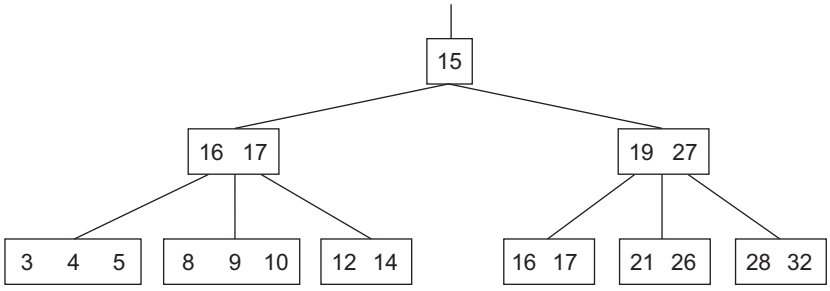


Figura 17.24. Árbol B de orden 5 con todas las claves.

A tener en cuenta

En el proceso de formación de un *árbol B* hay que destacar dos hechos relevantes. El primero, la división de un nodo prepara la estructura para nuevas inserciones, ya que deja *huecos* en los nodos implicados . En segundo lugar , la clave que *"sube"* al nodo antecedente es la clave *mediana* del nodo lleno, no tiene por qué coincidir con la clave que se está insertando; se puede afirmar que no importa el orden en que lleguen las claves en el balanceo del árbol.

17.8. BÚSQUEDA DE UNA CLAVE EN UN ÁRBOL B

Los elementos de un nodo en un árbol B son *claves de búsqueda*, dividen a las ramas inmediatas, rama izquierda y derecha, a la manera de un árbol de búsqueda. Por consiguiente, el algoritmo es similar al de búsqueda en un árbol binario ordenado, salvo que en los árboles B cuando se está analizando una *página* hay que inspeccionar las claves de que consta. La inspección da como resultado la posición de la clave, o bien la rama por donde seguir buscando (*camino de búsqueda*).

El método que implementa la operación, si encuentra la clave, devuelve la dirección del nodo que contiene a la clave y la posición que ocupa. Si la clave no está en el árbol, devuelve `NULL`. La búsqueda en cada nodo la realiza el método auxiliar `buscarNodo()`.

17.8.1. `buscarNodo()`

El método devuelve `true` si encuentra la clave en el árbol. Además, en el argumento *k* se obtiene la posición que ocupa la clave en la *página*, o bien la rama por donde continuar el proceso de búsqueda. Aprovecha la ordenación de las claves en el nodo, de tal forma que inspecciona las claves de la *página* en orden descendente y termina cuando `claves[index] <= cl`, (`actual->Oclave(index) <= cl`) así se asegura que *index* es la posición que ocupa la clave o el índice de la rama del árbol en la que puede ser encontrada.

```
bool ArbolB::buscarNodo(Pagina* actual, tipoClave cl, int & k)
{
     int index;
     bool encontrado;
     if (cl < actual->Oclave(1))
     {
       encontrado = false;
       index = 0;
     }
     else
     {      // orden descendente
       index = actual->Ocuenta();
       while (cl < actual->Oclave(index) && (index > 1))
         index--;
       encontrado = cl == actual->Oclave(index);
     }
     k = index;
     return encontrado;
}
```

17.8.2. `buscar()`

Este método privado de la clase `ArbolB` controla el proceso de búsqueda. El método *baja* por las ramas del árbol hasta encontrar la clave. El método se implementa recursivamente, la condición para terminar de hacer llamadas recursivas es que se haya localizado la clave, o bien el puntero a la página `actual es` a `NULL`, por tanto, no hay más nodos en los que buscar. Devuelve la referencia al nodo donde se encuentra la clave, o bien `NULL` . La función miembro está sobrecargada; la función interna privada realiza la búsqueda desde la raíz.

```
Pagina* ArbolB::buscar( tipoClave cl, int &n)
{
   return buscar( raiz, cl, n);
}

Pagina* ArbolB::buscar(Pagina* actual, tipoClave cl, int &n)
{
   if (actual == NULL)
   {
      return NULL;
   }
   else
   {
      bool esta = buscarNodo(actual, cl, n);
      if (esta) // la clave se encuentra en el nodo actual
        return actual;
      else
        return buscar(actual->Orama(n), cl, n);  //llamada recursiva
   }
}
```

17.9. INSERCIÓN EN UN ÁRBOL B

Debido a que las claves de un árbol B están organizadas como un árbol *m-ario de búsqueda*, la operación que inserta una clave en un árbol B sigue la misma estrategia que la inserción en un árbol binario de búsqueda. La nueva clave siempre se inserta en una hoja, para lo cual *baja* por el *camino de búsqueda*, hasta alcanzar el nodo hoja. Esta primera parte del algoritmo utiliza el método `buscarNodo()para` determinar la rama por donde *bajar*.

Una vez en el nodo hoja, si no está lleno, se inserta directamente en la posición que devuelve `buscarNodo()`, de tal forma que la clave queda ordenada. El método `meterPagina()`realiza esta parte de la operación.

Sólo si el nodo está lleno la inserción afecta a la estructura del árbol, ya que se ha de crear un nuevo nodo y, además, para que el árbol mantenga las características de *árbol B*, *asciende* la clave mediana al nodo antecedente. El método `dividirNodo()` es donde se implementan las acciones de esta parte del algoritmo.

La formulación recursiva es la más adecuada para reflejar el proceso de propagación, *hacia arriba*, en la *división de los nodos*, debido a que al retornar la llamada recursiva se *regresa por el camino de búsqueda*, se pasa por los nodos en sentido inverso a como se *bajó*.

El método interno `insertar()` tiene dos argumentos, la clave y el nodo raíz; ésta puede cambiar, si el árbol crece en altura y por ello devuelve la referencia al raíz. Pasa control a `empujar()` que, con una estrategia recursiva, realiza el proceso de inserción. Los métodos `meterPagina()` y `dividirNodo()` son llamados desde `empujar()`, según que la inserción se realice en un nodo con posiciones vacías o lleno, respectivamente.

17.9.1. Método `insertar()`

Se encarga de crear una nueva raíz si la *propagación, hacia arriba,* del proceso de división llega al actual raíz (el árbol aumenta su altura). El método está sobrecargado. En el primer caso,

el método es público y realiza la llamada al método `insertar()` pero teniendo como parámetro la raíz del árbol. Véase Figura 17.25.

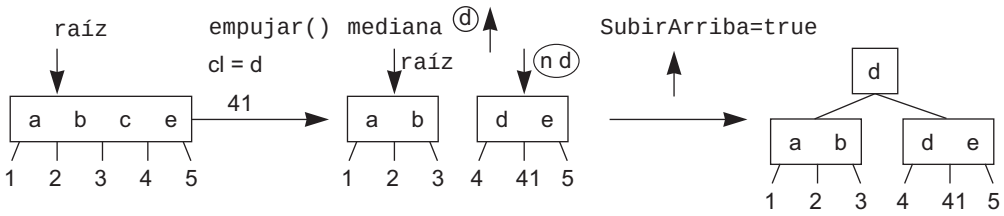


Figura 17.25. Crecimiento en altura de un árbol.

```
 // método público
void ArbolB::insertar(tipoClave cl)
{
    raiz = insertar(raiz, cl);
}
//método privado
Pagina* ArbolB::insertar(Pagina* raiz, tipoClave cl)
{
    bool subeArriba;
    int mediana;
    Pagina* nd;
    subeArriba = empujar(raiz, cl, mediana, nd);
    if (subeArriba)
    {
      // El árbol crece en altura por la raíz.
      // sube una nueva clave mediana y un nuevo hijo derecho nd
      // en la implementación se mantiene que las claves que son
      // menores que mediana se encuentran en raiz y las mayores en nd
       Pagina* p;
       p = new Pagina(orden);   // nuevo nodo
       p->Pcuenta(1);           // tiene una sola clave
       p->Pclave(1,mediana);
       p->Prama(0,raiz);        // claves menores
       p->Prama(1,nd);          // claves mayores
       raiz = p;
    }
    return raiz;
}
```

17.9.2. Método `empujar()`

Es el método más importante, controla la realización de las tareas de búsqueda y posterior inserción. Lo primero que hace es *"bajar"* por el *camino de búsqueda* hasta llegar a una rama vacía. A continuación, se prepara para *"subir"* (activa el indicador `subeArriba`) por las páginas del camino y realizar la inserción. En esta *vuelta atrás*, primero se encuentra con la *Página* hoja, si hay *hueco* mete la clave y el proceso termina. De estar completa, llama a `divi-`

`dirNodo()` que crea una nueva *Página* para repartir las claves; la clave *mediana* sigue *"subiendo"* por el *camino de búsqueda*.

La simulación de *"bajar"* y luego *"subir"* por el *camino de búsqueda* se implementa fácilmente mediante las llamadas recursivas de `empujar()`, de tal forma que cuando las llamadas retornan, se está volviendo (*"subiendo"*) a las de la *Páginas* por los que pasó anteriormente: *regresa* por el *camino de búsqueda*.

La inserción de la clave mediana en la *Página* antecesor, debido a la división de la *Página*, puede, a su vez, causar el desbordamiento de la misma, dando como resultado la propagación del proceso de partición *hacia arriba*, pudiendo llegar a la raíz. Ésta es la única forma de que aumente la altura del árbol B. En el caso de que la altura del árbol aumente es cuando el método `insertar()` crea un nuevo nodo que almacena una sola clave y dos punteros que apuntan a Páginas con claves menores y mayores respectivamente.

El método `empujar()` es privado y retorna el valor verdadero si hay clave que sube hacia arriba en cuyo caso se retorna en mediana, y además las claves más pequeñas están apuntadas por el puntero a la página actual y las mayores por el puntero a página nuevo.

```
bool ArbolB::empujar(Pagina* actual, tipoClave cl,
                     tipoClave &mediana, Pagina *& nuevo)
{
   int k ;
   bool subeArriba = false;
   if (actual == NULL)
   { // envía hacia arriba la clave cl y su rama derecha NULL
     // para que se inserte en la Página padre
      subeArriba = true;
      mediana = cl;
      nuevo = NULL;
       // el dato Página de nuevo está a NULL
   }
   else
   {
      bool esta;
      esta = buscarNodo(actual, cl, k);
      if (esta)
        throw "\nClave duplicada";
        // siempre se ejecuta
      subeArriba = empujar(actual->Orama(k), cl, mediana, nuevo);
        // devuelve control; vuelve por el camino de búsqueda
      if (subeArriba)
      {
         if (actual->nodoLLeno()) // hay que dividir la página
            dividirNodo(actual, mediana, nuevo, k);
      }
      else
      { //cabe en la página, se inserta la mediana y su rama derecha
        subeArriba = false;
        meterPagina(actual, mediana, nuevo, k);
      }
   }
   return subeArriba;
}
```

Nota de programación

En un árbol B los elementos son claves de búsqueda, por ello no se contempla que haya elementos repetidos. Ésa es la razón para que el método *empujar* levante una excepción cuando el método `buscarNodo()` encuentra la clave que se quiere insertar.

18.5.3. Método `meterPagina()`

Este método privado de la clase `ArbolB` inserta una clave en una *Página* que tiene un número de claves menor que el máximo. El método es invocado una vez que `empujar()` ha comprobado que hay *hueco* para añadir a la *Página* una nueva clave. Se le pasa, como argumentos, la dirección de la *Página*, la clave, la dirección de la rama con la *Página* sucesor, y la posición a partir de la cual se inserta. Véase Figura 17.26

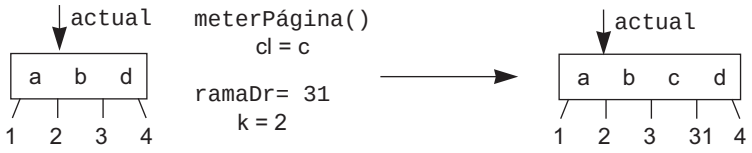


Figura 17.26. Proceso de Meter en una página.

```
void ArbolB::meterPagina(Pagina* actual, tipoClave cl,
                         Pagina *ramaDr, int k)
{
     // desplaza a la derecha los elementos para hacer un hueco
    for (int i = actual->Ocuenta(); i >= k + 1; i--)
     {
         actual->Pclave(i + 1, actual->Oclave(i));
         actual->Prama(i + 1, actual->Orama(i));
    }
     // pone la clave y la rama derecha en la posición k+1
    actual->Pclave(k + 1, cl);
    actual->Prama(k + 1, ramaDr);
     // incrementa el contador de claves almacenadas
    actual->Pcuenta(actual->Ocuenta()+1) ;
}
```

17.9.4. Método `dividirNodo()`

Este método resuelve el problema de que la *Página* donde se debe insertar la clave esté llena. Virtualmente, la *Página* se divide en dos y la clave mediana es enviada *hacia arriba*, para una *re-inserción* posterior en una *Página* padre o bien en una nueva raíz en el caso de que el árbol deba crecer en altura. Para ello, se crea una nueva *Página* a la que se desplazan las claves mayores de la mediana y sus correspondientes ramas, dejando en la *Página* original las claves menores.

El algoritmo, en primer lugar, encuentra la posición que ocupa la clave mediana; después mueve claves y ramas a la *Página* nueva y, según la posición k de inserción, mete la clave en la Página original o en la nueva. Por último, *"extrae"* la clave mediana que siempre se deja en el nodo original. Con la técnica de llamadas recursiva, el método `empujar()`insertará, posteriormente, la clave mediana en la *Página* antecedente. El argumento `mediana` recibe la clave que se inserta y se actualiza con la nueva mediana; de igual forma, el argumento `nuevo` recibe la rama derecha de la clave a insertar y se actualiza con la *Página* creada.

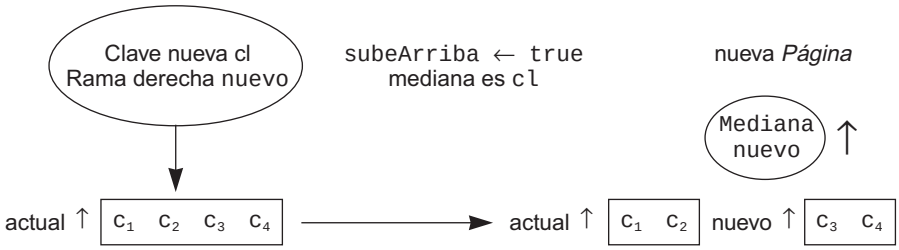


Figura 17.27. Proceso de partición de una *Página*.

```
void ArbolB::dividirNodo(Pagina* actual, tipoClave &mediana,
                         Pagina * &nuevo, int pos)
{
    int i, posMdna, k;
    Pagina *nuevaPag;
    k = pos;
     // posición de clave mediana
    posMdna = (k <= orden/2) ? orden/2 : orden/2 + 1;
    nuevaPag = new Pagina(orden);
    for (i = posMdna + 1; i < orden; i++)
    {
      /* desplazada la mitad derecha a la nueva Página, la clave
        mediana se queda en Página actual */
      nuevaPag->Pclave(i - posMdna, actual->Oclave(i));
      nuevaPag->Prama(i - posMdna, actual->Orama(i));
    }
    nuevaPag->Pcuenta ((orden - 1) - posMdna); // claves de nueva Página
    actual->Pcuenta(posMdna);   // claves en Página origen
     // inserta la clave y rama en la Página que le corresponde
    if (k <= orden / 2)
        meterPagina(actual, mediana, nuevo, pos); // en Página origen
    else
    {
        pos = k - posMdna;
        meterPagina(nuevaPag, mediana, nuevo, pos); // en Página nueva
    }
     // extrae clave mediana de la Página origen
    mediana = actual->Oclave(actual->Ocuenta());
     // Rama0 del nuevo nodo es la rama de la mediana
    nuevaPag->Prama(0, actual->Orama(actual->Ocuenta()));
    actual->Pcuenta(actual->Ocuenta() -1); // se quita la mediana
    nuevo = nuevaPag; // devuelve la nueva Página
}
```

17.9.5. Método `escribir()`

El método público `escribir()` realiza la tarea de visualizar en pantalla un `Arbolb` pequeño. Al igual que otros métodos está sobrecargado, el primero de ellos realiza la llamada al segundo con un argumento más que es la raíz del árbol. El parámetro h de escribir se usa para formatear la salida y que se pueda visualizar en la pantalla el árbol.

```
void ArbolB::escribir( )
{
     escribir (raiz, 1);
}
void ArbolB::escribir( Pagina * r, int h)
{
   int i;
   if (r != NULL)
   {
       escribir( r->Orama(0),h + 5);
       for (i = 1; i <= r->Ocuenta()/2;i++)
       { // llamadas recursivas a la mitad de los subárboles
          escribir(r->Orama(i),h + 5);
          cout << endl;
       }
        // visualización de las claves de la página apuntada por r
       for (i = 1; i<= r->Ocuenta();i++)
       {
           for (int j = 0; j <= h; j++)
              cout << " ";
           cout << r->Oclave(i) << endl;
       }
       // llamadas recursivas a la otra mitad de los subárboles
       for (i = r->Ocuenta() / 2 + 1 ; i<= r->Ocuenta();i++)
           escribir(r->Orama(i),h + 5);
       cout << endl;
   }
}
```

17.10. LISTADO DE LAS CLAVES DE UN ÁRBOL B

Recorrer una estructura de datos implica *visitar* cada uno de los elementos de que consta, así ocurre con los vectores, con las listas lineales y circulares, ... En los árboles se distinguen distintos tipos de recorrido, en profundidad y en anchura. El recorrido en profundidad se puede hacer en *preorden*, *inorden* y *postorden*. En los árboles B se puede aplicar los mismos criterios para visitar sus claves, con la particularidad de que un nodo (*Página*) de un árbol B hay, normalmente, más de una clave y más de dos subárboles.

Un recorrido que resulta interesante en un árbol B es el que *visita* las claves del árbol en orden creciente. Como cada clave de la *Página* del árbol B divide a las de sus ramas adyacentes a la manera de un árbol de búsqueda, se aplica el recorrido en *inorden* de un árbol binario, con una pequeña adaptación, para *visitar* en orden creciente las claves de un árbol B. La implementación se realiza con dos métodos: `listaCreciente()` que es el interfaz público de la clase `ArbolB`, y `inOrden()` de la clase `ArbolB` como método privado que realiza el recorrido por las claves.

```
void ArbolB::listarCreciente()
{
    inOrden(raiz) ;
}
void ArbolB::inOrden(Pagina *r)
{
    if (r)
    {
        inOrden(r->Orama(0));
        for (int k = 1; k <= r->Ocuenta(); k++)
        {
            cout << r->Oclave(k) << " ";
            inOrden(r->Orama(k));
        }
    }
}
```

Nota de programación

El recorrido en *inorden* del árbol B se puede implementar iterativamente. Para ello se utiliza una *Pila* en la que se guardan las referencias a las Páginas del árbol.

EJERCICIO 17.3. Escribir una aplicación para gestionar un árbol B de orden 5, utilizando las operaciones del TAD ya implementadas en la clase ArbolB . Deben realizarse estas acciones:

- *Dar de alta claves.*
- *Buscar una clave dada.*
- *Eliminar una clave dada.*
- *Listar las claves en orden creciente.*

El programa crea un `ArbolB` de con claves de tipo entero. Las operaciones se presentan en forma de menú; la opción *"dar de alta claves"* genera n claves aleatoriamente, en el rango de `1` a `9999`, que se añaden al árbol llamando al método `insertar()`. Las acciones de *buscar, eliminar y listar* se corresponden con las operaciones implementadas en la clase.

Codificación. Véase página web del libro

Lecturas recomendadas para profundizar en el tema

En la lectura recomendada Anexo A puede consultar una ampliación sobre "Eliminación en árboles AVL".
En la lectura recomendada Anexo B puede consultar una ampliación sobre "Eliminación en árboles B".

RESUMEN

Los árboles binarios de búsqueda son estructuras de datos importantes que permiten localizar una *clave de búsqueda* con un coste logarítmico. Sin embargo, el tiempo de búsqueda puede crecer hasta hacerse lineal si el árbol está degenerado; en ese sentido, la eficiencia de las operaciones dependen de lo aleatorias que sean las claves de entrada. Para evitar este problema surgen los árboles de búsqueda equilibrados, llamados árboles AVL. La eficiencia de la *búsqueda* en los árboles equilibrados es mayor que en los árboles de búsqueda no equilibrados.

Un árbol binario equilibrado es un árbol de búsqueda caracterizado por diferir las alturas de las ramas derecha e izquierda del nodo raíz a lo sumo en uno, y que los subárboles izquierdo y derecho son, a su vez, árboles equilibrados.

Las operaciones de los árboles de búsqueda son también operaciones de los árboles equilibrados. Las operaciones de inserción y de borrado añaden o eliminan, respectivamente, un nodo en el árbol de igual forma que en los árboles de búsqueda no equilibrados. Además, es necesario incorporar una segunda parte de reestructuración del árbol para asegurar el equilibrio de cada nodo. La solución que se aplica ante un desequilibrio de un nodo viene dada por las *rotaciones*. Se pueden aplican dos tipos de rotaciones, *rotación simple y rotación doble*; el que se aplique una u otra depende del nodo desequilibrado y del *equilibrio* del hijo que se encuentra en la rama más alta.

Las operaciones de inserción y de borrado en árboles binarios equilibrados son más *costosas* que en los no equilibrados, debido a que el algoritmo debe *retroceder* por el *camino de búsqueda*, para actualizar el factor de equilibrio de los nodos que lo forman y al *coste* de la rotación cuando se viola la condición de equilibrio. Por el contrario, la operación de búsqueda es, de promedio, menos costosa que en los árboles binarios no equilibrados.

Los *árboles B* tienen una serie de características que permiten que la búsqueda de una clave sea muy eficiente, por ello, si las claves se encuentran en un disco, el número de accesos al disco disminuye notablemente. En aplicaciones que acceden a archivos que están en disco, el mayor tiempo de proceso es el de las operaciones de *entrada/salida*. Reducir el número de pasos para buscar una clave, o para insertar, supone reducir el número de operaciones de entrada y salida y, por consiguiente, el tiempo de proceso. Entonces, un árbol-B es una estructura de datos utilizada para almacenar claves (*de búsqueda*) que residen en memoria externa, tal como un disco.

El número de claves de una *Página* (nodo) en un árbol B *de orden m*, que no sea la raíz, es como mínimo `m/2` y como máximo `m-1`. Además, las claves de una *Página* divide a las claves de sus ramas a la manera de árbol de búsqueda. El número de ramas descendientes de una *Página* es una más que el número de claves, excepto las *Páginas* que son hoja.

Un árbol B siempre está perfectamente equilibrado. La inserción de una nueva clave se realiza en un hoja, si está llena *asciende* la clave *mediana* a la *Página padre*, si éste también está lleno sigue *ascendiendo*, pudiendo llegar a la raíz, si también se encuentra llena se crea una nueva *Página* con la clave *mediana*. Se dice que los árboles B crecen hacia arriba, hacia la raíz. Los árboles B más utilizados son los de orden 5.

BIBLIOGRAFÍA RECOMENDADA

Aho V.; Hopcroft, J., y Ullman, J.: *Estructuras de datos y algoritmos*. Addison Wesley, 1983.

Garrido, A., y Fernández, J.: *Abstracción y estructuras de datos en C++*. Delta, 2006.

Joyanes, L., y Zahonero, L.: *Algoritmos y estructuras de datos. Una perspectiva en C*. McGraw-Hill, 2004.

Joyanes, L.; Sánchez, L.; Zahonero, I., y Fernández, M.: *Estructuras de datos en C*. Schaum, 2005.

Kruse Robert, L.: *Estructura de datos y diseño de programa*. Prentice-Hall, 1998.
Lipschutz, S.: *Estructura de datos*. McGraw-Hill, 1987.
Nyhoff, L.: *TADs, Estructuras de datos y resolución de problemas con C++*. Pearson, Prentice-Hall, 2005.
Weis, Mark Allen: *Estructuras de datos y algoritmos*. Addison Wesley, 1992.
Wirth Niklaus: *Algoritmos + Estructuras de datos = programas*. 1986.

EJERCICIOS

17.1. Dibujar el árbol binario de búsqueda equilibrado que resulta las claves: 14, 6, 24, 35, 59, 17, 21, 32, 4, 7, 15 y 22.

17.2. Dada la secuencia de claves enteras: 100, 29, 71, 82, 48, 39, 101, 22, 46, 17, 3, 20, 25, 10. Dibujar el árbol AVL correspondiente. Eliminar claves consecutivamente hasta encontrar un nodo que viole la condición de equilibrio y cuya restauración sea con una rotación doble.

17.3. En el árbol construido en el Ejercicio 17.1 eliminar el nodo raíz. Hacerlo tantas veces como sea necesario hasta que se desequilibre un nodo y haya que aplicar una rotación simple.

17.4. Encontrar una secuencia de n claves que al ser insertadas en un árbol binario de búsqueda vacío se apliquen las cuatro rutinas de rotación: `II`, `ID`, `DD`, `DI`.

17.5. Dibujar el árbol equilibrado después de insertar en orden creciente 31 ($2^5 - 1$) elementos del 11 al 46.

17.6. ¿Cuál es el número mínimo de nodos de un árbol binario de búsqueda equilibrado de altura 10?

17.7. Escribir el método recursivo `buscarMin()` que devuelva el nodo de clave mínima de un árbol de búsqueda equilibrado.

17.8. Dibujar un árbol AVL de altura 6 con el criterio del peor de los casos, es decir, en el que cada nodo tenga como factor de equilibrio ± 1.

17.9. En el árbol equilibrado formado en el Ejercicio 17.8, eliminar una de las hojas menos profundas. Representar las operaciones necesarias para restablecer el equilibrio.

17.10. Escribir el método recursivo `buscarMax()` que devuelva el nodo de clave máxima de un árbol de búsqueda equilibrado.

17.11. Escribir los métodos `buscarMin()` y `buscarMax()` en un árbol de búsqueda equilibrado de manera iterativa.

17.12. Dada la secuencia de claves enteras: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80; dibuja el árbol B de orden 5 formado con dichas claves.

17.13. La operación de inserción de una clave en una *Pagina* llena se ha realizado partiendo ésta y elevando la clave mediana. Analizar, si es posible, esta otra estrategia: buscar el número de claves del hermano izquierdo y derecho, si alguno no está lleno mover claves para realizar la inserción.

17.14. Un árbol B de orden 5 se insertan las claves de manera secuencial: 1, 2, 3, ... n. ¿Qué claves dan origen a la división de una *Pagina*? ¿Qué claves hacen que la altura del árbol crezca? Las claves son eliminadas en el mismo orden en que fueron insertadas. ¿Qué claves hacen que las *Paginas* se queden con un número de claves menor que 2 y den lugar a la unión de dos? ¿Qué claves hacen que la altura del árbol disminuya?

17.15. La búsqueda de una clave en un árbol B se ha implementado siguiendo una estrategia recursiva, implementar de nuevo la operación con una estrategia iterativa.

17.16. Se asume que la unidad de disco contiene un número de pistas, a su vez cada pista está dividida en un número de bloques de tamaño *m*. En cuanto a tiempo de proceso, las operaciones que realiza un programa de acceso a los archivos ubicados en disco prevalecen frente a las operaciones en memoria; el tiempo de acceso a disco es del orden de decenas de milisegundo (cada vez se mejora este tiempo). Con el fin de minimizar el tiempo de acceso al disco, el orden de árbol B debe ser tal que permita que una hoja de tamaño máximo quepa en un bloque del disco. Determinar el número máximo de accesos al disco necesario para encontrar un elemento en un árbol B de *n* elementos.

17.17. Se estima que un archivo va a tener 1 millón de registros, cada registro ocupa 50 bytes de memoria. En el supuesto de que cada bloque tenga una capacidad de 1.000 bytes y que la referencia al bloque ocupa 4 bytes, diseñar una organización con árboles B para este archivo.

17.18. Un árbol B* se considera un tipo de árbol B con la característica adicional de que cada nodo está, al menos, lleno en las dos terceras partes, en lugar de la mitad como ocurre a un árbol B, excepto quizá el nodo raíz. La inserción de una nueva clave en el árbol B* sigue los mismos pasos que la inserción en un árbol B, salvo que si el nodo donde se ha de insertar está lleno, se mueven claves entre el nodo padre, el hermano (con un número de claves menor que el máximo), y el nodo donde se inserta para dejar hueco y realizar la operación. De esta forma, se pospone la división del nodo hasta que los hermanos estén completos, entonces éstos pueden dividirse en tres nodos, cada uno estará con una ocupación de las dos terceras partes. Codificar los cambios que necesitan los métodos que implementan la operación de inserción para aplicarla a un árbol B*.

17.19. Dada la secuencia de claves enteras: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80; dibuja el árbol B* de orden 5 que se construye con esas claves.

PROBLEMAS

17.1. Dado un archivo de texto construir un árbol AVL con todas sus palabras y frecuencia. El archivo se denomina *carta.dat.*

17.2. Añadir al programa escrito en el Problema 17.1 una función que devuelva el número de veces que aparece en el texto, una palabra dada.

17.3. En el archivo *alumnos.txt* se encuentran los nombres completos de los alumnos de las escuelas taller de la Comunidad Alcarreña. Escribir un programa para leer el archivo y formar, inicialmente, un árbol de búsqueda con respecto a la clave *apellido*. Una vez formado el árbol, construir con sus nodos un árbol de Fibonacci.

17.4. La implementación de la operación insertar en un árbol equilibrado se realiza de manera natural en forma recursiva. Escribir de nuevo la codificación aplicando una estrategia iterativa.

17.5. Un archivo F contiene las claves, enteros positivos, que forman un árbol binario de búsqueda equilibrado R. El archivo se grabó en el recorrido por niveles del árbol R. Escribir un programa que realice las siguientes tareas: *a*) Leer el archivo F para volver a construir el árbol equilibrado. *b*) Buscar una clave, requerida al usuario, y en el caso de que esté en el árbol mostrar las claves que se encuentran en el mismo nivel del árbol.

17.6. La implementación de la operación *eliminar* en un árbol binario equilibrado se ha realizado en forma recursiva. Escribir de nuevo la codificación aplicando una estrategia iterativa.

17.7. En un archivo se ha almacenado los habitantes de n pueblos de la comarca natural *Peñas Rubias*. Cada registro del archivo tiene el nombre del pueblo y el número de habitantes. Se quiere asociar los nombres de cada habitante a cada pueblo, para lo que se ha pensado en una estructura de datos que consta de un array de n elementos. Cada elemento tiene el nombre del pueblo y la raíz de un árbol AVL con los nombres de los habitantes del pueblo.
Escribir un programa que cree la estructura. Como entrada de datos, los nombres de los habitantes que se insertarán en el árbol AVL del pueblo que le corresponde.

17.8. La operación de borrar una clave en un árbol AVL se ha realizado de tal forma que cuando el nodo de la clave a eliminar tiene las dos ramas se reemplaza por la clave mayor del subárbol izquierdo. Implementar la operación de borrado de tal forma que cuando el nodo a eliminar tenga dos ramas reemplace, aleatoriamente, por la clave mayor de la rama izquierda o por la clave menor de la rama derecha.

17.9. Al problema descrito en 17.7 se desea añadir la posibilidad de realizar operaciones sobre la estructura. Así, añadir la posibilidad de cambiar el nombre de una persona de un determinado pueblo. Esta operación debe de mantener el árbol como árbol de búsqueda, al cambiar el nombre y ser la clave de búsqueda el nombre puede ocurrir que se rompa la condición. Otra opción que debe de permitir es dar de baja un pueblo entero de tal forma que todos sus habitantes se añadan a otro pueblo de la estructura. Por último, una vez que se vaya a terminar la ejecución del programa grabar en un archivo cada pueblo con sus respectivos habitantes.

17.10. Una empresa de servicios tiene tres departamentos: *comercial*(*1*), *explotación*(*2*) y *marketing*(*3*). Cada empleado está adscrito a uno de ellos. Se ha realizado una redistribución del personal entre ambos departamentos, los cambios están guardados en el archivo *laboral.txt*. El archivo contiene en cada registro los campos: *identificador*, *origen*, *destino*. El campo origen puede tomar los valores 1, 2, 3 dependiendo del departamento origen del empleado, cuya identificación es una secuencia de 5 dígitos que es el primer campo del registro. El campo destino también puede tomar los valores 1, 2, 3 según el departamento al que es destinado.

Escribir un programa que guarde los registros del archivo en tres árboles AVL, uno por cada departamento origen y realice los intercambios de registros en los árboles, según el campo *destino*.

17.11. Escribir la codificación del método: `void listadoEnRango(int c1, int c2);` que muestre las claves de un árbol B mayores que c_1 y menores que c_2.

17.12. Escribir la codificación de el método `listadoDecreciente()`, que recorra y muestre las claves de un árbol B en orden decreciente.

17.13. Cada uno de los centros de enseñanza del estado consta de una biblioteca escolar. Cada centro de enseñanza está asociado con número de orden (valor entero), los centros de cada provincia tienen números consecutivos, en el rango de las unidades de 1.000. Así por ejemplo, a Madrid le corresponde del 1 al 1.000, a Toledo del 1.001 al 2.000 ... Escribir un programa que permita gestionar la información indicada, formando una estructura en memoria de árbol B con un máximo de 4 claves por página. La clave de búsqueda del árbol B es el número de orden del centro, además, tiene asociado el nombre del centro. El programa debe permitir añadir centros, eliminar, buscar la existencia de un centro por la clave y listar los centros existentes.

17.14. En el Problema 17.13 cuando se termina la ejecución se pierde toda la información. Modificar el programa para que al terminar la información se grabe la estructura en un archivo de nombre `centros.txt`. Escribir un programa que permita leer el archivo `centros.txt` para generar, a partir de el, la estructura árbol B. La estructura puede experimentar modificaciones: nuevos centros, eliminación de alguno existente, por consiguiente, al terminar la ejecución debe escribirse de nuevo el árbol en el archivo.

17.15. Se quiere dar más contenido a la información tratada en el Problema 17.13. Ya se ha especificado que la clave de búsqueda del árbol B es el número de orden del centro de enseñanza. Además, cada clave tiene que llevar asociada la raíz de un árbol binario de búsqueda que representa a los títulos de la biblioteca del centro. El árbol de búsqueda biblioteca tiene como campo clave el *título* del libro (tiene más campos, como *autor* …). Escribir un programa que partiendo de la información guardada en el archivo *centros.txt* cree un nuevo árbol B con los centros y el árbol binario de títulos de la biblioteca de cada centro.

17.16. A la estructura ya creada del Problema 17.13, añadir los métodos necesarios para realizar las siguientes operaciones:

- Dada una provincia cuyo rango de centros es conocido, por ejemplo de 3.001 a 3.780, eliminar en todos sus centros escolares los libros que estén repetidos en cada centro, y que informe del total de libros liberados.
- Dado un centro n, en su biblioteca se desea que de ciertos libros haya m ejemplares.

17.17. Implemente el TAD árbol B en memoria externa, en disco. Al hacerlo en memoria externa los enlaces deben ser números de registro en los que se almacena el nodo o página descendiente. Para crear un nuevo nodo hay que buscar un "*hueco*" en el archivo, representado por el número de registro, y escribir el nodo. Cada vez que se modifica un nodo, por una inserción de una clave o un borrado hay que escribir el registro.

CAPÍTULO 18
Grafos

Objetivos

Con el estudio de este capítulo usted podrá:

- Distinguir entre relaciones jerárquicas y otras relaciones.
- Definir un grafo e identificar sus componentes.
- Conocer estructuras de datos para representar un grafo.
- Conocer las operaciones básicas que se aplican sobre grafos.
- Encontrar los caminos que puede haber entre dos nodos de un grafo.
- Realizar en C++ la representación de los grafos y las operaciones básicas.
- Calcular el camino mínimo desde un vértice al resto de los vértices.
- Determinar si hay camino entre cualquier par de vértices de un grafo.
- Implementar los algoritmos más importantes sobre grafos.
- Representar con un grafo ciertas relaciones entre objetos y aplicar los algoritmos que determinan su conectividad.
- Saber cuándo se aplica un algoritmo de expansión de coste mínimo y cuándo un camino de coste mínimo.

Contenido

Conceptos clave

- Árbol de expansión.
- Camino.
- Caminos mínimos.
- Conexión y componente conexa.
- Factor de peso.
- Lista de adyacencia.
- Matriz de adyacencia.
- Ordenación topológica.
- Relación jerárquica.
- Vértice y arco.

INTRODUCCIÓN

Este capítulo introduce al lector a conceptos matemáticos importantes denominados grafos que tienen aplicaciones en campos tan diversos como sociología, química, geografía, ingeniería eléctrica e industrial, etc. Los grafos se estudian como estructuras de datos o tipos abstractos de datos. Este capítulo estudia las definiciones relativas a los grafos y la representación de los grafos en la memoria del ordenador. El capítulo investiga dos formas tradicionales de implementación de grafo, matriz de adyacencia y listas de adyacencia. También se estudian operaciones importantes y algoritmos de grafos que son significativos en informática.

Existen numerosos problemas que se pueden modelar en términos de grafos. Ejemplo de ello es la planificación de las tareas que completan un proyecto, encontrar las rutas de menor longitud entre dos puntos geográficos, calcular el camino más rápido en un transporte, determinar el flujo máximo que puede llegar desde una *fuente* a, por ejemplo, una urbanización.

La resolución de estos problemas requiere examinar todos los nodos y las aristas del grafo que representa al problema. Los algoritmos imponen implícitamente un orden en estas visitas: el nodo más próximo o las aristas más cortas, y así sucesivamente; no todos los algoritmos requieren un orden concreto en el recorrido del grafo.

Según lo anterior, se estudian en este capítulo el concepto de ordenación topológica, los problemas del camino más corto, junto con el concepto de árbol de expansión de coste mínimo. Estos algoritmos han sido desarrollado por grandes investigadores y en su honor se conocen por su nombre los algoritmos de Dijkstra, Warshall, Prim, Kruscal, Ford-Fulkerson ...

18.1. CONCEPTOS Y DEFINICIONES

Un grafo G agrupa *entes* físico o conceptuales y las relaciones entre ellos. Por tanto, un grafo está formado por un conjunto de vértices o nodos `V`, que representan a los *entes*, y un conjunto de arcos `A`, que representan las relaciones entre vértices. Se representa con el par `G = (V, A)`. La Figura 18.1 muestra un grafo formado por los vértices `V = {1,4,5,7,9}` y el conjunto de arcos `A = {(1,4), (4,1), (5,1), (1, 5), (7,9), (9,7), (7,5), (5,7), (4,9), (9,4)}`

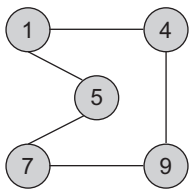


Figura 18.1. Grafo no dirigido.

Un arco o arista representa una *relación* entre dos nodos. Esta relación, al estar formada por dos nodos, se representa por `(u, v)` siendo `u`, `v` el par de nodos. El grafo es *no dirigido* si los arcos están formados por pares de nodos no ordenados, no apuntados; se representa con un segmento uniendo los nodos, u — v. El grafo de la Figura 18.1 es no dirigido.

Un grafo es dirigido, también denominado *digrafo*, si los pares de nodos que forman los arcos son ordenados; se representan con una flecha que indica la dirección de la relación,

u → v. El grafo de la Figura 18.2 que consta de los vértices, `V = {C, D, E, F, H}`, y de los arcos `A = {(C, D,), (D, F), (E, H), (H, E), (E, C)}` forman el grafo dirigido `G = {V, A}`

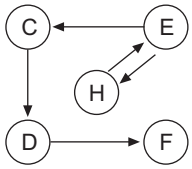


Figura 18.2. Grafo dirigido.

Dado el arco `(u, v)` de un grafo, se dice que los vértices u y v son adyacentes. Si el grafo es dirigido, el vértice u es adyacente a v, y v es adyacente de u.

En los modelos realizados con grafos, a veces, una relación entre dos nodos tiene asociada una *magnitud*, denominada factor de peso, en cuyo caso se dice que es un grafo valorado. Por ejemplo, los pueblos que forman una comarca junto a la relación entre un par de pueblos de *estar unidos por un camino:* esta relación tiene asociado el factor de peso, que es la distancia en kilómetros. La Figura 18.3 muestra un grafo valorado en el que cada arco tiene asociado un peso que es la longitud entre dos nodos.

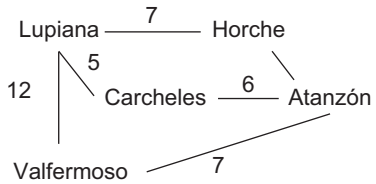


Figura 18.3. Grafo no dirigido valorado.

Definición

Un grafo permite modelar relaciones arbitrarias entre objetos. Un grafo `G = (V, A)` es un par formado por un conjunto de vértices o nodos, V, y un conjunto de arcos o aristas, A. Cada arco es el par `(u, w)`, siendo u, w dos vértices relacionados.

18.1.1. Grado de entrada, grado de salida de un nodo

El **grado** es una cualidad que se refiere a los nodos de un grafo. En un grafo no dirigido el grado de un nodo v, `grado(v)`, es el número de arcos que contiene a v. En un grafo dirigido se distingue entre grado de entrada y grado de salida; *grado de entrada* de un nodo v, `gradent(v)`, es el número de arcos que llegan a v; grado de salida de v, `gradsal(v)`, es el número de arcos que salen de v.

Así, en el grafo no dirigido de la Figura 18.3, `grado(Lupiana) = 3`. En el grafo dirigido de la Figura 18.2, `gradent(D) = 1` y el `gradsal(D) = 1`.

18.1.2. Camino

Un camino P de longitud n desde el vértice v_0 a v_n en un grafo G, es la secuencia de n + 1 vértices v_0, v_1, v_2, ..., v_n tal que (v_i, v_{i+1}) ∈ *A*(*arcos*)*para* $0 \leq i < n$. Matemáticamente el camino se representa por P = (v_0, v_1, v_2, ..., v_n).

En la Figura 18.4 se pueden encontrar más de un camino; por ejemplo, P1 = (4, 6, 9, 7) es un camino de longitud 3. Otro de los caminos es P2 = (10, 11), que tiene de longitud 1. Por tanto, se puede afirmar que la longitud del camino es el número de arcos que lo forman.

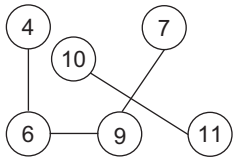


Figura 18.4. Grafo no dirigido de 5 vértices.

Definición

La longitud de un camino en un grafo no valorado es el número de arcos en el camino. En un grafo valorado, la longitud del camino con pesos es la suma de los pesos de los arcos en el camino.

En el grafo valorado de la Figura 18.3, el camino (Lupiana, Valfermoso, Atanzón) tiene de longitud 12 + 7 = 19.

En algunos grafos se dan arcos desde un vértice a sí mismo, (v, v); entonces el camino v → v es un bucle. Normalmente, en los grafos no hay nodos relacionados con sí mismo, no es frecuente encontrarse grafos con bucles.

Un camino P = (v_0, v_1, v_2, ..., v_n) es simple si todos los nodos que forman el camino son distintos, pudiendo ser iguales los extremos del camino v0, vn . En el grafo de la Figura 18.4 el camino P_1 y P_2 son caminos simples.

En un grafo dirigido, un ciclo es un camino simple cerrado. Por tanto, un ciclo empieza y termina en el mismo nodo, $v_0 = v_n$, y, además, debe tener más de un arco. Un grafo dirigido sin ciclos (acíclico) se acostumbra a denominar mediante la abreviatura GDA (Grafo Dirigido Acíclico). La Figura 18.5 muestra un grafo dirigido en el que los vértices (A, E, B, F, A) forman un ciclo de longitud 4. En general, un ciclo de longitud k se denomina *k-ciclo*.

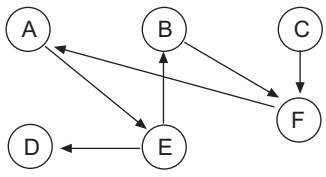


Figura 18.5. Grafo dirigido con ciclos.

Un grafo no dirigido es *conexo* si existe un camino entre cualquier par de nodos que forman el grafo. En el caso de un grafo dirigido con esta propiedad se dice que es *fuertemente conexo*. Además, un *grafo completo* es aquél que tiene un arco para cualquier par de vértices.

Figura 18.6. *a*) Grafo conexo. *b*) Grafo fuertemente conexo.

18.1.3. *Tipo Abstracto de Datos* Grafo

Un grafo consta de un conjunto de objetos o elementos junto a sus relaciones. Es preciso definir las operaciones básicas para construir la estructura y, en general, modificar sus elementos. En definiva, especificar el *tipo abstracto de datos grafo*.

Ahora se definen operaciones básicas, a partir de las cuales se construye el grafo. Su realización depende de la representación elegida (matiz de adyacencia, o listas de adyacencia).

arista (u, v).	Añade el arco o arista(`u, v`) al grafo.
aristaPeso(u,v, w).	Para un grafo valorado, añade el arco(`u,v`) al grafo y el coste del arco, `w`.
borraArco(u,v).	Elimina del grafo el arco(`u, v`).
adyacente(u,v).	Operación que devuelve *cierto* si los vértices `u, v` forman un arco.
nuevoVertice(u).	Añade el vértice `u` al grafo `G`.
borraVértice(u).	Elimina el vértice `u` del grafo `G`.

18.2. REPRESENTACIÓN DE LOS GRAFOS

Para trabajar con los grafos y aplicar algoritmos que permitan encontrar propiedades entre los nodos hay que pensar en su representación. Cómo representar un grafo en memoria interna, qué tipos o estructuras de datos se deben utilizar para considerar los nodos y los arcos.

Una primera simplificación que se hace es considerar a los vértices o nodos como números consecutivos, empezando por el vértice `0`. Hay que tener en cuenta que se tiene que representar un número (finito) de vértices y de arcos que unen dos vértices. Se puede elegir una representación secuencial, mediante un array bidimensional, conocida como *matriz de adyacencia*; o bien, una representación dinámica, mediante una estructura multienlazada, denominada *listas de adyacencia*. La elección de una representación u otra depende del tipo de grafo y de las operaciones que se vayan a realizar sobre los vértices y arcos. Para un grafo denso (tiene la mayoría de los arcos posibles) lo mejor es utilizar una matiz de adyacencia. Para un grafo disperso (tiene, relativamente, pocos arcos) se suele utilizar listas de adyacencia que se ajustan al número de arcos.

18.2.1. Matriz de adyacencia

La característica más importante de un grafo, que distingue a un grafo de otro, es el conjunto de pares de vértices que están *relacionados*, o que son adyacentes. Por ello, la forma más sen-

cilla de representar un grafo es mediante una matriz, de tantas filas / columnas como nodos, que permite modelar fácilmente esa cualidad.

Sea `G = (V, A)` un grafo de `n` nodos, siendo $V = \{v_0, v_1, \ldots, v_{n-1}\}$ el conjunto de nodos, y `A = {(vi, vj)}` el conjunto de arcos. Los nodos se pueden representar por números consecutivos de `0` a `n - 1`. La representación de los arcos se hace con una matriz `A` de $n_x n$ elementos, denominada matriz de adyacencia, tal que todo elemento a_{ij} puede tomar los valores:

$$a_{ij} = \begin{cases} 1 & \text{si hay un arco } (v_i, v_j) \\ 0 & \text{si no hay arco } (v_i, v_j) \end{cases}$$

La Figura 18.7 representa un grafo dirigido, suponiendo que el orden de los vértices es el de la secuencia `{D, F, K, L, R}`, entonces la matriz de adyacencia:

$$A = \begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix}$$

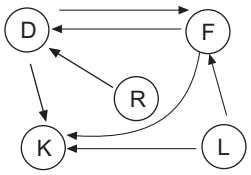


Figura 18.7. Grafo dirigido con los vértices `{D, F, K, L, R}`.

El grafo de la Figura 18.8 es no dirigido, está formado por `5` vértices. La matriz de adyacencia:

$$A = \begin{vmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{vmatrix}$$

puede observar, es una matriz simétrica. En los grafos no dirigidos la matriz de adyacencia siempre es simétrica ya que las relaciones entre vértices no son ordenadas: si v_i *está relacionado* con v_j, entonces v_j *está relacionado* con v_i.

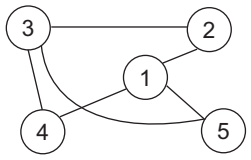


Figura 18.8. Grafo no dirigido con 5 vértices.

Los grafos que modelan problemas en los que un arco tiene asociado una magnitud, un *factor de peso*, también se representan mediante una matriz de tantas filas / columnas como nodos. Ahora un elemento cualquiera, a_{ij} representa el coste o *factor de peso* del arco (v_i, v_j). Un arco que no existe se puede representar con un valor imposible, por ejemplo, *infinito*; también, se puede representar con el valor `0`, dependiendo de que para el grafo el `0` no pueda ser un factor de peso significativo de un arco. A esta matriz también se la denomina *matriz valorada*.

El grafo valorado de la Figura 18.9 se corresponde con un *grafo dirigido con factor de peso*. Si se supone que los vértices se numeran en el orden de `V = {Alicante, Barcelona, Cartagena, Murcia, Reus}`, la matriz de pesos `P` en la que se representa con `0` la no existencia de arco:

$$P = \begin{vmatrix} 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 3 & 6 & 3 \\ 3 & 0 & 0 & 0 & 0 \\ 5 & 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{vmatrix}$$

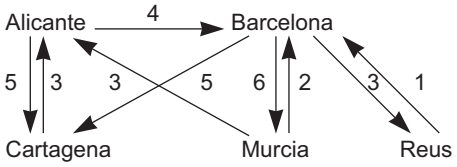


Figura 18.9. Grafo dirigido con factor de peso.

A considerar

La matriz de adyacencia representa los arcos, las relaciones entre un par de nodos de un grafo. Es una matriz de unos y ceros, que indican que dos vértices son adyacentes o no. En un grafo valorado, cada elemento representa el peso de la arista, y por ello se la denomina *matriz de pesos*.

18.2.2. Matriz de adyacencia: `class GrafoMatriz`

A todos los nodos del grafo se les da un nombre (`string`), y se le asigna un número, a partir de `0`, en la matriz de adyacencia. La clase `Vertice` representa un nodo del grafo. Sus tres constructores sobrecargados son los encargados de inicializar los dos atributos protegidos `nombre` y `numVertice`. El primero de ellos lo hace por defecto; el segundo inicializa el nombre y pone como número de vértice `-1;` el tercero inicializa el número de vértice y el nombre. El resto de los métodos de la clase `Vertice` realizan las funciones de acceder o modificar cada uno de los atributos. La función miembro `igual()`, decide si el nombre de un vértice que recibe como parámetro coincide con el del objeto actual.

```
class Vertice
{
protected:
    string nombre;
    int numVertice;
```

```
  public:
    Vertice () {}
    Vertice(string x)
    {     // inicializa el nombre y pone el número de vértice e -1
       nombre = x;
       numVertice = -1;
    }
    Vertice(string x, int n)
    {     // inicializa el nombre y el número de vértice
       nombre = x;
       numVertice = n;
    }
    string OnomVertice()
    {     // retorna el nombre del vértice
        return nombre;
    }
    void PnomVertice(string nom)
    {     // cambia el nombre del vértice
        nombre = nom;
    }
    bool igual(Vertice n)
    {     // decide entre la igualdad de nombres
       return nombre == n.nombre;
    }
    void PnumVertice(int n)
    {     // cambia el número del vértice
       numVertice = n;
    }
    int OnumVertice()
    {     // retorna el número del vértice
       return numVertice;
    }
};
```

La clase `GrafoMatriz` contiene como atributos protegidos su tamaño (número máximo de vértices), el número de vértices actual, la matriz de adyacencia de dimensión variable y un array dimensionable que almacena los distintos *Vértices*. Además, contiene todos los métodos públicos necesarios para acceder o modificar cada uno de sus atributos. Posteriormente, se añadirán métodos que implementan operaciones comunes.

```
class GrafoMatriz
{
    protected:
      int maxVerts;        // máximo numero de vértices
      int numVerts;        // número de vértices actual
      Vertice * verts;     // array de vértices
      int ** matAd;        // matriz de adyacencia
public:
  // métodos públicos de la clase GrafoMatriz
private:
  // métodos privados de la clase GrafoMariz
};
```

El constructor sin argumentos crea la matriz de adyacencia para un máximo de vértices preestablecido que en este caso es *1*; el otro constructor tiene un argumento con el máximo

número de vértices. Para dimensionar la matriz de adyacencia se necesita declarar como tipo predefinido un puntero a entero. La implementación de los métodos públicos es:

```
typedef int * pint;  // para el dimensionamiento de la matriz

GrafoMatriz::GrafoMatriz(int mx)
{
   maxVerts = mx;
   verts = new Vertice[mx] ;        // vector de vértices
   matAd = new pint[mx];            // vector de punteros
   numVerts = 0;
   for (int i = 0; i < mx; i++)
      matAd[i] = new int [mx];      // matriz de adyacencia
}

GrafoMatriz::GrafoMatriz()
{
   maxVerts = 1 ;
   GrafoMatriz(maxVerts);
}
```

Las funciones miembro públicas encargadas de obtener el número de vértices o de cambiar el número de vértices de la clase `GrafoMatriz` son:

```
int OnumeroDeVertices(){return numVerts;}
void PnumeroDeVertices(int n){numVerts = n;}
```

Añadir un vértice

La operación `nuevoVertice()` recibe el nombre de un vértice del grafo, comprueba si el nombre recibido se encuentra en la lista de vértices incrementando en caso negativo el número de vértices y asignándolo a la lista. La comprobación de si el nombre está en lista de vértices es realizado por el método `numVertice()`.

```
void GrafoMatriz::nuevoVertice (string nom)
{
    bool esta = numVertice(nom) >= 0;
    if (!esta)
    {
         Vertice v = Vertice(nom, numVerts);
         verts[numVerts++] = v; // se asigna a la lista.
         // No se comprueba que sobrepase el máximo
    }
}
```

`numVertice()` es un método público de la clase, busca el vértice en el array, retornando `-1` si no lo encuentra:

```
int GrafoMatriz::numVertice(string v)
{
   int i;
   bool encontrado = false;
   for ( i = 0; (i < numVerts) && !encontrado;)
```

```
    {
       encontrado = verts[i].igual(v);
       if (!encontrado) i++ ;
    }
     return (i < numVerts) ? i : -1 ;
}
```

Añadir un arco

El método sobrecargado y público `nuevoArco()` recibe el nombre de los dos vértices que forman el arco y busca en el array, el número de vértice asignado a cada uno de ellos marcando la matriz de adyacencia.

```
void GrafoMatriz::nuevoArco(string a, string b)
{
    int va, vb;
    va = numVertice(a);
    vb = numVertice(b);
    if (va < 0 || vb < 0) throw "Vértice no existe";
    matAd[va][vb] = 1;
}
```

Otra versión sobrecargada del método público recibe, directamente, los números de los vértices que forman el arco y marca la posición de la matriz.

```
void GrafoMatriz::nuevoArco(int va, int vb)
{
    if (va < 0 || vb < 0 || va > numVerts || vb > numVerts)
        throw "Vértice no existe";
    matAd[va][vb] = 1;
}
```

Este método tiene un tercer argumento para *grafos valorados*, que representa el *factor de peso* del arco que se asigna a la matriz. Por ejemplo, en el caso anterior, la versión de la función miembro es:

```
void GrafoMatriz::nuevoArco(int va, int vb, int valor)
{
    if (va < 0 || vb < 0 || va > numVerts || vb > numVerts)
       throw "Vértice no existe";
    matAd[va][vb] = valor;
}
```

Para realizar la entrada completa del grafo en memoria, primero se asignan los vértices y, a continuación, sus arcos. El tiempo de esta operación depende de la densidad del grafo, si se considera un grafo denso el tiempo de ejecución es cuadrático, $O(n^2)$, siendo n el número de nodos.

Adyacente

Determina si dos vértices, v_1 y v_2, forman un arco. Sencillamente, si el elemento de la matriz de adyacencia es `1`. Se escriben dos versiones de la función miembro pública `adyacente()`. La primera tiene los nombres de los vértices como parámetro y la segunda los números de vértice.

```
bool GrafoMatriz::adyacente(string a, string b)
{
    int va, vb;
    va = numVertice(a);
    vb = numVertice(b);
    if (va < 0 || vb < 0) throw "Vértice no existe";
    return matAd[va][vb] == 1;
}

bool GrafoMatriz::adyacente(int va, int vb)
{
    if (va < 0 || vb < 0 || va >= numVerts || vb >= numVerts)
       throw "Vértice no existe";
    return matAd[va][vb] == 1;
}
```

Valor de la matriz de adyacencia

El método público sobrecargado `Ovalor()` se encarga de retornar el valor almacenado en la matriz de adyacencia para los vértices que recibe como parámetro, identificados con el nombre del vértice o con el número asignado. Es muy similar al método `adyacente()`, excepto que en este caso se retorna el propio contenido de la matriz, por lo que puede servir para grafos valorados. Hay que tener en cuenta que esta función es muy importante para el tratamiento de los algoritmos de grafos, ya que retorna el valor de las distintas entradas de la matriz de adyacencia o de pesos para grafos no valorados o valorados respecivamene.

```
int GrafoMatriz::Ovalor(string a, string b)
{
    int va, vb;
    va = numVertice(a);
    vb = numVertice(b);
    if (va < 0 || vb < 0) throw "Vértice no existe”;
       return matAd[va][vb];
}

int GrafoMatriz::Ovalor( int va, int vb)
{
   if (va < 0 || vb < 0 || va >= numVerts || vb >= numVerts)
       throw "Vértice no existe";
   return matAd[va][vb];
}
```

Modificar el valor de la matriz de adyacencia

El método público `Pvalor()` modifica el valor de la posición de la matriz de adyacencia que recibe como parámetro, o bien mediante el nombre o bien mediante el número del vértice. La función miembro es interesante para usarla tanto en grafos valorados como no valorados ya que permite modificar una entrada de la matriz de pesos o de adyacencia.

```
void GrafoMatriz::Pvalor(int va, int vb, int v)
{
    if (va < 0 || vb < 0 || va >= numVerts || vb >= numVerts)
```

```
        throw "Vértice no existe";
    matAd[va][vb] = v;
}

void GrafoMatriz::Pvalor( char *a, char *b, int v)
{
    int va, vb;
    va = numVertice(a);
    vb = numVertice(b);
    if (va < 0 || vb < 0) throw "Vértice no existe";

    matAd[va][vb] = v;
}
```

Vértice asociado a un número entero

El método público `Overtice()` recibe como parámetro una identificación del vértice y retorna todos los atributos del vértice si está almacenado en el grafo. La importancia de esta función miembro radica en permitir obtener toda la información asociada a un vértice almacenada en el grafo, para su posterior consulta. Sólo se codifica la versión que recibe como parámetro el número entero que identifica al vértice, y se deja al lector la versión que recibe como parámetro el nombre del vértice.

```
Vertice Overtice(int va)
{
   if (va < 0 || va >= numVerts)
        throw "Vértice no existe";
   else return verts[va]
}
```

Resulta interesante el método público `Pvertice()` que permite modificar la información de un vértice, para el caso de que el vértice se identifique como un número entero.

```
void Pvertice( int va, Vertice vert)
{
   if (va < 0 || va >= numVerts)
        throw "Vértice no existe";
    else verts[va] = vert;
}
```

18.3. LISTAS DE ADYACENCIA

La representación de un grafo con matriz de adyacencia no es eficiente cuando el grafo es poco denso, (disperso), es decir tiene pocos arcos y, por tanto, la matriz de adyacencia tiene muchos ceros. Para grafos dispersos la matriz de adyacencia ocupa el mismo espacio que si el grafo tuviera muchos arcos (grafo denso). Cuando esto ocurre, se elige la representación del grafo mediante listas enlazadas, denominadas *listas de adyacencia*.

Las listas de adyacencia son una estructura multienlazada formada por una tabla directorio donde cada elemento de la tabla representa un vértice del grafo, del que emerge una lista enlazada con todos sus vértices adyacentes. Esto es, cada lista representa a los arcos con vértice origen el del nodo de la lista directorio, por eso se llama lista de adyacencia.

La Figura 19.10 representa a un grafo dirigido, la representación mediante listas de adyacencia se encuentra en la Figura 19.11. Si por ejemplo se analiza el vértice `5`, es adyacente a los vértices `1, 2 y 4`; por ello su lista de adyacencia consta de tres nodos, cada uno con el vértice destino que forma el arco. También se puede observar que el vértice `4` no es origen de ningún arco, como consecuencia la lista de adyacencia está vacía.

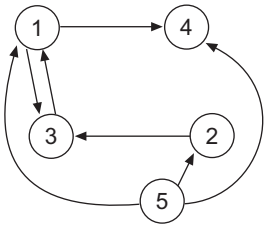


Figura 18.10. Gr afo dirigido.

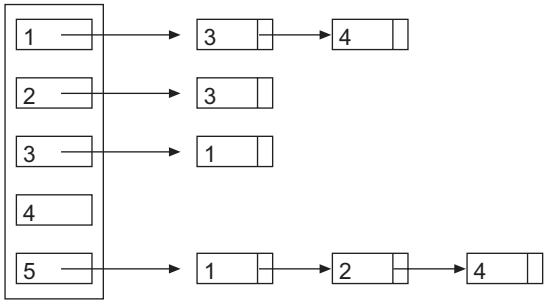


Figura 18.11. Listas de adyacencia del grafo 19.10.

Nota

La implementación de un g rafo con listas de ady acencia puede consultarse en la lectura recomendada del Anexo C del capítulo.

18.4. RECORRIDO DE UN GRAFO

En general, *recorrer una estructura* consiste en visitar (procesar) cada uno de los nodos a partir de uno dado; así se ha realizado el recorrido de un árbol en, por ejemplo, *preorden* partiendo del nodo raíz. De igual forma, *recorrer un grafo* consiste en visitar todos los vértices alcanzables a partir de uno dado. Muchos de los problemas que se plantean con los grafos exigen examinar las aristas de que consta y procesar los vértices del grafo.

Sea `V` el conjunto de vértices del grafo, los algoritmos de *recorrido* de grafos parten de un nodo, `v`, y mantienen un conjunto de nodos *marcados* (*procesados*), `W`, que inicialmente es el nodo o vértice de partida, `v`. En cada *pasada* del algoritmo se retira un nodo, `w`, del conjunto `W`, se procesa y para cada una de las aristas que tengan como origen el vértice `w`, `(w, u)`, su

vértice adyacente, u, se añade a W si no está marcado. El algoritmo continúa hasta que el conjunto W queda vacío, en ese momento todo vértice no marcado no es accesible desde el nodo de partida v. Si se necesita un recorrido completo, se puede continuar desde cualquier vértice no marcado.

A tener en cuenta

Hay dos formas: de recorrer un grafo: *recorrido en profundidad y recorrido en anchura.* Si el conjunto de nodos *marcados* se trata como una cola, entonces el recorrido es en anchura; si se trata como una pila, el recorrido es en profundidad.

18.4.1. Recorrido en anchura

Se utiliza una cola como estructura en la que se mantienen los vértices marcados que se van a procesar posteriormente. El proceso de los elementos en una cola (*primero en entrar primero en salir*) hace que, a partir del vértice de partida, v, se procesen primero todos los vértices adyacentes a v, después los adyacentes de éstos que no estuvieran ya *marcados* o *visitados*, y así sucesivamente con los adyacentes de los adyacentes.

El orden de visitar los nodos, en el recorrido en anchura, se puede expresar de manera más concisa en estos pasos:

1. *Marcar* el vértice de partida v.
2. *Meter* en la cola el vértice de partida v.
3. Repetir los pasos 4 y 5 hasta que se cumpla la condición *cola vacía*.
4. *Quitar* nodo frente de la cola, w, visitar w.
5. *Meter* en la cola todos los vértices adyacentes a w que no estén marcados, a continuación marcar esos vértices.
6. Fin del recorrido.

Como ejemplo, se va a recorrer en anchura el grafo dirigido de la Figura 18.12. El recorrido se inicia a partir del nodo D, se marca y se mete en la cola. Ahora iterativamente, se quita el nodo frente, se procesa, se mete en la cola sus adyacentes no marcados y se marcan. Los sucesivos elementos de la cola se muestran en la Figura 18.13. En la columna de vértices procesados el vértice en *negrita* es el que se procesa en esa pasada, y en la cola los vértices en cursiva son los que se meten en la cola y son marcados.

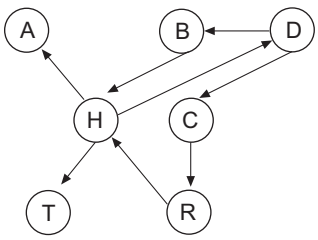


Figura 18.12. Grafo dirigido.

COLA	Vértices procesados
D	
B C	**D**
C *H*	**B**
H *R*	**C**
R *A T*	**H**
A T	**R**
T	**A**
cola vacía	**T**

Figura 18.13. Recorrido en anchura del grafo de Figura 18.12.

El recorrido del grafo en anchura a partir del nodo `D` del grafo 18.12, ha accedido a todos los nodos del grafo, por ello se puede afirmar que todos los vértices del grafo son alcanzables desde el vértice `D`.

18.4.2. Recorrido en profundidad

La búsqueda de los vértices y aristas de un grafo en profundidad persigue el mismo objetivo que el recorrido en anchura: *visitar* todos los vértices del grafo alcanzables desde un vértice dado. La diferencia de este recorrido con el recorrido en anchura es el orden en que se procesan los vértices adyacentes; en este nuevo recorrido el orden viene determinado por la estructura pila.

El recorrido empieza por un vértice `v` del grafo; éste se marca como visitado y se *mete* en la pila. Después se recorre en profundidad cada vértice adyacente a `v` no visitado; así hasta que no haya más vértices adyacentes no visitados. Esta estrategia de examinar los vértices se denomina en profundidad porque la dirección de *"visitar"* es hacia *adelante* mientras que sea posible; al contrario que la búsqueda en anchura que primero visita todos los vértices posibles en *amplitud*.

Como ejemplo, se recorre en profundidad el grafo dirigido de la Figura 18.12. El recorrido se inicia a partir del nodo `D`, se marca y se mete en la pila. Ahora, iterativamente, se quita el nodo cabeza, se procesa, se meten en la pila sus adyacentes no marcados y se marcan. Los sucesivos elementos de la pila se muestran en la Figura 18.14. En la columna de vértices procesados el nodo en *negrita* es el que se *visita* en esa pasada, en la pila los vértices en cursiva son los que se meten en la pila y a la vez son marcados.

PILA	Vértices procesados
D	
B C	**D**
B *R*	**C**
B *H*	**R**
B *A T*	**H**
B A	**T**
B	**A**
pila vacía	**B**

Figura 18.14. Recorrido en profundidad del grafo de Figura 18.12.

18.4.3. Implementación: clase `RecorreGrafo`

Los algoritmos comunes con grafos pueden implementarse como métodos de la clase `GrafoAdcia`[1] o de `GrafoMatriz`. Sin embargo, se ha preferido agrupar estos métodos en una nueva clase `RecorreGrafo`. Los algoritmos de recorrido en profundidad y en anchura se encuentran como métodos públicos en la clase `RecorreGrafo`, recibiendo como argumento el grafo (con matriz o con listas de adyacencia) y el vértice de partida del recorrido y realizando el recorrido del grafo independientemente de la implementación realizada o si la identificación del vértice se realiza con un entero o con un nombre.

```
class RecorreGrafo
{
    public:
      int CLAVE ; // Clave para vértice no marcado
      RecorreGrafo(){CLAVE = 0xffff;} //Clave vértice no marcado
      //operaciones con los recorridos
      int * recorrerAnchura(GrafoMatriz g, string org);
      int * recorrerAnchura(GrafoAdcia g,string org);
      int *recorrerProf(GrafoMatriz g, string org);
      int *recorrerProf(GrafoAdcia g, string org);
      // sobrecarga de los métodos pasando el número de vértice
      int * recorrerAnchura(GrafoMatriz g, int v);
      //...
};
```

Recorrido en anchura

El recorrido en anchura utiliza una cola en la que se guardan los vértices adyacentes al que se ha procesado. Para determinar si un vértice está o no marcado se pueden seguir varias alternativas, una de ellas utiliza el array `m[]`, de tantas posiciones como vértices y con una correspondencia directa entre índice y número de vértice, para indicar si un nodo está marcado. El estado de nodo `i` *no marcado* se establece con una `clave` previamente determinada, y si está marcado tiene un número finito.

Al recorrer el grafo se puede obtener cierta información relativa a los vértices. En la codificación que se realiza a continuación, se guarda en `m[i]` el número mínimo de aristas para cualquier camino desde el vértice de partida hasta el vértice `i`. El vértice de partida, `v`, se inicializa `m[v] = 0` ya que los caminos parten desde ese vértice; las otras posiciones de `m[i]` se inicializan al valor `clave` que expresa *vértice no marcado.* Cada vez que un vértice `w` es añadido al recorrido, si `(u, w)` es la arista que lo incluye entonces se hace `m[w] = m[u] + 1` ya que el camino a `w` tiene una arista más que el que lleva a `u`. La implementación que se presenta está hecha para un grafo representado por matriz de adyacencia y cuyo vértice origen viene dado por un número entero `u`. Si el vértice viene identificado por su nombre (`string`) basta con buscar previamente el número de vértice asociado, tal y como se hace en la implementación de la segunda versión de la función miembro `recorreAnchura()` sobrecargada. La realización con listas de adyacencia es exactamente la misma que la realizada con matriz de adyacencia ya que los métodos de ambas implementaciones tienen igual nombre y realizan las mismas funciones, por lo que sólo se diferencian en la declaración de los métodos.

[1] Véanse lecturas recomendadas, Anexo C, donde se resalizan la implementación de un grafo con listas de adyacencia.

El recorrido en anchura necesita de la case `Cola` que debe ser incluida en la codificación. Los métodos mínimos que debe contener son los siguientes:

```
class Cola
{
protected:
     // atributos
public:
  void insertar( int v);  // añade v a la cola
  bool colaVacia();       // decide si la cola está vacía
  int quitar(){};         // borra y retorna el primer elemento
     // resto de métodos
};

int * RecorreGrafo::recorrerAnchura(GrafoMatriz g, int v)
{
  int w;
  Cola cola;
  int * m;
  if (v < 0 || v > g.OnumeroDeVertices())
      throw "Vértice no existe";

  m = new int[g.OnumeroDeVertices()];
  // inicializa los vértices como no marcados
  for (int i = 0; i < g.OnumeroDeVertices(); i++)
    m[i] = CLAVE;

  m[v] = 0;    // vértice origen queda marcado

  cola.insertar(v);
  while (!cola.colaVacia())
  {
    int cw;
    cw = cola.quitar();
    w = cw;
    cout << "Vértice " << w << g.Overtice(w).OnomVertice()
         << " visitado";
    // inserta en la cola los adyacentes de w no marcados
    for (int u = 0; u < g.OnumeroDeVertices(); u++)
      if (g.Ovalor(w,u) && m[u] == CLAVE)
      {
                // se marca vértice u con número de arcos hasta el
        m[u] = m[w] +1;
        cola.insertar(u);
       }
  }
  return m;
}
```

La versión para el caso de que la identificación del vértice se realice con el nombre es:

```
int * RecorreGrafo::recorrerAnchura(GrafoMatriz g, string org)
{
  int v = g.numVertice(org);
```

```
  if (v < 0) throw " Vértice origen no existe";
  recorrerAnchura(g,v);
}
```

Para visitar todos los vértices del grafo, una vez que ha terminado el recorrido a partir de uno dado, `v`, hay que buscar si queda algún vértice sin marcar, en cuyo caso se vuelve a recorrer a partir de él y así, sucesivamente, hasta que todos los vértices estén marcados.

Recorrido en profundidad

La implementación utiliza una pila para establecer el orden de recorrido. También se podría realizar una implementación recursiva para evitar la utilización de la pila. Como en el recorrido en anchura, los vértices ya visitados se marcan en el array `m[](`pero ahora no representa el camino mínimo de v al vértice, sino el número de aristas del camino que se ha obtenido en el recorrido en profundidad). El grafo está representado mediante listas de adyacencia. Puede observarse que la implementación realizada es similar a la del recorrido en anchura cuando se hizo con el grafo implementado con matriz de adyacencia. Sólo se diferencia en el uso de una pila(propio del recorrido en profundidad), y que el parámetro `g` es ahora un objeto de la clase `GrafoAdcia`[2], en lugar de serlo de la clase `GrafoMatriz`. Se incluye, además, la declaración mínima de la clase `Pila`, que se supone ya implementada y que debe incluirse.

```
class Pila
{
protected:
  // atributos
public:
  void insertar( int v);   // añade v a la pila
  bool pilaVacia();        // decide si la pila está vacía
  int quitar(){};          // borra y retorna el primer elemento
                                                  // resto de métodos
};

int * RecorreGrafo::recorrerProf(GrafoAdcia g, int v)
{
     int w;
     Pila pila;
     int * m;
        if (v < 0 || v > g.OnumeroDeVertices())
         throw "Vértice no existe";

     m = new int[g.OnumeroDeVertices()];
     // inicializa los vértices como no marcados
     for (int i = 0; i < g.OnumeroDeVertices(); i++)
       m[i] = CLAVE;
     m[v] = 0;       // vértice origen queda marcado
     pila.insertar(v);
     while (!pila.pilaVacia())
     {
       int cw;
       cw = pila.quitar();
       w = cw;
       cout << "Vértice " << w << g.Overtice(w).OnomVertice()
```

[2] Véanse lecturas recomendadas: Anexo C.

```
            << " visitado";
              // inserta en la pila los adyacentes de w no marcados
       for (int u = 0; u < g.OnumeroDeVertices(); u++)
         if (g.Ovalor(w,u) && m[u] == CLAVE)
         {
                // se marca vértice u con número de arcos hasta el
           m[u] = m[w] +1;
           pila.insertar(u);
         }
     }
     return m;
}
int * RecorreGrafo::recorrerProf(GrafoAdcia g, string org)
{
    int v = g.numVertice(org);
    if (v < 0) throw " Vértice origen no existe”;
    recorrerProf(g,v);
}
```

18.5. CONEXIONES EN UN GRAFO

Al modelar un conjunto de objetos y sus relaciones mediante un grafo, una de las cuestiones, que generalmente interesa saber es si desde cualquier vértice se puede acceder al resto de los vértices del grafo, es decir, si todos los vértices están conectados, o simplemente, si el grafo es conexo. Para un grafo dirigido, la conectividad entre todos los vértices se denomina: *grafo fuertemente conexo*.

18.5.1. Componentes conexas de un grafo

Un grafo no dirigido `G` es conexo si existe un camino entre cualquier par de vértices que forman el grafo. En el caso de que el grafo no sea conexo resulta interesante determinar aquellos subconjuntos de vértices que mutuamente están conectados, esto es, las componentes conexas del mismo. El algoritmo para determinar las componentes conexas de un grafo `G` se basa en el recorrido del grafo (en *anchura* o en *profundidad)*. Los pasos a seguir son:

1. Realizar un recorrido del grafo a partir de cualquier vértice `w`.
2. Si en el recorrido se han *marcado* todos los vértices, entonces el grafo es conexo.
3. Si el grafo no es conexo, los vértices marcados forman una componente conexa.
4. Se toma un vértice no marcado, `z`, y se realiza de nuevo el recorrido del grafo a partir de `z`. Los nuevos vértices marcados forman otra componente conexa.
5. El algoritmo termina cuando todos los vértices han sido marcados (visitados).

18.5.2. Componentes fuertemente conexas de un grafo

Un grafo dirigido fuertemente conexo es aquél en el cual existe un camino entre cualquier par de vértices del grafo. De no ser fuertemente conexo se pueden determinar componentes fuertemente conexas del grafo. La Figura 18.15 muestra un grafo dirigido y sus dos componentes fuertemente conexas.

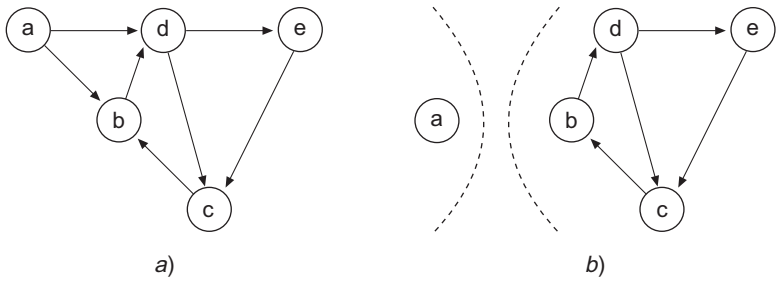


Figura 18.15. *a*) Grafo dirigido. *b*) Componentes fuertes del grafo.

El recorrido en profundidad o en anchura de un grafo a partir de un vértice dado permite diseñar un algoritmo para encontrar si un grafo es fuertemente conexo, o en su caso determinar las componentes fuertemente conexas. A continuación se indican los pasos que sigue:

1. Obtener el conjunto de vértices descendientes del vértice de partida `v`, `D(v)`, incluido el propio vértice `v`.
2. Obtener el conjunto de ascendientes de `v`, `A(v)`, incluido el propio vértice v.
3. Los vértices comunes del conjunto de descendientes y ascendientes de v: `D(v)` $\cap$ `A(v)`, es el conjunto de vértices que forman la componente fuertemente conexa a la que pertenece `v`. Si este conjunto es el conjunto de todos los vértices del grafo, entonces es fuertemente conexo.
4. Si no es un grafo fuertemente conexo se selecciona un vértice cualquiera, `w`, que no esté en ninguna componente fuerte de las encontradas (`w` $\notin$ `D(v)` $\cap$ `A(v)`) y se procede de la misma manera, es decir, se repite a partir del paso1 hasta obtener todas las componentes fuertes del grafo.

Para buscar los vértices descendientes de `v` se realiza un recorrido en profundidad o en anchura del grafo a partir de `v`. Los vértices que son alcanzados se guardan en el conjunto `D`.

Los vértices ascendientes de `v` son aquellos desde los que se puede alcanzar a `v`. Por consiguiente, se obtiene el grafo inverso, cambiando el sentido de los arcos, a continuación se recorre en profundidad o en anchura el grafo inverso a partir de `v`. Los vértices alcanzados en el recorrido son los ascendientes de `v`.

Al recorrer el grafo de la Figura 18.15 en profundidad, a partir del vértice `d`, el conjunto de vértices que alcanza: `{d, c, b, e}`. Repitiendo el recorrido en el grafo inverso, Figura 18.16, se obtiene los vértices ascendientes: `{d, b, c, e}`. Los vértices comunes: `(d, b, c, e)`, forman una componente fuertemente conexa.

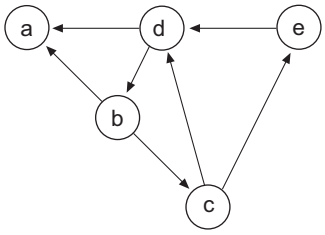


Figura 18.16. Grafo inverso del de la Figura 18.15.

EJERCICIO 18.1. Se tiene un g rafo dirigido de n nodos , representado por su matr iz de adyacencia, A. Se quiere determinar si el grafo es fuertemente conexo, o bien en el caso de que no lo sea, encontrar las componentes fuertemente conexas. Las compones fuertes se mostrarán por pantalla.

El algoritmo recorre el grafo a partir de un vértice, y también recorre el grafo inverso a partir del mismo vértice. El método `grafoInverso()` crea el grafo inverso, cambiando la dirección de los arcos originales.

Se parte de cualquier vértice, por ejemplo el primero, para encontrar el conjunto de vértices que están interconectados. Si son todos, el grafo es fuertemente conexo; en caso contrario, se repite el proceso a partir de un vértice que no esté formando parte de componentes anteriores. El recorrido en anchura o profundidad a partir de `v` encuentra los vértices descendientes de `v`; el recorrido se repite a partir del mismo vértice, pero con el grafo inverso, los vértices marcados forman los vértices ascendientes de `v`. Los vértices marcados en ambos recorridos forman una componente fuerte del grafo.

Al haber una correspondencia biunívoca entre el número de vértice y los índices de los arrays de vértices, puede utilizarse dos array paralelos `descentientes[]` y `ascendientes[]`, para almacenar los vértices ascendientes y descendientes de un vertice `w`, por lo que si un vértice `i` está presente en la componente fuertemente conexa activa pone a `true` la misma posición de ambos array paralelos. De esa forma se guarda el vértice en el correspondiente conjunto haciendo que la posición sea *true*. Esto facilita la operación de intersección (vértices comunes) ya que, simplemente, si se cumple que es *`true`* el valor de `ascendientes[i] && descendientes[i]` el nodo `i` pertenece a ambos conjuntos. Además, en el array `bosque[]` se marcan los vértices que ya están formando parte de una componente fuertemente conexa con el valor de `1`, y si no se encuentran con el valor de `0`. De esta forma, se tiene que se habrán obtenido todas las componentes fuertemente conexas del grafo cuando el array `bosque` tenga todas sus posiciones a `1`. El método `todosArboles()` devuelve un vértice que todavía no forma parte de las componentes fuertemente conexas, explorando el array `bosque`. El proceso termina cuando devuelve el valor clave `–1`.

Código fuente en página web del libro.

18.6. MATRIZ DE CAMINOS. CIERRE TRANSITIVO

Dado un grafo `G`, un camino de longitud `n` desde el vértice v_0 hasta el vértice v_n es una secuencia de `n+1` vértices $v_0, v_1, v_2, \ldots, v_n$ tal que el vértice inicial es v_0, el vértice final v_n y son adyacentes $(v_i, v_{i+1}) \in A(arcos) para\ 0 \le i \le n$. Encontrar caminos entre un par de vértices, de una determinada longitud es una tarea relativamente sencilla, aunque poco eficiente, si se tiene la matriz de adyacencia del grafo.

Si se considera que la matriz de adyacencia, `A`, es de tipo `bool`, la expresión $A_{i,k}$ `&&` $A_{k,j}$ es *verdadera si y sólo si* los valores de ambos operandos lo son. Esta hipótesis implica que hay un arco desde el vértice `i` al vértice `k` y otro desde el vértice `k` al `j`. También se puede afirmar, que si $A_{i,k}$ `&&` $A_{k,j}$ es *verdadera* existe un camino de longitud `2` desde el vértice `i` al `j`. Ese sencillo razonamiento se puede ampliar a la siguiente expresión:

$(A_{i1}$ && $A_{1j})$ || $(A_{i2}$ && $A_{2j})$ || ... ||$(A_{in}$ && $A_{nj})$

Que la expresión sea *verdadera* implica que hay, al menos, un camino de longitud 2 desde el vértice i al vértice j que pase por el vértice 1, o por el 2 ... o por el vértice n. Si ahora se cambia el operador && por el *producto* y || por *suma*, la expresión es el elemento A_{ij} de la matriz producto A^2. La conclusión es inmediata: los elementos (A_{ij}) de la matriz A^2 son distintos de cero si existe un camino de longitud 2 desde el vértice i al vértice j, $\forall i,j = 1..n$.

El razonamiento se puede extender para encontrar caminos de longitud 3; realizando el producto matricial $A^2 x A = A^3$ se encuentran caminos de longitud 3 entre cualquier par de vértices del grafo.

A tener en cuenta

Una manera de encontrar los caminos de longitud m entre cualquier par de vértices de un grafo es mediante el producto matricial de A^{m-1} por la matriz de adyacencia A.

A continuación se siguen los mismos razonamientos para obtener caminos del grafo de la Figura 18.17, cuya matriz de adyacencia es la siguiente:

$$A = \begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

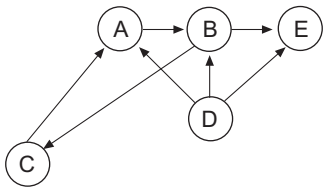


Figura 18.17. Grafo dirigido de 5 vértices.

El producto matricial AxA, considerando todo valor distinto de 0 como 1 (*verdadero*):

$$A^2 = \begin{vmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Que, por ejemplo, A_{15} tenga el valor 1 significa que hay un camino de longitud 2, desde A – E, formado por los arcos: (A → B → E).

De igual manera, el producto matricial $A^2 x A$:

$$A^3 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Si se observa A_{41}, su valor es `1` ya que hay un camino longitud `3` desde `D – A`, formado por los arcos: `(D → B → C → A)`.

Realizando el producto matricial, el valor almacenado en cualquier posición, A_{ij}, no sólo indica si hay camino de longitud `m`, sino, además, el número de caminos de dicha longitud entre los vértices `i` y `j`.

A recordar

La forma de obtener el número de caminos de longitud k entre cualquier par de vértices de un grafo es obtener el producto matricial A^2, A^3 ... A^k. Entonces el elemento $A_k(i,j)$ contiene el número de caminos de longitud k desde el vértice i hasta el vértice j.

A tener en cuenta

La eficiencia del algoritmo para encontrar el número de caminos de longitud k es muy pobre. El producto matricial se realiza con tres bucles anidados, complejidad cúbica $O(n^3)$, además, este producto se realiza k-1 veces.

18.6.1. Matriz de caminos y cierre transitivo

Sea `G` un grafo con `n` vértices, la matriz de caminos de `G` es la matriz `P` de `n*n` elementos, definida:

$$P_{ij} = \begin{cases} 1 & \textit{si hay un camino desde } v_i \text{ a } v_j \\ 0 & \textit{si no hay camino desde } v_i \text{ a } v_j \end{cases}$$

Se encuentra la siguiente relación entre la matriz de caminos `P`, la matriz de adyacencia `A` y las sucesivas potencias de `A` para encontrar los caminos de longitud `m`: dada la matriz $B_n = A + A^2 + A^3 + ... + A^n$, la matriz de caminos `P` es tal que un elemento $P_{ij} = 1$ *si y sólo si* $B_n(i,j) >= 1$ y en otro caso $P_{ij} = 0$.

El cierre transitivo o contorno transitivo de un grafo `G` es otro grafo, `G'`, con los mismos vértices y cuya matriz de adyacencia es la matriz de caminos del grafo `G`.

18.7. ORDENACIÓN TOPOLÓGICA

Una de las aplicaciones de los grafos es modelar las relaciones que existen entre las diferentes tareas, *hitos*, que deben de terminarse para finalizar un proyecto. Entre las tareas hay relaciones

de precedencia, una tarea r precede a la tarea t si es necesario que se complete r para poder empezar t. Estas relaciones de precedencia se representan mediante un grafo dirigido en el que los vértices son las tareas o *hitos* y hay una arista del vértice r al t si el inicio de la tarea t depende de la terminación de r. Una vez se dispone del grafo interesa obtener una planificación de las tareas para poder realizarlas que constituyen el proyecto, en definitiva, encontrar la *ordenación topológica* de los vértices que forman el grafo.

El grafo que representa estas relaciones de precedencia es un grafo dirigido *acíclico*, de tal forma que si hay un camino de u a v, entonces, en la ordenación topológica v es posterior a u. Es inmediato que el grafo no puede tener ciclos cuando representa relaciones de precedencia, de haber un ciclo significaría que si r y t son vértices del ciclo, r depende de t y a su vez t depende de la terminación de r.

A tener en cuenta

La ordenación topológica se aplica sobre grafos dirigidos sin ciclos. Es una ordenación lineal, tal que si v es anterior a w entonces hay un camino de v a w. La ordenación topológica no se puede realizar en grafos con ciclos.

La Figura 18.18 muestra un grafo acíclico que modela la estructura de *prerequisitos* de 8 cursos. Una arista cualquiera (r, s) significa que el curso r debe de completarse antes de empezar el curso s. Por ejemplo en la Figura 18.18, el curso M21 se puede empezar sólo cuando terminen los cursos E11 y T12; en ese sentido E11 y T12 son *prerequisitos* de M21.

Una ordenación topológica de estos cursos es cualquier secuencia de cursos que cumple los requerimientos (*prerequisitos*). Entonces, para un grafo dirigido acíclico no tiene por qué existir una única ordenación topológica.

Del grafo de requisitos de la Figura 18.18 se obtienen estas ordenaciones topológicas:

```
E11 - T12 - M12 - C22 - R23 - S31 - S32 - T41
T12 - E11 - R23 - C22 - M21 - S31 - S32 - T41
```

Un grafo G dirigido y sin ciclos se denomina un *gda* o grafo acíclico. Los *gda,s* son útiles para la representación de *ordenaciones parciales*. Una *ordenación parcial* R en un conjunto C es una relación binaria de precedencia tal que:

1. Para todo u ∈ C, u no está relacionado con sí mismo, u R u es falso, por consiguiente, la relación R es *no reflesiva*.
2. Para todo u, v, w ∈ C, si u R v y v R w entonces u R w. La relación R es *transitiva*.

Tal relación R sobre C se llama *ordenación parcial* de C. La relación de inclusión en conjuntos es un ejemplo de ordenación parcial. Un grafo G sin ciclos se puede considerar un conjunto parcialmente ordenado.

Nota

Se puede probar que un grafo es *acíclico* realizando un recorrido en profundidad, de tal forma que si se encuentra un arco de *retroceso* en el árbol de búsqueda, el grafo tiene al menos un ciclo, y si no se encuentra es acíclico

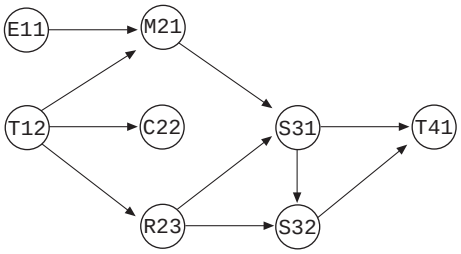


Figura 18.18. Grafo dirigido de prerequisitos.

18.7.1. Algoritmo para obtener una ordenación topológica

El algoritmo, en primer lugar, busca un vértice (una tarea) sin *predecesores* o *prerequisitos*, esto es, que no tenga arcos de entrada (este vértice tiene que existir ya que si no lo hubiera no se podría comenzar por ninguna tarea). El `v`, pasa a formar parte de la ordenación `T`. A continuación, todos los arcos que salen de `v` son eliminados, debido a que el prerequisito `v` ya se ha satisfecho. La estrategia se repite, se toma otro vértice `w` sin arcos incidentes (que llegan a él), se incorpora a la ordenación `T` y se elimina todos los arcos que salen de el. Se itera este proceso hasta completar la ordenación topológica.

Que un vértice `v` no tenga arcos incidentes se expresa con el grado de entrada, `gradoEntrada(v)== 0`. Eliminar los arcos que salen de `v`, implica que `gradoEntrada(w)` de todos los vértices `w` adyacentes de `v` disminuyen en `1`.

Por consiguiente, el algoritmo comienza calculando el grado de entrada de cada vértice del grafo. Los vértices cuyo grado entrada es `0` se guardan en una *Cola*. El elemento frente de la cola, `v`, pasa a formar parte de la ordenación `T`. A continuación, se disminuye en `1` el grado de entrada de los adyacentes de `v`, y aquellos vértices cuyo grado de entrada han pasado a ser 0 se meten en la cola. Este proceso se itera hasta que la cola esté vacía.

A tener en cuenta

Si al aplicar el algoritmo de ordenación topológica hay vértices del grafo que aún no han pasado a formar parte de la ordenación y la cola está vacía, entonces el grafo tiene ciclos.

18.7.2. Implementación del algoritmo de ordenación topológica

La codificación del algoritmo depende de la representación del grafo, con matriz de adyacencia o listas de adyacencia. Si el grafo tiene relativamente pocos arcos, es poco denso, la matriz de adyacencia tiene muchos ceros, es una matriz *esparcida* y, por consiguiente, el grafo se representa con listas de adyacencia. Para grafos dirigidos *densos* se prefiere, por eficiencia, la matriz de adyacencia. Con independencia de la representación, se utiliza una *Cola*, para almacenar los vértices con grado de entrada `0`.

La codificación que a continuación se presenta supone el grafo representado con listas de adyacencia (clase *GrafoAdcia*); los vértices de la ordenación topológica se escribe por pantalla,

y se almacenan en el vector `T[]`. El número de vértices almacenados en el vector se retorna en el parámetro por referencia `j`. También se escribe el método `gradoEntrada()`que calcula el grado de entrada de un vértice de un grafo que recibe como parámetro.

```
int gradoEntrada(GrafoAdcia g, int v)
{
  int cuenta = 0;
  for (int origen = 0; origen < g.OnumeroDeVertices(); origen++)
  {
       if (g.adyacente(origen, v))     // arco incidente a v
         cuenta++;
  }
  return cuenta;
}

// Método para obtener una ordenación topológica.
// Muestra los vértices que pasan a formar parte de la
// ordenación, y se almacenan T[]. El número de vértices lo da en j

void ordenTopologica(GrafoAdcia g, int T[], int &j)
{
   int *arcosInciden;
   int v, w;
   Cola cola;

   j = 0;
   arcosInciden = new int[g.OnumeroDeVertices()];
   // grado de entrada de cada vértice
   for (v = 0; v < g.OnumeroDeVertices(); v++)
     arcosInciden[v] = gradoEntrada(g, v);
   cout << "\n Ordenación topológica ";

   for (v = 0; v < g.OnumeroDeVertices(); v++)
      if (arcosInciden[v] == 0)
          cola.insertar(v); // vértices con grado de entrada 0

   while (!cola.colaVacia())
   {
       w = cola.quitar();
       cout << " " << w;
       T[j++] = w;
       Lista itl = g.listaAdyc(w);  // lista de adyacencia de g
       Nodo * l = itl.Ol();         // puntero al primer nodo de lista
       // iterado de lista
              // decrementa grado entrada de vértices adyacentes
   while ( l !=NULL) // mientras queden nodos en la lista
     {
       v = l->Oarco().Odestino(); //vértice destino del arco del nodo
       arcosInciden[v]--;
       if (arcosInciden[v] == 0)
             cola.insertar(v);      // vértice con grado de entrada 0
        l = l->Osig();
      }
   }
}
```

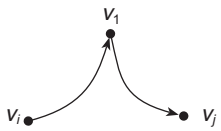
A tener en cuenta

La complejidad del algoritmo *ordenación topológica* de un grafo, representado con listas de adyacencia, es `O(a+n)` siendo a el número de arcos y n el de vér tices. Si la representación del grafo es con una matriz de adyacencia, la complejidad es $O(n^2)$.

18.8. MATRIZ DE CAMINOS: ALGORITMO DE WARSHALL

Este algoritmo calcula la matriz de caminos `P` (también llamado cierre transitivo) de un grafo `G` de `n` vértices, representado por su matriz de adyacencia `A`. La estrategia que sigue el algoritmo consiste en definir, a nivel lógico, una secuencia de matrices *n-cuadradas* P_0, P_1, P_2, P_3 ... P_n. Los elementos de cada una de las matrices $P_k[i,j]$ tienen el valor `0` si no hay camino, `1` si hay camino del vértice `i` y al `j` que no pasa por otro vértice a no ser que estén comprendidos entre `1 ... k` La matriz P_0 es la matriz de adyacencia.

Los elementos de P_1 son igual a `1` si lo son los de P_0, y los que son `0` en P_0 pasan a `1` en P_1 si se puede formar un camino entre el par de vértices, con la ayuda del vértice `1`. En definitiva, $P_1[i,j] = 1$ si lo es $P_0[i,j]$, o bien, hay un camino:



La matriz P_2 se forma a partir de P_1, añadiendo el vértice `2` para poder formar camino entre dos vértices todavía no conectados.

La diferencia entre dos matrices consecutivas P_k y P_{k-1} viene dada por la incorporación del vértice de orden `k`, para estudiar si es posible formar camino del vértice `i` y al `j` con la ayuda del vértice `k`. La descripción de cada matriz:

$$P_0[i,j] = \begin{cases} 1 & \text{si hay un arco del vértice } i \text{ al } j \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_1[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por ningún otro vértice a no ser por el vértice 1.} \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_2[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice a no ser por los que están comprendidos entre los vértices 1 .. 2.} \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_3[i,j] = \begin{cases} 1 & \text{si hay un camino simple de vi a vj que no pasa por otro vértice a no ser por los que están comprendidos entre los vértices 1 .. 3.} \\ 0 & \text{en otro caso.} \end{cases}$$

En cada paso se incorpora un nuevo vértice, el que coincide con el índice de la matriz `P`, a los anteriores para poder formar camino.

$$P_k[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice a no ser por los que están comprendidos entre los vértices } 1 \ldots k. \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_n[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por ningún otro vértice a no ser por los que están comprendidos entre los vértices } 1 \ldots n. \\ 0 & \text{en otro caso.} \end{cases}$$

Según estas definiciones, P_0 es la matriz de adyacencia del grafo. Y al ser un grafo de n vértices, la matriz P_n es la matriz de caminos.

El algoritmo de Warshall encuentra una relación recurrente entre los elementos de la matriz P_k y los elementos de la matriz P_{k-1}. Para que cualquier elemento $P_k[i,j] = 1$ ha de ocurrir uno de estos dos casos:

1. Exista un camino simple de v_i a v_j del que pueden formar parte los vértices de índice 1 al k - 1 (v_1 a v_{k-1}); por consiguiente que el elemento $P_{k-1}[i, j] = 1$.
2. Haya un camino simple de v_i a v_k y otro camino simple de v_k a v_j de los que pueden formar parte los vértices de índice 1 al k-1; por tanto, esto equivale:

$(P_{k-1}[i, k] = 1)$ y $(P_{k-1}[k, j] = 1)$

$v_i \rightarrow \ldots \rightarrow v_j$ (1) *Camino* de v_i a v_j

$v_i \rightarrow \ldots \rightarrow v_k \rightarrow \ldots \rightarrow v_j$ (2) *Camino* $v_i - v_k - v_j$

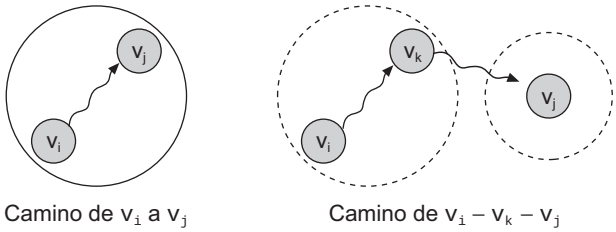


Figura 18.19. Posibilidad de camino de v_i a v_j.

La relación "encontrar los elementos de P_k" puede expresarse a la manera de una operación lógica:

$$P_k[i,j] = P_{k-1}[i,j] \vee (P_{k-1}[i,k] \wedge P_{k-1}[k,j])$$

18.8.1. Implementación del algoritmo de Warshall

Se codifica el algoritmo para un grafo G representado por su matriz de adyacencia. Él método devuelve la matriz de caminos P reservando previamente espacio en memoria. Se inicializa P con la matriz de adyacencia, y se aplica el algoritmo de *programación dinámica* descrito por Warshall.

```
int ** matrizCaminos(GrafoMatriz g)
{
  int n = g.OnumeroDeVertices();
  typedef int *pint;// pint es tipo que apunta a punteros enteros
  int ** P = new pint[n];//P apunta a un array de punteros enteros
    for (int i = 0; i < n; i++)
           P[i]= new int[n]; // P es ahora una matriz de enteros
      // Se dispone de memoria para la matriz de caminos
     // Se obtiene la matriz inicial: matriz de adyacencia
  for (int i = 0; i < n; i++)
     for (int j = 0; j < n; j++)
       P[i][j] = g.adyacente(i,j) ? 1 : 0;

  // se obtienen, virtualmente, a partir de P0, las sucesivas
// matrices P1, P2, P3 ,..., Pn-1, Pn que es la matriz de caminos

  for (int k = 0; k < n; k++) // etapas de programación dinámica
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
            if (P[i][k] * P[k][j])
            //observe el índice k de la programación dinámica
              P[i][j] = 1;
  return P;
}
```

Norma sobre eficiencia

La complejidad del algoritmo de *Warshall* es cúbica, $O(n^3)$ siendo n el número de vértices. Esto hace que el tiempo de ejecución crezca rápidamente par a grafos de relativamente muchos nodos.

18.9. CAMINOS MÍNIMOS CON UN SOLO ORIGEN: ALGORITMO DE DIJKSTRA

Uno de los problemas que se plantean con relativa frecuencia es el de determinar la longitud del camino más corto entre un par de vértices de un grafo. Por ejemplo, determinar la mejor ruta (menor tiempo) para ir desde un lugar a un conjunto de centros de la ciudad. Para este tipo de problemas se considera un grafo dirigido y *valorado* , es decir, cada arco (v_i, v_j)del grafo tiene asociado un coste c_{ij}. La longitud del camino más corto es la suma de los costes de los arcos que forman el camino. Matemáticamente, si $v_1, v_2 \ldots v_k$ es la secuencia de vértices que forman un camino:

$$\textit{Longitud de camino} = \sum_{i=1}^{k-1} C_{i,i+1}$$

se pretende encontrar el camino de longitud o coste mínimo.

Otro problema relativo a los caminos entre dos vértices v_1, v_k, es encontrar aquél de menor número de arcos para ir de v_1 a v_k. En definitiva, encontrar el camino de longitud mínima en un grafo no valorado. Este problema se resuelve con una *búsqueda en anchura* a partir del

vértice de partida (consultar en el Apartado 18.4, la longitud del camino mínimo viene dada en la matriz `m[]`).

El algoritmo que se describe a continuación encuentra el camino mínimo desde un vértice origen al resto de vértices, en un grafo valorado positivamente (con factor de peso positivo). Este algoritmo recibe el nombre de algoritmo de *Dijkstra* en honor del científico que lo expuso.

18.9.1. Algoritmo de Dijkstra

Dado un grafo dirigido `G = (V, A)` valorado y con factores de peso no negativos, el algoritmo de *Dijkstra* determina el camino más corto desde un vértice al resto de los vértices del grafo. Éste es un ejemplo típico de algoritmo *ávido(voraz)* que selecciona en cada paso la solución más optima en aras de resolver el problema. *En cada iteración se come el mejor pedazo.*

Se recuerda que en un grafo valorado, cada arco (v_i, v_j) tiene asociado un coste c_{ij}. De tal forma que si v_0 es el vértice origen, y $v_0, v_1, \ldots, v_k$ es la secuencia de vértices que forman el camino de v_0 a v_k, entonces $\sum_{i=1}^{k-1} c_{i,i+1}$ es la longitud del camino.

El algoritmo *voraz* de *Dijkstra* considera dos subconjuntos de vértices, `F` y `V – F`, siendo `V` el conjunto de todos los vértices. Se define la función *distancia(v)*: coste del camino mas corto del origen `s` a `v` que pasa solamente por los vértices de `F` (todos los vértices están en `F` excepto posiblemente el último del camino). Entonces, el primer vértice que se añade a `F` es el origen `s`. En cada paso se selecciona un vértice `v` de `V – F` cuya *distancia(v)* es la menor. El vértice v se *marca* para indicar que ya se conoce el camino mas corto de s a v. Ahora, todos los vértices que no estén en `F` actualizan el nuevo valor de distancia(v). Se siguen *marcando* nuevos vértices de `V – F` hasta que lo estén todos, en ese momento el algoritmo termina ya se conoce el camino más corto del vértice origen s al resto de los vértices.

Par realizar esta estrategia *voraz*, el array `D` almacena el camino más corto (coste mínimo) desde el origen `s` a cada vértice del grafo. Se parte con el vértice origen `s` en `F` y los elementos $D_{(i)}$, del array `D`, con el coste o peso de los arcos (s, v_i); si no hay arco de s a i se pone *coste* ∞ . En cada paso se agrega algún vértice `v` de los que todavía no están en `F`, esto es de `V - F`, que tenga el menor valor `D(v)`. Además, se actualizan los valores `D(w)` para aquellos vértices `w` que todavía no están en `F`, para que almacene el valor de `distancia(v)`. Para conseguirlo basta con hacer `D(w) = D(v)+`$c_{v,w}$ cuando este nuevo valor de `D(w)` sea menor que el anterior. De esta forma se asegura que la incorporación de `v` implica que hay un camino de `s` a `v` cuyo coste mínimo es `D(v)`.

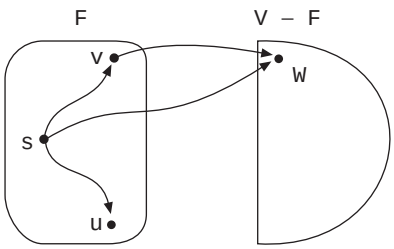


Figura 18.20. Al incorporar v la distancia de s a w , D[w] puede mejorar.

Para reconstruir el camino de *coste mínimo* que lleva de `s` a cada vértice `v` del grafo se almacena, para cada uno de los vértices, el último vértice que hizo el coste mínimo. Entonces, asociado a cada vértice resulta de interés dos datos, el *coste mínimo* y el último vértice del camino con el que se minimizó el *coste*.

El ejemplo 18.1 hace un seguimiento de los sucesivos pasos que sigue este algoritmo para encontrar los caminos mínimos desde un origen al resto de vértices del grafo.

EJEMPLO 18.1. Dado el grafo dirigido y con pesos no negativos, de la Figura 18.21, se quiere calcular el coste mínimo de los caminos desde el vértice 1 a los otros vértices del grafo, aplicando el algoritmo de Dijkstra.

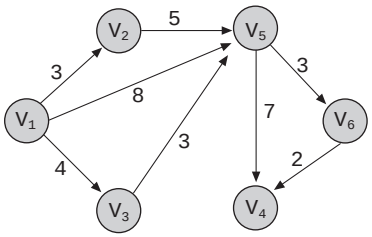


Figura 18.21. Grafo dirigido con factor de peso positivo.

La matriz de pesos de los arcos, considerando peso ∞ cuando no existe arco:

$$C = \begin{vmatrix} \infty & 3 & 4 & \infty & 8 & \infty \\ \infty & \infty & \infty & \infty & 5 & \infty \\ \infty & \infty & \infty & \infty & 3 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 7 & \infty & 3 \\ \infty & \infty & \infty & 2 & \infty & \infty \end{vmatrix}$$

Los sucesivos valores del conjunto `F` (vértices *marcados*), vértice `v` incorporado y el vector distancia, `D`, que se obtienen en cada paso se representan en forma tabular:

Paso	F	v	D[2]	D[3]	D[4]	D[5]	D[6]
Inicial	1		3	4		8	
1	1,2	2	3	4		8	
2	1,2,3	3	3	4		7	
3	1,2,3,5	5	3	4	14	7	10
4	1,2,3,5,6	6	3	4	12	7	10
5	1,2,3,5,6,4	4	3	4	12	7	10

Ejemplo, el camino mínimo de v_1 a v_6 es `10`, la secuencia de vértices que hacen el camino mínimo: $v_1 - v_3 - v_5 - v_6$.

18.9.2. Codificación del algoritmo de Dijkstra

Se define la clase `CaminoMinimo` para implementar el algoritmo. La codificación de la clase considera que el grafo está representado con una matriz de pesos o costes (objeto de la clase `GrafoMatriz`). La información almacenada en las distintas entradas de la matriz de pesos se obtiene mediante el método `Ovalor()` de la clase `GrafoMatriz` que retorna el coste de cada arco, bien el valor de `INFINITO = 0xFFFF` (constante con un valor inalcanzable) en el caso de que no haya arco. Como cada vértice tiene asociado dos informaciones, último vértice en el camino mínimo y la distancia mínima, la clase `CaminoMinimo` que contiene la función que calcula los caminos mínimos, tiene como atributos protegidos los arrays `último` y `D` para almacenarlos. Son atributos privados de la clase el vértice origen y el número de vértices del grafo actual. Los métodos públicos (se explicarán posteriormente) de la clase `CaminoMinimo` son: `dijkstra()`, `OdistanciaMinima()`, `recuperaCamino()` y `Oultimo()`. La clase contiene además un método privado `minimo()`.

```
class CaminoMinimo
{
 protected:
    int * ultimo;              // array de predecesores
    int * D;                   // array de distancias mínimas

 private:
    int s, n;                  // vértice origen y número de vértices

 public:
      CaminoMinimo (GrafoMatriz g, int origen); // constructor
      void Dijkstra(GrafoMatriz g, int origen); // dijkstra
      void recuperaCamino(int v);
      int * OdistanciaMinima();
      int *Oultimo();

    private:
      int minimo( bool F[]); // metodo privado usado por dijkstra
};
```

El constructor público `CaminoMinimo`(), realiza la tarea de asignar memoria a los atributos `ultimo` y `D` así como inicializar los atributos privados.

```
CaminoMinimo::CaminoMinimo (GrafoMatriz g, int origen)
{
       n = g.OnumeroDeVertices();
       s = origen;
       ultimo = new int[n];
       D = new int[n];
}
```

El método público `Dijkstra()`, se encarga de calcular los caminos mínimos del vértice origen al resto de los vértices. Para almacenar los vértices marcados, la función miembro usa un array lógico `F` al que se reserva memoria dinámicamente. Los valores de cada una de las posiciones de este array son `true` en el caso de que ya se conozca la distancia mínima del `origen` al vértice o bien `false` en otro caso.

```
void CaminoMinimo::Dijkstra(GrafoMatriz g, int origen)
{
     bool * F;
     F = new bool [n];
     // valores iniciales
     for (int i = 0; i < n; i++)
     {
       F[i] = false;
       D[i] = g.Ovalor(s, i);
       ultimo[i] = s;
     }
     F[s] = true; D[s] = 0;     //Marca origen e inicializa distancia
     // Pasos para marcar los n-1 vértices. Algoritmo voraz
     for (int i = 1; i < n; i++)
     {
       int v = minimo(F);
       //selecciona vértice no marcado de menor distancia
       F[v] = true;
       // actualiza distancia de vértices no marcados
       for (int w = 0; w < n; w++)
         if (!F[w])
           if (D[v] + g.Ovalor(v, w) < D[w])
           {
               D[w] = D[v] + g.Ovalor(v, w);
               ultimo[w] = v;
           }
     }
}
```

El método privado `minimo()` es usado por `dijkstra()` para encontrar el índice del vértice del grafo cuya distancia al `origen` sea mínima (elegido de entre los no marcados).

```
int CaminoMinimo::minimo( bool F[])
  {
     double mx = 0xFFFF; // valor de infinito
     int v;
     for (int j = 0; j < n; j++)
       if (!F[j] && (mx >= D[j]))
       {
         mx = D[j];
         v = j;
       }
     return v;
}
```

El tiempo de ejecución del algoritmo de `Dijkstra()` es cuadrático, $O(n^2)$, debido al bucle que realiza la llamada al método `minimo()` de complejidad lienal `n` (número de vértices). Ahora bien, en el caso de que el número de arcos, `a`, fuera mucho menor que n^2, el tiempo de ejecución mejora representando el grafo con listas de adyacencia. Entonces, puede obtenerse un tiempo `O(a*log n)`, usando la estructura de datos avanzada de montículo.

Recuperación de los caminos

Los sucesivos vértices que conforman el camino mínimo se obtienen *"volviendo hacia atrás"*, es decir, desde el *último* que hizo que el camino fuera mínimo, al *último del último* y así suce-

sivamente llegar al origen. Las llamadas recursivas del método público `recuperaCamino()` (de la clase `CaminoMinimo`) que se escribe a continuación permiten, fácilmente, volver atrás y reconstruir el camino.

```
void CaminoMinimo::recuperaCamino(int v)
{
     int anterior = ultimo[v];
     if (v != s)
     {
         recuperaCamino(anterior);      // vuelve al último del último
         cout << v << " V <--" ;
     }
     else
         cout << s;
}
```

Los métodos públicos que se escriben a continuación se encargan de poder obtener para su tratamiento los arrays `D` y `ultimo`.

```
int * CaminoMinimo::OdistanciaMinima(){ return D;}
int * CaminoMinimo::Oultimo(){ return ultimo;}
```

18.10. TODOS LOS CAMINOS MÍNIMOS: ALGORITMO DE FLOYD

En algunas aplicaciones resulta interesante determinar el camino mínimo entre todos los pares de vértices de un grafo dirigido y valorado. Considerando como vértice origen cada uno de los vértices del grafo, el algoritmo de *Dijkstra* resuelve el problema. Hay otra alternativa, más elegante y más directa que usa la programación dinámica, propuesta por *Floyd* que recibe el nombre de *algoritmo de Floyd*.

De nuevo, el grafo está representado por la matriz de pesos, de tal forma que todo arco (v_i, v_j) tiene asociado un peso c_{ij}; si no existe arco $c_{ij} = \infty$. Además, cada elemento de la diagonal, c_{ii}, se hace igual a `0` (en el caso de que no se realice esta inicialización, el algoritmo obtiene en la diagonal los ciclos o bucles de longitud mínima). El *algoritmo de Floyd* determina una nueva matriz, `D`, de $n_x n$ elementos tal que cada elemento, D_{ij}, contiene el coste del camino mínimo de v_i a v_j.

El algoritmo tiene una estructura similar al algoritmo de *Warshall* para encontrar la matriz de caminos. Se generan consecutivamente las matrices $D_1, D_2, \ldots, D_k, \ldots, D_n$ a partir de la matriz D_0 que es la matriz de pesos. En cada paso se incorpora un nuevo vértice y se estudia si con ese vértice se puede mejorar los caminos en cuanto a ser más cortos. El significado de cada matriz:

D_0[i,j] = { cij *coste*(peso) del arco del vértice i al vértice j
∞ si no hay arco.

D_1[i,j] = minimo(D_0[i,j],D_0[i,1]+D_0[1,j])
Menor de los *costes* entre el *anterior* camino mínimo de `i` a `j` y la suma de *costes* de caminos de `i` a `1` y de `1` a `j`.

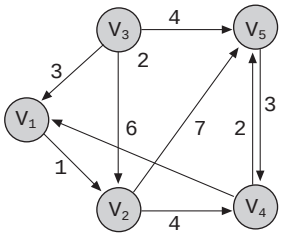
D_2[i,j] = minimo(D_1[i,j],D_1[i,2]+D_1[2,j])
Menor de los *costes* entre el *anterior* camino mínimo de `i` a `j` y la suma de *costes* de caminos de `i` a `2` y de `2` a `j`.

En cada paso se incorpora un nuevo vértice para determinar si hace el camino menor entre un par de vértices.

D_k[i,j] = minimo(D_{k-1}[i,j],D_{k-1}[i,k]+D_{k-1}[k,j])
Menor de los *costes* entre el *anterior* camino mínimo de `i` a `j` y la suma de *costes* de caminos de `i` a `k` y de `k` a `j`.

De forma recurrente, se añade en cada paso un nuevo vértice para determinar si se consigue un nuevo camino mínimo, hasta llegar al último vértice y obtener la matriz D_n que es la matriz de caminos mínimos del grafo.

EJEMPLO 18.2. La Figura 18.22 muestra un grafo dirigido con factor de peso y la correspondiente matriz de pesos. Aplicar el algoritmo de Floyd para obtener, en los sucesivos pasos, la matriz de caminos mínimos.



a)

$$C = \begin{vmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & \infty & 4 \\ 6 & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

b)

Figura 18.22. *a*) Grafo dirigido valorado de 5 vértices. *b*) Matriz de pesos del grafo.

El grafo consta de cinco vértices, por ello se forman cinco matrices: `D1`, `D2`, `D3`, `D4`, `D5` que es la matriz de caminos mínimos. La matriz `D0` es la matriz de pesos `C`.

$$D_1 = \begin{vmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & \infty & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice `1` ha cambiado D_1 `(4,2)` ya que `C(4,1) + C(1,2) < C(4,2)`.

$$D_2 = \begin{vmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice `2` ha habido varios cambios, así D_1`(1,4)`: D_1`(1,2) + `D_1`(2,4) < `D_1`(1,4)`.
También cambian D_1`(1,5)`, D_1`(2,5)` y D_1`(3,4)`.

$$D_3 = \begin{vmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 3 no hay cambios; al vértice 3 no llega ningún arco.

$$D_4 = \begin{vmatrix} 0 & 1 & \infty & 5 & 7 \\ 10 & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ 9 & 10 & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 4 cambian los elementos: $D_3(1,5)$, $D_3(2,1)$, $D_3(5,1)$ y $D_3(5,2)$.

$$D_5 = \begin{vmatrix} 0 & 1 & \infty & 5 & 7 \\ 10 & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ 9 & 10 & \infty & 3 & 0 \end{vmatrix}$$

La incorporación del vértice 5 no genera cambios.

La matriz D5 es la matriz de caminos mínimos. Dado cualquier par de vértices se puede conocer la longitud del camino más corto que hay entre ellos.

A tener en cuenta

Tanto el algoritmo de *Dijkstra* como el de *Floyd* permiten obtener los caminos mínimos, pero sólo se pueden aplicar en grafos valorados con *factor de peso* positivo, que por otro lado son los más frecuentes.

18.10.1. Codificación del algoritmo de Floyd

Se define la clase `TodoCaminoMinimo` para implementar el algoritmo para el caso de que el grafo sea un objeto de la clase `GrafoMatriz`. La clase tiene atributo privado el número entero `n` y como atributos protegidos la matriz de distancias `D` y la matriz `traza` que almacenará en cada posición el índice del último vértice que ha hecho que el camino sea mínimo desde el vértice v_i al v_j, para cada par de vértices del grafo. Los métodos de la clase se encargan de retornar las matrices de distancias y de la traza del camino mínimo, así como de obtenerla matriz de distancias mínimas y de visualizar la decodificación de los caminos mínimos.

```
class TodoCaminoMinimo
{
  protected:
   int ** traza;
   int ** D;

  private:
   int n;
```

```
  public:
   int ** Otraza() { return traza;}
   int ** Odistancia() { return D;}
   TodoCaminoMinimo(GrafoMatriz g);
   void Floyd();
   void decodifica (int i, int j);

  private:
     void deco( int i, int j)
};
```

El constructor inicializa el número de vértices, reserva memoria para las dos matrices de la clase e inicializa la matriz de distancias con la matriz de pesos del grafo y la traza de cada posición a `–1` para indicar que el camino si existe es directo.

```
TodoCaminoMinimo::TodoCaminoMinimo(GrafoMatriz g)
{
       n = g.OnumeroDeVertices();
       typedef int* pint;
       D = new pint[n];
       traza = new pint[n];
       // matriz inicial es la de pesos.
       for (int i = 0; i < n; i++)
          {
             D[i] = new int[n];
             traza[i] = new int [n];
             for (int j = 0; j < n; j++)
              {
                 D[i][j] = g.Ovalor(i, j);
                 traza[i][j] = -1; // para decodificar los caminos
              }
          }
}
```

La función miembro pública `Floyd()` se encarga de calcular la matriz de distancias mínimas así como la matriz de traza de los caminos mínimos.

```
void TodoCaminoMinimo::Floyd()
{
     int i, j, k;
     // Camino mínimo de un vértice a si mismo: 0
     for (i = 0; i < n; i++) D[i][i] = 0;
     /* En el caso de que no se realice esta inicialización el algoritmo
     obtiene en la diagonal los ciclos o bucles de longitud mínima */
     for (k = 0; k < n; k++)
      for (i = 0; i < n; i++)
       for (j = 0; j < n; j++)
         if ((D[i][k] + D[k][j]) < D[i][j])   // nuevo mínimo
         {
              D[i][j] = D[i][k] + D[k][j];
              traza[i][j] = k;
         }
}
```

Eficiencia del algoritmo

El tiempo de ejecución del algoritmo crece rápidamente para grafos de relativamente muchos vértices ya que la complejidad es $O(n^3)$.

Para obtener la sucesión de vértices que forman el camino mínimo entre dos vértices se escribe el método público `void recuperaCamino()`, que realiza una llamada al método recursivo `recupera()` que es el encargado de obtener todos los vértices intermedios del camino mínimo entre dos vértices. El código es similar al realizado en la función miembro `recuperaCamino()` de la clase `CaminoMinimo`, con la diferencia que ahora realiza dos llamadas recursivas y se utiliza la matriz `traza` en lugar del vector `ultimo`.

```
void TodoCaminoMinimo::recuperaCamino (int i, int j)
 {
      cout << " camino para ir de " << i << " a " << j << endl;
      recupera(i, j);
 };

void TodoCaminoMinimo::recupera( int i, int j)
 {
      int k = traza[i][j];
      if (k != -1)
       {
          recupera(i, k);
          cout << k ;
          recupera(k, j);
       }
}
```

18.11. ÁRBOL DE EXPANSIÓN DE COSTE MÍNIMO

Los grafos no dirigidos se emplean para modelar relaciones simétricas entre, los vértices del grafo. Cualquier arista `(v,w)` de un grafo no dirigido es igual que `(w,v)`, se dice que estas aristas están formadas por un par no ordenado de vértices. Como consecuencia directa, la representación de un grafo no dirigido da lugar a matrices simétricas.

Una propiedad, que normalmente interesa conocer, de un grafo no dirigido es si para todo par de vértices hay un camino que los une, en definitiva, si el grafo es *conexo*. A los grafos conexos también se les denomina *Red Conectada*. El problema del *árbol de expansión de coste mínimo* consiste en buscar un árbol que *cubra* todos los vértices del grafo, con suma de pesos de aristas mínimo. Los árboles de expansión se aplican en el diseño de redes de comunicación.

Un *árbol*, en una red, es un subconjunto `G'` del grafo `G` que es conectado y sin ciclos. Los árboles tienen dos propiedades importantes:

1. Todo árbol de `n` vértices contiene exactamente `n-1` aristas.
2. Si se añade una arista a un árbol entonces resulta un ciclo.

Definición

Árbol de expansión de coste mínimo: es un subconjunto del g rafo que abarca a todos los vértices que están conectados, cuyas aristas tienen una suma de pesos mínima.

Buscar un árbol de expansión de un grafo, en una red, es también una forma de averiguar si está conectado. Todos los vértices del grafo tienen que estar en el árbol de expansión para que sea grafo conectado. La Figura 18.23 muestra un grafo no dirigido y su árbol de expansión, el grafo es una *red conectada*.

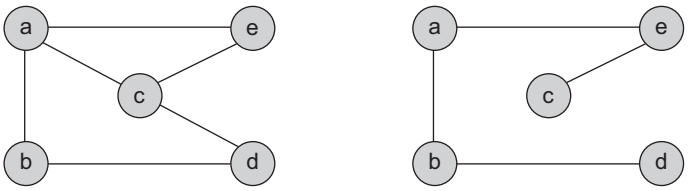


Figura 18.23. Grafo no dirigido y su árbol de expansión.

El planteamiento del problema es el siguiente: dado un grafo no dirigido ponderado y conexo, encontrar el subconjunto del grafo compuesto por todos los vértices, con conexión entre cada par de vértices, sin ciclos y que cumpla que la suma de los pesos de las aristas sea mínimo. Simplemente, encontrar el *árbol de expansión de coste mínimo*.

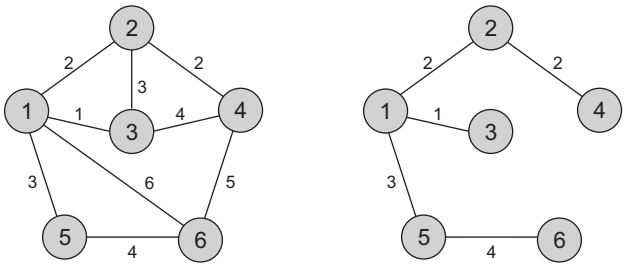


Figura 18.24. Grafo valorado y su árbol de expansión mínimo.

18.11.1. Algoritmo de Prim

El algoritmo de *Prim* encuentra el árbol de expansión de coste mínimo (se conectan todos los vértices del grafo con suma de aristas mínima) de un grafo no dirigido. Consta de sucesivos pasos, siguiendo la metodología clásica de los algoritmos *voraces*: realizar en cada paso *"lo mejor que se pueda hacer"*. En este problema, *lo mejor* consiste en incorporar al árbol una nueva arista del grafo de menor longitud.

Se parte de un grafo `G = (V, A)` no dirigido conectado, una red, siendo `c(i,j)`el *peso* o *coste* asociado al arco (v_i, v_j). Para describir el algoritmo, se supone los vértices están

numerados de 1 a n. El conjunto W va a contener los vértices que ya han pasado a formar parte del *árbol de expansión*.

El algoritmo arranca asignando un vértice inicial al conjunto W, por ejemplo el vértice 1: W = {1}. A partir del vértice inicial el *árbol de expansión* crece, añadiendo a W, en cada pasada otro vértice z todavía no incluido en W, tal que si u es un vértice cualquiera de W, la arista (u,z) es la *más corta*, la de *menor coste*. El proceso termina cuando todos los vértices del grafo están en W y, por consiguiente, el *árbol de expansión* con todos los vértices está formado, además es mínimo porque en cada pasada se ha añadido la menor arista.

La Figura 18.25 muestra un grafo conectado, formado por 10 vértices. Inicialmente se incorpora al conjunto W el vértice 1, W = {1}. De todas las aristas que forma parte el vértice 1, la de menor coste es (1, 2), por ello se añade a W el vértice 2, W = {1, 2}. La siguiente arista mínima, formada por cualquier vértice de W y los vértices no incorporados es (2, 4), entonces se añade a W el vértice 4, W = {1, 2, 4}. La siguiente pasada añade el vértice 5, al ser la arista (4, 5) la mínima. Una nueva pasada incorpora el vértice 8 ya que la arista (5, 8) es la de menor coste, W = {1, 2, 4, 5, 8}. A continuación, se incorpora el vértice 7 ya que la arista (5, 7) es la de menor coste, W = {1, 2, 4, 5, 8, 7}; le siguen los vértices 9, 10, 3 y termina el algoritmo con el vértice 6, W = {1, 2, 4, 5, 8, 7, 9, 10, 3, 6}. La Figura 18.26 muestra el árbol de expansión de coste mínimo formado.

Observación

En los sucesivos pasos del algoritmo de *Prim*, los vértices de W forman una componente conexa sin ciclos ya que las aristas elegidas tienen un vértice en W y el otro en los restantes vértices, V – W.

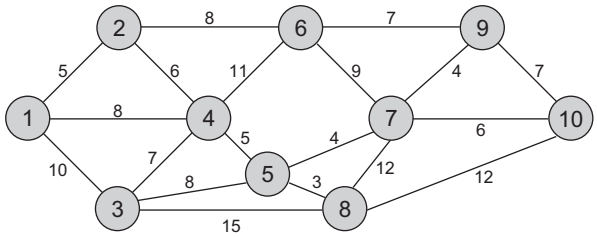


Figura 18.25. Grafo valorado conexo, representa una red telefónica.

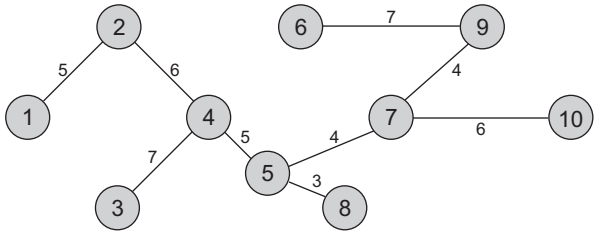


Figura 18.26. Árbol de expansión del grafo de la Figura 18.25.

El algoritmo puede expresarse:

```
arbolExpansion Prim(G, T)
{    grafo G
     <conjunto de arcos del árbol de coste mínimo> T
  variables locales
   < conjunto de vértices > W
   vertices u, w
  inicio
   T <- {}
   V <- {1..n}
   u <- 1
   W <- {u}
   mientras W <> V hacer
    <Encontrar u d W y v de V - W tal que (u,v) sea mínimo>
    W <- W + {v}
    T <- T +{(u, v)}
   fin_mientras
fin_arbolExpansion
```

18.11.2. Codificación del algoritmo de Prim

La clase *`ArbolExpansionMinimo`* implementa el algoritmo de Prim para calcular el árbol de expansión. El constructor inicializa la matriz de *costes* y el número de vértices. A continuación se detallan las acciones a realizar.

Para resolver el problema de encontrar, en cada pasada, la arista de menor *peso* que une un vértice de `W` con otro de `V - W` se utilizan dos arrays:

`masCerca`, tal que `masCerca[i]` contiene el vértice de `W` de menor *coste* respecto el vértice `i` de `V - W`.
`coste`, tal que `coste[i]` contiene el peso de la arista `(i, masCerca[i])`.

En cada pasada se revisa `coste` con el fin de encontrar para cada vértice `j` de `V – W` el vértice `z` de `W` que es el más cercano a `j`. A continuación, se actualizan `masCerca` y `coste` teniendo en cuenta que `z` ha sido añadido a `W`.

La clase `ArbolExpansionMinimo` tiene como atributos protegidos una matriz `T` que almacena el árbol de expansión de coste mínimo y un entero `longMin` que almacena la suma de las aristas del árbol de expansión. Son atributos privados el número de vértices `n` del grafo y el valor de `INFINITO`.

```
class ArbolExpansionMinimo
{
protected :
  int ** T;
  int longMin;
private:
  int n;
  int INFINITO;

public:
  int ** OT(){ return T;}     // Arbol de expansión
```

```
     int OlongMin(){ return longMin;} // peso del Árbol expansión
     ArbolExpansionMinimo(GrafoMatriz g);         // constructor
     int arbolExpansionPrim(GrafoMatriz g);       // algoritmo de Prim
};
```

El constructor de la clase inicializa el número de vértices `n`, el valor de `INFINITO` así como `longMin`. Se encarga además de reservar memoria en una tabla que almacena el árbol de expansión inicializando todas sus aristas a `INFINITO`.

```
ArbolExpansionMinimo::ArbolExpansionMinimo(GrafoMatriz g)
{
     n = g.OnumeroDeVertices();
     INFINITO = 0xFFFF;
     longMin = 0;
     typedef int * pint;
     T = new pint [n];
     for ( int i = 0 ; i < n; i++)
     {
       T[i] = new int [n];
       for(int j = 0; j < n; j++)
           T[i][j] = INFINITO;
     }
}
```

Para calcular el árbol de expansión se necesitan los arrays `coste`, `masCerca` y `W` a los que se reserva memoria en el método `arbolExpansionPrim`. Estos arays contienen respectivamente: `coste[j]` si `j` no está en `W` su valor es la arista mínima del vértice `j` a los vértices que están en `W` e `INFINITO` en caso de que esté en `W`. `MasCerca[j]` tiene por valor el vértice de W que da la arista cuyo peso se almacena en `coste[j]`. `W` contiene siempre el conjunto de vértices que ya están en el árbol de expansión.

```
int ArbolExpansionMinimo::arbolExpansionPrim(GrafoMatriz g)
{
  int menor;
  int i, j, z;
  int *coste = new int [n];
  int * masCerca = new int [n];
  bool *W = new bool [n];
  for (i = 0; i < n; i++)
       W[i] = false;      // conjunto vacío
  W[0] = true;            //se parte del vértice 0
                          // inicialmente, coste[i] es la arista (0,i)

  for (i = 1; i < n; i++)
  {
     coste[i] = g.Ovalor(0, i);
     masCerca[i] = 0;
  }
  coste[0] = INFINITO;
  for (i = 1; i < n; i++)
  {  // busca vértice z de V-W mas cercano,
     // de menor longitud de arista, a algún vértice de W
    menor = coste[1];
    z = 1;
```

```
    for (j = 2; j < n; j++)
      if (coste[j] < menor)
      {
        menor = coste[j];
        z = j;
      }
    longMin += menor;

    // se escribe el arco incorporado al árbol de expansión
    cout << "V" << masCerca[z] << " -> V" << z;
    W[z] = true;       // vértice z se añade al conjunto W
    T[masCerca[z]][z] = T[z][masCerca[z]] = coste[z];
    coste[z] = INFINITO;
                       // debido a la incorporación de z,
                       // se ajusta coste[] para el resto de vértices
    for (j = 1; j < n; j++)
      if ((g.Ovalor(z, j) < coste[j]) && !W[j])
      {
        coste[j] = g.Ovalor(z, j);
        masCerca[j] = z;
      }
  }
  return longMin;
}
```

18.11.3. Algoritmo de Kruscal

Kruskal propone otra estrategia para encontrar el árbol de expansión de coste mínimo. El árbol se empieza a construir con todos los vértices del grafo `G` pero sin aristas. Cada uno de estos vértices forma una componente conexa de un solo vértice. El algoritmo construye componente conexas de mayor tamaño hasta conseguir que todos los vértices del grafo se encuentren en una misma componte conexa. En cada iteración examina las aristas del grafo en orden creciente de *peso* formando una nueva componente conexa al unir dos componentes conexas distintas por una nueva arista del gafo de coste mínimo. Si la arista de coste mínimo conecta vértices que se encuentran en dos componentes conexas distintas, entonces se añade la arista al árbol de expansión `T` y se forma una nueva componente conexa eliminado las dos anteriores; en caso contrario se rechaza ya que daría lugar a un ciclo (no sería árbol). Cuando todos los vértices están en un sólo componente, `T`, éste es el árbol de expansión de coste mínimo del grafo `G`.

El algoritmo de *Kruskal* asegura que el árbol no tiene ciclos ya que para añadir una arista sus vértices deben estar en dos componentes distintas y, además, es de coste mínimo ya que examina las aristas en orden creciente de sus pesos, por lo que necesariamente obtiene el árbol de expansión de coste mínimo. A continuación, se obtiene su árbol de expansión, aplicando el algoritmo del grafo de la Figura 18.24. En primer lugar se obtiene la lista de aristas en orden creciente de sus pesos:

```
{(1, 3),(1, 2),(2, 4),(1, 5),(2, 3),(3, 4),(5, 6),(4, 5),(1, 6)}
```

La primera arista que se toma es `(1, 3)`, Figura 18.27 (*a*), a continuación las aristas `(1, 2)`, `(2, 4)` y `(1, 5)`, en las que se cumple que los vértices forman parte de dos compo-

nentes conexas diferentes, Figura 18.27(*b*). La siguiente arista, `(2, 3)`, sus vértices están en la misma componente, por tanto se descarta. Le sigue la arista `(3, 4)` que por el mismo motivo es descartada. La siguiente `(5, 6)`, sus vértices están en dos componentes diferentes, pasa a formar parte del árbol y el algoritmo termina ya que se han alcanzado todos los vértices.

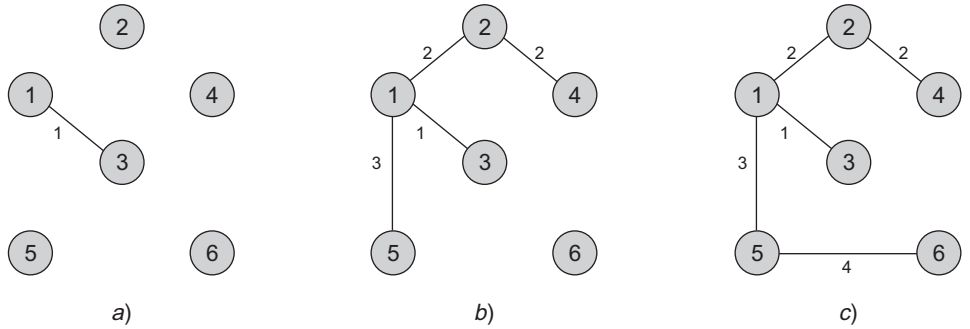


Figura 18.27. Formación del árbol de expansión del grafo de la Figura 18.24.

Lecturas recomendadas para profundizar en el tema (www.mhe.es/joyanes)

Anexo C: Implementación de un grafo con listas de adyacencia y los puntos de articulación de un grafo.
Anexo D : Grafos para representar redes de flujo.

RESUMEN

Un grafo `G` consta de un par de conjuntos `(G = (V, E))`: conjunto `V` de vértices o nodos y el conjunto `E` de *aristas* (que conectan dos vértices). Si las parejas de vértices que forman una arista no están ordenadas, `G` se denomina grafo *no dirigido*; si los pares están ordenados, entonces `G` se denomina grafo *dirigido*. El término grafo *dirigido* se suele también designar como *digrafo* y el término grafo sin calificación significa grafo *no dirigido*.

El método natural para dibujar un grafo es representar los vértices como puntos o círculos y las aristas como segmentos de líneas o arcos que conectan los vértices. Si el grafo está *dirigido*, entonces los segmentos de línea o arcos tienen puntas de flecha que indican la dirección.

Los grafos se implementan de dos formas: *matriz de adyacencia* y *listas de adyacencia*. Cada una tiene sus ventajas y desventajas relativas. La elección depende, entre otros factores, de la densidad del grafo; para grafos *densos*, con muchos arcos, se recomienda representarlos con la *matriz de adyacencia*. Grafos poco densos y que experimenten modificaciones en sus componentes se representan con *listas de adyacencia*.

Dos vértices de un grafo *no dirigido* se llaman adyacentes si existe una *arista* desde el primero al segundo. Un *camino* es una secuencia de vértices distintos, cada uno adyacente al siguiente. Un *ciclo* es un *camino* que contiene al menos tres vértices, tal que el último vértice en el camino es

adyacente al primero. Un grafo se denomina *conectado* si existe un camino desde cualquier vértice a cualquier otro vértice.

Un grafo *dirigido* se denomina *conectado fuertemente* si hay un *camino* dirigido desde un vértice a cualquier otro. Si se suprime la dirección de los arcos y el grafo *no dirigido* resultante se conecta, se denomina grafo dirigido *débilmente conectado*.

El *recorrido* de un grafo *visita* de cada uno de los vértices, puede ser, en analogía con los árboles, *recorrido en profundidad* y *recorrido en anchura*. El recorrido en profundidad es una generalización del recorrido en *preorden* de un árbol. El recorrido en anchura visita los vértices por *niveles*, al igual que el recorrido por anchura de un árbol, permite encontrar el número mínimo de aristas para alcanzar un vértice cualquiera desde un vértice de partida.

Los *puntos de articulación* de un grafo *conexo* son aquellos vértices que si se retiran del grafo, junto a sus aristas, dividen a éste en dos *componentes conexas*. El algoritmo que permite encontrar los *puntos de articulación*, construye el *árbol de expansión* del grafo a partir de un vértice origen; recorre en profundidad el grafo buscando *aristas de árbol* y *aristas hacia atrás*.

Si `G` es un *grafo dirigido sin ciclos*, entonces una ordenación topológica de `G` es un lista secuencial de los vértices de `G`, tal que para todos los vértices `v, w ∈ G` si existe una arista desde `v` a `w` entonces `v` precede a `w` en el listado secuencial. El término *acíclico* se utiliza con frecuencia para representar que un grafo no tiene ciclos.

La ordenación topológica es una forma de recorrer un grafo, pero sólo aplicable a grafos dirigidos acíclicos, que cumple la propiedad de que un vértice sólo se visita si ya han sido visitados todos sus predecesores. En un orden topológico, cada vértice debe aparecer antes que todos los vértices que son sus sucesores en el *grafo dirigido*.

En un grafo una de las operaciones clave es la búsqueda de *caminos mínimos*, es decir, caminos de menor longitud. La longitud de un camino se define como la suma de los *costes* de las aristas que lo componen. El *algoritmo de Dijkstra* determina el coste del camino mínimo desde un vértice al resto de vértices; es un algoritmo *voraz* que se aplica a un grafo valorado cuyas aristas tengan *coste positivo.*

El *algoritmo de Warshall* es importante ya que para todo par de vértices del grafo indica si hay un camino entre ellos. Con un estructura similar al anterior, el *algoritmo de Floyd* se utiliza para encontrar los caminos mínimos entre todos los pares de vértices de un grafo; también da la posibilidad de obtener la traza (secuencia de vértices) de esos caminos.

Un *árbol de expansión* es un árbol que contiene a todos los vértices de una red. En un árbol no hay ciclos; es una forma de averiguar si la red está conectada. Uno de los problemas más comunes que se plantean sobre las redes es encontrar aquel *árbol de coste mínimo*. El *algoritmo de Prim* y el *algoritmo de Kruskal* resuelven eficientemente el problema de encontrar el árbol de expansión mínimo.

Los grafos también tienen se aplican en modelar problemas de fluidos. En una red de flujo hay un vértice *fuente* y un vértice destino, *sumidero*, al que se dirige el flujo de objetos.

BIBLIOGRAFÍA RECOMENDADA

Aho, V.; Hopcroft, J., y Ullman, J.: *Estructuras de datos y algoritmos*. Addison Wesley, 1983.

Joyanes, L., y Zahonero, L.: *Algoritmos y estructuras de datos. Una perspectiva en C*. McGraw-Hill, 2004.

Joyanes, L.; Sánchez, L.; Zahonero, I., y Fernández, M.: *Estructuras de datos en C*. Schaum, 2005.

McHugh, J.: *Algorithmic Grap Theory*. Prentice-Hall, 1990.

Nyhoff, L.: *TADs, Estructuras de datos y resolución de problemas con C++*. Pearson, Prentice-Hall, 2005.

Sedgewick, R.: *Algoritmos en C++*. Addison-wesley/Díaz de Santos, 1992.

EJERCICIOS

18.1. Dado el grafo no dirigido, G, de la Figura 18.28.

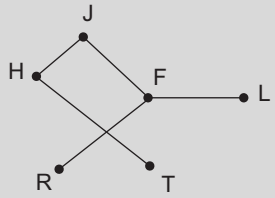


Figura 18.28. Gr afo no dirigido.

a) Describir G formalmente en términos del conjunto de nodos, V, y del conjunto A de arcos.
b) Encontrar el grado de cada nodo.

18.2. Dado el grafo dirigido, G, de la Figura 18.29.

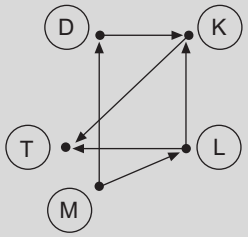


Figura 18.29. Gr afo dirigido.

a) Describir G formalmente en términos de su conjunto de nodos, V, y de su conjunto A de arcos.
b) Encontrar el grado de entrada y el grado de salida de cada vértice.
c) Escribir la secuencia de vértices que forman los caminos simples del vértice M al vértice T.

18.3. Dado el grafo no dirigido, G, de la Figura 18.30.

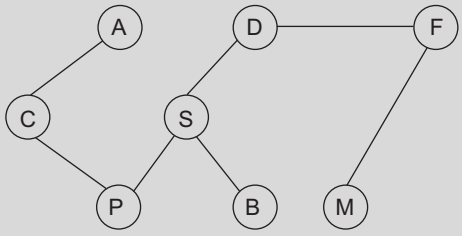


Figura 18.30. Gr afo no dirigido.

a) Escribir la secuencia de vértices que forman los caminos simples del vértice A al vértice F.
b) Encontrar el camino más corto (en cuanto a número de aristas) del vértice C al vértice D.
c) ¿Es un grafo conexo?

18.4. Dado el grafo dirigido y valorado, G, de la Figura 18.31.

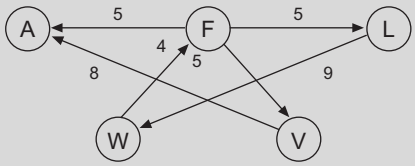


Figura 18.31. Grafo valorado.

a) Escribir la matriz de pesos del grafo.
b) Respresentar el grafo mediante listas de adyacencia.

18.5. Un grafo está formado por los vértices V = {A, B, C, D, E}, su matriz de adyacencia, suponiendo los vértices numerados del 0 al 4 respectivamente es:

$$M = \begin{vmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{vmatrix}$$

a) Dibujar el grafo que le corresponde.
b) Representar el grafo mediante listas de adyacencia.

18.6. Un grafo no dirigido conexo tiene la propiedad de ser biconexo si no hay ningún vértice que al suprimirlo del grafo haga que éste se convierta en no conexo.

a) Dibujar un grafo de 6 nodos biconexo.
b) Determinar si los grafos de la Figura 18.28 y 18.29 son biconexos.

18.7. Dado el grafo no dirigido del Ejercicio 18.5 realizar el recorrido en profundidad a partir del vértice C.

18.8. Dado el grafo no dirigido del Ejercicio 18.5 realizar el recorrido en anchura a partir del vértice C y la longitud de los caminos mínimos a los demás vértices.

18.9. En la Figura 18.32 se representan dos grafos dirigidos. Teniendo en cuenta que un grafo dirigido se considera *acíclico* si no tiene ciclos, también se denominan **gda**, indicar si los grafos de la figura son **gda**. De no se **gda**, buscar los ciclos.

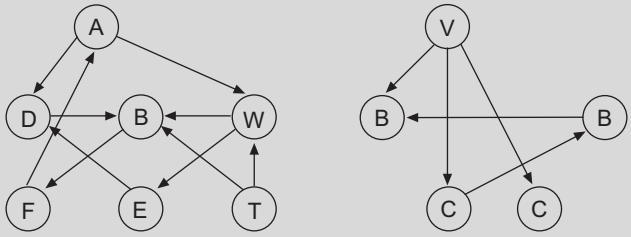


Figura 18.32. Grafos dirigidos.

18.10. Dado el grafo de la Figura 18.33 encontrar la componentes fuertemente conexas.

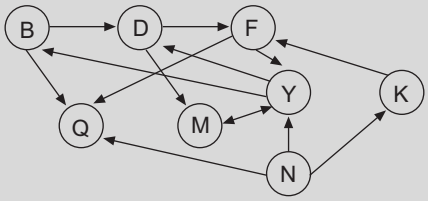


Figura 18.33. Grafos dirigido.

18.11. Dado el grafo G de la Figura 18.34.

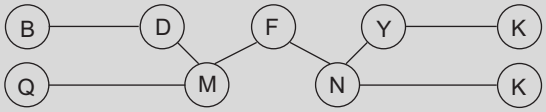


Figura 18.34. Grafos no dirigido.

a) Escribir la matriz de adyacencia del grafo.
b) Escribir la matriz de caminos de G.

18.12. Dado un grafo dirigido cuyos vértices son números enteros positivos y el par (x,y) es una arista si x-y es múltiplo de 3.

a) Representar el grafo formado por los vértices 3 al 14.
b) Determinar el grado de entrada y el grado de salida de cada vértice.

18.13. En los grafos no dirigidos de las Figuras 18.37 y 18.39.

a) Dibujar los correspondientes árboles de expansión.
b) Encontrar los puntos de articulación.

18.14. Dibujar un grafo dirigido cuyos vértices son números enteros positivos del 3 al 15 para cada una de las siguientes relaciones:

a) v es adyacente de w si v+2w es divisible entre 3.
b) v es adyacente de w si 10v+w < v*w.

18.15. Dada la red de la Figura 18.36 encontrar una ordenación topológica.

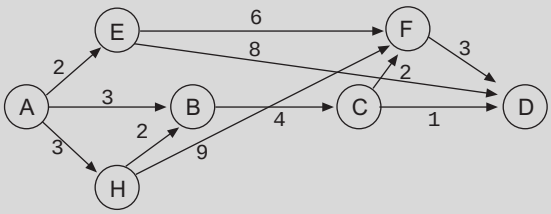


Figura 18.36. Gr afo dirigido.

18.16. En la red de la Figura 18.36 los arcos representan actividades y el *factor de peso* el tiempo necesario para realizar dicha actividad (un *Pert*). Cada vértice v de la red representa el tiempo que tardan todas las actividades que terminan en v. El ejercicio consiste en asignar a cada vértice v de la red el tiempo necesario para que todas las actividades que terminan en v se puedan realizar, esto se llamará *tn(v)*. Una estrategia que se puede seguir: asignar tiempo `0` a los vértices sin predecesores; si a todos los predecesores de v se les ha asignado tiempo, entonces tn(v) es el máximo, para cada predecesor, de la suma de tiempo del predecesor con el *factor de peso* del arco desde ese predecesor hasta v.

18.17. Considerando de nuevo la red de la Figura 18.36 y teniendo en cuenta el tiempo de cada vértice, *tn(v)*, calculado en el Ejercicio 18.16, se quiere calcular el tiempo límite en que todas las actividades que terminan en v pueden ser completadas, sin atrasar la terminación de todas las actividades, a este tiempo se le denomina *tl(v)*. La estrategia a seguir para este cálculo: asignar *tn(v)* a todos los vértices v sin sucesores, si todos los sucesores de v tienen tiempo asignado, entonces *tl(v)* es el mínimo, entre todos los sucesores, de la diferencia entre el tiempo asignado al sucesor, *tl(v')*, y el factor de peso desde v hasta el sucesor v'.

18.18. Una ruta crítica de una red es un camino desde un vértice que no tiene predecesores hasta otro vértice que no tiene sucesores, tal que para todo vértice v del camino se cumple que *tn(v)= = tl(v)*. Encontrar las rutas críticas de la red de la Figura 18.36.

19.19. La Figura 18.37 muestra una red conectada. Encontrar y dibujar un árbol de expansión de coste mínimo aplicando los pasos que propone el *algoritmo de Prim.*

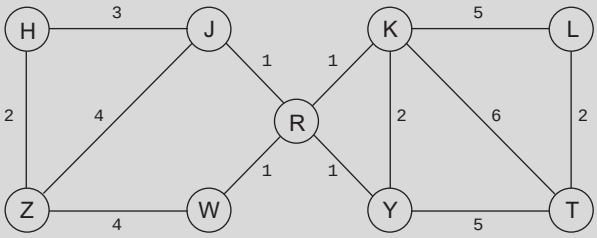


Figura 18.37. Red conectada.

18.20. Encontrar el árbol de expansión de coste mínimo de la red conectada de la Figura 18.37, siguiendo los pasos del *algoritmo de Kruscal*.

18.21. El grafo de la Figura 18.38 es dirigido y las aristas tienen asociado un coste. Determinar el camino más corto desde el vértice A al resto de los vértices del grafo.

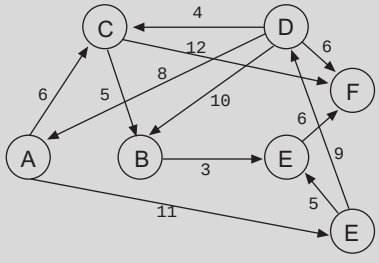


Figura 18.38. Grafo dirigido con factor de peso.

18.22. Dado el grafo de la Figura 18.38 encontrar el camino más corto desde el vértice *D* al resto de vértices. Se ha de seguir los pasos del *algoritmo de Dijkstra*, e incluir la secuencia de vértices que forman el camino.

18.23. En el grafo de la Figura 18.38 obtener los caminos más cortos entre todos los pares de vértices. Aplicar, paso a paso, el *algoritmo de Floyd*.

18.24. Dibujar un grafo dirigido con factor de peso en el que algún arco tenga factor de peso negativo. Al aplicar el *algoritmo de Dijkstra* desde un vértice origen se ha de obtener algún resultado erróneo.

18.25. Escribir las modificaciones necesarias para que, teniendo como base el *algoritmo de Dijkstra*, se calculen los caminos mínimos entre todos los pares de vértices del grafo. ¿Cuál es la eficiencia de este algoritmo?

18.26. Tanto el algoritmo de *Prim* como el de *Kruscal* resuelven el problema de determinar el árbol de expansión mínimo de una red conectada. ¿Se pueden aplicar estos algoritmos si alguna arista tiene factor de peso negativo?

18.27. Determinar, en el grafo de la Figura 18.39, los vértices que son *puntos de articulación*.

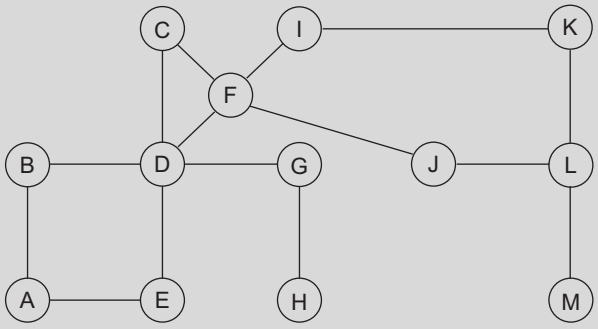


Figura 18.39. Grafo no dirigido con puntos de articulación.

18.28. Un *circuito de Euler*, en un grafo dirigido, es un ciclo en el cual toda arista es *visitada* exactamente una vez. Se puede demostrar que un grafo dirigido tiene un *circuito de Euler* si y solo si es *fuertemente conexo* y para todo vértice el grado de entrada es igual al grado de salida. Dibujar un grafo dirigido en el que se pueda encontrar un *circuito de Euler*.

PROBLEMAS

18.1. Un grafo valorado está formado por los vértices: 4, 7, 14, 19, 21, 25. Las aristas siempre van de un vértice de mayor valor numérico a otro de menor valor, y el peso es el módulo del vértice origen y el vértice destino.

a) Escribir un programa que represente el grafo con listas de adyacencia.
b) Añadir al programa el código necesario para realizar un recorrido en anchura desde un vértice dado.

18.2. Una región está formada por 12 comunidades. Se establece la relación de desplazamiento de personas en las primeras horas del día. Así, la comunidad A está relacionada con la comunidad B si desde A se desplazan n personas a B, de igual forma puede haber relación entre B y A si se desplazan m personas de B hasta A.

a) Escribir un programa que represente el grafo descrito mediante listas de adyacencia.
b) Determinar si el grafo formado tiene fuentes o sumideros.

18.3. Dado el grafo descrito en el Problema 18.2. Escriba un programa para representarlo mediante listas enlazadas de tal forma que cada nodo de la lista directorio contenga dos listas: una que contiene los arcos que salen del nodo, y la otra que contiene los arcos que terminan en el nodo.

18.4. Un grafo en el que los vértices son regiones y los arcos relacionan dos regiones entre las cuales hay un flujo de emigrantes (tiene factor de peso), está representado mediante una lista directorio que contiene a cada uno de los vértices y de las que sale una lista circular con los vértices adyacentes. Ahora se quiere representar el grafo mediante una matriz de pesos, de tal forma que si entre dos vértices no hay arco su posición en la matriz tiene 0, y si entre dos vértices hay arco su posición contiene el factor de peso que le corresponde. Escribir las funciones necesarias para que partiendo de la representación mediante listas se obtenga la representación mediante la matriz de pesos.

18.5. Se tiene un grafo no dirigido poco denso. Se elige la representación en memoria mediante listas de adyacencia. Escribir un programa en el que se dé entrada a los vértices del grafo y sus arcos y determine si el grafo tiene ciclos. En caso positivo, listar los vértices que forman un ciclo.

18.6. Supóngase un grafo no dirigido y conexo, escribir un programa que encuentre un camino que vaya a través de todas las aristas exactamente una vez en cada dirección.

18.7. Se denominan caminos *Hamiltonianos* a aquellos caminos que contienen exactamente una vez a todos y cada uno de los vértices del grafo. Se trata de escribir un programa que lea un grafo y visualice todos sus caminos Hamiltonianos.

18.8. Escribir un programa que compruebe si un grafo dirigido, leído del teclado tiene circuitos (ciclos) mediante el siguiente algoritmo:

1. Obtener los sucesores de todos los vértices.
2. Buscar un vértice sin sucesores y *tachar* (eliminar) ese vértice donde aparezca (conjuntos de sucesores).
3. Se continúa el proceso en el paso 2 siempre que haya algún vértice sin sucesores.
4. Si todos los vértices del grafo han sido eliminados, no tienen circuitos.

18.9. En grafo que representa a una red es poco denso; en un programa se quiere implementar el grafo mediante *listas de adyacencia*. El programa tiene que encontrar las *rutas críticas* del grafo.

18.10. Se tiene una red (un *Pert*) representada con su matriz de pesos. Escribir un programa que calcule el mínimo tiempo en el que todo trabajo se puede terminar si tantas tareas como sea posible son realizadas en paralelo. El programa debe de escribir el tiempo en el que se inicia y se termina toda actividad en la red.

18.11. Escribir un programa para que dada una red determine el tiempo mínimo de realización del trabajo si como máximo se pueden realizar *n* actividades (de las posibles) en paralelo. El programa tiene como entrada la matriz de pesos del grafo, la salida del programa muestra el tiempo de inicio y el de finalización de cada actividad.

18.12. Cada iteración que realiza el algoritmo de *Dijkstra* selecciona aquel vértice para el cual el camino es mínimo desde el vértice origen. Implementar el algoritmo teniendo en cuenta que si en una iteración hay dos vértices en los que coincide la distancia mínima, seleccionar aquél con el menor número de arcos.

18.13. Tomando como base el algoritmo de *Dijkstra*, implementar un algoritmo que resuelva el problema de que dado un grafo *G*, representado por su matriz de pesos, encuentre todos los caminos mínimos desde todo vértice v a un mismo vértice destino d.

18.14. Dado un grafo no dirigido con factor de peso, escribir un programa que tenga como entrada dicho grafo, lo represente en memoria y encuentre el árbol de expansión de coste mínimo.

18.15. Un *circuito de Euler* en un grafo dirigido es un ciclo en el cual toda arista es visitada exactamente una vez. Se puede demostrar que un grafo dirigido tiene un *circuito de Euler* si y sólo si es fuertemente conexo y para todo vértice el grado de entrada es igual al grado de salida. Escribir un programa que implemente un grafo dirigido mediante *listas de adyacencia* e implemente un algoritmo para encontrar, si existe, un *circuito de Euler*.

18.16. Un grafo G simula una red de centros de distribución. Cada centro dispone de una serie de artículos y un stock de ellos, representado mediante una estructura lineal ordenada respecto al código de artículo. Entre los centros hay conexiones que no tienen por qué ser bidireccionales. Escribir un programa que represente el grafo dirigido ponderado (el coste de las aristas es la distancia, en kilómetros, entre los dos vértices). El programa tiene que resolver el problema: un centro no tiene un artículo *z* y lo requiere, entonces el centro más cercano que disponga de z lo suministra.

18.17. Las ciudades dormitorio de una gran área metropolitana están conectadas a través de una red de carreteras, pasando por poblaciones intermedias, con el centro de la gran ciudad. Cada conexión entre dos nodos viene dada por el tiempo de desplazamiento, a la velocidad máxima de 50 km/h, entre los dos nodos. Escribir un programa que simule el supuesto mediante un grafo y determine el tiempo mínimo para ir desde una ciudad dormitorio al centro de la ciudad, y la sucesión de nodos por los que debe pasar.

Bibliografía

[AHOULL83] Aho, V.; Hopcroft, J., y Ullman J.: *Estructuras de datos y algoritmos*. Addison Wesley, 1983.

[BAILEY99] Bailey, Duane: *Data Structures in Java*. McGraw-Hill, 1999.

[BANKRE91] Banachowski L.; Kreczmar A., y Rutter W.: *Analysis of Algorithms and data structures.* Addisson Wesley, 1991.

[BRABRA90] Brassard, G., y Bratley, P.: *Algoritmia. Concepción y Análisis*. Prentice-Hall. Massen, 1990.

[BRABRA97] Brassard, G., y Bratley, P.: *Fundamentos de Algoritmia*. Prentice Hall, 1997.

[BRACOJ92] Brassard, G.; Cohe, G., y Lobstein: *Complexité algorithmique et problemes de communications.* Masson, 1992.

[BUDD01] Budd, Timothy: *Classic Data Structures in Java.* Addison-Wesley, 2001.

[CAIGUA93] Cairó, O., y Guardati, S.: *Estructuras de datos.* McGraw-Hill, 1993.

[CALGON93] Calvé, J., y González, C.: *Algorítmica. Diseño y análisis de algoritmos funcionales e imperativos*. Rama, 1993.

[COLMOR87] Collado, M., y Morales, F.: *Estructuras de datos. Realización en Pascal.* Díaz de Santos, 1987.

[DALLIL85] Dale, N., y Lilly, S.: *Pascal y Estructura de datos,* 1985.

[FRANCH94] Franch Gutiérrez, Xavier: *Estructuras de datos. Especificación, diseño e implementación,* Ediciones UPC, 1994.

[DEVIS93] Devis Botella, R.: *Programación en C++*. Paraninfo, 1993.

[GARFER06] Garrido, A., y Fernández, J.: *Abstracción y estructuras de datos en C++*. Delta, 2006.

[GILFOR01] Gilber, R., y Forouzan, B.: *Data Structures: A Pseudocode* Approach witch C++. Pacific Grove, 2001.

[HEILEMAN98] Heileman, G.: *Estructuras de datos, algoritmos y programación orientada a objetos.* McGraw-Hill, 1998.

[HERLAZ00] Hernández, R., y Lázar, J. C.: *Estructuras de datos y algoritmos.* Prentice-Hall, 2000.

[HERROD05] Hernández, Z., y Rodríguez, J.: *Fundamentos de Estructuras de datos. Soluciones en Ada, Java y C++*. Thomson, 2005.

[HOFRI95] Hofri, M.: *Análysis of algorithms. Computational Methods & Mathematical Tool*. Oxford Universiy Press, 1995.

[HOOSAH94] Hoorwitz, E., y Sahni, S.: *Fundamentals of data structures in Pascal*, 1994.

[HORSAH77] Horowitz, E., y Sahni, S.: *Algoritms: Desingn and Análisis,* Computer Science Press, Potomac, MD., 1977.

[JOYAZAH98] Joyanes, L., y Zahonero, L.: *Estructuras de datos. Algoritmos, abstracción y objetos.* McGraw-Hill, 1998

[JOYASAN99] Joyanes. L.; Sánchez, L.; Zahonero, I., y Fernández, M.: *Estructuras de datos. Libro de Problemas.* McGraw-Hill, 1999.

[JOYANES02] Joyanes, L.: *Programación en Java 2. Algoritmo, estructuras de datos y programación orientada a objetos*. McGraw Hill, 2002.

[JOYANES03] Joyanes, L.: *Fundamentos de programación*, 3.ª ed., McGraw-Hill, 2003.

[JOYAZAH04] Joyanes, L., y Zahonero, L.: *Algoritmos y estructuras de datos. Una perspectiva en C*. McGrawHill, 2004.

[JOYASAN06] Joyanes, L., y Sánchez, L. *Programación en C++*. Colección Schaum. McGraw-Hill, 2006.

[JOYANES06] Joyanes, L.: *Programación en C++. Algoritmos, estructuras de datos y objetos.* McGraw Hill, 2006.

[JOYASAN05] Joyanes, L.; Sánchez, L.; Zahonero, I., y Fernández, M.: *Estructuras de datos en C.* Colección Schaum. McGraw-Hill, 2005.

[KALDEW90] Kaldewaij. *Orogramming: The Derivation of Algorithms*. Prentice-Hall, 1990.

[KNUTH86] Knuth, D. E.: *El arte de programar Ordenadores. Algoritmos Fundamentales*, Volumen I. Reverté, 1986.

[KNUTH87] Knuth, D. E.: *El arte de programar Ordenadores. Clasificación y búsqueda*, Volumen III. Reverté, 1987.

[KRUSE94] Kruse, R.: *Data Estructures and Program Desing*. Prentice-Hall, 1994.

[LIPSCH97] Lipschutz, S.: *Estructura de datos*. McGraw-Hill, 1987.

[MAINSAVI01] *Data Structures and other objects using C++*. Second edition. Addison-Wesley, 2001.

[MARORG03] Martí, N., y Ortega, Y.: *Estructauras de datos y Métodos Algorímicos*. Prentice-Hall, 2003.

[MCHUG90] McHugh, J.: *Algorithmic Graph Theory*. Prentice-Hall, 1990.

[NYHOFF05] Nyhoff, L.: TADs, *Estructuras de datos y resolución de problemas con C++*. Pearson. Prentice Hall, 2005.

[PEÑA98] Peña Marí, R.: *Diseño de Programas. Formalismo y abstracción*. Prentice-Hall, 1998.

[SALMON91] Shaffer, W.: *Structures and Abstraction*, M. A. Irwinm, 1991.

[SEDGEW92] Sedgewick, R.: *Algoritmos en C++*. Addison-wesley/Díaz de Santos, 1992.

[SHAF97] Shaffer, C.: *A practical Introduction to Data Structures and Algorithm Analysis*. Upper Saddle River. Prentice Hall, 1997.

[STANDIS95] Standish, T.: *Data Structures, Algorithms & Software Principles in C*. Reading Massachussetts. Addison-Wesley, 1995.

[STEVE98] Steve, S. Skiena: *The Algorithm design manual*. Telos, 1998.

[STIVEN89] Stivens, R.: *Fractal: Programming in C*. M&T. Books, 1989.

[TANAUG81] Tanenbaum, A., y Augenstein, M. *Estructura de datos en Pascal*. Prentice-Hall. 1981.

[TREMBL80] Tremblay, J., y Bunt, R.: *Pascal estructurado*. McGraw-Hill, 1980.

[VANDEVO03] Vandevoorde, David, y Josuttis, Nicolai M.: *C++ Template. The Complete Guide.* Addison Wesley, 2003.

[WEIS92] Weis, Mark Allen: *Estructuras de datos y algoritmos*. Addison Wesley, 1992.

[WEIS95] Weis, Mark Allen: *Data Structures and Algorithm Analisis.* Benajmming Cummings, 1995.

[WEIS 00] Weis, Mark Allen: *Estructuras de datos en Java*. Prentice-Hall, 2000.

[WIRTH86] Wirth, Niklaus: *Algoritmos + Estructuras de datos = programas*, 1986.

[WIRTH86a] Wirth, Niklaus: *Introducción a la programación matemáticas*. El Ateneo, 1986.

[WOOD 93] Wood, D.: *Data Structures. Algorithms, and Performance.* Addison-Esley, 1993.

Índice

Estructura de datos en C++

Esta obra ha sido diseñada y construida como un libro eminentemente didáctico. En su contenido se han plasmado la gran experiencia docente y editorial de sus autores, con numerosos libros relacionados con algoritmos, programación y estructuras de datos. Se ha pretendido mostrar al lector los temas fundamentales de un área tan importante en informática y en computación como es Algoritmos y Estructuras de Datos. La organización del libro comienza con la iniciación a los algoritmos y a las estructuras de datos, así como los conceptos fundamentales de C++ y de programación orientada a objetos, tales como clases, objetos, clases derivadas, herencia, polimorfismo y genericidad (templates). El resto del libro y núcleo fundamental es el estudio de algoritmos y estructuras de datos clásicos como listas, colas, colas de prioridad, árboles, grafos, algoritmos de búsqueda y ordenación. Se incluye un capítulo específico (15) con una amplia referencia a la biblioteca estándar de plantillas de C++ (**STL**, Standard Template Library) con el estudio de iteradores, contenedores y algoritmos especiales, que luego se utilizarán ampliamente en los capítulos de árboles y grafos.

Uno del los objetivos fundamentales del libro es enseñar al estudiante, buenas reglas de programación y análisis de algoritmos, de modo que puedan desarrollar los programas con la mayor eficiencia posible.

El libro enseña como utilizar C++ en el análisis y diseño de algoritmos junto con un estudio profundo de estructuras de datos. Para ello se describen y analizan conceptos tan importantes como:

- **Algoritmos y programas**
- **Clases y Objetos**
- **Clases derivadas, herencia y polimorfismo**
- **Estructuras de datos básicas (*arrays* o arreglos, conjuntos, registros)**
- **Genericidad *(templates)***
- **Análisis y Eficiencia de algoritmos**
- **Recursividad**
- **Algoritmos de búsqueda y ordenación interna y externa**
- **Listas**
- **Pilas**
- **Colas y colas de prioridad**
- **Tablas de dispersión *(hash)***
- **Árboles: binarios, de búsqueda, ordenados, equilibrados, AVL y B**
- **Grafos**
- **Biblioteca de plantillas estándar (STL)**
- **Iteradores y contenedores**
- **Algoritmos avanzados**

Este libro dispone de OLC, Online Learning Center, página web asociada, lista para su uso inmediato y creada expresamente para facilitar la labor docente del profesor y el aprendizaje de los alumnos.
Se incluyen contenidos adicionales al libro y recursos para la docencia.

www.mhe.es/joyanes

www.mcgraw-hill.es

The McGraw-Hill Companies