

¡¡APRUEBA TU EXAMEN CON SCHAUM!!

Estructuras de datos en C

Schaum

Luis Joyanes Aguilar • Matilde Fernández Azuela
Lucas Sánchez García • Ignacio Zahonero Martínez

REDUCE TU TIEMPO DE ESTUDIO

EJEMPLOS DETALLADOS, CUESTIONES DE REPASO Y PROBLEMAS DE COMPRENSIÓN

PROBLEMAS RESUELTOS EN LA WEB

EJERCICIOS PRÁCTICOS DE APRENDIZAJE DE PROGRAMACIÓN

Utilízalo para las siguientes asignaturas:

✓ ALGORITMOS Y ESTRUCTURAS DE DATOS

✓ METODOLOGÍA DE LA PROGRAMACIÓN

✓ ESTRUCTURAS DE DATOS

✓ PROGRAMACIÓN ESTRUCTURADA

Estructuras de datos en C

Estructuras de datos en C

**LUIS JOYANES AGUILAR
MATILDE FERNÁNDEZ AZUELA
LUCAS SÁNCHEZ GARCÍA
IGNACIO ZAHONERO MARTÍNEZ**

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
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 • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO**

La información contenida en este libro procede de una obra original entregada por los autores. No obstante, McGraw-Hill/Interamericana de España no garantiza la exactitud o perfección de la información publicada. Tampoco asume ningún tipo de garantía sobre los contenidos y las opiniones vertidas en dichos textos.

Este trabajo se publica con el reconocimiento expreso de que se está proporcionando una información, pero no tratando de prestar ningún tipo de servicio profesional o técnico. Los procedimientos y la información que se presentan en este libro tienen sólo la intención de servir como guía general.

McGraw-Hill ha solicitado los permisos oportunos para la realización y el desarrollo de esta obra.

Estructura de datos en C. Serie Schaum

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.



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

DERECHOS RESERVADOS © 2005, respecto a la primera edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1ª planta
Basauri, 17
28023 Aravaca (Madrid)

www.mcgraw-hill.es
universidad@mcgraw-hill.com

ISBN: 84-481-4512-7
Depósito legal: M.

Editor: Carmelo Sánchez González
Compuesto en CD-FORM, S.L.
Impreso en

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Contenido

Prólogo	XI
Capítulo 1 Algoritmos, estructura de datos y programas	1
1.1 Resolución de problemas de programación	1
1.2 Abstracción de datos	1
1.3 Análisis de un problema.....	1
1.4 Diseño de un programa	2
1.5 Implementación (codificación) de un programa	2
1.6 Verificación	3
1.6.1 Método básico de verificación formal	3
1.6.2 descripción formal de tipos de datos	3
1.6.3 el lenguaje de la lógica de primer orden.....	4
Problemas resueltos básicos	6
Problemas resueltos avanzados	9
Problemas propuestos.....	12
Capítulo 2 Análisis de algoritmos	15
2.1 Medida de la eficiencia de un algoritmo.....	15
2.1.1 Evaluación de la memoria.....	15
2.1.2 Evaluación del tiempo.....	15
2.2 Notación O-GRANDE	16
2.2.1 Descripción de tiempos de ejecución	17
2.2.2 Definición conceptual	17
2.2.3 Definición formal.....	17
2.2.4 Propiedades de la notación O	17
2.2.5 Complejidad de las distintas sentencias y programas en C	17
2.2.6 Funciones de complejidad de algoritmos más comúnmente consideradas.....	18
2.2.7 Tabla comparativa de las distintas funciones de complejidad más usuales	18
2.2.8 Inconvenientes de la notación O-grande.....	19
Problemas básicos	19
Problemas avanzados	28
Problemas propuestos.....	31
Capítulo 3 Arrays o arreglos (listas y tablas), estructuras y uniones en C	33
3.1 Array unidimensional	33
3.2 Array multidimensionales	35
3.3 Array como parámetros a funciones	36
3.4 Estructuras	36
3.5 Uniones.....	38
3.6 Enumeraciones	38
3.7 Typedef	38
Problemas resueltos.....	39
Problemas avanzados	45
Problemas propuestos.....	54

Capítulo 4 Recursividad	57
4.1 Algoritmos recursivos	57
4.2 Casos en los que debe evitarse el uso de la recursividad	58
4.3 Recursividad directa e indirecta	59
4.4 Métodos para la resolución de problemas que usan recursividad	60
4.4.1 Divide y vence	60
4.4.2 <i>Backtracking</i> (retroceso)	60
Problemas básicos	62
Problemas avanzados	64
Problemas propuestos.....	94
Capítulo 5 Algoritmos de búsqueda y ordenación	97
5.1 Búsqueda	97
5.1.1 Búsqueda lineal	97
5.1.2 Búsqueda binaria	98
5.2 Clasificación interna.....	98
5.3 Ordenación por burbuja	99
5.4 Ordenación por selección	99
5.5 Ordenación por inserción	100
5.7 Métodos de ordenación por urnas	100
5.7.1 <i>BinSort</i>	100
5.7.2 <i>RadixSort</i>	100
5.8 Ordenación rápida (<i>QuickSort</i>)	101
5.9 Ordenación por mezcla	101
5.10 Clasificación por montículo	101
Problemas básicos	101
Problemas de seguimiento	107
Algoritmos avanzados	110
Problemas propuestos.....	119
Capítulo 6 Archivos y algoritmos de ordenación externa	121
6.1 Archivos en C.....	121
6.2 Operaciones con archivos	121
6.3 Ordenación externa	125
6.4 Ordenación por mezcla directa	127
6.5 Ordenación por mezcla natural	127
6.6 Método de la mezcla equilibrada múltiple.....	127
6.7 Método polifásico.....	128
Problemas de seguimiento	129
Problemas básicos	133
Problemas avanzados	139
Problemas propuestos.....	150
Capítulo 7 Tipos abstractos de datos y objetos	151
7.1 El papel de la abstracción	151
7.2 El tipo abstracto de datos (TAD)	151
7.2.1 Especificaciones de tipos abstractos de datos.....	152
7.2.2 Implementación de tipos abstractos de datos	153
7.3 Orientación a objetos	153
Problemas resueltos básicos	154
Problemas avanzados	159
Problemas propuestos.....	170
Capítulo 8 Listas, listas enlazadas	171
8.1 Estructuras de datos dinámicas	171

8.2 Punteros (apuntadores).....	171
8.3 Variables dinámicas.....	172
8.4 Tipos puntero predefinidos NULL y void	172
8.5 Conceptos generales sobre listas.....	173
8.6 Especificación del tipo abstracto de datos lista	174
8.7 Operaciones sobre listas enlazadas	174
8.8 Especificación formal del tipo abstracto de datos <i>lista ordenada</i>	175
8.9 Inserción y borrado de un elemento en lista enlazada simple	175
Problemas básicos	176
Problemas avanzados	183
Problemas propuestos.....	193
Capítulo 9 Modificaciones de listas enlazadas	195
9.1 Listas doblemente enlazadas	195
9.2 Inserción y borrado de un elemento en lista doblemente enlazada	196
9.3 Listas circulares simplemente enlazadas.....	197
9.4 Listas circulares doblemente enlazadas	199
Problemas básicos	199
Problemas avanzados	207
Problemas propuestos.....	219
Capítulo 10 Pilas y sus aplicaciones	221
10.1 El tipo abstracto de datos pila	221
10.2 Especificación del tipo abstracto de datos pila	222
10.3 Implementación mediante estructuras estáticas	222
10.4 Implementación mediante estructuras dinámicas	222
10.5 Transformación de expresiones aritméticas de notación infija a postfija.....	223
10.6 Evaluación de expresiones aritméticas	224
10.7 Eliminación de la recursividad.....	224
Problemas resueltos básicos.....	225
Problemas resueltos avanzados	229
Problemas propuestos.....	241
Capítulo 11 Colas, colas de prioridad y montículos	243
11.1 Colas	243
11.2 Especificación formal.....	244
11.3 Implementación con variables dinámicas	245
11.4 Colas circulares	247
11.5 Bicolos	247
11.6 Especificación formal de TAD bicola sin restricciones.....	247
11.7 Colas de prioridad	248
11.8 Especificación del tipo abstracto de datos “cola de prioridad”	248
11.9 Montículos	249
Problemas básicos	249
Problemas avanzados	265
Problemas propuestos.....	268
Capítulo 12 Tablas de dispersión y funciones <i>hash</i>	271
12.1 Tablas de dispersión	271
12.2 Funciones de transformación de clave.....	272
12.3 Tratamiento de sinónimos	274
Problemas básicos	275
Problemas avanzados	283
Problemas propuestos.....	285

Capítulo 13 Árboles, árboles binarios y árboles ordenados	287
13.1 Concepto de árbol	287
13.2 Árbol binario	288
13.2.1 Construcción de un árbol binario	288
13.2.2 Recorridos	288
13.3 Árboles binarios de expresiones	291
13.3.1 Construcción a partir de una expresión en notación convencional	292
13.4 Árboles binarios de búsqueda	294
Problemas básicos	295
Problemas avanzados	302
Problemas propuestos.....	316
Capítulo 14 Árboles binarios equilibrados.....	319
14.1. Árbol binario equilibrado, árboles AVL	319
14.2. Inserción en árboles AVL	320
14.3. Rotaciones en la inserción	321
14.4. La eliminación (o borrado en árboles AVL)	323
14.5. Rotaciones en la eliminación	324
Problemas de seguimiento	325
Problemas básicos	333
Problemas avanzados	340
Problemas propuestos.....	343
Capítulo 15 Árboles B	345
15.1 Árboles B	345
15.1.1 Búsqueda de una clave	346
15.1.2 Inserción de información.....	347
15.1.3 Borrado físico de un registro.....	348
15.2 Realización de un árbol B en memoria externa.....	349
15.3 Árboles B*	350
15.4 Árboles B+	350
Problemas de seguimiento	352
Problemas propuestos.....	367
Capítulo 16 Grafos I: representación y operaciones	369
16.1 Conceptos y definiciones	369
16.2 Representación de los grafos.....	370
16.3 Tipo Abstracto de Datos Grafo	371
16.4 Recorrido de un grafo	371
16.5 Componentes conexas	372
16.6 Componentes fuertemente conexas	373
Problemas resueltos básicos	374
Problemas resueltos básicos	389
Problemas propuestos.....	394
Capítulo 17 Grafos II: Algoritmos	397
17.1 Ordenación topológica	397
17.2 Matriz de caminos: Algoritmo de Warshall	398
17.3 Problema de los caminos más cortos con un sólo origen: algoritmo de Dijkstra	398
17.4 Problema de los caminos más cortos entre todos los pares de vértices: algoritmo de Floyd	398
17.5 Concepto de flujo. Algoritmo de Ford Fulkerson.....	399
17.6 Problema del árbol de expansión de coste mínimo	400
17.7 Algoritmo de Prim y algoritmo de Kruskal	401
Problemas resueltos básicos.....	402
Problemas básicos	414

Problemas avanzados	425
Problemas propuestos	431
Índice analítico	433

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 currícula 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 Niklaus Wirtz, y posteriormente en la década de los ochenta con la aparición de su obra –ya clásica– *Algorithms and Data Structures* en 1986.

¿Porqué C y no C++/Java o Visual Basic/C#? Muchas Facultades y Escuelas de Ingeniería, así como Institutos Tecnológicos, comienzan sus cursos de Estructuras de Datos con el soporte de C y muchas otras con el soporte de C++ o Java, fundamentalmente; de hecho, en nuestra propia universidad, en algunas asignaturas relacionadas con esta disciplina se aprende a diseñar y construir estructuras de datos utilizando C++ o Java. ¿Existe una solución ideal? Evidentemente, consideramos que no y cada una de ellas tiene sus ventajas y es la decisión del maestro y profesor quien debe elegir aquella que considera más recomendable para sus alumnos teniendo en cuenta el entorno y contexto donde se desarrolla su labor, al ser él quien llevará la dirección y responsabilidad de la formación de sus alumnos y pensará en su mejor futuro y encuadre dentro del currículo específico de su carrera.

Sin embargo, hay muchas razones por las cuales pensamos que C es más apropiado que C++ o Java, para la introducción y formación a nivel medio, inclusive avanzado, en estructuras de datos, siempre que se recurra al paradigma estructurado, con soporte en tipos abstractos de datos y no al puro enfoque orientado a objetos en cuyo caso C++ o Java tienen, sin duda, todas las ventajas y sería necesario utilizar uno u otro lenguaje, y también como antes el profesor tendría siempre la última palabra. Una de estas razones es evidente y se deduce del propio nombre de los lenguajes. C++ es un C más es decir, un C más amplio y potente que se construyó para manejar complejidad a gran escala. Iguales razones, incluso aumentadas, se puede decir de Java, al ser un lenguaje más moderno, con mejores funcionalidades, orientado a la programación en Web...

El primer problema que se suele presentar al estudiante de *estructura de datos* que, probablemente, procederá de un curso a 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. Aunque es verdad que C++¹ tiene muchas ventajas sobre C, muchas de estas ventajas no se hacen evidentes hasta que un programa se “vuelve” o “hace” más complejo y, si me apuran, más completo. En este caso el paradigma orientado a objetos (POO) es una herramienta de programación y organizativa muy poderosa y con grandes ventajas para la enseñanza y posterior tarea profesional.

Por otra parte, la mayoría de los estudiantes de informática, ciencias de la computación, ingeniería de sistemas o de telecomunicaciones, requieren conocer bien el flujo C-C++ y viceversa. Por consiguiente, parece más natural enseñar primero las estructuras de datos en C y una vez conocidas y mejor dominadas las técnicas de diseño y construcción de estructuras de datos en C, y cuando se tenga constancia de que el alumno dispone de conocimientos, al menos, básicos de POO entonces intentar pasar a C++ o Java. Por otra parte, aunque a primera vista por su enfoque orientado a objetos, C++ podría ser más interesante, en el caso del análisis y diseño de algoritmos y estructuras de datos esta propiedad añade una complejidad inherente, ya que pensamos la idea conceptual de algoritmo encaja mejor en el paradigma estructurado, aunque luego la implementación en clases y objetos, puede darle una nueva potencialidad. Pensando en esta transición es la razón por la cual se ha incluido un capítulo dedicado a conceptos teórico-prácticos de orientación a objetos. En cualquier caso, el curso está soportando la comprensión del Tipo Abstracto de Datos (TAD) de modo que, aunque, se enseñan las estructuras de datos bajo la metodología estructurada, el estilo de programación empleado en el texto se basa en el estudio de tipos abstractos de datos como base para la formación en orientación 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 cade-

¹ Véase otras obras del autor, publicadas también en McGraw-Hill, tales como Programación en C++ o Programación en Java 2

nas 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, y las memorias centrales de 512 MB, son prácticamente usuales en la mayoría de los PC 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 implementar 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 y éstas 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 adaptar 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 crítica 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 cuidadosas 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 en la mayoría de las veces en lenguaje C; para que el lector pueda verificar su programa antes de introducirlo 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 libro de problemas y de prácticas universitarias y profesionales

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, 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 currícula 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 currícula de 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 currícula 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 han 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

Este libro, está concebido como libro didáctico y eminentemente práctico, pensado en la resolución de problemas mediante algoritmos y codificación de los programas correspondientes. Se pretende enseñar los principios básicos requeridos para seleccionar o diseñar las estructuras de datos que ayudarán a resolver mejor los problemas que no a memorizar una gran cantidad de implementaciones. Por esta razón y siguiendo la filosofía de la colección *Schaum*, se presentan numerosos ejercicios y problemas resueltos en su totalidad, siempre organizados sobre la base del *análisis del problema* y el *algoritmo correspondiente en C*. Los lectores deben tener conocimientos a nivel de iniciación o nivel medio en programación. Es deseable haber estudiado al menos un curso de un semestre de introducción a los algoritmos y a la programación, con ayuda de alguna herramienta de programación, preferentemente y se obtendrá el mayor rendimiento si además se tiene conocimiento de un lenguaje estructurado tal como Pascal o 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 libros de teoría o simplemente del curso impartido por el maestro o profesor en su aula de clase. Pensando en su uso totalmente práctico se ha optado por seguir una estructura similar al libro *Algoritmos y Estructura de Datos* publicado en McGraw-Hill, por los profesores Joyanes y Zahonero de modo que incluye muchos de los problemas y ejercicios propuestos en esta obra. En caso de realizar su estudio de este modo conjunto, uno actuaría como libro de texto fundamentalmente y el otro como libro de prácticas para el laboratorio y el estudio en casa o en un curso profesional.

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 curricula del 91 y las actualizadas del 2001 y 2005 de ACM/IEEE, así como de los planes de estudio de los Ingenieros Informáticos e Ingenieros Técnicos en Informática de España y los de Ingenieros de Sistemas y Licenciados en Informática de muchas universidades latinoamericanas. Un resumen de los capítulos del libro se indica a continuación.

Capítulo 1. Algoritmos, Estructuras de Datos y Programas. 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*, completan el capítulo.

Capítulo 2. Análisis de algoritmos. La medida de la eficiencia de un algoritmo es, sin duda, una de las características fundamentales en cualquier programa. El tiempo de ejecución y los resultados dependerán de que esta medida sea rigurosa y fiable. El estudio de la notación *O-grande* junto con el primer análisis de algoritmos básicos de ordenación y búsqueda forman este capítulo tan importante para la realización de programas.

Capítulo 3. Arrays (Listas y tablas), estructuras y uniones en C. La estructura de datos básica más empleada en programación es el *array* (*arreglo* en Latinoamérica). Una revisión completa de este tipo de datos, clasificación, manipulación y utilización, se describen en el capítulo. Pero el lenguaje C tiene otras dos estructuras de datos básicas: las uniones y las estructuras. El concepto, acceso a los datos almacenados en ellas y los diferentes tipos de estructuras y uniones completan el capítulo. También se considera el tipo enumeración, tipos definidos por el usuario *typedef* y los campos de bits como elementos característicos de ayuda a buenos diseños de programas.

Capítulo 4. Recursividad. Una de las propiedades más importantes en el tratamiento de problemas, especialmente matemáticos y científicos, es la recursividad. Muchas situaciones y problemas de la vida diaria tienen naturaleza recursiva. Su concepto, tratamiento y algoritmos de resolución son una necesidad vital en la formación de un programador. Se consideran en el capítulo los algoritmos, y su codificación en C, más conocidos para resolver problemas de naturaleza recursiva *Las Torres de Hanoi*, *Backtracking*, *Salto del Caballo*, *Las Ocho Reinas* o *el Problema de la Selección Óptima*.

Capítulo 5. Algoritmos de búsqueda y ordenación. La ordenación y búsqueda, son dos de las operaciones más frecuentemente utilizadas en programación. Los algoritmos más reconocidos y más eficientes se analizan y describen con detalle en este capítulo: *Burbuja*, *Selección*, *Inserción*, *Shell*, *QuickSort*, junto con otros más avanzados y no tan populares como *MergeSort*, *Radix Sort* o *BinSort*. También se describen los métodos de búsqueda lineal o secuencial y binaria, junto con la búsqueda binaria recursiva.

Capítulo 6. Archivos y algoritmos de ordenación externa. Los archivos (ficheros) son una de las estructuras de datos más utilizadas en problemas de gestión de la información. Una revisión del tipo de dato y los métodos más usuales de procesamiento de datos situados en archivos externos (discos, cintas,...) constituyen este importante capítulo.

Capítulo 7. Tipos abstractos de datos (TAD) y Objetos. El concepto de Tipo Abstracto de Dato como origen del concepto de objeto, base fundamental de la programación moderna, se examina en el capítulo. Además se analizan los conceptos de objetos, reutilización de *software* y una comparación entre el método tradicional de programación (estructurado) y el método moderno de programación (objetos). La implementación de Tipos Abstractos de Datos en C se explica también en este capítulo.

Capítulo 8. Listas y listas enlazadas. Los conceptos de lista y de lista enlazada son vitales en un diseño avanzado de programas, debido fundamentalmente a la inmensa cantidad de organizaciones y estructuras de la vida diaria que tienen o se asemejan al concepto de lista. Las operaciones y algoritmos básicos para manipulación de listas se analizan con detalle.

Capítulo 9. Modificaciones de listas enlazadas. Las listas doblemente enlazadas y circulares son variantes de las listas enlazadas, también, muy empleadas en el importante campo de la programación. Este capítulo las trata en detalle.

Capítulo 10. Pilas y sus aplicaciones. La pila es una estructura de datos simple y cuyo concepto forma parte en un elevado porcentaje de la vida diaria de las personas y organizaciones. El TAD Pila se puede implementar con *arrays* o con punteros, y el capítulo 10 describe ambos algoritmos y sus correspondientes implementaciones en C.

Capítulo 11. Colas, colas de prioridad y montículos. Al igual que las pilas, las colas conforman otra estructura que abunda en la vida ordinaria. La implementación del TAD cola se puede hacer con *arrays*, listas enlazadas e incluso listas circulares. Además, junto con el concepto de montículo (*heap*, en inglés), se analizan detalladamente las colas de prioridad.

Capítulo 12. Tablas de dispersión y funciones *hash*. Las tablas aleatorias *hash* junto con los problemas de resolución de colisiones y los diferentes tipos de direccionamiento conforman este capítulo.

Capítulo 13. Árboles, árboles binarios y árboles ordenados (de búsqueda). Los árboles son, sin duda, una de las estructuras de datos no lineales, más empleadas en informática, tanto para resolver problemas de *hardware* como de *software*. Los árboles de directorios son una de las organizaciones más empleada por cualquier usuario o programador de una computadora. En el capítulo se describen los tipos de árboles más sobresalientes tales como los generales, binarios o binarios de búsqueda.

Capítulo 14. Árboles binarios equilibrados. Un tipo especial de árbol binario, no por ello menos importante, es el árbol binario equilibrado. Su eficiencia y las operaciones que se realizan sobre el mismo se describen en detalle junto con sus algoritmos y sus implementaciones en C.

Capítulo 15. Árboles B. Este tipo de árbol responde a la necesidad de representar diferentes tipos de organizaciones que no responden bien a una implementación eficiente. Su definición, representación, creación recorrido y eliminación de claves, junto con las implementaciones respectivas constituyen la base de este capítulo.

Capítulo 16. Grafos I: representación y operaciones. 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. Un programador de alto nivel no puede dejar de conocer en toda su profundidad la teoría de grafos y sus operaciones

Capítulo 17. Grafos II: algoritmos. Si el campo de los grafos en general, es una necesidad vital en la matemática, en la ingeniería, la toma de decisiones, etc. y sus aplicaciones son numerosísimas y complejas. Por estas razones se requiere conocer las aplicaciones estándar más eficientes. Por ello se tratan en este capítulo problemas tales como: Ordenación topológica, Caminos más cortos, Flujos de fluidos, o Algoritmos clásicos como Prim, Kruskal o Warshall.

Código en C disponible

Los códigos en C de todos los programas importantes de este libro están disponibles en la Web para que puedan ser utilizados directamente y evitar su “teclado” en el caso de los programas largos, o bien simplemente, para ser seleccionados, recorridos, modificados... 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.mhe.es/joyanes>. En esta página también encontrará más materiales y textos complementarios al libro.

AGRADECIMIENTOS

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

En particular, deseamos agradecer, como en otras ocasiones, a todos nuestros/as colegas – y sin embargo amigos/as – de nuestro departamento, Lenguajes y Sistemas Informáticos e Ingeniería de Software de la Facultad y Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca en el *campus* de Madrid, la colaboración que siempre nos prestan en la realización de nuestros libros. A todos ellos y en particular a nuestros colegas de las asignaturas de las áreas de Programación y Estructuras de Datos nuestro reconocimiento y agradecimiento.

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, ya que 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 obra. 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

Los autores
En Madrid, Mayo de 2005

Algoritmos, estructura de datos y programas

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. Se revisan en el capítulo los conceptos básicos de *dato*, *abstracción*, *algoritmos* y *programas*.

1.1. Resolución de problemas de programación

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. El **algoritmo** es la especificación concisa del método para resolver un problema con indicación de las acciones a realizar. Se requiere el almacenamiento de los datos y definir las operaciones que actuarán sobre ellos. El **tipo abstracto de datos (TAD)** es la entidad fundamental para almacenar los datos y definir las operaciones que actúan sobre ellos para resolver los problemas.

1.2. Abstracción de datos

La abstracción de datos se centra en el *conjunto de valores* que pueden tomar esos datos y las *operaciones* que se ejecutan sobre ellos y su implementación se efectúa en módulos independientes denominados **módulos de implementación**. La abstracción identifica los aspectos esenciales de los módulos mediante una parte llamada *vista externa* o *pública* y otra parte que deberá permanecer *oculta* (*vista privada*). El principio de **ocultación de la información** implica los detalles que se ocultan dentro del módulo y que además son inaccesibles. De esta forma, el usuario de un módulo necesita conocer su vista y no se preocupa de su implementación, al contrario que el desarrollador que se preocupa de los aspectos de la implementación.

1.3. Análisis de un problema

El análisis de un problema consiste en definir cuál es el problema y, a continuación, especificar lo que se necesita para resolverlo. Es preciso asegurarse de que el problema está bien definido y plantear claramente las siguientes cuestiones:

- Especificaciones precisas y completas de las entradas de datos necesarias.
- Especificaciones precisas y completas de la salida.
- Cuál es la documentación necesaria.
- Cómo debe ser de rápido el sistema.
- Cómo debe reaccionar ante datos incorrectos.
- Cuándo y cómo debe terminar.
- Cada cuánto tiempo será necesario efectuar cambios en el sistema y qué mejoras es probable haya que introducir, en el futuro, en el mismo.

1.4. Diseño de un programa

La especificación de un sistema indica *lo que* éste debe *hacer*. La etapa de diseño del sistema indica *cómo* ha de hacerse y eso se manifiesta en la construcción de un algoritmo. Por consiguiente, tras formular un planteamiento preciso del problema que debe solucionar el algoritmo, se seguirán los siguientes pasos:

- Elegir una estructura de datos adecuada.
- Dividir el proyecto en módulos utilizando los principios de diseño descendente.
- Determinar si se pueden utilizar subprogramas que ya existen o es preciso construirlos totalmente.
- Indicar la interacción entre módulos siendo importante especificar claramente el flujo de datos entre ellos. Un medio para realizar estas especificaciones y detallar las hipótesis de entrada y salida para cada módulo es escribir una precondición (descripción de las condiciones que deben cumplirse al principio del módulo) y una postcondición (descripción de las condiciones al final de un módulo).
- Utilizar pseudocódigo para especificar los detalles del algoritmo. El **pseudocódigo** es una herramienta excelente que facilita notablemente la codificación.

La fase de diseño de un programa suele llevar bastante tiempo. El resultado final del diseño descendente es una solución que sea fácil de traducir en estructuras de control y estructuras de datos de un lenguaje de programación específico.

EJEMPLO 1.1. *Las precondiciones y postcondiciones describen las condiciones que deben cumplirse a la entrada y a la salida de un módulo. Por ejemplo, se puede describir un procedimiento que ordena una lista (un array) de la forma siguiente:*

```
procedimiento ordenar (E/S lista: A; E entero: n)
{  Ordena una lista en orden ascendente
  precondición: A es un array (arreglo) de N enteros, 1 <= n <= Max.
    Max es el máximo de elementos de la lista.
  postcondición: A[1] <= A[2] <...<= A[n], n es inalterable
}
```

1.5. Implementación (Codificación) de un programa

En la etapa de *implementación* se traducen los algoritmos creados en la fase de diseño a un lenguaje de programación, debiendo seguirse las siguientes reglas:

- Cuando un problema se divide en subproblemas, los algoritmos que resuelven cada subproblema (tarea o módulo) deben ser codificados, depurados y probados independientemente.
- En programas estructurados es innecesario y no resulta conveniente utilizar la sentencia `goto`.
- Deben emplearse términos significativos para los identificadores, usando nombres para denominar a los datos, formas de los verbos *ser* y *estar* para las funciones.
- Las constantes literales se sustituirán por simbólicas o con nombre.
- Resulta necesaria una buena elección del modo de paso de parámetros. Cuando una función ha de devolver valores a la rutina llamadora usará el modo de paso de parámetros **por variable**. Cuando se pasan parámetros por variable cuyos valores permanecen inalterables, el programa se hace más propenso a errores a la hora de efectuar modificaciones en el mismo y por

tanto, en estos casos, a menos que exista una diferencia significativa en eficiencia, los parámetros deberán ser pasados **por valor**.

- Las funciones no deben asignar valores a variables globales ni usar parámetros por variable.
- En general, el uso de variables globales en subprogramas no es correcto. Sin embargo, el uso de la variable global, en sí, no tiene porqué ser perjudicial. Así, si un dato es inherentemente importante en un programa, de forma que casi todo subprograma debe acceder al mismo, ese dato es global por naturaleza.
- El sangrado (*indentación*) y los buenos comentarios facilitan la posterior lectura del código.

EJEMPLO 1.2. *La estructura básica de un programa en C es:*

```
/* declaración de importaciones */
/* definición de constantes */
/* definición de tipos */
int main ()
{
    // declaración de variables
    // instrucciones ejecutables
    return 0;
}
```

1.6. Verificación

Un algoritmo es *correcto* (*exacto*) si se obtiene la solución de la tarea para la cual ha sido diseñado sin pérdida (omisión) de ninguna de las etapas fundamentales. Una línea actual de investigación intenta aplicar las técnicas de la lógica formal para probar la exactitud o corrección de un programa. Es decir, el objetivo es aplicar la lógica formal para probar que el algoritmo representado en un programa hace lo que realmente debe hacer. La tesis fundamental es que reduciendo el proceso de verificación a un procedimiento formal, el programa está protegido de conclusiones no exactas que se pueden asociar con argumentos intuitivos. La verificación formal es compleja cuando se utiliza en programas grandes y requiere conocimientos matemáticos avanzados.

1.6.1. MÉTODO BÁSICO DE VERIFICACIÓN FORMAL

Para demostrar la ausencia de errores en un programa, se ha desarrollado un método para demostrar la corrección o exactitud de un programa. Este método, denominado *verificación formal* implica la construcción de pruebas matemáticas que ayudan a determinar si los programas hacen lo que se supone que han de hacer. Verificar un programa consiste en demostrar formalmente que funciona correctamente. El demostrar que un algoritmo es correcto es como probar un teorema matemático.

Verificar un programa es un proceso sistemático por el cual a partir de unas especificaciones de entrada y de las distintas sentencias en que se descompone, se demuestra que el programa es acorde con la especificación del problema. El diseño descendente, lleva a la descomposición gradual del futuro programa en abstracciones funcionales con sus propias especificaciones. El proceso acaba teóricamente cuando en los “refinamientos” se llega al nivel de sentencia. El proceso finaliza sin embargo, en la práctica cuando el analista/programador ha obtenido las abstracciones funcionales que aseguran la resolución del problema y su codificación de una forma inmediata. Por ejemplo, demostrar que un módulo es exacto (correcto) comienza con las *precondiciones* (axiomas e hipótesis en matemáticas) y muestra que las etapas del algoritmo conducen a las *postcondiciones*. Las especificaciones intermedias de las abstracciones funcionales (etapas del algoritmo) juegan un doble papel:

- Representan de hecho la forma de razonar sobre el problema.
- Sirven de punto de apoyo para la verificación de programas.

1.6.2. DESCRIPCIÓN FORMAL DE TIPOS DE DATOS

Las especificaciones precondición y postcondición son las condiciones que cumplen los datos en el punto del programa en que figuran para cualquier ejecución posible de éste, por lo tanto tienen naturaleza de “propiedades” que se expresan mediante predicados del lenguaje de la lógica de primer orden.

1.6.3. EL LENGUAJE DE LA LÓGICA DE PRIMER ORDEN

El conocimiento básico se puede representar en la lógica en forma de axiomas, a los cuales se añaden reglas formales para deducir cosas verdaderas (teoremas) a partir de los axiomas. La lógica de primer orden introduce un *lenguaje formal (syntax)*, una *semántica*, y un *método de cálculo deductivo* con el que se puede demostrar que los programas funcionan correctamente (sistema formal de Hoare)¹.

Syntax

La syntaxis de un lenguaje “formal” se edifica sobre un alfabeto de símbolos:

- Símbolos de constantes: 3, ‘a’, 25.7, etc...
- Símbolos de variables: x, y, z, i, j,...
- Símbolos de función: +, -, *, /, Div, Mod, o funciones de usuario no booleanas.
- Símbolos de predicado: símbolos de relación (= < > ≤ ≥, etc.), o cualquier función booleana definida por el usuario

A partir de estos símbolos se definen los conceptos de término y de fórmula de la siguiente forma:

Término:

- Cualquier símbolo de constante es un término.
- Cualquier símbolo de variable es un término.
- Si t_1, t_2, \dots, t_n son términos y f es una función que tiene n argumentos $f(t_1, t_2, \dots, t_n)$ es también un término.

Fórmula:

- Si t_1, t_2, \dots, t_n son términos y P es un símbolo de predicado con n argumentos entonces $P(t_1, t_2, \dots, t_n)$ es una fórmula.
- Si F_1 y F_2 son fórmulas, entonces también son fórmulas $\neg F_1, F_1 \wedge F_2, F_1 \vee F_2, (F_1 \Rightarrow F_2)$ son fórmulas.
- Si x es una variable y F_1 es una fórmula, entonces $\forall x F_1, \exists x F_1$ son fórmulas.

Semántica

Los términos y las fórmulas de la lógica de primer orden se interpretan (tiene significado) cuando sus variables tienen valores concretos del tipo al que pertenecen, dentro del modelo semántico en el que están definidas. Esto es, las variables toman un valor en un momento determinado del programa definido por su estado. De esta forma se pueden representar los posibles cómputos de los algoritmos o programas.

EJEMPLO 1.3. En el siguiente fragmento de programa se expresan distintos estados posibles e imposibles, dependiendo de los valores de las variables a y b

	POSIBLES	ESTADOS	IMPOSIBLES
$x = a; y = b;$			
$\text{while } (y > 0)$	$(a, 6)^2, (b, 4) (x, 6), (y, 4)$		$(a, 6), (b, 4) (x, 6), (y, 3)$
{	$(a, 6), (b, 4) (x, 7), (y, 3)$		$(a, 6), (b, 4) (x, 8), (y, 3)$
$x = x + 1;$			
$y = y - 1;$			
}			

¹ El sistema formal de Hoare es un conjunto de axiomas y reglas de inferencia para razonar acerca de la corrección parcial de programa. Establece la semántica axiomática del lenguaje de programación.

² (a,6), significa que la variable tiene almacenado el valor de 6.

Mediante las fórmulas de la lógica de primer orden se puede expresar la semántica (lo que ocurre) en un punto cualquiera del programa, para todos sus posibles estados.

EJEMPLO 1.4. *En el siguiente fragmento de programa se expresan todos los posibles estados mediante las fórmulas del lenguaje de primer orden correspondientes. Siempre que en la precondition de entrada figure que a y b sean positivos y enteros.*

```
                                {a ≥ 0 ∧ b ≥ 0}
x = a; y = b;                  {x = a ∧ y = b ∧ x, y ≥ 0}
while (y > 0)
{
    x = x + 1;
    y = y - 1;
                                {x + y = a + b ∧ x ≥ a ≥ 0 ∧ 0 < y ≤ b}
}
```

Invariantes

La verificación de programas tiene como parte importante la determinación de los invariantes de los bucles. Estos invariantes son predicados expresados informalmente (a través del lenguaje natural inicialmente) o bien formalmente mediante fórmulas de la lógica de primer orden. Estas fórmulas que expresan los invariantes deben cumplirse siempre: antes de la ejecución del bucle; al final de la ejecución de cada iteración del bucle; y al terminar el propio bucle (en este caso se cumple además la negación de la condición de entrada al bucle).

EJEMPLO 1.5. *El siguiente fragmento de programa escrito en C tiene como invariante s es el producto de a por i , ya que se cumple antes, al final de cada iteración del bucle y al terminar el bucle. Puede expresarse mediante las fórmulas de la lógica de primer orden de la siguiente forma: $s = a*i$.*

```
int i, a, s, n = 5
...
...
s = 0;
i = 0;
while (i! = n)
{
    i = i + 1;
    s = s + a;
}
```

PROBLEMAS RESUELTOS BÁSICOS

- 1.1.** *Escriba un pseudocódigo que exprese las precondition y postcondición de un algoritmo que lea un archivo y escriba un informe.*

Análisis

Se resuelve el problema mediante un informe secillo que consiste en poner una cabecera de página, indicar el número de línea, la línea y devolver en un parámetro por referencia el número de páginas que tiene el archivo que se recibe como parámetro.

Pseudocódigo

```
algoritmo TestArchivo(E/S Texto f; E/S entero numPagina)
// El algoritmo lee un archivo lo escribe y retorna el número de páginas
// Pre   Inicializar numPagina
// Post   Informe impreso. numPagina contiene número de páginas del informe
// Retorno Se devuelve número de líneas impresas
constante
  Cabecera_depagina= 'esto es una cabecera'
variables
  entero Nlineas
  cadena linea
inicio
  Abrir (f para leer)
  Nlineas ← 0
  Numpagina ← 0
  Escribir(cabecera_de_pagina)
  Mientras (no ff(f))                                // mientras no sea fin de archivo
  Leer (f, linea)
  si (página completa) entonces
    Avanzar página
    numPogina ← numPágina + 1
    Escribir( cabecera_de_página)
  fin_si
  nlinea ← nlinea+1
  Escribir(nlinea, linea)
fin_mientras
  cerrar (f)
Fin_TestArchivo
```

- 1.2.** *Escriba un programa C que lea 8 números enteros introducidos desde el teclado y calcule su producto. Exprese el invariante del bucle como una aserción hecha en lenguaje natural.*

Análisis

Para escribir el programa, basta con definir una constante *n* que tome el valor 8 y mediante un bucle controlado por el contador *c*, ir leyendo números del teclado en la variable *Numero* y multiplicarlos en un acumulador *Producto*. Por lo tanto, el invariante del bucle debe decir si se han leído *c* números siendo *c* menor o igual que *n* y en *Producto* se han acumulado los productos de los *c* números leídos.

Codificación

```

#include <stdio.h>
#define n 8
void main ( )
{
    int c, Numero, Producto;
    c = 0;
    Producto = 1;
    while(c< n)
    {
        c = c + 1;
        scanf("%d",&Numero);
        Producto = Producto * Numero;
        /* invariante= se han leído c números siendo c menor o igual que n y en Producto
           se han acumulado los productos de los c números leídos*/
    }
    printf("su producto es %d\n", Producto);
}

```

- 1.3. El siguiente segmento de programa está diseñado para calcular el producto de dos enteros estrictamente positivos 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 el programa? Justifique su respuesta.

```

Producto ← y
cuenta ← 1
mientras (cuenta < x) hacer
    producto ← producto + y
    cuenta ← cuenta + 1
fin_mientras

```

Análisis

Antes de comenzar el bucle en `producto` se tiene almacenado una sola vez (el número que contiene `cuenta`) el valor de y . En cada iteración `cuenta` se incrementa en una unidad y `producto` se incrementa en el valor de y , por lo que al final del bucle en `producto` se tiene almacenado tantas veces como indica `cuenta` el valor de y . El bucle itera siempre que `cuenta` sea menor que x , y como x es de tipo entero al igual que `cuenta`, el bucle termina cuando `cuenta` coincide con x . Por lo tanto cuando ha terminado el bucle en `producto` se han sumando tantas veces y como indique `cuenta`, cuyo valor es necesariamente x . Es decir `producto` tiene el valor de $x*y$. El programa es correcto.

- 1.4. Suponiendo la precondition el valor asociado con n es un entero positivo. Fijar un invariante de bucle que conduzca a la conclusión de que la siguiente rutina termina y a continuación asignar a `suma` el valor de $0+1+2+3+\dots+n$.

```

Suma ← 0
i ← 0
mientras (i < n) hacer
    i ← i + 1
    suma ← suma + i
fin_mientras

```

Solución

Antes de comenzar el bucle, `suma` tiene el valor de cero i también el valor de cero. En el bucle se incrementa i en una unidad y posteriormente se acumula en `suma` el valor de i . Por tanto, al final de la iteración de cada bucle en `suma` se tiene alma-

cenado el valor de $1+2+3+\dots+i$. Como antes de comenzar una iteración del bucle, i es menor que n se tiene que cuando termine el bucle, i es mayor o igual que n y como ambas son enteras, necesariamente i debe tomar el valor de n . Por lo tanto, el invariante del bucle es $\text{suma} = 1+2+3+4+\dots+i$ e $i < n$.

- 1.5.** ¿Cuál es el invariante del bucle siguiente escrito en C? Expréselo en una fórmula de la lógica de primer orden.

```
Indice = 0;
Producto = A[0];
while (Indice < N - 1)
{
    Indice = Indice + 1;
    Producto = Producto * A[Indice];
}
```

Análisis

Como ya se sabe el invariante del bucle debe ser un predicado que se cumpla antes de la ejecución del bucle, al final de la ejecución de cada iteración del bucle y, por supuesto, en la terminación. Se supone que A es un vector cuyos valores se puedan multiplicar y que los índices varían en el rango 0 hasta el $N-1$. En este caso el invariante del bucle debe expresar que en el acumulador Producto se han multiplicado los elementos del *array* hasta la posición Indice .

$$INV \equiv \left(\text{Producto} = \prod_{i=0}^{\text{Indice}} A(k) \right) \wedge (\text{Indice} \leq N - 1)$$

Solución

```
Indice = 0;
Producto = A[0];
while (Indice < N-1)
{
    Indice = Indice+1;
    Producto = Producto* A[Indice]
```

$$INV \equiv \left(\text{Producto} = \prod_{i=0}^{\text{Indice}} A(k) \right) \wedge (\text{Indice} \leq N - 1)$$

- 1.6.** Escriba el invariante del siguiente bucle escrito en C. Suponga que $n \geq 0$. Expréselo con una fórmula de la lógica de primer orden.

```
Indice = 0;
Minimo = A[0]
while (Indice != ( n - 1))
{
    Indice = Indice + 1;
    if ( Minimo > A[ Indice ])
        Minimo = A[Indice];
}
```

Análisis

El invariante debe expresar que Minimo contiene el elemento mmenor del *array* desde las posiciones 0 hasta la posición Indice y que además se encuentra en el *array*.

Solución

```
Indice = 0;   Minimo = A[0];
```

$$INV \equiv (0 \leq \text{Indice} \leq n-1) \wedge (\forall k (0 \leq k \leq \text{Indice} \rightarrow \text{Minimo} \leq A(k)) \wedge (\exists k (0 \leq k \leq \text{Indice} \wedge \text{Minimo} = A(k)) \wedge (\text{Indice} \leq n-1))$$

```
while (Indice != (n-1))
{
    Indice = Indice + 1;
    if (Minimo > A[Indice])
        Minimo = A[Indice];
```

$$INV \equiv (0 \leq \text{Indice} \leq n-1) \wedge (\forall k (0 \leq k \leq \text{Indice} \rightarrow \text{Minimo} \leq A(k)) \wedge (\exists k (0 \leq k \leq \text{Indice} \wedge \text{Minimo} = A(k)) \wedge (\text{Indice} \leq n-1))$$

```
}
```

$$INV \wedge (\text{Indice} = n-1)$$
PROBLEMAS RESUELTOS AVANZADOS

- 1.7.** *Escribir un programa en C que calcule el máximo común divisor (mcd) de dos números naturales usando sólo restas. Exprese fórmulas de la lógica de primer orden que indiquen la semántica del programa en cada uno de sus puntos.*

Análisis

Es bien conocido que si x es distinto de cero entonces se tiene:

$$\text{mcd}(x, y) = \text{mcd}(x, y-x) \text{ cuando } x \leq y$$

Si y vale cero entonces $\text{mcd}(x, 0) = x$

De forma análoga se tiene que si y es distinto de cero entonces se tiene:

$$\text{mcd}(x, y) = \text{mcd}(x-y, y) \text{ cuando } y \leq x$$

Por último si vale cero entonces

$$\text{mcd}(0, y) = y$$

De esta forma si se leen dos datos enteros positivos a y b , que son ambos distintos de cero, basta con inicializar las variables x e y a los valores de a y b respectivamente, y mediante un bucle mientras iterar hasta que o bien la variable x o bien la variable y tomen el valor de cero, haciendo que o bien x tome el valor de $x-y$ o bien y tome el valor de $y-x$ dependiendo de que x sea menor o igual que y o no lo sea respectivamente. Además el máximo común divisor estará en la variable y si x toma el valor de cero, o en la variable x si la variable y toma el valor de cero.

Codificación

Una posible codificación en C puede ser:

```
include <stdio.h>
int main()
{
```

```

int x,y,a,b,m;
printf(" dame a >0 y b >0");
scanf(" %d %d",&a,&b);
x = a;
y = b;
while ((x > 0) & (y > 0))
    if (x <= y)
        y = y - x;
    else
        x = x-y;
if (x!=0 )
    m = x;
else
    m = y;
printf("%d %d %d\n",a, b, m);
}

```

A continuación se expresa la secuencia de fórmulas entre las sentencias del algoritmo para expresar la semántica del algoritmo.

```

int main()
{
    int x,y,a,b,m;
    //{a>0 ^b>0}≡ {F1}
    printf(" dame a >0 y b >0");
    scanf(" %d %d",&a,&b);
    //{a>0 ^b>0} ≡ {F2}
    x = a;
    //{x>0 ^b>0^x=a} ≡ {F3}
    y = b;
    //{x>0 ^y>0^x=a^y=b} ≡ {F4} → { mcd(x,y)= mcd(a,b) ^x>0 ^y>0} ≡ {F5}
    while ((x > 0) &(y > 0))
        //{ mcd(x,y) = mcd(a,b) ^ x>0 ^y >0} ≡ {F6}
        if (x <= y)
            //{ x<=y^ mcd(x,y)= mcd(a,b) ^x>0 ^y^0} ≡ {F7}→
            //{ mcd(x,y-x)= mcd(a,b) ^x>0^y-x>=0 }≡ {F8}
            y = y - x;
            //{ mcd(x,y)= mcd(a,b) ^x>0 ^y>=0 } ≡ {F9}
        else
            //{ y<=x^ mcd(x,y)= mcd(a,b) ^x>0 ^y^0} ≡ {F10}→
            //{ mcd(x-y,y)= mcd(a,b) ^x-y>=0^y>0 }≡ {F11}
            x = x - y;
            //{ mcd(x,y)= mcd(a,b) ^x>=0 ^y>0 }≡ {F12}
        if ( x != 0 )
            //{ x= mcd(x,y) ^mcd(x,y)= mcd(a,b)}≡ {F13}
            m = x;
            //{ m= mcd(x,y) ^mcd(x,y)= mcd(a,b)} ≡ {F14}
        else
            //{y=mcd(x,y)^mcd(x,y)= mcd(a,b)} ≡ {F15}
            m = y;
        //{F14}
        //{m= mcd(a,b)}
    printf("%d %d %d\n",a, b, m);
}

```

- 1.8. *Escribir un programa que resuelva el problema de elección por mayoría absoluta que se describe a continuación. Cada votante rellena una papeleta, colocando todos y cada uno de los candidatos a elección de acuerdo con sus preferencias. En el primer recuento de votos, sólo se tiene en cuenta el primer candidato de cada votante. Si ningún candidato obtiene la mayoría absoluta en este recuento, se elimina el candidato con menos votos, y se realiza un recuento en el que se ignoran los votos emitidos para el candidato eliminado; es decir, se cuentan las segundas opciones de aquellos votantes que han optado en primer lugar por el candidato eliminado. Se continúa con las eliminaciones y recuentos de esta forma hasta que algún candidato obtenga la mayoría absoluta, o hasta que todos los candidatos no eliminados tengan exactamente igual cantidad de votos, en cuyo caso se declara un empate.*

Análisis

Los datos del programa deben tomarse de un archivo de texto, en el que el primer dato es el número de candidatos, y posteriormente aparecen los datos de una papeleta de cada votante con su orden de elección correspondiente. Los datos leídos del archivo no pueden almacenarse en un *array* (arreglo) ya que la capacidad de almacenamiento está muy limitada por la memoria del ordenador. En lugar de ello se usa un archivo binario en el que se almacenan todos los datos de cada votante que sean correctos. Este archivo será reinicializado y leído en cada recuento de votos.

Una primera aproximación a la solución del problema es:

```
inicio
  <Inicializar con el archivo de texto los contadores de cada uno de los rivales y el
  archivo de datos>
  <Contar los votos de cada uno de los rivales en liza>
  mientras sea necesario hacer un nuevo recuento hacer
    <Eliminar de entre los rivales el candidato que haya obtenido menos votos>
    <Contar los votos de cada uno de los rivales que queden en liza>
  fin_mientras
  <Dar el informe final>
fin
```

La solución que se presenta se ha estructurado en las siguientes funciones:

- **LeeryValidarUnDato.** Lee del archivo de texto los datos de un votante e informa si la elección efectuada es correcta. Es decir ha elegido a todos los posibles candidatos en un orden determinado (su preferido).
- **AlmacenaDatosvalidos.** Lee el archivo *f* y lo vuelca en el archivo binario, después de haber validado los datos de cada votante. Si una papeleta de un votante no es válida no se escribe en el fichero binario.
- **IniciarRecuento.** Prepara el archivo de texto *f* para ser leído. Lee el número de candidatos, y pone todos los candidatos como rivales entre sí.
- **Inicializar.** Llama a las dos funciones anteriores.
- **EliminarCandidato.** Elimina un candidato como rival del resto, por haber obtenido el menor número de votos.
- **CuentaVotosEInforma.** Usando el archivo binario y los candidatos aún rivales, realiza un nuevo recuento, informando del número de votos que ha obtenido cada uno de los que siguen en liza, así como quienes son los candidatos que más y menos votos han obtenido.
- **InformeFinal.** Informa del resultado final de la votación.
- **El programa principal main.** Llama inicialmente al módulo inicializar, y mediante un bucle *mientras* itera, decidiendo en cada iteración si algún candidato ha obtenido la mayoría absoluta, hay que eliminar a un candidato, o bien ha habido empate entre los candidatos que aún siguen en liza.

Código fuente completo en página web oficial del libro.

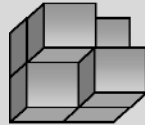
PROBLEMAS PROPUESTOS

1.1. Utilice el método de las aproximaciones sucesivas para resolver el problema del juego de la vida de Conway. Este problema consiste en imaginar una rejilla rectangular en la que cada una de sus casillas puede estar ocupada o no por un microorganismo vivo y, siguiendo las reglas de reproducción y muerte para los microorganismos que se indicarán a continuación, averiguar el estado de la población al cabo de un cierto número de generaciones. Las leyes para la reproducción y muerte de los microorganismos son:

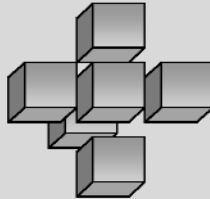
- Si en una celda hay un microorganismo vivo y a su alrededor sólo hay otro vivo o ninguno, muere de soledad.
- Los que tienen dos vecinos se mantienen como están.
- En las celdas con 3 vecinos nace un microorganismo en la próxima generación
- En aquellas que tengan 4, 5, 6, 7 u 8 vecinos el ser vivo que hubiera en ellas muere por hacinamiento.
- Todos los nacimientos y muertes tienen lugar al mismo tiempo.

1.2. El juego de la vida, de Conway puede trasladarse al espacio tridimensional. En él las células, no son cuadradas, sino cúbicas, y tienen 26 vecinos en lugar de ocho. Ciertas versiones análogas al juego de Conway generan fenómenos aún más fantásticos. Entre ellas cabe destacar las versiones Vida 4555 y Vida 5766, de Cartes Bays. Las denominaciones derivan de un sobrio léxico ideado por Bays. Los dos primeros dígitos dictan la suerte de las células vivas. El primero estipula el número mínimo de vecinas vivas que han de rodear a la célula para que no perezca por desnutrición; el segundo, el máximo de las que puede tener sin que la asfixie la superpoblación. El tercer y cuarto dígitos gobiernan la suerte de las casillas muertas. El tercero indica el número mínimo de vecinas vivas que ha de tener la casilla para que cobre vida; y el cuarto, el máximo de las vecinas que puede tener para cobrar vida. En esta notación la vida de Conway es Vida 2333.

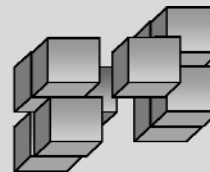
- Escriba un programa para simular la vida tridimensional de Bays, y simule Vida 4555.



Pedestal

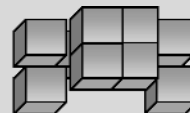
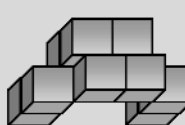
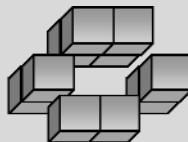
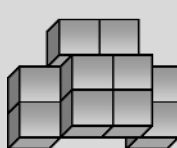


Cruz



Escalón

- Compruebe la naturaleza cíclica de las siguientes figuras.



- Escriba un programa para simular la vida tridimensional de Bays, y simule la Vida 5766.

1.3. Se dispone en un archivo que almacena una tabla de armamento atómico en la que consta para cada arma su denominación, carácter (ofensivo, defensivo) y efectividad (cierto número entero). Existen un total de $n=4$ superpotencias, con cierto arsenal atómico, se trata de obtener una política de alianzas en dos blo-

ques de dos potencias que minimice el riesgo de enfrentamiento. Este riesgo se calcula comparando las potencias ofensivas y defensivas de cada bloque, constituyendo peligro nulo el que la capacidad defensiva de un bloque sea mayor que la ofensiva del contrario, y en el caso de que ambos bloques se

amenacen, se calcula restando del valor absoluto de la diferencia de tales amenazas la décima parte del cuadrado de la menor de ellas, tomándose riesgo cero si tal diferencia fuese negativa.

- 1.4.** A la entrada de un aparcamiento, un automovilista retira un ticket en el cual está indicado su hora de llegada. Antes de abandonar el aparcamiento se introduce el ticket en una máquina que le indica la suma a pagar. El pago se realiza en una máquina automática que devuelve cambio. Escriba un programa que simule el trabajo de la máquina. Se supone que:

- La duración del estacionamiento es siempre inferior a 24 horas.
- La máquina no acepta más que monedas de 1 €, 50, 20, 10, 5, 2, 1 céntimos.
- Las tarifas de estacionamiento están definidas por tramos semihorarios (1/2 hora).

- 1.5.** Se desea diseñar un programa que permita adiestrar a un niño en cálculos mentales. Para ello el niño debe elegir entre las cuatro operaciones aritméticas básicas; la computadora le presentará la operación correspondiente entre dos números, y el niño debe introducir desde el teclado el resultado. El niño dispone de tres tentativas. Caso de acertar, la computadora debe visualizar “Enhorabuena”, y en caso de fallo “Lo siento, inténtalo otra vez”.

- 1.6.** Muchos bancos y cajas de ahorro calculan los intereses de las cantidades depositadas por los clientes diariamente según las siguientes premisas. Un capital de 1.000 euros, con una tasa de interés del 6 por 100, renta un interés en un día de 0,06 multiplicado por 1.000 y dividido por 365, esta operación producirá 0,16 euros de interés y el capital acumulado será 1.000,16. El interés para el segundo día se calculará multiplicando 0,06 por 1.000 y dividiendo el resultado por 365. Diseñar un algoritmo que reciba tres entradas: el capital a depositar, la tasa de interés y la duración del depósito en semanas, y calcule el capital total acumulado al final del periodo de tiempo.

- 1.7.** Escriba un bucle para calcular el menor elemento de un vector y exprese su invariante.

- 1.8.** Escriba un fragmento de programa que decida si un número natural es primo y exprese su invariante. Un número primo sólo puede ser divisible por él mismo y por la unidad.

- 1.9.** Escriba un programa en C que presente en pantalla todas las potencias enteras de 2 que sean menores o iguales que 100. Exprese el invariante del bucle formalmente.

- 1.10.** Escriba un programa en C que calcule la potencia de dos números naturales positivos, usando sólo sumas y pro-

ductos y escriba fórmulas de la lógica de primer orden después de cada sentencia.

- 1.11.** Escriba un programa en C que calcule la parte entera de la raíz cuadrada positiva de un número entero positivo y escriba fórmulas de la lógica de primer orden después de cada sentencia.

- 1.12.** Suponiendo la precondition de que el valor asociado con n es un entero positivo. Fijar un invariante de bucle que conduzca a la conclusión de que el siguiente código termina y a continuación asignar a suma el valor de

```
0+1/1+1/2+1/3+... 1/n
suma ← 0;
i ← 0;
mientras (i < n) hacer
    i ← i + 1
    suma ← suma + 1/i
fin_mientras
```

- 1.13.** El siguiente segmento de programa es un intento de 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 el siguiente programa?

```
Cuenta ← 0;
Resto ← Dividendo;
repetir
    Resto ← Resto - Divisor
    Cuenta ← Cuenta + 1
hasta_que (resto < divisor)
Cociente ← cuenta
```

- 1.14.** 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”.

- 1.15.** 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.

- 1.16.** Escriba un algoritmo que calcule la potencia de un número real elevado a un número natural, exprese el invariante del bucle y realice una verificación formal.

- 1.17.** Escriba un algoritmo que calcule el producto escalar de dos vectores de dimensión n . Escriba el invariante del bucle.

$$a.b = \sum_{i=1}^n a(i)*b(i)$$

Análisis de algoritmos

Las técnicas matemáticas básicas para analizar algoritmos, son fundamentales para el tratamiento de temas avanzados de computación y analizan medios para formalizar el concepto de que un algoritmo es significativamente más eficiente que otros. El análisis de algoritmos es una parte muy importante de las ciencias de la computación para poder analizar los requisitos de tiempo y espacio de un algoritmo para ver si existe dentro de límites aceptables. En este capítulo se formaliza el concepto de eficiencia de los algoritmos, se muestra cómo analizar la eficiencia de algoritmos, para determinar los requisitos de tiempo y espacio.

2.1. Medida de la eficiencia de un algoritmo

Para evaluar un algoritmo se debe considerar:

- Su facilidad de codificación y depuración.
- Su funcionamiento correcto para cualquier posible valor de los datos de entrada.
- Inexistencia de otro algoritmo que resuelva el problema utilizando menos recursos (tiempo en ejecutarse y memoria consumida). El recurso espacio y el recurso tiempo suelen ser contrapuestos.

2.1.1. EVALUACIÓN DE LA MEMORIA

La *complejidad en relación al espacio* de un programa es la cantidad de memoria que se necesita para ejecutarse. La evaluación de la memoria estática es muy fácil de realizar: se calcula sumando la memoria que ocupan las variables simples, los campos de los registros y las componentes de los vectores. La memoria dinámica depende de la cantidad de datos y del funcionamiento del programa. Se calcula de forma similar a como se hace la evaluación del tiempo. Basta con tener en cuenta las órdenes de reserva y liberación de memoria y la posición que ocupan en el programa.

2.1.2. EVALUACIÓN DEL TIEMPO

La *complejidad en relación al tiempo* de un programa es la cantidad de tiempo que se necesita para su ejecución, para lo cual es necesario considerar el "*principio de la invarianza*" (la eficiencia de dos implementaciones distintas de un mismo algoritmo difiere tan sólo en una constante multiplicativa). El enfoque matemático considera el consumo de tiempo por parte del algoritmo como una función del total de sus datos de entrada.

Se define el tamaño de un ejemplar de datos de entrada, como el número de unidades lógicas necesarias para representarlo en la computadora de una manera razonablemente compacta. Para la evaluación de un algoritmo es importante considerar que el tiempo que tarda un programa en ejecutarse depende del tamaño del ejemplar de entrada. Hay que tener en cuenta que el tiempo de ejecución puede depender también de la entrada concreta.

EJEMPLO 2.1. *Al ordenar una lista el tiempo empleado en ello depende del número de elementos de la lista, pero puede variar este tiempo con el orden en el que se encuentran almacenados los elementos de la lista. Por ejemplo, cuando se aplica el siguiente algoritmo para ordenar una lista el número de iteraciones será muy diferente según como la lista se encuentre de ordenada.*

```
// precondition: A es un array de n enteros, 1 ≤ n ≤ Max.

procedimiento burbujaMejorado(E/S arr: A; E entero: n)
var
  entero: Sw, j, k
inicio
  Sw ← 0
  j ← 1
  mientras ((j < n) y (Sw=0)) hacer
    Sw ← 0
    k ← 1
    mientras (k < (n - j)) hacer
      si (A[k+1] < A[k]) entonces
        Aux ← A[k]
        A[k] ← A[k+1]
        A[k+1] ← Aux
      Sw ← 1
    fin-si
  fin_mientras
  j ← j + 1
fin-mientras
fin_procedimiento

// postcondición: A[1] ≤ A[2] ≤ ... ≤ A[n], n es inalterable
```

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, en nuestro caso en C, ejecutadas por el programa, y la evaluación se podrá efectuar desde diferentes puntos de vista:

Peor caso. Se puede hablar de $T(n)$ como el tiempo para el 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. Se habla de $T(n)$ como el tiempo para el mejor caso. Indica el tiempo mejor que se puede 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 del rendimiento en la práctica, pero es, normalmente, mucho más difícil de calcular que el tiempo de ejecución en el caso peor.

2.2. Notación O-GRANDE

Si se ha escrito y depurado un programa y se ha seleccionado la entrada específica de tamaño n para considerar su ejecución. El tiempo de ejecución de un programa se expresa normalmente utilizando la notación "*O-grande*" que está diseñada para expresar factores constantes tales como:

- Número medio de instrucciones máquina que genera un compilador determinado.
- Número medio de instrucciones máquina por segundo que ejecuta una computadora específica.

Así, se dirá que un algoritmo determinado 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 ".

2.2.1. DESCRIPCIÓN DE TIEMPOS DE EJECUCIÓN

Sea $f(n)=T(n)$ el tiempo de ejecución de algún programa, medido como una función de la entrada de tamaño n . Sea $f(n)$ una función definida sobre números naturales. Se dice " $f(n)$ es $O(g(n))$ ", si $f(n)$ es menor o igual que una constante de tiempo c multiplicada por $g(n)$, excepto posiblemente para algunos valores pequeños de n . De modo más riguroso, se dice $f(n) \in O(g(n))$ si existe un entero n_0 y una constante real $c > 0$ tal que si un natural $n \geq n_0$ se tiene que $f(n) \leq c \cdot g(n)$.

2.2.2. DEFINICIÓN CONCEPTUAL

Específicamente, la notación $f(n) = O(g(n))$ significa que $|f(n)| \leq c|g(n)|$ para $n \geq n_0$. Por consiguiente $|g(n)|$ es un límite superior para $|f(n)|$. La función que se suele considerar es la más próxima que actúa como límite de $f(n)$.

EJEMPLO 2.2

Si $f(n) = 2n^2 + 3n - 1$, entonces

$$f(n) \geq 3.5 n^2 \text{ para } n \geq 2; \quad f(n) \leq 2n^3 \text{ para } n \leq 2; \quad f(n) \leq n^4 \quad \text{para } n \geq 2.$$

Por consiguiente, $f(n) = O(n^2)$, $f(n) = O(n^5)$ y $f(n) = O(n^6)$ y, naturalmente, se continúa hasta potencias superiores.

2.2.3. DEFINICIÓN FORMAL

Se dice que $f(n) = O(g(n))$ si existen constantes $c > 0$ y n_0 tales que para $n \geq n_0$ $f(n) \leq c \cdot g(n)$

Es decir $f(n)$ es $O(g(n))$ si

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c < \infty$$

2.2.4. PROPIEDADES DE LA NOTACIÓN O

De la definición formal dada anteriormente se deducen las siguientes propiedades de la notación O , aplicando simplemente las propiedades de los límites.

1. $c \cdot O(f(n)) = O(f(n))$
2. $O(f(n)) + O(f(n)) = O(f(n))$
3. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
4. $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
5. $O(O(f(n))) = O(f(n))$
6. $\text{Max}(O(f(n)), O(g(n))) = O(\text{Max}(f(n), g(n)))$
7. $O(\log_a(n)) = O(\log_b(n))$ para $a, b > 1$
8. $O(\log(n!)) = O(n \cdot \log(n))$
9. Para $k > 1$ $O(\sum_{i=1}^n i^k) = O(n^{k+1})$
10. $O(\sum_{i=1}^n i^{-1}) = O(\sum_{i=1}^n 1/i) = O(\log(n))$
11. Para todo $c > 1$ y $a \neq b$ $O(c^{\log_a(n)}) \neq O(c^{\log_b(n)})$
12. $O(\sum_{i=1}^n (\log(i))) = O(n \log(n))$

EJEMPLO 2.3. Las propiedades expuestas son fácilmente demostrables a partir de las definiciones dadas.

$$O(\log_a(n)) = O(\log_b(n)) \text{ para } a, b > 1$$

$$O(\log_a(n)) = O(\log_b(n) / \log_b(a)) = O((1 / \log_b(a)) \cdot \log_b(n)) = O(c \cdot \log_b(n)) = O(\log_b(n))$$

2.2.5. COMPLEJIDAD DE LAS DISTINTAS SENTENCIAS Y PROGRAMAS EN C

Para estudiar la complejidad de los **algoritmos iterativos** han de tenerse en cuenta las siguientes consideraciones:

- Las asignaciones, lecturas, escrituras y comparaciones son todas de orden 1, $O(1)$, excepto que sean tipos estructurados de datos.
- La complejidad de una secuencia es la suma de las complejidades de las sentencias que la forman.
- La complejidad de una selección es igual a 1 más el máximo de la complejidad de cada una de las partes que forman la selección.

- La complejidad de un bucle se calcula como la suma de las complejidades para cada iteración de las instrucciones de dentro del bucle. Ha de estimarse el número de ejecuciones para el peor caso.
- La llamada a una función es de orden 1 (constante) siempre que los parámetros sean todos simples. En caso de que los parámetros sean estructurados, y se transmitan por valor, hay que considerar el tiempo de transmisión de los parámetros.
- El cálculo de la complejidad de un algoritmo que contenga llamadas a distintas funciones, se calcula sumando la complejidad de cada una de las funciones que son llamadas.

El análisis de los **algoritmos recursivos** es bastante distinto de los iterativos; suele implicar la solución de ecuaciones en diferencias finitas, que son en general difíciles. Para estudiar la complejidad de los algoritmos recursivos se emplean sobre todo los tres métodos siguientes:

1. **La inducción matemática.** Se basa en suponer una solución, y mediante una inducción constructiva desarrollar la solución y mediante la técnica de inducción demostrativa demostrar que es cierta. Ver ejercicios 2.20.
2. **Expansión de recurrencias.** Consiste en sustituir la recurrencia por su igualdad hasta llegar a un caso conocido. Posteriormente se aplican fórmulas conocidas para encontrar la solución. Ver ejercicios. 2.6, 2.7, 2.8, 2.15.
3. **Ecuaciones en diferencias finitas.** Se usan soluciones generales conocidas y se comparan con las que se obtienen.

2.2.6. FUNCIONES DE COMPLEJIDAD DE ALGORITMOS MÁS COMÚNMENTE CONSIDERADAS

- $O(1)$. Complejidad constante. La más deseada. Aparece en algoritmos sin bucles.
- $O(\log(n))$. Complejidad logarítmica. Es una complejidad óptima. Aparece en la búsqueda binaria.
- $O(n)$. Complejidad lineal. Es muy buena y muy usual. Aparece en los bucles simples. Lectura de un vector.
- $O(n \cdot \log(n))$. Aparece en algoritmos recursivos con un bucle simple y dos llamadas recursivas de tamaño mitad. Por ejemplo en el método de ordenación por mezcla directa.
- $O(n^2)$. Complejidad cuadrática. Bucles anidados dobles. Lectura de una matriz cuadrada.
- $O(n^3)$. Complejidad cúbica. Bucles anidados triples. Para n grande crece excesivamente. Multiplicación de dos matrices. Algoritmo de Floyd.
- $O(n^k)$. Complejidad polinómica. Para $k \geq 3$ crece demasiado rápidamente.
- $O(2^n)$. Complejidad exponencial. Aparece en algoritmos recursivos, cuyo tamaño del ejemplar disminuye en sólo una unidad en cada llamada, y que tienen dos llamadas recursivas (Torres de Hanoi).
- $O(k^n)$. Para $k > 2$. Aparece en algoritmos recursivos, cuyo tamaño del ejemplar disminuye en sólo una unidad en cada llamada, y que tienen k llamadas recursivas (caso del problema del caballo $k = 8$, laberinto $k = 4$).

2.2.7. TABLA COMPARATIVA DE LAS DISTINTAS FUNCIONES DE COMPLEJIDAD MÁS USUALES

La siguiente tabla muestra los distintos valores de las funciones de complejidad más usuales para algunos datos concretos de n .

n	$\log(n)$	n	$n \cdot \log(n)$	n^2	n^3	n^4	2^n
1	0	1	0	1	1	1	2
5	0,7	5	3,5	25	125	625	32
10	1,0	10	10	100	1.000	10.000	1024
20	1,3	20	26	400	8.000	160.000	1.048.576
50	1,7	50	85	25.000	125.000	$6,25 \cdot 10^6$	$1,12590 \cdot 10^{15}$
100	2,0	100	200	10.000	$1,0 \cdot 10^6$	$1,0 \cdot 10^9$	$1,26765 \cdot 10^{30}$
200	2,3	200	460	40.000	$8,0 \cdot 10^6$	$1,6 \cdot 10^9$
500	2,7	500	1349	250.000	$1,25 \cdot 10^8$	$6,25 \cdot 10^{10}$
1.000	3,0	1.000	3.000	$1,0 \cdot 10^6$	$1,0 \cdot 10^9$	$1,0 \cdot 10^{12}$
10.000	4,0	10.000	40.000	$1,0 \cdot 10^8$	$1,0 \cdot 10^{12}$
100.000	6,0	100.000	600.000	$1,0 \cdot 10^{10}$

2.2.8. INCONVENIENTES DE LA NOTACIÓN O-GRANDE

Un inconveniente de la notación *O-grande* es que simplemente proporciona un límite superior para la función. Por ejemplo, si f es $O(n^2)$, entonces f también es $O(n^2 + 5n + 3)$, $O(n^5)$ y $O(n^{10} + 3)$. Otro inconveniente de la notación *O-grande* es que aproxima el comportamiento de una función sólo para argumentos arbitrarios grandes.

PROBLEMAS BÁSICOS

2.1. Probar las siguientes afirmaciones utilizando la definición de la notación *O grande*.

para $n \geq 1$

a. $n^2 + 2n + 1$ es $O(n^2)$

b. $n^2 (n+1)$ es $O(n^3)$

c. $n^2 + (1/n)$ es $O(n^2)$

Para $r \geq 10$

d. $r + r \log r$ es $O(r \log r)$

para $x \geq 1$

e. $x^2 + \ln x$ es $O(x^2)$

Solución

para $n \geq 1$ se tiene

a. $n^2 + 2n + 1$ es $O(n^2)$

b. $n^2 (n+1)$ es $O(n^3)$

c. $n^2 + (1/n)$ es (n^2)

$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = (1+2+1)n^2 = O(n^2)$

$n^2 (n+1) = n^3 + n^2 \leq (1+1) n^3 = O(n^3)$

$n^2 + (1/n) \leq n^2 + 1 \leq (1+1)n^2 = O(n^2)$

para $r \geq 10$

d. $r + r \log r$ es $O(r \log r)$

$r + r \log r \leq r \log r + r \log r = (1+1)r \log r = O(r \log r)$

para $x \geq 1$

e. $x^2 + \ln x$ es $O(x^2)$

$x^2 + \ln x \leq x^2 + x^2 = (1+1)x^2 = O(x^2)$

2.2. Exprese el orden de las siguientes funciones en notación *O grande*.

a. $n(n+1)/2$.

c. $e^{-2n} - n(n+1)$.

e. $n^4 (n^2 - 1)$.

g. $n^2 \ln n + e^{-n} - 6n^8$.

b. $(1/n) - n(n+1)$.

d. $n^2 + 5n \ln n$

f. $n \ln n + e^n - 6e^{-8}$.

h. $n^2 \ln n + e^n - 6e^8$

Solución

a. $n(n+1)/2$. es $O(n^2)$

b. $(1/n) - n(n+1)$. es $O(n^2)$

c. $e^{-2n} - n(n+1)$. es $O(n^2)$

d. $n^2 + 5n \ln n$. es $O(n^2)$

e. $n^4 (n^2 - 1)$. es $O(n^2)$

f. $n \ln n + e^n - 6e^{-8}$. es $O(n^2)$

g. $n^2 \ln n + e^{-n} - 6n^8$. es $O(n^2)$

h. $n^2 \ln n + e^n - 6e^8$. es $O(n^2)$

2.3. Suponga un algoritmo que tarda 5 segundos para resolver un ejemplar de un determinado problema de tamaño $n=10000$. ¿Cuánto tardará en resolver un ejemplar de tamaño $n=30000$ para los casos: a) $O(n^2)$; b) $O(n^5)$; c) $O(n \log n)$; d) $O(2^n)$; e) $O(5^n)$?

Solución

a) $(30000/10000)^2 * 5$ segundos = 45 segundos.

b) $(30000/10000)^5 * 5$ segundos = 1215 segundos = 20,25 minutos.

c) $((30000) \log (30000)) / (10000 \log (10000)) * 5 = 16,8$ segundos.

d) $2^{30000} / 2^{10000} * 5 = 2^{20000} * 5$ segundos.

e) $5^{30000} / 5^{10000} * 5 = 5^{20000} * 5$ Segundos.

- 2.4. Un algoritmo *A* resuelve un problema de tamaño n en tiempo n^2 horas. Otro algoritmo *B* resuelve el mismo problema de tamaño n en n^3 milisegundos. Determine el umbral n_0 a partir del cual el algoritmo *A* es mejor que el *B*.

Solución

1 hora = $1 * 60$ minutos = $60 * 60$ segundos = $60 * 60 * 1000$ milisegundos = 3600000 milisegundos.

Para que el algoritmo *A* sea mejor que el algoritmo *B*, debe ocurrir que $n^2 * 3600000 < n^3$. Por lo tanto $3600000 < n$. De esta forma el umbral n_0 a partir del cual es mejor el algoritmo *A* que el *B* es para un tamaño de 3600000. ¡El ejemplar debe ser de tamaño mayor que 3 millones y medio!

- 2.5. Teniendo en cuenta que en todo el ejercicio deberá poner 1 en lugar de $O(1)$, es decir que en todas las sentencias cuya complejidad sea constante deberá considerar que tardan tiempo 1. Analice el tiempo de ejecución de los siguientes fragmentos de programa.

a. $z = 0;$
 for ($i = 1; i \leq n; i++$)
 $z++;$

b. $i = 1; x = 0;$
 while ($i \leq n$)
 {
 $x++;$
 $i += 3;$
 };

c. $i = 1; x = 0;$
 while ($i \leq n$)
 {
 $x += 3;$
 $i *= 3;$
 };

d. $i = 1; x = 0;$
 do
 { $j = 1;$
 while ($j \leq n$)
 {
 $x++;$
 $j *= 2;$
 }
 $i++;$
 }
 while ($i \leq n$);

e. $x = 0;$
 for ($i = 1; i \leq n; i++$)
 for ($j = 1; j \leq n; j++$)
 $x += 2;$

f. $x = 0;$
 for ($i = 1; i \leq n; i++$)
 for ($j = 1; j \leq n; j++$)
 for ($k = 1; k \leq n; k++$)
 $x += 2;$

g. $x = 0;$
 for ($i = 1; i \leq n; i++$)
 for ($j = 1; j \leq n; j++$)
 for ($k = 1; k \leq j; k++$)
 $x += 2;$

h. $x = 0;$
 for ($i = 1; i \leq n; i++$)
 for ($j = 1; j \leq i; j++$)
 for ($k = 1; k \leq j; k++$)
 $x += 2;$

i. for ($i = 1; i \leq n; i++$)
 { $j = 1;$
 while ($j \leq i$)
 { $j *= 2;$
 $x += 2;$
 }
 }

Solución

a. La sentencia de dentro del bucle es de orden 1. Por lo tanto: $T(n) = \sum_{k=1}^n 1 = n \in O(n)$

b. Las sentencias de dentro del bucle son de orden 1. Por lo tanto: $T(n) = \sum_{k=1,4,7}^n 1 = \frac{n}{3} \in O(n)$

c. Para estudiar la complejidad se modifica el algoritmo añadiendo una variable auxiliar t de la siguiente forma:

```
i = 1; x = 0; t = 0;
while (i <= n)
{
```

```

    x += 3;
    i *= 3;
    t++;
};

```

De esta forma se tiene que $i = 3^t$. Por lo tanto cuando termine el bucle mientras (while) se tiene que $3^t > n$ y $3^{t-1} \leq n$. Es decir que $t-1 \leq \log_3(n)$ y $t > \log_3(n)$. Así la complejidad del algoritmo es $O(\log_3(n))$.

d. Utilizando el resultado obtenido en el ejercicio anterior y teniendo en cuenta el bucle externo se tiene

e. La sentencia más interior de los bucles es de orden 1. Por lo tanto: $T(n) = \sum_{i=1}^n \sum_{j=1}^n 1 = \sum_{i=1}^n n = n*n = n^2 \in O(n^2)$

f. La sentencia de los bucles es de orden 1. Así: $T(n) = \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n 1 = \sum_{i=1}^n \sum_{j=1}^n n^2 = n^2 = n^3 \in O(n^3)$

g. Las sentencias de los bucles son de orden 1.

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^j 1 = \sum_{i=1}^n \sum_{j=1}^n j = \sum_{i=1}^n \frac{n*(n+1)}{2} = n* \frac{n*(n+1)}{2} \in O(n^3)$$

h. La sentencia más interior de los bucles es de orden 1.

$$T(n) = \sum_{i=1}^n \sum_{j=1}^i j = \sum_{i=1}^n \frac{i*(i+1)}{2} = \sum_{i=1}^n \frac{i*i}{2} + \sum_{i=1}^n \frac{1i}{2} = \frac{n*(n+1/2)*(n+1)}{6} + \frac{n*(n+1)}{4} \in O(n^3)$$

i. Para estudiar la complejidad se modifica el algoritmo añadiendo una variable auxiliar t de la siguiente forma :

```

x = 0;
for (i = 1; i <= n; i++)
{
    j=1;
    k=0;
    while(j <= i)
    {
        j *= 2;
        k++;
        x += 2;
    }
}

```

De esta forma se tiene que $j = 2^k$. Por lo tanto cuando termine el bucle mientras, tendremos que $2^k > j$ y $2^{k-1} \leq j$. Es decir que $k-1 \leq \log_2(j)$ y $k > \log_2(j)$. Así la complejidad del bucle mientras $\log_2(j)$. Por lo tanto:

$$T(n) = \sum_{j=1}^n \log_2(j) \in O(n \log(n))$$

2.6. Analice el tiempo de ejecución de las siguientes funciones recursivas.

```

a. int f1 (int n: integer)
{
    if (n <= 1)
        return(1);
    else
        return (n * f1(n - 1))
}

```

```

b. int f2 (int n)
{
    if (n <= 1)
        return(1);
    else
        return (n * f1(n/2))
}

```

```
c. int f3 (int n)
{
    int x,i;
    if (n <= 1)
        return(1);
    else
    {
        x = f3(n / 2);
        for(i = 1; i <= n; i++)
            x++;
        return(x);
    }
}
```

```
d. int f3 (int n)
{
    int x,i;
    if (n <= 1)
        return(1);
    else
    {
        x=f3(n - 1);
        for(i = 1; i <= n; i++)
            x++;
        return(x);
    }
}
```

```
e. int f5 (int n)
{
    if (n <= 1)
        return (1);
    else
        return (f5(n - 1) + f5(n - 1))
}
```

```
f. int f6 (int n)
{int x, i;
  if (n <= 1)
    return (1);
  else
  {x = f6(n / 2) + f6(n / 2);
   for (i = 1; i <= n; i++)
       x++;
   return (x);
  }
}
```

```
g. int f7 (int n)
{ int x, i,j;
  if (n <= 1)
    return (1);
  else
  {
    x = f7(n / 2) + f7(n / 2);
    for (i = 1; i <= n; i++)
        for(j = 1; j <= n; j++)
            x +=j ;
    return (x);
  }
}
```

En todo el ejercicio, se usa $O(1)$. Es decir todas las sentencias cuya complejidad sea constante, se dice que tardan tiempo 1.

Solución

a. Sólo hay una llamada recursiva, y por tanto la recurrencia es:

$$T(n) = 1 + T(n-1) \text{ si } n > 1 \text{ y } 1 \text{ en otro caso.}$$

De esta forma, aplicando expansión de recurrencias se tiene:

$$T(n) = 1 + T(n-1) = 1 + 1 + T(n-2) = \dots = 1 + 1 + 1 + \dots + 1(n \text{ veces}) = n \in O(n).$$

b. Se tiene una llamada. Por tanto la recurrencia es:

$$T(n) = 1 + T(n/2) \text{ si } n > 1 \text{ y } 1 \text{ en otro caso.}$$

Por expansión de recurrencias se tiene:

$$T(n) = 1 + T(n/2) = 1 + 1 + T(n/4) = \dots = 1 + 1 + \dots + 1 \text{ (k veces} = \log_2(n)) = \log_2(n) \in O(\log_2(n))$$

c. Sólo hay una llamada recursiva, y dentro de cada recursividad de tamaño n hay una iteración que tarda tiempo n por tanto la recurrencia es: $T(n) = n + T(n/2)$ si $n > 1$ y 1 en otro caso.

$$T(n) = n + T(n/2) = n + n/2 + T(n/4) = \dots = n + n/2 + n/4 + \dots + 4 + 2 + 1 =$$

$$n \sum_{k=0}^{\log_2(n)} \frac{1}{2^k} = n \left(\frac{1 - \frac{1}{2^{n+1}}}{1 - \frac{1}{2}} \right) = 2n - 1 \in O(n)$$

d. Sólo hay una llamada recursiva, y una iteración que tarda tiempo n y la recurrencia es:

$$T(n) = n + T(n-1) \text{ si } n > 1 \text{ y } 1 \text{ en otro caso.}$$

$$\text{Así } T(n) = n + T(n-1) = n + n-1 + T(n-2) = \dots = n + n-1 + n-2 + \dots + 2 + 1 = \frac{n(n+1)}{2} \in O(n^2)$$

e. Hay dos llamadas recursivas, y una sentencia que tarda tiempo 1 por lo tanto la recurrencia es:

$$T(n) = 1 + 2T(n-1) \text{ si } n > 1 \text{ y } 1 \text{ en otro caso.}$$

De esta forma, aplicando expansión de recurrencias se tiene:

$$T(n) = 1 + 2T(n-1) = 1 + 2 + 4T(n-2) = \dots = 1 + 2 + 4 + 2^3 + \dots + 2^{n-1} = 2^n - 1 \in O(2^n)$$

f. Hay dos llamada recursivas, y una iteración que tarda tiempo n por lo tanto la recurrencia es:

$$T(n) = n + 2T(n/2) \text{ si } n > 1 \text{ y } 1 \text{ en otro caso.}$$

De esta forma, aplicando expansión de recurrencias se tiene:

$$T(n) = n + 2T(n/2) = n + 2n/2 + 4T(n/4) = \dots = n + n + n + \dots + n \text{ (k} = \log_2(n) \text{ veces)} = n \log_2(n) \in O(n \log_2(n))$$

g. Hay una llamada recursiva, y una doble iteración que tarda tiempo n^2 por lo que la recurrencia es:

$$T(n) = n^2 + 2T(n/2) \text{ si } n > 1 \text{ y } 1 \text{ en otro caso.}$$

Aplicando expansión de recurrencias se tiene:

$$n^2 \sum_{k=0}^{\log_2(n)} \frac{1}{2^k} = n^2 \left(\frac{1 - \frac{1}{2^{n+1}}}{1 - \frac{1}{2}} \right) = 2n^2 - 1$$

$$T(n) = n^2 + 2T(n/2) = n^2 + 2 \left(\frac{n^2}{2} \right) + 4T(n/4) = n^2 + \frac{n^2}{2} + \frac{n^2}{2^2} + 2^3 T \frac{n^2}{2^3} = \dots =$$

$$n^2 \sum_{k=0}^{k=\log(n)} \frac{1}{2^k} = n^2 \left(\frac{\frac{1}{2n} - 1}{\frac{1}{2}} \right) \in O(n^2)$$

- 2.7.** Calcule el tiempo de ejecución de los procedimientos recursivos para recorrer un árbol binario en inorden, preorden y postorden.

Solución

En las tres funciones hay dos llamadas recursivas, y una sentencia que tarda tiempo 1 (véase el tema de árboles) por lo tanto la recurrencia es:

$T(n) = 1 + 2T(n-1)$ si $n > 1$ y 1 en otro caso. De esta forma, aplicando expansión de recurrencias se tiene:

$$T(n) = 1 + 2T(n-1) = 1 + 2 + 4T(n-2) = \dots = 1 + 2 + 4 + 2^3 + \dots + 2^{n-1} = 2^n - 1 = 2^n - 1 \in O(2^n)$$

Se supone que n es la profundidad del árbol.

- 2.8.** Escriba una función para calcular el número combinatorio y $\frac{n!}{m!(n-m)}$ calcule su complejidad.

Solución (Se encuentra en la página web del libro)

- 2.9.** Escriba una función que determine el menor elemento de un vector de n elementos y calcule su complejidad.

Solución

Un algoritmo clásico para calcular el menor elemento de un vector mediante un algoritmo voraz de izquierda a derecha es el siguiente:

```
float Menor(float a[], int n)
{
    int i;
    float m;
    m = a[0];
    for (i = 1; i < n; i++)
        if (m > a[i])
            m = a[i];
    return (m);
}
```

La sentencia más interior del bucle es de orden 1. Por lo tanto: $(n) = \sum_{k=1}^n 1 = n \in O(n)$

- 2.10.** Escriba una función que evalúe un polinomio de grado n en el punto x . Calcule su complejidad.

Análisis

La función evaluar que se codifica, está basada en el cálculo de las sucesivas potencias de x , en el mismo bucle donde se calcula la suma.

Solución

```
float Evaluar(float a[], int n, float x)
/*los coeficientes del polinomio vienen dados en el array a, de tal manera que el
coeficiente de xi se encuentra en a[i]. El grado del polinomio es n */
{
    int i;
    float Suma, Pro;
    Suma = 0;
    Pro = 1;
    for (i = 0; i <= n; i++)
    {
        /*en cada iteración Pro = xi */
        Suma = Suma + Pro * A[i];
        Pro *= x;
    }
    return (Suma);
}
```

La sentencia más interior de los bucles son de orden 1. Por tanto $T(n) = \sum_{i=1}^n 1 = n \in O(n)$

- 2.11.** Escriba una función que calcule la suma de dos matrices cuadradas de orden n . Calcule su complejidad.

Solución

Si a , y b son dos matrices cuadradas y c es la suma, se sabe que $c[i,j] = a[i,j] + b[i,j]$ para todo i y todo j , por lo que una función que sume las matrices a y b y deje el resultado en c es el siguiente:

```
void SumaMatrices(float a[][max], float b[][max], float c[][max], int n)
/* se supone que max es una constante previamente declarada */
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

Cálculo de la complejidad:

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n 1 = \sum_{i=1}^n n = n * n = n^2 \in O(n^2)$$

- 2.12.** Escriba una función que calcule el producto de dos matrices cuadradas de orden n . Calcule su complejidad.

Solución

Si a , y b son dos matrices cuadradas y c es la matriz producto, se sabe que $c[i,j] = \sum_{k=0}^{n-1} a[i,k] * b[k,j]$ para todo i

y todo j , por lo que una función que sume las matrices a y b y deje el resultado en c es el siguiente:

```
void Producto(float a[][max], float b[][max], float c[][max], int n)
/* se supone que max es una constante previamente declarada */
{
```

```

int i, j, k;
float aux;
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
        aux=0;
        for (k = 0; k < n; k++)
            aux += a[i][k] * b[k][j];
        c[i][j] = aux;
    }
}

```

La sentencia más interior de los bucles es de orden 1. Por tanto:

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n 1 = \sum_{i=1}^n \sum_{j=1}^n n = \sum_{i=1}^n n * n = n^3 \in O(n^3)$$

2.13. *Determinar la complejidad de la función:*

```

void traspuesta(float d[n][n])
{
    int i, j, t;
    for (i = n - 2; i > 0; i--)
    {
        for (j = i + 1; j < n; j++)
        {
            t = d[i][j];
            d[i][j] = d[j][i];
            d[j][i] = t;
        }
    }
}

```

Solución

El número total de iteraciones es: $\sum_{k=1}^{n-1} k = O(n^2)$

2.14. *Analice el algoritmo de Warshall.*

Solución

El algoritmo de *Warshall* se explica y codifica en el capítulo 17 (Grafos II: Algoritmos).

Análisis de la complejidad

El algoritmo tiene dos bucles anidados de inicialización, y tres bucles anidados de cálculos. Por lo tanto la complejidad del algoritmo es:

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n 1 + \sum_{k=1}^n \sum_{i=1}^n \sum_{j=1}^n 1 = n^2 + n^3 = O(n^3)$$

Es decir la complejidad del algoritmo de *Warshall* es cúbica.

2.15. *Analice el algoritmo de Dijkstra.*

Solución

El análisis se efectúa sobre la codificación mostrada en el capítulo 17 (Grafos II: Algoritmos).

Análisis de la complejidad

Se estudia en primer lugar la complejidad de la función *Mínimo*. Esta función sólo tiene un bucle. Por tanto la función de tiempo será

$$T(n) = \sum_{i=2}^n 1 = n - 1 = O(n)$$

La función *Dijkstra*, tiene: una inicialización mediante un bucle *desde*, y un bucle que hace una llamada a la función *mínimo* y anidado con él otro bucle que actualiza el vector *D*, así como operaciones sobre conjuntos. Se considera que estas operaciones son constantes.

En este caso, la complejidad viene dada por:

$$T(n) = \sum_{i=1}^n 1 + \sum_{i=1}^{n-1} (n + \sum_{k=2}^n 1) = \sum_{i=1}^{n-1} 2n - 1 = (2n - 1)(n - 1) = O(n^2)$$

Por tanto la complejidad del algoritmo de Dijkstra con la implementación de matrices es de orden cuadrático.

2.16. Analice el algoritmo de Floyd**Solución**

Véase la codificación mostrada en el capítulo 17 (*Grafos II: Algoritmos*)

Análisis de la complejidad

El algoritmo tiene dos bucles anidados de inicialización, y tres bucles anidados de cálculos. Por lo tanto la complejidad del algoritmo es:

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n 1 + \sum_{k=1}^n \sum_{i=1}^n \sum_{j=1}^n 1 = n^2 + n^3 = O(n^3)$$

Es decir la complejidad del algoritmo de Floyd es cúbica.

2.17. Analice el algoritmo de Prim.**Solución**

Usando la el algoritmo codificado en el capítulo 17 (*Grafos II: Algoritmos*), para analizar la complejidad, se estudia en primer lugar la complejidad de la función *mínimo*. Esta función sólo tiene un bucle. Por tanto la función de tiempo será:

$$T(n) = \sum_{i=2}^n 1 = n - 1 = O(n)$$

La función *Prim*, tiene: una inicialización mediante dos bucles anidados, y un bucle que hace una llamada a la función *mínimo* y anidado con él otro bucle que actualiza el vector *D*, así como operaciones sobre conjuntos. Se considera que estas operaciones son constantes. En este caso, la complejidad viene dada por:

$$T(n) = \sum_{i=1}^n 1 + \sum_{i=1}^{n-1} (n + \sum_{k=2}^n 1) = n^2 + \sum_{i=1}^{n-1} 2n - 1 = n^2 + (2n - 1)(n - 1) = O(n^2)$$

Por consiguiente la complejidad del algoritmo de Prim con la implementación de matrices es de orden cuadrático.

PROBLEMAS AVANZADOS

2.18. Escriba una función para calcular el máximo común divisor de dos números enteros positivo y calcule su complejidad.

Solución

El algoritmo clásico de Euclides para calcular el máximo común divisor es el siguiente:

```
int Euclides(int m, int n)
/*m,n son ambas positivas*/
{
    int r;
    while ( m > 0)
    {
        r = n % m;
        n = m;
        m = r;
    }
    return (n);
}
```

Se comprueba que la complejidad del algoritmo es $O(\log(m)) = O(\log(n))$.

Para ello se demuestra que $n \% m$ es siempre menor que $n/2$ (cociente de la división entera). En efecto

si $m > n/2$ entonces $n \% m = n - m < n - n/2 = n/2$.

si $m \leq n/2$ es evidente que $n \% m < n/2$.

Sea k el número de iteraciones. Sean n_i y m_i los valores de n y m al final de cada iteración.

Así $m_k = 0$ ya que al final del bucle m vale cero y $n_i = m_i, m_i = n_{i-1} \% m_{i-1}$

De esta forma $m_i = n_{i-1} \% m_{i-1} < n_{i-1}/2 = m_{i-2}/2$.

Si k impar es decir hay un número natural t tal que $k = 2t+1$. Se tiene:

$m_{k-1} < m_{k-3}/2 < m_{k-5}/2^2 < \dots < m_0/2^t$. De esta forma $k-1 \leq 2 \log_2(m_0)$. Es decir $k = O(\log(m)) \in O(\log(m))$

De forma análoga se procede cuando k es par.

2.19. Escriba una función que permita encontrar el k -ésimo elemento de un vector de n datos. El elemento buscado es el que ocuparía la posición k en orden ascendente si el vector estuviera ordenado. Calcule su complejidad.

Análisis

La función *K_esimo* está basada en la siguiente idea:

Se divide el vector en dos mitades, a la izquierda los pequeños, y a la derecha los grandes. Posteriormente se decide buscar en la izquierda, o bien en la derecha, o bien se devuelve el elemento buscado. El vector no se ordena, pero en cambio se puede encontrar el elemento que ocupa la posición si estuviera ordenado.

Solución

```

float K_esimo (float A[], int Izq, int Der, int k);
{
    int Centro, aux, x, i, j;
    Centro = ( Izq + Der ) / 2;
    x = A[Centro];
    i = Izq;
    j = Der;
    do
    {
        while ( A[i] < x)                /* aumentar i porque A[i] es pequeño */
            i ++;
        while ( A[j] > x)                /* disminuir j porque A[j] es grande */
            j --;
        if ( i <= j)                    /* no se han cruzado, hay que cambiar de orden */
        {
            Aux = A[i];
            A[i] = A[j];
            A[j] = Aux;
            i++;
            j--;
        }
    }
    while ( i <= j);

    /*se ha terminado la partición, los pequeños a la izquierda y los grandes a la derecha */

    if ( j < k)
        Izq=i;                        /* hay que buscar por la derecha */
    if (k>i)
        Der=j;                        /* hay que buscar por la izquierda */
    if (Izq < Der)                    /* no se ha encontrado la posición */
        return (K_esimo(A, Izq, j, k));
    else
        return (A[k]);                /*se ha encontrado el elemento */
}

```

Cada llamada recursiva a la función `K_esimo` requiere una pasada a través del *array*, y esto tarda $O(n)$. Eso significa que el número total de operaciones para cada nivel de recursión son operaciones $O(n)$. De esta forma la recurrencia será en el mejor de los casos y caso esperado: $T(n) = n + T(n/2)$ si $n > 1$ y 1 en otro caso.

Aplicando expansión de recurrencias se tiene:

$$T(n) = n + T(n/2) = n + n/2 + T(n/4) = \dots = n + n/2 + n/4 + \dots + 4 + 2 + 1 =$$

$$n \left(\sum_{k=0}^{k=\log(n)} \frac{1}{2^k} \right) = n \left(\frac{1 - \frac{1}{2n}}{\frac{1}{2}} \right) = n \left(2 - \frac{1}{n} \right) = 2n - 1 \in O(n)$$

En el peor de los casos la recurrencia será: $T(n) = n + T(n-1)$ si $n > 1$ y 1 en otro caso.

Aplicando expansión de recurrencias se tiene:

$$T(n) = n + T(n-1) = n + n-1 + T(n-2) = \dots = n + n-1 + n-2 + \dots + 2 + 1 = \frac{n(n+1)}{2} \in O(n^2)$$

Por lo tanto el tiempo medio y mejor de ejecución es $O(n)$ y en el peor de los casos es $O(n^2)$

El tiempo esperado será $O(n)$ ya que lo normal es dividir A en dos partes aproximadamente iguales.

- 2.20.** *Escriba un algoritmo recursivo para resolver el problema de las Torres de Hanoi y mediante la técnica de inducción calcule su complejidad.*

Análisis

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 3 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. Cada disco era ligeramente inferior en diámetro al que estaba justo debajo de él. El problema original consiste en llevar los n discos de la varilla A a la varilla C usando las siguientes reglas:

- Sólo se puede llevar un disco cada vez.
- Un disco sólo puede colocarse encima de otro con diámetro ligeramente superior.
- Si se necesita puede usarse la varilla C.

Para resolver el problema basta con observar que si sólo hay un disco $n = 1$, entonces se lleva directamente de la varilla A a la C.

Si hay que llevar $n > 1$ discos de la varilla A a la varilla C, entonces

- Se llevan $n - 1$ discos de la varilla A a la B.
- Se lleva un sólo disco de la varilla A a la C.
- Se llevan los $n - 1$ discos de la varilla B a la C.

Codificación

```
void hanoi(char A, char B, char C, int n)
{
    if (n == 1)
        printf("Mover disco %d desde varilla %c a varilla %c \n ", n, A, C);
    else
    {
        hanoi(A, C, B, n - 1);
        printf("Mover disco %d desde varilla %c a varilla %c \n ", n, A, C);
        hanoi(B, A, C, n - 1);
    }
}
```

La recurrencia para encontrar el tiempo que tarda en ejecutarse viene dada por:

$$T(1)=1$$

$$T(n) = 2T(n-1)+1 \text{ si } n>1.$$

Para realizar la inducción debe procederse en primer lugar con una inducción constructiva que construye la posible solución y una inducción demostrativa que demuestra la solución propuesta.

Inducción constructiva

Para calcular la complejidad por el método de la inducción se supone que la solución a buscar es de la forma $a \cdot 2^n$, ya que la solución debe ser de la forma 2^n por una constante por tener dos llamadas recursivas. Así se tiene por tanto que $T(1) = a \cdot 2^1 = 1$, lo que significa que $a = 1/2$. Por otro lado como $T(n) = 2T(n-1) + 1$ si $n > 1$, se obtiene que $1/2 \cdot 2^n = 1/2 \cdot 2^{n-1} + 1$. De esta forma $2^n = 2^{n-1} + 2$ que es falso en general (para $n=3$ por ejemplo 8 es distinto de $4+2$). De lo anterior se deduce que la hipótesis de partida es falsa, pero es debido a que aparece un *sumando extraño*. Para intentar paliar el problema, se fortalece la solución a buscar, suponiendo que la solución es de la forma $a \cdot 2^n + b$. Se tiene ahora que $T(1) = a \cdot 2^1 + b = 1$. Por otro lado, como $T(n) = 2T(n-1) + 1$ si $n > 1$, se obtiene $a \cdot 2^n + b = a \cdot (2 \cdot 2^{n-1} + b) + 1$, por lo que $b = 2b + 1$. Es decir, que $b = -1$. Sustituyendo el valor de b en $a \cdot 2^n + b = 1$ se tiene que $a = 1$. Por lo que, la solución a buscar tiene la forma $2^n - 1$.

Inducción demostrativa

Se supone ahora que $T(n) = 2^n - 1$ por hipótesis de inducción. Hay que demostrar que es cierta la igualdad para todo número natural n positivo. Se comprueba que es cierto para el caso $n = 1$. Para $n = 1$, $T(1) = 1$ por la igualdad. La fórmula expresa por su parte $T(n) = 2^n - 1$, que para el caso $n=1$ es $T(1) = 2 - 1 = 1$. Por lo que la fórmula es correcta. Se supone que es cierto para el caso $n - 1$ y se demuestra para el caso n . Para el caso $n - 1$ se tiene por hipótesis de inducción que $T(n-1) = 2^{n-1} - 1$. Hay que demostrar que $T(n) = 2^n - 1$. Pero $T(n) = 2T(n-1) + 1$ por la recurrencia, por lo que $T(n) = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1 = T(n)$. Como se quería demostrar.

PROBLEMAS PROPUESTOS

- 2.1. Escribir una biblioteca para tratar números enteros grandes (números que no pueden ser almacenados en memoria mediante los tipos predefinidos por el compilador), que contenga las operaciones, lectura, escritura, suma, resta, producto y cociente. Calcule la complejidad de cada uno de los módulos.
 - Una lista de 20 enteros.
 - Una lista de 20 registros, suponiendo que las claves están en memoria y los registros en disco
 - Una lista de 2000 registros en orden aleatorio.
 - Una lista de 20.000 registros, el 95% de los cuales ya están ordenados.
- 2.2. Calcule la complejidad de los módulos de programa definidos en los ejercicios resueltos 1.1 y 1.2 del capítulo 1.
- 2.3. Escriba un módulo de programa que calcule la traspuesta de una matriz sobre sí misma y calcule su complejidad.
- 2.4. ¿Qué método de ordenación elegiría para ordenar cada una de las siguientes situaciones?, ¿por qué?
 - Una lista enlazada de 20 registros.
 - Una lista enlazada de 20.000 registros en orden aleatorio.
- 2.5. Suponga que tiene un total de n equipos (puede suponer que n es potencia de 2) que deben enfrentarse entre sí en una competición liguera. Escriba un programa que elabore el calendario deportivo, y calcule la complejidad de cada uno de los módulos de programas que defina.
- 2.6. Escriba módulos de programa que usando el problema 14.1 decida:
 - Si un número natural grande es primo. ¿Cuál es la complejidad del algoritmo?
 - Si un número natural grande es perfecto. Un número es perfecto si la suma de sus divisores incluyendo el uno y excluyéndose él mismo coincide consigo mismo (el 6 es un número perfecto). Calcule su complejidad.
 - Si un número es capicua (323454323 es capicúa). Calcule su complejidad.
- 2.7. El problema matemático de calcular el determinante de una matriz cuadrada $n \times n$ se puede resolver mediante un algoritmo recursivo, o bien por el método de Gauss.

- Escriba un algoritmo recursivo que calcule el determinante de una matriz cuadrada de orden n y calcule su complejidad.
- Aplicar el método de Gauss de resolución de sistemas de ecuaciones lineales para calcular el determinante de una matriz cuadrada de orden n y determine la complejidad.

2.8. Responda a los siguientes supuestos:

- Escriba un algoritmo que encuentre el elemento mayor de un vector de tamaño n usando un máximo de n comparaciones.
- Escriba un algoritmo que encuentre el elemento menor de un vector de tamaño n usando un máximo de n comparaciones.
- Escriba un algoritmo que encuentre el elemento mayor y el elemento menor de un vector de tamaño n exactamente en tiempo $3/2 n + c$, siendo c una constante. Idea avance con un bucle de dos en dos y realice exactamente tres comparaciones.

2.9. Resuelva los siguientes supuestos:

- Escriba un algoritmo recursivo que calcule la inversa de una matriz cuadrada de orden n y calcule su complejidad.
- Aplicar el método de Gauss-Jordan de resolución de sistemas de ecuaciones lineales para calcular la matriz inversa de una matriz cuadrada de orden n , y calcule su complejidad.

2.10. Escriba un algoritmo que realice un giro de 90° de una figura representada en una matriz cuadrada de orden n . Analice su complejidad.

2.11. Un vector de n números enteros se dice que es mayoritario si hay un número almacenado en alguna posición tal que se repite al menos $n/2$ veces. Escriba un algoritmo para resolver el problema de determinar si un vector es mayoritario y calcule su complejidad.

2.12. Escriba un algoritmo para decidir si existe un índice i de un vector que almacena n datos tal que $a[i] = i$. El vector a está ordenado crecientemente y contiene números enteros.

Arrays o arreglos¹ (listas y tablas), estructuras y uniones en C

En los lenguajes de programación se distinguen entre *tipos de datos básicos* o *simples* (carácter, entero y coma flotante) y *tipos de datos estructurados*. Los *arrays* son tipos de datos estructurados, cuyas componentes son todas del mismo tipo, por lo que a los *arrays* se les considera estructuras de datos homogéneas. La capacidad para crear nuevos tipos es una característica importante y potente de C y libera al programador de restringirse al uso de los tipos ofrecidos por el lenguaje. Una *estructura* contiene múltiples variables, que pueden ser de tipos diferentes lo que permite la creación de programas potentes, tales como bases de datos u otras aplicaciones que requieran grandes cantidades de datos. La *unión*, es otro tipo de dato no tan importante como las estructuras *array* pero necesario en algunos casos. Un *tipo de dato enumerado* es una colección de elementos con nombre que tienen valores enteros equivalentes. Este capítulo examina el tipo *array* o *arreglo* (lista o tabla), estructuras, uniones, enumeraciones y tipos definidos por el usuario que permite a un programador crear nuevos tipos de datos.

3.1. Array unidimensional

Un *array* (lista o tabla) es una secuencia de datos del mismo tipo y cuyas componentes se numeran consecutivamente desde las posiciones 0, 1, 2, 3..., hasta una constante entera positiva máxima. 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*. La sintaxis de declaración de un *array* de una dimensión constante determinada es:

```
tiponombreArray[numeroDeElementos];
```

El acceso a los elementos de un *array* se realiza mediante:

```
nombreArray[expresiónentera];
```

Donde *expresiónentera* es una expresión cuyo valor debe estar entre 0 y *numeroDeElementos-1*, ya que C no verifica que las expresiones de los índices de los *array* estén dentro del rango declarado. El almacenamiento de los datos de un *array* se realiza en bloques de memoria contigua a partir de la dirección base inicial 0. Cualquier *array* que sea declarado en C debe ser inicializado mediante sentencias explícitas de asignación o bien en la propia declaración.

¹ En España, el término inglés “array” no se suele traducir, mientras que en latinoamérica se traduce normalmente por **arreglo**.

EJEMPLO 3.1. Las siguientes sentencias declaran e inicializan arrays.

```
float num[5] = {11,21,31,41,51,60};
int n[] = {3, 4, 5} /*Declara un array de 3 elementos*/
char Nombre [ ] = {'L','u','c','a','s'}; /*Declara un array de 5 caracteres*/
```

Una cadena de texto (*string*) es un conjunto de caracteres, tales como “Esto es un texto”. C soporta cadenas de texto utilizando un *array* de caracteres que contenga una secuencia de caracteres. La *diferencia* entre un *array* de caracteres y una cadena de caracteres o, simplemente, una cadena es que las cadenas contienen un carácter nulo (‘\0’) al final de los caracteres. Un *array* de caracteres es una secuencia de caracteres individuales que puede tener, o no, el carácter nulo que es el fin de cadena.

EJEMPLO 3.2. La declaración e inicialización siguiente declara un array de caracteres que es una cadena. Incluye como último carácter el nulo ‘\0’ cuyo código ascii es el cero.

```
char cadena[] = “Esto es un texto”; “se añade ‘\o’ al final”
```

EJEMPLO 3.3. Aunque es posible trabajar con cadenas carácter a carácter en general su manipulación se hace utilizando funciones especiales. El siguiente ejemplo muestra las funciones de cadena fundamentales.

```
#include <stdio.h>
#include <string.h>

void compararCadenas (char a[], char b[])
{
    int i = strcmp(a,b);
    if (!i)
        printf(“iguales\n”);
    else
        if (i < 0)
            printf(“%s < %s\n”,a,b);
        else
            printf(“%s > %s\n”,a,b);
}

void concatenarCadenas (char a[], char b[])
{
    if (strlen(a)+strlen(b) < 80)
    {
        strcat (a,b);
        printf (“Concatenadas: %s\n”,a);
    }
}

void copiarCadenas (char a[], char b[])
{
    strcpy(a,b);
    printf (“1ª %s y 2ª %s\n”,a,b);
}

void calcularLongitud (char a[])
{
    printf(“longitud %s = %d\n”,a,strlen(a));
}

int main()
{
    char a[80], b[80];
    char numPtr[5] = “1234”;
```

```
//Conversión

printf("La cadena %s ha sido convertida en numero %d", numPtr, atoi(numPtr));
//Lectura

printf ("Introduzca una cadena\n");
scanf ("%s",a);
printf ("Introduzca otra cadena\n");
scanf ("%s",b);
//Funciones de uso frecuente
calcularLongitud(a);
calcularLongitud(b);
compararCadenas(a,b);
concatenarCadenas(a,b);
copiarCadenas(b,a);
return 0;
}
```

3.2. Array multidimensionales

Los *arrays multidimensionales* son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los *arrays* más usuales son 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. La sintaxis para la declaración de un *array* de tres dimensiones es:

<tipo de datoElemento> <nombre array> [<N1>] [<N2>][<N3>];

EJEMPLO 3.4. Las siguientes sentencias declaran arrays multidimensionales.

```
float mat[25][80]; /* matriz bidimensional real*/
int p[6][8][10]; /* array de tres dimensiones de tipo entero*/
int e[4][30][2][5]; /* array de cuatro dimensiones de tipo entero*/
```

Los *arrays* multidimensionales se pueden inicializar, al igual que los de una dimensión, cuando se declaran, teniendo en cuenta que se almacenan por filas. La inicialización consta de una lista de constantes separadas por comas y encerradas entre llaves.

EJEMPLO 3.5. Las siguientes sentencias declaran e inicializan un array bidimensional:

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

en ambas declaraciones los elementos se inicializan a:

t[0][0]=51, t[0][1]= 52, t[0][2]=53, t[1][0]= 54, t[1][1]=55, t[1][2]=56.

El acceso a los elementos de *arrays* bidimensionales o multidimensionales se realiza de forma análoga a la de los *arrays* unidimensionales. La diferencia reside en que en los elementos bidimensionales por ejemplo deben especificarse los índices de la fila y la columna. El formato general para asignación o extracción de elementos es:

Inserción de elementos

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

Extracción de elementos

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

3.3. Array como parámetros a funciones

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 el *array* en la función llamada. C trata automáticamente la llamada a la función como si hubiera situado el operador de dirección delante del nombre del *array*. La especificación de que un parámetro formal es un *array* se efectúa mediante los corchetes. En *arrays* unidimensionales los corchetes pueden incluir el tamaño o bien dejarse vacíos y después añadir un segundo parámetro que indique el tamaño del *array*. Para el caso de los *array* bidimensionales es conveniente indicar el número de columnas en la declaración.

EJEMPLO 3.6. *La declaración de que un parámetro es un array puede realizarse de la siguiente manera:*

```
#define M 100
float datos[M];
float Suma(float datos[M]);
```

o dejar los corchetes en blanco y añadir un segundo parámetro que indica el tamaño del *array*:

```
float Suma(float datos[], int n);
```

a la función *Suma* 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*:

```
printf(("Suma %2f \n", Suma(datos, MAX));
```

La función *Suma* se puede escribir mediante un bucle *while* que suma los elementos del *array* y una sentencia *return* devuelve el resultado de nuevo al llamador:

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

Para el caso de los *array* bidimensionales es conveniente indicar el número de columnas en la declaración.

```
float Calcula (float datos1[N][M]);
float Calcula1 (float datos1[ ][M], int n);
```

la llamada podría ser:

```
printf(("calcula %2f \n", Calcula(datos1));
printf(("calcula1 %2f \n", Calcula1(datos1, MAX));
```

3.4. Estructuras

Una **estructura** es un tipo de datos definido por el usuario como colección de uno o más tipos de elementos denominados miembros posiblemente de diferente tipo. El formato de declaración es:

```

struct <nombre de la estructura>
{
    <tipo de dato miembro1> <lista de nombres de miembros 1>
    <tipo de dato miembro2> <listas de nombres de miembros 2>
    ...
    <tipo de dato miembro n> <lista de nombres miembros n>
} nombre1, nombre2... ;

```

Las variables de estructuras se pueden definir de dos formas: listándolas inmediatamente después de la llave de cierre de la declaración de la estructura (`nombre1, nombre2...`), o listando el tipo de la estructura creado seguido por las variables correspondientes en cualquier lugar del programa antes de utilizarlas (`struct <nombre de la estructura> nombre1, nombre2...`).

Una estructura puede inicializarse dentro de la propia definición especificando los valores iniciales, entre llaves, después de la definición de variables estructura. El formato general es:

```

struct <tipo>
{
    <tipo de dato miembro1> <lista de nombres de miembros 1>
    <tipo de dato miembro2> <listas de nombres de miembros 2>
    ...
    <tipo de dato miembro n> <lista de nombres miembros n>
} <nombre variable estructura>= { valor miembro1, valor miembro2,... valor miembron};

```

Una estructura es un tipo de dato definido por el usuario similar a cualquier tipo estandar de C como `int` o un `char`, por lo que pueden hacerse asignaciones completas de variables del mismo tipo de estructuras. El operador `sizeof` se aplica sobre un tipo de datos, o bien sobre una variable. Se puede aplicar para determinar el tamaño que ocupa en memoria una estructura.

El acceso a los miembros de una estructura se puede realizar mediante el operadores punto (`.`), o bien el operador puntero `->`. El operador punto (`.`) proporciona un acceso directo a los miembros de la estructura. La sintaxis para la asignación es:

```
<nombre variable estructura>.<nombre miembro> = dato;
```

El operador puntero, `->`, permite acceder a los datos de la estructura a partir de un puntero. La asignación de datos a estructuras, debe ser realizada después de asignar memoria mediante, por ejemplo, la función `malloc()`. La asignación mediante el operador puntero tiene el siguiente formato:

```
<puntero estructura> -> <nombre miembro> = dato;
```

Una estructura puede contener dentro de su declaración otras estructuras llamadas *estructuras anidadas*.

EJEMPLO 3.7. *El siguiente ejemplo muestra estructuras anidadas.*

```

struct Info_Direccion
{
    char Direccion[60];
    long int cod_postal;
};

struct Info_Empleado
{
    char Nombre_de_Empleado[30];
    struct Info_Direccion Direccion_de_Empleado;
    double SalarioNeto;
} Empleado1;

```

Se puede crear un *array* de estructuras tal como se crea un *array* de otros tipos como expresa el siguiente ejemplo:

```
struct Info_Empleado Empleados[100];
```

Se pueden pasar estructuras a funciones, bien por valor o bien por referencia, utilizando el operador &. Si desea pasar la estructura por referencia, se necesita situar un operador de referencia & antes del parámetro actual de llamada `Empleado1` en la llamada a una función, por ejemplo `EntradaEmpleado(&Empleado1)`. El parámetro formal correspondiente de la función `EntradaEmpleado()` debe de ser tipo puntero `struct Info_Empleado* pp`.

3.5. Uniones

Las uniones son similares a las estructuras en cuanto que agrupa a una serie de variables, pero la forma de almacenamiento es diferente y por consiguiente tiene efectos diferentes. Una estructura (`struct`) permite almacenar variables relacionadas juntas y almacenadas en posiciones contiguas en memoria. Las uniones, declaradas con la palabra reservada `union`, almacenan también miembros múltiples en un paquete; sin embargo, en lugar de situar sus miembros unos detrás de otros, en una unión, todos los miembros se solapan entre sí en la misma posición para ahorrar memoria. El tamaño ocupado por una unión es la anchura de la variable más grande. La sintaxis de declaración, su acceso, así como su tratamiento es similar a `struct`; se utiliza la palabra reservada `union` para su declaración. Por ejemplo:

```
union <nombre de la union>
{
    <tipo de dato miembro1> <lista de nombres de miembros 1>
    <tipo de dato miembro2> <listas de nombres de miembros 2>
    ...
    <tipo de dato miembron> <lista de nombres miembros n>
} nombre1, nombre2,... ;
```

3.6. Enumeraciones

Una enumeración (`enum`) es un tipo definido por el usuario con constantes de nombre de tipo entero. Para la declaración de un tipo `enum` se escribe una lista de identificadores que internamente se asocian con las constantes enteras 0, 1, 2 ... su sintaxis de declaración es:

```
enum
{
    enumerador1, enumerador2, ...enumeradorn
};
```

En la declaración del tipo `enum` pueden asociarse a los identificadores valores constantes en vez de la asociación que por defecto se hace (0, 1, 2 ...). Para ello se utiliza este formato:

```
enum nombre
{
    enumerador1 = expresión_constante1,
    enumerador2 = expresión_constante2,
    ...
    enumeradorn = expresión_constanten
};
```

3.7 Typedef

La palabra reservada `typedef` permite a los programadores crear sinónimos de tipos de datos definidos previamente. La sintaxis de declaración es: `typedef tipoDeDato nuevoTipo;`. A partir de la declaración hecha con `typedef` se puede hacer uso

del nuevo sinónimo del tipo de dato para definir variables, en general, donde se utiliza ese tipo de datos. Normalmente los programadores utilizan sinónimos de tipos de datos para simplificar los tipos y hacerlos más *amigables*.

EJEMPLO 3.8. *Se declara el tipo enumerado `diaLaborable` y el tipo `boolean`.*

```
typedef enum {lunes=0, martes=1, miercoles=2, jueves=3, viernes=4} diaLaborable;
typedef enum {FALSE=0, TRUE=1} boolean;
```

PROBLEMAS RESUELTOS

- 3.1.** *Escriba un programa que genere aleatoriamente el número de elementos a almacenar en un vector de enteros, lo rellene aleatoriamente y lo presente de tal forma que se presenten 10 números por línea.*

Análisis

Se declara una constante `Max` que dimensiona el *array*. Posteriormente con la función `Rellena` se genera aleatoriamente el número de elementos `n` que tendrá el vector, y posteriormente se rellena el vector de datos aleatoriamente. La función `Escribe` presenta en pantalla el vector completo, dando un salto de línea cada vez que escribe 10 números.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#define Max 30

void Escribe( int A[], int n);
void Rellena(int A[], int *n);

void Escribe( int A[], int n)
{
    int i;
    for(i = 0; i < n; i++)
        if((i+1)%10 == 0)
            printf("%7d\n", A[i]);
        else
            printf("%7d", A[i]);
}

void Rellena(int A[], int *n)
{
    int i;
    randomize();
    *n=random(Max);
    for(i=0; i<(*n); i++)
        A[i]= random(Max);
}

void main (void)
{
    int a[Max];
    int n;
```



```

    Rellena(a,&n);
    printf(" datos almacenados aleatoriamente en el vector\n");
    Escribe(a,n);
}

```

Los resultados de una posible ejecución del programa son los siguientes:

datos almacenados aleatoriamente en el vector									
25	10	17	29	1	22	21	19	1	20
25	18	17	20	1	28	12	4	10	10
16	18	29	21	1	13	17			

- 3.2** *Escribir una función que decida si un vector es mayoritario. Un vector es mayoritario si existe un elemento almacenado en el vector tal que se repite más de la mitad de las veces.*

Análisis

Un bucle controlado por la variable *i* recorre todos los elementos del vector que recibe como parámetro la función *Mayoritario*, y se sale de él cuando ha recorrido todos los elementos del vector o cuando ya conoce que el vector es *Mayoritario*. (Obsérvese que se puede mejorar este bucle terminando cuando *i* sobrepasa la posición central del vector $((n+2)/2)$ ya que en este caso no puede haber ningún elemento a su derecha que haga que el vector sea mayoritario). Una vez elegido un elemento *A[i]*, otro bucle controlado por la variable *j* cuenta el número de veces que el elemento *a[i]* está a la derecha de la posición *i* (no necesita contar los elementos de la izquierda, porque si al contarlos el vector fuera mayoritario, ya se habría salido del bucle controlado por la variable *i*). Una vez terminado el bucle, basta con decidir si ha contado más números que antes en cuyo caso se actualiza la variable lógica *Yaes* con la decisión correspondiente y por supuesto la variable *ContM* que contiene el número máximo de repeticiones hasta el momento.

Codificación

```

int Mayoritario(int A[], int n)
{
    int i, j, Cont, ContM, Yaes;
    Yaes = 0; ContM = 0;
    for (i = 0 ; ( i < n ) && ( ! Yaes); i++)
    {
        Cont=0;
        for( j= i; j < n ; j++)
            if(A[i] == A[j])
                Cont++;
        if (ContM < Cont)
        {
            ContM = Cont;
            Yaes = ContM > (n+1)/2;
        }
    }
    return (Yaes);
}

```

- 3.3.** *Escribir un programa que lea en un array 5 números enteros, los presente en pantalla y esciba la suma de los elementos que sean negativos.*

Análisis

Se declara una constante `MAX` que dimensiona el *array*. Posteriormente en un bucle `for`, se leen los datos del *array*, para más tarde en otro bucle presentarlos en pantalla. Por último, otro bucle se encarga de acumular los valores negativos que han sido leídos.

Codificación

```
#include <stdio.h>
#define MAX 5
int main()
{
    int A[MAX], i, sumanegativos = 0;
    for (i = 0; i < MAX; i++)
    {
        printf(" introduzca el número: %d ", i+1); scanf("%d",&A[i]);
    }
    printf("\n Lista de números: ");
    for (i = 0; i < MAX; i++)
    {
        printf("%5d", A[i]);
    }
    for (i = 0; i < MAX; i++)
    {
        if (A[i]<0)
            sumanegativos += A[i];
    }
    printf("\nLa suma de los números negativos es %d",sumanegativos);
    return 0;
}
```

- 3.4.** *Escribir un programa C que lea la dimensión n de una matriz cuadrada, genere aleatoriamente los valores enteros de la matriz y decida si la matriz es simétrica dos, presentando los resultados en pantalla.*

Análisis

Se declara una constante `N` que dimensiona el *array*. Posteriormente se lee la dimensión entera n . Se genera aleatoriamente la matriz mediante la función `generaAleatoriamenteMat`. Una matriz es *simétrica dos*, si:

$$a[i][j] = a[j][i] + 2$$

si $i < j$. La función `EsSimetricaDos`, decide si la matriz cumple la condición. La función `EscribeMatriz` presenta la matriz en pantalla.

Codificación

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define N 8

void generaAleatoriamenteMat(int a[][N], int n);
int EsSimetricaDos(int a[][N], int n);
void EscribeMatriz(int a[][N], int n);
```

```

int main(void)
{
    int a[N][N], n, EsSimDos;
    randomize();
    do
    {
        printf("\nTamaño de cada dimensión de la matriz, máximo %d: ", N);
        scanf("%d", &n);
    } while (n < 2 || n > N);
    generaAleatoriamenteMat(a, n);
    EsSimDos = EsSimetricaDos(a, n);
    if (EsSimDos)
        puts("\n\ matriz simétrica dos.\n");
    else
        puts("\n\ No matriz simétrica dos.\n");
    EscribeMatriz(a, n);
    return 0;
}

void generaAleatoriamenteMat(int a[][N], int n)
{
    // genera aleatoriamente la matriz cuadrada de números aleatorios
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            a[i][j] = random(N);
}

int EsSimetricaDos(int a[][N], int n)
{
    // decide si una matriz cuadrada es simétrica.
    int i, j, esSimetrica;
    for (esSimetrica = 1, i = 0; i < n-1 && esSimetrica; i++)
    {
        for (j = i+1; j < n && esSimetrica; j++)
            if (a[i][j] != a[j][i])
                esSimetrica = 0;
    }
    return esSimetrica;
}

void EscribeMatriz(int a[][N], int n)
{
    // escribe la matriz cuadrada de números enteros.
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
            printf("%5d", a[i][j]);
        printf("\n");
    }
}

```

- 3.5.** *Escribir un programa C que reciba como parámetro una matriz cuadrada y retorne la suma de cada uno de los elementos de cada fila y cada columna.*

Análisis

Se supone que N es una constante que cumple la condición $0 < n \leq N$. La función `CalculaSumaFilasColumnas` recibe como parámetro la matriz cuadrada a y devuelve en los *arrays* unidimensionales `Fila` y `Columna` la suma de cada una de las filas y columnas respectivamente de la matriz.

Codificación

```
void CalculaSumaFilasColumnas( int a[][N], int n, int Fila[],int Columna[])
{
    int i,j;
    for( i= 0; i < n; i++)
    {
        // cálculo de la suma de cada una de las filas
        Fila[i]=0;
        for (j = 0; j < n;j ++)
            Fila[i] += a[i][j];
    }
    for( j = 0;j < n; j++)
    {
        // cálculo de la suma de cada una de las columnas
        Columna[j]=0;
        for (i = 0; i < n; i++)
            Columna[j] += a[i][j];
    }
}
```

3.6. Escribir una función que genere el triángulo de Tartaglia y otra que lo escriba.

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
```

Análisis

En el triángulo de Tartaglia cada número es la suma de los dos números situados encima de él. La función `TrianguloTartaglia` calcula el triángulo de Tartaglia de orden n y lo almacena en un *array* bidimensional del que se supone que previamente se ha declarado la constante N . La función `escribeTrianguloTartaglia` recibe como parámetro el *array* bidimensional y lo escribe.

Codificación

```
void TrianguloTartaglia( int a[][N], int n)
{
    int i,j;
    for (i = 0; i < n; i++)
        for (j = 0; j <= i; j++)
            if(j == 0 || j == i)
                a[i][j] = 1;
            else
```

```

        a[i][j] = a[i-1][j-1]+a[i-1][j];
    }

void escribeTrianguloTartaglia(int a[][N], int n)
{
    int i,j;
    printf(" Triangulo de Tartaglia de orden %d\n", n );
    for ( i = 0; i < n; i++)
    {
        for ( j = 0; j <= i; j++)
            printf("%4d", a[i][j]);
        printf(" \n");
    }
}

```

- 3.7.** Realizar un programa C que permita leer y escribir la información de un empleado que contenga su nombre dirección, código postal, salario neto y fecha de nacimiento.

Análisis

Se declara un tipo estructurado `fecha` que contiene los campos `día`, `mes` y `año`. El tipo estructurado `Info_Dirección` (contiene la Dirección y el código postal `Cod_postal`) y el tipo estructurado `Info_Empleado` con toda la información solicitada. Por último, se definen las funciones `entrada` y `salida` encargadas de realizar la entrada y salida de datos.

Codificación

```

#include <stdio.h>
struct fecha
{
    unsigned int dia, mes, anyo;
};
struct Info_Direccion
{
    char Direccion[60];
    long int cod_postal;
};
struct Info_Empleado
{
    char Nombre_de_Empleado[30];
    struct Info_Direccion Direccin_de_Emleado;
    double SalarioNeto;
    struct fecha fec;
};

void entrada(struct Info_Empleado *p);
void muestra(struct Info_Empleado p);

void main()
{
    struct Info_Empleado p;
    entrada(&p);      /* En la llamada se transmite la dirección de la estructura */
    muestra(p);        /* salida de los datos almacenados */
}

```

```

void entrada(struct Info_Empleado *p)
{
    printf("\nIntroduzca su nombre: "); gets(p->Nombre_de_Empleado);
    printf("\n direccion: "); gets(p->Direccin_de_Emleado.Direccion);
    printf("\n codigo postal: "); scanf("%d",&p->Direccin_de_Emleado.cod_postal);
    printf("Introduzca su salario: "); scanf("%d",&p->SalarioNeto);
    printf("Introduzca su fecha de nacimiento: ");
    scanf("%d %d %d", &p->fec.dia,&p->fec.mes, &p->fec.anyo);
}

void muestra(struct Info_Empleado p)
{
    puts("\n\n\tDatos de un empleado");
    puts("\n\n\t— — — —");
    printf("Nombre: %s \n", p.Direccin_de_Emleado.Direccion);
    printf("%d\n",p.Direccin_de_Emleado.cod_postal);
    printf("%d\n",p.SalarioNeto);
    printf("fecha de nacimiento: %d-%d-%d\n",p.fec.dia,p.fec.mes,p.fec.anyo);
}

```

PROBLEMAS AVANZADOS

3.8. Escribir las primitivas de gestión de una pila implementada con un array.

Análisis

Se define en primer lugar una constante `MaxPila` que tendrá el valor máximo de elementos que podrá contener la pila. Se define la pila (*último en entrar primero en salir*; véase capítulo 9) como una estructura cuyos campos (miembros) serán la variable entera `cima` que apuntará siempre al último elemento añadido a la pila y un *array* `A` cuyos índices variarán entre 0 y `MaxPila-1`. Posteriormente se implementa las primitivas de gestión de la pila que: `VaciaP`, `EsvaciaP`, `AnadeP`, `BorraP`, `PrimeroP`. Además se implementan las funciones `Estallenap` definidas de la siguiente forma:

- `VaciaP`. Crea la pila vacía poniendo la `cima` a `-1`.
- `EsvaciaP`. Indica cuándo estará la pila vacía. Ocurre cuando su `cima` vale `-1`.
- `EstallenaP`. Si bien no es una primitiva básica de gestión de una pila al realizar la implementación con un *array*, conviene tenerla para prevenir posibles errores. La pila está llena cuando la `cima` vale `MaxPila-1`.
- `AnadeP`. Añade un elemento a la pila. Para hacerlo comprobar que la pila no esté llena, y en caso afirmativo, incrementa la `cima` en una unidad, para posteriormente poner en el *array* `A` en la posición `cima` el elemento.
- `PrimeroP`. Comprueba que la pila no esté vacía, y en caso de que así sea retorna el elemento del *array* `A` almacenado en la posición apuntada por la `cima`.
- `BorrarP`. Elimina el último elemento que entró en la pila. En primer lugar se comprueba que la pila no esté vacía en cuyo caso, disminuye la `cima` en una unidad.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#define MaxPila 100

```

```
typedef float TipoElemento;
typedef struct
{
    TipoElemento A[MaxPila];
    int cima;
}Pila;
int EsVaciaP(Pila P) ;
void VaciaP(Pila* P);
void AnadeP(Pila* P,TipoElemento e);
TipoElemento PrimeroP(Pila P);
void BorrarP(Pila* P);

int EsVaciaP(Pila P)
{
    return P.cima == -1;
}

void VaciaP(Pila* P)
{
    P -> cima = -1;
}

int EstallenaP(Pila P)
{
    return P.cima == MaxPila-1;
}
void AnadeP(Pila* P,TipoElemento e)
{
    if (EstallenaP(*P))
    {
        puts("Desbordamiento pila");
        exit (1);
    }
    P->cima++;
    P->A[P->cima] = e;
}

TipoElemento PrimeroP(Pila P)
{
    TipoElemento Aux;
    if (EsVaciaP(P))
    {
        puts("Se intenta sacar un elemento en pila vacía");
        exit (1);
    }
    Aux = P.A[P.cima];
    return Aux;
}

void BorrarP(Pila* P)
{
    if (EsVaciaP(*P))
    {
```

```

    puts("Se intenta sacar un elemento en pila vacía");
    exit (1);
}
P->cima= -;
}

```

3.9. Escribir las primitivas de gestión de una cola implementada con un array no circular.

Análisis

Se define en primer lugar una constante `MAXIMO` que tendrá el valor máximo de elementos que podrá contener la cola. Se define la cola (Primero en entrar primero en salir; véase capítulo 11) como una estructura cuyos campos (miembros) serán las variables enteras `frente` que apuntará al primer elemento de la cola y `final` que apuntará al último elemento de la cola. Los datos de la cola se almacenarán en el miembro de la estructura `A` definido como un *array* de elementos. Posteriormente, se implementan las primitivas que son: `VaciaC`, `EsvaciaC`, `AnadeC`, `BorraC`, `PrimeroC`. Además se implementa la función `EstallenaC`, de la siguiente forma:

- *VaciaC*. Crea la cola vacía poniendo `frente` a 0 y `final` a -1.
- *EsvaciaC*. La cola no tiene elementos. Ocurre cuando su `frente` coincide con `final` -1.
- *EstallenaC*. Si bien no es una primitiva básica de gestión de una cola al realizar la implementación con un *array*, conviene tenerla para prevenir posibles errores. Cada vez que se añade a la cola se llama a la función `EstallenaC` que se encarga además de decidir si la cola está llena, de reestructurar la cola si es necesario. Esto es: si hay *hueco* en el *array* que almacena los elementos (el *hueco* siempre está desde las posiciones 0 a `frente`), entonces los elementos del *array* retroceden las posiciones vacías existentes en la cola para permitir que puedan añadirse más datos a la cola. Esto es lo que en el capítulo 11 se denomina cola con reestructuración o retroceso.
- *AnadeC*. Añade un elemento a la cola. Para hacerlo comprobar que la cola no esté llena, y en caso afirmativo, incrementa el `final` en una unidad, para posteriormente poner en el *array* `A` en la posición el elemento.
- *PrimeroP*. Comprueba que la cola no esté vacía, y en caso de que así sea retorna el elemento del *array* `A` almacenado en la posición apuntada por el `frente`.
- *BorrarP*. Elimina el primer elemento que entró en la cola. Se comprueba en primer lugar que la cola no esté vacía en cuyo caso, aumenta el `frente` en una unidad.

Codificación

```

#include <stdlib.h>
#include <stdio.h>
#define MAXIMO 100
typedef char TipoElemento;
typedef struct
{
    TipoElemento A[MAXIMO];
    int frente, final;
} Cola;

void VaciaC(Cola* C);
void AnadeC(Cola* C, TipoElemento elemento);
void BorraC(Cola* C);
TipoElemento PrimeroC(Cola C);
int EsvaciaC(Cola C);
int EstallenaC(Cola *C);

void VaciaC(Cola* C)
{

```



```
C -> frente = 0;
C -> final = -1;
}

void AnadeC(Cola* C, TipoElemento e)
{
    if (EstallenaC(C))
    {
        puts("Cola completa");
        exit (1);
    }
    C -> final ++;
    C -> A[C->final] = e;
}

void BorraC(Cola* C)
{
    if (EsvaciaC(*C))
    {
        puts("Se intenta sacar un elemento en C vacía");
        exit (1);
    }
    C -> frente++;
}

int EsvaciaC(Cola C)
{
    return C.final < C.frente;
}

int EstallenaC(Cola *C)
{
    int i;
    if (C->final == MÁXIMO - 1 && C->frente != 0)
    {
        for (i = C->frente; C->final; i++)
            C->A[i - C->frente] = C->A[i];
        C->final -= C->frente;
        C->frente = 0;
    }
    return C->final == MAXIMO - 1;
}

TipoElemento PrimeroC(Cola C)
{
    if (EsvaciaC(C))
    {
        puts(" Error de ejecución, C vacía");
        exit (1);
    }
    return C.A[C.frente];
}
```

- 3.10.** *Escribir un programa que resuelva un sistema de n ecuaciones con n incógnitas por el método de Gauss (Triangularización de matrices). Calcular su complejidad.*

Análisis

El método de resolución de ecuaciones de Gauss consiste en transformar el sistema de ecuaciones en otro equivalente de tal manera que sea triangular inferior. Es decir, debajo de la diagonal principal siempre hay ceros. Para ello se emplea el método de reducción hacia delante. El algoritmo empleado es el siguiente:

```

k ← 0  error ← falso
mientras (k < n-1) y no error  hacer
    <elegir pivote en columna k tal que sea el máximo en valor absoluto de todos los elementos de
    la columna k que ocupen filas mayores o iguales que k>
    Si la fila en la que se encuentra el pivote no es la k entonces
        <intercambiar la fila k con la fila en la que se encuentre>
    fin si
    Si el pivote tiene valor absoluto menor que un cierto épsilon entonces
        <el sistema no es compatible determinado. Poner error a verdadero>
    sino
        <dividir la fila k por el elemento que ocupa la posición ( k,k,)>
        desde cada fila i ← k+1 hasta n-1  hacer
            <a la fila i restarle la fila k multiplicada por a[i,k] para hacer ceros en la
            columna k debajo de la fila k>
            incrementar el contador de filas k en una unidad.
        fin desde
    fin si
fin mientras

```

Posteriormente se resuelve el sistema por el método de reducción hacia atrás, resolviendo un sistema triangular superior. Es decir, un sistema de n ecuaciones con n incógnitas sabiendo que es compatible determinado y que debajo de la diagonal principal hay ceros, y en la diagonal principal hay unos.

El programa que se presenta tiene las siguientes funciones:

- Una función `leer` que se encarga de solicitar todos los datos del sistema, conociendo el valor de ecuaciones y de incógnitas n , almacenando los resultados en la matriz de coeficientes a y en el término independiente b .
- Una función `escribir` presenta los datos del sistema.
- Una función `eleccion_de_pivote` que recibe como parámetro la matriz a el término independiente b , el número de ecuaciones n y el valor de k que indica la columna que hay que hacer ceros por debajo de la diagonal principal (posición k, k) y usando la técnica de elección parcial de pivote, obtiene el valor máximo de la columna k que se encuentra debajo de la diagonal principal. Y por último intercambiando filas deja en la posición $a[k, k]$, el valor máximo.
- Una función `gauss_simple` realiza la triangularización del sistema. Mediante un bucle controlado por el valor de la variable k y por no haber obtenido error, llama a la función `eleccion_de_pivote`. Si el elemento que ocupa la posición $a[k, k]$ es cero, entonces el sistema no tiene solución, ya que hay toda una columna que a partir de la diagonal principal tiene sólo ceros. En otro caso, divide la fila k por $a[k, k]$, para dejar un 1 en la diagonal principal, para posteriormente hacer ceros debajo de la diagonal principal. La fila $a[i] = \text{fila}[i] - \text{fila}[k] * a[i][k]$.
- Una función `solucion` que una vez triangularizado el sistema, obtiene la solución, mediante la técnica de reducción hacia atrás, que mediante un bucle descendente, obtiene la solución i -ésima, usando la sustitución en la ecuación i -ésima de las soluciones ya obtenidas $i+1$ -ésima, $i+2$ -ésima, hasta la n -ésima.
- La función `main` se encarga de realizar las llamadas correspondientes.

Codificación

```
#include <stdio.h>
#include <math.h>
#define m 5
void leer(float a[m][m],float b[m],int n);
void gauss_simple(float a[m][m],float b[m],int *error, int n);
void eleccion_de_pivote(float a[m][m],float b[m], int k, int n);
void escribe(float a[m][m], float b[m],int n);
void solucion(float a[m][m],float b[m],float x[m],int n);

void main(void)
{
    int n,i, error;
    float a[m][m], b[m],x[m];
    do
    {
        printf("dame valor de n \n" );
        scanf("%d", &n);
    }
    while ((n <= 0) || (n > (m - 1)));
    leer(a,b,n);
    escribe(a,b,n);
    gauss_simple(a,b,&error,n);
    printf(" sistema reducido\n");
    escribe(a,b,n);
    if (!error)
    {
        solucion(a,b,x,n);
        printf(" solucion del sistema\n");
        for(i = 0;i < n;i++)
            printf("x[%ld]= %6.2f ",i+1, x[i]);
    }
    else
        printf("error no tiene solucion\n");
}

void leer(float a[m][m],float b[m],int n)
{
    /* se encarga de leer el sistema de n ecuaciones con n incognitas
    y almacenarlo en la matriz de coeficientes a y termino independiente b */
    int i,j;
    for (i = 0; i < n; i++)
    {
        for(j = 0;j < n; j++)
        {
            printf("valor de a[%ld,%ld]\n", i+1,j+1);
            scanf("%f",&a[i][j]);
        }
        printf("introduzca valor de b[%ld]\n", i+1);
        scanf("%f",&b[i]);
    }
}
```

```

void gauss_simple(float a[m][m], float b[m], int *error, int n)
{
    /*Realiza la triangularización del sistema de n ecuaciones con n
    incógnitas, realizando combinaciones lineales de filas, para hacer ceros debajo de la
    diagonal principal de la matriz de coeficientes */
    int i , j , k = 0;
    float valor, eps=0.001;
    *error = 0; //false
    do {
        eleccion_de_pivote(a, b, k, n);
        valor = a[k][k];
        if (fabs( valor) < eps)
        {
            printf("error división por cero %f %d\n", valor, k);
            *error = ! (*error);
        }
        else
        {
            for (j = k; j < n; j++)
                a[k][j] = a[k][j]/ valor;
            b[k] = b[k] / valor;
            for (i = k+1; i < n; i++)          /*(hacer 0 a[i,k] sumando a la fila i la fila k*/
            {
                valor = a[i][k];
                for(j = k; j < n; j++)
                    a[i][j] = a[i][j] - a[k][j] * valor;
                b[i] = b[i] - b[k] * valor;
            }
        }
        k++;
    } while (( k < n ) && (! (*error)));
}

void eleccion_de_pivote(float a[m][m], float b[m], int k, int n)
{
    /* Elige el máximo de la columna k, desde la posición(k+1,k) hasta la posición(n,k) de la matriz
    de coeficientes. Posteriormente intercambia filas si es necesario, para dejar el elemento
    pivote en la posición (k,k) de la matriz de coeficientes */
    int pivo, i, j;
    float aux, max;
    max = fabs(a[k][k]);
    pivo = k;
    for ( i = k + 1; i < n; i++)
        if (fabs(a[i][k]) > max)
        {
            max = fabs (a[i][k]);
            pivo = i;
        }
    if (pivo != k)
    {
        for (j= k; j<n; j++)
        {
            aux = a[k][j];

```

```

        a[k][j] = a[pivo][j];
        a[pivo][j] = aux;
    }
    aux = b[k];
    b[k] = b[pivo];
    b[pivo] = aux;
}
}

void escribe(float a[m][m], float b[m],int n)
{
    /* escribe el sistema*/
    int i, j;
    for( i = 0; i < n; i++)
    {
        for(j = 0; j < n; j++)
            printf("%8.2f", a[i][j]);
        printf("=%7.2f\n", b[i]);
    }
}

void solucion(float a[m][m],float b[m],float x[m],int n)
{
    /*Resuelve el sistema de n ecuaciones con n incógnitas, siempre que el sistema esté
    triangularizado previamente */
    int i, j;
    for (i = n - 1; i >= 0; i--)
    {
        x[i] = b[i];
        for (j = i+ 1; j < n; j++)
            x[i] = x[i] - x[j] * a[i][j] ;
    }
}

```

Resultados de ejecución del programa

```

dame valor de n
2
valor de a [1,1]
1
valor de a [1,2]
2
introduzca valor de b[1]
3
valor de a [2,1]
-1
valor de a [2,2]
2
introduzca valor de b[2]
-1
    1.00    2.00=    3.00
    -1.00   2.00=   -1.00
sistema reducido
    1.00    2.00=    3.00
    0.00    1.00=    0.50
solución del sistema
x[1]=    2.00  x[2]= 0.50

```

Estudio de la complejidad:

La función `leer` tiene la siguiente complejidad
ya que tiene dos bucles anidados

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n 1 = \sum_{i=1}^n n = n * n = n^2 \in O(n^2)$$

La función `Escribe` tiene la siguiente complejidad:
ya que tiene dos bucles anidados

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n 1 = \sum_{i=1}^n n = n * n = n^2 \in O(n^2)$$

La función `Elección_de_pivote` tiene la siguiente complejidad

$$T(n) = \sum_{i=k}^n \sum_{j=1}^n 1 = n - k \in O(n)$$

La función `Gauss_simple` tiene la siguiente complejidad

$$T(n) = \sum_{k=1}^n \sum_{i=k}^n \sum_{j=k}^n 1 = \sum_{k=1}^n \sum_{i=k}^n (n - k + 1) = \sum_{k=1}^n (n - k + 1) k^2 = \frac{n * (n + 1/2) * (n + 1)}{3} \in O(n^3)$$

La función `Solución` tiene la siguiente complejidad

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n 1 = \sum_{i=1}^n (n - i + 1) = \sum_{i=1}^n i = \frac{n * (n + 1)}{2} \in O(n^2)$$

Por lo tanto el algoritmo total es de complejidad $n^3 \in O(n^3)$

- 3.11.** Escribir una función *C* que reciba como parámetro un número en base 10 en una variable entera y una nueva base entre 2 y 16, y devuelva el número escrito en la nueva base como cadena de caracteres.

Análisis

Para realizar la transformación solicitada se usa una cadena de caracteres llamada `caracteresbase` que almacena los distintos caracteres válidos para cualquier base dentro del rango (por ejemplo el número 5 es el '5', el número 10 es el carácter 'A', el número 11 es el carácter 'B', etc). Se define un vector de enteros `auxiliar` que almacena en cada una de las posiciones (como resultado intermedio) el dígito entero al que corresponde cuando se pasa a la base que se recibe como parámetro, usando como esquema: *mientras la variable número sea distinta de cero hacer* que en la posición *i* del vector `auxiliar` se almacene el *resto de la división entera del número número* entre la base, para posteriormente almacenar en la variable `número` el cociente de la división entera del número entre la base. Por último, se pasa el número almacenado en `auxiliar` a cadena de caracteres usando el array `caracteresbases`.

Codificación

```
#include <stdio.h>
void cambio(long int  numero, int  base, char salida[]);
void main (void)
{
    char salida[10];
    cambio(1004, 14, salida);
    puts(salida);
}

// cambia el numero escrito en 10 a un numero escrito
// como cadena de caracteres en la nueva base
```

```

void cambio (long int  numero, int  base, char salida[])
{
    char caracteresbase[17]={'0','1','2','3','4','5','6','7','8','9',
                             'A','B','C','D','E','F','\0'};
    int  i=0,n,auxiliar[20];
    while (numero != 0)
    {
        auxiliar[i] = numero % base;
        numero/= base;
        i++;
    }
    i--;
    n=i;
    for(;i >= 0; i--)
        salida[n-i] = caracteresbase[auxiliar[i]];
    n++;
    salida[n] = '\0';
}

```

Ejecución

Si a la función se le llama desde el programa principal que aparece en la codificación el resultado:

51A número 1004 escrito en base 14 $1004 = 5 \cdot 14 \cdot 14 + 14 + 10 = 980 + 14 + 10$.

PROBLEMAS PROPUESTOS

- 3.1. Diseñar una función que acepte como parámetro un vector que puede contener elementos duplicados. La función debe sustituir cada valor repetido por -5 y devolver al punto donde fue llamado el vector modificado y el número de entradas modificadas.
- 3.2. 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).
- 3.3. Escribir un programa que convierta un número romano (en forma de cadena de caracteres) en número arábigo.
Reglas de conversión: M=1000, D=500, C=100, L=50, X=10, V=5, I=1.
- 3.4. Realizar un programa de facturación de clientes. Los clientes tienen un nombre, el número de unidades solicitadas, el precio de cada unidad y el estado en que se encuentra: moroso, atrasado, pagado.
- 3.5. Escribir un programa que permita hacer las operaciones de suma, resta, multiplicación y división de números complejos. El tipo complejo ha de definirse como una estructura.
- 3.6. Escribir un programa para operar con números racionales. Las operaciones a definir son la suma, resta, multiplicación y división; además de una función para simplificar cada número racional.
- 3.7. Desarrollar un programa C que use una estructura para guardar tres diferentes producciones que están planificadas en un mes de trabajo, cada una de ellas tiene los datos: número de identificación, horas planificadas, horas realizadas, nombre del cliente. El programa también debe utilizar una estructura de empleado (cinco empleados) que son los que realizan las diferentes producciones. Los campos de la estructura empleado: número de identificación, nombre, producción asignada, nombre y horas trabajadas.

3.8. Escribir un programa para calcular el número de días que existen entre dos fechas; declarar fecha como una estructura.

3.9. Diseñar un programa que determine la frecuencia de aparición de cada letra mayúscula en un texto escrito por el usuario (fin de lectura, el punto o el retorno de carro, ASCII 13).

3.10. 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, y columna son iguales.

Ejemplo

8	1	6
3	5	7
4	9	2

3.11. 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).

3.12. Escribir un programa en el que se genere aleatoriamente un vector de 20 números enteros. El vector 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 vector original y el vector con la distribución indicada

3.13. Se quiere informatizar los resultados obtenidos por los equipo de baloncesto y de fútbol de la localidad alcarreña Lupiana. La información de cada equipo es:

- Nombre del equipo.
- Número de victorias.
- Número de derrotas.

Para los equipos de baloncesto añadir la información:

- Número de perdidas de balón.
- Número de rebotes capturados.
- Nombre del mejor anotador de triples.
- Número de triples del mejor triplista.

Para los equipos de futbol añadir la información:

- Número de empates.
- Número de goles a favor.
- Número de goles en contra.
- Nombre del goleador del equipo
- Número de goles del goleador.

Realizar un programa para introducir la información para todos los equipos integrantes de ambas ligas

Recursividad

La **recursividad (recursión)** es aquella propiedad que posee una función por la cual dicha función puede llamarse así misma. La recursión es una herramienta poderosa e importante en resolución de problemas y en programación. Se puede utilizar la recursividad 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 llamadas suplementarias a las funciones; 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.

4.1. Algoritmos recursivos

Una función *recursiva* es aquella que se llama a sí misma o bien directamente o bien a través de otra función. Una función que tiene sentencias entre las que se encuentra al menos una que llama a la propia función se dice que es *recursiva*. La definición de factorial de un número natural n se define como $n! = n * (n - 1)!$ para todo número n mayor que 0, y $0! = 1$. Una programación recursiva de la función factorial puede consultarse en el Ejercicio Resuelto 4.1. En el diseño de un algoritmo recursivo hay que tener en cuenta que:

- El instrumento necesario para expresar los programas recursivamente es la función.
- Una función tiene **recursividad directa** cuando se llama a sí misma dentro de su propia definición.
- Una función tiene **recursividad indirecta** si llama a otra función que, a su vez, contiene una referencia directa o indirecta a la primera.
- El cuerpo de una función recursiva debe tener una o varias instrucciones selectivas donde establece para los casos generales las soluciones generales, y para los casos triviales las soluciones triviales.
- Cada llamada recursiva debe disminuir el tamaño del ejemplar de entrada de tal manera que se acerque a la(s) solución(es) trivial(es).
- Cada vez que se llama a una función recursiva todos los valores de los parámetros formales y variables locales son almacenados en una pila.
- Cuando termine de ejecutarse la función recursiva retorna al nivel inmediatamente anterior. En la pila se recuperan los valores tanto de los parámetros como de las de variables locales, y se continúa con la ejecución de la siguiente instrucción a la llamada recursiva.

EJEMPLO 4.1. La función *sumaimpares* emplea la recursividad para hallar la suma de los *n* primeros números enteros impares.

```
#include <stdio.h>
int main()
{
    int n;
    double sumaimpares(int);

    printf("Escriba un número entero positivo: \n");
    scanf("%d", &n);
    printf("Suma de los %d primeros impares: %lf\n", n, sumaimpares(n));
}

double sumaimpares(int n)
{
    if(n <= 1)
        return 1.0;
    return ((2*n - 1) + sumaimpares(n - 1));
}
```

EJEMPLO 4.2. El siguiente programa permite observar el funcionamiento de la recursividad y cómo los valores de los parámetros son recuperados de la pila a la salida de la recursividad. La función *contar* decreuenta el parámetro y lo muestra y si el valor es mayor que cero, se llama a sí misma donde decreuenta el parámetro una vez más, etc. Cuando el parámetro alcanza el valor de cero, la función ya no se llama a sí misma y se producen sucesivos retornos al punto siguiente a donde se efectuaron las llamadas, donde se recupera de la pila el valor del parámetro y se muestra.

```
# include <stdio.h>

void contar(int valor)
{
    valor - ;
    printf ( "%2d\n", valor ) ;
    if (valor > 0)
        contar(valor);
    printf ( "%2d\n", valor ) ;
}

int main( )
{
    int num ;
    num = 8 ;
    contar(num) ;
    return 0 ;
}
```

4.2. Casos en los que debe evitarse el uso de la recursividad

Las funciones recursivas que tienen explosión exponencial en cuanto a tiempo de ejecución pero que el número de subproblemas esencialmente distintos que deben resolverse es de tiempo polinomial deben evitarse. Un ejemplo clásico y sencillo viene dado por la función de Fibonacci que se define de la siguiente forma;

$Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n - 2)$ y $Fibonacci(0) = 0, Fibonacci(1) = 1$.

La definición dada es recursiva y su implementación puede consultarse en el Ejercicio Resuelto 4.1. La solución es mala, ya

que se repiten los cálculos de los valores inferiores de n lo que hace que su tiempo sea exponencial. Si se usa la técnica de programación dinámica, o bien la de parámetros acumuladores, se pueden almacenar los resultados intermedios y diseñarlo iterativamente mediante el uso de dos variables auxiliares lo que permite que el tiempo de ejecución disminuya, y se convierta en lineal. Siempre que la solución recursiva repita cálculos es conveniente no usarla. Una solución iterativa puede consultarse en el Ejercicio Resuelto 4.1. En general siempre que una función recursiva repita cálculos pueden almacenarse los resultados intermedios permitiendo bajar la complejidad algorítmica a tiempos asintóticamente inferiores. Es en estos casos cuando nunca conviene usar un diseño recursivo.

4.3. Recursividad directa e indirecta

La recursividad puede considerarse de cabeza, cola o intermedia según que las operaciones se hagan después, antes o antes y después de la llamada recursiva. Por otra parte, también existe la recursividad múltiple, con diversas llamadas recursivas y la anidada. La recursividad indirecta se produce cuando una función llama a otra, que eventualmente termina llamando de nuevo a la primera función. Un ejemplo de recursividad indirecta puede consultarse en el Ejercicio Resuelto 4.1.

EJEMPLO 4.3. *La función de Ackerman es una función recursiva anidada, donde la llamada recursiva utiliza como parámetro el resultado de una llamada recursiva. A continuación se implementa la función de Ackerman cuya definición se establece en forma recursiva de la siguiente manera*

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0; \\ A(m - 1, 1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m - 1, A(m, n - 1)), & \text{si } m > 0 \text{ y } n > 0; \end{cases}$$

```
# include <stdio.h>

int Ackerman( int m, int n )
{
    if(m==0 )
        return(n+1);
    else
        if(n==0)
            return(Ackerman(m-1,1));
        return(Ackerman(m-1, Ackerman(m,n-1)));
}

int main( )
{
    int i, j;
    for (i=0; i<=3; i++)
    {
        for (j=0; j<=4; j++)
            printf("%4d", Ackerman(i, j));
        printf("\n");
    }
    return 0 ;
}
```

El resultado de la ejecución muestra la rápida forma de crecer de los números en esta serie
Los sucesivos valores de la serie son:

1	2	3	4	5
2	3	4	5	6
3	5	7	9	11
5	13	29	61	125

$A(m,n)$	$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$
$m = 0$	1	2	3	4	5
$m = 1$	2	3	4	5	6
$m = 2$	3	5	7	9	11
$m = 3$	5	13	29	61	125
$m = 4$	13	65533	$2^{65536}-3$		

4.4. Métodos para la resolución de problemas que usan recursividad

Entre las distintas técnicas para diseñar algoritmos recursivos destacan la estrategia de *Divide y Vence* y la de *Retroceso* o *Backtracking*.

4.4.1. 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 puede construir fácilmente una solución del problema completo (*vencerás*). Un algoritmo “*divide y vence*” puede ser definido de manera recursiva, de tal modo que se llama a sí mismo sobre un conjunto menor de elementos. La condición para dejar de hacer llamadas es, normalmente, la obtención de un solo elemento (o muy pocos elementos) lo que se denomina el *caso base* o *caso trivial*. El algoritmo general del método es el siguiente:

```

algoritmo DivideyVence(E <límites del problema a tratar>; ...)
inicio
  si <Caso trivial> entonces
    <Solución trivial>
  si_no
    <dividir el problema en subproblemas>
    desde <cada subproblema i> hacer
      DivideyVence(<Subproblema i>, ...)
    fin desde
    <combinar las soluciones de los subproblemas>
  fin_si
fin_algoritmo

```

Un ejemplo de aplicación de esta técnica podría ser la implementación recursiva de la búsqueda binaria de un elemento en un vector ordenado (Ejercicio 5.2), las torres de Hanoi (Ejercicio 10.6 en el que se resuelve el problema recursivamente e iterativamente), ordenación por mezcla (Ejercicio 5.18).

4.4.2 BACKTRACKING (RETROCESO)

Es un método de resolución de problemas basado en el tanteo sistemático (prueba y error) y en el que la solución del problema se divide en pasos. La solución del paso i -ésimo se resuelve en función del paso i -ésimo + 1. Las técnicas de *Backtracking* se pueden emplear con la finalidad de encontrar una solución, todas las soluciones o la solución óptima al problema planteado. Un algoritmo que devuelve una solución es:

```

procedimiento Ensayar(E entero: i; E/S logico: exito; ...)
inicio
  <inicializa las posibilidades>
  mientras <queden posibilidades> y no exito hacer
    <toma la siguiente posibilidad>
    si <la posibilidad es aceptable> entonces
      <anota la posibilidad como parte de la solución>
      si <se encontró una solución general> entonces
        exito ← verdad

```

```

    si_no
        ensayar(i + 1, éxito, ...)
    fin_si
    si no exito entonces
        <desanota la posibilidad como parte de la solución>
    fin_si
fin_si
fin_mientras
fin_procedimiento

```

Un algoritmo que devuelve todas las soluciones se consigue modificando el anterior de la siguiente forma:

```

procedimiento Ensayar(E entero: i..)
inicio
    <inicializa las posibilidades>
    mientras <queden posibilidades> hacer
        <toma la siguiente posibilidad>
        si <la posibilidad es aceptable> entonces
            <anota la posibilidad como parte de la solución>
            si <se encontró solución general> entonces
                <presenta la solución obtenida>
            si_no
                ensayar(i + 1, éxito, ...)
            fin_si
        <desanota la posibilidad como parte de la solución>
    fin_si
fin_mientras
fin_procedimiento

```

Un algoritmo para la búsqueda de la solución óptima genera todas las posibilidades y se queda con la mejor:

```

procedimiento Ensayar(E entero: i..)
inicio
    <inicializa las posibilidades>
    mientras <queden posibilidades> hacer
        <toma la siguiente posibilidad>
        si <la posibilidad es aceptable> entonces
            <anota la posibilidad como parte de la solución>
            si <se encontró solución general> entonces
                si <la solución encontrada es óptima> entonces
                    <guarda solución>
            fin_si
        si_no
            ensayar(i + 1, éxito, ...)
        fin_si
        <desanota la posibilidad como parte de la solución>
    fin_si
fin_mientras
fin_procedimiento

```

PROBLEMAS BÁSICOS

- 4.1.** *Escriba una función recursiva que calcule el factorial de un número entero positivo y una función recursiva y una iterativa que calcule los números de Fibonacci; a continuación escriba un programa C que mediante recursividad indirecta decida si un número es par o impar.*

Análisis

Las definiciones recursivas de las funciones pedidas son:

- $\text{Factorial}(n) = n * \text{Factorial}(n-1)$ si $n > 0$ y 0 en otro caso.
- $\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$ si $n > 1$; $\text{Fibonacci}(n) = n$ en otro caso.
- $\text{Par}(n) = \text{Impar}(n-1)$ si $n > 1$. $\text{Impar}(n) = \text{Par}(n-1)$ si $n > 1$; $\text{Par}(0) = \text{True}$; $\text{Impar}(0) = \text{False}$.

Para eliminar la recursividad de la función Fibonacci basta con observar que el término n -ésimo depende del $n-1$ -ésimo y del $n-2$ -ésimo, por lo que si en dos variables auxiliares $a1$ y $a2$ se almacenan los términos inmediatamente anteriores, entonces para conseguir el siguiente término basta con sumar en una variable auxiliar $a3$ los valores de $a1$ con $a2$. De esta forma si $a1$ se inicializa $a0$ y $a2$ se inicializa $a1$, basta con usar un bucle ascendente que recalculé el término siguiente $a3$ como suma de $a1$ y $a2$ y posteriormente actualice $a1$ con $a2$ y $a2$ con $a3$.

Codificación

```
long Factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return (n * Factorial (n - 1));
}

long Fibonacci(int n)
{
    if (n == 0 || n == 1)
        return (n);
    else
        return(Fibonacci(n - 1) + Fibonacci(n - 2));
}

long FibonacciIterativa(int n)
{
    long a1 = 0; a2 = 1; a3, i;
    for ( i = 1; i <= n; i++)
    {
        a3 = a1 + a2;
        a1 = a2;
        a2 = a3;
    }
    return(a1);
}
```

El siguiente programa permite decidir si los números leídos desde teclado son pares o impares.

```
#include <stdio.h>
int par(int n);
```

```

int impar(int n);
void main(void)
{
    int n;
    do
    {
        printf("Introduzca número positivo: "); scanf("%d", &n);
    } while (n < 0);
    if (par(n))
        printf(" es par");
    else
        printf (" es impar");
}

int par(int n)
{
    if (n == 0)
        return 1;
    else
        return impar(n-1);
}

int impar(int n)
{
    if (n == 0)
        return 0;
    else
        return par(n - 1);
}

```

- 4.2.** *Escriba funciones del tipo “divide y vence” para resolver los siguientes problemas: sumar los dígitos de un número natural; calcular el máximo común divisor de dos números naturales positivos y calcular el producto de dos números naturales.*

Análisis

Las definiciones recursivas de las funciones solicitadas son:

- $\text{SumaRecursiva}(n) = n \bmod 10 + \text{SumaRecursiva}(n \text{ div } 10)$ *si* $n > 0$ *y* 0 *en otro caso.*
- $\text{Mcd}(m,n) = \text{Mdc}(n,m)$ *si* $m < n$; $\text{Mcd}(m,n) = \text{Mdc}(n,m)$ *si* $n < m$; $\text{Mcd}(m,n) = n$ *si* $n \leq m$ *y* $n \bmod m = 0$.
- $\text{Producto}(a,b) = 0$ *si* $b = 0$; $\text{Producto}(a,b) = a + \text{Producto}(a, b - 1)$ *si* $b > 0$.

Codificación

```

int SumaRecursiva(int n)
{
    if (n <= 0)
        return 0;
    else
        return (SumaRecursiva(n / 10) + n % 10);
}

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);
}

int Producto(int a, int b)
{
    if (b == 0)
        return 0;
    else
        return (a + Producto(a, b - 1));
}

```

PROBLEMAS AVANZADOS

- 4.3.** Se trata de diseñar un programa que simule un tablero de ajedrez (n filas y n columnas) y un caballo que se mueve según las reglas de dicho juego y averigüe si es posible que el caballo efectúe un recubrimiento del tablero completo; es decir visite cada cuadro del tablero exactamente una vez.

Análisis

Se representa el tablero mediante una matriz `Tab` cuyos índices varían entre $[1..n, 1..n]$ y cuyos valores son enteros. Esta matriz si contiene en una posición el valor cero indica que no ha sido pisada por el caballo y si contiene un número i distinto de cero indica el número de movimiento con el que ha sido visitada por el caballo. En un principio se inicializa la matriz `Tab` toda ella a ceros y se coloca el caballo en la casilla `[1][1]` haciendo que `Tab[1][1]=1` (esta posición inicial puede cambiarse), y se prueba a pasarlo a una nueva casilla siguiendo los movimientos estipulados en el juego del ajedrez para el caballo. *Anotar un movimiento* consiste en poner en la posición correspondiente del tablero el valor del número de movimiento. *Borrar un movimiento* consiste en poner en la posición correspondiente del tablero el valor de cero. La condición que determina si el problema se ha resuelto es que se haya pasado por las n^2 casillas. En ese momento se pone a `True` la variable `*exito`. Los únicos parámetros que tiene la función recursiva son los necesarios para realizar un nuevo movimiento. Estos son, el número de salto (i), las coordenadas actuales (`Posx, Posy`) y una variable por referencia `*exito` que indica si se ha completado el problema. La matriz que representa al tablero `Tab` es global. Las variables locales `Cx, Cy` y `Mov` representan las coordenadas de los posibles movimientos que resulten aceptables cuando no hayan sido ocupados previamente y no caigan fuera del tablero y los posibles movimientos. Los ocho posibles movimientos del caballo se obtienen sumando a la posición actual, los desplazamientos relativos que permiten obtenerlos; es decir:

```
Movimientos = {(2,1), (1,2), (-1,2), (-2,1), (-2,-1), (-1,-2), (1,-2), (2,-1)}
```

Codificación

```

#include <stdio.h>
#define N 8
#define n (N+1)
#define Ncuadrado N*N
/* Para representar el tablero en 1..n, 1..n*/

```

```

#define True 1
#define False 0

void Caballo(int i,int Posx,int Posy,int* exito);
void PresentaResultado();
int Tab[n][n];
int Movimiento[8][2]={2,1},{1,2},{-1,2},{-2,1},{-2,-1},{-1,-2},{1,-2},{2,-1}};
void Caballo(int i,int Posx,int Posy,int* exito)
{
    // esquema recursivo de una solución
    int Cx, Cy, Mov;
    *exito = False;
    Mov = 0;
    while ((Mov < 8) && !(*exito))
    {
        Cx = Posx+Movimiento[Mov][0];
        Cy = Posy+Movimiento[Mov][1];
        /* determina si nuevas coordenadas son aceptables */
        if ((Cx >= 1) && (Cx <= N) &&(Cy >= 1) && (Cy <= N) &&(Tab[Cx][Cy]==0))
        {
            Tab[Cx][Cy]= i;
            if (i < Ncuadrado)
            {
                Caballo(i+1,Cx,Cy,exito);
                if (!(*exito))
                {
                    /* no hay solución */
                    Tab[Cx][Cy] = 0;
                }
            }
            else
            {
                *exito = True;
            }
            Mov++;
        }
    }
}

void PresentaResultado()
{
    int i,j;
    for(i = 1; i <= N; i++)
    {
        for(j = 1; j <= N; j++)
        {
            printf("%d %c", Tab[i][j],(j < N ? ' ': '\n'));
        }
    }
}

void main(void)
{
    int solucion;
    Tab[1][1] = 1;
    Caballo(2, 1, 1, &solucion);
    if (solucion)
    {
        puts("\nEl problema tiene solucion:\n");
    }
}

```

```

        PresentaResultado();
    }
    else
        puts(" No se ha encontrado solucion al problema \n ");
}

```

- 4.4.** *Escribir un programa que encuentre una solución del problema de colocar $N = 8$ reinas en un tablero de ajedrez sin que se ataquen de auerdo con las reglas del ajedrez.*

Análisis

Para resolver el problema se colocan las reinas en un *array* (arreglo) *Reinas* de tal manera que si *Reinas[i]=j* hay una reina en la posición (*i* , *j*) del tablero. Para que haya coincidencia entre las posiciones del tablero (1,1) (1,2),.....(8,8) con las de los *arrays*, se define $N = 8$ y $n = 9$, de esta manera se pueden dimensionar los *array* $n \times n$, y sin usar la fila cero y la columna cero, hay coincidencia entre las posiciones reales y las de almacenamiento en las estructuras de datos. Las diagonales principales de un *array* $n \times n$ cumplen la condición $i - j = \text{constante}$ para todo par de índices *i*, y *j* que varíen respectivamente por las filas y las columnas. Más en concreto si *i* varía entre 1 y *n*, y la variable *j* entre 1 y *n*, entonces las diagonales principales son un total de 15 definidas por las constantes $-7, -6, \dots, 6, 7$, con lo que si le suma el valor de *N* se tiene que las diagonales principales varían entre 1 y 15. Las diagonales secundarias de un *array* $n \times n$ cumplen la condición $i + j = \text{constante}$, con lo que las diagonales secundarias varían entre los valores 2, 3,.....,15, 16. Como las reinas se ataran entre sí por filas, columnas, diagonales principales y diagonales secundarias, se usan los *arrays* lógicos *DiagPrin*[2*n], *DiagSec*[2*n], y *Col*[n] que toman el valor *True* si la posición está libre y *False* si está ocupada.

Hay que observar que la posición (*i*, *j*) del tablero define la diagonal principal *DiagPrin*[*i-j*+ *N*] y la diagonal secundaria *DiagSec*[*i+j*-1]. El tablero se rellena por filas consecutivas, y como cada reina se puede colocar en una columna, los posibles movimientos son las columnas *j* del tablero. La función *Anota* coloca una reina en la fila *i* y la columna *j*, poniendo *Reinas*[*i*]=*j*, *Col*[*j*]=*False*, *DiagPrin*[*i-j*+*N*]=*False*; y *DiagSec*[*i+j*-1]=*False*; . La función *BorraAnotacion* quita la reina de la posición fila *i* columna *j*, haciendo que *Reinas*[*i*]=0, *Col*[*j*]=*True*, *DiagPrin*[*i-j*+*N*]=*True* y *DiagSec*[*i+j*-1]=*True*. La función *Aceptable* decide si la posición fila *i* columna *j* están libre en el tablero consultado las estructuras de datos. La función recursiva es *Ensayareina* que se programa de acuerdo con lo indicado y el esquema general de encontrar una solución.

Codificación

```

#include <stdio.h>
#define N 8
#define n (N+1)
#define True 1
#define False 0

void Ensayareina(int i, int* solucion);
void EscribeSolucion();
void Inicializa();
void Anota(int i, int j) ;
void BorraAnotacion(int i, int j);
int Reinas[n];
int DiagPrin[2 * N], DiagSec[2 * N], Col[n];

void Inicializa()
{
    int i;
    for(i = 1; i < n; i++)
    {
        Col[i] = True;
    }
}

```

```
        Reinas[i] = 0;
    }
    for(i =1 ;i < 2 * N; i++)
    {
        DiagSec[i] = True;
        DiagPrin[i] = True;
    }
}

void Anota(int i, int j)
{
    Col[j]=False;
    DiagPrin[i - j + N] = False;
    DiagSec[ i + j - 1] = False;
    Reinas[i] = j;
}

void BorraAnotacion(int i, int j)
{
    Col[j] = True;
    Reinas[i] = 0;
    DiagPrin[i - j + N] = True;
    DiagSec[i + j - 1] = True;
}

int Aceptable( int i, int j)
{
    return (Col[j] && DiagPrin[i - j + N] && DiagSec[ i + j - 1]);
}

void EnsayaReina(int i, int* solucion)
{
    // esquema recursivo de una solución
    int j = 0;                                     /* inicializar posibles movimientos */
    do {
        j++;                                       /* tentativa de colocar reina i en fila j */
        if (Aceptable(i,j))
        {
            Anota(i,j) ;
            if (i < N)
                EnsayaReina(i+1,solucion);
            else
                *solucion = True;                /* todas las reinas colocadas */
            if (!(*solucion))
                BorraAnotacion(i,j);
        }
    } while(!(*solucion) && (j < N));
}

void EscribeSolucion()
{
    int i;
    puts("\n Número de columna se colocan cada reina\n");
    for ( i = 1; i <= N; i++)
```

```

        printf("\t Reina %d  en columna %d \n",i,Reinas[i]);
    }

void main(void)
{
    int TieneSolucion;
    TieneSolucion = False;
    Inicializa();
    Ensayareina(1,&TieneSolucion);
    if (TieneSolucion)
        EscribeSolucion();
}

```

Los resultados de ejecución del programa anterior son los siguientes:

Número de columna se colocan cada reina

```

Reina 1 en columna 1
Reina 2 en columna 5
Reina 3 en columna 8
Reina 4 en columna 6
Reina 5 en columna 3
Reina 6 en columna 7
Reina 7 en columna 2
Reina 8 en columna 4

```

4.5. Modificar el programa anterior para encontrar todas las posibles soluciones del problema de las $N=8$ reinas.

Análisis

El problema es similar al planteado en el Ejercicio 4.4, diferenciándose de él en que, dentro de la técnica de *Backtracking*, el algoritmo a aplicar es el *que* devuelve todas las soluciones. Por lo tanto se elimina la variable **solucion*. El bucle de las posibilidades prueba con todas, y cada vez que se encuentra una solución se escribe. Sólo se presenta el módulo recursivo *EnsayatodasLasSoluciones*. El resto de las funciones y declaraciones coinciden con las del ejercicio anterior.

Codificación

```

void EnsayatodasLasReinas(int i)
{
    // esquema recursivo de todas la soluciones
    int j = 0;                                     /* inicializar posibles movimientos */
    do { j++;
        if (Aceptable(i,j))
        {
            Anota(i,j) ;
            if (i < N)
                EnsayatodasLasReinas(i+1);
            else
                EscribeSolucion();
            BorraAnotacion(i,j);
        }
    } while(j < N);
}

```

Si se ejecuta la función `EnsayaNuevasReinas` con el programa del Ejercicio 4.5, se obtienen las 92 soluciones distintas del problema de las 8 reinas.

4.6. Se trata de colocar ocho torres en un tablero de ajedrez, sin que se ataquen. Se sabe que las torres se atacan por filas y columnas. Escribir un programa recursivo que:

a) Cuento el número total de soluciones del problema.

b) Encuentre una solución que minimice la siguiente función:

si las ocho torres se encuentran en las posiciones : $(1, j_1) (2, j_2) (3, j_3) (4, j_4) (5, j_5) (6, j_6) (7, j_7) (8, j_8)$

$$\sum_{j_i}^8 \text{abs}(\text{abs}(i - j_i) + \text{abs}(i + j_i))$$

c) Cuento el número de soluciones que minimizan la función expresada en b.

d) Presente una solución óptima.

Análisis

El programa es parecido al planteado en el Ejercicio 4.4 excepto que ahora las torres de acuerdo con las reglas del ajedrez sólo se atacan por filas y por columnas. Por lo tanto basta con conservar las Estructuras de datos `Reinas` que pasa a denominarse `Torres` y la estructura de datos `Col`. El tablero se representa de forma análoga a como se hace en el Ejercicio 4.4. La función `distancia` calcula la distancia expresada en el apartado b. La función `EscribeSolución` es parecida a la programada en el Ejercicio 4.4. Las funciones `Anota`, `BorraAnotación` y `Aceptable` son omitidas y escritas en el propio código recursivo debido a su sencillez. Se usa una variable `Cont` que cuenta las soluciones. La programación se realiza de acuerdo con el esquema de la mejor solución.

Codificación

```
#include <stdio.h>
#include <math.h>
#define N 8
#define n (N+1)
#define True 1
#define False 0
void EscribeSolucion();
void Inicializa();
int Torres[n], Optimo[n], Col[n];
float Cont=0;
float DistanciaActual, DistanciaOptima;

float Distancia()
{
    // calcula la distancia definida en el problema
    float aux=0;
    int i;
    for (i = 1; i < n; i++)
        aux= aux + fabs(Torres[i] - i) + fabs(Torres[i] + i);
    return aux;
}

void Copia(int x[n], int y[n])
{
    // copia x en y
    int i;
    for (i = 1; i < n; i++)
        y[i] = x[i];
}
```

```
}

void Inicializa()
{
    int i;
    for(i = 1; i < n; i++)
    {
        Col[i] = True;
        Torres[i] = 0;
    }
    DistanciaOptima= 32767; //infinito
}

void EscribeSolucion()
{
    int i;
    puts("\n Solución\n");
    for ( i = 1; i <= N; i++)
        printf("\t Torre %d  en fila  %d \n",i,Torres[i]);
}

void EnsayaNuevasTorres(int i)
{
    // esquema genérico de mejor solución
    int j = 0;                                     /* inicializar posibles movimientos */
    do
    {
        j++;
        if (Col[j])
        {
            Col[j] = False;
            Torres[i] = j ;
            if (i < N)
                EnsayaNuevasTorres(i+1);
        }
        else
        {
            DistanciaActual = Distancia();
            Cont++;
            if (DistanciaActual<DistanciaOptima)
            {
                Copia(Torres,Optimo);
                DistanciaOptima=DistanciaActual;
            }
        }
        Col[j] = True;
        Torres[i] = 0;
    } while(j < 8);
}

void main(void)
{
    Inicializa();
    EnsayaNuevasTorres(1);
}
```

```

    Copia(Optimo,Torres);
    printf("numero de soluciones %10.0f \n", Cont);
    EscribeSolucion();
}

```

La ejecución del programa anterior da el siguiente resultado:

```

número de soluciones      40320

Número de columna se colocan cada torre

Torre 1 en fila 1
Torre 2 en fila 2
Torre 3 en fila 3
Torre 4 en fila 4
Torre 5 en fila 5
Torre 6 en fila 6
Torre 7 en fila 7
Torre 8 en fila 8

```

- 4.7.** *Escribir un programa que, dado un conjunto de objetos con un peso y un valor, encuentre la selección de los mismos cuya suma de valores sea la más alta posible, pero cuya suma de pesos no supere un peso límite dado.*

Análisis

Como el objetivo del problema es conseguir el valor máximo, se considera que éste se puede alcanzar es la suma de los valores de cada objeto. Las posibilidades de cada objeto son incluirlo en la selección actual o excluirlo. Un objeto i es adecuado incluirlo en la selección actual, si al sumar su peso al peso acumulado en la selección en curso no sobrepasa el peso límite. El objeto i es adecuado excluirlo si al restar su valor al valor total todavía alcanzable por la selección en curso no es menor que el valor óptimo (máximo) encontrado hasta ahora. La inclusión de un objeto supone un incremento del peso acumulado en el peso que tenga asociado el objeto. A su vez excluir un objeto de la selección supone que el valor que puede alcanzar la selección tiene que ser decrementado en el valor del objeto excluido. El programa que se codifica usa la técnica de la mejor solución explicada en la teoría, podando las posibilidades con el límite de peso y con el valor alcanzable de acuerdo con lo expresado anteriormente.

Codificación

```

#include <stdio.h>
#define Maximo 14
struct Objeto
{
    int Peso, Valor;
};
struct
{
    int n; // número de elementos
    Objeto Objetos[Maximo]; // número máximo de objetos
} a;
struct Conjunto
{
    int k; // numero de objetos elegidos
    int b[Maximo]; // los índices de los objetos (índices de 0 a k-1)
} ;

```



```

Conjunto Selecció, SeleccionOptima;
/* 2 estructuras para almacenar la selección actual
   y la selección más óptima encontrada hasta el momento */
int LimiteDePeso;
void EnsayarObjeto(int i,int PesoAlcanzado,int ValorAlcanzable,int* MejorValor);
void LeerObjetos();
void EscribirOptimo(int mejor);

void main()
{
    int i, MaxValor, MejorValor;
    LeerObjetos();
    printf("Peso máximo = ");
    scanf("%d", &LimiteDePeso);
    /* Suma de los valores de cada objeto */
    for (MaxValor = i = 0; i < a.n; i++)
        MaxValor += a.Objetos[i].Valor;
    MejorValor = 0;
    Seleccion.k = SeleccionOptima.k = 0;
    /* el 0 es el primer objeto, el 2º 0 es el peso del primero
    MaxValor es el máximo valor alcanzable -> para hacer un podado en la exclusión
    MejorValor es la variable que nos tienen que dar */
    EnsayarObjeto(0, 0, MaxValor, &MejorValor);
    EscribirOptimo(MejorValor);
}

void LeerObjetos()
{
    int j;
    do
    {
        printf("Número de objetos: ");
        scanf("%d", &a.n);
    }
    while ((a.n < 1) || (a.n > Maximo));
    for (j = 0; j < a.n; j++)
    {
        printf("\tObjeto %d , <peso,valor> = ", j + 1);
        scanf("%d %d", &a.Objetos[j].Peso, &a.Objetos[j].Valor);
    }
}

void EnsayarObjeto(int i,int PesoAlcanzado,int ValorAlcanzable,int* MejorValor)
{
    // esquema de mejor solución
    int ValorExclusion;
    if (PesoAlcanzado+a.Objetos[i].Peso <= LimiteDePeso) /* objeto i se incluye */
    {
        // si entra es aceptable la inclusión del objeto
        Seleccion.b[Seleccion.k ] = i;
        Seleccion.k++;
        // incrementa el contador
        if (i < a.n-1)
            EnsayarObjeto(i+1,PesoAlcanzado+a.Objetos[i].Peso, ValorAlcanzable, MejorValor);
        Else
            //se está en una solución

```

```

        if (ValorAlcanzable > (*MejorValor))/*ver si es óptimo*/
        {
            // es una nueva solución mejor que la anterior. Hay que almacenarla.
            SeleccionOptima = Seleccion;
            (*MejorValor) = ValorAlcanzable;
        }
        Seleccion.k-- ;
    }
    /* proceso de exclusión del objeto i para seguir */
    /* proceso de exclusión del objeto i para seguir */
    ValorExclusion = ValorAlcanzable - a.Objetos[i].Valor;
    /*el valor que puede conseguir decrece*/
    if (ValorExclusion > (*MejorValor))
        /* se puede alcanzar una mejor selección, sino poda la búsqueda */
        if (i < a.n - 1)
            EnsayarObjeto(i+1, PesoAlcanzado, ValorExclusion, MejorValor);
        else
        {
            // se ha encontrado una solución mejor que la que tenía.
            SeleccionOptima = Seleccion;
            (*MejorValor) = ValorExclusion;
        }
    }
}

void EscribirOptimo(int Mejor)
{
    int j;
    printf("Selección de objetos que forma la selección óptima\n");
    for (j = 0; j < SeleccionOptima.k; j++)
        printf("Objeto %d = <%d,%d> \n", SeleccionOptima.b[j] + 1,
            a.Objetos[SeleccionOptima.b[j]].Peso, a.Objetos[SeleccionOptima.b[j]].Valor);
    printf(" Valor óptimo = %d \t para un peso máximo = %d", Mejor, LimiteDePeso);
}

```

Si la función LeerObjetos se cambia por función leeautomatico:

```

void Leeautomatico()
{
    a.n= 4;
    a.Objetos[0].Peso=10; a.Objetos[0].Valor=18;
    a.Objetos[1].Peso=11; a.Objetos[1].Valor=20;
    a.Objetos[2].Peso=12; a.Objetos[2].Valor=17;
    a.Objetos[3].Peso=13; a.Objetos[3].Valor=19;
}

```

que inicializa el valor de n a 4 y el resto de los pesos y valores tal y como aparece indicado en la codificación, y además se introduce como dato de límite de peso el valor de 40 entonces se obtiene como resultado la siguiente ejecución.

```

Peso máximo = 40
Selección de objetos que forma la selección óptima
Objeto 1 = <10,18>
Objeto 2 = <11,20>
Objeto 4 = <13,19>
Valor óptimo = 57           para un peso máximo = 40

```

- 4.8. Obtener todos los números de m ($m \leq 9$) cifras, todas ellas distintas de cero y distintas entre sí, de tal forma que el número formado por las n primeras cifras, cualquiera que sea n ($n \leq m$), sea múltiplo de n . Por ejemplo para $m = 4$ son números válidos, entre otros, los siguientes:

1236 ya que 1 es múltiplo de 1, 12 de 2, 123 de 3 y 1236 de 4.

9872 pues 9 es múltiplo de 1, 98 de 2, 987 de 3 y 9872 de 4.

Análisis

Para resolver el problema se tiene en cuenta que si se representa el número de m cifras con un *array* d de m componentes cuyos valores varían entre 1 y 9 las reglas de divisibilidad son las siguientes:

- Un número es divisible por dos si termina en un número par.
- Un número es divisible por tres si la suma de los valores de sus dígitos es divisible por tres.
- Un número es divisible por cuatro si sus dos últimas cifras son divisibles por cuatro.
- Un número es divisible por cinco si termina en cero o en cinco.
- Un número es divisible por seis si es divisible por dos y por tres.
- Un número de siete cifras es divisible por siete si $\text{abs}(s) \bmod 7 = 0$, siendo
 $s = d[1] \cdot d[4] + d[7] + 3 \cdot (d[6] \cdot d[3]) + 2 \cdot (d[5] \cdot d[2])$
- Un número es divisible por ocho si el número formado por sus tres últimas cifras es divisible por ocho.
- Un número es divisible por nueve si la suma de los valores de sus dígitos es divisible entre nueve.

El algoritmo que se presenta se estructura de la siguiente forma:

- La función `divisible` da la condición de divisibilidad de acuerdo con las reglas indicadas.
- La función `EscribeSolucion` escribe la solución actual almacenada en el *array* `d` global.
- Una función `Recursivo`, que encuentra todas las soluciones. Para ello, en el parámetro conjunto `c` lleva los dígitos que ya han sido utilizados y que no pueden ser puestos en el número en construcción. De esta forma cuando un nuevo dígito es añadido al número, debe indicarse añadiéndolo al conjunto y suprimiéndolo de él cuando el dígito es eliminado. Los parámetros n , y m indican respectivamente la posición del dígito que se debe añadir al número, y el número de dígitos que debe contener el número. Cuando $n = m$ se tiene una solución y el número debe ser escrito. Por otro lado, puesto que en la posición n del número en construcción se puede colocar cualquier dígito de $1, 2, \dots, 9$, se debe poner un bucle de posibilidades que pase por todos estos números (el bucle `for`). Ahora bien puesto que los dígitos no pueden repetirse, necesariamente el dígito que se está probando no debe estar en el conjunto `c`, y además debe verificarse la condición de divisibilidad.
- Se usa una implementación de conjunto que es omitida.

Codificación

```
#include <stdio.h>
#include <math.h>
#define Maximo 10
#define True 1
#define False 0

int Cont, d[Maximo];
struct Conjunto
{
    .....
};
int Divisible(int n)
{
    int i, s,q;
    switch (n)
    {
```

```

    case 1,9: return (True);
    /*en este caso siempre es divisible por uno y por nueve. Debido a que
    la suma de las 9 cifras significativas distintas es divisible entre 9*/
    case 2: return (d[2] % 2 == 0);
    case 3,6: { s = 0;
                for (i = 1; i <= n; i++)
                    s += d[i];
                q = s % 3 == 0;
                if (n == 6)
                    if (q)
                        q = d[6] % 2 == 0;
                return q;
            }
    case 4: return ((10 * d[3] + d[4]) % 4 == 0);
    case 5: return (d[5] == 5);
    case 7: { s = d[1] - d[4] + d[7] + 3 * (d[6] - d[3]) + 2 * (d[5] - d[2]);
            return(int(fabs(s)) % 7 == 0);
        }
    case 8: return ((100 * d[6] + 10 * d[7] + d[8]) % 8 == 0);
    }
}

void EscribeSolucion(int m)
{
    int i;
    Cont++;
    for (i = 1; i <= m; i++)
        printf(" %d ", d[i]);
    printf("\n");
}

void Recursivo(int n, int m, Conjunto C)
{
    // esqema de todas las soluciones
    int i;
    for (i = 1; i <= 9; i++)
    {
        d[n] = i;
        if (!PerteneceConj(i, C) && Divisible(n))
            if (n == m)
                EscribeSolucion(m);
            else
            {
                AnadeConj(i, &C);
                Recursivo(n + 1, m, C);
                BorraConj(i, &C);
            }
    }
}

void main ()
{
    int m;

```

```

Conjunto C;
scanf("%d", &m);
Cont = 0;
VacíaConj(&C);
Recursivo(1, m, C);
printf(" total de orden m= %d es %d", m, Cont);
}

```

4.9. *El celebre presidiario Señor S quiere fugarse de la cárcel de Carabanchel, por el sistema de alcantarillado, pero tiene dos problemas:*

- *El diámetro de la bola que arrastra es de 50 cm y resulta ser demasiado grande para pasar por algunos pasillos.*
- *Los pasillos que comunican unas alcantarillas con otras tienen demasiada pendiente y sólo puede circular por ellos en un sentido.*

Ha conseguido hacerse con los planos que le indican el diámetro de salida de las alcantarillas, así como el diámetro de los pasillos y el sentido de la pendiente que lo conectan.

Suponer que:

- *El Número de alcantarillas es n.*
- *El Señor S se encuentra inicialmente en la alcantarilla 0 (dentro de la prisión).*
- *Todas las alcantarillas (excepto la 0) tienen su salida fuera de la prisión si bien puede que sean demasiado "estrechas" para sacar la bola fuera.*
- *Se puede pasar de un pasillo a otro a través de las alcantarillas "estrechas" aunque a través de dichas alcantarillas no se pueda salir al exterior.*
- *Todos los pasillos tienen un diámetro si bien éste puede ser demasiado "estrecho" para poder pasar con la bola.*

Escribir

- *Una función para generar aleatoriamente los diámetros de los túneles del alcantarillado así como los diámetros de salida de las distintas alcantarillas. (Tenga en cuenta que si se puede ir de la alcantarilla i a la j por el túnel, entonces no se puede ir por el túnel desde j hasta i)*
- *Una función de Backtracking que resuelva el problema del Señor S.*
- *Una función para escribir en pantalla el sistema de conexiones y otro procedimiento para escribir una posible solución.*

Análisis

El programa utiliza las siguientes estructuras de datos:

- Un array `ds` (Diámetro de salida) que contiene el diámetro de salida de cada una de las salidas.
- Un array bidimensional `a` de tal manera que la entrada `a[i][j]`, contiene el diámetro del túnel que une la alcantarilla `i` con la `j`. La pendiente del pasillo está implícita en el array `a`. Si `a[i][j]` contiene un diámetro entonces `a[j][i]` tiene menos infinito (no se puede pasar).
- La solución está representada en un registro con los campos número de pasos (`np`) y un array solución (`sol`) para almacenar la solución en curso.
- El conjunto `visitados`, sirve para controlar en la recursividad el no volver a pasar por la misma alcantarilla.

La función solicitada en el apartado 1 se ha realizado en `Inicializa` que genera los diámetros de la salida `ds` aleatoriamente, así como el diámetro de los túneles `a`. También inicializa la solución. La función solicitada en el apartado 2 es `recursiva` y lleva como parámetros la variable `i`, que indica la alcantarilla en la que se encuentra, y por la cual no puede salir al exterior. La variable booleana `Encontrado` se pondrá a `True` cuando encuentre una solución y el conjunto `visitados`, que informa de qué alcantarillas ya han sido visitadas. Las funciones solicitadas en 3 son: `EscribeDatos` y `SoluciónEncontrada`. No se incluye la implementación de conjuntos.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#define n 5
#define True 1
#define False 0
#define Dbola 50
#define Minfinito -10

struct Solucion
{
    int np, Sol[n];
};
struct Conjunto
{
};
int a[n][n];
int ds[n]; /*diametro de las alcantarillas de la salida*/
Solucion Solu ;

void Inicializa()
{
    int i, j, x, aux;
    randomize();
    for( i = 0; i < n; i++)
    {
        ds[i] = 40 + random(30); //diametro de salida de alcantarilla
        for (j = 0; j < i; j++) //genera aleatoriamente
            a[i][j] = 40 + random(30);
        for (j = i; j < n; j++)
            a[i][j] = Minfinito;
    }
    randomize();
    for (i = n - 1; i >= 0; i--)
        for (j = 0; j < i ; j++)
        {
            x = random(2);
            switch (x)
            {
                case 1:
                    aux = a[i][j];
                    a[i][j] = a[j][i];
                    a[j][i] = aux;
            }
        }
    Solu.np = 0;
    ds[0] = 0; /*por la primera alcantarilla no puede salirse*/
}

void EscribeDatos()
{
    int i, j;

```

```

for(i = -1 ; i < n; i++)                                //-1 diámetro de salida de alcantarilla
    printf("%5d",i );
printf("\n");
for (i = 0; i < n; i++)
{
    printf("%5d",ds[i]);
    for (j = 0; j < n; j++)
        if (a[i][j] != Minfinito)
            printf("%5d", a[i][j]);
        else
            printf("%5d",0);
    printf("\n\n");;
}
}

void Recursivo(int i, int *Encontrado, Conjunto * Visitados)
{
    // esquema de una solución
    int j = 0;;
    do                                                    //las posibilidades son todas las alcantarillas
    {
        if((a[i][j] != Minfinito) && (a[i][j] >= Dbola) && (!PerteneceConj(j,*Visitados)))
            /*aceptable, evitando ciclos*/
            {
                AnadeConj(j,Visitados);
                Solu.np ++;
                Solu.Sol[Solu.np] = j;
                if (ds[j] >= Dbola)
                    *Encontrado = True ; /*solución*/
                else
                {
                    /*avance de la recursividad*/
                    Recursivo(j, Encontrado ,Visitados);
                    if (!(*Encontrado))
                    {
                        BorraConj(j, Visitados );
                        Solu.np --;
                    }
                }
            }
        j++;
    }
    while(! ((j == n) || *Encontrado));
}

void SolucionEncontrada()
{
    int i;
    printf("solución\n");
    for (i= 1;i<= Solu.np;i++)
        printf("%5d %5d\n",Solu.Sol[i], a[Solu.Sol[i-1]][Solu.Sol[i]]);
    printf("\n");
};

```

```

void main(void)
{
    Conjunto Visitados;
    int Encontrado;
    Inicializa ();
    EscribeDatos();
    Solu.np = 0;
    Solu.Sol[Solu.np] = 0;
    VacíaConj(&Visitados);
    AnadeConj(0,&Visitados);
    Encontrado = False;
    Recursivo(0, &Encontrado, &Visitados);
    if (Encontrado)
        SolucionEncontrada();
    else
        printf("no hay solución");
}

```

- 4.10.** Se dispone de un tablero de ajedrez de orden $N \times N$ (8×8). Se dice que una casilla del tablero está cubierta si sobre dicha casilla está posicionado un caballo, o si existe un caballo en otra posición que con un movimiento pueda desplazarse hasta dicha casilla. Obsérvese que con un único caballo es posible cubrir un máximo de 9 casillas y un mínimo de 1. Se pide escribir un programa que determine el menor número de caballos que son necesarios para cubrir todas las casillas de un tablero de orden $N \times N$.

Análisis

Siempre hay como mínimo una solución (¡muy mala!). Poner un caballo en cada casilla del tablero. El caballo no se desplaza por el tablero de acuerdo con las reglas del caballo, sino que ocupa casillas. Por supuesto, siempre puede ponerse un caballo en una casilla que esté cubierta. Otra cosa es que interese para obtener el mínimo.

Las estructuras de datos y variables que usa el problema son:

- Dos *array* unidimensionales paralelos *a*, *b* para indicar las casillas que cubre el caballo de acuerdo con el Ejercicio 4.3. Equivalen a los movimientos allí indicados.
- Cuatro *arrays* bidimensionales *Pisado* que indica el número de veces que ha sido pisada o cubierta una casilla; *PisadoOptimo* sirve para llevar el óptimo hasta el momento. El *array* bidimensional *Caballo* sirve para llevar el número de caballo que ha sido asignado a una determinada casilla en la solución en curso. Si una casilla no tiene asignado caballo el valor de caballo será cero. La variable *CaballoOptimo* lleva el óptimo hasta el momento.
- La variable *NumeroDePisados* indica el número de casillas que se han pisado en la solución en curso. De tal manera que cuando *NumeroDePisados* valga $N \times N$ se tiene una solución.
- La variable *cab* indica el número de caballos que se han colocado en la solución en curso, y la variable *CabOpt* el número de caballos que se han colocado en la solución en curso óptima.

La solución se estructura en las siguientes funciones:

1. La función *solución* escribe una solución encontrada.
2. La función *ensayarn* resuelve recursivamente el problema. El parámetro *cab* indica el número de caballo que se va a poner en la solución en curso. El número de posibilidades son todas las casillas del tablero. Es aceptable una posición cuando el valor del *array* *Caballo* es cero (no tiene asignado caballo). A la hora de anotar hay que marcar en el *array* *Caballo* el número de caballo, y en todas las posiciones que se cubran desde la posición hay que decirlo en el *array* *Pisado*, de tal manera que si hay alguna que sea marcada de nuevo, el contador *NumeroDePisados* se debe incrementar en una unidad. Se tiene una solución cuando el número de casillas cubiertas sea todo el tablero, o lo que es lo mismo, cuando la variable *NumeroDePisados* valga $n \times n = N \times N$. En el caso de que se tenga una solución, habrá que decidir si la solución en curso es mejor que la óptima que se tiene hasta el momento. El resto de los detalles se indican en la propia función.

Codificación

```

#include <stdio.h>
#define N 8
#define N1 (N+1)
int a[9],b[9], Pisado[N1][N1], PisadoOptimo[N1][N1];
int Caballo[N1][N1], CaballoOptimo[N1][N1];
int ContDeSoluciones,NumeroPisados,Cabopt,n, Ncuadrado;

void Solucion()
{
    int i, j;
    for (i = 1; i < n; i++)
    {
        for (j = 1 ;j < n; j++)
            printf("%5d",Pisado[i][j]);
        printf(" valor de p = %d \n",NumeroPisados);
    }
    printf(" solución %d,\n", ContDeSoluciones);
}

void CopiaArray(int PisadoOptimo[N1][N1],int Pisado2[N1][N1])
{
    int i,j;
    for(i = 1; i < n; i++)
        for(j = 1; j < n; j++)
            Pisado2[i][j] = PisadoOptimo[i][j];
}

void ensayarn(int cab)
{
    int i, j, k, u, v;
    for (i = 1; i < n; i++)
        for (j= 1; j < n; j++)
            if (Caballo[i][j] == 0)
                /*ceptable, ya que no es accesible desde ningún caballo, y anotar*/
                {
                    Pisado[i][j]--;
                    if (Pisado[i][j] == -1)
                        NumeroPisados ++;
                    Caballo[i][j] = cab;
                    for (k =1; k <= 8;k++)
                        {
                            u = i + a[k];
                            v = j + b[k];
                            if ((u < n) && ( v < n) && (u>0) && (v >0))
                                {
                                    Pisado[u][v]--;
                                    if (Pisado[u][v] == -1)
                                        /*nueva casilla pisada*/
                                        NumeroPisados++;
                                }
                        }
                }
    if (NumeroPisados <Ncuadrado)

```

```

        if (cab < Cabopt)                /*se puede conseguir una solución mejor*/
            ensayarn (cab +1);
        else ;
    else                                /*solución ver si es óptima*/
        if (cab < Cabopt)
        {
            //Solucion();
            Cabopt = cab;
            CopiaArray(Pisado,PisadoOptimo);
            CopiaArray(Caballo,CaballoOptimo);
            ContDeSoluciones++;
        };
    /*siempre hay que borrar anotación*/
    if (Pisado[i][j] == -1)
        NumeroPisados --;
        Pisado[i][j] ++;
        Caballo[i][j] = 0;
        for (k = 1;k<= 8;k++)
        {
            u = i + a[k];v = j + b[k];
            if ((u < n) && (v < n) && (u > 0) &&(v >0))
            {
                if (Pisado[u][v] == -1)
                    NumeroPisados --;
                    Pisado[u][v] ++;
            }
        }
    }
}

void main(void)
{
    int i,j;
    printf("valor de n pequeño 3,4,5 mayor es muy lento ");
    scanf("%d",&n);
    Ncuadrado = n * n;
    n++;
    a[1] = 2; b[1] = 1; a[2] = 1; b[2] = 2; a[3] = -1; b[3] = 2;
    a[4] = -2; b[4] = 1; a[5] = -2; b[5] = -1; a[6] = -1; b[6] = -2;
    a[7] = 1; b[7] = -2; a[8] = 2; b[8] = -1;
    ContDeSoluciones=0;
    NumeroPisados=0;
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
        {
            Pisado[i][j] = 0;
            Caballo[i][j] = 0;
        }
    Cabopt = Ncuadrado + 1;
    NumeroPisados = 0;
    ensayarn(1);
    printf("caboptimo%d \n",Cabopt );
    for (i = 1; i < n; i++)

```

```

{
    for (j = 1; j < n; j++)
        printf("%5d", PisadoOptimo[i][j]);
    printf("\n");
}
printf("caballos\n");
for (i = 1; i < n; i++)
{
    for (j = 1; j < n; j++)
        printf("%5d", CaballoOptimo[i][j]);
    printf("\n");
}
}

```

La ejecución del programa anterior da el siguiente resultado, donde se observa que con cuatro caballos se han pisado todas las posiciones del array, ya que en todas las posiciones del tablero hay números negativos.

```

valor de n pequeño 3,4,5 mayor es muy lento 4
caboptimo 4

-1  -1  -1  -1
-1  -1  -1  -1
-2  -1  -1  -2
-1  -1  -1  -1

caballos
0   1   2   0
0   3   4   0
0   0   0   0
0   0   0   0

```

4.11. Un número Natural p se dice que es p -Extraño, si se cumple la siguiente condición: la suma de los dígitos que ocupan un lugar par es igual a p veces la suma de los dígitos que ocupan un lugar impar.

Escribir un programa que genere y escriba todos los números p -extraños de como máximo max dígitos (pes un dato). Modifique el apartado *a* para calcular el número p -Extraño más cercano a un número natural d de $n1$ dígitos

$1 \leq n1 \leq \text{max}(\text{dato})$. Para su cálculo debe usar la siguiente definición de distancia:

Si x es un número natural de $n1$ dígitos y d es un número natural de $n2$ dígitos.

si $n1 \neq n2$ $\text{distancia}(x, d, n1, n2) = \text{infinito}$

si $n1 = n2 = n$ $\text{distancia}(x, d, n, n) = \sum_{i=1}^n \text{abs}(x(i) - d(i)) * 2^i$

Análisis

Las estructuras de datos necesarias son un vector a para poder almacenar los números, y un registro s que almacene el vector y la distancia para el apartado *b*. La solución del ejercicio pasa por definir las funciones:

- Escribir, se encarga de mostrar un número.
- Solución, decide si un número es una solución.
- Distancia, calcula la distancia entre dos números de acuerdo con el enunciado.
- Ensayar, resuelve el apartado *a*. Para ello construye todos los números de Max cifras y los escribe.

- **Ensaryarb**, resuelve el apartado *b*. Es similar al apartado *a* excepto que cuando encuentra una solución además de no escribirla, calcula la distancia al dato, *d* y si satisface la condición de optimalidad la almacena. Esta función usa la función *CopiaArray*, copia un vector en otro.
- La función *main* lee el valor de *p*, para el apartado *a*, los valores de *n1*, del número que se almacena en *d*, y llama a las correspondientes funciones.

Solución

```
#include <stdio.h>
#include <math.h>
#define Max 5
struct Soluciones
{
    int a[Max], Distancia;
} ;
Soluciones s;
int d[Max];

void CopiaArray(int a[Max],int b[Max])
{
    int i;
    for(i = 0; i < Max; i++)
        b[i] = a[i];
}

void Escribir(int a[Max], int n)
{
    int i;
    for (i = 0; i <= n; i++)
        printf("%1d",a[i]);
    printf("\n");
}

int Solucion(int a[Max], int n, int p)
{
    int i = 0,s1 = 0,s2 = 0;
    for (;i <= n; i++)
        if (i % 2 == 1)
            s1 += a[i];
        else
            s2 += a[i];
    return (s1 == p * s2);
}

int Distancia(int a[Max],int n, int n1)
{
    int i, s, Pot;
    if (n1 == n)
    {
        s = 0; Pot = 2;
        for (i = 0; i <= n; i++)
        {
            s += (int)fabs(a[i] - d[i]) * Pot;
        }
    }
}
```

```

        Pot *= 2;
    }
    return( s);
}
return ( 32767);
}

void Ensayar(int a[Max],int n, int p)
{
    int i;
    if (n < Max)
        for (i = 0; i <= 9; i++)
        {
            a[n] = i;
            if (Solucion(a, n, p))
                Escribir(a, n);
            Ensayar(a, n + 1, p);
            a[n] = 0;
        }
}

void Ensayarb(int a[Max],int n,int n1, int p)
{
    int i,Dis;
    if (n < Max)
        for (i = 0; i <= 9; i++)
        {
            a[n] = i;
            if (Solucion(a, n, p))
            {
                Dis = Distancia(a,n,n1);
                if (Dis < s.Distancia)
                {
                    CopiaArray(a,s.a);
                    s.Distancia=Dis;
                }
            }
            Ensayarb(a, n + 1, n1, p);
            a[n] = 0;
        }
}

void main(void)
{
    int i,p, a[Max],n1;
    do
    {
        printf(" dato p ");
        scanf("%d",&p) ;
    }
    while (p <= 0);
    Ensayar(a, 0,p);
    printf(" parte b\n ");
}

```

/*para probar con otro dígito*/

/*para probar con otro dígito*/

```

do
{
    printf(" dato n ");
    scanf("%d",&n1) ;
}
while ((n1 < 1) || (n1 > Max));
for (i = 0; i < n1; i++)
{
    scanf("%d",&d[i]);
    s.a[i] = 0;
}
s.Distancea=32767;
Ensayarb(a, 0, n1 - 1,p);
printf(" optimo");
Escribir(s.a, n1 -1);
}

```

- 4.12.** Dadas las letras del abecedario $a, b, c, d, e, f, g, h, i, j$ y dos números enteros $0 < n \leq m \leq 10$, escribir un programa que calcule las variaciones de los m primeros elementos tomados de n en n .

Análisis

Las variaciones de m elementos tomadas de n en n son todas las posibles formas de elegir n elementos de entre los n de tal manera que dos formas distintas se diferencian entre sí en el orden de colocación de un elemento o bien en la naturaleza de uno de ellos. Es decir las variaciones de las letras a, b, c tomadas de dos en dos son: ab, ac, ba, bc, ca, cb . Para facilitar la solución se calculan las variaciones de los m primeros números naturales tomadas de n en n , y se van almacenando en un vector de a . Una vez calculada una variación mediante una indexación del vector de letras previamente inicializado a las correspondientes letras del alfabeto se hace la transformación en la función `EscribeVector` para escribir las variaciones de las m primeras letras tomadas de n en n .

El problema se resuelve aplicando el esquema general de encontrar todas las soluciones: En este caso las posibilidades son los números naturales $1, 2, 3, \dots, m$. Una posibilidad es `Aceptable` (realizado mediante la función `AceptableVariación`) si el número correspondiente no se ha colocado previamente en la solución parcial que se está obteniendo (es un número distinto o todos los ya almacenados en el vector). Anotar una posibilidad j en la posición i es poner en la posición i del vector a el elemento j ($A[j]=i$). borrar una posibilidad j de la posición i es realizar la operación ($a[ij]=i$). Se ha encontrado una solución cuando se han colocado ya los n elementos. En este caso debido a que se comienza a rellenar el vector a por la posición 0, se tiene que hay una solución cuando la variable que controla el avance de la recursividad i toma el valor de $n - 1 = nmenosuno$.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#define max 10
int a[max], nmenosuno;
char letras[max] = {'a','b','c','d','e','f','g','h','i','j'};

int EscribeVector( int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%3c", letras[a[ i ]]);
    printf("\n");
}

```

```

int RellenaVector( int n)
{
    int i;
    for (i = 0; i < n; i++)
        a[ i ] = 0;
}

int Anota (int i, int j)
{
    a[ i ]= j;
}

int Borra( int i, int j)
{
    a[ i ] = -1;                // pone un valor que no está dentro del rango.
}

int AceptableVariacion (int i,int j)
{
    // comprueba que j no está en el vector a desde la posición 0 hasta la i-1
    int k, sw=1;
    for(k = 0 ; k < i && sw; k++)
        sw =(a[ k ]!= j);
    return sw;
}

int Variaciones( int m, int n, int i)
{
    int j;
    for(j = 0; j < m; j++)
        if(AceptableVariacion(i,j))
        {
            Anota(i,j);
            if( i== nmenosuno)
                EscribeVector(n);
            else
                Variaciones(m, n, i + 1);
            Borra(i,j);
        }
}

int LeeDatos( int *m, int *n)
{
    do
    {
        printf(" datos de m y n m>=n por ejemplo 3 y 2 \n");
        scanf("%d %d",m,n);
    } while( *m<=0 || *n<=0 || *m<*n);
}

int main(int argc, char *argv[])
{
    int n ,m;
    LeeDatos( &m, &n);
}

```

```

nmenosuno=n-1;
RellenaVector(n);
system("cls");
printf( "resultados de la ejecucion\n");
Variaciones(m,n,0);
system("PAUSE");
return 0;
}

```

Si los datos de entrada del problema son 3 y 2 entonces el resultado de la ejecución del programa anterior es:

```

resultados de la ejecución
a  b
a  c
b  a
b  c
c  a
c  b
Presione una tecla para continuar...

```

- 4.13.** Dadas las letras del abecedario *a, b, c, d, e, f, g, h, i, j* y dos números enteros $0 < n \leq m \leq 10$, escribir un programa que calcule las Combinaciones de los *m* primeros elementos tomados de *n* en *n*.

Análisis

Las Combinaciones de *m* elementos tomadas de *n* en *n* son todas las posibles formas de elegir *n* elementos de entre los *n* de tal manera que dos formas distintas se diferencian entre sí sólo en la naturaleza de uno de ellos. Es decir las combinaciones de las letras *a, b, c* tomadas de dos en dos son: *ab, ac, bc*. El Análisis resuelto 4.12 es completamente válido para este problema si se cambia el criterio de posibilidad aceptable (realizado mediante la función `AceptableCombinación`). En este caso el número es aceptable si el número no se ha colocado previamente en la solución parcial que se está obteniendo y además es estrictamente mayor (es un número distinto o todos los ya almacenados en el vector y además es mayor que todos). Por tanto cambiando la función `AceptableVariacion` del Ejercicio 4.16 por la función `AceptableCombinacion` que se codifica se tiene resuelto el problema. La Codificación es la misma que la realizada en el ejercicio anterior. Sólomente se incluye la función indicada y la función recursiva `Combinaciones`.

Codificación

```

int AceptableCombinacion (int i, int j)
{
    // comprueba que que el vector a desde la posición 0 hasta la i-1 tiene datos crecientes

    int k, sw=1;
    for( k = 0 ; k < i && sw; k++)
        sw =(a[ k ] < j);
    return sw;
}

int Combinaciones( int m, int n, int i)
{
    int j;
    for(j = 0; j < m; j++)
        if(AceptableCombinacion(i,j))
        {
            Aota(i,j);
            if( i== nmenosuno)
                EscribeVector(n);
        }
}

```



```

        else
            Combinaciones(m, n, i+1);
        Borra(i,j);
    }
}

```

Si los datos de entrada del problema son 3 y 2 entonces el resultado de la ejecución del programa anterior es:

```

resultados de la ejecución
a b
a c
b c
Presione una tecla para continuar...

```

- 4.14.** Dadas las letras del abecedario $a, b, c, d, e, f, g, h, i, j$ y dos números enteros $0 < n \leq m \leq 10$, escribir un programa que calcule las variaciones con repetición de los m primeros elementos tomados de n en n .

Análisis

Las Combinaciones de m elementos tomadas de n en n son todas las posibles formas de elegir n elementos de entre los n de tal manera que dos formas distintas se diferencian entre sí por la naturaleza de algún elemento o por el orden de colocación de alguno de ellos. Además en este caso los elementos se pueden repetir. Es decir las variaciones con repetición de las letras a, b, c tomadas de dos en dos son: $aa, ab, ac, ba, bb, bc, ca, cb, cc$. El análisis resuelto 4.12 es completamente válido para este problema si se cambia el criterio de posibilidad aceptable (realizado mediante la función `AceptableVariaciónConRepeticion`). En este caso el número es siempre aceptable. Por tanto cambiando la función `AceptableVariacion` del Ejercicio 4.16 por la función `AceptableVariacionConRepeticion` que se codifica se tiene resuelto el problema. La Codificación es la misma que la realizada en el Ejercicio 4.13. Solamente se incluye la función indicada y la función recursiva `VariacionesConRepeticion`.

Codificación

```

int AceptableVariacionesConRepeticion (int i,int j)
{
    return 1;
}

int VariacionesConRepeticion( int m, int n, int i)
{
    int j;
    for(j = 0; j < m; j++)
        if(AceptableVariacionesConRepeticion(i,j))
        {
            Anota(i,j);
            if( I == nmenosuno)
                EscribeVector(n);
            else
                VariacionesConRepeticion(m, n, i + 1);
            Borra(i,j);
        }
}

```

Si los datos de entrada del problema son 3 y 2 entonces el resultado de la ejecución del programa anterior es:

```
resultados de la ejecución
a a
a b
a c
b a
b b
b c
c a
c b
c c
Presione una tecla para continuar...
```

4.15. Escribir un programa que usando los cuatro movimientos del laberinto, arriba, abajo, izquierda y derecha calcule: una solución, si existe; el total de las soluciones; la mejor solución en cuanto a menor número de pasos. Los datos de entrada se reciben en un fichero de texto `laberinto.dat` que tiene la siguiente estructura:

- La primera línea tiene dos números enteros que son el número de filas n_f y el número de columnas n_c del laberinto. (ambos números están en el rango 2..15 y no hace falta comprobar que lo cumplen)
- Las siguientes n_f líneas tienen un total de n_c números enteros cada una de ellas separados por blancos que pueden ser: ceros, unos, doses o treses o cuatros: cero indica muro (no se puede pisar la casilla); uno indica libre; dos indica libre con penalización de un paso; tres indica libre con penalización de dos pasos; cuatro indica libre con penalización de tres pasos. Es decir si se pasa por una de ellas se cuenta para el caso: Uno, un paso; Dos, dos pasos; Tres, tres pasos; Cuatro cuatro pasos.
- La última línea del fichero tiene cuatro números enteros F_e = fila de entrada. C_e = Columna de entrada. F_s = fila de salida. C_s = columna de salida.

Análisis

La solución del problema se ha planteado mediante las siguientes funciones:

- `Escribe`. Es una función que recibe como parámetro un laberinto así como el número de filas y columnas y lo presenta en pantalla.
- `Copia`. Copia un laberinto en otro, para poder hacer llamadas a una solución, todas las soluciones, o mejor solución.
- `ensayar`. Es la función recursiva que encuentra una solución. Recibe como parámetro las coordenadas x e y actuales del laberinto así como el número de movimiento por el que vamos en la variable i . Retorna en la variable `exit` la información de si se ha encontrado una solución. Usa el esquema general de todas las soluciones. Las posibilidades son los cuatro movimientos que se encuentran almacenados en los `array` globales `MovX`, y `MovY` previamente inicializados. Se calculan las coordenadas de las nuevas posiciones en las variables u , y v . Aceptable es que se encuentre dentro del tablero el movimiento posible dado en las variables u y v , así como que no tenga muro y que no haya sido pisado (contiene un valor que puede ser 1,2,3,4). Anotar movimiento es poner en las coordenadas u y v el valor del parámetro $-i$ para no confundir con los valores 1, 2, 3, 4. Previamente se almacena el valor 1,2,3,4 en una variable auxiliar `aux`, para poder restaurar el verdadero valor en la parte de borrar la anotación. Se está en una solución, cuando las coordenadas u y v coinciden con la f_s y c_s .
- `ensayart`. Es la función que encuentra el número de soluciones del problema. En este caso usa el esquema de todas las soluciones, y lo explicado para la función `ensayar`, pero cada vez que encuentra una solución se incrementa un contador `c` en una unidad.
- `ensayarmejor`. Encuentra la mejor solución con el esquema genérico. Hay que tener en cuenta que debido a las especificaciones del problema, las llamadas recursivas deben hacerse con el valor de $i + aux$, para tener en cuenta la penalización. Además anotar el movimiento pone el valor de $-i - aux$, por la misma razón.
- `leelaberinto`. Se encarga de abrir el archivo y cargar los datos en memoria.
- `main`. Realiza las inicializaciones correspondientes y las llamadas a las distintas funciones.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#define maxF 16
#define maxC 16
#define maxf (maxF-1)
#define maxc (maxC-1)
int MovX[4];
int MovY[4];
int nf, nc, fe, ce, fs, cs, c = 0, mejor = maxF*maxC;
int laberinto[maxF][maxC], laberintoopt[maxF][maxC];

int Escribe(int laberinto[maxF][maxC], int nf, int nc)
{
    int i, j ;
    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nc; j++)
            printf("%3d", laberinto[i][j]);
        printf("\n");
    }
}

void Copia(int laberinto[maxF][maxC], int aux[maxF][maxC])
{
    int i, j ;
    for (i = 1; i <= nf; i++)
        for (j = 1; j <= nc; j++)
            aux[i][j] = laberinto[i][j];
}

int ensayar(int i, int x, int y, int *Exito)
{
    int k, u, v, aux;
    k=-1;
    do
    {
        k++;
        u= x+ MovX[k]; v= y + MovY[k];
        if ( 1 <= u && u <= nf && 1 <= v && v <= nc)
            if (1 <= laberinto[u][v] && laberinto[u][v] <= 4)
            {
                aux = laberinto[u][v];
                laberinto[u][v] = -i;
                if ((u == fs) && (v == cs))
                    (*Exito) = 1;
                else
                    ensayar(i+1, u, v, Exito);
            }
        if (!(*Exito))
            laberinto[u][v] = aux;
    }
}

```

```

    } while (k < 3 && !(*Exito));
}

void Ensayart(int i, int x, int y)
{
    int k, u, v, aux;
    k=-1;
    do
    {
        k++;
        u = x+ MovX[k];
        v = y + MovY[k];
        if ( 1 <= u && u <= nf && 1 <= v && v <= nc)
            if (1 <= laberinto[u][v] && laberinto[u][v] <= 4)
            {
                aux = laberinto[u][v];
                laberinto[u][v] = -i;
                if ((u == fs) && (v == cs))
                    c++;
                else
                    Ensayart(i+1,u,v);
                laberinto[u][v]=aux;
            }
    } while (k < 3);
}

void ensayarmejor(int i, int x, int y)
{
    int k, u, v, aux;
    k=-1;
    do
    {
        k++;
        u= x + MovX[k];
        v = y + MovY[k];
        if ( 1 <= u && u <= nf && 1 <= v && v <= nc)
            if (1 <= laberinto[u][v] && laberinto[u][v] <= 4)
            {
                aux = laberinto[u][v];
                laberinto[u][v] = -i - aux;
                if ((u != fs) || (v != cs))
                    ensayarmejor(i + aux, u,v);
                else
                    if (i + aux < mejor) // es solución
                                            // es solución y es óptima
                    {
                        mejor = i + aux;
                        Copia(laberinto,laberintoopt);
                    }
                laberinto[u][v] = aux;
            }
    } while (k < 3);
}

```

```

void leelaberinto()
{
    int i,j;
    FILE *f;
    f=fopen("laber3.dat","rt");
    if (f==NULL)
    {
        puts(" error al abrir el archivo origen\n");
        system("PAUSE");
        exit(-1);
    }
    fscanf(f," %d %d\n", &nf,&nc);
    printf("  Filas = %d Columnas = %d\n",nf, nc);
    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nc; j++)
            fscanf(f, "%d",&laberinto[i][j]);
        fscanf(f,"\n");
    }
    fscanf(f,"%d%d%d%d",&fe,&ce,&fs,&cs);
    printf(" fe = %d ce = %d fs = %d cs = %d\n",fe,ce,fs,cs);
    fclose(f);
}

int main(int argc, char *argv[])
{
    int Exito,laberinto1[maxF][maxC];
    leelaberinto();
    Escribe(laberinto,nf,nc);
    Copia(laberinto,laberinto1);
    MovX[0] = 0; MovY[0] = 1; MovX[1] = -1; MovY[1] = 0;
    MovX[2] = 0; MovY[2] = -1; MovX[3] = 1; MovY[3] = 0;
    if (laberinto[fe][ce] == 0)
    {
        printf( " no se puede entrar\n");
        system("PAUSE");
    }
    else
    {
        {
            laberinto[fe][ce] =-1;
            Exito=0;
            ensayar(2,fe,ce,&Exito);
            if (Exito )
            {
                printf( "Encontrada solucion \n");
                Escribe(laberinto,nf,nc);
            }
        }
        else
            printf(" no solucion\n");
        system("PAUSE");
        if (Exito)
        {

```

```

        Copia(laberinto1,laberinto);
        Ensayart(2,fe,ce);
        printf( "numero total de soluciones = %d\n",c);
        system("PAUSE");
        Copia(laberinto1,laberinto);
        printf( " laberinto optimo\n");
        Escribe(laberinto,nf,nc);
        laberinto[fe][ce]=-laberinto[fe][ce];
        ensayarmejor(-laberinto[fe][ce],fe,ce);
        printf(" mejor solucion encontrada \n");
        Escribe(laberintoopt,nf,nc);
        printf(" numero de pasos dados = %d\n" , mejor);
        system("PAUSE");
    }
}

```

El problema se ha ejecutado con tres laberintos de entrada distintos cuyos resultados son los siguientes:

Primera ejecución:

```

Filas = 6 columnas = 6
fe = 4 ce = 1 fs = 5 cs = 6
1 1 1 0 1 1
1 1 1 0 1 1
1 1 1 0 1 1
2 3 3 0 1 1
1 3 2 4 1 1
1 1 0 0 0 0
Encontrada solucion
1 1 1 0 1 1
1 1 1 0 1 1
1 1 1 0 1 1
-1 -2 -3 0 1 1
1 3 -4 -5 -6 -7
1 1 0 0 0 0

```

```

Presione una tecla para continuar...
numero total de soluciones = 1335
Presione una tecla para continuar...
laberinto optimo
1 1 1 0 1 1
1 1 1 0 1 1
1 1 1 0 1 1
2 3 3 0 1 1
1 3 2 4 1 1
1 1 0 0 0 0
mejor solucion encontrada
1 1 1 0 1 1
1 1 1 0 1 1
1 1 1 0 1 1
-2 3 3 0 1 1
-3 -6 8 -12 -13 -14
1 1 0 0 0 0
numero de pasos dados = 14
Presione una tecla para continuar...

```

Segunda ejecución:

```

Filas = 6 Columnas = 6
fe = 4 ce = 1 fs = 5 cs = 6
1 1 1 0 1 1
1 1 1 0 1 1
1 1 1 0 1 1
1 1 1 0 1 1
1 1 1 0 1 1
1 1 1 0 1 1
1 1 0 0 0 0
no solución
Presione una tecla para continuar...

```

Tercera ejecución:

```

Filas = 6 Columnas = 6
fe = 4 ce = 1 fs = 5 cs = 6
1 1 1 0 1 1
1 1 1 0 1 1
1 1 1 0 1 1
0 1 1 0 1 1
1 1 1 1 1 1
1 1 0 0 0 0
no se puede entrar
Presione una tecla para continuar...

```

PROBLEMAS PROPUESTOS

- 4.1. (El juego de la hormiga borracha)** Se dispone de un terreno cuadrado rectangular de dimensiones $n*m$. En algunas cuadrículas se han colocado terrones de 3 tipos: (1) Terrones de azúcar; (2) Terrones de licor; (3) Terrones de azúcar con veneno. Una hormiga, puede moverse en cualquier dirección paso a paso, salvo que esté borracha. En este caso se supone conocido una función $Hipo(s,d)$ que da como salida un par de números $s \in \{1,2,3\}$ que indica cuántos pasos seguidos da; $d \in \{1,2,3,4,5,6,7,8\}$ {indica las ocho casillas limítrofes donde se encuentra la hormiga} que indica la dirección en la que se mueve esos s pasos.

Se coloca la hormiga sobria en la cuadrícula (x_i, y_i) en la dirección d_i . La hormiga se mueve en la dirección d_i , hasta que encuentra un terrón. Entonces:

- Si el terrón es del tipo 1 se lo come y avanza en la misma dirección que llevaba.
- Si es del tipo 2, se lo come, se emborracha y sigue avanzando según le diga la función $Hipo$ durante un total de 10 pasos seguidos (las borracheras son acumulables).

- Si es del tipo 3 y está sobria, se los come y sigue avanzando. Si está borracha se lo come y muere.
- Si la hormiga llega a los bordes del campo entonces:
- Si está sobria gira a la derecha y continúa.
 - Si está borracha choca contra la pared cada vez que intente ir en la dirección que no debe.
- Escriba un programa que ¡NARRE! las aventuras y desventuras de la hormiga hasta que muera o bien se libere tras un total de p sufridos pasos.

- 4.2.** Escribir un programa que tenga como entrada una secuencia de números enteros positivos (mediante una variable entera). El programa debe 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.

- 4.3.** Sea A una matriz cuadrada de $n \times n$ elementos, el determinante de A se puede definir de manera recursiva:
- Si $n = 1$ entonces $Deter(A) = A(1,1)$.

Si $n > 1$, 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 la que se encuentra el elemento. Se puede expresar:

$$Det(A) = \sum_{i=1}^n (-1)^{i+j} * A(i,j) * Det(Menor(A(i,j))) \text{ para todo } j$$

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 columna 1 para calcular el determinante.

- 4.4.** 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.
- 4.5.** Leer un número entero positivo $n < 10$. Calcular el desarrollo del polinomio $(x + 1)^n$. Imprimir cada potencia x^2 en la forma $"$.

Sugerencia:

$$(x + 1)^n = C_{n,n}x^n + C_{n,n-1}x^{n-1} + C_{n,n-2}x^{n-2} + \dots + C_{n,2}x^2 + C_{n,1}x^1 + C_{n,0}x^0$$

donde $C_{n,n}$ y $C_{n,0}$ son 1 para cualquier valor de n .

La relación de recurrencia de los coeficientes binomiales es:

$$C(n, 0) = 1$$

$$C(n, n) = 1$$

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

- 4.6.** Escribir un programa para resolver el problema de la subsecuencia monótona creciente más larga. La entrada es una secuencia de n números a_1, a_2, \dots, a_n . Hay que encontrar las subsecuencias más largas $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ que $a_{i_1} < a_{i_2} < a_{i_3} < \dots < a_{i_k}$ y que $i_1 < i_2 < i_3 < \dots < i_k$. 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, 8.
- 4.7.** Dados n números encontrar combinación con sumas o restas que más se aproxime a un objetivo $"$. La aproxi-

mación puede ser por defecto o por exceso. La entrada son los $"$ números y el objetivo y la salida la combinación más próxima al objetivo.

- 4.8.** 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 $1 \leq j \leq 8$. 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.
- 4.9.** 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.
- 4.10.** Desarrollar un método recursivo que cuente el número de números binarios de n -dígitos que no tengan dos 1 en una fila. (Sugerencia: El número comienza con un 0 ó un 1. Si comienza con 0, el número de posibilidades se determina por los restantes $n-1$ dígitos. Si comienza con 1. ¿Cuál debe ser el siguiente?)
- 4.11.** Un palíndromo es una palabra que se escribe exactamente igual leído en un sentido o en otro. Palabras tales como *level*, *deed*, *ala*, etc., son ejemplos de palíndromos. Aplicar el esquema de los algoritmos *divide y vence* para escribir una función recursiva que devuelva 1, si una palabra pasada como argumento es un palíndromo y devuelva 0. Escribir un programa en el que se lea una cadena hasta que ésta sea palíndromo.
- 4.12.** Escribir un programa que mediante la técnica de *Backtracking* encuentre, si es posible, la manera de colorear un mapa con el mínimo número de colores. Para que un color sea viable para un determinado país, debe ocurrir que sea distinto del que tienen asignado todos los países limítrofes.

Algoritmos de búsqueda y ordenación

Muchas actividades humanas requieren que a diferentes colecciones de elementos utilizados se pongan en un orden específico, por lo que 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 y búsqueda básicos y avanzados más usuales así como su implementación en C. De igual modo, se estudiará el análisis de los diferentes métodos de ordenación con el objeto de conseguir la máxima eficiencia en su uso real.

5.1. Búsqueda

La búsqueda permite la recuperación de datos previamente almacenados. Si el almacenamiento se realiza en memoria, la búsqueda se denomina interna. Los métodos de búsqueda en listas ordenadas más usuales: *lineal* y *binaria*.

5.1.1. BÚSQUEDA LINEAL

La **búsqueda lineal** consiste en recorrer cada uno de los elementos hasta alcanzar el final de la lista de datos. Si en algún lugar de la lista se encontrara el elemento buscado, el algoritmo deberá informarnos sobre *la o las posiciones* donde ha sido localizado. Si la lista se encuentra ordenada y el dato a buscar no está en la lista, la búsqueda termina cuando el elemento a encontrar sea menor que el elemento en curso en una lista ordenada de modo ascendente, o cuando sea mayor si se busca en una lista ordenada de modo descendente.

EJEMPLO 5.1. *El siguiente código corresponde a una búsqueda secuencial en un vector en el que no existen elementos repetidos.*

```
int BusquedaSec (int vect[], int n, int e )
{
    int j = 0;
    for (j = 0; j < n; j++)
        if (e == vect[j])
            return j;
    //no encontrado
    return (-1);
}
```

5.1.2. BÚSQUEDA BINARIA

Este método de búsqueda requiere que los elementos se encuentren almacenados en una estructura de acceso aleatorio de forma ordenada, es decir clasificados, con arreglo al valor de un determinado campo. La búsqueda binaria consiste en comparar el elemento buscado con el que ocupa en la lista la posición central y, según sea igual, mayor o menor que el central, parar la búsqueda con éxir, o bien, repetir la operación considerando una sublista formada por los elementos situados entre el que ocupa la posición $\text{central}+1$ y el último, ambos inclusive, o por aquellos que se encuentran entre el primero y el colocado en $\text{central}-1$, también ambos inclusive. El proceso termina con búsqueda en fracaso cuando la sublista de búsqueda se quede sin elementos.

EJEMPLO 5.2. *El siguiente pseudocódigo corresponde a una búsqueda binaria en un vector ordenado en el que no existen elementos repetidos.*

```
entero funcion BusquedaBin (E/ Tvector vect; E/ Entero n, e)
inicio
    entero izq, der, central
    izq ← 1; der ← n;
    mientras( izq ≤ der )
        central ← (izq + der) div 2;
        si( vect[central] = e )
            devolver central
        si( e > vect[central] )
            izq ← central + 1;
        sino
            der ← central - 1;
    fin si
fin mientras
//no encontrado
devolver (-1);
fin
```

5.2. Clasificación interna

Clasificación interna es la clasificación u ordenación de datos que se encuentran en la memoria principal del ordenador. Esta clasificación podrá ser efectuada en orden ascendente o descendente. Los métodos de clasificación interna se dividen en dos tipos fundamentales al evaluar su complejidad en cuanto al tiempo de ejecución:

Directos.	De complejidad $O(n^2)$:	burbuja (intercambio directo), selección e inserción.
Logarítmicos.	De complejidad $O(n \log n)$:	método del montículo, ordenación por mezcla o radixSort.

En este capítulo los métodos expuestos utilizan el orden ascendente sobre vectores (*arrays* unidimensionales).

EJEMPLO 5.3. *Ordenar cuatro números leídos desde teclado.*

Para ordenar cuatro números leídos desde teclado puede seguirse el siguiente método. Comparar la primera variable leída con la segunda e intercambiar sus valores si la primera es mayor, a continuación se comparan, e intercambian sus valores en caso necesario, la segunda con la tercera y la tercera con la cuarta. Con estas acciones se habrá conseguido que la última variable tenga el valor mayor. El proceso se repite para seleccionar el penúltimo y antepenúltimo valor mayor.

<i>a = 20</i>	<i>b = 10</i>	<i>c = 14</i>	<i>d = 7</i>
10	20	14	7
10	14	20	7
10	14	7	20
10	14	7	20
10	7	14	20
7	10	14	20

```
#include <stdio.h>
void intercambiar(int* a, int* b)
{
    int aux;
    aux = *a;
    *a = *b;
    *b = aux;
}

int main ( )
{
    int a, b, c, d;
    printf ("Introduzca cuatro números ");
    scanf("%d%d%d%d", &a, &b, &c, &d);
    if (a > b)
        intercambiar(&a, &b);
    if (b > c)
        intercambiar(&b, &c);
    if (c > d)
        intercambiar(&c, &d);
    if (a > b)
        intercambiar(&a, &b);
    if (b > c)
        intercambiar(&b, &c);
    if (a > b)
        intercambiar(&a, &b);
    printf ("Valores ordenados ascendentemente\n");
    printf ("%d %d %d %d\n", a, b, c, d);
    return 0;
}
```

5.3. Ordenación por burbuja

La ordenación por burbuja ("*bubble sort*") se basa en comparar elementos adyacentes de la lista (vector) e intercambiar sus valores si están desordenados. De este modo se dice que los valores más pequeños *burbujean* hacia la parte superior de la lista (hacia el primer elemento), mientras que los valores más grandes se *hunden* hacia el fondo de la lista. La ordenación por burbuja se basa en comparar elementos contiguos del vector e intercambiar sus valores si están desordenados. Si el vector tiene los elementos $a[0]$, $a[1]$, ..., $a[n-1]$. El método comienza comparando $a[0]$ con $a[1]$; si están desordenados, se intercambian entre sí. A continuación se compara $a[1]$ con $a[2]$. Se continua comparando $a[2]$ con $a[3]$, intercambiándolos si están desordenados,... hasta comparar $a[n-2]$ con $a[n-1]$ intercambiándolos si están desordenados. Estas operaciones constituyen la primera pasada a través de la lista. Al terminar esta pasada el elemento mayor está en la parte superior de la lista. El proceso descrito se repite durante $n-1$ pasadas teniendo en cuenta que en la pasada i se ha colocado el elemento mayor de las posiciones $0, \dots, n-i$ en la posición $n-i$. De esta forma cuando i toma el valor $n-1$, el vector está ordenado.

5.4. Ordenación por selección

El algoritmo de ordenación por selección de una lista (vector) de n elementos se realiza de la siguiente forma: se encuentra el elemento menor de la lista y se intercambia el elemento menor con el elemento de subíndice 0. A continuación, se busca el elemento menor en la sublista de subíndices $1 \dots n-1$, e intercambiarlo con el elemento de subíndice 1. Después, se busca el elemento menor en la sublista $2 \dots n-1$ y así sucesivamente.

5.5. Ordenación por inserción

Este método consiste en tomar una sublista inicial con un único elemento que se podrá considerar siempre ordenada. En la sublista ordenada se irán insertando sucesivamente los restantes elementos de la lista inicial, en el lugar adecuado para que dicha sublista no pierda la ordenación. La sublista ordenada aumentará su tamaño cada vez que se inserta un elemento hasta llegar a alcanzar el de la lista original, momento en el que habrá terminado el proceso. Los métodos de ordenación por inserción pueden ser directo o binario, dependiendo de que la búsqueda se realice de forma secuencial o binaria.

5.6. Ordenación Shell

La idea general de algoritmo es la siguiente: se divide la lista original (n elementos) en $n/2$ grupos de dos con un intervalo entre los elementos de cada grupo de $n/2$ y se clasifica cada grupo por separado (se comparan las parejas de elementos y si no están ordenados se intercambian entre sí de posiciones). Se divide ahora la lista en $n/4$ grupos de cuatro con un intervalo o salto de $n/4$ y, de nuevo, se clasifica cada grupo por separado. Se repite el proceso hasta que, en un último paso, se clasifica el grupo de n elementos.

5.7. Métodos de ordenación por urnas

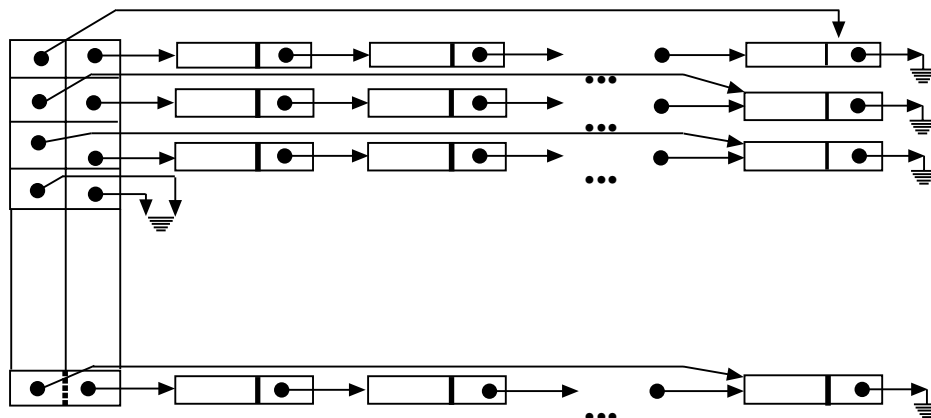
Son métodos que necesitan estructuras adicionales y unas características muy concretas en cuanto a claves.

5.7.1. BINSORT

Inicialmente se puede decir que el método consiste en tener un *array* U con tantas celdas o urnas como posibles valores de las claves de los elementos que deseamos ordenar ($0 \dots m-1$). Cada elemento del *array* a ordenar T se coloca en la urna que le corresponde, aquella cuyo subíndice coincide con la clave $U[T[i].clave]$. Al recorrer por orden las urnas los elementos salen ordenados. El *array* de urnas podría ser un *array* de listas y los elementos repetidos se colocarían en la lista de la urna correspondiente. El proceso de ordenación es el siguiente: inicializar todas las urnas a vacío, recorrer los elementos del *array* y según su contenido colocarlos en la urna $U[T[i].clave]$ y recorrer las urnas en orden ascendente colocando los elementos en el *array* que se pretende ordenar.

5.7.2. RADIXSORT

El método *RadixSort* es una modificación del método anterior que se emplea para efectuar la clasificación utilizando un número de urnas inferior al de posibles claves. La idea básica de la ordenación *Radixsort* (también llamada por residuos) es clasificar por urnas primero respecto al dígito (o letra cuando se trate de ordenaciones alfabéticas) de menor peso de la clave (menos significativo) d_k , después concatenar las urnas y clasificar de nuevo respecto al siguiente dígito d_{k-1} , y así sucesivamente hasta alcanzar el dígito más significativo d_1 , en ese momento la secuencia estará ordenada.



5.8. Ordenación rápida (*QuickSort*)

El método consiste en: dividir el *array* en dos particiones, una con todos los elementos menores a un cierto valor específico y otra con todos los mayores que él. Dicho valor recibe la denominación de pivote. Posteriormente se ordena los pequeños y después los mayores. Si el *array* no tiene datos el vector ya está ordenado.

5.9. Ordenación por mezcla

Este método de ordenación consiste en dividir el vector por su posición central en dos partes. Ordenar la parte izquierda, ordenar la derecha y después realizar la mezcla ordenada de ambas partes. Si el *array* no tiene datos, ya está ordenado.

5.10. Clasificación por montículo

Para analizar este método hay que tener en cuenta que un *array* se puede utilizar para efectuar la implementación de un árbol de la siguiente forma: en la primera posición del *array* se almacenará la raíz del árbol; Los hijos del nodo situado en la posición i del *array* están colocados en las posiciones $2*i$ y $2*i+1$. Un montículo en mínimo (en máximo) es una agrupación en forma piramidal de elementos en la que para cualquier nivel el peso de estos es menor o igual (mayor o igual) 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). Un montículo binario en mínimo 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 que el contenido de su hijos.

Usando la estructura de montón máximo se construye el montón máximo considerando que los nodos del montículo del último nivel del árbol son cada cada uno un submontículo de 1 nodo. Subiendo un nivel en el árbol, se considera cada nodo como la raíz de un árbol que cumple la condición del montículo, excepto quizás en la raíz (su rama izquierda y derecha cumplen la condición ya que se está *construyendo de abajo a arriba*), entonces al aplicar la función *hundir* (reconstruye el montículo *hundiendo* la raíz) se asegura un nuevo submontículo que cumple la condición de ordenación. 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. Una vez construido el montículo se quita la raíz, llevándola a la última posición del *array*, ya que es el elemento mayor, y se coloca el elemento que estaba al final como raíz. Se rehace el montículo sin considerar el último elemento y se repite el proceso con un subarray que no tiene el último elemento. El proceso termina cuando el subarray a tratar conste de un único elemento.

PROBLEMAS BÁSICOS

5.1. *Escribir y analizar la complejidad del método de búsqueda secuencial.*

Análisis

La función que realiza la búsqueda secuencial de un elemento x en un *array* A de n elementos, se programa mediante un bucle voraz de izquierda a derecha. Retorna el valor -1 si el elemento no se encuentra en el *array* y la posición de la primera aparición en otro caso. Se busca cada elemento por turno, comenzando con el primero hasta que, o bien se encuentra el elemento deseado o se alcanza el final de la colección de datos. Una codificación en C es la siguiente:

Solución

```
int BusquedaSecuencial(float A[], int n, float x)
{
    int i, Enc = 0;
    i = 0;
    while ((i < n) &&(!Enc))
        if (A[i] == x)
```

```

        Enc = 1;
    else
        i++;
    if (!Enc)
        return(-1);
    else
        return (i);
}

```

Estudio de la complejidad

El mejor caso, se produce cuando el elemento buscado sea el primero que se examina, de modo que sólo se necesita una comparación. En el peor caso, el elemento deseado es el último que se examina, de modo que se necesitan n comparaciones. En el caso medio, se encontrará el elemento deseado aproximadamente en el centro de la colección, haciendo $n/2$ comparaciones. Su recurrencia es:

$$T(n) = \sum_{i=1}^n 1 = \varepsilon O(n)$$

Por consiguiente, el algoritmo es $O(n)$ en el peor caso.

5.2. Escribir y analizar la complejidad del método de búsqueda binaria.

Análisis

La función que realiza la búsqueda binaria de un elemento x en un *array* A de n elementos ordenado crecientemente retorna el valor -1 si el elemento no se encuentra en el *array* y la posición de una aparición en otro caso. Para ello pone dos índices *Izq* y *Der* en los extremos del *array*. Se calcula la posición *Centro*. Si el elemento a buscar coincide con el que se encuentra en la posición *Centro*, se termina la búsqueda con éxito. En otro caso se mueve el índice *Izq* a la posición *Centro* +1 o bien el índice *Der* a la posición *Centro* -1. La búsqueda termina en fracaso si los índices *Izq* y *Der* se cruzan.

Codificación

```

int BusquedaBinaria(float A[], int n ,float x)
{
    int Izq, Der, Centro, Enc = 0;
    Izq = 0;
    Der = n-1;
    while ((! Enc) && (Izq <= Der))
    {
        Centro = (Izq + Der) / 2;
        if (A[Centro] == x)
            Enc = 1;
        else
            if (A[Centro] < x)
                Izq = Centro + 1;
            else
                Der = Centro - 1;
    }
    if (Enc)
        return (Centro);
    else
        return (-1);
}

```

Estudio de la complejidad

El algoritmo determina en qué mitad está el elemento y descarta la otra mitad. En cada división, el algoritmo hace una comparación. El número de comparaciones es igual al número de veces que el algoritmo divide el *array* por la mitad. Si se supone que n es aproximadamente igual a 2^k entonces k o $k+1$ es el número de veces que n se puede dividir hasta tener un elemento encuadrado ($k = \log_2 n$). Su recurrencia es:

$$T(n) = \sum_{i=1}^{\log(n)} 1 = \log_2(n) \in O(\log_2(n))$$

Por consiguiente el algoritmo es $O(\log_2 n)$ en el peor de los casos. En el caso medio $O(\log_2 n)$ y $O(1)$ en el mejor de los casos.

5.3. Escribir y analizar la complejidad del método de ordenación por selección.

Análisis del problema

Se utiliza la idea de que en cada iteración se busca la posición del elemento mayor del *array* (que no ha sido ya colocado) y lo coloca en la posición que le corresponde. En primer lugar se coloca en la posición $n-1$, luego en la $n-2$, posteriormente en la $n-3$ y así hasta la posición 1.

Codificación

```
void OrdenarSeleccion (float A[], int n)
{
    int i, j, jmax;
    float Aux;
    for (i = n - 1; i >= 1; i--)
    {
        jmax = 0;
        for (j = 0; j <= i; j++)
            if (A[j] > A[jmax])
                jmax = j;
        Aux = A[i];
        A[i] = A[jmax];
        A[jmax] = Aux;
    }
}
```

/* A[jmax] tiene la clave mayor*/

Estudio de la complejidad

Como primera etapa en el análisis de algoritmos se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. El procedimiento siempre realiza $n-1$ intercambios. (n , número de elementos del *array*). Existen $n-1$ llamadas a intercambio. El bucle interno hace $i-1$ comparaciones cada vez; el bucle externo itera $n-1$ veces, de modo que el número total de comparaciones es $C \in O(n^2)$, por lo que el número de comparaciones claves es cuadrático. Ha de observarse que el algoritmo no depende de la disposición inicial de los datos, esto supone una ventaja de un algoritmo de selección. Y que el número de intercambios de datos en el *array* es lineal. $O(n)$.

5.4. Escribir y analizar la complejidad del método de ordenación por burbuja.

Análisis

Se realiza la ordenación poniendo en primer lugar el elemento mayor en la última posición del *array*. A continuación se coloca el siguiente elemento mayor en la penúltima posición y así sucesivamente. Sólo se utilizan comparaciones de elementos consecutivos, intercambiándolos en el caso de que no estén colocados en orden.

Solución

```

void Burbuja( int  n, float  A[])
{
    int i,j;
    float  aux;
    for (i = 0; i < n-1; i++)
        for(j = 0; j < n - 1 -i; j++)
            if (A[j] > A[j + 1])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
            }
}

```

Estudio de la complejidad

En este método de ordenación, los dos bucles comienzan en cero y terminan en $n-2$ y $n-i-1$. El método de ordenación requiere como máximo $n-1$ pasadas a través de la lista o el *array*. Durante la pasada 1, se hacen $n-1$ comparaciones y como máximo $n-1$ intercambios, durante la pasada 2, se hacen $n-2$ comparaciones y como máximo $n-2$ intercambios. En general, durante la pasada i , se hacen $n-i$ comparaciones y a lo más, $n-i$ intercambios. Por consiguiente, en el peor caso, habrá un total de $(n-1) + (n-2) + \dots + 1 = n*(n-1)/2 = O(n^2)$ comparaciones y el mismo número de intercambios. Por consiguiente, la eficiencia del algoritmo de burbuja en el peor de los casos es $O(n^2)$.

5.5. Escribir y analizar la complejidad del método de ordenación por inserción lineal.**Análisis**

El subarray $0..i-1$ está ordenado, y se coloca el elemento que ocupa la posición i del *array* en la posición que le corresponde mediante una búsqueda lineal. De esta forma el *array* se ordena entre las posición $0..i$. Si i varía desde 1 hasta $n-1$ se ordena el vector.

Solución

```

void Insercionlineal( int  n, float  A[])
{
    int  i,j,enc,falso,verdadero;
    float  aux;
    falso = 0;
    verdadero = 1;
    for (i = 1; i < n; i++)
    {
        aux = A[i];
        j = i - 1;
        enc= falso;
        while (( j >= 0)&& !enc)
            if (A[j] > aux)
            {
                A[j + 1] = A[j];
                j--;
            }
    }
}

```

```

        else
            enc = verdadero;
        A[j + 1] = aux;
    }
}

```

Estudio de la complejidad

El bucle `for` de la función se ejecuta $n-1$ veces. Dentro de este bucle existe un bucle `while` que se ejecuta a lo más el valor de i veces para valores de i que están en el rango 1 a $n-1$. Por consiguiente, en el peor de los casos, las comparaciones del algoritmo vienen dadas por

$$1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2}.$$

Por otra parte, el algoritmo mueve los datos como máximo el mismo número de veces. Existen por lo tanto, en el peor de los casos, los siguientes movimientos:

$$\frac{n(n-1)}{2}$$

Por consiguiente, el algoritmo de ordenación por inserción es $O(n^2)$ en el caso peor.

5.6. Escribir y analizar la complejidad del método de ordenación por inserción binaria.

Análisis del problema

El subarray $0 \dots i-1$ está ordenado, y se coloca el elemento que ocupa la posición i del array en la posición que le corresponde mediante una búsqueda binaria. De esta forma el array se ordena entre las posición $0 \dots i$. Si i varía desde 1 hasta $n-1$ se ordena el vector.

Codificación

```

void Insercionbinaria( int  n, float  A[] )
{
    int  i,j,p,u,c;
    float  aux;
    for(i = 1; i < n; i++)
    {
        aux = A[i];
        p = 0;
        u = i - 1;
        while (p <= u)
        {
            c = ( p + u ) / 2;
            if(A[c] > aux)
                u = c - 1;
            else
                p = c + 1;
        }
        for(j = i - 1; j >= p; j--)
            A[j + 1] = A[j];
        A[p] = aux;
    }
}

```

Estudio de la complejidad

Los métodos de ordenación por inserción lineal e inserción binaria, sólo se diferencian entre sí conceptualmente en el método de la búsqueda. Por lo tanto el número de movimientos de claves será el mismo en ambos casos es:

$$\frac{n(n-1)}{2}$$

En cuanto al número de comparaciones, cuando el intervalo tiene i elementos, se realizan $\log_2(i)$ comparaciones. Por lo tanto el número de comparaciones será:

$$C = \sum_{i=1}^{n-1} \log_2(i) \approx n \log_2(n)$$

Es decir el número de comparaciones es $O(n \log_2(n))$, y el número de movimientos es $O(n^2)$.

5.7. Codifique en C el método de ordenación Shell.

Análisis del problema.

El método se basa en realizar comparaciones entre elementos que pueden ser no consecutivos, separados por una distancia salto. El valor de salto va decreciendo en cada iteración hasta llegar a valer uno, por ello compara elementos consecutivos. El vector se encontrará ordenado cuando salto valga uno y no se puedan intercambiar elementos consecutivos porque están en orden.

Codificación

```
void Shell( int  n, float  A[max])
{
    int  salto, k, j, ordenado;
    float  aux;
    salto = n ;
    while (salto > 1)
    {
        salto = salto / 2;
        do
        {
            ordenado = 1;
            for(j = 0; j <= n - 1 - salto; j++)
            {
                k = j + salto;
                if (A[j] > A[k])
                {
                    aux = A[j];
                    A[j] = A[k];
                    A[k] = aux;
                    ordenado = 0;
                }
            }
        } while (!ordenado);
    }
}
```

PROBLEMAS DE SEGUIMIENTO

5.8. Realizar un seguimiento del método de ordenación por selección para los datos de entrada;

3 7 9 9 0 8 9

Solución

Si se realiza un seguimiento del método de ordenación por selección se observa que en cada iteración elige la posición del elemento mayor y lo coloca en el lugar que le corresponde. Al escribir al final de cada iteración del bucle i los datos del vector, se tiene los resultados que se presentan posteriormente (se han obtenido al ejecutar la función del Ejercicio Resuelto 5.3 con la correspondiente modificación de escritura).

datos desordenados						
3	7	9	9	8	8	9
comienza seleccion						
3	7	9	9	0	8	9
3	7	8	9	0	9	9
3	7	8	8	9	9	9
3	7	0	8	9	9	9
3	0	7	8	9	9	9
0	3	7	8	9	9	9
datos ordenados						
0	3	7	8	9	9	0

5.9. Realizar un seguimiento del método de ordenación de la burbuja para los datos de entrada 11, 10, 14, 7.

Solución

Si se realiza un seguimiento del método de ordenación de la burbuja codificado en el ejercicio resuelto 5.4 y se escribe el contenido del vector cada vez que se realiza un intercambio de datos se obtienen los resultados que se presentan a continuación.

datos desordenados			
11	10	14	7
comienza Burbuja			
10	11	14	7
10	11	7	14
10	7	11	14
7	10	11	14
datos ordenados			
7	10	11	14

5.10. Realizar un seguimiento del método de ordenación por Inserción lineal para los datos de entrada:

7, 4, 13, 11, 3, 2, 7, 9

Solución

Si se realiza un seguimiento del método de ordenación de Inserción lineal codificado en el Ejercicio Resuelto 5.5 y se escribe el contenido del vector al final del bucle controlado por la variable i , se obtiene los resultados:

datos desordenados							
7	4	13	11	3	2	7	
comienza Insercionlineal							
4	7	13	11	3	2	7	
4	7	13	11	3	2	7	
4	7	11	13	3	2	7	
3	4	7	11	13	2	7	
2	3	4	7	11	13	7	
2	3	4	7	7	11	13	
2	3	4	7	7	9	11	
datos ordenados							
2	3	4	7	7	9	11	

- 5.11.** Realizar un seguimiento del método de ordenación por inserción binaria para los datos de entrada: 2, 7, 14, 3, 14, 13, 9, 1

Solución

Si se realiza un seguimiento del método de ordenación de inserción binaria codificado en el Ejercicio Resuelto 5.6 y se escribe el contenido del vector al final del bucle controlado por la variable *i*, se obtiene los resultados.

datos desordenados							
2	7	14	3	14	13	9	1
comienza Insercion Binaria							
2	7	14	3	14	13	9	1
2	7	14	3	14	13	9	1
2	3	7	14	14	13	9	1
2	3	7	14	14	13	9	1
2	3	7	13	14	14	9	1
2	3	7	9	13	14	14	1
1	2	3	7	9	13	14	14
datos ordenados							
1	2	3	7	9	13	14	14

- 5.12.** Realizar un seguimiento del método de ordenación Shell para los datos de entrada: 5, 4, 0, 14, 8, 1, 2, 3.

Solución

Al realizar un seguimiento del método de ordenación Shell codificado en el Ejercicio Resuelto 5.7 y si se escribe el contenido del vector al final del bucle controlado por la variable *j* cuando la variable *ordenado* está a *false*, se obtiene los resultados que se presentan. Se observa que en la primera línea de ejecución, después de comenzar Shell el salto vale 4. En la segunda línea el salto vale 2. En la tercera línea el salto sigue valiendo 2. En la tercera línea el salto vale 1 y el vector queda ordenado.

datos desordenados							
5	4	0	14	8	1	2	3
comienza Shell							
5	1	0	3	8	4	2	14
0	1	5	3	2	4	8	14
0	1	2	3	5	4	8	14
0	1	2	3	4	5	8	14
datos ordenados							
0	1	2	3	4	5	8	14

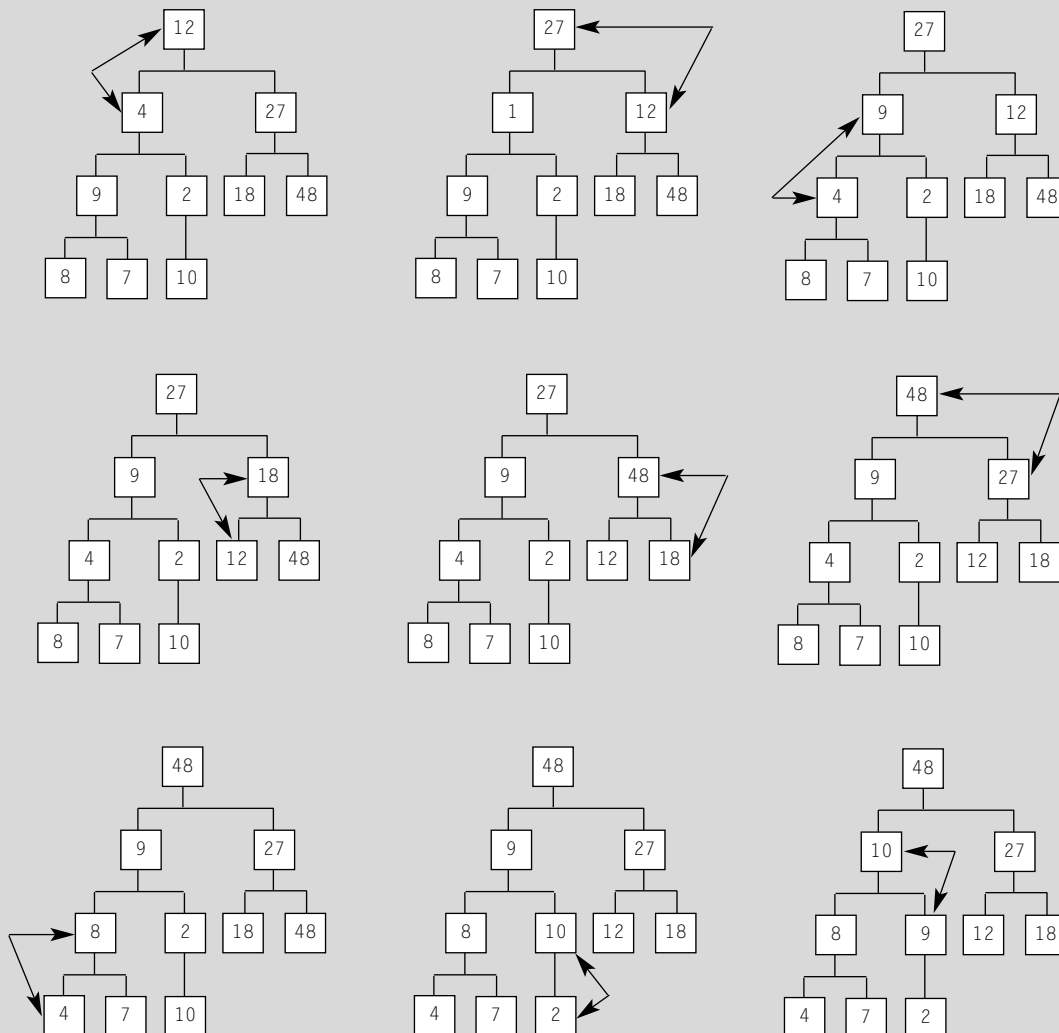
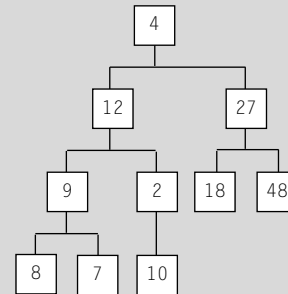
5.13. Usando la estructura de montón máximo hacer un seguimiento para realizar la ordenación del vector siguiente por el método del montón.

4	12	27	9	2	18	48	8	7	10
---	----	----	---	---	----	----	---	---	----

Seguimiento

Se puede considerar que es la implementación del siguiente árbol.
Para construir el montículo:

Desde el 2° ($i = 2$) hasta n vamos cogiendo cada elemento del *array* y le preguntamos por su padre, que estará en $(i / 2)$, si fuera mayor que el padre lo intercambiamos y subimos, de padre en padre, hacia arriba hasta que no haga falta intercambiar más o se termine el *array* por arriba.



La creación del montículo también se puede representar:

1	4	12	27	27	27	27	27	27	27	27	27	27	27	27	27	27
2	12	4	4	9	9	9	9	9	9	9	9	9	9	9	9	9
3	27		12	12	12	12	12	12	12	12	12	12	12	12	12	12
4	9			4	4	4	4	4	4	4	4	4	4	4	4	4
5	2					2	2	2	2	2	2	2	2	2	2	2
6	18						12	12	12	12	12	12	12	12	12	12
7	48								18	18	18	18	18	18	18	18
8	8										4	4	4	4	4	4
9	7												7	7	7	7
10	10														2	2

Quitar la raíz y rehacer el montículo.

48	2	27	27	7	18	18	4	12	12	8	10	10	4	9	9	2	8	8	2	7	2	4	2	2
10	10	10	10	10	10	10	10	10	10	10	2	9	9	4	8	8	2	4	4	4	4	4	2	4
27	27	2	18	18	7	12	12	4	7	7	7	87	7	7	7	7	7	7	7	2	7	7	7	7
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	4	4	4	2	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9	9	9	5	2	2	2	2	9	9	9	9	9	9	9	9
12	12	12	12	12	7	7	7	4	4	4	10	10	10	10	10	10	10	10	10	10	10	10	10	10
18	18	18	2	2	2	2	2	2	2	2	12	12	12	12	12	12	12	12	12	12	12	12	12	12
4	4	4	4	4	4	4	4	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
7	7	7	7	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
2	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48	48

ALGORITMOS AVANZADOS

5.14. Escribir y analizar el método de ordenación *RadixSort* para ordenar números naturales positivos.

Análisis

El programa que se presenta, usa una cola implementada con `frente` y `final`, (consúltase el tema de listas enlazadas) así como una función `EnlazarColas(Cola *C, Cola C1)` que enlaza con la cola `C` la cola `C1`, dando el resultado en `C`. El método `RadixSort`, se implementa para números enteros positivos menores o iguales que 32767. Las funciones `rellena` y `escribe` se encargan de rellenar de números aleatorios un vector y de presentarlo en pantalla.

Solución

```
#include <stdio.h>
#include <stdlib.h>
#define Max1 10
#define Max2 100
typedef int TipoDato;
```

```

struct Nodo
{
    TipoDato el;
    struct Nodo* sig;
};

typedef struct
{
    Nodo * Frente;
    Nodo * Final;
}Cola;

void EnlazarColas(Cola *C, Cola C1)
{
    if ((C->Frente != NULL) && (C1.Frente !=NULL))
    {
        C->Final-> sig = C1.Frente;
        C->Final = C1.Final;
    }
    else
    {
        if (C1.Frente != NULL)
        {
            C->Frente = C1.Frente;
            C->Final = C1.Final;
        }
    }
}

void RadixSort (TipoDato A[],int n)
{
    Cola U[Max1],C ;
    TipoDato Reg;
    int j,i, Aux, Aux1,Aux2, Exp, Cont;
    Aux =32767;
    Aux1 = 0;
    /* cuenta el número de dígitos que tiene 32767*/
    while (Aux > 0)
    {
        Aux = Aux / 10;
        Aux1++;
    }
    Exp = 1;
    for (Aux = 1; Aux <= Aux1; Aux++)
    {
        for (i = 0; i <= Max1 - 1; i++)
            VacíaC(&(U[i]));
        for (j = 0; j < n; j++)
        {
            Aux2 = A[j];
            i = (Aux2 / Exp) %10;           /*índice de la urna*/
            Reg = A[j];
            AnadeC(&(U[i]),Reg);
        }
        Exp = Exp * 10;
    }
}

```



```

    /*el frente y final de la lista U[0] son enlazados con el resto*/
    for (i = 1; i <= Max1 - 1; i++)
        EnlazarColas(&(U[0]), U[i]);
    C = U[0];
    Cont = 1;
    /* coloca de nuevo todos los elementos en el array A*/
    while (!EsVaciaC(C))
    {
        A[Cont - 1] = PrimeroC(C);
        Cont++;
        BorrarC(&C);
    }
}

void Rellena(TipoDato A[Max2], int *n)
{
    int i;
    randomize();
    *n = random(Max2);
    for(i = 0; i < (*n); i++)
        A[i] = random(Max2);
}

void Escribe( TipoDato A[], int n)
{
    int i;
    for(i = 0; i < n; i++)
        if((i + 1) % 10 == 0)
            printf("%7d\n",A[i]);
        else
            printf("%7d",A[i]);
}

void main (void)
{
    TipoDato a[Max2];
    int n;
    Rellena(a,&n);
    Escribe(a,n);
    RadixSort(a,n);
    printf(" datos ordenados\n");
    Escribe(a,n);
}

```

Estudio de la complejidad

La función `Rellena` y la función `Escribe`, son ambas de complejidad lineal. Por lo tanto la complejidad del algoritmo viene dada por la que se obtenga de la función `RadixSort`. Se supone que el vector de registros A tiene n elementos. Al ser el campo clave entero el número de urnas es $d = 10$. Además el número de dígitos de que consta el campo clave va a ser k . Con estas premisas y teniendo en cuenta los dos bucles anidados de que consta el algoritmo `RadixSort`, tenemos que el tiempo de ejecución es $O(k * n + k * d)$. Si las claves se consideran como cadenas binarias de longitud $\log(n)$ entonces $K = \log(n)$ y el método `Radix Sort` tomará un tiempo de ejecución $O(n \log n)$.

- 5.15.** Escribir y analizar una función recursiva que implemente el método de ordenación rápida *QuickSort* para ordenar un array de n elementos.

Análisis

El algoritmo de ordenación rápida divide el array a en dos subarrays (sublistas) que se pueden ordenar de modo independiente. Se selecciona un elemento específico del array $a[\text{centro}]$ llamado pivote y se divide el array original en dos subarrays que se ordenarán de modo independiente mediante llamadas recursivas del algoritmo.

Codificación

```
void OrdenaQuickSort( float a[n], int iz, int de)
{
    int i, j;
    float X, W;
    i = iz;
    j = de;
    X = a[(iz + de) / 2];
    do
    {
        while (a[i] < X)
            i ++;
        while (X < a[j])
            j --;
        if (i <= j)
        {
            W = a[i];
            a[i] = a[j];
            a[j] = W;
            i ++;
            j --;
        }
    }
    while (i <= j);
    if (iz < j) OrdenaQuickSort(a, iz, j);
    if (i < de) OrdenaQuickSort(a, i, de);
};
```

Estudio de la complejidad

Cada llamada recursiva al procedimiento de ordenación requiere una pasada a través del array y esto tarda $O(n)$. Eso significa que el número total de operaciones para cada nivel de recursión son operaciones $O(n)$. Por consiguiente, la ordenación rápida tiene un tiempo de ejecución que es $O(n)$ veces la profundidad de recursión. En otras palabras, la ordenación rápida tiene un tiempo de ejecución que es *(constante) * n * (profundidad de recursión)*. La profundidad de recursión será el número de veces que se puede dividir n por la mitad. El número de veces que se puede dividir n por 2 es aproximadamente $\log_2 n$. El algoritmo ejecuta $O(n)$ operaciones en cada nivel de recursión, y entonces el algoritmo de ordenación rápida se ejecutará en el tiempo $O(n \log_2 n)$ y se comporta como un algoritmo $O(n \log_2 n)$ en la mayoría de los casos. La ordenación rápida tiene un tiempo de ejecución en el caso medio que es $O(n \log_2 n)$. En el peor de los casos y si las particiones están muy desequilibradas, habrá n niveles de llamadas recursivas. Dado que la ordenación rápida ejecuta $O(n)$ operaciones en cada nivel de recursión, la ordenación rápida es, por consiguiente, un algoritmo $O(n^2)$ en el peor de los casos. El tiempo medio de ejecución es $O(n \log n)$ y en el peor de los casos es $O(n^2)$.

- 5.16.** *Escribir y analizar el método de ordenación del monton máximo. Escriba además una función que rellene aleatoriamente de datos el vector a ordenar y nos presente los resultados ya ordenados.*

Análisis del problema

Se considera la implementación del método de ordenación usando el montón máximo. No se tiene en cuenta el índice 0, ya que en este caso no se cumple la condición de tener dos hijos ($2*0 = 0$, $2*0+1 = 1$). La función `hundir` trabaja con un montón máximo, por lo tanto se deja bajar la clave por el camino donde se encuentre las claves máximas, y asciende la clave mayor. La función que ordena es `monton` y lo hace de la siguiente forma: un bucle descendente controlado por la variable `k` que comienza en la posición $n / 2$ y termina en la posición 1, construye un montón máximo para todas las claves del vector; posteriormente otro bucle también descendente que comienza en la posición `n` y termina en la posición 1, va tomando los elementos mayores del montón (que se encuentra en la posición 1), los intercambia con el elemento que ocupa el lugar que le corresponde a el mayor, y mediante la función `hundir` coloca el elemento que no cumple la condición del montón (ahora está en la posición 1) en una posición que sí que lo cumpla.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define Max2 100
#define randomize (srand time ( NULL ) ) //la función randomize genera la semilla
#define random( num ) ( rand()%(num)) //define la función random

void hundir ( int a[], int primero, int ultimo)
{
    int m,x;
    m = 2 * primero;
    x = a[primero];
    while (m <= ultimo)
    {
        if (m < ultimo)
            if( a[m] < a[m + 1])
                m++;
        /*con esto se consigue que m sea el hijo de primero con la clave mayor*/
        if (x >= a[m]) // x reside en la posición correcta*/
            m = ultimo + 1; //para salir del bucle */
        else
        {
            a[primero] = a[m];
            primero = m; m = 2 * primero;
        }
    }
    //en ese momento en primero es donde se debe meter x */
    a[primero] = x;
}

void monton ( int a[], int n)
{
    /*método de ordenación del montón*/
    int k, x;
    for (k = n / 2; k > 0; k--)
        hundir(a, k, n );
    /*se ha construido el monton*/
}
```

```

        for (k = n ; k > 0; k--)
        {
            x = a[k];
            a[k] = a[1];
            a[1] = x;
            hundir(a, 1, k - 1);
        }
    }

void Rellena(int A[], int *n)
{
    int i;
    randomize;
    *n=random(Max2);
    for(i = 1; i <= (*n); i++)
        A[i] = random(Max2);
}

void Escribe( int A[], int n)
{
    int i;
    for(i = 1; i <= n; i++)
        if((i) % 10 == 0)
            printf("%d\n",A[i]);
        else
            printf("%d",A[i]);
}

int main
{
    int a[Max2];
    int n;
    Rellena(a,&n);
    printf(" datos desordenados\n");
    Escribe(a,n);
    monton(a, n);
    printf("\n");
    printf(" datos ordenados\n");
    Escribe(a,n);
    system("PAUSE");
    return 0;
}

```

Estudio de la complejidad

Para analizar la complejidad del algoritmo, basta con observar que la función `monton` realiza una constante por n llamadas a la función `hundir`. Como esta función en cada iteración del bucle la variable `m` se multiplica por 2 y el valor inicial mínimo es 0 y el máximo es $n-1$ aporta una complejidad $\log_2(n)$. Es decir la complejidad del método de ordenación es $n \log_2(n)$. Los resultados de una ejecución aleatoria del programa anterior si se manda escribir el vector de datos antes de comenzar la ejecución, al terminar el `hundir` de cada llamada del bucle `for (k = n / 2; k > 0; k--)` del `monton`, y al terminar el `hundir` de cada llamada del bucle `for (k = n ; k > 0; k--)` que desamontona el `monton` para ordenar se obtiene el siguiente resultado:

datos desordenados								
11	14	7	12	9	2	8	14	4
<i>Sucesivas llamadas a hundir para construir monton</i>								
11	14	7	14	9	2	0	12	4
11	14	7	14	9	2	0	12	4
11	14	7	14	9	2	0	12	4
14	14	7	12	9	2	0	11	4
<i>Sucesivas llamadas a hundir para ordenar</i>								
14	12	7	11	9	2	0	4	14
12	11	7	4	9	2	0	14	14
11	9	7	4	0	2	12	14	14
9	4	7	2	0	11	12	14	14
7	4	0	2	9	11	12	14	14
4	2	0	7	9	11	12	14	14
2	0	4	7	9	11	12	14	14
0	2	4	7	9	11	12	14	14
0	2	4	7	9	11	12	14	14
datos ordenados								
0	2	4	7	9	11	12	14	14

5.17. Escribir una función que realice la ordenación interna de un vector de n elementos, por el método Shell mediante la técnica de inserción lineal.

Análisis

El método es una mejora de la ordenación por inserción directa que se utiliza cuando el número de elementos a ordenar es grande. En el método de clasificación por inserción cada elemento se compara con los elementos contiguos de su izquierda uno tras otro. Donald Shell modificó los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño con lo que en general se consigue una ordenación más rápida. El método se basa en fijar el tamaño de los saltos constantes, pero de más de una posición. Se inicializa `salto` a la mitad de la longitud de la lista. Los elementos del *array* que están a distancia `salto` constituyen una lista que hay que ordenar, por lo tanto hay un total de número de elementos del vector dividido por `salto` de listas. Entonces el *array* estará ordenado cuando `salto` valga uno y la única lista existente esté ordenada. Por lo tanto si `salto` al final vale uno, se consigue el objetivo. En la codificación `salto` se va dividiendo en cada iteración por la mitad, hasta que toma el valor de uno. Para ordenar cada una de las listas se usa la técnica de inserción lineal. Las listas terminan en la posición i que avanza desde `salto` hasta $n-1$ para tratar todas y cada una de ellas. El elemento que ocupa la posición i , todavía puede que no esté bien ordenando en lista. Para conseguirlo se inserta ordenadamente mediante la búsqueda secuencial en la lista formada por los elementos $A[i - \text{salto}]$, $A[i - 2 * \text{salto}]$, $A[i - 3 * \text{salto}]$, que sí está ordenada.

Codificación

```
void Shell( int n, float A[max])
{
    int i,j,k,salto;
    float aux;
    salto = n / 2;
    while (salto > 0)
    {
        for (i = salto; i < n; i++)
        {
            j = i - salto;
            while( j >= 0)
                /*inserción lineal en la lista que termina en i formada por elementos a distancia
```

```

        salto*/
        k = j + salto;
        if (A[j] <= A[k])
            j = -1;                                // ruptura de la búsqueda.
        else
        {                                           // intercambio de elementos
            aux = A[j];
            A[j] = A[k];
            A[k] = aux;
            j = j - salto;
        }
    }
    salto = salto / 2;
}

```

5.18. Escribir y analizar el método de ordenación “merge sort” de un vector de n elementos.

Análisis del problema

Este método de ordenación divide el vector por la posición central, ordena cada una de las mitades y después realiza la mezcla ordenada de las dos mitades. El caso trivial es aquel que recibe un vector con ningún elemento o con 1 elemento ya que obviamente está ordenado.

Codificación

```

#define M 20
#define Telemento float
void MezclaLista(Telemento a[], int Izq, int Centro, int Der);
void MergeSort(Telemento a[], int Izq, int Der)
{
    int Centro;
    if (Izq < Der)
    {
        Centro = (Izq+Der) / 2;
        MergeSort(a, Izq, Centro);
        MergeSort(a, Centro + 1, Der);
        MezclaLista(a, Izq, Centro, Der);
    }
}

void MezclaLista(Telemento a[], int Izq, int Centro, int Der)
{
    Telemento Temporal[M];
    int i, j, k;
    i = k = Izq;
    j = Centro + 1;
    /* bucle de mezcla, utiliza Temporal[] como array auxiliar, */
    while (i <= Centro && j <= Der)
    {
        if (a[i] <= a[j])
            Temporal[k++] = a[i++];
    }
}

```

```

else
    Temporal[k++] = a[j++];
}
    /* bucles para mover elementos que quedan de sublistas */
while (i <= Centro)
    Temporal[k++] = a[i++];
while (j <= Der)
    Temporal[k++] = a[j++];
    /* Copia de elementos de Temporal[] al array a[] */
for (k = Izq; k <= Der; k++)
    a[k] = Temporal[k];
}

```

Estudio de la complejidad

El algoritmo es recursivo esa razón que se requiere determinar el tiempo empleado por cada una de las tres fases del algoritmo *divide y vencerás*. Cuando se llama a la función `MezclaLista` se deben mezclar las dos listas más pequeñas en una nueva lista con los n elementos. La función hace una pasada a cada una de las sublistas. Por consiguiente, el número de operaciones realizadas será como máximo el producto de una constante multiplicada por n . Si se consideran las llamadas recursivas se tendrá entonces el número de operaciones: *constante* * n * (*profundidad de llamadas recursivas*). El tamaño de la lista a ordenar se divide por dos en cada llamada recursiva, de modo que la profundidad de las llamadas recursivas es aproximadamente igual al número de veces que n se puede dividir por dos, parándose cuando el resultado es menor o igual a uno. Por consiguiente, la ordenación por mezcla ejecutará, aproximadamente, el siguiente número de operaciones: alguna constante multiplicada por n y después multiplicada por $\log_2 n$. Resumiendo hay dos llamada recursivas, y una iteración que tarda tiempo n por lo tanto la recurrencia es:

$T(n) = n + 2T(n/2)$ si $n > 1$ y 1 en otro caso. De esta forma, aplicando expansión de recurrencias se tiene:
 $T(n) = n + 2T(n/2) = n + 2(n/2) + 4T(n/4) = \dots = n + n + n + \dots + n$ ($k = \log_2(n)$ veces) =
 $n \log_2(n) \in O(n \log_2(n))$. Por lo tanto la ordenación por mezcla tiene un tiempo de ejecución de $O(n \log n)$

- 5.19.** Dado un vector x de n (impar) elementos reales, diseñar una función que calcule y devuelva la mediana de este vector (valor tal que la mitad de los números son mayores o iguales y la otra mitad son menores o iguales).

Análisis

Para resolver el problema basta con encontrar el k -ésimo elemento de un vector de n elementos, para el caso de que k sea la mediana. Teniendo en cuenta que en el Ejercicio 2.11 se encuentra ya resuelto el problema del k -ésimo elemento así como el análisis del problema, la solución consiste en hacer una llamada con el valor de $n / 2 + 1$.

Codificación

```

float mediana( float a[ ], int n)
{
    return  (Kesimo(a,1,n,n/2+1));
}

```

PROBLEMAS PROPUESTOS

- 5.1. Un método de ordenación se dice que es estable si dos elementos que tienen la misma clave permanecen en el orden en que aparecieron en la entrada. Indicar qué métodos de ordenación del capítulo son estables y cuáles no.
- 5.2. Experimente el método de ordenación *QuickSort* comparando los tiempos de ejecución tomando como pivotes: el primer elemento del vector; el elemento que ocupa la posición central; el último elemento del vector.
- 5.3. La fecha de nacimiento de una persona está representada por el día, mes y año. Escribir el método *RadixSort* para ordenar por fecha (sin convertir fecha).
- 5.4. Escribir una función que elimine los elementos duplicados de un vector ordenado. Calcule su eficiencia.
- 5.5. Construir un método que permita ordenar por fechas y de mayor a menor un vector de n elementos que contiene datos de contratos ($n \leq 50$). Cada elemento del vector debe ser un registro 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 el montón.
- 5.6. Codificar el método de ordenación *RadixSort* para ordenar cadenas de caracteres.
- 5.7. 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* y *RadixSort*. Escribir el tiempo empleado en la ordenación de cada método.
- 5.8. Se quiere ordenar un *array* de registros según el campo clave *fecha de nacimiento*. Dicho campo consta de tres subcampos: día, mes y año de 2, 2 y 4 dígitos respectivamente. Adaptar el método de ordenación *RadixSort* a esta ordenación.
- 5.9. Escribir una función que elimine los elementos duplicados de un vector ordenado en orden decreciente. ¿Cuál es la eficiencia de esta función?
- 5.10. 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*.
- 5.11. 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*.
- 5.12. 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 20.000.
 - 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. La búsqueda debe ser binaria.
- 5.13. 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.
- 5.14. 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.

Archivos y algoritmos de ordenación externa

Se denomina **ordenación externa** aquella que se produce cuando todos los datos a ordenar no caben en la memoria principal de la computadora y están en un dispositivo de almacenamiento externo tal como un disquete o un disco óptico. En este caso es necesario aplicar nuevas técnicas de ordenación que se complementen con las ya estudiadas. Estas técnicas se basan en efectuar particiones y mezclas sucesivas de los datos del archivo original. Al mencionar la palabra *mezcla*, se quiere indicar la combinación de dos (o más) secuencias en una sola secuencia ordenada por medio de una selección repetida entre los componentes accesibles en ese momento. En el capítulo se revisa el tratamiento de archivos en C y se analizan los métodos de ordenación de archivos, ordenación externa, más populares y eficaces.

6.1. Archivos en C

Un *archivo de datos* es un conjunto de registros relacionados, que se trata como una unidad y se almacena sobre un dispositivo de almacenamiento externo. La *organización* de un archivo define la forma en la que los registros se disponen sobre el soporte de almacenamiento y se consideran tres tipos fundamentales: *secuencial*, *directa o aleatoria* (“*random*”) y *secuencial indexada* (“*indexed*”). Cada organización posee una forma propia de acceso a los registros; no obstante, sobre algunas de ellas, pueden programarse modos de acceso que no son los característicos de la organización. Los posibles modos de acceso son: *secuencial* y *directo*. El acceso secuencial se efectúa siempre que para llegar a un registro se recorren todos los precedentes, mientras que el acceso directo implica el posicionamiento directo sobre un determinado registro mediante la especificación de un índice, que da la posición del registro respecto al origen del archivo y sólo se puede implementar en soportes direccionables.

En C, la transferencia de datos a/o desde un dispositivo se hace mediante el concepto de **flujo** o *corriente* de datos que fluyen entre un origen o fuente y un destino; además hay que tener en cuenta que los datos transferidos se almacenan en un *buffer* intermedio desde el que sólo se vuelcan cuando, de una forma u otra, se da la orden de vaciarlo. Respecto a los flujos, C distingue dos tipos: de texto y binarios. Los **flujos de texto** se caracterizan por estar constituidos por una secuencia indefinida de caracteres que, en ocasiones, sufren conversiones para ser almacenados en la memoria principal; por el contrario en los **flujos binarios** existe correspondencia exacta entre la información almacenada en el dispositivo y la almacenada en memoria principal. Los flujos `stdin` y `stdout` son flujos de texto que representan la entrada y salida estándar y son automáticamente abiertos cuando se inicia la ejecución de un programa. Otro flujo de texto que se abre automáticamente es `stderr` que representa la impresora estándar.

6.2. Operaciones con archivos

El trabajo con archivos en C requiere la declaración de un puntero a una estructura predefinida que almacena información sobre el archivo, tal como la dirección del *buffer* que utiliza, el modo de apertura del archivo, el último carácter leído del *buffer*, etc..

La primera operación necesaria será pues la declaración de una variable de este tipo (`FILE*`) cuya estructura se encuentra definida en la biblioteca `stdio.h`.

```
#include <stdio.h>
FILE * ptrf
```

Las operaciones típicas de archivos son:

Creación. Consiste en definir el archivo mediante un nombre y unos atributos. Si el archivo existiera lo destruiría.

Apertura. Establece la comunicación con el dispositivo de soporte físico del fichero. La operación de abrir archivos se puede aplicar para operaciones de entrada, salida o entrada/salida.

Clausura. Cierra la conexión entre el identificador y el dispositivo de almacenamiento externo.

Lectura de datos. Copia los registros del fichero sobre variables en memoria central.

Escritura de datos. Copia la información contenida en variables sobre un registro del fichero.

Las instrucciones proporcionadas por C para efectuar las citadas operaciones se encuentran en la biblioteca `stdio.h` y pueden resumirse de la siguiente forma:

Creación y apertura de archivos

La primera operación para procesar un archivo es abrir dicho archivo. La sintaxis de la función de apertura `fopen` es

```
FILE * fopen (const char nombre_externo, const char modo_apertura);
```

Recibe como parámetros un puntero al nombre externo del archivo y otro al modo de apertura y devuelve un puntero a `FILE` o `NULL` si el archivo no puede abrirse. Mediante el `modo_apertura` 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 siguiente tabla:

modo	Significado
"r"	Abre para lectura.
"w"	Abre para crear nuevo archivo (si existe se pierden sus datos).
"a"	Abre para añadir al final.
"r+"	Abre archivo ya existente para modificar (leer/escribir).
"w+"	Crea un archivo para escribir/leer (si existe, pierde los datos).
"a+"	Abre el archivo para escribir/leer al final. Si no existe es como w.

A estos modos básicos se les añade, como último carácter de la cadena o como carácter intermedio, una `t` para indicar archivo de texto, o una `b` si se desea especificar binario.

Cierre

```
int fclose ( FILE * ptrf);
```

Recibe como parámetro un puntero de tipo `FILE` y devuelve 0 cuando la operación se realiza con éxito y `EOF`, constante definida en `stdio.h` con valor -1, si se genera algún error.

Lectura

```
int fgetc (FILE * ptrf);
```

Recibe un puntero de tipo `FILE` y devuelve el carácter leído del archivo abierto en modo lectura y pasado como argumento o `EOF` si se ha alcanzado el final del archivo.

```
char* fgets (char * cadena, int longitud, FILE * ptrf);
```

Lee una cadena del archivo pasado como argumento hasta completar (*longitud - 1*) caracteres o encontrar el carácter de salto de línea y completa la cadena añadiendo el carácter nulo como marca de fin. Devuelve `NULL` cuando se alcanza el fin de archivo o se produce un error.

```
int fscanf (FILE * ptrf , const char * formato, lista de variables);
```

Permite leer de un archivo una serie de entradas formateadas. El número de especificadores de formato incluidos en el parámetro *formato* y el de direcciones de variables deberá ser igual al número de campos. Los especificadores de formato (`%d`, `%f` ...) indican a C la transformación que debe de realizar con la secuencia de caracteres (conversión a entero ...). La función devuelve el número de campos leídos convertidos y almacenados con éxito o `EOF` cuando detecta el fin de archivo.

```
int fread(void * buffer, int tamaño, int n, FILE * ptrf);
```

Lee de un archivo *n* bloques de bytes y almacena lo leído en la parcela de memoria apuntada por *buffer*. El parámetro *buffer* suele ser el nombre de la variable donde los datos se almacenan, *tamaño* se refiere a la ocupación en memoria de la misma, determinable mediante `sizeof`, *n* suele ser `1` y *ptrf* es el puntero de tipo `FILE` al archivo de donde se efectúa la lectura. Esta sentencia permite leer datos estructurados. La función devuelve el número de elementos leídos o `0` en caso de error.

Escritura

```
int fputc (int c, FILE *ptrf);
```

Escribe el carácter que se le indica como primer parámetro en el archivo especificado como segundo parámetro. Devuelve el carácter escrito cuando la operación se realizó con éxito y `EOF` en caso contrario.

```
int fputs (const char * cadena, FILE * ptrf);
```

Escribe la cadena especificada como primer parámetro en el flujo especificado como segundo, suprime el carácter nulo que marca el final de la cadena y en su lugar escribe fin de línea. Devuelve `EOF` cuando se produce un error.

```
int fprintf (FILE * ptrf, const char * formato, lista de argumentos);
```

Escribe en el archivo asociado a *ptrf* los argumentos que se le pasan como parámetros tras aplicar a cada uno su correspondiente formato, incluido en la cadena *formato*. Devuelve `EOF` cuando se produce un error.

```
int fwrite (const void * variable, int tamaño, int n, FILE * ptr);
```

Escribe, en el archivo que se le indica como argumento, *n* elementos con la longitud en bytes especificada como segundo parámetro desde la zona de memoria apuntada por el primer parámetro. Devuelve el número de elementos escritos a través del nombre de la función.

Otros procedimientos y funciones de interés son:

Archivos de texto y binarios

```
int feof (FILE * ptrf);
```

Recibe un archivo como parámetro, devuelve un valor distinto de `0` (`true`) cuando se leer el carácter de fin de archivo, en caso contrario devuelve `0` (`false`). Esta función puede aplicarse tanto a archivos de texto como binarios.

```
void rewind (FILE * ptrf)
```

Sitúa el puntero de archivo al inicio del mismo.

Archivos binarios

```
int fseek (FILE * ptrf, long desplazamiento, int origen);
```

Sitúa el puntero del archivo en una posición aleatoria, dependiendo del desplazamiento y el origen relativo que se pasan como argumentos. Esta función devuelve un valor entero, distinto de cero si se comete un error en su ejecución; cero si no hay error.

El argumento `origen` puede tener tres valores representados por tres constantes definidas en `stdio.h`:

- 0 (`SEEK_SET`) : Cuenta desde el inicio del archivo.
- 1 (`SEEK_CUR`) : Cuenta desde la posición actual del puntero al archivo.
- 2 (`SEEK_END`) : Cuenta desde el final del archivo.

La función `fseek()` permite el acceso directo en archivos binarios y, aunque puede ser empleada con ellos, no es conveniente su uso en archivos de texto debido fundamentalmente a las conversiones de tipos que a veces se efectúan durante las operaciones de lectura o escritura en los mismos.

```
long int ftell (FILE * ptrf);
```

Recibe el archivo como parámetro y devuelve la posición actual como un número de bytes (entero largo) desde el inicio del archivo (byte 0).

Borrado de archivos

```
int remove(const char * nombre_archivo);
```

Borra el archivo cuyo nombre completo, incluida la ruta, se le pasa como parámetro. El archivo debe encontrarse cerrado.

Renombrar un archivo

```
int rename (const char * nombre_antiguo, const char * nombre_nuevo);
```

Renombra un archivo, cambia `nombre_antiguo` por `nombre_nuevo`.

EJEMPLO 6.1. *El siguiente programa abre para lectura del archivo de texto `Ejemplo.txt` que se supone está en el directorio raíz de la unidad actual, lo lee carácter a carácter y muestra su contenido en pantalla.*

```
#include <stdio.h>
int main(void)
{
    FILE *f;

    if ((f = fopen("\\Ejemplo.txt", "rt")) == NULL)
    {
        fprintf(stderr, "Cannot open input file.\n");
        return 1;
    }
    while (!feof(f))
        fputc(fgetc(f), stdout);
    fclose(f);
    printf("\n\nPulse ENTER para terminar");
    fputc(fgetc(stdin), stdout);
    return 0;
}
```

EJEMPLO 6.2. *Los archivos binarios permiten el posicionamiento directo en un determinado registro. Así, supuesto*

que desea efectuar la lectura del registro 6 en un archivo binario cuyos registros tienen la siguiente estructura.

```
typedef struct
{
    char nombre[34];
    int edad;
    int salario[12];
}Registro;
```

Tras efectuar las declaraciones adecuadas y abrir el archivo

```
// declaraciones
Registro reg;
FILE * punteroarchivo;

// permite leer desde un archivo binario existente
punteroarchivo = fopen("Ejemplo2.dat", "rb");
```

puede efectuarse el posicionamiento directo y la lectura mediante las siguientes instrucciones :

```
//posicionarse
fseek(punteroarchivo, 5*sizeof(Registro), SEEK_SET);

//leer
fread(&reg, sizeof(Registro), 1, punteroarchivo);
```

6.3. Ordenación externa

Para realizar operaciones de búsqueda, clasificación o mezcla en archivos, se puede recurrir a cargar los registros en *arrays* y, a continuación, seguir cualquiera de los métodos internos ya explicados.

En ocasiones, puede suceder que el volumen de datos a tratar sea demasiado grande y no quepa en la memoria interna del ordenador, resultando obligatorio entonces trabajar directamente con archivos, es decir con los datos en la memoria externa o auxiliar. Esto origina un aumento del tiempo de ejecución, debido a las sucesivas operaciones de lectura y escritura en el soporte físico.

Búsqueda externa es el proceso de localizar un registro en un archivo con un valor específico en uno de sus campos. Los archivos secuenciales obligan a realizar búsquedas secuenciales. Los archivos de organización directa son estructuras de acceso aleatorio, por lo que permiten un mayor número de posibilidades a la hora de realizar las búsquedas, pudiendo implementar búsquedas tanto secuenciales como directas y trabajar con ellos de modo similar a como se hace con los *arrays*. El método a aplicar dependerá de la forma de colocación de los registros en el archivo.

La **fusión** o mezcla consiste en reunir varios archivos en uno sólo intercalándose los registros de unos en otros y siguiendo unos criterios determinados. El caso más corriente es la mezcla de dos archivos que tienen sus registros colocados de forma secuencial y ordenados con respecto al valor de un determinado campo, el campo clave, para obtener un tercer archivo, también secuencial y en el que los registros sigan encontrándose ordenados con arreglo al contenido de dicho campo. El análisis del problema sería similar al ya estudiado relativo a los vectores.

Por **ordenación externa** se entiende la clasificación de los registros de un archivo directamente en la memoria auxiliar, ascendente o descendentemente con arreglo al valor de un determinado campo o a los valores de una jerarquía de campos. De acuerdo a los valores de una jerarquía de campos quiere decir, clasificar primero por el contenido de un determinado campo, a igualdad de valor en ese campo, clasificar por el contenido de otro y así sucesivamente.

EJEMPLO 6.3. Ordenar un archivo binario.

Los archivos binarios pueden ordenarse con métodos similares a los que se emplean para ordenar *arrays*. Así el archivo del ejemplo 6.2 podría ordenarse por el campo `nombre` mediante el método *QuickSort*.

```

void QuickSort(FILE *punteroarchivo, long inicio, long fin)
{
    long izq, der;
    char central[34];
    char cad[34];

    izq = inicio;
    der = fin;
    strcpy(central, LeerClave(punteroarchivo, (izq+der)/2, cad));
    do
    {
        while(strcmp(LeerClave(punteroarchivo, izq, cad), central) < 0    && izq < fin)
            izq++;
        while(strcmp(central, LeerClave(punteroarchivo, der, cad)) < 0    && der > inicio)
            der--;
        if(izq < der)
            Intercambiar(punteroarchivo, izq, der);
        if(izq <= der)
        {
            izq++;
            der--;
        }
    } while(izq <= der);
    if(inicio < der)
        QuickSort(punteroarchivo, inicio, der);
    if(izq < fin)
        QuickSort(punteroarchivo, izq, fin);
}

void Intercambiar(FILE *punteroarchivo, long izq, long der)
{
    Registro reg1, reg2;

    fseek(punteroarchivo, izq*sizeof(Registro), SEEK_SET);
    fread(&reg1, sizeof(Registro), 1, punteroarchivo);
    fseek(punteroarchivo, der*sizeof(Registro), SEEK_SET);
    fread(&reg2, sizeof(Registro), 1, punteroarchivo);
    fseek(punteroarchivo, izq*sizeof(Registro), SEEK_SET);
    fwrite(&reg2, sizeof(Registro), 1, punteroarchivo);
    fseek(punteroarchivo, der*sizeof(Registro), SEEK_SET);
    fwrite(&reg1, sizeof(Registro), 1, punteroarchivo);
}

char *LeerClave(FILE *punteroarchivo, long n, char *clave)
{
    Registro reg;

    fseek(punteroarchivo, n*sizeof(Registro), SEEK_SET);
    fread(&reg, sizeof(Registro), 1, punteroarchivo);
    strcpy(clave, reg.nombre);
    return clave;
}

```

6.4. Ordenación por mezcla directa

Consiste en realizar sucesivas particiones y fusiones de un archivo de forma que se produzcan secuencias ordenadas de longitud cada vez mayor del siguiente modo:

1. Se establece como longitud inicial para la secuencia el valor uno, $N = 1$.
2. Se efectúa la partición del archivo sobre otros dos copiando alternativamente en cada uno de ellos secuencias de longitud N .
3. Se mezclan los N registros de cada secuencia, de tal forma que se obtenga una secuencia ordenada de longitud $2 * N$. Las sucesivas mezclas de las particiones sobrescriben el archivo original.
4. La longitud de la secuencia, N , se multiplica por dos y se vuelve al segundo paso hasta que la longitud de la secuencia para la partición sea mayor o igual que el número de elementos del archivo original.

6.5. Ordenación por mezcla natural

Intenta aprovechar la posible ordenación interna de las secuencias del archivo, para lo cual: efectúa la partición del archivo desordenado sobre otros dos utilizando métodos de selección por sustitución o selección natural.

Selección por sustitución:

Este método divide el archivo inicial en otros dos con largas secuencias ordenadas. La forma de obtener estas secuencias es:

1. Leer N registros del archivo desordenado, almacenándolos en un *array* y poniéndoles a todos una marca que denominaremos de no congelados.
2. Obtener el registro, $a[m]$, con clave más pequeña de entre los no congelados y escribirlo en la partición.
3. Leer el siguiente registro, r , del archivo de entrada y almacenarlo en $a[m]$. Si la clave de r es más pequeña que la del último registro escrito marcar $a[m]$ como congelado.
4. Cuando todos los registros del *array* estén congelados o se haya alcanzado el fin de fichero, comenzará una nueva partición y deberemos descongelar los registros y cambiar el archivo destino.
5. Si no se alcanzó el fin del archivo inicial se repiten las acciones desde el paso 2. Si se llegó al fin del archivo de entrada, hasta que se hayan escrito todos los registros, se ejecutan también esas mismas acciones, pero sin leer y marcando todos los registros que se escriban como congelados.

Selección natural:

Este método evita tener que almacenar registros en un *array* y consiste en ir tomando secuencias ordenadas de la máxima longitud que copiaremos alternativamente sobre dos archivos.

1. Realiza sucesivas fusiones de las secuencias ordenadas en estos archivos, sobrescribiendo con ellas el archivo inicial. La fusión ha de realizarse bajo el criterio de que en cada una de las secuencias resultantes exista ordenación.
2. Los pasos 1º y 2º se repetirán hasta que el archivo esté ordenado.

6.6. Método de la mezcla equilibrada múltiple

La mezcla equilibrada múltiple utiliza m archivos auxiliares, de los que $m/2$ son de entrada y $m/2$ de salida y, para representarlos, se utilizará el tipo *array* de archivos de forma que puedan ser referenciados mediante un subíndice. El método puede ser expresado así:

1. Efectuar la partición del archivo inicial desordenado sobre los $m/2$ primeros archivos auxiliares. Se podrán utilizar los métodos ya vistos de selección por sustitución o selección natural modificados para que escriban las particiones, en lugar de sobre dos, sobre $m/2$ archivos. Estos archivos, se considerarán archivos de entrada.
2. Tomar un tramo creciente de cada uno de los $m/2$ ficheros de entrada que no estén vacíos y escribir el resultado de la fusión en el primer archivo de salida. Repetir la operación de fusión por tramos hasta que se escriban los registros de

todos los archivos de entrada, colocando los resultados de cada fusión consecutivamente en los distintos archivos de salida. Cuando se alcance el último archivo de salida se volverá a empezar por el primero.

En la fusión por tramos hay que considerar que, al mezclar el correspondiente tramo de cada uno de los ficheros de entrada, no se llega al fin del mismo en todos los archivos al mismo tiempo. Un tramo termina cuando se alcanza el fin de archivo o bien la clave siguiente es menor que la actual, la operación de mezcla sobre un determinado fichero de salida continuará, ignorando ese fichero, hasta el fin de tramo en todos los archivos de entrada.

3. Cambiar la finalidad de los archivos, los de entrada pasan a ser de salida y viceversa; repetir, a partir de 2, hasta que quede un único tramo, momento en el que los datos iniciales estarán ordenados.

Una optimización del método consistiría en modificar los procesos de partición y mezcla de forma que al originarse una nueva secuencia, en lugar de ser destinada siempre al siguiente archivo, se considerara la posibilidad de situarla en otro. La nueva secuencia se colocaría en el archivo siguiente, según la norma habitual, sólo cuando se comprobara que no era capaz de convertirse en una prolongación de la última secuencia de algún otro.

6.7. Método polifásico

En el método polifásico las secuencias ordenadas existentes en el archivo inicial se reparten en $n - 1$ archivos de entrada colocando en cada uno de ellos un número de secuencias o tramos específico, el archivo restante (vacío) se considera de salida. La mezcla se efectúa por tramos ordenados desde los $n - 1$ archivos de entrada al archivo de salida. En el momento en el que se acaban los tramos o secuencias (reales y ficticias) colocadas en uno de los archivos de entrada hay un cambio de cometido, este archivo pasa a ser el de salida y el que anteriormente era de salida pasa a ser de entrada uniéndose a los que ya lo eran anteriormente. La mezcla de tramos continúa hasta la obtención del archivo ordenado.

Por ejemplo, si se quiere aplicar este método para ordenar un archivo inicial (considérese que tiene 49 secuencias ordenadas o tramos) utilizando 5 archivos auxiliares, el número de secuencias a colocar en un primer momento en cada uno de los $n - 1 = 4$ archivos de entrada es predeterminado y, como se tratará de explicar, debería ser el siguiente ($15 + 14 + 12 + 8 = 49$).

F_1	F_2	F_3	F_4
15	14	12	8

Así, en la primera mezcla hay en el archivo F_1 15 tramos, en el F_2 14, en el F_3 12, en el F_4 8, y el F_5 es de salida y, como la mezcla de m tramos de los archivos de entrada se transformará siempre en m tramos en el archivo de salida, después de haberse realizado la primera mezcla quedan en los archivos los siguientes tramos, en F_1 7 ($15 - 8$), en F_2 6 ($14 - 8$), en F_3 4 ($12 - 8$), en F_4 0 (es el de salida para la siguiente mezcla), y en F_5 8 ($0 + 8$). La segunda mezcla toma a F_1 , F_2 , F_3 y F_5 como archivos de entrada y F_4 de salida y, tras su ejecución, deja los siguientes tramos en los archivos F_1 3 ($7 - 4$), en F_2 2 ($6 - 4$), en F_3 0 ($4 - 4$, es el de salida para la siguiente mezcla), en F_4 4 ($0 + 4$), y en F_5 4 ($8 - 4$). El proceso continúa hasta que al final el archivo F_1 es de salida y el resto de archivos son de entrada conteniendo un sólo tramo ordenado, lo que permite que la ordenación final quede en el archivo F_1 .

F_1	F_2	F_3	F_4	F_5
15	14	12	8	0
7	6	4	0	8
3	2	0	4	4
1	0	2	2	2
0	1	1	1	1
1	0	0	0	0

Es decir, en la última mezcla resulta obligado que ningún archivo de entrada tenga más de una secuencia ordenada. Por el mismo razonamiento, en la mezcla anterior, el número de tramos ordenados que deben tener los archivos de entrada (en el peor de los casos) debe ser de dos, en todos excepto en uno de ellos, que debe tener una sola secuencia ordenada. Continuando estas deducciones y calculando ascendientemente, es decir en orden inverso a como se efectuaron las mezclas se ve que el número máximo de secuencias de los archivos de entrada debe ser fijo y puede obtenerse teniendo en cuenta los tramos generados en los sucesivos archivos de salida.

F ₁	F ₂	F ₃	F ₄	F ₅
1	0	0	0	0
0	1	1	1	1
0+1	0	1+1	1+1	1+1
0+1+2	0+2	0	1+1+2	1+1+2
0+1+2+4	0+2+4	0+4	0	1+1+2+4
0+1+2+4+8	0+2+4+8	0+4+8	0+8	0

Continuando con el esquema expresado se observa que el método tiene la dificultad de que el número de tramos en cada archivo ha de ser específico (secuencias de Fibonacci de orden $n-1$) y estará en función de las secuencias existentes en el archivo inicial y del número de archivos que se usen. Como normalmente el número de tramos del archivo a ordenar no coincide con el requerido por el método, se deben simular como vacíos los tramos necesarios para que exista coincidencia. Para obtener la distribución inicial de los tramos en los archivos se calculan los números de Fibonacci de cada nivel en forma progresiva hasta alcanzar aquel en el que se agote el archivo de entrada. La distribución inicial debe repartir los tramos ficticios lo mas uniformemente posible en los $n-1$ archivos. En el proceso de mezcla, la selección de un tramo ficticio de un archivo i consiste simplemente en ignorar el archivo, y por consiguiente hay que desechar dicho archivo de la mezcla del tramo correspondiente.

Las principales diferencias de la ordenación polifásica respecto a la mezcla equilibrada múltiple son:

- En cada pasada hay un sólo archivo destino (salida), en vez de los $n/2$ que necesita la mezcla equilibrada múltiple.
- La finalidad de los archivos (entrada, salida) cambia en cada pasada rotativamente. Esto se controla mediante una tabla de correspondencia de índices de archivo. En la mezcla múltiple el archivo de salida no es rotativo, siempre se intercambian $n/2$ archivos origen (entrada) por $n/2$ archivos destino (salida).
- Puede ocurrir que se alcance el fin de un archivo de entrada y sin embargo ser precisas más mezclas que utilicen tramos ficticios de ese archivo. El criterio de terminación de una fase radica en el número de tramos a ser mezclados de cada 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; ahora, partiendo de los números del último nivel pueden recalcularse hacia atrás para conocer el número de tramos que van a ser mezclados antes de cambiar el archivo de salida.

PROBLEMAS DE SEGUIMIENTO

- 6.1. Explicar paso a paso el proceso de ordenación por mezcla directa de un archivo cuyos registros almacenan la siguiente información en el campo clave.

F:

3	31	14	42	10	15	8	13	63	18	50
---	----	----	----	----	----	---	----	----	----	----

El número de archivos auxiliares que se utilizan en este proceso de ordenación debe ser dos.

Solución

Este método de ordenación se basa en realizar una partición en secuencias con longitud fija de los registros y mezclar luego los registros en esa secuencia. El pseudocódigo que representa el método es:

```

procedimiento ordmezcladirecta(E cadena: nombreach)
var
    entero: long, lgtud
inicio
    // calcularlong(nombreach) es una función definida por el usuario que
    // devuelve el número de registros del archivo original
    long ← calcularlong(nombreach)
    lgtud ← 1
    mientras lgtud < long hacer

```

```

partir(nombreach, 'archaux1', 'archaux2', lgtud)
mezclar('archaux1', 'archaux2', nombreach, lgtud)
lgtud ← lgtud*2
fin_mientras
borrar('archaux1')
borrar('archaux2')
fin_procedimiento

```

Partición en secuencias de longitud 1

F1

3	14	10	8	63	50
---	----	----	---	----	----

 F2

31	42	15	13	18
----	----	----	----	----

Fusión de secuencias de longitud 1

F

3	31	14	42	10	15	8	13	18	63	50
---	----	----	----	----	----	---	----	----	----	----

Partición en secuencias de longitud 2

F1

3	14	10	15	18	63
---	----	----	----	----	----

 F2

14	42	8	13	50
----	----	---	----	----

Fusión de secuencias de longitud 2

F

3	14	31	42	8	10	13	15	18	50	63
---	----	----	----	---	----	----	----	----	----	----

Partición en secuencias de longitud 4

F1

3	14	31	42	18	50	63
---	----	----	----	----	----	----

 F2

8	10	13	15
---	----	----	----

Fusión de secuencias de longitud 4

F

3	8	10	13	14	15	31	42	18	50	63
---	---	----	----	----	----	----	----	----	----	----

Partición en secuencias de longitud 8

F1

3	8	10	13	14	15	31	42
---	---	----	----	----	----	----	----

 F2

18	50	63
----	----	----

Fusión de secuencias de longitud 8

F

3	8	10	13	14	15	18	31	42	50	63
---	---	----	----	----	----	----	----	----	----	----

El proceso termina cuando la longitud de las secuencias es mayor o igual que el número de registros del archivo original.

- 6.2.** *Describa paso a paso el proceso de ordenación del mismo archivo considerado en el problema 6.1 cuyos registros almacenan la siguiente información en el campo clave.*

F

3	31	14	42	10	15	8	13	63	18	50
---	----	----	----	----	----	---	----	----	----	----

por el método de mezcla natural. Utilice también únicamente dos archivos auxiliares.

Solución

El pseudocódigo correspondiente es:

```

procedimiento ordmezclanatural(E cadena: nombreach)
var
    entero: numerodesecuencias
    logico: ordenado
inicio
    ordenado ← falso
    mientras no ordenado hacer
        partir(nombreach, 'archaux1', 'archaux2')
        numsec ← 0
        mezclar(nombreach, 'archaux1', 'archaux2', numerodesecuencias)
        si numerodesecuencias <= 1 entonces
            ordenado ← verdad
        fin_si
    fin_mientras
    borrar('archaux1')
    borrar('archaux2')
fin_procedimiento

```

El método consiste en aprovechar secuencias ordenadas que pudieran existir inicialmente en el archivo, obteniendo con ellas particiones ordenadas de longitud variable sobre dos archivos auxiliares. A partir de estos ficheros auxiliares se construye un nuevo fichero, mezclando los segmentos crecientes maximales de cada uno de ellos. El proceso se repetirá desde el principio hasta conseguir la ordenación completa del archivo. Para el archivo F se observa que las secuencias ordenadas que existen inicialmente son:

F:

3	31
---	----

14	42
----	----

10	15
----	----

8	13	63
---	----	----

18	50
----	----

y el proceso constaría de los siguientes pasos:

F1

3	31
---	----

10	15	18	50
----	----	----	----

 F2

14	42
----	----

8	13	63
---	----	----

F:

3	14	31	42
---	----	----	----

8	10	13	15	18	50	63
---	----	----	----	----	----	----

F1

3	14	31	42
---	----	----	----

 F2

8	10	13	15	18	50	63
---	----	----	----	----	----	----

F:

3	8	10	13	14	15	31	42	18	50	63
---	---	----	----	----	----	----	----	----	----	----

6.3. *Explicar paso a paso el proceso de ordenación por mezcla polifásica de un archivo cuyos registros almacenan la siguiente información en el campo clave*

F:

3	30	14	42	10	15	8	13	63	18	50	12	34	4	46	54	20	31	52
---	----	----	----	----	----	---	----	----	----	----	----	----	---	----	----	----	----	----

El número de archivos auxiliares debe ser tres.

Solución

El algoritmo correspondiente puede expresarse en la siguiente forma:

```

procedimiento ordpolifásica(E cadena: nombreach)

inicio
    <Asignar nombre externo a los n archivos auxiliares>
    <Hasta que se agote el archivo de origen, distribuir sus secuencias entre los n-1 primeros
    archivos auxiliares efectuando cálculos progresivos del número de secuencias a colocar en

```

cada uno de ellos. Cuando se agote el archivo origen simular que las secuencias colocadas en cada archivo fueron todas las calculadas y considerar las que pudieran faltar como secuencias ficticias>

repetir

<Mezclar sobre el archivo de salida tantas secuencias de los archivos de entrada como secuencias teóricas, reales o ficticias, tiene el archivo de entrada que precede a dicho archivo de salida>

<Rotar la finalidad de los archivos de forma que:

(a) El último de entrada, el precedente al de salida que se ha quedado sin secuencias, pasa a ser el de salida.

(b) El anterior archivo de salida pasa a ser considerado como archivo de entrada. El proceso se considera circular y, por tanto, al primer archivo auxiliar le precede el último>

hasta que <no haya secuencias para mezclar>

fin_procedimiento

Para determinar el número de secuencias a colocar en cada archivo el cálculo a efectuar es:

<u>NIVEL</u>	<u>F1</u>	<u>F2</u>	<u>F3</u>	<u>Total secuencias colocadas</u>
1	1	1	0	2
2	2	1	0	3
3	3	2	0	5
4	5	3	0	8

Los cálculos se efectúan hasta que se agota el archivo original. Como el archivo inicial tiene 8 secuencias crecientes ordenadas se colocan 5 en el archivo F1 y 3 en el archivo F2 y el archivo F3 se considera de salida.

F1

3	30	10	15	8	13	63	12	34	20	31	52
---	----	----	----	---	----	----	----	----	----	----	----

F2

14	42	18	50	4	46	54
----	----	----	----	---	----	----

F3

Se mezclan 3 secuencias de F1 y F2 y se escriben en F3

F1

12	34	20	31	52
----	----	----	----	----

F2

F3

3	14	30	42	10	15	18	50	4	8	13	46	54	63
---	----	----	----	----	----	----	----	---	---	----	----	----	----

El archivo F2 es ahora el de salida y se mezclan dos secuencias de F3 y F1 y se escriben en F2

F1

F2

3	12	14	30	34	42	10	15	18	20	31	50	52
---	----	----	----	----	----	----	----	----	----	----	----	----

F3

4	8	13	46	54	63
---	---	----	----	----	----

El archivo F1 es ahora el de salida y se mezcla una secuencia de F3 con otra de F2 y se escriben en F1

F1

3	4	8	12	13	14	30	34	42	46	54	63
---	---	---	----	----	----	----	----	----	----	----	----

F2

10	15	18	20	31	50	52
----	----	----	----	----	----	----

F3

El archivo de salida vuelve a ser F3 y se mezcla la secuencia existente en F1 con la que hay en F2.

F1

F2

F3

3	4	8	12	13	14	15	18	20	30	31	42	46	50	52	54	63
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

PROBLEMAS BÁSICOS

6.4. Realizar un pseudocódigo que describa el proceso de ordenación por mezcla equilibrada múltiple.

Solución

```

procedimiento OrdenaciónMezclaEquilibrada(E cadena: nombreach)

    inicio
        /* asignar a todos los archivos auxiliares, representados en el array a, un nombre
           de archivo en disco
        */
        asignartodos(a)
        primeroentrada ← 1
        primerosalida ← totalentrada + 1
        numsecuenciasdistribuidas ← 0
        partir(nombreach, a, primeroentrada, numsecuenciasdistribuidas)
        /* parte el fichero inicial entre los de entrada y devuelve el número de secuencias
           distribuidas */
        mientras numsecuenciasdistribuidas > 1 hacer
            numsecuenciasdistribuidas ← 0

            /* mezcla los archivos de entrada, elementos del array a que van desde el
               primeroentrada hasta primeroentrada + totalentrada - 1, en los archivos
               de salida, que también son elementos del array a, pero desde primerosalida
               a primerosalida + totalentrada - 1. Devuelve el número de secuencias
               distribuidas */

            mezclar(a, primeroentrada, primerosalida, numsecuenciasdistribuidas)
            intercambiar(primeroentrada, primerosalida)
        fin_mientras
        <Borrar el archivo inicial>
        <Renombrar el archivo auxiliar a[primeroentrada] como el inicial, puesto que al
           salir del bucle mientras se intercambió el primerosalida con el primeroentrada>
        <Borrar los archivos auxiliares>
    fin_procedimiento

```

- 6.5.** Diseñar un programa efectúe la creación de un archivo inventario de nombre *Stock.dat*. El archivo será binario con los registros colocados en él de forma secuencial. Cada registro tiene los campos nombre, código, cantidad, precio y fabricante. El programa permitirá también la consulta de un artículo por el número de código.

Análisis

La apertura del archivo deberá hacerse en el modo *w+* que crea el archivo y permite tanto operaciones de escritura como de lectura en el mismo. La consulta efectuará una búsqueda secuencial en un archivo de datos desordenado, que terminará cuando se encuentre el dato buscado o se llegue al final del archivo. Se utiliza *fseek* para que la operación de búsqueda comience siempre en el primer registro.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define True 1
#define False 0

typedef struct
{
    char Nombre[30];
    char Fabricante[30];
    charCodigo[6];
    int Precio;
    int Cantidad;
} Regarticulo;

void Pausa()
{
    char ch;

    printf("\nPulse return para continuar\n");
    while ((ch = getchar()) != '\n');
}

void Creararticulo(Regarticulo * Articulo)
{
    printf("Introduzca nombre: "); gets(Articulo->Nombre);
    printf("Introduzca fabricante: "); gets(Articulo->Fabricante);
    printf("Introduzca codigo: "); gets(Articulo->Codigo);
    printf("Introduzca precio (numero entero): ");
    scanf ("%d", &Articulo->Precio);
    printf("Introduzca cantidad: "); scanf ("%d", &Articulo->Cantidad);
}

void Mostrararticulo(Regarticulo Articulo)
{
    printf("nombre: %s\n", Articulo.Nombre);
    printf("fabricante: %s\n", Articulo.Fabricante);
    printf("codigo: %s\n", Articulo.Codigo);
    printf("precio: %d\n", Articulo.Precio);
    printf("cantidad: %d\n", Articulo.Cantidad);
}
```

```
void Consultar(FILE * F)
{
    int Hallado;
    Regarticulo Articulo;
    charCodigo[6];

    printf("\nIntroduzca codigo articulo a buscar: ");
    gets(Codigo);
    Hallado = False;
    fseek( F,0L, SEEK_SET);
    while( !feof(F) && !Hallado)
    {
        fread(&Articulo, sizeof(Articulo),1, F);
        Hallado = (strcmp(Codigo, Articulo.Codigo)==0) ;
    }
    if (Hallado)
        Mostrararticulo(Articulo);
    else
        printf("No existe el codigo "" %s "" en el almacén", Codigo);
    Pausa();
}

void Generarfichero(FILE * F)
{
    char Masdatos;
    Regarticulo Articulo;
    do
    {
        Creararticulo(&Articulo);
        fwrite(&Articulo, sizeof(Regarticulo),1, F);
        printf("\nMas articulos: (S/N) \n");
        while ((Masdatos = getchar()) == '\n');
        while (getchar() != '\n');
    }while (Masdatos != 'N' && Masdatos != 'n');
}

int main ()
{
    FILE* Stock;
    char Masconsultas;

    Stock = fopen("stock.dat","wb+");
    if (Stock == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }
    Generarfichero(Stock);
    do
    {
        Consultar(Stock);
        printf("\nMas consultas: (S/N) \n");
        while ((Masconsultas = getchar()) == '\n');
```



```

    while (getchar() != '\n');
}while (Masconsultas != 'N' && Masconsultas != 'n');
fclose(Stock);
printf("FIN");
Pausa();
return 0;
}

```

- 6.6.** *Supuesto un archivo binario en el que la información ha sido almacenada de forma secuencial y cuyos registros tienen los campos: código (de tipo entero), cantidad y descuento. Escribir un algoritmo que ordene los registros de modo ascendente por el código, teniendo en cuenta que dicho archivo es tan grande que no cabe en memoria para su clasificación.*

Análisis

Puesto que es binario y la longitud de sus registros fija, el archivo admite posicionamiento con `fseek` y para ordenarlo, pueden aplicarse métodos similares a los de ordenación interna. El método aplicado en esta ocasión es una versión del método de *Shell*. Las modificaciones por tratarse de un archivo en lugar de un *array* (o *arreglo*) son las debidas a las operaciones de lectura y escritura de registros. Se incluye una función `Crear` que genera el archivo si éste no existiera. Y otro, `Mostrar`, que se llama al final para verificar que el archivo se encuentra ordenado.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define True 1
#define False 0

typedef struct
{
    int cod;
    int cant;
    int dto;
}reg;

void Crear(FILE * f);
void Mostrar (FILE * f);

int main ()
{
    FILE* f;
    long n, salto, j;
    reg auxil, auxi2;
    int ordenado;

    f = fopen("Fdd.dat","rb+");
    if (f == NULL)
    {
        f = fopen("Fdd.dat","wb+");
        if (f == NULL)
        {
            puts("\nError en la operación de abrir archivo.");
            exit(-1);
        }
    }
}

```

```

    }
    Crear(f);
}
fseek(f, 0L, SEEK_END);
n = ftell(f)/sizeof(reg);
salto = n;
while (salto > 1)
{
    salto = salto / 2;
    do
    {
        ordenado = True;
        for (j = 1; j <= n - salto; j++)
        {
            fseek(f, (j - 1) * sizeof(reg), SEEK_SET);
            fread(&auxil, sizeof(reg), 1, f);
            fseek(f, (j + salto - 1) * sizeof(reg), SEEK_SET);
            fread(&auxi2, sizeof(reg), 1, f);
            if (auxil.cod > auxi2.cod)
            {
                fseek(f, (j - 1)* sizeof(reg), SEEK_SET);
                fwrite(&auxi2, sizeof(reg), 1, f);
                fseek(f, (j + salto - 1) * sizeof(reg), SEEK_SET);
                fwrite(&auxil, sizeof(reg), 1, f);
                ordenado = False;
            }
        }
    } while (!ordenado);
}
Mostrar (f);
getchar();
return 0;
fclose(f);
}

/* Los datos a introducir en el archivo se generan de forma aleatoria.
   Es decir que los códigos cantidades y descuentos no se piden al usuario y
   van a ser números aleatorios en un determinado rango. Los descuentos solo
   podrán tomar los valores 0, 5 o 10.
*/

void Crear(FILE * f)
{
    int i, cuantos;
    reg r;

    /* srand() inicializa el generador de números aleatorios al
       valor devuelto por time() */
    /* time() devuelve el tiempo transcurrido en segundos desde el
       1 de enero de 1970 */
    srand((unsigned) time(NULL));
    cuantos = rand() %20 + 1;
    for (i = 1; i <= cuantos; i++)

```

```
{
    r.cod = rand() % 30 + 1;
    r.cant = rand() % 100 + 1;
    r.dto = rand() % 3 * 5;
    fwrite(&r, sizeof(reg), 1, f);
}

void Mostrar (FILE * f)
{
    reg r;

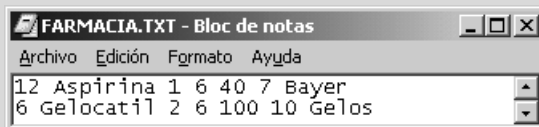
    printf("Cod. Cant. Dto.\n");
    fseek(f, 0L, SEEK_SET);
    if (! feof(f))
        fread(&r, sizeof(reg), 1, f);
    while (! feof(f))
    {
        printf(" %2d   %3d   %2d \n", r.cod, r.cant, r.dto);
        fread(&r, sizeof(reg), 1, f);
    }
}
```

Ejecución

Cod.	Cant.	Dto.
1	3	0
3	89	10
5	17	10
10	66	10
13	47	0
17	16	10
22	11	5
23	28	0
24	43	0

PROBLEMAS AVANZADOS

- 6.7. Una farmacia mantiene su stock de medicamentos en un archivo de texto, donde almacena el código, nombre, precio, IVA, stock, stock máximo, stock mínimo y proveedor de cada uno de sus productos. La estructura del archivo es la mostrada en la figura.



12	Aspirina	1	6	40	7	Bayer
6	Gelocatil	2	6	100	10	Gelos

Considerar como campo clave el código del producto y escribir un programa que ordene el archivo por el método de mezcla equilibrada múltiple.

Análisis

El problema se resuelve mediante la implementación del método de ordenación por mezcla equilibrada múltiple, explicado en la pregunta teórica correspondiente. El enunciado indica que el archivo es de texto para que el método se desarrolle bajo una estricta consideración de las características de los archivos secuenciales.

Al abrir el archivo “FARMACIA.TXT” tras la ejecución del programa aparecerán los registros ordenados por el campo código. Las funciones fundamentales son:

- `partir()`. Reparte las secuencias crecientes que encuentra en el fichero inicial entre los de archivos de entrada y devuelve el número de secuencias distribuidas.
- `mezclar()`. Mezcla los archivos de entrada, cuyos nombres se encuentran en el `array auxi` desde el `primeroentrada` hasta `primeroentrada + totalentrada - 1`, en los archivos de salida, cuyos nombres se encuentran en el `array auxi`, desde `primerosalida` a `primerosalida + totalentrada - 1`. Este tratamiento permite cambiar los archivos de entrada por los de salida y viceversa con facilidad, simplemente intercambiando los valores de las variables numéricas `primeroentrada` y `primerosalida`. La mezcla se efectúa por tramos de secuencias crecientes máximas y el método devuelve el número de secuencias distribuidas.
- `asignar()`. Asigna nombre a los archivos auxiliares de entrada y salida de forma automática. El número de archivos auxiliares utilizados puede modificarse simplemente cambiando el valor de las constantes `total` y `totalentrada`.
- `abrirescritura()`. Abre para escritura los archivos cuyos nombres se encuentran en el `array auxi` desde `primeroentrada` hasta `primeroentrada + totalentrada - 1`, y les adjudica como punteros los elementos del `array a` en el mismo rango.
- `abrirlectura()`. Se comporta de forma similar a `abrirescritura` aunque naturalmente no abre los mismos archivos y utiliza el modo opuesto.

En el proceso de mezcla, cuando la información leída de uno de los archivos es menor que la ya escrita en el destino y, por tanto, no puede formar parte de la actual secuencia creciente se dice que se “congela”. La función `descongelar()` quita las marcas de congelado puestas a estos registros cuando comienza a escribirse una nueva secuencia para que dichos registros puedan incorporarse a la misma.

Codificación

```
#include <stdio.h>
#include <string.h>
#define True 1
#define False 0
#define longitud 81
#define totalentrada 3
#define total 7
```

```

typedef struct
{
    int cod;
    char nombre[10];
    int precio, iva, stockmax, stockmin;
    char proveedor[10];
}datos;

typedef struct
{
    datos info;
    char marca;
}reg;

void asignar(char auxi[total][longitud]);
void partir(char nombre[], char auxi[][longitud], int primeroentrada,
                                                    int * numsecdistr);
void mezclar(char auxi[][longitud], int primeroentrada, int primerosalida,
                                                    int* numsecdistr);
void abrirescritura(char auxi[][longitud], int primero, FILE * a[]);
void abrirlectura(char auxi[][longitud], int primero, FILE * a[]);
void cerrar(FILE * a[], int primero);
int menor(reg r[], int primeroentrada);
void descongelar(reg r[], int primeroentrada);
void intercambiar(int * primeroentrada, int * primerosalida);
void borrar (char auxi[][longitud], int primeroentrada);

int main ()
{
    char nombre[longitud];
    char auxi[total][longitud], pausa;
    int primeroentrada, primerosalida, numsecdistr;

    asignar(auxi);
    primeroentrada = 1;
    primerosalida = totalentrada + 1;
    numsecdistr = 0;
    strcpy(nombre, "farmacia.txt");
    partir(nombre, auxi, primeroentrada, &numsecdistr);
    while (numsecdistr > 1)
    {
        numsecdistr = 0;
        mezclar(auxi, primeroentrada, primerosalida, &numsecdistr);
        intercambiar(&primeroentrada, &primerosalida);
    }
    remove("farmacia.txt");
    rename(auxi[primeroentrada], "farmacia.txt");
    borrar(auxi, primeroentrada);
    printf("FIN\n");
    pausa = getc(stdin);
    return 0;
}

void asignar(char auxi[total][longitud])

```

```

{
    int i, columna;
    for (i = 1 ; i <= total; i++)
    {
        strcpy(auxi[i], "aux");
        columna = strlen(auxi[i]);
        auxi[i][columna] = '0'+i;
        auxi[i][columna+1] = '\0';
        strcat(auxi[i], ".txt");
    }
}

/* El procedimiento Partir va tomando secuencias ordenadas de la máxima
   longitud y copiándolas alternativamente sobre los archivos auxiliares
   indicados por totalentrada y devuelve el número de secuencias
   distribuidas
*/

void partir(char nombre[], char auxi[][longitud], int primeroentrada, int * numsecdistr)
{
    FILE * f;
    FILE * a[total];
    int crece;
    datos r;
    int ant, i, err;

    f = fopen(nombre,"rt");
    if (f == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }
    abrirescritura(auxi, primeroentrada, a);
    i = primeroentrada;
    err = fscanf(f, "%d%s%d%d%d%s", &r.cod, r.nombre,
                &r.precio, &r.iva, &r.stockmax,
                &r.stockmin, r.proveedor);
    while (err != EOF)
    {
        ant = r.cod;
        crece = True;
        while (crece && err != EOF)
        {
            if (ant <= r.cod)
            {
                fprintf(a[i], "%d %s %d %d %d %s\n", r.cod, r.nombre,
                    r.precio, r.iva, r.stockmax, r.stockmin,
                    r.proveedor);
                ant = r.cod;
                err = fscanf(f, "%d%s%d%d%d%s", &r.cod, r.nombre,
                    &r.precio, &r.iva, &r.stockmax,
                    &r.stockmin, r.proveedor);
            }
        }
    }
}

```

```

        else
            crece = False;
    }
    *numsecdistr = *numsecdistr + 1;
    i = i + 1;
    if (i > totalentrada)
        i = 1;
    }
    cerrar(a, primeroentrada);
    fclose(f);
}

void abrirescritura(char auxi[][longitud], int primero, FILE * a[])
{
    int i;

    for (i = primero; i <= primero + totalentrada - 1; i++)
    {
        a[i] = fopen(auxi[i], "wt");
        if (a[i] == NULL)
        {
            puts("\nError en la operación de abrir archivo.");
            exit(-1);
        }
    }
}

/* mezcla los archivos de entrada desde primerodeentrada hasta
   (primerodeentrada + totalentrada - 1) en los archivos de salida que van
   desde primerodesalida a (primerodesalida + totalentrada - 1) y devuelve
   el número de secuencias distribuidas
*/

void mezclar(char auxi[][longitud], int primeroentrada, int primerosalida,
              int* numsecdistr)
{
    reg r[total];
    /* r es un array de registros paralelo al de archivos donde cada
       registro lleva un campo donde marcar los registros que han de
       ser ignorados cuando se busque el de menor código para escribirlo
       en el destino y la causa, que puede ser: falta de nuevo valor para el registro por
       haberse alcanzado el fin de archivo, o elemento que ya no puede pertenecer a la
       actual secuencia pues es mayor que otro que ya ha sido escrito.
    */

    int ant, i, j, hanterminado, numcongelados, err;
    FILE * a[total];

    abrirlectura(auxi, primeroentrada, a);
    abrirescritura(auxi, primerosalida, a);
    j = primerosalida; hanterminado = 0;
    for (i = primeroentrada; i <= primeroentrada + totalentrada - 1; i++)

```

```

{
    err = fscanf(a[i], "%d%s%d%d%d%s", &r[i].info.cod, r[i].info.nombre,
        &r[i].info.precio, &r[i].info.iva, &r[i].info.stockmax,
        &r[i].info.stockmin, r[i].info.proveedor);

    r[i].marca = ' ';
    if (err == EOF)
    {
        hanterminado = hanterminado + 1;
        r[i].marca = 'f';
    }
}
while (hanterminado < totalentrada)
{
    numcongelados = hanterminado;
    while (numcongelados < totalentrada)
    {
        i = menor(r, primeroentrada);
        fprintf(a[j], "%d %s %d %d %d %d %s\n", r[i].info.cod,
            r[i].info.nombre, r[i].info.precio,
            r[i].info.iva, r[i].info.stockmax,
            r[i].info.stockmin, r[i].info.proveedor);
        ant = r[i].info.cod;
        err = fscanf(a[i], "%d%s%d%d%d%s", &r[i].info.cod, r[i].info.nombre,
            &r[i].info.precio, &r[i].info.iva, &r[i].info.stockmax,
            &r[i].info.stockmin, r[i].info.proveedor);
        if (err == EOF)
        {
            hanterminado = hanterminado + 1;
            r[i].marca = 'f';
            numcongelados = numcongelados + 1;
        }
        else
        {
            if (r[i].info.cod < ant)
            {
                r[i].marca = 's';
                numcongelados = numcongelados + 1;
            }
            else
            {
                r[i].marca = ' ';
            }
        }
        *numsecdistr = *numsecdistr + 1;
        j = j + 1;
        if (j > primerosalida + totalentrada - 1)
            j = primerosalida;
        descongelar(r, primeroentrada);
    }
    cerrar(a, primeroentrada);
    cerrar(a, primerosalida);
}

int menor(reg r[], int primeroentrada)
{

```



```

    int j, min;

    min = primeroentrada;
    while (r[min].marca != ' ')
        min = min + 1;
    for (j = min + 1; j <= primeroentrada + totalentrada - 1; j++)
        if (r[j].marca == ' ')
            if (r[j].info.cod < r[min].info.cod)
                min = j;
    return min;
}

void abrirlectura(char auxi[][longitud], int primero, FILE * a[])
{
    int i;

    for (i = primero; i <= primero + totalentrada - 1; i++)
    {
        a[i] = fopen(auxi[i], "rt");
        if (a[i] == NULL)
        {
            puts("\nError en la operación de abrir archivo.");
            exit(-1);
        }
    }
}

void cerrar(FILE * a[], int primero)
{
    int i;

    for (i = primero; i <= primero + totalentrada - 1; i++)
        fclose(a[i]);
}

/* quita la marca que indica que el registro no puede formar parte de
   la secuencia actual
*/
void descongelar(reg r[], int primeroentrada)
{
    int i;

    for (i = primeroentrada; i <= primeroentrada + totalentrada - 1; i++)
        if (r[i].marca == 's')
            r[i].marca = ' ';
}

void intercambiar(int * primeroentrada, int * primerosalida)
{
    int auxi;

    auxi = *primeroentrada;
    *primeroentrada = *primerosalida;

```

```

    *primerosalida = auxi;
}

/* borra los archivos auxiliares excepto el primero de entrada, ya que
   en el primero de entrada se encuentra el resultado final
*/
void borrar (char auxi[][longitud], int primeroentrada)
{
    int i;

    for (i = 1; i <= total; i++)
        if (i != primeroentrada)
            remove(auxi[i]);
}

```

- 6.8.** *Implementar el método de ordenación por mezcla directa con dos archivos auxiliares de un archivo binario que contiene, la siguiente información: código del producto (de tipo entero), cantidad y descuento.*

Análisis

El archivo tiene la misma estructura que el del problema 6.5 por ello las declaraciones, etc. son análogas. La función `OrdenarPorMezclaDirecta()` muestra claramente las distintas fases del método de ordenación tal y como fueron explicadas en la parte teórica correspondiente. La longitud o número de registros del archivo se obtiene mediante `CalcularLongitud()`

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define True 1
#define False 0

typedef struct
{
    int cod;
    int cant;
    int dto;
}reg;

void Crear (char nombre[]);
void Mostrar (char nombre[]);
void OrdenarPorMezclaDirecta(char nombre[]);
void Partir (char nombre[], char nombre1[], char nombre2[], long longsec);
void Mezclar (char nombre1[], char nombre2[], char nombre[], long longsec);
void CalcularLongitud (char nombre[], long * longitud);

int main ()
{
    Crear("Fdt.dat");
    OrdenarPorMezclaDirecta("Fdt.dat");
    Mostrar ("Fdt.dat");
}

```

```

    getchar();
    return 0;
}

void OrdenarPorMezclaDirecta(char nombre[])
{
    long longitud, longsec;

    CalcularLongitud(nombre, &longitud);
    longsec = 1;
    while (longsec <= longitud)
    {
        Partir(nombre, "aux1.dat", "aux2.dat", longsec);
        Mezclar("aux1.dat", "aux2.dat", nombre, longsec);
        longsec = longsec * 2;
    }
    //se borran los archivos auxiliares
    remove("aux1.dat");
    remove("aux2.dat");
}

//Partir() parte en secuencias de una determinada longitud

void Partir (char nombre[], char nombre1[], char nombre2[], long longsec)
{
    long i;
    int nerr, sw;
    FILE * f, *f1, *f2;
    reg auxi;

    if ((f = fopen(nombre,"rb")) == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }
    if ((f1 = fopen(nombre1,"wb") )== NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }
    if ((f2 = fopen(nombre2,"wb")) == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }
    i = 0;
    sw = True;
    nerr = fread(&auxi, sizeof(reg),1, f);
    while (nerr)
    {
        if (sw)
            fwrite(&auxi, sizeof(reg),1, f1);
        else

```

```
fwrite(&auxi, sizeof(reg),1, f2);
i = i + 1;
if (i == longsec)
{
    sw = !sw;
    i = 0;
}
nerr = fread(&auxi, sizeof(reg),1, f);
}
fclose(f);
fclose(f1);
fclose(f2);
}

//Mezclar, mezcla secuencias de una determinada longitud

void Mezclar (char nombre1[], char nombre2[], char nombre[], long longsec)
{
    FILE *f, * f1, * f2;
    int i, j, fin1, fin2;
    reg auxil, auxi2;

    if ((f = fopen(nombre,"wb")) == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }
    if ((f1 = fopen(nombre1,"rb")) == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }
    if ((f2 = fopen(nombre2,"rb")) == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }

    i = 0; j = 0; fin1 = False; fin2 = False;
    // fread devuelve 0 cuando se alcanza el fin de archivo y no puede leer
    fin1 = !fread(&auxil, sizeof(reg),1, f1);
    fin2 = !fread(&auxi2, sizeof(reg),1, f2);
    while (!fin1 || !fin2)
    {
        while (!fin1 && !fin2 && (i < longsec) && (j < longsec))
        {
            if (auxil.cod < auxi2.cod)
            {
                fwrite(&auxil, sizeof(reg),1, f);
                fin1 = !fread(&auxil, sizeof(reg),1, f1);
                i = i + 1;
            }
            else
```

```

        {
            fwrite(&auxi2, sizeof(reg),1, f);
            fin2 = !fread(&auxi2, sizeof(reg),1, f2);
            j = j + 1;
        }
    }
    while (!fin1 && (i < longsec))
    {
        fwrite(&auxi1, sizeof(reg),1, f);
        fin1 = !fread(&auxi1, sizeof(reg),1, f1);
        i = i + 1;
    }
    while (!fin2 && (j < longsec))
    {
        fwrite(&auxi2, sizeof(reg),1, f);
        fin2 = !fread(&auxi2, sizeof(reg),1, f2);
        j = j + 1;
    }
    i = 0; j = 0;
    } // fin del mientras no fin1 o no fin2
fclose(f);
fclose(f1);
fclose(f2);
}

void CalcularLongitud (char nombre[], long * longitud)
{
    FILE *f;

    if ((f = fopen(nombre,"rb")) == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }
    fseek(f, 0L, SEEK_END);
    *longitud = ftell(f)/sizeof(reg);
    fclose(f);
}

/* se supone que el archivo existe y no esta ordenado;
   procedimiento Crear solo es para probar el algoritmo y
   no tendria que ser implementado
*/
void Crear (char nombre[])
{
    int i, cuantos;
    reg r;
    FILE * f;

    f = fopen(nombre,"wb");
    if (f == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }

```

```
    }
    srand((unsigned) time(NULL));
    cuantos = rand()%20+1;
    for (i = 1; i <= cuantos; i++)
    {
        r.cod = rand()%30+1;
        r.cant = rand()%100+1;
        r.dto = rand()%3* 5;
        fwrite(&r, sizeof(reg),1, f);
    }
    fclose(f);
}

void Mostrar (char nombre[])
{
    reg r;
    FILE * f;

    f = fopen(nombre,"rb");
    if (f == NULL)
    {
        puts("\nError en la operación de abrir archivo.");
        exit(-1);
    }
    printf("\nCód. Cant. Dto.\n");
    if (!feof(f))
        fread(&r, sizeof(reg),1, f);
    while (!feof(f))
    {
        printf(" %2d   %3d   %2d \n", r.cod, r.cant, r.dto);
        fread(&r, sizeof(reg),1, f);
    }
    fclose(f);
}
```

PROBLEMAS PROPUESTOS

- 6.1. Un archivo de texto consta en cada línea de literales enteros separados por, al menos, tres espacios, de los cuales uno será un signo (+, o -) y los otros blancos. Se prohíbe que el signo preceda o siga inmediatamente a un dígito. Se pide el resultado del cálculo indicado por los datos de cada una de las líneas.
- 6.2. Se tiene un archivo con los datos de los alumnos de un centro escolar *Pineda*. Cada alumno está representado por un registro con el campo clave primer apellido. A igualdad de apellido se toma como segundo campo clave el nombre del alumno. Escriba un programa para que dado tal archivo sea ordenado por el método de mezcla directa.
- 6.3. Comparar teórica y prácticamente el tiempo de ejecución de los distintos métodos de ordenación externa vistos en el capítulo.
- 6.4. Algunos alumnos del centro escolar referido en el problema 1 han experimentado modificaciones, cambios de domicilio ... En el archivo *Cambios* están registradas las modificaciones, de tal forma que un registro contiene apellido, nombre y los campos modificados. El archivo *Cambios* está ya ordenado respecto la misma clave que *Pineda*. Escriba un programa que modifique el archivo *Pineda* confrontándolo con el archivo *Cambios*. El proceso se ha de hacer en memoria externa y aprovechando el hecho de estar ordenados.
- 6.5. 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.
- 6.6. Las pruebas de acceso a la Escuela de Negocios “La Alcarria” (ENLA), 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 cada registro contenga el nombre del alumno, las puntuaciones de cada apartado y la puntuación total.
- 6.7. Escribir un programa que compare dos archivos de texto. El programa ha de mostrar las diferencias entre el primer archivo y el segundo, precedidas por el número de línea y de columna.
- 6.8. Una farmacia quiere mantener su stock de medicamentos en un archivo. De cada producto interesa guardar el código, precio y descripción. Escribir un programa que genere el archivo pedido, almacenándose los registros de manera secuencial.
- 6.9. Supóngase que se dispone del archivo ordenado de puntuaciones de la ENLA, problema 6.6 y del archivo de la Universidad de Mágina que tiene el mismo formato y también está ordenado. Escribir un programa que mezcle ordenadamente los dos archivos en un tercero.

Tipos abstractos de datos y objetos

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como `int`, `char` y `float` en C y C++, o bien `integer`, `real` o `boolean` en Pascal. Algunos lenguajes de programación 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 *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. En este capítulo se examinan el concepto de *abstracción de datos*.

7.1. El papel de la abstracción

Una abstracción es una simplificación de un proceso de la realidad en la cual sólo se consideran los aspectos más significativos o relevantes. Los mecanismos de abstracción pueden aplicarse en programación desde dos puntos de vista, dando lugar a abstracciones de datos y abstracciones funcionales.

Las abstracciones funcionales surgen al plantear de manera abstracta las operaciones del problema, permiten dotar a la aplicación de operaciones que no están directamente definidas en el lenguaje en el que se está trabajando. Se corresponden con el mecanismo de función o subprograma. Lo importante de una abstracción funcional es qué hace pero es irrelevante el cómo lo hace. Por lo tanto, lo importante son los argumentos a través de los cuales recibe información y los resultados que obtiene a partir de ellos.

Las abstracciones de datos se definen especificando los posibles valores y las operaciones que manipulan los nuevos tipos de datos; cada una de las operaciones de manipulación constituye de hecho una abstracción funcional.

7.2. El tipo abstracto de datos (TAD)

El tipo abstracto de datos es el conjunto de valores que pueden tomar los datos de ese tipo y las operaciones que los manipulan, de forma que sólo se debe acceder a los valores de los datos empleando las operaciones abstractas definidas sobre ellos. Un **TAD (Tipo Abstracto de Datos)** es un tipo definido por el usuario que:

- Tiene un conjunto de valores y un conjunto de operaciones.
- Se puede manejar sin conocer la representación interna.

7.2.1. ESPECIFICACIONES DE TIPOS ABSTRACTOS DE DATOS

La especificación describe el comportamiento del TAD, pero no su implementación. Incluye los valores y las operaciones asociadas al tipo, siendo posible realizar dos tipos de especificaciones: informal o formal. La especificación informal consiste en detallar el tipo de datos en lenguaje natural:

```
TAD Nombre del tipo de datos (VALORES: valores que pueden tomar los datos del tipo;
                                OPERACIONES: nombre de las operaciones que los manipulan)
Nombre de operación(tipo de argumento) → tipo de resultado
Efecto: Descripción de la operación
Excepción: Posibles excepciones
```

EJEMPLO 7.1. Especificar las operaciones del tipo de dato Conjunto.

El tipo genérico Conjunto es una colección no ordenada de elementos en el que no existen duplicados, con las operaciones básicas para añadir y eliminar elementos de uno en uno y determinar si un elemento pertenece o no al conjunto.

```
TAD Conjunto (VALORES: conjunto de elementos sin repetición;
              OPERACIONES: VacíaC, EsvacioC, AñadeC, Borrarc, PerteneceC)
VacíaC → Conjunto
Efecto: Crea un conjunto vacío
AñadeC(Conjunto, Elemento) → Conjunto
Efecto: Añade a un conjunto un elemento. No hace nada si ya está en el conjunto
Borrarc(Conjunto, Elemento) → Conjunto
Efecto: Elimina el elemento si está en él y no hace nada en otro caso
EsvacioC(Conjunto) → Boolean
Efecto: Decide si el conjunto está o no vacío
PerteneceC(Conjunto, Elemento) → Boolean
Efecto: Decide si un elemento está o no en el conjunto
```

La especificación formal de un TAD requiere para cada una de las operaciones la descripción de su **sintaxis** con los tipos de los argumentos y tipo de resultado, y de su **semántica**, donde se detallará para los distintos argumentos la expresión del resultado que se debe obtener. La especificación formal deberá seguir el siguiente esquema:

```
TAD Nombre del tipo de datos(VALORES: valores que toman los datos del tipo; OPERACIONES: nombre de
                                las operaciones que los manipulan)
```

Sintaxis:

```
/* Forma de las operaciones.- Lista de las funciones de la abstracción indicando el tipo de
argumentos y los resultados */
```

```
Nombre de operación(Tipo de argumentos) ⇒ Tipo de resultado
```

Semántica:

```
/* Significado de las operaciones.- Expresión que refleja, para unos determinados argumentos, el
resultado que se obtiene*/
```

```
Nombre de operación(valores particulares) ⇒ Expresión del resultado
```

En todos los TAD hay unas operaciones que son axiomas o constructores de los TAD, operaciones mediante las cuales se pueden crear todos los posibles valores de ese TAD, y que no se suelen detallar en la semántica excepto para indicar situaciones anómalas. En la sintaxis se marcarán los constructores escribiéndolos precedidos por el carácter asterisco. Por ejemplo, puesto que en el tipo genérico conjunto cuando se añade varias veces un mismo elemento el conjunto resultante no varía, hay que definir ecuaciones entre los constructores que expresen la propiedad conmutativa y especifiquen que los elementos sólo aparecerán una vez en el conjunto aunque se inserten en él de forma repetida, es decir que en ese caso el constructor no daría origen a un nuevo elemento.

EJEMPLO 7.2. *Se especifica la función igual para el tipo abstracto de dato Conjunto.*

En el TAD Conjunto hay que comparar los elementos del conjunto, es necesario por tanto tener una función Igual entre los elementos del mismo que cumpla las propiedades reflexiva, simétrica y transitiva. Esta función Igual, en realidad no es del tipo conjunto y se encontrará definida en el TAD elemento. En el tipo Conjunto se podrá indicar que se usa el tipo elemento y la especificación del TAD elemento hacerla de forma independiente.

```

Reflexiva:    Igual(e, e) → verdad
Simétrica :   si Igual(e, e1) = verdad entonces
               Igual(e1, e) → verdad
               fin_si
Transitiva:   si (Igual(e, e1) = verdad) y (Igual(e1, e2) = verdad) entonces
               Igual(e, e2) → verdad
               fin_si

```

7.2.2. IMPLEMENTACIÓN DE TIPOS ABSTRACTOS DE DATOS

La implementación de un tipo abstracto de dato consta de:

- La elección de las estructuras adecuadas para la representación de los datos del mismo.
- La elección de los algoritmos para efectuar las operaciones sobre ellos.

7.3. Orientación a objetos

La orientación a objetos puede describirse como el *conjunto de disciplinas (ingeniería) que desarrollan y modelizan software que facilitan la construcción de sistemas complejos a partir de componentes*.

Un *objeto* es un tipo abstracto de datos al que se añaden innovaciones en cuanto a compartición de código y reutilización, así como mecanismos de *herencia* y *polimorfismo*. Los *métodos* son las rutinas asociadas a un tipo de objeto. Normalmente un tipo *objeto* tiene sus campos de datos privados y todos o casi todos sus métodos públicos. El acceso a los datos se debe realizar a través de los métodos del objeto.

La *herencia* es la capacidad para crear nuevos tipos objetos (descendientes) que se construyen sobre otros existentes y permite que diferentes tipos de datos puedan compartir el mismo código, con lo que se obtiene una reducción en el tamaño del código de los programas y un incremento en la funcionalidad. La reutilización de código en programación orientada a objetos se realiza creando una subclase que constituye una restricción o extensión de la clase base, de la cual hereda sus propiedades y comportamiento.

El *polimorfismo* consigue que un mismo mensaje (comunicación entre objetos) pueda actuar sobre diferentes tipos de objetos y comportarse de modo distinto. El polimorfismo adquiere su máxima expresión en la derivación o extensión de clases, es decir, cuando se obtienen nuevas clases (tipos objeto) a partir de una ya existente mediante la propiedad de derivación de clases o herencia. Un método polimórfico es una rutina de un tipo objeto base, particularizada (redefinida) por otros tipos objeto derivados de él y capaz de comportarse con cada uno de la forma apropiada al mismo.

PROBLEMAS RESUELTOS BÁSICOS

7.1. Construir el TAD Booleano con los constructores *True* y *False* y las operaciones *Not*, *And*, *Or*, *Implica* y *SiysoloSi*.

Análisis

Si los constructores son *True* y *False*, hay que expresar todas las operaciones en función de ellos, por lo tanto tendrán el aspecto de una tabla de verdad.

Solución

```
TAD Boolean(VALORES: True, False;
            OPERACIONES: Not, And, Or, Implica, SiysoloSi)
```

Sintaxis:

```
*True           → Boolean
*False          → Boolean
Not(Boolean)     → Boolean
And(Boolean, Boolean) → Boolean
Or(Boolean, Boolean) → Boolean
Implica(Boolean, Boolean) → Boolean
SiysoloSi(Boolean, Boolean) → Boolean
```

Semántica: para todo *e* de tipo Boolean

```
Not(True)       ⇒ False
Not(False)      ⇒ True
And(e, True)    ⇒ e
And(e, False)   ⇒ False
Or(e, True)     ⇒ True
Or(e, False)    ⇒ e
Implica(True, e) ⇒ e
Implica(False, e) ⇒ True
SiysoloSi(True, True) ⇒ True
SiysoloSi(False, False) ⇒ True
SiysoloSi(True, False) ⇒ False
SiysoloSi(False, True) ⇒ True
```

7.2. Construir el TAD Booleano con los constructores *True* y *Not* y las operaciones *And*, *Or*, *Implica* y *SiySoloSi*.

Análisis

Si los constructores son *True* y *Not*, hay que añadir ecuaciones entre constructores. Ya que la expresión de *False* puede ser *False* = *Not* (*True*) = *Not*(*Not*(*Not*(*True*)))...

Solución

```
TAD Boolean(VALORES: True, False;
            OPERACIONES: Not, And, Or, Implica, SiysoloSi)
```

Sintaxis:

```
*True           → Boolean
*Not(Boolean)    → Boolean
And(Boolean, Boolean) → Boolean
```

```

Or(Boolean, Boolean)      → Boolean
Implica(Boolean, Boolean) → Boolean
SiysoloSi(Boolean, Boolean) → Boolean

```

Semántica:

```

Not(True)           ⇒ False
Not(Not(e))         ⇒ e
And(e, True)        ⇒ e
And(e, Not(True))   ⇒ False
Or(e, True)         ⇒ True
Or(e, Not(True))    ⇒ e
Implica(e, e1)      ⇒ Or(Not(e), e1)
Siysolosi(e, e1)    ⇒ And(Implica(e, e1), Implica(e1, e))

```

7.3. Construir el TAD Entero con los constructores Cero, Sucesor y Antecesor y las operaciones: Suma, Diferencia y Producto.

Análisis

Si los constructores son Cero, Sucesor, y Antecesor puesto que hay ambigüedad en la expresión de los elementos es necesario expresar ecuaciones entre los constructores.

$-1 = \text{Predecesor}(\text{Cero}) = \text{Predecesor}(\text{Predecesor}(\text{Sucesor}(\text{Cero})))$

El significado de las operaciones es:

Cero. Da el número entero 0.

Sucesor. El sucesor de un número entero es el entero que le sigue en su orden natural. El sucesor de -3 es -2.

Antecesor. El antecesor de un número entero es el entero que le precede en el orden natural. El antecesor del 5 es 4.

Suma. Realiza la suma de dos enteros ($2+3 = 5$).

Diferencia. Da la diferencia de dos enteros ($2-3 = -1$).

Producto. Da el producto de dos enteros ($2*(-3) = -6$).

Solución

```

TAD Entero(VALORES: Números enteros;
            OPERACIONES: Cero, Sucesor, Antecesor, Suma, Resta Producto)

```

Sintaxis:

```

*Cero           → Entero
*Sucesor(Entero) → Entero
*Antecesor(Entero) → Entero
Suma(Entero, Entero) → Entero
Diferencia(Entero, Entero) → Entero
Producto(Entero, Entero) → Entero

```

Semántica: para todo m, n de tipo entero

```

Sucesor(Antecesor(n)) ⇒ n
Antecesor(Sucesor(n)) ⇒ n
Suma(n, Cero)         ⇒ n
Suma(n, Sucesor(m))   ⇒ Sucesor(Suma(n, m))
Suma(n, Antecesor(m)) ⇒ Antecesor(Suma(n, m))
Diferencia(n, Cero)   ⇒ n
Diferencia(n, Sucesor(m)) ⇒ Antecesor(Diferencia(n, m))
Diferencia(n, Antecesor(m)) ⇒ Sucesor(Diferencia(n, m))

```

Producto(n, Cero)	\Rightarrow Cero
Producto(n, Sucesor(m))	\Rightarrow Suma(Producto(n, m), n)
Producto(n, Antecesor(m))	\Rightarrow Diferencia(Producto(n, m), n)

7.4. Construir el TAD Natural con los constructores Cero y Sucesor y las operaciones Escero, Igual, Suma, Antecesor, Diferencia, Producto, Menor.

Análisis del problema

Si los constructores son Cero y Sucesor hay que expresar todas las operaciones en función de ellas. Se usa Error para indicar las excepciones. Además como no hay ambigüedad en la expresión de los elementos no es necesario indicar ecuaciones entre los constructores. El significado de las operaciones es:

Cero: Da el número natural 0.

Sucesor: El sucesor de un número natural es el natural que le sigue en su orden. El sucesor de 2 es el 3.

Antecesor: El antecesor de un número natural es el natural que le precede en el orden. El antecesor del 5 es 4.

Suma: Realiza la suma de dos Naturales ($2+3 = 5$).

Diferencia: Da la diferencia de dos Naturales ($3-2 = 1$).

Producto: Da el producto de dos Naturales ($2*3 = 6$).

Escero: Indica si un Natural es el número cero.

Igual: Indica si dos Naturales son iguales.

Solución

TAD Natural (VALORES: números enteros positivos con el cero;

OPERACIONES: Cero, Sucesor, Escero, Igual, Suma, Antecesor, Diferencia, Producto, Menor)

Sintaxis:

*Cero	\rightarrow Natural
*Sucesor(Natural)	\rightarrow Natural
Escero(Natural)	\rightarrow Boolean
Igual(Natural, Natural)	\rightarrow Boolean
Suma(Natural, Natural)	\rightarrow Natural
Antecesor(Natural)	\rightarrow Natural
Diferencia(Natural, Natural)	\rightarrow Natural
Producto(Natural, Natural)	\rightarrow Boolean
Menor(Natural, Natural)	\rightarrow Boolean

Semántica: para todo m, n de tipo natural

EsCero(Cero)	\Rightarrow verdad
EsCero(Sucesor(n))	\Rightarrow falso
Igual(Cero, n)	\Rightarrow EsCero(n)
Igual(Sucesor(n), Cero)	\Rightarrow falso
Igual(Sucesor(n), Sucesor(m))	\Rightarrow Igual(n, m)
Suma(Cero, n)	\Rightarrow n
Suma(Sucesor(m), n)	\Rightarrow Sucesor(Suma(m, n))
Antecesor(Cero)	\Rightarrow Error
Antecesor(Sucesor(n))	\Rightarrow n
Diferencia(n, Cero)	\Rightarrow n
Diferencia(Cero, Sucesor(n))	\Rightarrow Error
Diferencia(Sucesor(m), Sucesor(n))	\Rightarrow Diferencia(m, n)
Producto(Cero, n)	\Rightarrow Cero
Producto(Sucesor(n), m)	\Rightarrow Suma(Producto(n, m), m)

```

Menor(n, Cero)           ⇒ verdad
Menor(Cero, Sucesor(n))  ⇒ falso
Menor(Sucesor(m), Sucesor(n)) ⇒ Menor(m, n)

```

7.5. *Añadir al TAD Natural del ejercicio anterior las operaciones DivisiónEntera, Módulo.*

Análisis del problema

La función DivisiónEntera encuentra la división entera entre dos naturales. DivisiónEntera(7,3) es 2

La función Módulo da el resto de la división entera entre dos naturales. Módulo(7,3) es 1.

Solución

Sintaxis:

```

DivisiónEntera(Natural, Natural) → Natural
Módulo(Natural, Natural)        → Natural

```

Semántica: para todo n, m de tipo natural

```

DivisiónEntera(n, Cero)           ⇒ Error
DivisiónEntera(Cero, Sucesor(n))  ⇒ Cero
DivisiónEntera(Sucesor(m), n)    ⇒
    si Menor(Sucesor(m), n) entonces
        Cero
    si_no
        Sucesor(DivisionEntera(Diferencia(Sucesor(m), n), n))
    fin_si
Módulo(n, Cero)                   ⇒ Error
Módulo(Cero, Sucesor(n))          ⇒ Cero
Módulo(Sucesor(m), n)             ⇒
    si Menor(Sucesor(m), n) entonces
        n
    si_no
        Módulo(Diferencia(Sucesor(m), n), n)
    fin_si

```

7.6. *Construir el TAD Vector como colección de elementos de un mismo tipo a los que se accede mediante un índice entero, entre el rango Comienzo, Final con las operaciones Crear, Existe, Asignar y Valor.*

Análisis

Los constructores serán Crear y Asignar. El primero de ellos crea el vector vacío en los límites indicados, y el segundo asigna a una posición de un vector un elemento. La operación existe, nos indicará si una posición del vector, tiene almacenado algún dato. La operación Valor nos dirá el valor que tiene almacenado una posición del array. Para que pueda ser tratado correctamente el problema, ha de tenerse en cuenta que hacen falta dos constructores del tipo entero que los llamaremos Comienzo y Final de tal manera que el constructor Crear que crea el vector vacío, lo pueda hacer entre los rangos determinados.

Solución

TAD Vector (VALORES: Elementos de un cierto tipo;
OPERACIONES: Crear, Existe, Asignar y Valor)

Sintaxis:

```

*Crear                                → Vector
*Asignar(Vector, Entero, Elemento) → Vector
Existe(Vector, Entero)                → Boolean
Valor(Vector, Entero)                 → Elemento

```

Semántica: Para todo m, n de tipo entero; para todo $e, e1, e2$ elemento; para todo v de tipo vector.

/*Se supone que m, n son enteros dentro del rango comienzo, final. La primera especificación es una ecuación que indica la semántica del constructor Asignar*/

```

Asignar(Asignar(v, m, e1), n, e) ⇒ si m = n entonces
                                Asignar(v, m, e)
                                si_no
                                Asignar(Asignar(v, n, e), m, e1))
                                fin_si
Existe(Crear, m)                 ⇒ False
Existe(Asignar(v, m, e), n)     ⇒ or(m = n, Existe(v, n))
Valor(Crear, m)                 ⇒ Error
Valor(Asignar(v, m, e), n)     ⇒ si m = n entonces
                                e
                                si_no
                                Valor(v, n)
                                fin_si

```

7.7. Definir la sintaxis y la semántica del TAD Alumnos de Segundo, como conjunto de todos los alumnos de segundo curso, y defina las operaciones:

Insertar: introduce un nuevo alumno.

Borrar: borra un alumno del conjunto.

EncontrarTodos: encuentra todos los alumnos que tengan el mismo nombre que uno dado.

ContarAlumnos: nos indicará el total de alumnos del curso.

Análisis

Los constructores serán `Adsvacio` e `Insertar`. Se suponen definidos los Tipos Abstractos de Datos: `Natural`, con la constante `Cero` y la función `Sucesor`, y `Boolean`, con las constantes `True` y `False`.

Solución

```

TAD Ads(VALORES: Alumnos;
        OPERACIONES: Insertar, Borrar, EncontrarTodos, ContarAlumnos, Esvacio, Adsvacio)

```

```

Sintaxis:
*Adsvacio                                → Ads
Esvacio(Ads)                             → Boolean
*Insertar(Ads, Alumno)                   → Ads
Borrar(Ads)                              → Ads
EncontrarTodos(Ads, Alumno)              → Ads
ContarAlumnos(Ads)                       → Natural

```

Semántica: Para todo $e1, e2$ de tipo Alumnos; para todo C de tipo Ads

```

Esvacio(Adsvacio)                       ⇒ True
Esvacio(Insertar(C, e1))                 ⇒ False

```

```

Borrar(Adsvacio)           ⇒ Error
Borrar(Insertar(C, e1))    ⇒ C
/* Borra el último elemento que entró en la clase*/
EncontrarTodos(Adsvacio, e1) ⇒ Adsvacio
EncontrarTodos(Insertar(C,e1),e2) ⇒ si e1 = e2 entonces
                                   Insertar(EncontrarTodos(C, e2), e1)
                                   si_no
                                   EncontrarTodos(C, e2)
                                   fin_si
ContarAlumnos(Adsvacio)    ⇒ Cero
ContarAlumnos(Insertar(C, e1)) ⇒ Sucesor(ContarAlumnos(C))

```

PROBLEMAS AVANZADOS

- 7.8.** Construir el TAD Conjunto como colección de elementos, no ordenados sin repetición con los constructores *ConjuntoVacio*, *Anade* y las operaciones *Borra*, *EsVacio*, *Pertenece*, *Unión*, *Intersección*, *Diferencia*, *DiferenciaSimétrica*, *Igual*, *Incluido*, *Cardinal*.

Análisis

Dado que pueden añadirse varias copias de un mismo elemento al conjunto y el conjunto resultante es el mismo, hay que definir ecuaciones entre los constructores que expresen esta propiedad. Además, puesto que hay que comparar elementos del conjunto es necesario tener una función *Iguales* entre los elementos del conjunto que cumpla las propiedades reflexiva, simétrica y transitiva (consultar ejercicio 7.2). Para la especificación de las operaciones se supone definido el Tipo Abstracto de Datos *Natural* con la función *Sucesor*.

El significado de las diferentes operaciones es:

ConjuntoVacio: Crea el conjunto vacío.

Anade: Añade un elemento a un conjunto.

Borra: Elimina un elemento de un conjunto.

EsVacio: Decide si un conjunto es vacío.

Pertenece: Decide si un elemento está en un conjunto.

Unión: Da la unión de dos conjuntos.

Intersección: Da la intersección de dos conjuntos.

Diferencia: Da la diferencia de dos conjuntos.

DiferenciaSimétrica: Da la diferencia simétrica de dos conjuntos: elementos de ambos conjuntos excepto los elementos comunes.

Igual: Decide si dos conjuntos son iguales.

Incluido: Decide si un conjunto está incluido en otro.

Cardinal: Indica el número de elementos que tiene un conjunto.

Solución

```

TAD Conjunto (VALORES: Conjunto de elementos sin repetición;
              OPERACIONES: ConjuntoVacio, Anade, Borra, EsVacio, Pertenece, Unión,
                          Intersección, Diferencia, DiferenciaSimétrica, Igual,
                          Incluido, Cardinal)

```


Sintaxis:

*ConjuntoVacio	→ Conjunto
*Anade(Conjunto, elemento)	→ Conjunto
Borra(Conjunto, Elemento)	→ Conjunto
Esvacio(Conjunto)	→ Boolean
Pertenece(Conjunto, elemento)	→ Boolean
Unión(Conjunto, Conjunto)	→ Conjunto
Intersección(Conjunto, Conjunto)	→ Conjunto
Diferencia(Conjunto, Conjunto)	→ Conjunto
DiferenciaSimétrica(Conjunto, Conjunto)	→ Conjunto
Igual(Conjunto, Conjunto)	→ Boolean
Incluido(Conjunto, Conjuto)	→ Boolean
Cardinal(Conjunto)	→ Natural

Semántica: Para todo e, e1 de tipo elemento y todo C, D de tipo Conjunto

Anade(Anade(C, e), e)	⇒ Anade(C, e)
Anade(Anade(C, e), e1)	⇒ Anade(Anade(C, e1), e)
Borra(ConjuntoVacio, e)	⇒ ConjuntoVacio
Borra(Anade(C, e1), e)	⇒
	si Iguales(e, e1) entonces
	Borra(C, e)
	si_no
	Anade(Borra(C, e), e1)
	fin_si
EsVacio(ConjuntoVacio)	⇒ True
Esvacio(Anade(C, e))	⇒ False
Pertenece(ConjuntoVacio, e)	⇒ false
Pertenece(Anade(C, e1), e)	⇒
	si Iguales(e, e1) entonces
	True
	si_no
	Pertenece(C, e)
	fin_si
Unión(ConjuntoVacio, C)	⇒ C
Unión(anade(C, e), D)	⇒ Anade(uni3n(C, D), e)
Intersecci3n(ConjuntoVacio, C)	⇒ ConjuntoVacio
Intersecci3n(Anade(C, e), D)	⇒
	si Pertenece(D, e) entonces
	Anade(Intersecci3n(C, D), e)
	si_no
	Intersecci3n(C, D)
	fin_si
Diferencia(C, ConjuntoVacio)	⇒ C
Diferencia(Anade(C, e), D)	⇒
	si Pertenece(D, e) entonces
	Diferencia(C, D)
	si_no
	Anade(Diferencia(C, D), e)
	fin_si
DiferenciaSimétrica(C, D)	⇒ Diferencia(Uni3n(C,D), Intersecci3n(C,D))
Igual(C, D)	⇒
	si Union(Diferencia(C, D), Diferencia(D, C)) es ConjuntoVacio entonces
	True

```

    si_no
      False
    fin_si
Incluido(C, D) ⇒
    si Diferencia(C, D) es ConjuntoVacio entonces
      True
    si_no
      False
    fin_si
Cardinal(ConjuntoVacio) ⇒ Cero
Cardinal(Anade(C, e)) ⇒
    si Pertenece(C, e) entonces
      Cardinal(C)
    si_no
      Sucesor(Cardinal(C))
    fin_si

```

- 7.9.** Construir el TAD Bolsa como colección de elementos, no ordenados con repetición con los constructores Bolsavacia, Añadir y las operaciones Esvacia, Esta, Cuantos, Union, Total.

Análisis

Si los constructores son Bolsavacia y Añadir y Sucesor hay que expresar todas las operaciones en función de ellos. Tal y como ocurre en el TAD Conjunto se necesitará una función Igual entre elementos que cumpla las propiedades reflexiva, simétrica y transitiva.

El significado de las distintas operaciones es:

Bolsavacia: Crea una bolsa sin ningún elemento.
 Esvacia: Decide si una bolsa está vacía.
 Anadir: Añade un elemento a una bolsa.
 Esta: Decide si un elemento se encuentra dentro de una bolsa.
 Cuantos: Indica la cantidad de veces que está repetido un elemento.
 Total: Indica el cardinal de la bolsa.
 Union: Construye la bolsa unión de dos bolsas.
 Interseccion: Construye la bolsa intersección de dos bolsas.

Solución

TAD Bolsa(VALORES: bolsa de Elementos ordenados que pueden repetirse;
 OPERACIONES: Bolsavacia, Esvacia, Añadir, Esta, Cuantos, Union, Total,
 Interseccion)

Sintaxis:

```

*Bolsavacia          → Bolsa
Esvacia(Bolsa)       → Boolean
*Añadir(Bolsa, Elemento) → Bolsa
Esta(Bolsa, Elemento) → Boolean
Cuantos(Bolsa, Elemento) → Natural
Total(Bolsa)         → Natural
Union(Bolsa, Bolsa)   → Bolsa
Interseccion(Bolsa, Bolsa) → Bolsa

```

Semántica: Sean B, C de tipo Bolsa; e, e1 de tipo elemento

Anadir(Añadir(B, e), e1)	⇒ Anadir(añadir(B, e1), e)
EsVacia(BolsaVacia)	⇒ True
EsVacia(Añadir(B, e))	⇒ False
Esta(BolsaVacia, e)	⇒ False
Esta(Añadir(B, e), e1)	⇒ si Igual(e, e1) entonces True si_no Esta(B, e1) fin_si
Cuantos(BolsaVacia, e)	⇒ Cero
Cuantos(Añadir(B, e), e1)	⇒ si Igual(e, e1) entonces Sucesor(Cuantos(B, e1)) si_no Cuantos(B, e1) fin_si
Total(BolsaVacia)	⇒ Cero
Total(Añadir(B, e))	⇒ Sucesor>Total(B))
Union(B, BolsaVacia)	⇒ B
Union(B, Anadir(C, e))	⇒ Anadir(Union(B, C), e)
Interseccion(B, BolsaVacia)	⇒ BolsaVacia
Interseccion(B, Añadir(C, e))	⇒ si esta(B, e) entonces Anadir(Interseccion(B, C), e) si_no Interseccion(B, C) fin_si

7.10. Definir la sintaxis y la semántica del tipo abstracto lista de números enteros con las operaciones:

ListaVacia: Crea la lista vacía.

Esvacia: Informa si la lista está vacía.

Insertar: Añade un número a la lista.

Máximo: Da el mayor número de la lista.

Mínimo: Da el menor número de la lista.

QuitarMayor: Elimina el mayor elemento de la lista (una aparición).

QuitarMenor: Elimina el menor elemento de la lista (una aparición).

Análisis

Los constructores son ListaVacia e Insertar.

Solución

TAD Lista (VALORES: números enteros;
OPERACIONES: Listavacia, Esvacia, Insertar, Maximo, Minimo,
QuitarMayor, QuitarMenor)

Sintaxis:

*ListaVacia	→ Lista
EsVacia(Lista)	→ Boolean
*Insertar(Lista, Entero)	→ Lista
Maximo(Lista)	→ Entero
Minimo(Lista)	→ Entero

```

QuitarMayor(Lista)           → Lista
QuitarMenor(Lista)          → Lista

Semántica: Para todo L de tipo Lista; para todo n de tipo entero
EsVacia(ListaVacia)          ⇒ True
EsvaVacia(Insertar(L, n))    ⇒ False
Maximo(ListaVacia)           ⇒ Error
Maximo(Insertar(L, n))       ⇒ si EsVacia(L) entonces
                                n
                                si_no
                                si n > Maximo(L) entonces
                                    n
                                    si_no
                                        Maximo(L)
                                fin_si
                                fin_si
Minimo(ListaVacia)           ⇒ Error
Minimo(Insertar(L, n))       ⇒ si EsVacia(L) entonces
                                n
                                si_no
                                si n < Minimo(L) entonces
                                    n
                                    si_no
                                        minimo(L)
                                fin_si
                                fin_si
QuitarMayor(ListaVacia)      ⇒ Error
QuitarMayor(Insertar(L, n))  ⇒ si n ≥ Máximo(L) entonces
                                L
                                si_no
                                    Insertar(QuitarMayor((L), n)
                                fin_si
QuitarMenor(ListaVacia)      ⇒ Error
QuitarMenor(Insertar(L, n))  ⇒ si n ≤ Minimo(L) entonces
                                L
                                si_no
                                    Insertar(Quitarmenor((L), n)
                                fin_si

```

7.11. Definir el TAD cadena como secuencia de caracteres, con las siguientes operaciones definidas sobre ella:

CadenaVacia : Crea la cadena vacía.

Poner: Añade un carácter al final de la cadena.

Esvacia: Decide si una cadena está vacía.

Longitud: Nos da la longitud de una cadena.

Concatenar: Nos devuelve una cadena resultado de la concatenación de otras dos.

Borrarcar: Elimina de la cadena todas las apariciones del carácter.

CaracterN: Nos da el carácter que se encuentra en la posición indicada.

Análisis

Los constructores serán *CadenaVacia*, y *Poner*. Se supone definidos los tipos Abstractos de Datos: *Natural*, con la constante *Cero* y la función *Sucesor*, y *Boolean* con las constantes *True* y *False*.

Solución

TAD Cadena(VALORES: Caracter;
 OPERACIONES: Cadenavacia, Poner, Esvacia, Longitud, Concatenar,
 Borrarcad, CaracterN)

Sintaxis:

*CadenaVacia	→ Cadena
*Poner(Cadena, Carácter)	→ Cadena
Esvacia(Cadena)	→ Cadena
Longitud(Cadena)	→ Entero
CaracterN(cadena, Entero)	→ Caracter
Concatenar(Cadena, Cadena)	→ Cadena
Borrarcad(Cadena, Carácter)	→ Cadena

Semántica: Para todo cad, cad1 de tipo Cadena; para todo c, c1 de tipo carácter; para todo n de tipo Natural

Esvacia(cadenavacia)	⇒ True
Esvacia (Poner(cad, c))	⇒ False
Longitud(CadenaVacia)	⇒ Cero
Longitud(Poner(cad, c))	⇒ Sucesor(Longitud(cad))
Concatenar(cad, CadenaVacia)	⇒ cad
Concatenar(cad, Poner(cad1, c))	⇒ Poner(Concatenar(cad, cad1), c)
CaracterN(CadenaVacia, n)	⇒ nulo
CaracterN(Poner(cad, c), cero)	⇒ nulo
CaracterN(poner(cad, c), n)	⇒ si Longitud(cad) = n entonces c si_no si (longitud(cad) < n entonces nulo si_no CaracterN(cad, n) fin_si fin_si
Borrarcad(CadenaVacia, c)	⇒ CadenaVacia
Borrarcad(Poner(cad, c), c1))	⇒ si c = c1 entonces Borrarcad(cad, c1) si_no poner(Borrarcad(cad, c1), c) fin_si

7.12. Implementar el TAD conjunto capaz de almacenar 512 valores de tipo entero.

Análisis

La estructura de datos que se utilizará en el TAD será un registro con dos campos: uno que indica el número de elementos actualmente en el conjunto y otro que es un *array* de enteros y contiene los elementos introducidos en él ordenados crecientemente. La variable *errorconjunto* se pone a verdadero cuando hay algún tipo de error. Se implementan las siguientes funciones del **TAD Conjunto**:

- **ConjuntoVacio.** Crea el conjunto vacío poniendo el miembro *NumElementos* a 0.

- **Añade.** Recibe como parámetro un entero *n* y lo añade al conjunto *c*.
- **EsVacio.** Decide si un conjunto está vacío.
- **Pertenece.** Decide si un entero está en el conjunto.
- **Borra.** Elimina un elemento del conjunto.
- **Cardinal.** Indica el cardinal del conjunto.
- **Buscar.** Decide si un elemento está en el conjunto, indicando además la dirección de memoria donde se almacena el entero.

Codificación

```
#include <stdio.h>
typedef struct
{
    int Numelementos;
    int L[512];           /* empieza en 0 */
} Conjunto;

int Errorconjunto;
void ConjuntoVacio(Conjunto *c);
void Anade(Conjunto *c, int n);
int EsVacio(Conjunto c);
int Pertenece(Conjunto c, int n);
void Borra(Conjunto *c, int n);
int Cardinal(Conjunto c);
void Buscar(Conjunto c, int n, int *encontrado, int *p);

void ConjuntoVacio(Conjunto *c)
{
    Errorconjunto = 0;
    c->Numelementos = 0 ;
}

void Buscar(Conjunto c, int n, int *encontrado, int *p)
{
    // la búsqueda se realiza de forma binaria.
    int primero, ultimo, central;
    primero = 0;
    ultimo = c.Numelementos - 1;
    encontrado = 0;
    while ((primero <= ultimo) && (!encontrado))
    {
        central = (primero + ultimo)/ 2;
        if (c.L[central] == n)
            *encontrado = 1;
        else
            if (c.L[central] < n)
                primero = central + 1;
            else
                ultimo = central - 1 ;
    }
    if (!encontrado)
        *p = primero;
    else
```

```

        *p = central;
    };

    void Anade(Conjunto *c,int n)
    {
        int encontrado, p, i;
        Buscar(*c, n, &encontrado, &p);
        if (!encontrado)
            if (c->Numelementos < 512)
            {
                for (i = c->Numelementos-1; i >= p; i--)
                    c->L[i+1] = c->L[i];           // desplaza datos a la derecha.
                c->L[p] = n;
                c->Numelementos++;
            }
            else
                Errorconjunto = 1;
    }

    int EsVacio(Conjunto c)
    {
        return(c.Numelementos == 0);
    }

    int Cardinal(Conjunto c)
    {
        return(c.Numelementos);
    }

    int Pertenece(Conjunto c, int n)
    {
        int encontrado, p;
        Buscar(c, n, &encontrado, &p);
        return (encontrado);
    }

    void Borra(Conjunto *c, int n)
    {
        int encontrado,p,i;
        Buscar(*c, n, &encontrado, &p);
        if (encontrado)
        {
            // desplaza datos hacia la izquierda.
            for (i = p; i < c->Numelementos - 1 ; i++)
                c->L[i] = c->L[i + 1];
            c->Numelementos --;
        }
    }

```

7.13. *Añadir al TAD Conjunto del ejercicio anterior las funciones incluido, unión, diferencia, diferenciasimétrica, e igualdad de conjuntos.*

Análisis

- La función *Unión* se basa en la mezcla ordenada de dos *array* de números enteros controlando que no aparezcan errores en la fusión.
- La función *Intersección* realiza la mezcla de dos listas ordenadas incluyendo los elementos que están en las dos listas.
- La función *Diferencia* añade al conjunto *c2* los elementos que estén en el conjunto *c1* y no estén en el *c2*, realizando un recorrido del conjunto *c1*.
- La función *DiferenciaSimetrica*, *igual* e *incluido* se implementan usando las propiedades de los conjuntos.

Codificación

```

void Union(Conjunto c1, Conjunto c2, Conjunto *c3);
void Interseccion(Conjunto c1, Conjunto c2, Conjunto *c3);
void Diferencia(Conjunto c1, Conjunto c2, Conjunto *c3);
void Diferenciasimetrica(Conjunto c1, Conjunto c2, Conjunto *c3);
int Igual(Conjunto c1, Conjunto c2);
int Incluido(Conjunto c1, Conjunto c2);

void Union(Conjunto c1, Conjunto c2, Conjunto *c3)
{
    int i = 0, j = 0, k = -1; int Enorconjunto = 0;
    while ((i <= Cardinal(c1) - 1) && (j <= Cardinal(c2) - 1) && (! Errorconjunto))
    {
        k ++;
        if (k > 511)
            Errorconjunto = 1;
        else
        {
            c3->Numelementos = k+1;
            if (c1.L[i] < c2.L[j])
            {
                c3->L[k] = c1.L[i];
                i ++;
            }
            else
            {
                if(c1.L[i] > c2.L[j])
                {
                    c3->L[k] = c2.L[j];
                    j ++;
                }
                else
                {
                    if (c1.L[i]== c2.L[j])
                    {
                        c3->L[k] = c2.L[j];
                        j ++;
                        i ++;
                    }
                }
            }
        }
    }

    while ((i <= Cardinal(c1)-1) && (! Errorconjunto))
    {
        k ++;
    }
}

```



```

        if (k > 511)
            Errorconjunto = 1;
        else
        {
            c3->Numelementos = k+1;
            c3->L[k] = c1.L[i];
            i++;
        }
    }
    while ((j <= Cardinal(c2)-1) && (! Errorconjunto))
    {
        k ++;
        if (k > 511)
            Errorconjunto = 1;
        else
        {
            c3->Numelementos = k;
            c3->L[k] = c2.L[j];
            j++;
        }
    }
}

```

```

void Interseccion(Conjunto c1, Conjunto c2,Conjunto *c3)
{
    int i = 0, j = 0;
    c3->Numelementos = 0;
    while ((i <= Cardinal(c1)-1) && (j <= Cardinal(c2)-1))
        if (c1.L[i] < c2.L[j])
            i ++;
        else
            if (c1.L[i] > c2.L[j])
                j ++;
            else
                if (c1.L[i] == c2.L[j])
                {
                    c3->Numelementos ++;
                    c3->L[c3->Numelementos-1]= c2.L[j]; j ++ ;
                    i ++ ;
                }
    }
}

```

```

void Diferencia(Conjunto c1, Conjunto c2,Conjunto *c3)
{
    int i = 0, j = 0, r1;
    c3->Numelementos = 0;
    while ((i <= Cardinal(c1)-1) && (j <= Cardinal(c2)-1))
        if (c1.L[i] < c2.L[j])
        {
            c3->Numelementos ++;
            c3->L[c3->Numelementos-1] = c1.L[i];
            i ++;
        }
    }
}

```

```
        else
            if (c1.L[i] > c2.L[j])
                j ++;
            else
                if (c1.L[i] == c2.L[j])
                {
                    j ++;
                    i ++;
                };
        for (r1 = i; i < c1.Numelementos; r1++)
        {
            c3->L[c3->Numelementos] = c1.L[r1];
            c3->Numelementos++;
        }
    }

void DiferenciaSimetrica(Conjunto c1, Conjunto c2, Conjunto *c3)
{
    Conjunto cu, ci;
    Union(c1, c2, &cu);
    Interseccion(c1, c2, &ci);
    Diferencia(cu, ci, c3);
}

int Incluido(Conjunto c1, Conjunto c2)
{
    Conjunto cd;
    Diferencia(c1, c2, &cd);
    return (EsVacio(cd));
}

int Igual(Conjunto c1, Conjunto c2)
{
    Conjunto cd1, cd2, cu;
    Diferencia(c1, c2, &cd1);
    Diferencia(c2, c1, &cd2);
    Union(cd1, cd2, &cu);
    return (EsVacio(cu));
}
```

PROBLEMAS PROPUESTOS

- 7.1. Diseñar un tipo abstracto de datos pila de números enteros y que al menos soporte las siguientes operaciones:

Pilavacia: Devuelve una pila vacía.

Cima: Devuelve el elemento cima de la pila. Si la pila está vacía produce error.

Meter: Añade un nuevo elemento en la cima de la pila.

Sacar: Devuelve la pila sin el elemento cima. Si la pila está vacía produce error.

Borrar: Elimina todos los números de la pila.

Copiar: Hace una copia de la pila actual.

Longitud: Devuelve un número natural igual al número de objetos de la pila.

Llena: Devuelve *verdadero* si la pila está llena (no existe espacio libre en la pila)

Vacía: Devuelve *verdadero* si la pila está vacía y *falso* en caso contrario.

Igual: Devuelve *verdadero* si existen dos pilas que tienen la misma profundidad y las dos secuencias de números son iguales cuando se comparan elemento a elemento desde sus respectivas cimas de la pila; *falso* en caso contrario.

- 7.2. Diseñar el tipo abstracto *Cola* que sirva para implementar una estructura de datos cola.
- 7.3. Añadir al TAD *Natural* del Ejercicio Resuelto 7.4 las operaciones *Mcd*, *Mcm* y *Pot*.
- 7.4. Añadir al TAD número Entero las operaciones:
- *Negativo*: Decide si un número entero es negativo.
 - *Menor*: Decide si un número entero es menor que otro.
 - *Mayor*: Decide si un número entero es mayor que otro.
 - *Menor_o_igual*: Decide si un entero es menor o igual que otro.
 - *Mayor_o_igual*: Decide si un número entero es mayor o igual que otro.
- 7.5. Diseñar el TAD número *Racional* y sus diferentes operaciones.
- 7.6. Añadir al TAD lista de número enteros, las operaciones
- *Insertar_en_orden*: Inserta en una lista de enteros ordenadamente.
 - *Ordenado*: Decide si un vector está ordenado.
 - *Mezclar*: Mezcla dos listas ordenadas dando otra lista ordenada.
 - *Ordenar*: Ordena una lista.
- 7.7. 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 cuando tienen 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.
- 7.8. 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.
- 7.9. Implementar el TAD *Bolsa*. Probar la implementación con un programa que invoque a las operaciones del tipo abstracto *Bolsa*.
- 7.10. Implementar el TAD *Cadena*. Probar la implementación con un programa que realice diversas operaciones con cadenas.
- 7.11. Implementar el TAD *Vector* con una estructuras dinámica.

Listas, listas enlazadas

En este capítulo se comienza el estudio de las estructuras de datos dinámicas. Al contrario que las estructuras de datos estáticas (*arrays* o *arreglos* - 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 estudia en este capítulo es la **lista enlazada** (**ligada** o **enca-denada**, “**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”.

8.1. Estructuras de datos dinámicas

Una estructura de datos dinámica es una colección de elementos llamados **nodos**, que se enlazan o encadenan unos con otros. Los enlaces se establecen gracias a que cada nodo contiene *al menos* un elemento de tipo puntero que apunta a otro nodo. Estos punteros permiten la reserva o liberación de las posiciones de memoria que tienen asociadas. Existen diferentes tipos de estructuras dinámicas de datos, siendo las más importantes *listas enlazadas*, *pilas*, *colas*, *árboles* y *grafos*.

8.2. Punteros (Apuntadores)

Un *puntero*¹ es una variable estática que almacena la dirección de memoria o posición que puede corresponder o no a una variable declarada en el programa. La declaración de una variable puntero se realiza de la siguiente forma:

<tipo de dato apuntado> *<identificador de puntero>. Una variable puntero que posteriormente puede generar una lista puede declararse de la siguiente forma.

```
struct Nodo          typedef struct Nodo          typedef double Elemento;
{ int info;          { int info;          struct nodo
  struct Nodo* sig;    struct Nodo *sig;    { Elemento info;
}p;                  }NODO;          struct nodo *sig}p;
```

¹ En Latinoamérica, el término utilizado para definir este concepto suele ser *apuntador*.

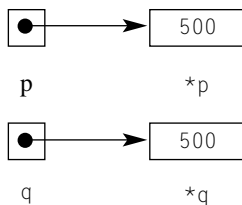
Operaciones con variables puntero.

- **Inicialización.** `p = NULL`. La palabra `NULL` representa el **puntero nulo**. `NULL` es una constante de la biblioteca estándar `stdlib.h` de C. Este puntero se usa: en el campo `sig` del nodo final de una lista enlazada o en una lista vacía.
- **Operador de selección de miembro.** Si `p` es un puntero a una estructura y `m` es un miembro de la estructura, entonces `p → m` accede al miembro `m` de la estructura apuntada por `p`. El símbolo “`->`” se denomina *operador de selección de componente*, y `p->m` significa lo mismo que `(*p).m`.
- **Comparación.** `p == q`. Sólo se admite los operadores `==` ó `!=`, siendo los punteros del mismo tipo.
- **Asignación.** `q = p`, implica hacer que el puntero `q` apunte a donde apunta `p`.
- **Aritmética de punteros.** A un puntero se le puede sumar o restar un entero `n`. Esto hace que apunte `n` posiciones adelante o atrás de la actual. A una variable puntero se le puede aplicar el operador `++` o el operador `--`. Esto hace que contenga la posición del siguiente o anterior elemento.
- **Creación de variables dinámicas.** `p = (tipo*)malloc(sizeof(tipo))`, reserva espacio en memoria para la variable dinámica del tipo dado.
- **Eliminación de variables dinámicas.** `free(p)` libera el espacio en memoria ocupado por la variable dinámica.
- **Paso de punteros como parámetros.** El paso de punteros se realiza como el de cualquier otro tipo de dato y podrá efectuarse por valor o por variable.
- **Funciones de tipo puntero.** Una función puede ser de tipo puntero, es decir puede devolver un puntero.
- **Dirección.** Si `x` es una variable de un cierto tipo &`x` es una variable puntero que apunta a la variable `x`.

8.3. Variables dinámicas

Una *variable dinámica* es una variable simple o estructura de datos sin nombre y creada en tiempo de ejecución. Las variables dinámicas no se declaran. Para acceder a una variable dinámica, como no tiene nombre se emplea `*nombre_variable_tipo_puntero`. La asignación entre punteros no debe confundirse con la asignación entre las variables dinámicas apuntadas.

EJEMPLO 8.1. *Dado `int *p, *q`, y `*p` almacena el número 500, `*q = *p` significa*



8.4. Tipos puntero predefinidos `NULL` y `void`

Existen dos punteros especiales muy utilizados son los punteros `void` y `NULL`. El puntero nulo `NULL` no direcciona ningún dato válido en memoria. Se usa para conocer cuando un puntero no direcciona un dato (final de lista). El puntero genérico `void` direcciona datos de un tipo no especificado. Un puntero tipo `void` se puede igualar a `NULL` si no direcciona ningún dato válido. `NULL` es un valor; `void` es un tipo de dato.

EJEMPLO 8.2. *Se declara un tipo denominado `Punto`, representa un punto en el plano con su coordenada `x` e `y`. También se declara el tipo `Nodo` con el campo `dato` del tipo `Punto`. Por último, se define un puntero a `Nodo`.*

```
#include <stdlib.h>

typedef struct punto
{
    float x, y;
} Punto;
```

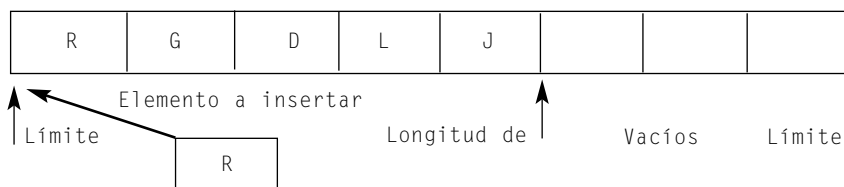
```
typedef struct nodo
{
    Punto dato;
    struct nodo* enlace;
}Nodo;

Nodo* cabecera;
cabecera = NULL;
```

8.5. Conceptos generales sobre listas

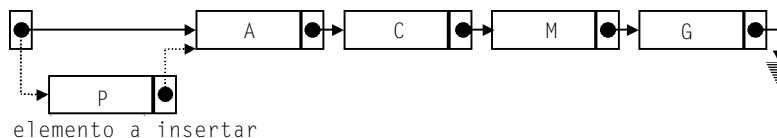
Una lista es una secuencia de 0 o más elementos de un tipo dado almacenados en memoria. Son estructuras lineales, donde cada elemento de la lista, excepto el primero, tiene un único predecesor y cada elemento de la lista, excepto el último, tiene un único sucesor. El número de elementos de una lista se llama longitud. Si una lista tiene 0 elementos se denomina lista vacía. Es posible considerar distintos tipos de listas:

Contiguas. Los elementos son adyacentes en la memoria del ordenador y tienen unos límites, izquierdo y derecho, que no pueden ser rebasados cuando se añade un nuevo elemento. Se implementan a través de *arrays*. La inserción o eliminación de un elemento en la lista suele implicar la traslación de otros elementos de la misma.

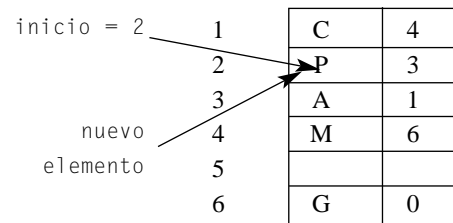
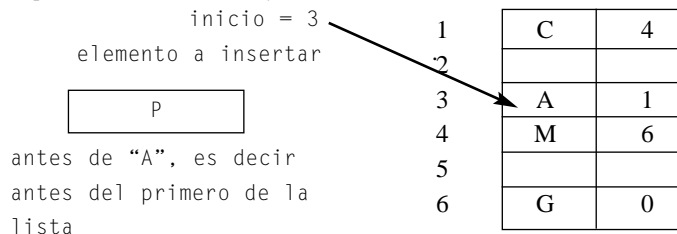


Enlazadas. Los elementos se almacenan en posiciones de memoria que no son contiguas o adyacentes, por lo que cada elemento necesita almacenar la posición o dirección del siguiente elemento de la lista. Son mucho más flexibles y potentes que las listas contiguas. La inserción o borrado de un elemento de la lista no requiere el desplazamiento de otros elementos de la misma. Se deben implementar de forma dinámica, pero también es posible efectuarlo a través de *arrays*, aunque esto limita el número de elementos que dicha lista podrá contener y establece una ocupación en memoria constante.

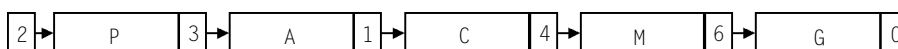
Implementación con punteros:



Implementación con arrays:



Como se puede observar, tanto cuando se implementa con punteros como cuando se hace a través de *arrays*, la inserción de un nuevo elemento no requiere el desplazamiento de los que le siguen. Para observar la analogía entre ambas implementaciones se puede recurrir a representar la implementación con *arrays* de la siguiente forma:



Circulares. Son una modificación de las enlazadas en las que el puntero del último elemento apunta al primero de la lista.

Doblemente encadenadas. Su recorrido puede realizarse tanto de frente a final como de final a frente. Cada nodo de dichas listas consta de un campo con información y otros dos de tipo puntero (*Ant* y *Sig*) y será referenciado por dos punteros, uno de su nodo sucesor y otro del anterior.

Listas doblemente encadenadas circulares. En este tipo de listas el campo *ant* del primer nodo de la lista apunta al último y el campo *sig* del último nodo al primero.

8.6. Especificación del Tipo Abstracto de Datos Lista

La especificación formal del tipo Abstracto de Datos *Lista* es:

TAD *Lista*(VALORES: secuencia de elementos; OPERACIONES: *VaciaL*, *EsVaciaL*, *PrimeroL*, *InsertaPL*, *RestoL*, *ModificaL*, *BorraPL*)

Sintaxis:

<i>*VaciaL</i> ()	→	<i>Lista</i>
<i>EsVaciaL</i> (<i>Lista</i>)	→	Boolean
<i>PrimeroL</i> (<i>Lista</i>)	→	elemento
<i>*InsertaPL</i> (<i>Lista</i> , elemento)	→	<i>Lista</i>
<i>Resto</i> (<i>Lista</i>)	→	<i>Lista</i>
<i>ModificaL</i> (<i>Lista</i> , elemento)	→	<i>Lista</i>
<i>BorraPL</i> (<i>Lista</i>)	→	<i>Lista</i>

Semántica: Sea *L*: *Lista*; *x*, *y*: elemento

<i>EsVaciaL</i> (<i>VaciaL</i> ())	⇒	verdad
<i>EsVaciaL</i> (<i>InsertaPL</i> (<i>L</i> , <i>x</i>))	⇒	falso
<i>PrimeroL</i> (<i>VaciaL</i> ())	⇒	error
<i>PrimeroL</i> (<i>InsertaPL</i> (<i>L</i> , <i>x</i>))	⇒	<i>x</i>
<i>RestoL</i> (<i>VaciaL</i> ())	⇒	error
<i>RestoL</i> (<i>InsertaPL</i> (<i>L</i> , <i>x</i>))	⇒	<i>L</i>
<i>ModificaL</i> (<i>VaciaL</i> (), <i>x</i>)	⇒	error
<i>ModificaL</i> (<i>InsertaPL</i> (<i>L</i> , <i>x</i>), <i>y</i>)	⇒	<i>InsertaPL</i> (<i>L</i> , <i>y</i>)
<i>BorraPL</i> (<i>VaciaL</i> ())	⇒	error
<i>BorraPL</i> (<i>InsertaPL</i> (<i>L</i> , <i>x</i>))	⇒	<i>L</i>

8.7. Operaciones sobre listas enlazadas

Las operaciones sobre listas enlazadas más usuales son: inicialización o creación; insertar elementos en una lista; eliminar elementos de una lista; buscar elementos de una lista; recorrer una lista enlazada. comprobar si la lista está vacía.

EJEMPLO 8.3. *Crear una lista simplemente enlazada de elementos que almacenen datos de tipo entero.*

Un elemento de la lista se puede definir con la ayuda de la estructura siguiente:

```
typedef struct Elemento
{
    int dato;
    struct Elemento * siguiente;
} Nodo;
```

El siguiente paso para construir la lista es declarar la variable *primero* que apuntará al primer elemento de la lista : `Nodo *primero = NULL`. Se inicializa a 0 o a `NULL` para indicar que la lista no tiene elementos. Ahora se crea un elemento de la lista, para ello hay que reservar memoria, tanta como tamaño tiene cada nodo, y asignar la dirección de la memoria reservada al puntero *primero* : `primero = (Nodo*)malloc(sizeof(Nodo))`; a partir de este momento es cuando se puede asignar valor a los campos. La forma de efectuarlo puede ser: `primero->dato = 11; primero->siguiente = NULL`; o bien `(*primero).dato = 11; (*pri-`

mero).siguiente = NULL; La operación de crear un nodo se puede hacer en una función a la que se pasa el valor del campo dato y del campo siguiente. La función devuelve un puntero al nodo creado:

```
Nodo* crearNodo(int x, Nodo* enlace)
{
    Nodo *p;
    p = (Nodo*)malloc(sizeof(Nodo));
    p->dato = x;
    p->siguiente = enlace;
    return p;
}
```

La llamada a la función crearNodo() para crear el primer nodo de la lista: primero = crearNodo(11, NULL); Si ahora se desea añadir un nuevo elemento con un valor 6, y situarlo en el primer lugar de la lista se escribe simplemente: primero = crearNodo(6, primero);

8.8. Especificación formal del Tipo Abstracto de Datos *Lista ordenada*

Una lista está ordenada cuando sus elementos están organizados, en orden creciente o decreciente, por el contenido de uno de sus campos. Al TAD Lista se añaden dos nuevas funciones que son InsertaOrd y BorraOrd de la siguiente forma:

TAD Lista_ordenada_en_orden_ascendente (VALORES: secuencia ordenada de elementos en orden ascendente; OPERACIONES: <todas las anteriores>, InsertaOrd, BorraOrd)

Sintaxis:

```
<todas las especificadas del TAD Lista>
InsertaOrd(lista, elemento)    → Lista
BorraOrd(lista, elemento)     → Lista
```

Semántica: Sea L: Lista; x, y: elemento

```
<todas las especificadas del TDA Lista>
InsertaOrd(VaciaL(), x)       ⇒ InsertaL(VaciaL(), x)
InsertaOrd(InsertaPL(L, y), x) ⇒ si menor(x, y) entonces
                                InsertaPL(InsertaPL(L, y), x)
                                si_no
                                InsertaPL(InsertaOrd(L, x), y)
                                fin_si
BorraOrd(VaciaL(), x)         ⇒ error
BorraOrd(InsertaPL(L, y), x)  ⇒ si igual(x, y) entonces
                                L
                                si_no
                                si menor(x, y) entonces
                                error
                                si_no
                                InsertaPL(BorraOrd(L, x), y)
                                fin_si
                                fin_si
```

8.9. Inserción y borrado de un elemento en lista enlazada simple

El algoritmo empleado para añadir o insertar un elemento en una lista enlazada es el siguiente:

```
inicio
<Buscar la posición donde insertar el nuevo elemento>
<Asignar memoria para un elemento (NuevoPuntero) >
NuevoPuntero->info ← elemento
Si la inserción se realiza en el comienzo de la lista entonces
```



```

    NuevoPuntero->sig ← Lista
    Lista ← NuevoPuntero
sino
    <en Anterior se debe tener la dirección del nodo que está antes del NuevoPuntero (siempre existe)>
    NuevoPuntero->sig ← Anterior->Sig
    Anterior->sig ← NuevoPuntero
fin si
fin

```

El algoritmo empleado para borrar un elemento en la lista enlazada es el siguiente:

```

inicio
<Buscar el posición pos del nodo que contiene el elemento a borrar>
Si el borrado se realiza en el comienzo de la lista entonces
    Lista ← pos->sig
sino
    <en Anterior se debe tener la dirección del nodo que está antes de la posición pos (siempre existe)>
    Anterior->sig ← pos->sig
fin si
<liberar la memoria apuntada por pos>
fin

```

PROBLEMAS BÁSICOS

8.1. Efectuar la implementación del TAD *Lista*.

Análisis

Siguiendo la especificación para el TAD *Lista*, se implementan las funciones mediante las primitivas

- *VaciaL*, crea la lista vacía.
- *EsVaciaL*, decide si la lista está vacía.
- *PrimeroL*, devuelve el primer elemento de la lista.
- *InsertaPL*, inserta un elemento como primer elemento de la lista.
- *RestoL*, retorna un puntero que apunta al segundo elemento de la lista que recibe como parámetro.
- *ModificaL*, cambia la información del primer elemento de la lista.
- *BorraPL*, borra el primer elemento de la lista.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
typedef int Telemento;
typedef struct Registro
{
    Telemento e;
    struct Registro* Sig;
}

```

```
} Nodo;

void Vacial(Nodo ** L)
{
    *L = NULL;
}

int EsVacial(Nodo * L)
{
    return( L == NULL);
}

void AnadePL(Nodo** L, Telemento e)
{
    Nodo *Nuevo ;
    Nuevo = (Nodo*)malloc(sizeof(Nodo));
    Nuevo -> e = e;
    Nuevo -> Sig = *L;
    *L = Nuevo;
}

void BorraPL (Nodo** L)
{
    Nodo *Ptr;
    Ptr = *L;
    if(Ptr == NULL)
    {
        printf("error en BorraP");
        return;
    }
    *L = Ptr->Sig;
    free(Ptr);
}

Nodo *Resto( Nodo *L)
{
    if (L == NULL)
    {
        printf("error en restoL \n") ;
        return NULL;
    }
    return L->Sig;
}

void ModificaL(Nodo **L, Telemento e)
{
    if (L == NULL)
    {
        printf("error en ModificaL \n") ;
        return;
    }
    (*L)->e = e;
}
```

```

Telemento PrimeroL (Nodo* L)
{
    if (L == NULL)
    {
        printf("error en PrimeroL \n") ;
        return NULL;
    }
    return L->e;
}

```

8.2. Implementar el TAD lista ordenada.

Análisis

Para la implementación del TAD Lista Ordenada se usa el ejercicio 8.1 y se implementan las siguientes funciones:

La función `NuevoNodo` se encarga de crear un nodo donde almacenar el dato que recibe como parámetro, y coloca el campo siguiente a `NULL`.

La función `InsertarOrd` recibe como parámetro una lista enlazada ordenada y un dato y lo inserta dejándola de nuevo ordenada. Para realizarlo, primeramente crea el nodo donde almacenará el dato, si es el primero de la lista lo inserta, y en otro caso mediante un bucle `while` recorre la lista hasta encontrar donde colocar el dato. Una vez encontrado el sitio se realiza la inserción de acuerdo con el algoritmo correspondiente especificado en la teoría.

La función `BorrarOrd` se encarga de buscar la primera aparición del dato en la lista enlazada ordenada y borrarlo. Para ello realiza la búsqueda de la posición donde se encuentra la primera aparición del dato quedándose con el puntero `Pos` y con el puntero `Ant` (anterior). Posteriormente realiza el borrado teniendo en cuenta que sea el primero de la lista o que no lo sea.

Codificación

```

Nodo* NuevoNodo(Telemento e)
{
    Nodo *nn ;
    nn = (Nodo*)malloc(sizeof(Nodo));
    nn -> e = e;
    nn -> Sig = NULL;
    return nn;
}

void InsertarOrd(Nodo** L, Telemento e)
{
    Nodo *Nuevo, *Ant, *Pos;
    Nuevo = NuevoNodo( e);
    if (*L == NULL)
        *L = Nuevo;
    else
        if ( e <= (*L)-> e)
        {
            Nuevo -> Sig = *L;
            *L = Nuevo;
        }
        else
        {
            /* la insercion se realiza no en la primera poscion de la lista*/
            Ant = Pos = *L;
            while (( e > Pos-> e)&&(Pos->Sig != NULL) )
            {

```

```

        Ant = Pos;
        Pos = Pos->Sig;
    }
    if ( e > Pos-> e)                                // falta por comprobar el ultimo
        Ant = Pos;
    Nuevo -> Sig = Ant -> Sig;
    Ant -> Sig = Nuevo;
}
}

void BorrarOrd(Nodo** L, Telemento e)
{
    Nodo *Ant, *Pos;
    int Encontrado=0;
    Ant=NULL;
    Pos= *L;
    while ((!Encontrado)&&(Pos != NULL))
    {
        Encontrado= ( e <= (Pos-> e));
        if (!Encontrado)
        {
            Ant = Pos;
            Pos = Pos->Sig;
        }
    }
    if (Encontrado)                                  /* se corto la busqueda hay que ver si esta en lista*/
        Encontrado = ((Pos-> e) == e);
    if (Encontrado)                                  /* si es verdadero hay que borrar*/
    {
        if (Ant == NULL)                             /* se borra en la primera posicion*/
            *L = Pos->Sig;
        else                                           /* borrado en centro o final de lista*/
            Ant->Sig = Pos->Sig;
        free(Pos);
    }
}

```

8.3. *Escribir funciones para: contar el número de nodos de una lista enlazada; eliminar el nodo que ocupa una posición en una lista enlazada; buscar el nodo en el que se encuentra almacenado un elemento; insertar en una lista enlazada un elemento conociendo el puntero inmediatamente anterior de donde hay que insertarlo.*

Análisis

Se usa la función `NuevoNodo` del ejercicio 8.2 y las declaraciones del ejercicio 8.1. La solución se codifica en las funciones:

- **NumeroDeNodosDeLaLista** Recorre la lista contando el número de nodos que tiene.
- **EliminaPosicion** Busca la posición del nodo que hay que borrar y en caso de que la encuentre, lo elimina teniendo en cuenta que puede ser el primero de la lista o no serlo.
- **InsertarLista**. Como conoce el nodo inmediatamente anterior lo único que necesita es añadir el nuevo nodo y mover dos punteros.
- **BuscarEnLista**. Busca en la lista la primera aparición del elemento en caso de que esté. En otro caso devuelve `NULL`.

Codificación

```

int NumeroDeNodosDeLaLista(Nodo *L)
{
    int k = 0;
    Nodo *p;
    p = L;
    while (p != NULL)
    {
        k++;
        p = p->Sig;
    }
    return(k);
}

void EliminaPosicion (Nodo** L, int i)
{
    int k = 0;
    Nodo *Ptr,*Ant;
    Ptr = *L;
    Ant = NULL;
    while ( (k < i) && (Ptr != NULL))
    {
        k++;
        Ant = Ptr;
        Ptr = Ptr->Sig;
    }
    if(k == i) /* borrado*/
    {
        if(Ant == NULL) /* se borra en la primera posición*/
            *L = Ptr->Sig;
        else
            Ant->Sig = Ptr->Sig;
        free(Ptr);
    }
}

void InsertarLista(Nodo* Ant,Telemento e)
{
    Nodo *Nuevo;
    Nuevo = (Nodo*)malloc(sizeof(Nodo));
    Nuevo -> e = e;
    Nuevo -> Sig = Ant -> Sig;
    Ant -> Sig = Nuevo;
}

Nodo* BuscarEnLista (Nodo* L, Telemento e)
{
    Nodo *Ptr;
    for (Ptr = L; Ptr != NULL; Ptr = Ptr ->Sig )
    if (Ptr-> e == e)
        return Ptr;
    return NULL; /* no encontrado*/
}

```

- 8.4.** *Escribir una función que reciba como parámetro dos listas enlazadas ordenadas crecientemente y de como resultado otra lista enlazada ordenada que sea mezcla de las dos.*

Análisis

Para mezclar dos listas enlazadas, se usa un nodo ficticio apuntado por el puntero `Primero`, para asegurar que todos los elementos se insertarán al final de la lista (nunca en la primera posición) que será la mezcla. Para ello se lleva un puntero `Ultimo` que apunta siempre al último elemento de la lista que se está creando y que debido al nodo ficticio siempre existirá. Al final de la mezcla se elimina el Nodo ficticio. La mezcla de las dos listas se realiza avanzando con dos punteros `Puntero1` y `Puntero2` por las listas `L1` y `L2`. Un primer bucle `while` avanza o bien por `L1` o bien por `L2` insertando en la lista mezcla, dependiendo de que el dato más pequeño esté en `L1` o en `L2`, hasta que una de las dos listas se termine. Los dos bucles `while` posteriores se encargan de terminar de añadir a la lista mezcla los elementos que queden o bien de `L1` o bien de `L2`. Se usan las declaraciones del problema 8.1 y la función `NuevoNodo` del problema 8.2.

Codificación

```
void MezclarListasOrdenadas(Nodo *L1, Nodo *L2, Nodo **L3)
{
    Nodo *Puntero1,*Puntero2,*Primero,*Ultimo, *NodoNuevo;
    NodoNuevo = NuevoNodo(-32767);
    Primero = NodoNuevo;
    Ultimo = NodoNuevo;
    Puntero1 = L1;
    Puntero2 = L2;
    while (Puntero1 && Puntero2)
        if (Puntero1->e < Puntero2->e)
        {
            NodoNuevo = NuevoNodo(Puntero1->e);
            Ultimo->Sig = NodoNuevo;
            Ultimo = NodoNuevo;
            Puntero1 = Puntero1->Sig;
        }
        else
        {
            NodoNuevo = NuevoNodo(Puntero2->e);
            Ultimo->Sig = NodoNuevo;
            Ultimo = NodoNuevo;
            Puntero2 = Puntero2->Sig;
        }
    while (Puntero1)
    {
        NodoNuevo = NuevoNodo(Puntero1->e);
        Ultimo->Sig = NodoNuevo;
        Ultimo = NodoNuevo;
        Puntero1 = Puntero1->Sig;
    }
    while (Puntero2)
    {
        NodoNuevo = NuevoNodo(Puntero2->e);
        Ultimo->Sig = NodoNuevo;
        Ultimo = NodoNuevo;
        Puntero2 = Puntero2->Sig;
    }
}
```

```

    L3 = Primero->Sig;                /*la lista comienza en el siguiente de Primero*/
    free(Primero);
}

```

- 8.5.** *Escribir una función que decida si una cadena de caracteres es subcadena de otra cadena. Ambas cadenas vienen dadas por listas enlazadas de caracteres.*

Análisis

La función `EsSubcadenaDeCadena` resuelve el problema. El cuerpo principal de la función se basa en lo siguiente: para cada carácter almacenado en la lista de `Cadena` (primer bucle `while`) decidir si todos los caracteres de `Subcadena` están almacenados en posiciones consecutivas de la posición que se trata de `Cadena` (segundo bucle `while`).

Codificación

```

#include <stdio.h>
struct Lista
{
    char ch;
    Lista *Sig;
};

int EsSubcadenaDeCadena (Lista *Subcadena, Lista *Cadena)
{
    Lista *Puntero1, *Puntero2;
    int Encontrado, Coincide;
    if (Cadena == NULL)
        return (Subcadena == NULL);
    else
        if (Subcadena == NULL)
            return 1;
        else
        {
            Encontrado = 0;
            while ((Cadena != NULL) && (! Encontrado))
            {
                if (Cadena->ch != Subcadena->ch)
                    Cadena = Cadena->Sig;
                else
                    /* Intento de búsqueda de coincidencia*/
                    {
                        Puntero1 = Cadena->Sig;
                        Puntero2 = Subcadena->Sig;
                        Coincide = 1;
                        while ((Puntero1 != NULL) && (Puntero2 != NULL) && Coincide)
                            if (Puntero1->ch == Puntero2->ch)
                            {
                                Puntero1 = Puntero1->Sig;
                                Puntero2 = Puntero2->Sig ;
                            }
                            else
                                Coincide = 0;
                        if (! Coincide)
                            Cadena = Cadena->Sig ;                /* vuelve a avanzar en cadena*/
                    }
            }
        }
    }

```

```

        else
            Encontrado = Puntero2 == NULL;
        }
    }
    return Encontrado;
}
}

```

PROBLEMAS AVANZADOS

- 8.6.** *Escribir un programa que permita sumar, restar, y multiplicar polinomios con una única variable mediante listas enlazadas ordenadas decrecientemente.*

Análisis

Los polinomios se presentan como listas enlazadas ordenadas descendientemente por el grado del polinomio. Los campos de la lista son `el.Exponente` y `Coeficiente` que almacenan el exponente y el coeficiente de cada término, y por supuesto el campo `Sig` para la implementación de la lista enlazada. El código se ha estructurado en las siguientes funciones:

- `AgregaTérmino`: agrega al polinomio cuyo primer elemento es `Primero` y cuyo último elemento es `Ultimo`, el término `el`.
- `SumaPolinomios`: suma los polinomios `p` y `q` dejando el resultado en `Suma`. Realiza una mezcla ordenada de dos listas ordenadas decrecientemente, teniendo en cuenta que cuando coinciden los exponentes hay que sumar los coeficientes de las dos términos insertarlos si la suma es distinta de cero y avanzar en ambas listas.
- `CambiaDeSigno`: cambia el signo de cada uno de los términos del polinomio `q`.
- `RestaPolinomios`: al polinomio `p` le resta el polinomio `q` y lo deja en `Resta`. Para ello cambia de signo al polinomio sustrayendo `q` y luego se lo suma al minuendo `p`.
- `MultiplicaPorMonomio`: multiplica el polinomio `p` por el `Monomio` y deja el resultado en el polinomio `Pro`. Para hacerlo basta con multiplicar todos los términos del polinomio `p` por el coeficiente de `Monomio` y sumar los exponentes de los términos de `p` con el de `Monomio`.
- `MultiplicaPolinomios`: multiplica el polinomio `p` por `q` y lo deja en `Producto`. Para hacerlo basta con ir multiplicando el polinomio `p` por cada uno de los monomios de `q` sumando los resultados en el polinomio `Producto`.
- `NuevoNodo` crea un nodo para un término de un polinomio que se le pase como parámetro.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct
{
    int Exponente;
    float Coeficiente;

```



```

} Telemento;
struct polinomio
{
    Telemento el;
    struct polinomio *Sig;
};
typedef struct polinomio Polinomio;
Polinomio* NuevoNodo(Telemento el)
{
    Polinomio *NodoNuevo ;
    NodoNuevo = (Polinomio*)malloc(sizeof(Polinomio));
    NodoNuevo -> el = el;
    NodoNuevo -> Sig = NULL;
    return NodoNuevo;
}

void AgregaTermino(Telemento el, Polinomio ** Primero, Polinomio** Ultimo)
    /*Agrega al polinomio con primer nodo Primero con último Ultimo el término el*/
{
    Polinomio *Aux;
    Aux = NuevoNodo(el);
    if (*Ultimo == NULL)
        *Primero = Aux;                                /*no hay elementos en la lista*/
    else
        (*Ultimo)->Sig = Aux;
    *Ultimo = Aux;
}

void SumaPolinomios(Polinomio *p, Polinomio *q, Polinomio ** Suma)
{
    /*fecto suma los polinomios p y q y lo deja en Suma. Modifica Suma*/
    float c;
    Telemento e;
    Polinomio *Primero, *Ultimo;
    Primero = NULL;
    Ultimo= NULL;                                /* se crea la lista vacia de elementos*/
    while ((p != NULL) && (q !=NULL))
        if (p->el.Exponente == q->el.Exponente)
        {
            c = p->el.Coeficiente + q->el.Coeficiente;
            if (fabs(c) > 0.00001 )
            {
                /* c<>0 hay que añadir un nuevo elemento a la suma de polinomios*/
                e.Coeficiente = c;
                e.Exponente = p->el.Exponente;
                AgregaTermino(e, &Primero, &Ultimo);
            }
            /*no necesita else el coeficiente es cero y no se añade a la suma*/
            p = p->Sig;
            q = q->Sig;                                /* se avanza en los dos polinomios*/
        }
    else
        if (p->el.Exponente > q->el.Exponente)
        {

```

```

        AgregaTermino(p->el, &Primero, &Ultimo);
        p = p->Sig;                                /* se avanza solo en el polinomio p*/
    }
    else
    {
        AgregaTermino(q->el, &Primero, &Ultimo);
        q = q->Sig;                                /* se avanza solo en el polinomio q*/
    }
    /* 0 bien el polinomio p o bien el polinomio q se han terminado*/
while (p != NULL)
{
    AgregaTermino(p->el, &Primero, &Ultimo);
    p = p->Sig;                                    /* se avanza solo en el polinomio p*/
}
while (q != NULL)
{
    AgregaTermino(q->el, &Primero, &Ultimo);
    q = q->Sig;                                    /* se avanza solo en el polinomio q*/
}
*Suma = Primero;
/* Se toma el primer elemento de la lista como el polinomio suma*/
}

void CambiaDeSigno( Polinomio **q)
{
    Polinomio *Aux;                                /*cambia de signo el polinomio q*/
    Aux = *q;
    while (Aux != NULL)
    {
        Aux->el.Coeficiente = -Aux->el.Coeficiente;
        Aux = Aux->Sig;
    };
    *q = Aux;
};

void RestaPolinomios(Polinomio *p, Polinomio *q, Polinomio **Suma)
{
    /*al polinomio p le resta el polinomio q y lo deja en resta*/
    CambiaDeSigno(&q);
    SumaPolinomios(p, q, Suma);
    CambiaDeSigno(&q);                            /* para no modificar el polinomio q*/
}

void MultiplicaPorMonomio(Polinomio *p, Polinomio *Monomio, Polinomio **Pro)
{
    /*multiplica el polinomio p por el monomio y deja el resultado en pro*/
    Telemento x, y, z;
    Polinomio *Aux, *Primero, *Ultimo;
    Primero = NULL;
    Ultimo = NULL;
    Aux = p;
    y = Monomio->el;
    while (Aux != NULL)
    {
        x = Aux->el;

```

```

        z.Exponente = x.Exponente + y.Exponente;
        z.Coeficiente = x.Coeficiente * y.Coeficiente;
        AgregaTermino(z, &Primero, &Ultimo);
        Aux = Aux->Sig;
    }
    *Pro = Primero;
}

void MultiplicaPolinomios(Polinomio *p, Polinomio *q, Polinomio ** Producto)
{
    /*Efecto multiplica p por q y lo deja en Producto. Modifica Producto*/
    Polinomio *Aux, *Aux1, *ProductoAuxiliar;
    *Producto = NULL;
    Aux = p;
    Aux1 = q;
    while (Aux1 != NULL)
    {
        MultiplicaPorMonomio(Aux, Aux1, &ProductoAuxiliar);
        SumaPolinomios(*Producto, ProductoAuxiliar, Producto);
        Aux1 = Aux1->Sig;
    }
}

```

- 8.7.** *Añadir al ejercicio 8.6 funciones para dividir dos polinomios, calcular el valor de un polinomio en un punto, leer un polinomio, y calcular el polinomio derivada.*

Análisis

Las funciones pedidas en el ejercicio son las siguientes:

- **DividePolinomios** que divide el polinomio p por el polinomio q y deja el cociente y el resto en los polinomios Cociente y Resto. Usa el algoritmo clásico de división de polinomios.
- **Valor** que evalúa el polinomio p en el punto x. Calcula el valor numérico del polinomio.
- **Deriva** que deriva simbólicamente el polinomio p y deja el resultado en el propio p según las reglas clásicas de derivación.
- **InsertarOrden** que inserta el elemento el en el polinomio manteniéndolo ordenado descendientemente, suponiendo que no hay términos repetidos.
- **LeePolinomio** que lee el polinomio p de la entrada. Se supone el polinomio no está necesariamente ordenado pero que no hay términos con mismo exponente.

Codificación

```

/*divide el polinomio p por el polinomio q y obitene el Cociente y el Resto*/
void DividePolinomios(Polinomio *p, Polinomio *q, Polinomio **Cociente,
                     Polinomio **Resto)
{
    Polinomio *Dividendo, *Divisor, *Multiplicando, *Suma, *Primero, *Ultimo;
    Telemento x, y, z;
    int expdeDividendo, expdeDivisor;
    Dividendo = p;
    Divisor = q;
    Primero = NULL;
    Ultimo = NULL;
    if (Divisor == NULL)
        expdeDivisor = 32767;
}

```

```

else
{
    expdeDivisor = Divisor->el.Exponente;
    y = Divisor->el;
}
if (Dividendo == NULL)
    expdeDividendo = 0;
else
{
    expdeDividendo = Dividendo->el.Exponente;
    x = Dividendo->el;
}
while (expdeDividendo >= expdeDivisor)
{
    z.Exponente = x.Exponente - y.Exponente;
    z.Coeficiente = x.Coeficiente / y.Coeficiente;
    Multiplicando = NuevoNodo(z);
    MultiplicaPolinomios(Divisor, Multiplicando, &Suma);
    AgregaTermino(z, &Primero, &Ultimo);
    RestaPolinomios(Dividendo, Suma, &Dividendo);
    if (Dividendo == NULL)
        expdeDividendo = 0 ;
    else
    {
        expdeDividendo = Dividendo->el.Exponente;
        x = Dividendo->el;
    }
}
*Cociente = Primero;
*Resto = Dividendo;
}

/*Evalúa el polinomio p en el punto x*/
float Valor(Polinomio *p, float x)
{
    Polinomio *Aux;
    float Suma;
    Telemento z;
    Suma = 0;
    Aux = p;
    while (Aux != NULL)
    {
        z = Aux->el;
        Suma = Suma + z.Coeficiente * pow(x, z.Exponente);
        Aux = Aux->Sig;
    }
    return( Suma);
}

/*Deriva el polinomio p simbólicamente y deja el resultado en el propio p*/
void Deriva(Polinomio **p)
{
    Polinomio *Aux, *Anterior;
    Telemento y;

```

```

int sw;
Aux = *p;
Anterior = NULL;
sw = 0;;
while (Aux !=NULL)
{
    y = Aux -> el;
    if (y.Exponente == 0)
    {
        /* Como el polinomio está ordenado Descendentemente hay que borrar el
           ultimo elemento de p ya que la derivada de una constante es cero*/
        if (Anterior == NULL)
            *p = NULL;
        else
            Anterior->Sig = NULL;
        Anterior = Aux;
        Aux = NULL;
        sw = 1;
    }
    else
    {
        y.Coefficiente = y.Coefficiente * y.Exponente;
        y.Exponente --;
        Aux->el = y;
        Anterior = Aux;
        Aux = Aux->Sig ;
    }
    if (sw)
        free(Anterior);
}
}

/*Inserta e en la lista manteniéndola ordenada descendientemente*/
void InsertarOrden(Polinomio **L, Telemento e)
{
    int Encontrado;
    Polinomio *Nuevo, *Aux, *Anterior;
    Nuevo=NuevoNodo(e);
    Aux=*L;
    Encontrado=0;
    while ((Aux != NULL) && (! Encontrado))
        if (Aux->el.Exponente > e.Exponente)
        {
            Anterior = Aux;
            Aux = Aux->Sig;
        }
        else
            Encontrado = 1;
    if (Aux == *L)
    {
        Nuevo->Sig = *L;
        *L = Nuevo;
    }
    else
    {

```

```

        Nuevo->Sig = Aux;
        Anterior->Sig = Nuevo;
    }
}

/*Lee el polinomio p no necesariamente ordenado pero sin términos repetidos*/
void LeePolinomio(Polinomio **p)
{
    Telemento x;
    *p = NULL;
    do
    {
        scanf("%f %d", &x.Coeficiente, &x.Exponente);
        if (x.Exponente >= 0)
            InsertarOrden(p, x);
    }
    while (x.Exponente >= 0);
}

```

- 8.8.** *Escribir una función que reciba como parámetro una lista enlazada de números enteros así como su longitud y nos devuelva la lista ordenada, moviendo solamente punteros.*

Análisis

La solución planteada usa el método clásico de la burbuja (intercambia datos contiguos) moviendo los punteros en lugar de la información. Consúltase el método de ordenación por burbuja del capítulo 5.

Codificación

```

#include <stdio.h>
struct lista
{
    int x;
    struct lista *Sig;
};
typedef struct lista Lista;
void ordena(Lista **L, int n)
{
    int i, j;
    Lista *Auxiliar, *Auxiliar1, *Anterior;
    for (i = 1 ; i < n; i++)
    {
        j = 1;
        Auxiliar = *L;
        Anterior = NULL;
        while (j <= (n - i))
        {
            Auxiliar1 = Auxiliar->Sig;
            if (Auxiliar->x > Auxiliar1->x)
            {
                Auxiliar->Sig = Auxiliar1->Sig;
                Auxiliar1->Sig = Auxiliar;
                if (Anterior == NULL)
                {

```

```

        *L = Auxiliar1;
        Anterior = *L;
    }
    else
    {
        Anterior->Sig = Auxiliar1;
        Anterior = Auxiliar1;
    }
    Auxiliar = Anterior->Sig;
}
else
{
    Anterior = Auxiliar;
    Auxiliar = Auxiliar1;
}
j++;
}
}
}

```

8.9. Implementar un programa con la siguiente estrategia de asignación de memoria en un TAD vector:

- Operación de creación del vector se ha de reservar memoria para $Mx(20)$ elementos.
- Operación de asignación de un elemento con la que se van añadiendo consecutivamente los nuevos elementos en la memoria reservada y si no hubiera espacio se amplía a otros Mx elementos.
- Operación de borrado de un elemento ha de ser de tal manera que un espacio equivalente al ocupado por el elemento quede libre para posteriores asignaciones.

Análisis

Se implementa el vector con los campos:

Comienzo: apunta al primer elemento de la lista enlazada (TAD vector) que contiene la memoria reservada.

Ultimo: apunta al último elemento que ha sido usado de la lista enlazada (TAD vector).

n que indica cuántos elementos han sido usados del TAD vector, previamente reservados.

De esta forma la codificación es sencilla. Se usan dos funciones, EncuentraDirección que encuentra la dirección del nodo que contiene un elemento y BuscaAnterior que busca la dirección del nodo anterior a uno dado que facilitan la comprensión de la función de borrado, así como la función ReservaMemoria, que se encarga de hacer la reserva de memoria de $Mx(20)$ elementos en $Mx(20)$ elementos.

La función AsignarUnElemento no comprueba que el elemento ya esté en el vector. La asignación se realiza de la siguiente forma: si el vector está vacío lo pone en Ultimo. En caso contrario lo pone en el siguiente de Ultimo, cambiando posteriormente el puntero Ultimo. Si no hubiera memoria disponible en el vector, previamente se reserva memoria.

La función BorrarUnElemento, busca el elemento y su anterior con la funciones correspondientes, y posteriormente borra el elemento del vector, cambiando los enlaces y colocándolo a partir de Ultimo.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#define Mx 20
typedef int Telemento ;
struct lista
{
    Telemento el;

```

```

    struct lista *Sgte;
};
typedef struct lista Lista;
typedef struct
{
    Lista *Comienzo, *Ultimo;
    int n;
} Tvector;

Lista * ReservaMemoria()
{
    Lista *Puntero1, *Puntero2;
    int i;
    Puntero2= (Lista*)malloc (sizeof(Lista));
    Puntero1=Puntero2;
    for (i = 2; i <= Mx; i++)
    {
        Puntero1->Sgte = (Lista*)malloc (sizeof(Lista));
        Puntero1 = Puntero1->Sgte ;
    }
    Puntero1->Sgte=NULL;
    return Puntero2;
}

void CreacionDelVector(Tvector *V)
{
    (*V).Comienzo = ReservaMemoria();
    (*V).Ultimo = NULL;    (*V).n = 0;
}

void AsignarUnElemento(Tvector *V, Telemento T)
{
    if( (*V).Ultimo == NULL)
    {
        (*V).Comienzo->el = T;
        (*V).Ultimo = (*V).Comienzo;
    }
    else
    {
        if((((*V).n % Mx) == 0) && ((*V).Ultimo->Sgte ==NULL))
            (*V).Ultimo->Sgte = ReservaMemoria();
        (*V).Ultimo = (*V).Ultimo->Sgte;
        (*V).Ultimo->el = T;
    }
    (*V).n ++ ;
}

Lista* EncuentraDireccion (Tvector V, Telemento el)
{
    Lista *Puntero1;
    int Encontrado;
    Encontrado = 0; Puntero1 = V.Comienzo;
    while (! Encontrado && Puntero1 != NULL)

```



```

    {
        Encontrado = Puntero1->el == el;
        if (! Encontrado)
            Puntero1 = Puntero1->Sgte ;
    }
    return Puntero1;
}

Lista * BuscaAnterior(Tvector V, Lista *Puntero1)
{
    /*Se supone que Puntero1 siempre está en V*/
    Lista *Anterior;
    Anterior = V.Comienzo;
    if (Anterior != Puntero1)
        while (Anterior->Sgte != Puntero1)
            Anterior = Anterior->Sgte;
    return Anterior;
}

void BorrarUnElemento(Tvector *V, Telemento T)
{
    Lista *Puntero1, *Anterior, *Aux;
    Puntero1 = EncuentraDireccion(*V, T);
    if (Puntero1 != NULL)
    {
        Anterior = BuscaAnterior(*V, Puntero1);
        if (Puntero1 == (*V).Ultimo)
            (*V).Ultimo = Anterior ;
        else
        {
            if ((*V).Comienzo == Puntero1)
                (*V).Comienzo = (*V).Comienzo->Sgte;
            else
                Anterior->Sgte = Puntero1->Sgte;
            /*enlaza a partir de V.Ultimo el nodo a borrar Puntero1*/
            Aux = (*V).Ultimo->Sgte;
            (*V).Ultimo->Sgte = Puntero1;
            Puntero1->Sgte = Aux;
        }
        (*V).n--;
    }
}

```

- 8.10.** *Escribir un programa que lea dos números enteros, almacene la descomposición en factores primos de los dos números enteros en sendas listas simplemente enlazadas, guardando en cada nodo el Divisor y su número de veces. Posteriormente debe calcular el máximo común divisor y el mínimo común múltiplo de los dos números a partir de las dos listas anteriores.*

Análisis

La estructura de datos para almacenar la factorización en factores primos es una lista enlazada ordenada crecientemente por el campo divisor que contiene los campos `Divisor` y `Veces`. La función `AnadeLista` añade un nodo a la lista enlazada que es apuntada a su último nodo por `fc`. Además de crear el nodo y poner los campos en él, se encarga de mover el puntero `fc`. La función `FactorizaNumero`, recibe como parámetro un número entero y almacena en una lista enlazada la descomposición en factores primos teniendo en cuenta que el único posible factor primo par es el dos y el resto son impares.

La función `MaximoComunDivisor` recibe la descomposición en factores primos de dos números y calcula el máximo común divisor de los números tomando los factores comunes con de menor exponente. La función `MinimoComunMultiplo` recibe la descomposición en factores primos de dos números y calcula el mínimo común múltiplo de los números tomando los factores no comunes y comunes con el mayor exponente.

Codificación (se encuentra en la página web del libro)

PROBLEMAS PROPUESTOS

- 8.1. Se tiene una lista enlazada ordenada con claves repetidas. Realizar un procedimiento de inserción de una clave en la lista, de tal forma que si la clave ya se encuentra en la lista la inserte al final de todas las que tienen la misma clave.
- 8.2. Dos cadenas de caracteres están almacenadas en dos listas. Se accede a dichas listas mediante los punteros `L1`, `L2`. Escribir una función que devuelva la dirección en la cadena `L2` a partir de la cual se encuentra la cadena `L1`.
- 8.3. Dada una cadena de caracteres almacenada en una lista `L`. Escribir un subprograma que transforme la cadena `L` de tal forma que no haya caracteres repetidos.
- 8.4. Se quiere representar el tipo abstracto de datos conjunto de tal forma que los elementos estén almacenados en una lista enlazada. Escribir las funciones necesarias para implementar el TAD conjunto mediante listas. Debe contener los tipos de datos necesarios y las operaciones: conjunto vacío; añadir un elemento al conjunto; unión de conjuntos; intersección de conjuntos; diferencia de conjuntos. Los elementos del conjunto que sean de tipo cadena.
- 8.5. Escribir un programa en el que dados dos archivos de texto `F1`, `F2` se formen dos conjuntos con las palabras respectivas de `F1` y `F2`. Posteriormente encontrar las palabras comunes a ambos y mostrarlas por pantalla. Utilizar el TAD conjunto.
- 8.6. Escribir un programa que forme una lista ordenada de registros de empleados. La ordenación ha de ser respecto al campo entero `Suel`do. Con esta lista ordenada realizar las siguientes acciones: mostrar los registros cuyo sueldo S es tal que: $P_1 \leq S \leq P_2$; aumentar en un 7% el sueldo de los empleados que ganan menos de P euros; aumentar en un 3% el sueldo de los empleados que ganan más de P euros; dar de baja a los empleados con más de 35 años de antigüedad.
- 8.7. Se quiere listar en orden alfabético las palabras de que consta un archivo de texto junto con los números de línea en que aparecen. Para ello hay que utilizar una estructura multienlazada en la que la lista directorio es la lista ordenada de palabras. De cada nodo con la palabra emerge otra lista con los números de línea en que aparece la palabra en el archivo. Escribir un programa que resuelva el problema de leer el archivo y crear la estructura multienlazada.
- 8.8. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa para representar mediante listas un vector disperso. Y realizar las operaciones: suma de dos vectores dispersos; producto escalar de dos vectores dispersos.
- 8.9. Para representar un entero largo, de más de 30 dígitos, se usa una lista simplemente enlazada. Escribir un programa en el que se introduzcan dos enteros largos y se obtenga su suma, su diferencia, su producto y su cociente.
- 8.10. Escribir un programa que lea un texto, separe las palabras y las introduzca en una lista enlazada ordenada crecientemente indicando el número de veces que se repite cada palabra en el texto.
- 8.11. Escribir una función recursiva que inserte elementos en una lista enlazada ordenada crecientemente.
- 8.12. Escribir una función recursiva que reciba una lista enlazada y la de la vuelta. El último elemento se convierta en el primero, el penúltimo en el segundo, etc.
- 8.13. En un sistema multiprogramado, muchos procesos se pueden encontrar presentes en la memoria del ordenador. A medida que transcurre el tiempo, unos procesos llegan al sistema pidiendo memoria, y otros procesos finalizan liberando memoria, debido a esta situación es frecuente que la memoria del ordenador se fragmente en

huecos. Se debe controlar por tanto los huecos de la memoria que están libres. Una forma de controlar la memoria puede ser crear una lista de huecos libres. Se pide realizar: las declaraciones necesarias para implementar una lista de huecos libres, y una función para inicializarla; una función (`UbicarMemoria`) que se

llamará siempre que un proceso pida memoria. Una función (`LiberarMemoria`) que se llamará siempre que un proceso finalice y salga de la memoria.

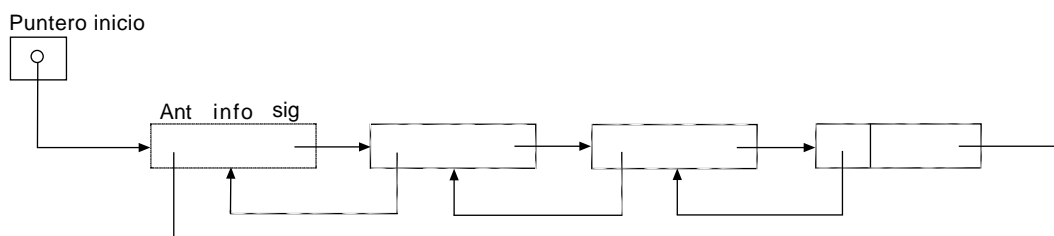
(La solución se encuentra en la página web del libro).

Modificaciones de listas enlazadas

Las listas enlazadas son estructuras muy flexibles y con numerosas aplicaciones en el mundo de la programación. Para algunas de estas aplicaciones es conveniente que las listas sean circulares, como es el caso de la implementación de una cola, o bien dobles, como es el caso del desarrollo de un menú desplegable. En este capítulo se estudian las listas doblemente enlazadas, las circulares y un ejemplo de listas multienlazadas para implementar una matriz dispersa.

9.1. Listas doblemente enlazadas

Las **listas doblemente enlazadas** se caracterizan porque su recorrido puede realizarse tanto de frente a final como de final a frente. Cada nodo de dichas listas consta de un campo con información y otros dos de tipo puntero (*Ant* y *Sig*) y será referenciado por dos punteros, uno de su nodo sucesor y otro del anterior. Es posible implementar este tipo de listas tanto con estructuras dinámicas de datos como con *arrays*, aunque lo más conveniente sea el empleo de las primeras. La diferencia fundamental entre listas simplemente enlazadas y bidireccionales es que, en éstas últimas, los nodos presentan un campo adicional, un apuntador al elemento anterior. Este campo adicional permite el recorrido de las listas hacia atrás y constituye el puntero al elemento anterior necesario para las eliminaciones.



EJEMPLO 9.1. Una lista doblemente enlazada con valores de tipo *int* necesita dos punteros y el valor del campo de datos. El siguiente código muestra como se agrupan estos datos en una estructura y la forma de, posteriormente, referenciar los miembros.

```
#include <stdio.h>
#include <stdlib.h>
```

```

typedef int Item;
struct unnodo
{
    Item dato;
    struct unnodo *adelante;
    struct unnodo *atras;
};

typedef struct unnodo Nodo;

int main()
{
    Nodo * p = NULL;
    p = (Nodo*)malloc(sizeof(Nodo));
    scanf("%d",&(*p).dato);
    (*p).adelante = NULL;
    (*p).atras = NULL;
    printf("%d\n",(*p).dato);
    return 0;
}

```

9.2. Inserción y borrado de un elemento en lista doblemente enlazada

En la operación de inserción será necesario tener en cuenta si se trata del primer elemento de la lista y, cuando esto no sea así, si el nuevo elemento ha de colocarse por delante del primero, en una posición intermedia o al final. Un algoritmo para añadir o insertar un elemento en una lista doblemente enlazada es el siguiente:

```

inicio
  <Buscar la posición donde insertar el nuevo elemento>
  <Asignar memoria para un elemento (NuevoPuntero) >
  NuevoPuntero->info ← elemento
  si la inserción se realiza en el comienzo de la lista entonces
    NuevoPuntero->Sig ← lista
    NuevoPuntero->Ant ← NULL
    Lista ← NuevoPuntero
  sino
    <en Anterior se debe tener la dirección del nodo que está antes del NuevoPuntero
    (siempre existe)>
    si la inserción se realiza al final de la lista entonces
      NuevoPuntero->Sig ← NULL
      NuevoPuntero->Ant ← Anterior
      Anterior->Sig ← NuevoPuntero
    sino
      <en Siguiente se debe tener la dirección del nodo que está después del NuevoPuntero
      (siempre existe)>
      NuevoPuntero->Sig ← Siguiente
      NuevoPuntero->Ant ← Anterior
      Anterior->Sig ← NuevoPuntero
      Siguiente->Ant ← NuevoPuntero
  fin si
fin si
fin

```

El borrado debe contemplar si se desea eliminar un elemento al principio de la lista en medio o al final y además la posibilidad de que la lista conste de un único elemento y quede vacía tras su eliminación. Un algoritmo para borrar un elemento en la lista enlazada es el siguiente:

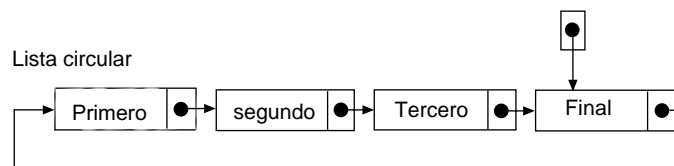
```

inicio
  <Buscar el posición pos del nodo que contiene el elemento a borrar>
  si la lista tiene un solo nodo que es el que hay que borrar entonces
    Lista ← NULL
  sino
    si el borrado se realiza al final de la lista entonces
      <en Anterior se debe tener la dirección del nodo que está antes de la posición pos
      (siempre existe)>
      anterior->Sig ← NULL
    sino
      si el borrado se realiza en el comienzo de la lista entonces
        <en Siguiente se debe tener la dirección del nodo que está después de la posición
        del nodo a borrar>
        Lista ← pos->Sig
        Siguiente->Ant ← NULL
      sino
        <en Anterior se debe tener la dirección del nodo que está antes de la posición pos
        (siempre existe)>
        <en Siguiente se debe tener la dirección del nodo que está después de la posición
        del nodo a borrar>
        Anterior->Sig ← pos->sig
        Siguiente->Ant ← pos->ant
    fin si
  fin si
  <liberar la memoria apuntada por pos>
fin si
fin

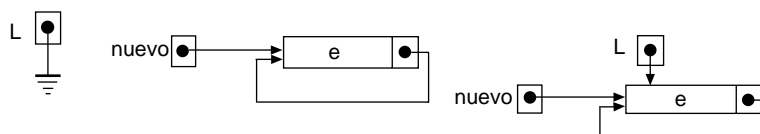
```

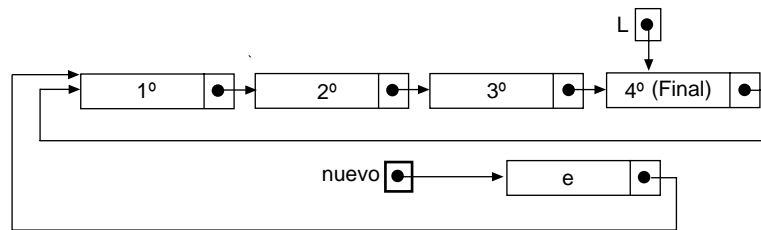
9.3. Listas circulares simplemente enlazadas

Estas listas son una modificación de las listas simplemente enlazadas en las que el puntero del último elemento apunta al primero de la lista. Para su implementación se considerará que el puntero externo referencia al último nodo de la lista y el siguiente es el primero.

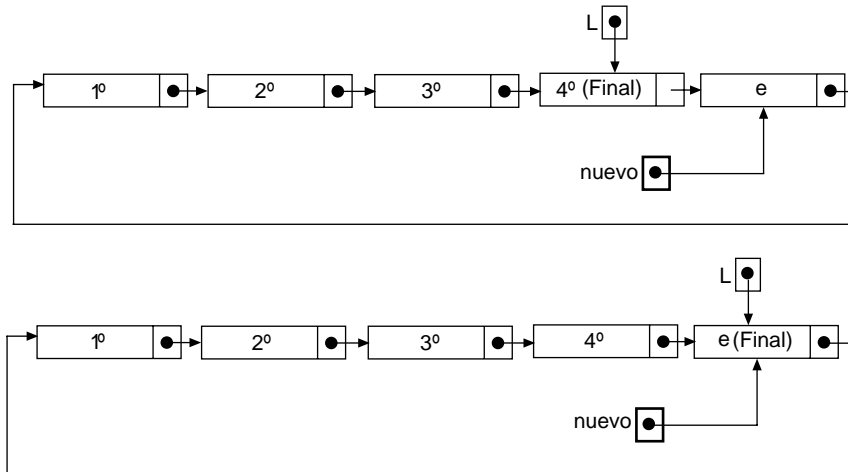


La inserción de datos en una lista simplemente enlazada circular implementada con estructuras dinámicas puede esquematizarse considerando los siguientes casos; cuando la lista está vacía la inserción del primer elemento será:





Si ya tiene elementos, para añadir otro nuevo:



EJEMPLO 9.2. El algoritmo empleado para añadir o insertar un elemento en una lista circular varía dependiendo de la posición en que se desea insertar el elemento. Suponer que se desea añadir nuevos elementos al final y que el nodo de acceso a la lista, *lc*, tiene la dirección del último nodo insertado. A continuación se escribe la declaración de un nodo, una función que crea un nodo y la función que inserta el nodo en la lista circular.

```
typedef char* Item;
typedef struct Elemento
{
    Item dato;
    struct Elemento* siguiente;
}Nodo;

Nodo* crearNodo(Item x)
{
    Nodo *a ;
    a = (Nodo*)malloc(sizeof(Nodo));
    a -> dato = x;
    a -> siguiente = a;          /* apunta así mismo, es un nodo circular */
    return a;
}

void insertaCircular(Nodo** lc, Item entrada)
{
    Nodo* nuevo;

    nuevo = crearNodo(entrada);
    if (*lc != NULL)
    {
        /* lista circular no vacía */
    }
}
```

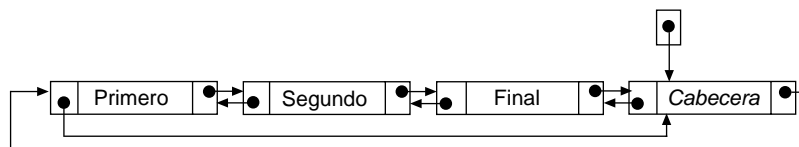
```

    nuevo -> siguiente = (*lc) -> siguiente;
    (*lc) -> siguiente = nuevo;
}
lc = nuevo;
}

```

9.4. Listas circulares doblemente enlazadas

Son listas circulares con punteros que permiten recorrerlas tanto en el sentido de avance de las agujas de un reloj como en el sentido contrario. Es útil y frecuente implementarlas con un nodo *cabecera*, permanentemente asociado a la existencia de la lista, que separa el principio del final de la misma y cuyo campo para almacenar información no se utiliza como el de los restantes elementos de la lista.



PROBLEMAS BÁSICOS

- 9.1.** *Escribir una función que reciba como parámetro un puntero a una Lista Doblemente Enlazada un valor almacenado en él, y elimine la primera aparición de ese elemento en la lista Doble y otra que elimine la primera aparición de un dato en una Lista Doblemente Enlazada y ordenada crecientemente.*

Análisis

En primer lugar se procede a buscar la primera aparición del elemento en la lista doblemente enlazada mediante una bucle `while` y una variable lógica `Encontrado`. Las dos búsquedas (de los dos elementos borrados) son muy similares, sólo se diferencian en que en una de ellas se puede salir del bucle cuando se está seguro de no poder encontrar el elemento. Una vez que el elemento se ha encontrado, se resuelve el problema del borrado al comienzo de la lista moviendo el puntero `*Ld` y si es necesario el puntero `Ptr->Sig`. Posteriormente se resuelve el problema de borrado en el centro de la lista, para lo cual hay que mover los punteros `Ptr->Ant->Sig` y `Ptr->Sig->Ant`. A continuación se resuelve el problema del borrado al final moviendo el puntero `Ptr->Sig`. Una vez realizados los enlaces, se libera la memoria. La parte de código de las dos funciones que realizan el borrado difieren en la implementación en el orden en que se realizan los enlaces.

Codificación

```

#include<stdio.h>
#include<stdlib.h>
typedef char Telemento;
struct listaD
{
    Telemento el;
    struct listaD *Ant, *Sig;
};
typedef struct listaD ListaD;
typedef struct

```



```

{
    ListaD *Cabecera, *Final;
} RegLD;

void EliminaLD(ListaD **Ld, Telemento el)
{
    ListaD* Ptr;
    int Encontrado = 0;
    Ptr = *Ld;
    while ((Ptr != NULL) && (!Encontrado))
    {
        Encontrado = (Ptr->el == el);
        if (!Encontrado)
            Ptr = Ptr -> Sig;
    }
    if (Ptr != NULL)
    {
        if (Ptr == *Ld)
        {
            *Ld = Ptr->Sig;
            if (Ptr->Sig != NULL)
                Ptr->Sig->Ant = NULL;
        }
        else
        {
            if (Ptr->Sig != NULL)
            {
                Ptr -> Ant -> Sig = Ptr -> Sig;
                Ptr -> Sig -> Ant = Ptr -> Ant;
            }
            else
            {
                Ptr -> Ant -> Sig = NULL; /* final*/
            }
        }
        free(Ptr);
    }
}

void BorrarEnOrdenLD(ListaD** Ld, Telemento el)
{
    ListaD *Ant, *Ptr;
    int Encontrado = 0;
    Ant=NULL;
    Ptr= *Ld;
    while ((!Encontrado) && (Ptr != NULL))
    {
        Encontrado = (el<=(Ptr->el));
        if (!Encontrado)
        {
            Ant = Ptr;
            Ptr = Ptr->Sig;
        }
    }
    if (Encontrado)

```

```

    Encontrado= ((Ptr->el) == el);
    if (Encontrado)
    {
        if (Ant==NULL)                                /* principio de lista*/
            if (Ptr->Sig == NULL)                      /* comienzo y final*/
                *Ld=NULL;
            else                                       /* comienzo y no final*/
            {
                Ptr->Sig->Ant = NULL;
                *Ld = Ptr->Sig;
            }
        else /* no comienzo*/
            if (Ptr->Sig == NULL)                      /* no comienzo y final*/
                Ant->Sig = NULL;
            else /* Centro*/
            {
                Ant->Sig = Ptr->Sig;
                Ptr->Sig->Ant = Ant;
            }
            free(Ptr);
    }
}

```

9.2 Escribir funciones para añadir y borrar datos en una lista doblemente enlazada ordenada de nombres con nodo cabecera y final.

Análisis

Se gestiona una lista ordenada con cabecera y final con un interruptor (*por_delante*) para decidir si se gestiona por delante o por detrás. Debido al nodo cabecera y final, las inserciones y borrados se realizan siempre en el centro de la lista. Para hacerlo se crean dos nodos ficticios con unos valores muy pequeño el cabecera y muy grande el final, almacenados en las variables globales *mini* y *maxi* respectivamente. Posteriormente se programan las funciones:

Esvacialista, decide si la lista está vacía lo que ocurrirá cuando sólo tenga los dos nodos el cabecera y el final.

CreaListaVacía, crea la lista vacía con los nodos cabecera y final.

Anade, que inserta un nombre en la lista doblemente enlazada ordenada realizando la búsqueda de la posición donde se debe insertar por la izquierda o derecha según indique la variable lógica *por_delante*.

sacar, elimina un nombre de la lista ordenada si se encuentra, devolviéndolo con un * en delante. La búsqueda del elemento a borrar se realiza comenzando por la izquierda o derecha dependiendo del valor del parámetro *por_delante*.

Codificación

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define mx 50
char maxi[mx], mini[mx];
struct pointer
{
    char Nombre[mx];
    struct pointer *Sig, *Ant;
};
typedef struct pointer puntero;
int Esvacialista(puntero *comienzo)

```

```

{
    puntero *pa;
    pa = comienzo->Sig;
    if (strcmp(pa->Nombre,maxi) == 0)                /*sólo tiene dos nodos*/
        return 1;
    else
        return 0;
}

void CreaListaVacía (puntero **comienzo,puntero** final)
{
    puntero *pa, *lista;
    pa = (puntero*)malloc(sizeof(puntero));          /*Nodo final*/
    strcpy(pa->Nombre,maxi);
    pa->Sig = NULL;
    lista = (puntero*)malloc(sizeof(puntero));        /*Nodo cabecera*/
    strcpy(lista->Nombre, mini);
    lista->Sig = pa;
    lista->Ant = NULL;
    pa->Ant = lista;
    *comienzo = lista;
    *final = pa;
}

void Anade (puntero *comienzo,puntero *final,char Nombre[mx],int por_delante)
{
    puntero *pa, *Ant, *pal;
    pa = (puntero*)malloc(sizeof(puntero));          /*Nodo a insertar*/
    strcpy(pa->Nombre, Nombre);
    if (por_delante)
    {
                                                /* búsqueda por delante*/
        pal = comienzo;
        while (strcmp(pal->Nombre,Nombre)<0)
        {
            Ant = pal;
            pal = pal->Sig ;
        }
        /*inserción*/
        Ant->Sig = pa;
        pa->Sig = pal;
        pa->Ant = Ant;
        pal->Ant = pa;
    }
    else
    {
                                                /*búsqueda por detrás*/
        pal = final;
        while (strcmp(pal->Nombre , Nombre) > 0)
        {
            Ant = pal;
            pal = pal->Ant;
        }
        /*inserción*/
        Ant->Ant = pa;
        pa->Sig = Ant;
    }
}

```

```

    pa->Ant = pal;
    pal->Sig = pa;
}
}

void sacar (puntero *comienzo,puntero *final,char Nombre[mx],int por_delante)
{
    puntero *pa, *Ant;
    if (por_delante)
    {
        pa = comienzo;
        while ((strcmp(pa->Nombre,Nombre) < 0) && (pa->Sig != NULL))
        {
            Ant = pa;
            pa = pa->Sig ;
        }
        if (strcmp(pa->Nombre,Nombre) == 0)
        {
            strcpy(Nombre,"*");
            strcat(Nombre, pa->Nombre);
            Ant->Sig = pa->Sig;
            pa->Sig->Ant = pa->Ant;
            free(pa);
        }
    }
    else
    {
        pa = final;
        while ((strcmp(pa->Nombre, Nombre)>0)&&(pa->Ant != NULL))
        {
            Ant = pa;
            pa = pa->Ant;
        }
        if (strcmp(pa->Nombre,Nombre) == 0)
        {
            strcpy(Nombre,"*");
            strcat(Nombre, pa->Nombre);
            Ant->Ant = pa->Ant;
            pa->Ant->Sig = pa->Sig;
            free(pa);
        }
    }
}

void main(void)
{
    strcpy(mini,"\0");
    strcpy(mini,">") ;
    .....
}

```

9.3. *Escribir funciones para generar aleatoriamente una lista circular simplemente enlazada y eliminar un elemento de una lista circular.*

Análisis

En primer lugar se realizan las declaraciones necesarias para tratar la Lista Circular. El programa se ha estructurado en funciones de la siguiente forma:

- *NuevoNodoLc* devuelve un puntero a un nuevo nodo en el que se almacena el dato *x*.
- *InsertaListaCircular* realiza la inserción en una lista circular del valor *dato*. Lo hace teniendo en cuenta que *Ultimo* es un puntero que apunta al último elemento que se añadió a la lista. Insertar un nuevo nodo en la Lista Circular como último elemento, para lo cual aparte de realizar los correspondientes enlaces, debe mover el puntero *Ultimo* para que apunte siempre al último elemento que se añadió. De esta forma el primer elemento de la lista siempre estará en el nodo *Ultimo->sig*.
- *GeneraPorElFinalLc* crea una lista circular de números enteros aleatorios, realizando las inserciones con la función *InsertaListaCircular*.
- *EliminarLc* se encarga de buscar la primera aparición de *dato* y borrarla de la lista circular. Lo hace de la siguiente forma. Si la lista está vacía no hay nada que hacer. En otro caso con una variable lógica *encontrado* y con un puntero *Puntero* realiza la búsqueda del dato, controlando no realizar un bucle infinito. Una vez encontrado el elemento se realiza el borrado teniendo en cuenta: si la lista contiene un sólo valor se quedará vacía; si el nodo a borrar es el apuntado por *Ultimo*, habrá que mover este puntero, en otro caso no habrá que moverlo. Siempre que se borre un nodo habrá que *Puntearlo*.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 1000
typedef int Telemento;
typedef struct NuevoNodo
{
    Telemento el;
    struct NuevoNodo* sig;
}NodoLc;

NodoLc* NuevoNodoLc(Telemento el)
{
    NodoLc *nn ;
    nn = (NodoLc*)malloc(sizeof(NodoLc));
    nn -> el = el;
    nn -> sig = nn;
    return nn;
}

void InsertaListaCircular(NodoLc ** Ultimo,Telemento el)
{
    NodoLc* nn;
    nn = NuevoNodoLc(el);
    if (*Ultimo != NULL)
    {
        nn -> sig = (*Ultimo) -> sig;
        (*Ultimo) -> sig = nn;
    }
    *Ultimo = nn;
}
```

```

void GeneraPorElFinalLc(NodoLc **Ultimo)
{
    Telemento d;
    NodoLc *p;
    p = NULL;
    randomize();
    for (d = random(MAX); d; )
    {
        InsertaListaCircular(&p,d);
        d = random(MAX);
    }
    *Ultimo=p;
}

void EliminarLc (NodoLc** Ultimo, Telemento el)
{
    NodoLc* Puntero,*p;
    int Encontrado = 0;
    Puntero = *Ultimo;
    if (Puntero == NULL)
        return;
    /* búsqueda mientras no encontrado y no de la vuelta*/
    while ((Puntero->sig != *Ultimo) && (!Encontrado))
    {
        Encontrado = (Puntero->sig->el == el);
        if (!Encontrado)
            Puntero = Puntero -> sig;
    }
    Encontrado = (Puntero->sig->el == el);          /* aquí se debe encontrar el dato*/
    if (Encontrado)
    {
        p = Puntero->sig;
        if (*Ultimo == (*Ultimo)->sig)             /* solo hay un dato*/
            *Ultimo = NULL;
        else
        {
            if (p == *Ultimo)
                *Ultimo = Puntero;
            Puntero->sig = p->sig;
        }
        free(p);
    }
}

```

9.4. *Escribir funciones para generar aleatoriamente una lista circular doblemente enlazada y eliminar un elemento de la lista .*

Análisis

Después de realizar las declaraciones de una lista doblemente enlazada, se implementa la función `GeneraPorElFinalLDC` que crea una lista circular doblemente enlazada de números enteros generados aleatoriamente, realizando las inserciones por el final de la lista circular. Esta inserción la realiza teniendo en cuenta que `Ultimo` es un puntero que apunta al último elemento que se añadió a la lista doblemente enlazada circular. Insertar un nuevo nodo en la lista doblemente enlazada circu-

lar como último elemento, para lo cual además de realizar los correspondientes enlaces, debe mover el puntero `Ultimo` para que apunte siempre al último elemento que se añadió. De esta forma el primer elemento de la lista doblemente enlazada circular siempre estará en el nodo `Ultimo->sig`. La función `EliminarLDc` se encarga de buscar la primera aparición de dato y borrarla de la lista circular. Lo hace de la siguiente forma. Si la lista esta vacía no hay nada que hacer. En otro caso con una variable lógica `encontrado` y con un *pointer* `Puntero` realiza la búsqueda del dato, controlando no realizar un bucle infinito. Una vez encontrado el elemento se realiza el borrado de tal forma que si la lista contiene un sólo valor se quedará vacía. Si el nodo a borrar es el apuntado por `Ultimo`, habrá que mover este puntero, en otro caso no habrá que moverlo. Siempre que se borre un nodo habrá que puentearlo.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define Max 1000
typedef int Telemento;
typedef struct NuevoNodo
{
    Telemento el;
    struct NuevoNodo* sig, *ant;
}NodoLDc;

void GeneraPorElFinalLDc(NodoLDc **Ultimo)
{
    Telemento dato;
    NodoLDc * NuevoN;
    *Ultimo = NULL;
    randomize();
    for (dato = random(Max); dato; )
    {
        NuevoN = (NodoLDc*)malloc(sizeof(NodoLDc));
        NuevoN -> el = dato;
        if (*Ultimo != NULL)
        {
            NuevoN -> sig = (*Ultimo) -> sig;
            NuevoN->ant = *Ultimo;
            (*Ultimo)->sig->ant = NuevoN;
            (*Ultimo) -> sig = NuevoN;
        }
        *Ultimo = NuevoN;
        dato = random(Max);
    }
}

void EliminarLDc (NodoLDc** Ultimo, Telemento el)
{
    NodoLDc* Puntero,*paborrar;
    int Encontrado = 0;
    Puntero = *Ultimo;
    if (Puntero==NULL)
        return;
    /* búsqueda mientras no encontrado y no de la vuelta*/
    while ((Puntero->sig != *Ultimo) && (!Encontrado))
```

```

    {
        Encontrado = (Puntero->sig->el == el);
        if (!Encontrado)
            Puntero = Puntero->sig;
    }
    Encontrado = (Puntero->sig->el == el);          /* aquí se debe encontrar el dato*/
    if (Encontrado)
    {
        paborrar = Puntero->sig;
        if (*Ultimo == (*Ultimo)->sig)             /* solo hay un dato*/
            *Ultimo = NULL;
        else
        {
            if (paborrar == *Ultimo)
                *Ultimo = Puntero;
            Puntero->sig = paborrar->sig;
            paborrar->sig->ant = Puntero;
        }
        free(paborrar);
    }
}

```

PROBLEMAS AVANZADOS

9.5. *Escribir un programa que permita la manipulación de una lista ordenada doblemente enlazada cuyos elementos sean una serie de letras mayúsculas, y que permita inicializar la lista, leer números enteros, k , del teclado y ejecutar las siguientes operaciones:*

si $k < 0$ el programa termina.

si $k = 0$ escribe la lista en orden descendente.

si $k = 1$ escribe la lista ordenada ascendentemente.

si $k \geq 2$ busca el k -ésimo elemento de la lista empezando por el menor y

- *Si éste elemento no existe (la lista tiene menos de k elementos), genera una letra de forma aleatoria y la inserta en el lugar adecuado para que la lista no pierda su ordenación.*
- *Si es una consonante suprime todas las consonantes que existan en la lista.*
- *Si es una vocal no hace nada.*

Análisis

La implementación que se realiza usa una lista doblemente enlazada ordenada crecientemente con nodos cabecera y final. Como la información que se almacena son letras mayúsculas, se puede colocar el espacio en blanco como elemento que marque el principio, pues el valor ordinal de este carácter es menor que el de cualquier mayúscula, y una minúscula como elemento que marque el final, ya que el valor ordinal de las minúsculas es siempre superior al de las mayúsculas. El proceso de iniciación (`InicializarListaConCabecera`) crea el nodo cabecera y el nodo final tal y como se ha indicado anteriormente. La función `RecorreHaciaLaDerecha` presenta la lista doblemente enlazada ascendentemente. La función `RecorreHaciaLaIzquierda` presenta la lista doblemente enlazada descendientemente. La función `BuscaEnLista` busca el elemento que ocupa la posición k en la lista doblemente enlazada e indica en caso de que exista el elemento

(Encontrado=True) la información que contiene. La función `InsertarEnListaOrdenada` inserta en una lista doblemente enlazada ordenada con nodo cabecera y final un elemento. La función `BorrarConsonantes` borra todas las consonantes de la lista doblemente enlazada con nodo cabecera y final. El programa principal genera aleatoriamente las letras y solicita la introducción, por parte el usuario, del valor de `k`, ejecutando según sea éste unas operaciones u otras, tal y como expresa el enunciado.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
struct ListaD
{
    char    el;
    ListaD *Ant, *Sig;
};
struct RegLD
{
    ListaD *Cabecera, *Final;
};

void InicializarListaConCabecera(RegLD *Ld)
{
    (*Ld).Final = (ListaD*)malloc(sizeof(ListaD));
    (*Ld).Cabecera = (ListaD*)malloc(sizeof(ListaD));
    (*Ld).Final->el = 'a';           /*las minúsculas son mayores que cualquier mayúscula*/
    (*Ld).Final->Ant = (*Ld).Cabecera;
    (*Ld).Final->Sig = NULL;
    (*Ld).Cabecera->el = ' ';       /*el blanco es menor que las mayúscula*/
    (*Ld).Cabecera->Ant = NULL;
    (*Ld).Cabecera->Sig = (*Ld).Final;
}

void RecorreHaciaLaDerecha(RegLD Ld)
{
    ListaD *Actual;
    Actual = Ld.Cabecera->Sig;
    while (Actual != Ld.Final)
    {
        printf("%3c",Actual->el);
        Actual = Actual->Sig;
    }
}

void RecorreHaciaLaIzquierda(RegLD Ld)
{
    ListaD *Actual;
    Actual = Ld.Final->Ant;
    while (Actual != Ld.Cabecera)
    {
        printf("%3c",Actual->el);
        Actual = Actual->Ant;
    }
}
```

```
void BuscaEnLista(RegLD Ld, int k, char *c, int *Encontrado)
{
    ListaD *Actual;
    int Contador = 1;
    Actual = Ld.Cabecera->Sig;
    while ((Contador < k) && (Actual != Ld.Final))
    {
        Actual = Actual->Sig;
        Contador ++;
    }
    *Encontrado = (Contador == k) && (Actual != Ld.Final);
    if (*Encontrado)
        *c = Actual->el;
}

void InsertarEnListaOrdenada(RegLD Ld, char e)
{
    ListaD *nuevo, *Actual;
    Actual = Ld.Cabecera->Sig;
    while (Actual->el < e)          /*busca la posición de inserción en orden ascendente*/
        Actual = Actual->Sig;
    /*hay que Insertar en la lista ordenada a partir del puntero Actual*/
    nuevo = (ListaD*)malloc(sizeof(ListaD));
    nuevo->el = e;
    nuevo->Sig = Actual;
    nuevo->Ant = Actual->Ant;
    Actual->Ant = nuevo;
    nuevo->Ant->Sig = nuevo;
}

int EsUnaVocal (char e)
{
    return(e == 'A' || e == 'E' || e == 'I' || e == 'O' || e == 'U');
}

void BorrarConsonantes(RegLD Ld)
{
    ListaD *Actual, *Auxiliar;
    Actual = Ld.Cabecera->Sig;
    while (Actual != Ld.Final)
    {
        if (! EsUnaVocal(Actual->el))
        {
            Auxiliar = Actual; Actual = Actual->Sig;          /* suprime la consonante*/
            Auxiliar->Ant->Sig = Auxiliar->Sig ;
            Auxiliar->Sig->Ant = Auxiliar->Ant;
            free(Auxiliar) ;
        }
        else
            Actual = Actual->Sig ;
    }
}
```

```

void main (void)
{   RegLD Ld;
    int k, Encontrado;
    char c;
    InicializarListaConCabecera(&Ld);
    printf("valor de k");
    scanf("%d",&k);
    randomize();
    while (k >= 0)
    {
        switch (k)
        {
            case 0: RecorreHaciaIzquierda(Ld);
                    break;
            case 1: RecorreHaciaDerecha(Ld);
                    break;
            default:
                BuscaEnLista(Ld, k, &c, &Encontrado);
                if (! Encontrado)
                { c = 'A' + random('Z' - 'A' + 1);
                  InsertarEnListaOrdenada(Ld, c);
                }
                else
                { if( Encontrado && (!EsUnaVocal(c)))
                  BorrarConsonantes(Ld);
                }
                printf("\n");printf("valor de k ");
                scanf("%d",&k);
            }
        }
    }
}

```

- 9.6.** *Escribir un programa que lea una sucesión de números enteros de un archivo de texto y cree una lista doblemente encadenada. Posteriormente lea un número entero de la entrada, y si está en la lista hacer una reestructuración de la lista de tal forma que dicho elemento sea la cabecera de la lista y su predecesor el final.*

Análisis

La lista se crea mediante el módulo `Crearlista`, estableciendo primeramente los enlaces en un sentido (anteriores), para posteriormente, establecer los del otro sentido (siguientes). Se tiene un `Frente` y un `Final` que apuntaran a ambos extremos de la lista. Cuando el dato numérico se lea de la entrada, la función `buscar` lo busca en la lista y en caso de que se encuentre en ella se cambian los punteros `Frente` y `Final`, y los de sus contenidos para que se cumpla la condición dada. Se programa además la función `escribelista` que se encarga de presentar la lista doble, y un programa principal que realiza la lectura de datos del archivo y las correspondientes llamadas a las funciones.

Codificación

```

#include<stdio.h>
#include<stdlib.h>
typedef struct listaD
{
    struct listaD *Ant, *Sig;
    int n;
} ListaD;

```

```

/*Esta función crea la lista doble; devuelve el puntero Frente al primer nodo, y el puntero Final
al último nodo de la lista*/
void Crearlista(ListaD **Frente, ListaD **Final, FILE *f)
{
    int    Numero;
    ListaD *p;
    *Final = NULL ;
    fscanf(f,"%d",&Numero);
    while (Numero != 0)
    {
        /*lee y crea solo los punteros de la izquierda*/
        p= (ListaD*)malloc(sizeof(ListaD));
        p->Ant = *Final;
        p->n = Numero;
        *Final = p;
        fscanf(f,"%d", &Numero);
    }
    *Frente = *Final;
    if (*Final != NULL)
    {
        (*Frente)->Sig = NULL;
        while ((*Frente)->Ant != NULL)
        {
            /*crea los punteros de la derecha*/
            (*Frente)->Ant->Sig = *Frente;
            *Frente = (*Frente)->Ant;
        }
    }
}

/*Búsqueda en la lista de un nodo con el dato n. Si encuentra el nodo este será el Frente y su
nodo anterior el Final*/
void buscar(ListaD **Frente, ListaD **Final, int n, int *s)
{
    /*La funcion transmite los punteros de Frente y Final*/
    ListaD *f1;
    int    t;
    f1 = *Frente;
    if (f1 == NULL)
    {
        *s = 1;
        printf(" la lista está vacía error");
    }
    else
    {
        t = 1;
        while (t)
        {
            if (f1->n == n)
            {
                /* reestructuración de la lista*/
                (*Final)->Sig = *Frente;
                (*Frente)->Ant = *Final;
                *Frente = f1;
                *Final = f1->Ant;
                (*Frente)->Ant = NULL;
                (*Final)->Sig = NULL;
                /*nuevo Frente*/
                /*nuevo Final*/
                /*anular los punteros antiguos*/
            }
        }
    }
}

```

```

        t = 0; *s = 0;
    }
    else
    {
        if (f1 == *Final)
        {
            *s = 2;
            t = 0;
        }
        else
        {
            f1 = f1->Sig;
        }
    }
}

void escribelista(ListaD *Auxiliar)
{
    printf("%10c", ' ');
    while (Auxiliar->Sig != NULL)
    {
        printf("%5d",Auxiliar->n);
        Auxiliar = Auxiliar->Sig;
    }
    printf("%5d\n", Auxiliar->n);
}

void main (void)
{
    ListaD *Frente, *Final, *Auxiliar;
    int ni, s;
    FILE *f;
    if ((f = fopen("datos.dat","wt")) == NULL)
    {
        printf(" error en archivo texto");
        exit(1);
    }
    Crearlista(&Frente, &Final,f );
    printf(" creada la lista cuyo orden es");
    Auxiliar = Frente;
    escribelista(Auxiliar);
    printf("\n dato");
    scanf("%d",&ni);
}

```

```

    buscar(&Frente, &Final, ni, &s);
    if (s == 1)
        printf("la lista está vacía error") ;
    else
    {
        printf(" dado el valor  %d  la nueva lista es", ni);
        if (s == 2)
        {
            printf(" dato no encontrado\n");
            s = 0;
        }
        else
        {
            Auxiliar = Frente;
            escribelista(Auxiliar);
            printf("\n");
        }
    }
    printf("\n");
    fclose (f);
}

```

- 9.7.** *Escribir un programa que permita tratar matrices esparcidas, mediante listas. Una matriz esparcida es aquella que tiene muchos datos iguales. En nuestro caso se supone que los datos iguales son el cero.*

Análisis

El programa que se presenta trata las matrices mediante listas multienlazadas enlazadas circulares. Los datos que almacenan los nodos contienen la siguiente información:

- **Info**, es la información que contiene la matriz.
- **F**, indica la fila en la que está almacenada **Info**.
- **C**, indica la columna en la que está almacenada **Info**.
- **SigF** (Siguiente Fila), es un puntero que apunta al siguiente nodo en la columna **C** que contiene información relevante en este caso distinta de cero.
- **SigC** (Siguiente Columna), es un puntero que apunta al siguiente nodo en la fila **F** que contiene información relevante en este caso distinta de cero.

Para facilitar el tratamiento:

Las filas y las columnas varían en un rango $1..MaxF$, y $1..MaxC$ respectivamente.

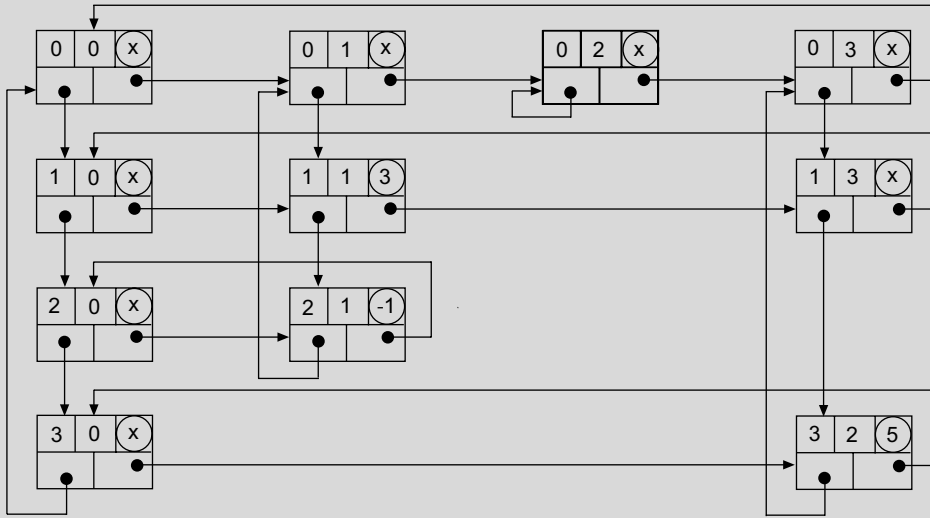
Se añade una fila ficticia, la fila 0, y una columna ficticia la columna 0, para facilitar las búsquedas. Los datos no relevantes de la matriz no se representarán y valdrán cero.

Por ejemplo la matriz dada por:

3	0	2
-1	0	0
0	0	5

Se representa de la siguiente forma:

Fila	Columna	Valor
SigF	SigC	



Las distintas funciones que se presentan para poder tratar las matrices esparcidas como matrices de la forma natural son:

- **EAF**. Es una función que recibe como parámetro una fila *F* y una columna *C* dentro de los rangos y devuelve un puntero al nodo de la fila *F* que precede inmediatamente a la posición donde debe estar un nodo en la columna *C* si éste estuviera en la matriz.
- **EAC**. Es una función que recibe como parámetro una fila *F* y una columna *C* dentro de los rangos y devuelve un puntero al nodo en la columna *C* que precede inmediatamente a la posición donde debe estar un nodo en la fila *F* si éste estuviera en la matriz.
- **Borrar**. Es una función que borra un nodo sabiendo los nodos antecesores en la fila y en la columna. Retorna el valor borrado.
- **Insertar**. Es una función que inserta un nodo con una información conociendo los nodos anteriores en la fila y en la columna correspondientes.
- **CreaMatriz**. Es una función que crea la matriz vacía. La fila cero y la columna cero ficticias.
- **Asignar**. Es una función que asigna un valor. $A[i][j]=x$.
- **Leer**. Es una función que retorna el valor de $A[i][j]$. Hay que observar que si el nodo que ocupa la posición *i*, *j* no está en la matriz retorna cero y en otro caso su contenido.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MaxF 4
#define MaxC 4
typedef int Telemento;
typedef struct mat
{
    Telemento Info;
    int F, C;
    struct mat *SigF, *SigC;
}Matriz;

Matriz* EAF(Matriz *A, int F, int C)
{
    /* 1<= F <= MaxF, 1 <= C <= MaxC */
    Matriz *P, *Pant, *Psig;
```

```

    P = A;
    while (P->F < F)
        P = P->SigF;                                /*P apunta a la fila F*/
    Pant = P;
    Psig = P->SigC;                                /*Pant va por detrás de P, y Psig por delante*/
    while ((P->C < C) && (Psig->C != 0))
    {
        /*Si la columna es 0, he dado la vuelta*/
        Pant = P;
        P = Psig;
        Psig = Psig->SigC;
    }
    if (P->C < C)
        return P;
    else
        return Pant;
}

Matriz* EAC(Matriz *A, int F, int C)
{
    /*Encuentra anterior en la columna 1 <= F <= MaxF, 1 <= C <= MaxC*/
    Matriz *P, *Pant, *Psig;
    P = A;
    while (P->C < C)
        P = P->SigC;                                /*P apunta a la columna C*/
    Pant = P;
    Psig = P->SigF;
    while ((P->F < F) && (Psig->F != 0))
    {
        Pant = P;
        P = Psig;
        Psig = Psig->SigF;
    }
    if (P->F >= F)
        return Pant;
    else
        return P;
}

void Borrar(Matriz *P, Matriz *Q, Telemento *x)
{
    /*Borra un nodo sabiendo que el anterior en la fila es P y en la columna es Q*/
    Matriz *Aux;
    Aux = P->SigC;
    P->SigC = Aux->SigC;
    Q->SigF = Aux->SigF;
    *x = Aux->Info;
    free(Aux);
}

void Insertar(Matriz *P, Matriz *Q, Telemento x)
{
    /*Inserta un nodo dados, P anterior en fila, Q anterior en columna*/
    Matriz *Aux;
    Aux = (Matriz*)malloc(sizeof(Matriz));
    Aux->Info = x;
    Aux->F = P->F;

```



```

    Aux->C = Q->C;
    Aux->SigC = P->SigC;
    Aux->SigF = Q->SigF;
    P->SigC = Aux;
    Q->SigF = Aux;
}

void CreaMatriz(Matriz ** A)
{
    /*Crea la matriz sin ningún dato*/
    int i;
    Matriz *Aux, *Aux1;
    Aux= (Matriz*)malloc(sizeof(Matriz));
    Aux->F = 0;
    Aux->C = 0;
    *A = Aux; /*nodo (0,0)*/
    for (i = 1; i <= MaxC; i++)
    {
        /*fila 0*/
        Aux1= (Matriz*)malloc(sizeof(Matriz));
        Aux1->F = 0;
        Aux1->C = i;
        Aux1->SigF = Aux1;
        Aux->SigC = Aux1;
        Aux = Aux1 ;
    }
    Aux->SigC = *A;
    Aux = *A;
    for (i = 1; i <= MaxF; i++)
    {
        /*Columna 0*/
        Aux1= (Matriz*)malloc(sizeof(Matriz));
        Aux1->F = i;
        Aux1->C = 0;
        Aux1->SigC = Aux1;
        Aux->SigF = Aux1;
        Aux = Aux1;
    }
    Aux->SigF = *A;
}

void Asignar(Matriz *A, int i, int j, Telemento x)
{
    /*A[i,j]=x*/
    Matriz *P, *Q;
    P = EAF(A, i, j);
    Q = EAC(A, i, j);
    if (P->SigC == Q->SigF)
    {
        /*está*/
        if (x == 0)
            Borrar(P, Q, &x);
        else
            P->SigC->Info = x ;
    }
    else
    {
        if (x != 0)
            Insertar(P, Q, x);
    }
}

```

```

Telemento Leer(Matriz *A, int i,int j)
{
    Matriz *P, *Q;
    P = EAF(A, i, j);
    Q = EAC(A, i, j);
    if (P->SigC == Q->SigF)
        return P->SigC->Info;
    else
        return 0;
}

```

9.8. Usando la matriz esparcida del ejercicio anterior, escribir una función que permita multiplicar una fila por un número y funciones que permitan, intercambiar filas o columnas, sumar o multiplicar matrices así como leer y escribir matrices.

Análisis

Para resolver el problema se usan las funciones definidas en el problema 9.7 EAC, EAF, y borrar para la función MultiplicaFila y las funciones Asignar y Leer para el resto. Las funciones solicitadas se han codificado de la siguiente forma:

- **MultiplicaFila.** Función que multiplica una fila F por un valor moviéndose por punteros en la matriz. Se observa que este producto puede producir un borrado de toda una fila si el valor es cero.
- **Suma.** Función que calcula la suma de dos matrices. Se realiza usando las funciones Asignar y leer.
- **Multiplica.** Función que calcula el producto de dos matrices usando las funciones Asignar y leer, utilizando el algoritmo clásico de multiplicación de matrices.
- **IntercambiaFilas.** Función que intercambia dos filas usando las funciones Asignar y leer.
- **IntercambiaColumnas.** Función que intercambia dos columnas usando las funciones Asignar y leer.
- **Leematriz.** Función que lee una matriz del teclado.
- **EscribeMatriz.** Función que escribe la matriz.

Codificación

```

/*Multiplica una fila f de la matriz por un valor*/
void MultiplicaFila(Matriz *A, int F, int Valor)
{
    Matriz *P, *Q, *R;
    Telemento x;
    P = EAF(A, F, 0);
    Q = P->SigC;
    if (Valor !=0)
        while (Q->C > 0)
        {
            Q->Info = Q->Info * Valor;
            Q = Q->SigC;
        }
    else
        while (Q->C > 0)
        {
            R = EAC(A, F, Q->C);
            Borrar(P, R, &x);
            Q = Q->SigC;
        }
}

```

```
void Suma(Matriz *A, Matriz *B , Matriz **C)
{
    int i, j;
    Telemento x;
    CreaMatriz(C);
    for (i = 1; i <= MaxF; i++)
        for (j = 1; j <= MaxC; j++)
        {
            x = Leer(A, i, j) + Leer(B, i, j);
            Asignar(*C, i, j, x);
        }
}

void Multiplica (Matriz *A, Matriz *B , Matriz **C)
{
    int i, j, k;
    Telemento x;
    CreaMatriz(C);
    for (i = 1; i <= MaxF; i++)
        for (j = 1; j <= MaxC; j++)
        {
            x=0;
            for (k = 1; k <= MaxC; k++)
                x+= Leer(A, i, k) * Leer(B, k, j);
            Asignar(*C, i, j, x);
        }
}

void IntercambiaFilas(Matriz *A, int f1, int f2)
{
    int j;
    Telemento x, y;
    for (j = 1; j <= MaxC; j++)
    {
        x = Leer(A, f1, j);
        y = Leer(A, f2, j);
        Asignar(A, f1, j, y);
        Asignar(A, f2, j, x) ;
    }
}

void IntercambiaColumnas(Matriz *A, int c1, int c2)
{
    int i;
    Telemento x, y;
    for (i = 1; i <= MaxC; i++)
    {
        x = Leer(A, i, c1);
        y = Leer(A, i, c2);
        Asignar(A, i, c1, y);
        Asignar(A, i, c2, x) ;
    }
}
```

```

void EscribeMatriz(Matriz *A)
{
    int i, j;
    Telemento x;
    for (i = 1; i <= MaxF; i++)
        for (j = 1; j <= MaxC; j++)
        {
            x = Leer(A, i, j);
            printf("A[%2d,%2d]=%3d\n", i,j,x);
        }
}

void LeeMatriz(Matriz **A)
{
    int i, j;
    Telemento x;
    CreaMatriz(A);
    for (i = 1; i <= MaxF; i++)
        for (j = 1; j <= MaxC; j++)
        {
            printf("dato %d %d ", i,j);
            scanf("%d",&x);
            Asignar(*A,i,j,x);
        }
}

```

PROBLEMAS PROPUESTOS

- 9.1. Sea L una lista doblemente enlazada que contiene caracteres. Escribir un subprograma que transforme la lista doblemente enlazada L de tal forma que en ella no aparezcan caracteres repetidos.
- 9.2. Se tiene una lista enlazada doblemente y ordenada con claves repetidas. Realizar una función de inserción de una clave en la lista, de tal forma que si la clave ya se encuentra en la lista la inserte al comienzo de todas las que tienen la misma clave.
- 9.3. Codificar el TAD lista ordenada mediante una lista doblemente enlazada sin emplear estructuras dinámicas.
- 9.4. Escribir un programa que lea un archivo de texto y cree una lista doblemente enlazada y ordenada con las palabras del texto. En los nodos de la lista doble se almacenará una palabra y un puntero a una lista simplemente enlazada circular que contengan los números de líneas en los cuales ha aparecido la palabra. Al final de la lectura del archivo, se debe escribir las palabras del texto original ordenadas crecientemente, indicando para cada una de ellas los números de línea en el que apareció en el archivo original.
- 9.5. Escribir una función que reciba como parámetro tres listas enlazadas y ordenadas y cree una lista doblemente enlazada circular con los nodos que aparecen en, al menos, dos de las tres listas, y otra lista simplemente enlazada circular con los nodos que aparecen en sólo una de ellas.
- 9.6. Implemente el TAD matriz dispersa (matriz con muchos ceros) mediante listas doblemente enlazadas.
- 9.7. Escriba un programa para gestionar un menú desplegable mediante listas doblemente enlazadas circulares.
- 9.8. Implemente mediante una lista circular doblemente enlazada el TAD cadena.

- 9.9.** Para representar un entero largo, de más de 30 dígitos, utilizar una lista circular teniendo el campo dato de cada nodo un dígito del entero largo. Escribir un programa en el que se introduzcan dos enteros largos y se obtenga su suma.
- 9.10.** 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 un puntero al primer nodo y otro al último nodo.
- 9.11.** 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 tiene la lista ordenarla alfabéticamente y escribirla por pantalla.
- 9.12.** Una lista circular de cadenas está ordenada alfabéticamente. El puntero `lc` tiene la dirección del nodo alfabéticamente mayor, y su siguiente apunta al nodo siguiente en el orden. Escribir una función para añadir una nueva palabra a la lista en el orden que le corresponda.

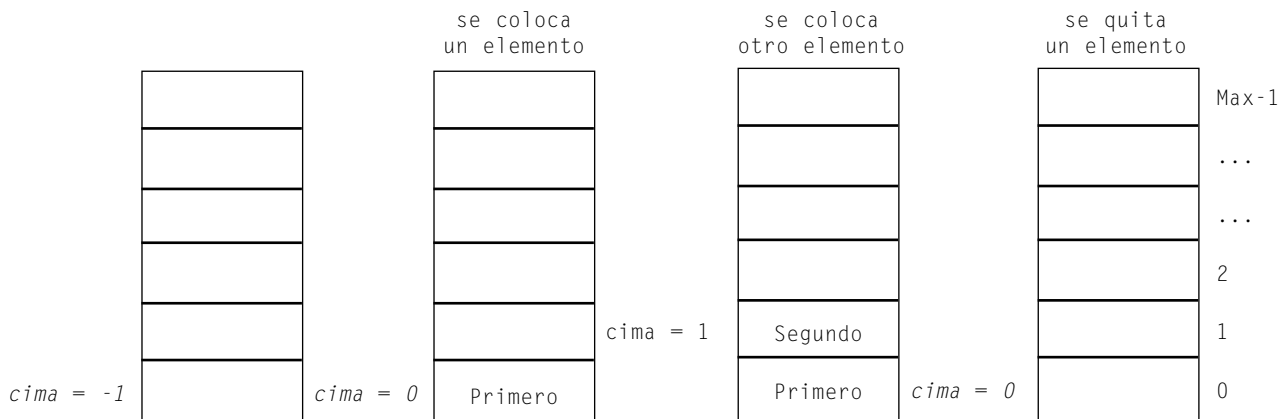
Pilas y sus aplicaciones

En este capítulo se estudia en detalle la estructuras de datos pila utilizada frecuentemente en los programas más usuales y en la vida diaria, la organización de una pila por todos es conocida. 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), todas las inserciones y extracciones de elementos se realiza por un mismo extremo denominado *cima* de la pila. El desarrollo las de pilas como tipos abstractos de datos LIFO es el motivo central de este capítulo. En el mismo se verá cómo utilizar el *TAD* pila para distintas aplicaciones como la de evaluar expresiones algebraicas o eliminar la recursividad.

10.1. El tipo abstracto de datos Pila

Una **pila** (*stack*) es una estructura de datos que cumple la condición: los elementos se añaden o quitan (borran) de la misma sólo por la parte superior (**cima**) de la pila. 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 *AnadeP* y *BorraP*. La operación **AnadeP** (*push*) añade un elemento en la cima de la pila y la operación **BorraP** (*pop*) elimina o saca un elemento de la pila. La pila se puede implementar mediante *arrays* en cuyo caso se limita el máximo número de elementos de la pila que puede contener y mediante punteros o listas enlazadas en cuyo caso se utiliza memoria dinámica y no existe limitación en su tamaño.

EJEMPLO 10.1. Colocación y eliminación de elementos en una pila implementada con arrays.



10.2. Especificación del Tipo Abstracto de datos Pila

La especificación formal del tipo abstracto de datos pila es la siguiente:

TAD Pila (VALORES: pila de elementos; OPERACIONES: VacíaP, EsVacíaP, AnadeP, BorraP, PrimeroP)

Sintaxis:

*VacíaP()	→ Pila
*AnadeP(Pila, elemento)	→ Pila
EsVacíaP(Pila)	→ Boolean
BorraP(Pila)	→ Pila
PrimeroP(Pila)	→ elemento

Semántica: para todo P de tipo Pila; para todo n de tipo elemento

EsVacíaP(VacíaP)	⇒ true
EsVacíaP(AnadeP(P, n))	⇒ false
PrimeroP(VacíaP)	⇒ error
PrimeroP(AnadeP(P, n))	⇒ n
BorraP(VacíaP)	⇒ error
BorraP(AnadeP(p, n))	⇒ P

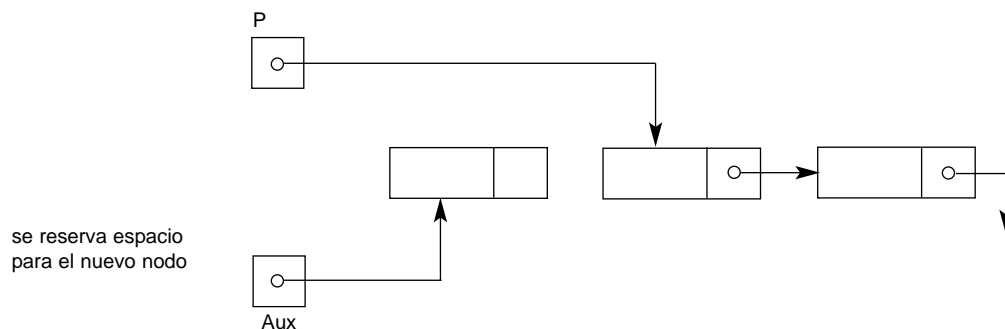
10.3. Implementación mediante estructuras estáticas

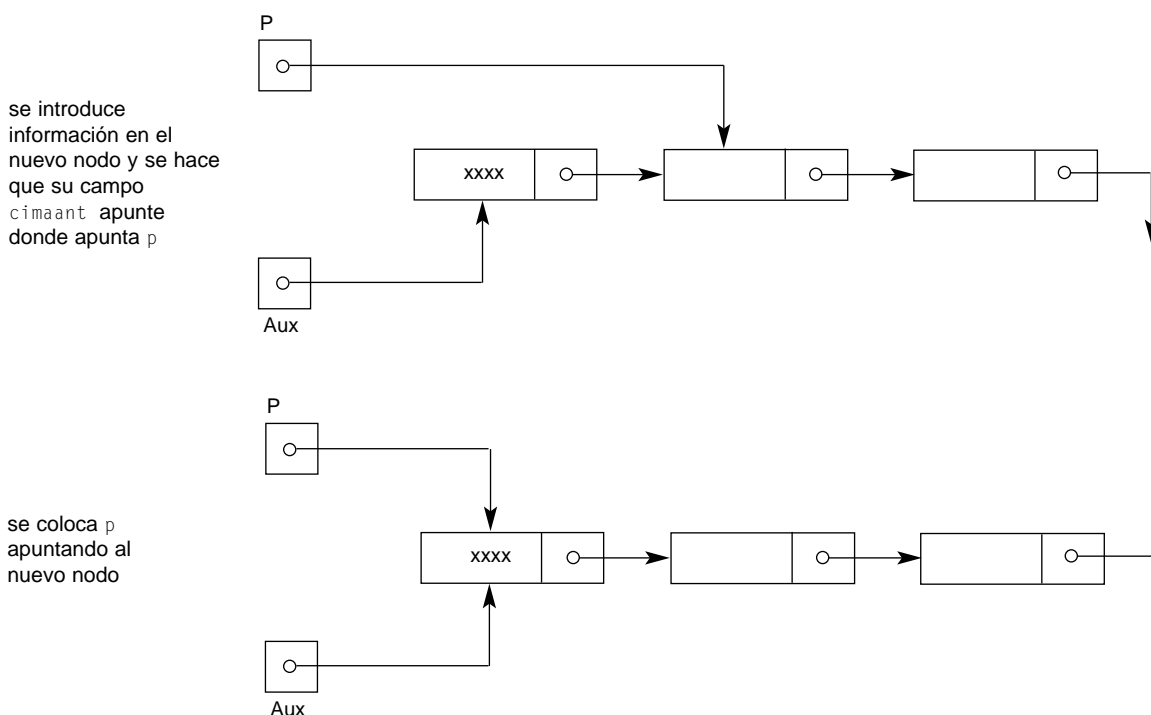
Si los datos se van a almacenar en un *array*, la estructura pila será una estructura compuesta por un *array* del tipo base de la pila (TipoElemento) y una variable entera, P.cima, que indica la posición del último elemento de la pila. El número máximo de elementos de la pila vendrá determinado por una constante (MaxPila). El función PilaVacía, cuando se trabaja con *arrays*, se limita a inicializar el entero P.cima a -1. La función lógica Vacía, indica si una pila está vacía, es decir, si su puntero P.cima = -1. Para insertar un elemento en la pila, simplemente se deberá incrementar P en una unidad y en la posición P.cima del *array* de elementos insertar el nuevo elemento. Hay que advertir que se puede dar una condición de error si no hay sitio en el *array*, lo que ocurrirá cuando P.cima sea igual a MaxPila-1. El único elemento accesible de la pila es el que ocupa la posición P.cima. La función PrimeroP se utiliza para recuperar dicho elemento. No se podrá recuperar si la pila está vacía. Por último la función BorraP se limita a decrementar la P.cima en una unidad. Una implementación con *array* puede consultarse en el Ejercicio Resuelto 3.8.

10.4. Implementación mediante estructuras dinámicas

Los elementos de la pila van a ser los nodos de una lista, con un campo para guardar el elemento y otro de enlace; la base para todo ello es la utilización de variables puntero. La pila se declara como un puntero a nodo, este puntero señala el extremo de la lista enlazada por el que se efectúan las operaciones. Siempre que se quiera poner un elemento se hace por el mismo extremo que se extrae. Una implementación de una pila con listas enlazadas puede consultarse en el Ejercicio Resuelto 10.1.

EJEMPLO 10.2. Representación gráfica de la inserción en una pila implementada mediante estructuras dinámicas de un nuevo elemento.





10.5. Transformación de expresiones aritméticas de notación infija a postfija

Las expresiones aritméticas pueden representarse en distintas notaciones. Las mas usadas son las siguientes:

- **Notación Infija.** Los operadores aparecen en el centro y los operandos a los lados como por ejemplo $a*(b+c)-d/e$. Para cambiar a prioridad de los operadores se usan los paréntesis.
- **Notación Postfija.** Los operadores aparecen detrás y los operandos delante. Por ejemplo, la expresión anterior en notación postfija es $abc-*de/-$. Tiene la ventaja que los operadores aparecen en el orden en que deben ser evaluados, y que para evaluar una expresión puede realizarse en una sólo pasada de izquierda a derecha usando una pila como estructura de datos auxiliar.
- **Notación prefija.** Los operadores aparecen delante de los operandos y en el orden en que deben ser evaluados.

EJEMPLO 10.3. Escritura de expresiones en notación postfija:

Según la definición establecida

la expresión $2 + 4 * 3 * (1 + 2)$ en notación *postfija* es $2\ 4\ 3\ *\ 1\ 2\ +\ *\ +$

la expresión $3 + 4 * 5 ^ (1 / 2)$ en notación *postfija* es $3\ 4\ 5\ 1\ 2\ /\ ^\ * +$

la expresión $6.25 * 4 + 5.7$ en notación *postfija* es $6.25\ 4\ *\ 5.7\ +$

Para transformar una expresión de notación infija a postfija se utiliza una pila y se sigue el siguiente algoritmo:

Se usa una pila por la cual que pasar todos los operadores. Se crea inicialmente la pila vacía. Los operandos pasan directamente a la salida. Los paréntesis abiertos pasan directamente a la pila de operadores. La llegada de un paréntesis cerrado obliga a extraer operadores de la pila y ponerlos en la salida hasta que salga el paréntesis abierto que no se escribe. Los operadores se van poniendo en la pila sucesivamente hasta encontrar uno con menor prioridad que el de la cima, en cuyo caso se sacan los que hubiera en la pila de mayor o igual prioridad pasándolos a la salida, y se coloca en ella éste último. Cuando se termina la entrada de datos se extraen todos los operadores de la pila y se pasan a la salida hasta que la pila se quede vacía. La prioridad de los operadores es la siguiente:

Tabla 10.1 Prioridad de operadores

Operador	Prioridad dentro de la pila	Prioridad fuera de la pila
^	3	4 (evaluación de derecha a izquierda)
*, /	2	2 (evaluación de izquierda a derecha)
+, -	1	1 (evaluación de izquierda a derecha)
(0	5 (operador especial)

Una codificación en C del algoritmo se encuentra en el Ejercicio Resuelto 10.2.

10.6. Evaluación de expresiones aritméticas

Una vez que una expresión aritmética está en notación postfija su evaluación se realiza usando una pila de valores en una sola pasada de izquierda a derecha mediante el siguiente algoritmo:

```

inicio
mientras <queden datos en la expresión postfija> hacer
    <tomar el primer elemento>
    si <es un operando> entonces
        <añadirlo a la pila>
    si_no
        <extraer y borrar el primer elemento de la pila X>
        <extraer y borrar el primer elemento de la pila Y>
        <evaluar el operando X y el operando Y con el operador y poner el resultado en la pila>
    fin si
fin mientras
<extraer y borrar el primer elemento de la pila. Este elemento es el valor de la expresión>
fin

```

Una codificación en C del algoritmo se encuentra en el Problema Resuelto 10.3.

10.7. Eliminación de la recursividad

Todo algoritmo recursivo puede ser transformado en otro de tipo iterativo, pero para ello a veces se necesita utilizar pilas donde almacenar los cálculos parciales y el estado actual del subprograma recursivo. Es posible estandarizar los pasos necesarios para convertir un algoritmo recursivo en iterativo, aunque el algoritmo así obtenido requerirá una posterior labor de optimización. Los pasos a seguir consisten en efectuar sobre el algoritmo recursivo las siguientes modificaciones:

Al principio del subprograma se añaden las instrucciones necesarias para: declarar pilas para las variables locales, los parámetros por valor, los parámetros por variable que se llaman con distintas variables y, en el caso de las funciones, también para los valores de la función que se vayan calculando; declarar una pila de direcciones o estados (almacenará etiquetas) para los retornos de las llamadas recursivas y una etiqueta, denominada por ejemplo *Final*, para colocarla en la última línea del subprograma. Inicializar la pila de direcciones la etiqueta *Final*. Posteriormente se inicializan todas las pilas a vacío, y se crea una etiqueta o estado, de nombre por ejemplo *E1*, y se marca con ella la primera sentencia del programa recursivo.

Sustituir cada llamada recursiva por lo siguiente:

Cuando se está transformando una función se sustituye la asignación de un valor a la función por la inserción de dicho valor en la pila creada para ello. Almacenar todos los parámetros y variables locales en sus pilas correspondientes. Crear una etiqueta o estado *EX* (la *X* será un número 2, 3, 4, ... que indicará si se trata de la 1ª, 2ª, 3ª, ... llamada recursiva) y almacenar *EX*, aún no colocada en el programa para marcar una línea, en la pila de direcciones. Actualizar los valores de los parámetros con los de una nueva llamada. Suprimir la llamada recursiva y colocar una instrucción de salto a *E1* en su lugar. Marcar con la etiqueta *EX* la siguiente línea del programa. Si se está transformando una función, recuperar el valor de la función en una variable, consultando el valor de la cima de la pila correspondiente. Quitar de las pilas los valores de los parámetros y variables locales y asignarlos a sus correspondientes variables (no se quita nada de la pila de direcciones).

Al final del subprograma se añaden instrucciones necesarias para recuperar una etiqueta de la pila de direcciones y ejecutar un goto a la etiqueta recuperada. Es decir, si la pila tiene en la cima una etiqueta EX, se desapila y ejecuta un goto EX (por ejemplo con el auxilio de switch). Por último, si es una función se le asigna la cima de la pila y si no se marca con Final el fin del subprograma.

PROBLEMAS RESUELTOS BÁSICOS

10.1. Escribir las primitivas de gestión de una pila implementada con una lista simplemente enlazada.

Análisis

Se define la pila como una lista simplemente enlazada. Posteriormente se implementan las primitivas:

- **VaciaP** crea la pila vacía poniendo la pila P a NULL.
- **EsvaciaP** indica cuándo estará la pila vacía. Ocurre cuando P valga NULL.
- **AnadeP** añade un elemento a la pila. Para hacerlo se añade un nuevo nodo que contenga como información el elemento que se quiere insertar y ponerlo como primero de la lista enlazada.
- **PrimeroP** en primer lugar se comprueba que la pila (lista) no esté vacía, y en caso de que así sea retorna el campo almacenado en el primer nodo de la lista enlazada.
- **BorrarP** elimina el último elemento que entró en la pila. Se comprueba que la pila no esté vacía en cuyo caso, se borra el primer nodo de la pila (lista enlazada).

Codificación

```
#include <stdio.h>
#include <stdlib.h>
typedef float TipoElemento;
typedef struct UnNodo
{
    TipoElemento e;
    struct UnNodo *sig;
}Nodo;
typedef Nodo Pila;

int EsVaciaP(Pila *P)
{
    return P == NULL;
}

void VaciaP(Pila** P)
{
    *P = NULL;
}

void AnadeP(Pila** P, TipoElemento e)
{
    Nodo * NuevoNodo;
    NuevoNodo = (Nodo*)malloc(sizeof(Nodo));
    NuevoNodo->e = e;    NuevoNodo->sig= (*P);
    *P = NuevoNodo;
```

```

}

TipoElemento PrimeroP(Pila *P)
{
    TipoElemento Aux;
    if (EsVacíaP(P))
    {
        puts("Se intenta sacar un elemento en pila vacía");    exit (1);
    }
    Aux = P->e;
    return Aux;
}

void BorrarP(Pila** P)
{
    Pila *NuevoNodo;
    if (EsVacíaP(*P))
    {
        puts("Se intenta sacar un elemento en pila vacía");
        exit (1);
    }
    NuevoNodo=(*P);
    (*P) = NuevoNodo->sig;
    free(NuevoNodo);
}

```

10.2. Escribir un programa que lea una expresión en notación infija y la pase a notación postfija.

Análisis

De acuerdo con lo explicado en la teoría se necesita una pila de caracteres para poder transformar la expresión de infija a postfija. Además de los operadores estándar y su prioridad se considera que el paréntesis abierto (es un operador ficticio que fuera de la pila tiene una prioridad muy alta (5 en este caso) y dentro de la pila una prioridad muy baja (0 en este caso) para obligar a que todos los operadores que están fuera de la pila entren a ella si el primero de la pila es el paréntesis abierto. Se programan dos funciones `PrioridadDentro` y `PrioridadFuera` que dan la prioridad de los operadores dentro y fuera de la pila. La función `operador` decide si un carácter es un operador. La función `postfija` recibe una expresión en notación infija y la pasa a postfija.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int PrioridadDentro(char opdor)
{
    switch (opdor)
    {
        case '(': return 0; case '^': return 3; case '/': return 2;
        case '*': return 2; case '+': return 1; case '-': return 1;
    }
    return 0;
}

```

```
int PrioridadFuera(char opdor)
{
    switch (opdor)
    {
        case '(': return 5;    case '^': return 4; case '/': return 2;
        case '*': return 2;    case '+': return 1; case '-': return 1;
    }
    return 0;
}

int Operador( char ch)
{
    return ( ch=='(' || ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='^' );
}

void Postfija(char Linea[80], char post[80])
{
    Pila *P;
    char ch, aux[2];
    int i, Apilado ;
    VacíaP(&P);
    aux[1] = '\0' ;
    post[0] = '\0';
    for (i = 0; i < strlen(Linea); i++)
        if (Operador(Linea[i]))
        {
            Apilado = 0;
            while (!Apilado)
                if (EsVacíaP(P))
                {
                    AnadeP(&P,Linea[i]);
                    Apilado = 1;
                }
            else
            {
                ch = PrimeroP(P);
                if(PrioridadDentro(ch) >= PrioridadFuera(Linea[i]))
                {
                    aux[0] = ch;
                    strcat(post,aux);
                    BorrarP(&P);
                }
                else
                {
                    AnadeP(&P,Linea[i]);
                    Apilado = 1;
                }
            }
        }
    else
        if (Linea[i] == ')')
        {
            ch = PrimeroP(P);
```

```

        BorrarP(&P);
        while (ch != '(')
        {
            aux[0] = ch;
            strcat(post,aux);
            ch = PrimeroP(P);
            BorrarP(&P);
        }
    }
    else
    {
        aux[0] = Linea[i];
        strcat(post,aux);
    }
    while(!EsVaciaP(P))
    {
        ch = PrimeroP(P);
        BorrarP(&P);
        aux[0] = ch;
        strcat(post,aux);
    }
}

void main (void)
{
    char post[80], Linea[80];
    puts(" expresion a pasar a Postfija\n");
    gets(Linea);
    Postfija(Linea,post);puts("\n Postfija\n");puts(post);
}

```

Si se ejecuta el programa anterior usando la pila del ejercicio 10.1 con el Telemento `char` se obtiene la siguiente salida:

```

expresion a pasar a postfija
a+b*(c-d)+e
Postfija
abcd-*+e+

```

10.3. Escribir una función que reciba una expresión en notación postfija y la evalúe.

Análisis

Para evaluar la expresión en notación postfija basta usar una pila de números reales y usando una función `valor` que calcule el valor de los operandos, programar el algoritmo indicado en la teoría.

Codificación

```

float valor(char c)
{
    return (float) c;
}

float evaluar ( char post[80])

```

```

{
    float valor1, valor2, valor3;
    int i;
    Pila *P;
    VacíaP(&P);
    for(i = 0; i < strlen(post); i++)
        if (operador(post[i]))
        {
            valor2 = PrimeroP(P);
            BorrarP(&P);
            valor1 = PrimeroP(P);
            BorrarP(&P);
            switch (post[i])
            {
                case '^' : valor3 = pow(valor1, valor2); break;
                case '/' : valor3 = valor1/valor2; break;
                case '*' : valor3 = valor1*valor2; break;
                case '+' : valor3 = valor1+valor2; break;
                case '-' : valor3 = valor1-valor2;
            }
            AnadeP(&P, valor3);
        }
        else
        {
            valor3 = valor(post[i]);
            AnadeP(&P, valor3);
        }
        valor1 = PrimeroP(P);
        BorrarP(&P);
        return valor1;
    }
}

```

PROBLEMAS RESUELTOS AVANZADOS

10.4. *Se dispone de un archivo de texto en el que se almacenan en cada línea números enteros positivos arbitrariamente grandes, que no pueden leerse directamente utilizando los tipos numéricos predefinidos. Utilizando estructuras dinámicas calcular la suma de todos los números.*

Análisis

Se usa una lista enlazada de dígitos como caracteres cuya declaración es omitida. Los números grandes se almacenan en la lista de tal manera que los dígitos de menor peso se encuentran al comienzo de la lista (estructura de pila). Sólo se usará la primitiva de gestión de pilas `anadeP` que añade un elemento a una pila. La solución se plantea mediante las siguientes funciones:

- `DestruyeLista` es una función que deja disponible en la memoria todos los elementos.
- `EscribeListaAlReves` escribe la lista al revés recursivamente.
- `SumarListaDeNumeros` recibe dos números almacenados en dos listas enlazadas, de tal manera que los dígitos de menor peso se encuentran al comienzo de la lista. El resultado de la suma de los dos números lo devuelve en el primer parámetro.

Para realizar la suma, se usa la estructura dinámica que implementa la lista, para ir sumando los dígitos de los números, de acuerdo con las reglas de la suma habitual. Para usar los operadores de suma, resta, módulo y división entera, se han de convertir, los caracteres, a números, y para poder almacenar el resultado en la lista se vuelve a convertir a caracteres.

- El programa principal, se encarga de preparar el archivo de entrada y llamar a las distintas funciones.

Codificación

```
void DestruyeLista(Lista **p)
{
    Lista *Auxiliar;
    while (*p != NULL)
    {
        Auxiliar = *p;
        (*p) = (*p)->sig;
        free(Auxiliar);
    }
}

void EscribeListaAlReves(Lista *p)
{
    /*escribe la pila al revés recursivamente*/
    if (p!=NULL)
    {
        EscribeListaAlReves(p->sig);
        printf("%c",p->el);
    }
}

void SumarListasDeNumeros (Lista **Total, Lista *Sumando)
{
    /* Total y sumando nunca son vacios*/
    int acarreo, el ;
    Lista *Nuevo, *TotalAux, *AntetiorTotal, *s;
    printf("Total parcial:");
    EscribeListaAlReves (*Total);
    printf("\n'Sumando: \n");
    EscribeListaAlReves(Sumando); printf("\n");
    acarreo = 0;
    TotalAux = *Total;
    AntetiorTotal = NULL;
    s = Sumando;
    while ((TotalAux != NULL) && (s != NULL))
    {
        /* suma de los dos*/
        el = TotalAux->el + s->el + acarreo - 2 * 48;
        TotalAux->el = el% 10 + 48;
        acarreo = el / 10;
        AntetiorTotal = TotalAux;
        TotalAux = TotalAux->sig; s = s->sig;
    }
    while (s !=NULL)
    {
        /* suma de s*/
        el = s->el - 48 + acarreo;
        Nuevo =(Lista*)malloc(sizeof(Lista));
        Nuevo->el = el % 10 + 48;
        acarreo = el / 10;
    }
}
```

```

    Nuevo->sig = NULL;
    AntetiorTotal->sig = Nuevo;
    AntetiorTotal = Nuevo;
    s = s->sig;
}
while (TotalAux !=NULL)
{
    /* suma de TotalAux*/
    el = acarreo + TotalAux->el - 48;          /* se convierte carácter a número*/
    acarreo = el / 10; TotalAux->el = el %10 + 48;
    AntetiorTotal = TotalAux; TotalAux = TotalAux->sig ;
}
if (acarreo == 1)
{
    Nuevo =(Lista*)malloc(sizeof(Lista));
    Nuevo->el = '1';
    Nuevo->sig = NULL;
    AntetiorTotal->sig = Nuevo;
}
printf("Nuevo Total Parcial: \n");
EscribeListaAlReves(*Total); printf("\n");
}

void main (void)
{
    Lista *Total, *Sumando;
    FILE *f;
    char el ;
    if ((f = fopen("numeros.dat","r+t")) == NULL)
    {
        puts("error de apertura texto");
        exit(1);
    }
    Total = NULL;
    Sumando = NULL;
    if (!feof(f))
    {
        el = getc(f);
        while (el != '\n') /*fin de linea*/
        {
            AnadeP(&Total,el);
            el= getc(f);
        }
        EscribeListaAlReves(Total);
        printf(" primer paso\n");
        while (!feof (f))
        {
            el = getc(f);
            while (el != '\n')          /* fin de linea*/
            {
                AnadeP(&Sumando,el);
                el = getc(f);
            }
            SumarListasDeNumeros (&Total, Sumando);

```



```

        DestruyeLista (&Sumando);
    }
}
else
    printf("Fichero de datos vacío");
fclose (f);
}

```

10.5. Eliminar la recursividad del problema de las Torres de Hanoi.

Análisis

El problema de las Torres de Hanoi tiene tres varillas *Origen*, *Destino* y *Auxiliar*. En la varilla *Origen* se alojan los n discos de tamaños diferentes que se pueden pasar de una varilla a otra libremente. Se trata de llevar los n discos de la varilla *Origen* a la varilla *Destino* utilizando las siguientes reglas: sólo se puede llevar un disco cada vez; un disco sólo puede colocarse encima de otro con diámetro ligeramente superior; si se necesita puede usarse la varilla *Auxiliar*.

Para resolver el problema basta con observar que si sólo hay un disco $n = 1$, entonces se lleva directamente de la varilla *Origen* a la varilla *Destino*.

Si hay que llevar $n > 1$ discos de la varilla *Origen* a la varilla *Destino*, entonces:

- Se llevan $n-1$ discos de la varilla *Origen* a la *Auxiliar*.
- Se lleva un sólo disco de la varilla *Origen* a la *Destino*.
- Se llevan los $n-1$ discos de la varilla *Auxiliar* a la *Destino*.

El programa codifica primeramente una función recursiva y usa una codificación de pilas para el tratamiento no recursivo en el que el *TipoDato* de la pila es el siguiente:

```

enum Estados {uno, dos} ;
struct elemento
{
    int n, d, h, u;    // n, origen, desde (d) destino, hasta (h) auxiliar usa(u)
    Estados estado;
};
typedef elemento TipoDato;

```

donde n, d, h, u son respectivamente el número de discos a llevar y las varillas *Origen*, *Destino* y *Auxiliar*. Los estados *uno* y *dos* representan las dos llamadas recursivas que tiene la función *Hanoi*.

La función que resuelve el problema de las Torres de Hanoi no recursivamente es *Hanoi nr*. Está basado en:

- En primer lugar se vacía la pila y se añade a la pila el primer elemento con los datos de llamada a la función.
- Un bucle itera mientras la pila no esté vacía, extrayendo la cima y borrando el elemento a tratar de la pila, para posteriormente tratarlo de acuerdo con el estado correspondiente.
- En el estado *uno* se hace lo siguiente. Primeramente se comprueba si se está en el caso trivial, de este modo, se lleva el disco correspondiente desde la varilla *Origen* a la varilla *Destino*. En el caso de no ser el caso trivial, se almacenan en la pila dos elementos que representan las dos llamadas recursivas. La primera con el estado *uno*, decrementando el número de discos en una unidad, e intercambiando las varillas *Destino* y *Auxiliar* que es como se llama a la función. La segunda indicando que hay que ir al *dos* con los mismos datos en el elemento en el resto de los campos.
- En el estado *dos* se da primeramente la orden de llevar el disco de la varilla *Origen* a la varilla *Destino* y se almacena en la pila el elemento con estado *uno*, decrementando el número de discos a llevar en una unidad e intercambiando las varillas *Origen* y *Auxiliar*.
- Se incluye también una versión recursiva de *Hanoi*. Se omite la declaración y las funciones de tratamiento de la pila.

Codificación

```
void Hanoi(int n, int d, int h, int u)
{
    if (n == 1)
        printf(" llevo disco %3d del palo %3d al palo %3d\n",n,d,h);
    else
    {
        Hanoi(n - 1, d, u, h);
        printf(" llevo disco %3d del palo %3d al palo %3d\n",n,d,h);
        Hanoi(n - 1, u, h, d);
    }
}

void HanoiNr(int n, int d, int h, int u)
{
    elemento e, e1;
    Pila *p;
    VacíaP(&p);
    e.n = n; e.d = d;
    e.h = h;
    e.u = u;
    e.estado = uno;
    AnadeP(&p, e);
    while (! EsVacíaP(p))
    {
        e = PrimeroP(p);
        BorrarP(&p);
        switch (e.estado)
        {
            case uno:
            {
                if (e.n == 1)
                    printf(" llevo disco %3d del palo %3d al palo %3d\n",e.n,e.d,e.h);
                else
                {
                    e1.n = e.n-1;
                    e1.estado = uno;
                    e1.d = e.d;
                    e1.h = e.u;
                    e1.u = e.h;
                    e.estado = dos;
                    AnadeP(&p, e);
                    AnadeP(&p, e1);
                }
                break;
            }
            case dos:
            {
                printf(" llevo disco %3d del palo %3d al palo %3d\n",e.n,e.d,e.h);
                e1.n = e.n - 1;
                e1.estado = uno;
                e1.d = e.u;
            }
        }
    }
}
```

```

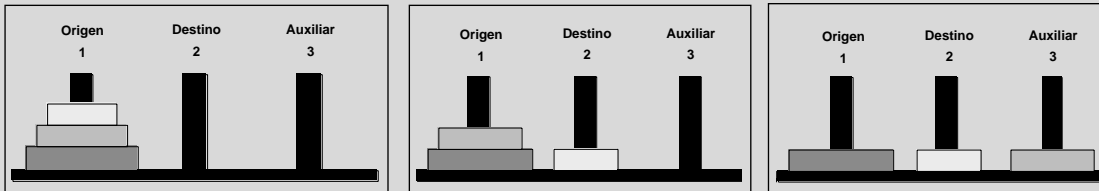
        e1.h = e.h;
        e1.u = e.d;
        AnadeP(&p, e1);
    }
}

void main(void)
{
    HanoiNr(4, 1, 2, 3);
}

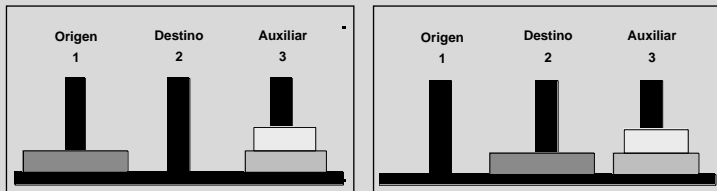
```

Una solución del problema de las Torres de Hanoi para el caso $n=3$ está dada por las siguientes figuras que pueden ser obtenidas mediante una función que represente gráficamente la información en lugar de escribir con la función `printf`.

Fases del Paso 1



Paso 2



10.6. Escribir un algoritmo que resuelva recursivamente e iterativamente el siguiente problema: se da un objetivo *Objetivo* y una colección de pesos p_1, p_2, \dots, p_n de enteros positivos. Se pide determinar si existe una selección de pesos que totalice exactamente el objetivo *Ob*. Por ejemplo si $Ob=12$ y los pesos son 7,5,4,1 se puede elegir el primero, el tercero y el cuarto, ya que su suma es $7+4+1=12$.

Análisis

La función recursiva `MochilaExacta` se plantea de la siguiente forma:

- Cada vez que se hace una llamada recursiva se incrementa el parámetro i en uno.
- Se termina con éxito cuando se obtiene en `Objetivo = 0`. Se Termina con fallo cuando `Objetivo < 0` o bien i coincide con n .
- Se plantean las dos posibilidades la inclusión y la exclusión del elemento número i con $0 \leq i < n$. En el caso de la inclusión `Posibilidad=1`, la llamada recursiva se plantea disminuyendo el `Objetivo` en `pesos[i]` y en el segundo caso no se disminuye `Objetivo` en ninguna unidad.

La función `MochilaExacta_N_R` se plantea de la siguiente forma:

- La variable `i` puede ser global e incrementarse en uno cada vez que se hace la llamada y decrementarse en uno cada vez que se retorna.
- Se mantiene la dirección de retorno de la recursividad mediante unos estados de la siguiente forma :
 - Ninguno : Indica que la llamada se hace desde fuera del modulo mochila. Es decir, siempre que se inicializa el código completo de la mochila.
 - Incluido: Indica que la llamada viene de la inclusión.
 - Excluido: Indica que la llamada viene de la exclusión.
- Objetivo puede manejarse como una variable global. Cuando el estado cambia de ninguno a incluido se resta `Pesos[i]` a `Objetivo`; de Incluido a excluido se suma `pesos[i]` a `Objetivo`.
- Por supuesto es necesario usar pilas para poder gestionar la eliminación de la recursividad. Se incluye sólo la declaración del `TipoDato`.

Codificación

```
#define n 5
enum Estados {Ninguno, Incluido, Excluido} ;
struct elemento
{
    Estados estado;
};
typedef elemento TipoDato;
int Pesos[n], OobjetivoAalcanzar;

void Inicializa()
{
    Pesos[0]= 7; Pesos[1] = 5; Pesos[2] = 4; Pesos[3] = 4; Pesos[4] = 1;
    OobjetivoAalcanzar = 10;
}

void MochilaExacta(int OobjetivoAalcanzar,int i, int *Solucion)
{
    int Encontrado, Posibilidad;
    /*Se termina con éxito siOobjetivo=0 o bien con fallo sio Objetivo<0 o bien i=n*/
    if (( OobjetivoAalcanzar <= 0) || (i == n))
        if (OobjetivoAalcanzar == 0 ) /*solución*/
            *Solucion = 1;
        else ; /*no hacer nada porque Objetivo<0 o bien i=n*/
    else
    {
        Posibilidad = 0;
        Encontrado = 0;;
        do
        {
            Posibilidad ++;
            switch (Posibilidad)
            {
                case 1: /*Inclusión*/
                    MochilaExacta(OobjetivoAalcanzar-Pesos[i], i+1, &Encontrado);
                    if( Encontrado)
                        printf("%d ",Pesos[i]); /*se ha incluido y se tiene solución*/
                    break;
            }
        }
    }
}
```

```

        case 2:                                     /*exclusión*/
            MochilaExacta(ObjetivoAalcanzar, i+1, &Encontrado);
            /*aunque Encontrado sea verdadero, no hace falta hacer nada*/
        }
    }
    while (!((Posibilidad == 2) || Encontrado));
    *Solucion = Encontrado;
}
}

void MochilaExacta_N_R(int ObjetivoAalcanzar)
{
    int i, Solucion;
    Pila *P;
    elemento e, el;
    i = 0;
    Solucion = 0;
    e.estado = Ninguno;
    VacíaP(&P);
    AnadeP(&P, e);          /*Asigna el valor inicial a la pila P considerando Pesos[0]*/
    while (! EsVacíaP(P))
    {
        e = PrimeroP(P);
        BorrarP(&P);
        if (Solucion)          /*si se tiene ya una solución entonces escribirla*/
        {
            if (e.estado == Incluido)
                printf("%d \n", Pesos[i]);          /*{solo se escribe algo si está incluido*/
            i --;          /*retrocede*/
        }
        else                    /*solucion es false*/
            if (((ObjetivoAalcanzar <= 0) && (e.estado == Ninguno)) || (i == n))
            {
                if (ObjetivoAalcanzar == 0)
                    Solucion = 1;          /*siempre se obtiene solución*/
                i --;          /*el i no se ha probado. Se retrocede en la recursividad*/
            }
        else                    /*no hay decisión se considera el estado del candidato actual*/
            switch (e.estado)
            {
                case Ninguno:          /*Primero se incluye avance de la recursividad*/
                    ObjetivoAalcanzar -= Pesos[i];
                    i ++;
                    el.estado=Incluido;
                    AnadeP(&P, el);
                    el.estado = Ninguno;
                    AnadeP(&P, el);          /* se va al comienzo*/
                    break;
                case Incluido:          /*Ahora se excluye avance de la recursividad*/
                    /*se retrocede i--; en opción else de q*/
                    ObjetivoAalcanzar += Pesos[i];
                    /*como se quitó hay que sumarlo*/
                    i ++;
            }
    }
}

```

```

        el.estado=Excluido;
        AnadeP(&P, e1);
        el.estado=Ninguno; AnadeP(&P, e1);
        break;
    case Excluido:
        /*la elección actual no dio resultado se ha terminado la recursividad*/
        i--;
    }
}

void main (void)
{
    int Solucion;
    Inicializa();
    MochilaExacta(ObjetivoAalcanzar, 0, &Solucion);
    if( Solucion)
        printf("Solución dada para Objetivo = %d \n", ObjetivoAalcanzar);
    Solucion = 0;
    ObjetivoAalcanzar = 10;
    MochilaExacta_N_R(ObjetivoAalcanzar);
    if (Solucion)
        printf(" se encontró solución");
}

```

10.7. Escribir una función recursiva sin parámetros y otra no recursiva que resuelva el problema de encontrar todas las soluciones al problema del caballo en un tablero de dimensiones $n \times n$ (n dato). (Problema 4.1).

Análisis

Para resolver el problema de encontrar todas las soluciones sin parámetros recursivamente, basta con:

- Los parámetros de la función recursiva x, y (coordenadas) e i , se convierten en variables globales. Como se obtienen todas las soluciones no es necesario tener ninguna variable booleana que nos indique si hay solución. Ver problema resuelto 4.1. ($x = Posx$ $y = Posy$).
- Las posibilidades del caballo son los ocho movimientos. Cada vez que se itera una posibilidad se calculan las nuevas coordenadas sobre las variables x e y ; y cuando se termina la posibilidad se vuelven a recalcular las coordenadas iniciales sobre las que se partió. Cada vez que se llama a la función recursiva la variable i se incrementa en una unidad y en la vuelta de la recursividad se decrementa en una unidad.
- Posibilidad aceptada es: las coordenadas estén dentro del tablero y en la posición correspondiente no se haya colocado antes el caballo. El anotar movimiento es poner en el tablero en las coordenadas correspondientes el número de caballo (i) que corresponde, y el borrar anotación, el restaurar el valor de cero.

Para la eliminación de la recursividad se usa una pila cuyos elementos tendrán dos campos: estado que puede tomar los valores uno y dos y k que puede tomar los valores $-1, 0, 1, 2, 3, 4, 5, 6, 7$.

- Un bucle `mientras` controlará el fin del proceso, extrayendo el primer elemento de la pila.
- La variable i se incrementará en una unidad tanto en la llamada recursiva, como en el caso de obtener una solución. De esta forma se puede unificar el retorno de la recursividad y la solución en un sólo estado que es el dos.
- En el estado dos, se realizarán las operaciones de borrar anotación, y además $x = x + DespX[Posibilidad]$ $y = y + DespY[Posibilidad]$ $i = i - 1$, siendo `Posibilidad` el valor almacenado en el elemento que se ha extraído de la pila. La última instrucción $i \leftarrow i - 1$ obliga a que aún cuando se encuentre una solución la variable i se incremente en una unidad tal y como se ha dicho previamente.

- En el estado uno se realizará el resto del proceso.
- Una variable `colocado` se pondrá a `true` cuando se encuentre un movimiento aceptable a partir del último que se almacenó en el campo `Posibilidad` que estaba almacenado en el elemento de la pila.
- Si `colocado` está a `false`, no hace falta hacer nada. Si se puso a `true`, entonces se anota el movimiento y
 - En el caso de que se esté ante una solución se añade a la pila un elemento con el estado `dos` y el último valor de `Posibilidad`.
 - En el caso de que no se esté ante una solución, hay que añadir a la pila tres elementos en el siguiente orden: El primero con estado uno y valor de `Posibilidad` el último que se tuvo para poder seguir con el siguiente movimiento. El segundo con estado `dos` y valor de `Posibilidad` el último que se tuvo para poder borrar la última anotación. El tercero con estado uno y con valor de `Posibilidad` cero para comenzar otra recursividad. Por supuesto el valor de `i` se incrementa en una unidad.

Codificación

```
#define nn 8
enum Estados {uno, dos} ;
struct elemento
{
    Estados Estado;
    int Posibilidad;
};
typedef elemento TipoElemento;
int Contador, i, j, x, y, DespX[8], DespY[8], Tablero[8][8], n ,nCuartado;
void Escribesolucion()
{
    int i, j;
    for(i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
            printf("%5d",Tablero[i][j]);
        printf("\n");
    };
    printf("solución %3d\n",Contador);
};

void ensayartodas()
{
    int Posibilidad = -1;
    do
    {
        Posibilidad++;
        x += DespX[Posibilidad];
        y += DespY[Posibilidad];
        if ((x < n) && (y < n) && (x >= 0) && (y >= 0))
            if (Tablero[x][y] == 0)
            {
                Tablero[x][y] = i;
                if (i < nCuadrado)
                {
                    i++;
                    ensayartodas();
                    i--;
                }
            }
    }
}
```

```

        else
        {
            Escribeolucion();
            Contador++;
        }
        Tablero [x] [y] = 0
    }
    x -= DespX[Posibilidad];
    y -= DespY[Posibilidad];
}
while(Posibilidad<8-1);
}

void ensayartodasnr()
{
    int Posibilidad,colocado;
    Pila *P;
    elemento e, e1;
    VacíaP(&P);
    e.Posibilidad = -1;
    e.Estado = uno;
    AnadeP(&P, e);
    while (! EsVacíaP(P))
    {
        e1=PrimeroP(P);
        BorrarP(&P);
        e = e1;
        switch (e.Estado)
        {
            case uno:
            {
                Posibilidad = e.Posibilidad;
                colocado = 0;
                while (! colocado && (Posibilidad < 8-1))
                {
                    Posibilidad++;
                    x += DespX[Posibilidad];
                    y +=DespY[Posibilidad];
                    if ((x < n) && (y < n) && (x >= 0) && (y >= 0))
                        if (Tablero[x][y] == 0)
                            colocado = 1;
                    if (! colocado)
                    {
                        x -= DespX[Posibilidad];
                        y -= DespY[Posibilidad];
                    }
                }
                if (colocado)
                {
                    Tablero[x][y] = i;
                    if (i == nCuadrado)
                    {
                        Escribeolucion();

```



```

        Contador ++;
        e.Estado = dos;
        e.Posibilidad = Posibilidad;
        i ++;
        /*se pone para no distinguir entre la vuelta de ensayar y solución*/
        AnadeP(&P, e);
    }
    else
    {
        e.Estado = uno;
        e.Posibilidad = Posibilidad;
        AnadeP(&P, e);
        e.Estado = dos;
        e.Posibilidad = Posibilidad;
        AnadeP(&P, e);
        e.Estado = uno;
        e.Posibilidad = -1;
        AnadeP(&P, e);
        i ++;
    }
    break;
}
case dos:
{
    Posibilidad = e.Posibilidad;
    Tablero[x][y] = 0;
    i --;
    x -= DespX[Posibilidad];
    y -= DespY[Posibilidad];
}
}
}
}
void main(void)
{
    printf("valor de n ");
    scanf("%d",&n);
    nCuadrado = n * n;
    DespX[0] = 2;   DespY[0] = 1;   DespX[1] = 1;   DespY[1] = 2;
    DespX[2] = -1;  DespY[2] = 2;   DespX[3] = -2;  DespY[3] = 1;
    DespX[4] = -2;  DespY[4] = -1;  DespX[5] = -1;  DespY[5] = -2;
    DespX[6] = 1;   DespY[6] = -2;  DespX[7] = 2;   DespY[7] = -1;
    Contador=1;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            Tablero[i][j] = 0;
    Tablero[0][0] = 1;
    i = 2;
    x = 0;
    y = 0;
    ensayartodasnr();
    ensayartodas();
}

```

- 10.8.** *Dispone de una baraja española. Se trata de colocar las cuarenta cartas en sus correspondientes palos por orden creciente, partiendo inicialmente de las cartas previamente barajadas y disponiendo de un montón de descartes. Al comenzar el juego, los palos están vacíos, y sólo se puede colocar en cada uno (el as correspondiente). El juego funciona de la siguiente forma, se toman cartas de la baraja por parejas y se van colocando en el montón de descartes, de forma que la carta extraída en el primer lugar queda por debajo de la segunda en el montón de descartes. En la última extracción sólo se tomará una carta si en la baraja queda una. Tras haber puesto una pareja en el montón de descartes, se llevan todas las cartas que sean posibles a los palos, teniendo en cuenta que sólo es accesible en cada momento la carta situada en la parte superior; y que sólo se podrá colocar si es la siguiente en el orden de la última que se colocó en el palo correspondiente. Si en la baraja no quedan cartas, entonces se pasan todas las cartas del montón de descartes a la baraja en el orden contrario al que se encuentran. El solitario termina cuando no puede colocarse ninguna carta más en los correspondientes palos. El solitario tiene éxito si no quedan cartas en la baraja.*

Solución (se encuentra en la página web del libro)

PROBLEMAS PROPUESTOS

- 10.1.** Construir un programa en el que se manipulen un total de n pilas P_i , $i=1, 2, \dots, n$. La entrada de datos serán pares de números enteros (i, j) con $1 \leq \text{Abs}(i) \leq n$. Si i es positivo indica que se debe insertar el elemento j en la pila P_i . Si i es negativo indica que se debe eliminar el elemento j en la pila P_i . Si i es cero indica que se ha terminado el proceso. Inicialmente las pilas están vacías. Los datos están almacenados en un archivo de texto, de tal manera que cada línea del archivo contiene un par de datos, excepto la primera que contiene el número de pilas. El programa deberá gestionar las pilas y escribir al final del proceso los resultados de las n pilas en n archivos de texto.
- 10.2.** Modificar el programa del problema 10.1 para admitir tripletes de número enteros (i, j, k) , donde i, j tienen el mismo significado que en problema anterior, y k es un número entero que puede tomar los valores -1 y 0 con el siguiente significado:
 -1 indica borrar todos los datos de la pila i , inicializarla a vacío; 0 proceso normal con el valor de i y de j .
- 10.3.** Modificar el programa del ejercicio 10.2 de tal manera que en el caso $k = -1$ la pila i es eliminada y a partir de ese momento todas las referencias a la pila i son asignadas a la siguiente en su orden natural (pila $i+1$, o bien pila 1 en el caso de que no exista la pila $i+1$).
- 10.4.** Eliminar la recursividad de los problemas resueltos del capítulo 4.
- 10.5.** Escribir un programa que permita gestionar dos pilas en un *array* unidimensional de n elementos. Los módulos no deben declarar un error a menos que se hayan usado todas las posiciones del *array*. Modificarlo para usar tres pilas.
- 10.6.** Obtener una secuencia de 10 elementos reales, guardarlos en un *array* y ponerlos en una pila. Imprimir la secuencia original y, a continuación, imprimir la pila extrayendo los elementos.
- 10.7.** Escribir funciones para calcular el mayor, menor y la media de una pila de números reales.
- 10.8.** 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 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 cada línea del archivo y determine si es palíndromo.
- 10.9.** Escribir un programa que, haciendo uso del *TAD Pila de caracteres*, 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: $((a+b)*5)-7$. A ésta otra expresión le falta un corchete: $2*((a+b)/2.5 + x - 7*y$.
- 10.10.** Escribir una función que reciba una pila y devuelva una copia de ella.
- 10.11.** Escribir un algoritmo para resolver el problema del laberinto y elimine la recursividad usando pilas. Se tiene un laberinto y se trata de encontrar si existe un camino que lleve del punto de entrada al laberinto al punto de salida (ver Ejercicio Resuelto 4.14).

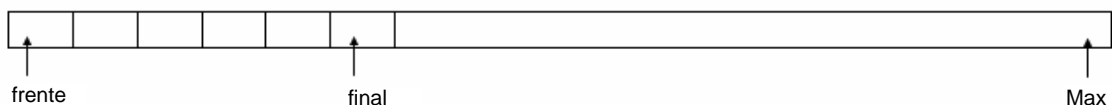
Colas, colas de prioridad y montículos

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. La estructura de datos *cola de prioridad*, es una estructura utilizada para planificar tareas en aplicaciones informáticas donde la prioridad de la tarea se corresponde con la clave de ordenación. También se aplica 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. En este capítulo se estudian las estructuras de datos cola, *colas de prioridades*, y los montículos como estructuras para implementar eficientemente las colas de prioridades

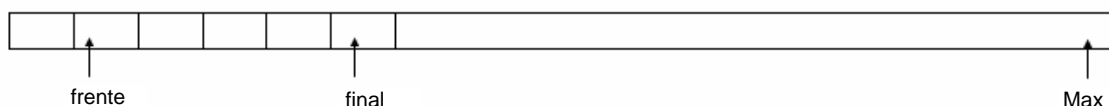
11.1. Colas

Una cola es una estructura de datos lineal en donde las eliminaciones se realizan por uno de sus extremos, denominado frente, y las inserciones por el otro, denominado final. Se las conoce como estructuras FIFO (*First Input First Output*). Una cola se deberá implementar de forma dinámica, es decir con punteros, pero también se podrá realizar mediante un array y dos variables numéricas (*frente*, *final*) que marquen la posición de los elementos extremos de la cola. La realización de una cola con *arrays* exige reservar memoria contigua para el máximo de elementos previstos. En muchas ocasiones esto da lugar a que se desaproveche memoria; también puede ocurrir lo contrario, se llene la cola y no se pueda seguir con la ejecución del programa.

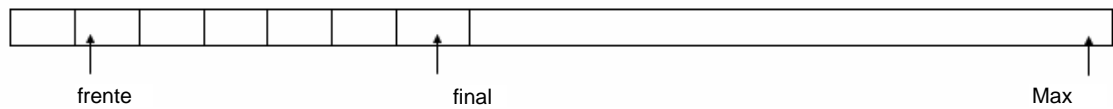
EJEMPLO 11.1. Representación gráfica de la colocación y eliminación de elementos en una cola implementada con arrays.



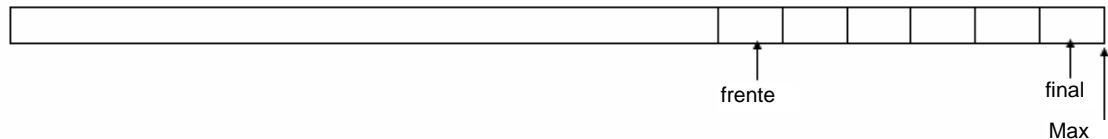
Tras extraer un elemento las posiciones de frente y final son:



Si ahora se añade un elemento



Esta implementación tiene como inconveniente que puede ocurrir que la variable `final` llegue al valor máximo de la tabla, con lo cual no se puedan seguir añadiendo elementos a la cola, aún cuando queden posiciones libres a la izquierda de la posición `frente`.



Dado que al implementar una cola con *arrays*, el avance lineal de `frente` y `final` puede dejar *huecos* por la izquierda del *array*, se plantean implementaciones que ofrezcan solución a este problema:

Retroceso. Consiste en mantener fijo a 0 el valor de `frente`, realizando un desplazamiento de una posición para todas las componentes ocupadas cada vez que se efectúa la supresión de un elemento.

Reestructuración. Cuando `final` llega al máximo de elementos se desplazan las componentes ocupadas hacia atrás las posiciones necesarias para que el principio coincida con el principio de la tabla. Una implementación en C puede consultarse en el Ejercicio Resuelto 3.9.

Mediante un *array* circular. Un *array* circular es aquel en el que se considera que la primera posición sigue a la última. (Figura 11.1). Una implementación en C puede consultarse en el Ejercicio Resuelto 11.1.

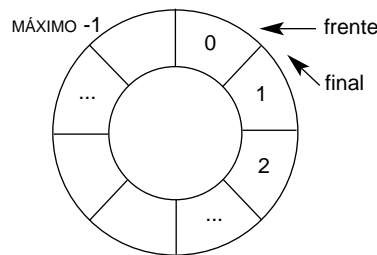


Figura 11.1. Array circular.

11.2. Especificación formal

TAD Cola (VALORES: cola de elementos;
OPERACIONES: Colavacia, Vacía, Primero, Poner, Quitar)

Sintaxis:

*Colavacia() → Cola
Vacía(Cola) → Boolean

```

Primero(Cola)      → elemento
*Poner(Cola, elemento) → Cola
Quitar(Cola)       → Cola
Semántica: Sea C: Cola;  $\forall x \in \text{elemento}$ 
Vacia(Colavacia()) ⇒ true
Vacia(Poner(C, x))  ⇒ false
Primero(Colavacia()) ⇒ error
Primero(Poner(C, x)) ⇒ si Vacia(C) entonces
                        x
                        si_no
                        Primero(C)
                        fin_si
Quitar(Colavacia()) ⇒ error
Quitar(Poner(C, x)) ⇒ si Vacia(C) entonces
                        Colavacia()
                        si_no
                        Poner(Quitar(C), x)
                        fin_si

```

11.3. Implementación con variables dinámicas

Cuando la cola se implementa utilizando variables dinámicas (Figura 11.2), la memoria utilizada se ajusta en todo momento al número de elementos de la cola, pero se consume algo de memoria extra para realizar el encadenamiento entre los elementos de la cola. Se utilizan dos punteros para acceder a la cola, *frente* y *final*, que son los extremos por donde salen los elementos y por donde se insertan respectivamente.

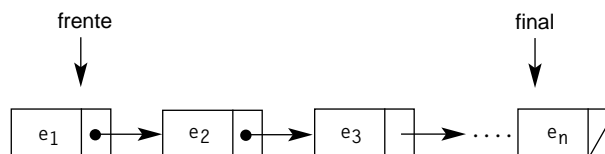
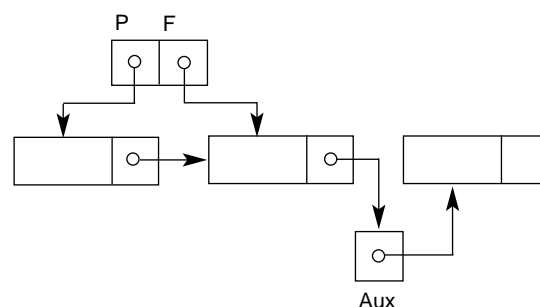


Figura 11.2. Cola implementada utilizando variables dinámicas.

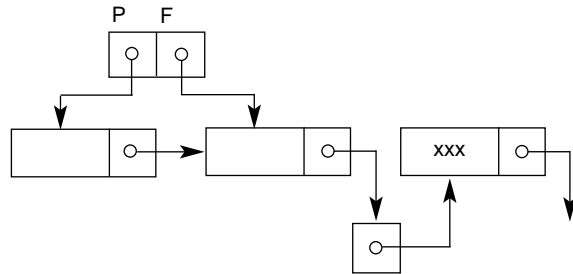
EJEMPLO 11.2. Diseñar un archivo, *cola.h*, donde se declaren los tipos de datos y los prototipos de las funciones que representan las operaciones de mantenimiento de una cola.
(Solución en la página web del libro)

EJEMPLO 11.3. Representar gráficamente la colocación en una cola de un nuevo elemento.

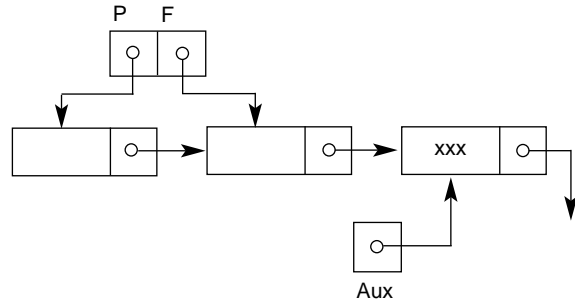
Para **insertar** un elemento, primero hay que reservar espacio para él.



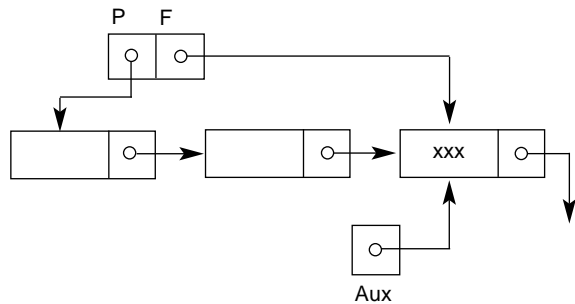
Después se introduce en el campo info el elemento a insertar y, como se trata del último elemento de la estructura, se pone su campo sig a **NULL**.



Si la cola está vacía, es decir sin el primer elemento de la estructura, es necesario hacer que c.p apunte también a ese nuevo elemento. En caso contrario el campo sig del último elemento, es decir, el nodo apuntado por c.f, debe apuntar al nuevo elemento.

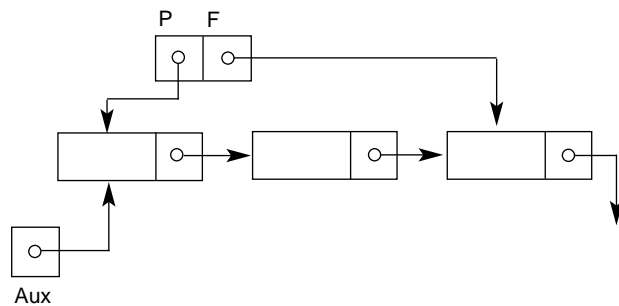


Por último c.f siempre deberá apuntar al nuevo elemento.

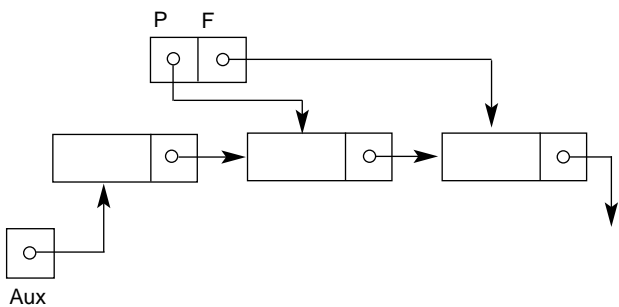


EJEMPLO 11.4. Representación gráfica de la eliminación de un elemento de una cola.

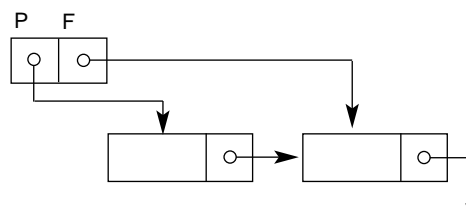
Para **borrar** un elemento, se asigna a una variable auxiliar la dirección del nodo a borrar (el apuntado por c.p).



Después c.p debe apuntar al nodo que apunta el campo sig del primer elemento.



Por último hay que liberar la posición de memoria ocupada por el nodo apuntado por Aux. Además, si la cola se ha quedado vacía (si c.p es **NULL**), se debe hacer que también c.f sea **NULL**.



La implementación en C de las operaciones del tipo abstracto Cola puede consultarse en el Ejercicio Resuelto 11.2.

11.4. Colas circulares

Constituyen una variante de las listas circulares. Al realizar una lista circular, por conveniencia se establece que el puntero de acceso a la lista referenciaba al último nodo, y que el nodo siguiente es considerado el primero. Según este convenio y teniendo en cuenta la definición de cola, las inserciones de nuevos elementos en la cola se realizan siempre por delante del nodo siguiente al apuntado por el puntero de acceso y las eliminaciones se efectúan siempre en el nodo siguiente al referenciado por dicho puntero. El único elemento accesible de la cola será el primero. Las operaciones son similares a las realizadas con listas circulares. Una implementación en C puede consultarse en el Ejercicio Resuelto 11.3.

11.5. Bicolos

Una **bicola** es un conjunto de elementos al que se pueden añadir o quitar elementos en cualquier extremo del mismo. Se puede decir que es una cola bidireccional. A los dos extremos de una bicola los llamamos Izquierdo y Derecho respectivamente. Para representar una bicola se puede elegir una representación estática, con *arrays*, o una representación dinámica, con punteros.

Es posible establecer restricciones en una bicola con respecto al tipo de entrada o al tipo de salida. Una bicola con entrada restringida es aquella que sólo permite inserciones por uno de los dos extremos, pero acepta las eliminaciones por ambos. Una bicola con restricción de salida es aquella que permite inserciones por los dos extremos, pero sólo permite retirar elementos por uno de ellos.

11.6. Especificación formal de TAD bicola sin restricciones

TAD Bicola (VALORES: bicola de elementos;
OPERACIONES: VacíaBi, EsVacíaBi, AnadeBiP, AnadeBiF, BorrarBiP, B o r r a r B i F ,
PrimeroBiP, PrimeroBiF)

Sintaxis:

*VacíaBi	→	Bicola
EsVacíaBi	→	Boolean
*AnadeBiP(Bicola, elemento)	→	Bicola
*AnadeBiF(Bicola, elemento)	→	Bicola
BorrarBiP(Bicola)	→	Bicola
BorrarBiF(Bicola)	→	Bicola
PrimeroBiP(Bicola)	→	Elemento
PrimeroBiF(Bicola)	→	Elemento

Semántica: Sea B: Bicola B; $\forall e \in \text{elemento}$

EsVacíaBi(VacíaBi)	\Rightarrow	True
EsVacíaBi(AnadeBiP(B, e))	\Rightarrow	false
EsVacíaBi(AnadeBiF(B, e))	\Rightarrow	false
PrimeroBiP(VacíaBi)	\Rightarrow	Error
PrimeroBiP(AnadeBiP(B, e))	\Rightarrow	e
PrimeroBiP(AnadeBiF(B, e))	\Rightarrow	si EsVacíaBi(B) entonces

e

		si_no
		PrimeroBiP(B)
		fin_si
PrimeroBiF(VaciaBi)	⇒	Error
PrimeroBiF(AnadeBiF(B, e))	⇒	e
PrimeroBiF(AnadeBiP(B, e))	⇒	si EsVaciaBi(B) entonces
		e
		si_no
		PrimeroBiP(B)
		fin_si
BorrarBiP(VaciaBi)	⇒	Error
BorrarBiP(AnadeBiP(B, e))	⇒	B
BorrarBiP(AnadeBiF(B, e))	⇒	si EsVaciaBi(B) entonces
		B
		si_no
		AnadeBiF(BorrarBiP(B),e)
		fin_si
BorrarBiF(VaciaBi)	⇒	Error
BorrarBiF(AnadeBiP(B, e))	⇒	B
BorrarBiF(AnadeBiP(B, e))	⇒	si EsVaciaBi(B) entonces
		B
		si_no
		AnadeBiP(BorrarBiF(B),e)
		fin_si

11.7. Colas de prioridad

El TAD Cola de Prioridad es un tipo abstracto de datos que ha de permitir, al menos, las siguientes dos operaciones: insertar un elemento con una cierta prioridad y eliminar el de menor prioridad. La operación *insertar* tiene su significado habitual, es decir se inserta como último elemento de todos lo que tengan la misma prioridad, mientras que *eliminar* es especial pues se ha de eliminar siempre el primero de entre los que tengan menor prioridad.

Las colas de prioridad se pueden implementar de muy diversas maneras. Por ejemplo con una lista enlazada ordenada. Al ordenar la lista ascendentemente por la prioridad, para eliminar el menor bastará con tomar el primer elemento, pero la inserción se realiza mediante la inserción ordenada de una lista. Una forma bastante sencilla y eficiente de implementar las colas de prioridad es con un *array* en el que sus elementos sean punteros a colas. Los elementos se colocarán en una u otra cola directamente según su prioridad (Figura 11.3)

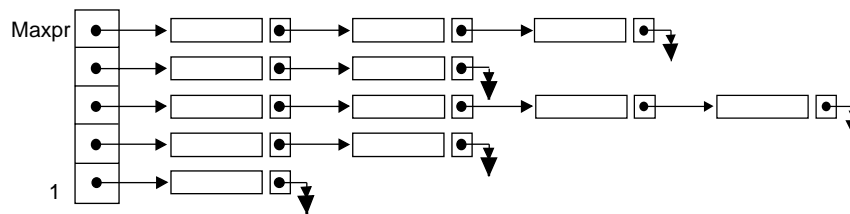


Figura 11.3. Cola de prioridad.

11.8. Especificación del Tipo Abstracto de Datos “Cola de Prioridad”

En cuanto a la sintaxis y semántica de la cola de prioridad ha de tenerse en cuenta que es válida la sintaxis y semántica dada para las listas ordenadas (ascendentemente), con la simple modificación de que en la inserción ordenada se ha de añadir el elemento al final de los que tengan la misma prioridad.

(Especificación en la página web del libro)

11.9. Montículos

Un **montículo** en mínimo (en máximo) es una agrupación en forma piramidal de elementos en la que para cualquier nivel el peso de estos es menor o igual (mayor o igual) 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). Un montículo binario en mínimo 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 que el contenido de su hijos.

Se entiende por árbol binario casi completo como un árbol que tiene todos los niveles completos excepto, posiblemente, en último nivel pero en el que los nodos se almacenan de izquierda a derecha. La disposición de las claves de un árbol binario completo (las posiciones se corresponde con los niveles del árbol, para un nivel de izquierda a derecha) permite representar el árbol en un *array* lineal de una forma eficiente de tal manera que si un nodo ocupa la posición i , su padre (si lo tiene) ocupa la posición $i/2$, el nodo hijo izquierdo ocupa en la posición $2*i$ y el nodo hijo derecho en $2*i+1$ (si los tienen). Utilizando la representación en un *array* de un montículo en mínimo de tamaño n se tiene que: $a[i] \leq a[2*i]$, $a[i] \leq a[2*i+1]$ $\forall i = 1..n/2$. Las operaciones básicas para tratar un montículo tanto en mínimo (como en máximo) se corresponden con las de tratamiento de una cola de prioridad y son de tiempo constante o logarítmico.

Operaciones de tiempo constante:

`CrearMonticulo` que crea el montículo vacío (basta con indicar que no tiene elementos).

`Es Vacio` que decide si un montículo está vacío.

`BuscarMinimo`, que busca el mínimo del montón (siempre se encuentra en la posición 1 del montón).

Operaciones de tiempo logarítmico:

`Insertar`, añade un nuevo elemento al montón. Para realizarlo se incrementa el número de elementos del montón en una unidad, y posteriormente si el elemento añadido no es menor que su padre se intercambian las dos posiciones, repitiéndose el proceso hasta que se cumpla la condición de montón.

`EliminarMinimo` elimina un elemento del montón. Este mínimo se encuentra en la posición 1 del montón. Como el montón disminuye en una unidad, el último elemento del montón se pone en la posición 1 del montón (que ha sido eliminada) y posteriormente se comprueba si es menor o igual que sus hijos. En caso de que no lo sea se intercambia con el menor de sus hijos. El proceso se repite hasta que la estructura sea un montón.

PROBLEMAS BÁSICOS

11.1. Implementar una cola en un array circular que pueda almacenar un máximo de 99 elementos.

Análisis

Se define en primer lugar una constante `MAXIMO` que tendrá el valor máximo de elementos (100-1) que podrá contener la cola. Se define la cola como una estructura cuyos campos (miembros) serán las variables enteras frente que apuntará al primer elemento de la cola y final que apuntará al último elemento de la cola. Los datos de la cola se almacenarán en el miembro de la estructura `A` definido como un *array* de elementos. Posteriormente se implementa las primitivas que son: `VaciaC`, `EsvaciaC`, `AnadeC`, `BorraC`, `PrimeroC`. Además se implementan la función `EstallenaC`, de la siguiente forma:

- `VaciaC`. Crea la cola vacía poniendo frente a 0 y final `MAXIMO - 1`.
- `sig`. Es una función auxiliar que permite poner en círculo los miembros del *array*, haciendo que el siguiente de la posición `MAXIMO - 1` sea la posición 0.

- *EsvaciaC*. Indica cuándo estará la cola vacía. Ocurre cuando el siguiente de *final* coincide con *frente*.
- *EstallenaC*. Si bien no es una primitiva básica de gestión de una cola al realizar la implementación con un *array*, hay que tenerla para prevenir posibles errores. Cada vez que se añade a la cola se llama a la función *EstallenaC* que decide si la cola está llena. Ocurre si el siguiente del siguiente de *final* coincide con *frente*.
- *AnadeC*. Añade un elemento a la cola. Para hacerlo comprueba que la cola no esté llena, y en caso afirmativo, pone en *final* el siguiente de *final*, para posteriormente poner en esa posición el elemento.
- *PrimeroP*. Comprueba que la cola no esté vacía, y en caso de que así sea retorna el elemento del *array A* almacenado en la posición apuntada por el *frente*.
- *BorrarP*. Elimina el primer elemento que entró en la cola. Primeramente comprueba que la cola no esté vacía en cuyo caso, pone en *frente* el siguiente de *frente*.

Codificación (se encuentra en la página web del libro)

11.2. Escribir las declaraciones necesarias y las primitivas de gestión de una cola implementada con listas enlazadas.

Análisis

Se declara en primer todos los tipos de datos necesarios para una lista enlazada. Una cola será una estructura con dos punteros a la lista, *Frente* que apuntará al primer elemento de la cola y *Final* que apuntará al último elemento de la lista y de la cola.

- *VaciaC*. Crea una cola vacía, para lo cual basta con poner el *Frente* y el *Final* a *NULL*.
- *EsVaciaC*. Decide si una cola está vacía. Es decir si *Frente* y *Final* valen *NULL*.
- *PrimeroC*. Extrae el primer elemento de la cola que se encuentra en el nodo *Frente*. Previamente a esta operación ha de comprobarse que la cola no esté vacía.
- *AnadeC*. Añade un elemento a la cola. Este elemento se añade en un nuevo nodo que será el siguiente de *Final* en el caso de que la cola no esté vacía. Si la cola está vacía el *Frente* debe apuntar a este nuevo nodo. En todo caso el *Final* siempre debe moverse al nuevo nodo.
- *BorrarC*. Elimina el primer elemento de la cola. Para hacer esta operación la cola no debe estar vacía. El borrado se realiza avanzando *Frente* al nodo siguiente, y liberando la memoria correspondiente.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
typedef int TipoElemento;
struct nodoCola
{
    TipoElemento e;
    struct nodoCola* sig;
} NodoCola;

typedef struct
{
    NodoCola * Frente,* Final;
}Cola;

void VaciaC(Cola* C);
void AnadeC(Cola* C, TipoElemento e);
void BorraC(Cola* C);
TipoElemento PrimeroC(Cola C);
int EsvaciaC(Cola C);
```

```

void VaciaC(Cola* C)
{
    C->Frente = NULL;
    C->Final = NULL;
}

int EsVaciaC(Cola C)
{
    return (C.Frente == NULL);
}

void AnadeC(Cola* C, TipoElemento e)
{
    NodoCola* a;
    a = (NodoCola*)malloc(sizeof(NodoCola));
    a->e = e;
    a->sig = NULL;
    if (EsVaciaC(*C))
        C->Frente = a;
    else
        C->Final->sig = a;
    C->Final = a;
}

void BorrarC(Cola* C)
{
    NodoCola *a;
    if (!EsVaciaC(*C))
    {
        a = C->Frente;
        C->Frente = C->Frente->sig;
        if (C->Frente == NULL)
            C->Final = NULL;
        free(a);
    }
    else
    {
        puts("Error eliminacion de una cola vacía");
        exit(-1);
    }
}

TipoElemento PrimeroC(Cola C)
{
    if (EsVaciaC(C))
    {
        puts("Error: cola vacía");
        exit(-1);
    }
    return (C.Frente->e);
}

```

11.3. *Escribir las declaraciones necesarias y las primitivas de gestión de una cola implementada con una lista circular.*

Análisis

Se declara en primer lugar todos los tipos de datos necesarios para una lista enlazada. Una Cola será un puntero al nodo de la lista denominado `NodoCola`. Todas las primitivas tienen la misma estructura que en el problema anterior 11.2. La implementación se realiza de tal manera que si la Cola no está vacía el apuntador C apunta al último elemento de la cola y el siguiente de C apunta siempre al primer elemento de la cola. Sólo se incluye el typedef de la definición de la Cola y la implementación de las primitivas. El resto de los tipos de datos, son los mismos que en el problema 11.2.

Codificación

```
.....
typedef NodoCola *Cola;
.....
void VacíaC(Cola* C)
{
    *C = NULL;
}

int EsVacíaC(Cola C)
{
    return (C == NULL);
}

void AnadeC(Cola* C, TipoElemento e)
{
    NodoCola* a;
    a = (NodoCola*)malloc(sizeof(NodoCola));
    a->e = e;
    if (! EsVacíaC(*C))
    {
        a->sig = (*C)->sig;
        (*C)->sig = a;
    }
    else
        a->sig = (*C);
    *C = a;
}

void BorrarC(Cola* C)
{
    NodoCola *a;
    if (!EsVacíaC(*C))
    {
        a = (*C)->sig;
        if(a == *C)
            (*C) = NULL;
        else
            (*C)->sig = a->sig;
        free(a);
    }
    else
    {
        puts("Error eliminacion de una cola vacía");
        exit(-1);
    }
}
```

```

    }
}

TipoElemento PrimeroC(Cola C)
{
    if (EsVaciaC(C))
    {
        puts("Error: cola vacía");
        exit(-1);
    }
    return (C->sig->e);
}

```

11.4. Se define una bicola como una cola de doble entrada. Los elementos pueden entrar y salir por cada uno de los extremos. Las primitivas son:

<i>VaciaBi</i>	<i>Crea una bicola que está vacía.</i>
<i>EsVaciaBi</i>	<i>Nos dice si una bicola es vacía.</i>
<i>AnadeBiP</i>	<i>Añade a una bicola un elemento por el principio.</i>
<i>AnadeBiF</i>	<i>Añade a una bicola un elemento por el final.</i>
<i>PrimeroBiP</i>	<i>Extrae el primer elemento por el principio de una bicola.</i>
<i>PrimeroBiF</i>	<i>Extrae el primer elemento por el final de una bicola.</i>
<i>BorraBiP</i>	<i>Borra el primer elemento por el principio de una bicola.</i>
<i>BorraBiF</i>	<i>Borra el primer elemento por el final de una bicola.</i>

Definir la sintaxis y la semántica de las funciones de una bicola. Implementar el TAD anterior mediante listas doblemente enlazadas.

Análisis

Se define la sintaxis y la semántica de la forma habitual (véase el apartado 11.6). La implementación se realiza mediante listas dobles sin nodo cabecera ni final. La bicola será un registro con dos apuntes a la lista doble: *Frente* y *Final*.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef int Telemento;
struct Puntero
{
    Telemento Info;
    struct Puntero *Sig, *Ant;
}puntero;
typedef struct
{
    puntero *Frente, *Final;
}Bicola;

void VaciaBi(Bicola* B);
int EsvaciaBi(Bicola B);
void AnadeBiP( Bicola *B, Telemento e);
void AnadeBiF( Bicola *B, Telemento e);

```

```

void PrimeroBP(Bicola B,Telemento *e);
void PrimeroBF(Bicola B, Telemento *e);
void BorraBiP(Bicola *B);
void BorraBiF( Bicola *B);

void VaciaBi(Bicola* B)
{
    (*B).Frente = NULL;
    (*B).Final = NULL;
}

int EsvaciaBi(Bicola B)
{
    return B.Frente == NULL;
}

void AnadeBiP( Bicola *B, Telemento e)
{
    puntero *aux;
    aux = (puntero*)malloc(sizeof(puntero));
    aux->Info = e;
    aux->Ant = NULL;
    aux->Sig = (*B).Frente;
    if ((*B).Frente == NULL)
        (*B).Final = aux;
    else
        (*B).Frente->Ant = aux;
    (*B).Frente = aux;
};

void AnadeBiF( Bicola *B, Telemento e)
{
    puntero *aux;
    aux = (puntero*)malloc(sizeof(puntero));
    aux->Info = e;
    aux->Sig = NULL;
    aux->Ant =(*B).Final;
    if ((*B).Final == NULL)
        (*B).Frente = aux;
    else
        (*B).Final->Sig = aux;
    (*B).Final = aux;
};

void PrimeroBP(Bicola B, Telemento *e)
{
    if (B.Frente != NULL)
        *e = B.Frente->Info;
};

void PrimeroBF( Bicola B, Telemento *e)
{
    if (B.Final != NULL)

```

```

        *e = B.Final->Info;
    };

void BorraBiP( Bicola *B)
{
    puntero *aux;
    if ((*B).Frente != NULL)
    {
        aux = (*B).Frente;
        if ((*B).Frente == (*B).Final)
        {
            (*B).Frente = NULL;
            (*B).Final = NULL;
        }
        else
        {
            (*B).Frente = aux->Sig;
            (*B).Frente->Ant = NULL;
        }
        aux->Sig = NULL;
        free(aux);
    }
}

void BorraBiF( Bicola *B)
{
    puntero *aux;
    if ((*B).Final != NULL)
    {
        aux = (*B).Final;
        if ((*B).Frente == (*B).Final)
        {
            (*B).Frente = NULL;
            (*B).Final = NULL;
        }
        else
        {
            (*B).Final = aux->Ant;
            (*B).Final->Ant = NULL;
        }
        aux->Ant = NULL;
        free(aux);
    }
}

```

11.5. Una variante del conocido problema de José es el siguiente. Determinar los últimos datos que quedan de una lista inicial de $n > 5$ números sometida al siguiente algoritmo:

Se inicializa $n1$ a 3.

Se retiran de la lista los números que ocupan las posiciones 2, $2+n1$, $2+2*n1$, $2+3*n1$, ..., etc.

Se incrementa $n1$ en una unidad.

Si quedan en la lista menos de $n1$ elementos se para, en otro caso se itera.

Análisis

Para resolver el problema se usa la cola implementada en el problema 11.1, o el 11.2 o bien el 11.3. La variable *n* indica en cada iteración del bucle el número de datos que hay en la cola. La variable *n2*, contiene en cada iteración el número de elementos que van quedando en la nueva cola. El número *n* de elementos de la cola inicial se genera aleatoriamente, así como la composición inicial de la cola. Se usa una función `mostrarCola`, que se encarga de mostrar los datos de una cola una vez terminado el algoritmo.

Codificación

```
void mostrarCola(Cola* C)
{
    // muestra los elementos de una cola que se le pase como parámetro.
    TipoElemento e;
    while (!EsVaciaC(*C))
    {
        e = PrimeroC(*C);
        printf("%d ", e);
        BorrarC(C);
    }
}

void main()
{
    Cola C;
    int n, n1, n2, n3, i;
    randomize();
    n = 1 + random(500);
    VacíaC(&C);
    for (i = 1; i <= n; i++)
        AnadeC(&C, 1+random(1024));
    n1 = 3;
    while (n1 <= n)
    {
        printf("\n Se quitan elementos a distancia %d ", n1);
        n2 = 0; /*Contador de elementos que quedan */
        for (i = 1; i <= n; i++)
        {
            n3 = PrimeroC(C); BorrarC(&C);
            if (i % n1 == 2)
                printf("\t %d se quita.", n3);
            else
            {
                AnadeC(&C, n3); /* se vuelve a meter en la cola */
                n2++;
            }
        }
        n = n2;
        n1++;
    }
    printf("\n Los números de la suerte: ");
    mostrarCola(&C);
}
```

- 11.6.** *Escribir un programa que lea una línea de la entrada y determine si la frase leída es palíndroma. Una frase se dice que es palíndroma, si se puede leer lo mismo de izquierda a derecha y de derecha a izquierda, una vez eliminados los blancos.*

Análisis

Un ejemplo de una frase palíndroma es: “dabale arroz a la zorra el abad“. Para resolver el problema basta con seguir la siguiente estrategia: añadir cada carácter de la frase que no sea blanco a una pila y a la vez a una cola. La extracción simultánea de caracteres de ambas y su comparación determina si la frase es o no palíndroma. La solución está dada en la función `palindroma`. Se usa además una función, `SonIguales`, que decide si una `Pila` y una `Cola` que recibe como parámetros constan de los mismos caracteres. La `Pila` está implementada con una lista enlazada (codificación en el ejercicio 10.1).

Codificación

```
int SonIguales(Pila *P, Cola C)
{
    int sw=1;
    TipoDato e,el;
    while (! EsVaciaP(P) && ! EsVaciaC(C) && sw)
    {
        e = PrimeroP(P);
        BorrarP(&P);
        el = PrimeroC(C);
        BorrarC(&C);
        sw = (e == el);
    }
    return (sw && EsVaciaP(P) && EsVaciaC(C));
}

int palindroma()
{
    Pila *P;
    Cola C;
    char ch;
    puts(" frase a comprobar que es palindroma");
    VacíaP(&P);
    VacíaC(&C);
    for( ;(ch = getchar()) != '\n'; )
    if (ch != ' ')
    {
        AnadeP(&P,ch);
        AnadeC(&C,ch);
    }
    return (SonIguales(P,C));
}
```

- 11.7.** *Implementar una Bicola en un array circular que pueda almacenar un máximo de 99 elementos.*

Análisis

Se define en primer lugar una constante `MAXIMO` que tendrá el valor máximo de elementos (100-1) que podrá contener la bicola. Se define la bicola como una estructura cuyos campos (miembros) serán las variables enteras `frente` que apuntará al primer elemento de la bicola y `final` que apuntará al último elemento de la bicola. Los datos de la bicola se almacenará

en el miembro de la estructura A definido como un *array* de elementos. A continuación se implementan las operaciones definidas en el problema 11.4. Además, se usan las funciones siguientes:

- *ant*. Es una función auxiliar que permite poner en círculo los miembros del *array*, haciendo que el siguiente de la posición MÁXIMO-1 sea la posición 0, pero retrocediendo hacia atrás.
- *AnadeBiF*. Añade un elemento a la bicola por el *final*. Para hacerlo comprobar que la bicola no esté llena, y en caso afirmativo, pone en frente el anterior a frente, para posteriormente poner en el *array* A en la posición *final* el elemento.
- *PrimeroBiF*. Comprueba que la bicola no esté vacía, y en caso de que así sea retorna el elemento del *array* A almacenado en la posición apuntada por el *final*.
- *BorrarBiF*. Elimina el último elemento que entró en la bicola. Primeramente comprueba que la bicola no esté vacía en cuyo caso, pone en *final* el anterior de *final*.

Codificación (En la página web del libro)

11.8. Implementar una cola de prioridad mediante un array de colas.

Análisis

La solución se presenta mediante una estructura cuyos miembros son una variable entera que indica el máximo número de prioridades y un *array* de colas. El máximo número de prioridades admitidas es 12. Cada *trabajo* lleva asociado un nombre que es una cadena de caracteres de longitud máxima 20 y una prioridad que debe variar entre 1 y 12. Todos los trabajos con prioridad *i* se almacenan en la posición *i*-1 del *array* de la cola de prioridades de acuerdo con la estructura FIFO. Las primitivas de gestión de la cola de prioridad usan las primitivas de una *Cola* que debe estar previamente declarada.

Codificación

```
#define Maxprio 12
typedef struct
{
    int prioridad;
    char NombreT[20];
}Trabajo;

typedef Trabajo TipoElemento;
/*
    Gestión de una cola
*/
#include "cola.h".....
typedef struct
{
    int NP;
    Cola colas[Maxprio];
} ColaPrioridad;

void VacíaCp(ColaPrioridad* cp);
void AnadeCP(ColaPrioridad* cp, Trabajo t);
Trabajo PrimeroCp(ColaPrioridad cp);
void BorrarCp(ColaPrioridad* cp);
int EsvaciaCp(ColaPrioridad cp);

void VacíaCp(ColaPrioridad* cp)
```

```

{
    int j;
    cp->NP = Maxprio;
    for (j = 0; j < Maxprio; j++)
        VacíaC(&(cp->colas[j]));
}

void AnadeCp(ColaPrioridad* cp, Trabajo t)
{
    if ((1 < t.prioridad) && (t.prioridad <= cp->NP))
        AnadeC(&cp->colas[t.prioridad - 1],t);
    else
        puts("Trabajo con prioridad fuera de rango");
}

Trabajo PrimeroCp(ColaPrioridad cp)
{
    int i = 0, ICola = -1;
        /* búsqueda de la primera cola no vacía */
    do
    {
        if (!EsvaciaC(cp->colas[i]))
        {
            ICola = i;
            i = cp->NP;                                /* termina el bucle */
        }
        else
            i++;
    } while (i < cp->NP);
    if (ICola == -1)
    {
        puts("Cola de prioridades vacía");
        exit(1);
    }
    return PrimeroC((cp->colas[ICola]));
}

void BorrarCp(ColaPrioridad* cp)
{
    int i = 0, ICola = -1 ;
        /* búsqueda de la primera cola no vacía */
    do
    {
        if (!EsvaciaC(cp->colas[i]))
        {
            ICola = i;
            i = cp->NP;                                /* termina el bucle */
        }
        else
            i++;
    } while (i < cp->NP);
    if (ICola == -1)
    {

```

```

        puts("Cola de prioridades vacía");
        exit(1);
    }
    BorraC(&(cp->colas[ICola]));
}

int EsVaciaCP(ColaPrioridad cp)
{
    int i = 0;
    while (EsvaciaC(cp.colas[i]) && i < cp.NP - 1)
        i++;
    return EsvaciaC(cp.colas[i]);
}

```

- 11.9.** *Escribir un programa que implemente una cola de prioridad mediante una lista enlazada ordenada y permita la introducción de datos interactivamente.*

Análisis

Cada trabajo lleva asociado un nombre que es una cadena de caracteres de longitud máxima 20 y una prioridad que entera cualquiera. Todos los trabajos con prioridad *i* se almacenan en la cola de prioridades de acuerdo con la estructura FIFO dentro de la misma prioridad. Las primitivas de gestión de la cola de prioridad son las definidas en el ejercicio 11.8. Se incluye una función, *AnadeCpR*, que hace lo mismo que la función *AnadeCp*, con la diferencia que la primera inserta en la cola de prioridad recursivamente y la segunda iterativamente. Se presenta una función *escribir* que se encarga de presentar en pantalla una cola de prioridad y un programa principal que introduce datos en la cola de prioridad y permite borrarlos. Se programa además la función *NuevoNodoCp* que se encarga de crear un nodo de la lista enlazada que almacene un nuevo trabajo.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
typedef struct
{
    int prioridad;
    char NombreT[20];
}Trabajo;
typedef struct RegistroCP
{
    Trabajo trabajo;
    struct RegistroCP* sig;
}NodoCp;
typedef NodoCp *ColaPrioridad;

void VaciaCp(ColaPrioridad* Cp)
{
    *Cp = NULL;
}

int EsvaciaCp(ColaPrioridad Cp)
{
    return (Cp == NULL);
}

Trabajo PrimeroCp(ColaPrioridad Cp)

```

```
{
    if(EsvaciaCp(Cp))
    {
        printf (" cola de prioridad vacia");
        exit(1);
    }
    return (Cp->trabajo);
}

void BorrarCp(ColaPrioridad* Cp)
{
    NodoCp *ab;
    if (EsvaciaCp(*Cp))
    {
        printf (" cola de prioridad vacia");
        exit(1);
    }
    ab = *Cp;
    *Cp = ab->sig;
    free (ab);
}

void AnadeCpR(ColaPrioridad* Cp, Trabajo trabajo)
{
    NodoCp *nuevonodo;
    if (*Cp == NULL)
    {
        nuevonodo = NuevoNodoCp(trabajo);
        *Cp = nuevonodo;
    }
    else if (trabajo.prioridad < (*Cp)->trabajo.prioridad)
    {
        nuevonodo = NuevoNodoCp(trabajo);
        nuevonodo->sig = *Cp;
        *Cp = nuevonodo;
    }
    else
        AnadeCpR(&(*Cp)->sig, trabajo);
}

void AnadeCp(ColaPrioridad* Cp, Trabajo trabajo)
{
    NodoCp *nuevonodo, *ant, *p;
    nuevonodo = NuevoNodoCp(trabajo);
    if (*Cp == NULL)
        *Cp = nuevonodo;
    else
        if (trabajo.prioridad < (*Cp)->trabajo.prioridad)
        {
            nuevonodo->sig = *Cp;
            *Cp = nuevonodo;
        }
        else
```

```

    {
        ant = p = *Cp;                                // se sabe que no es el primero
        while ((trabajo.prioridad > p->trabajo.prioridad) && (p->sig != NULL) )
        {
            ant = p;
            p = p->sig;
        }
        if (trabajo.prioridad > p->trabajo.prioridad) // falta por comprobar el ultimo
            ant = p;
        nuevonodo->sig = ant->sig;
        ant->sig = nuevonodo;
    }
}

NodoCp* NuevoNodoCp(Trabajo trabajo)
{
    NodoCp *nuevonodo ;
    nuevonodo= (NodoCp*)malloc(sizeof(NodoCp));
    nuevonodo->sig = NULL;
    nuevonodo->trabajo = trabajo;
    return nuevonodo;
}

void Escribir(ColaPrioridad Cp)
{
    printf("\n\t\t Cola de prioridad \n");
    for (; Cp; Cp = Cp->sig)
        printf(" %s   %d \n",Cp->trabajo.NombreT, Cp->trabajo.prioridad);
    printf("\n\n");
}

void main()
{
    Trabajo trabajo;
    int d;
    ColaPrioridad Cp;
    Cp = NULL;
    do
    {
        printf(" Introduzca prioridad del trabajo -1 = fin\n");
        scanf("%d", &trabajo.prioridad);
        if (trabajo.prioridad != -1)
        {
            puts(" nombre del trabajo ");
            scanf("%s", &trabajo.NombreT);
            AnadeCpR(&Cp,trabajo);
        }
    } while (trabajo.prioridad != -1);
    Escribir(Cp);
    do
    {
        printf(" borrar -1 = fin\n");
        scanf("%d", &d);
    }
}

```

```

    if (d != -1)
        BorrarCp(&Cp);
    Escribir(Cp);
} while (d != -1);
}

```

11.10. Codificar en C las primitivas de gestión de un montículo mínimo.

Análisis

El montículo se implementa mediante una estructura cuyos miembros son una variable entera *n* que indica el número de elementos que tiene el montículo, y un *array* *a* que almacena los elementos desde las posiciones 1 hasta un valor constante MAXIMO. No se usa la posición 0 del *array* ya que por definición los hijos del nodo *i* se encuentran en la posición $2*i$ y $2*i+1$ (hijo izquierdo y derecho), y si *i* vale 0 el hijo izquierdo se encuentran en la posición $2*0=0$ que es la misma que su padre. Las primitivas de gestión del Montículo que se implementan son:

- **CrearMontículo**, crea el montículo vacío poniendo el miembro *n* a cero.
- **EsVacio**, decide si un montículo está vacío comprobando si el miembro *n* es cero.
- **BuscarMinimo**, busca el mínimo del montón que se encuentra en la posición 1 del miembro *a*.
- **Insertar**, recibe como parámetro el montículo y una nueva *Clave* y la añade al montón. Para realizarlo incrementa el número de elementos del montón en una unidad coloca la *Clave* en la última posición del miembro *a* del montón y mediante una función *Subir* rehace la condición del montón subiendo elemento añadido por el árbol hasta que lo coloca en una posición que satisface la condición del montón mínimo (la clave sube si el contenido de su padre es mayor).
- **EliminarMinimo**. Elimina el elemento más pequeño del montón que se encuentra en la posición 1 del miembro del montón *a*. Para realizarlo pone en la posición 1 del miembro *a* el último elemento del montón, decrementa el número de elementos del montón, y mediante una función *criba*, rehace la condición del montón mínimo cribando (bajando o hundiendo) el elemento que ocupa la posición 1 hasta una posición del *array* en la que se satisfaga la condición del mínimo.

Codificación

```

#define MAXIMO 100
typedef int Telemento;
typedef struct
{
    Telemento a[MAXIMO+1];
    int n;
}Monticulo;

void Subir(Monticulo* monticulo, int pos)
{
    Telemento Clave;
    int padre, EsMonton=0;
    Clave = monticulo->a[pos];
    padre = pos/2;
    while ((padre >= 1) && !EsMonton)
        if (monticulo->a[padre] > Clave)
        {
            monticulo->a[pos] = monticulo->a[padre];
            pos = padre; padre=padre/2;
        }
    else
        EsMonton=1;
    monticulo->a[pos] = Clave;
}

```



```

}

void Insertar (Monticulo* monticulo, Telemento Clave)
{
    if (monticulo->n == MAXIMO)
        puts("No es posible insertar nuevas claves: montículo lleno");
    else
    {
        (monticulo->n)++;
        monticulo->a[monticulo->n] = Clave;
        Subir(monticulo, monticulo->n);
    }
}

void CrearMonticulo(Monticulo* monticulo)
{
    monticulo->n = 0;
}

int EsVacio (Monticulo monticulo)
{
    return monticulo.n == 0;
}

Telemento BuscarMinimo(Monticulo monticulo)
{
    if (monticulo.n == 0)
        puts( "monticulo vacio");
    return monticulo.a[1];
}

void Criba (int a[], int primero, int ultimo)
{
    /* primero: indice del nodo raiz */
    int EsMonticulo = 0, HijoMenor, aux;
    while ((2 * primero <= ultimo) && !EsMonticulo)
    {
        /* primera condición expresa que no sea un nodo hoja */
        if (2*primero == ultimo) /* tiene un único descendiente */
            HijoMenor = 2* primero;
        else if (a[2 * primero] < a[2 * primero + 1])
            HijoMenor = 2* primero;
        else
            HijoMenor = 2 * primero + 1;
        /* compara raiz con el menor de los hijos */
        if (a[HijoMenor] < a[primero])
        {
            aux= a[primero];
            a[primero] = a[HijoMenor];
            a[HijoMenor] = aux;
            primero = HijoMenor; /* continua por la rama de claves mínimas */
        }
        else
    }
}

```

```

        EsMonticulo = 1;
    }
}

void EliminarMinimo(Monticulo* monticulo)
{
    if (EsVacio(*monticulo))
    {
        puts("No es posible eliminar clave: montículo vacío");
        exit(1);
    }
    else
    {
        monticulo->a[1] = monticulo->a[monticulo->n];
        monticulo->n = monticulo->n - 1;
        Criba(monticulo->a, 1, monticulo->n);
    }
}

```

- 11.11.** Escribir un programa que rellene aleatoriamente de datos enteros un vector y, mediante la estructura de montón mínimo ordene el vector crecientemente y lo presente en pantalla.

Análisis

Usando la declaración y primitivas de gestión de un montón mínimo realizadas en el problema 11.10, se declara una constante `Max2` para declarar un vector de enteros. Se define las funciones `Rellena` que genera aleatoriamente el vector de enteros y la función `Escribe` que presenta el vector de enteros en pantalla. Por último, el programa principal se encarga de llamar a las funciones para generar aleatoriamente el vector, cargarlo en el montículo y posteriormente volver a poner el montículo en el vector para dejarlo ordenado.

Codificación (solución en página web del libro).

PROBLEMAS AVANZADOS

- 11.12.** El estacionamiento de coches (carros) de un aparcamiento se realiza en línea, con una capacidad máxima de hasta 12 coches. Los vehículos (carros) pueden incorporarse por la parte izquierda o por la derecha a la línea. La salida de un coche puede realizarse también por la parte izquierda o por la derecha de la línea. Si un coche llega a la línea y no hay capacidad para aparcar entonces se incorporará a una cola de espera para ser atendido. Esta cola de espera tiene una capacidad una capacidad máxima de 12 coches. Cuando un coche se sale de la línea, al quedar un aparcamiento libre permite que un coche de la cola de espera (si es que existe) sea eliminado de ella e incluido en la línea por el mismo sitio por el que salió el último coche.

Escriba un programa interactivo para emular este aparcamiento teniendo en cuenta que *I* es Incorporación por la izquierda, *D* es Incorporación por la derecha, *i* salida por la izquierda, *d* salida por la derecha y *x* fin de la simulación.

Análisis

El estacionamiento va a estar representado por una bicola `Bi` y por una cola `C` con capacidades de hasta 12 coches cada una de ellas. Se supone que los coches vienen determinados por su matrícula que es una cadena de hasta 51 caracteres. Se usa la bicola definida en el ejercicio 11.7 así como la cola definida en el ejercicio 11.1, donde el `TipoElemento` es, en este caso, el coche.

Codificación

```

typedef struct
{
    matricula[51];
} Coche;
typedef Coche TipoElemento;                                /* Carro en Latinoamérica*/
/*
    cola y bicola como se definen en los ejercicios 11.1 y 11.7 con 12 elementos como
    máximo
*/
void main()
{
    Coche coche;
    char ch;
    Bicola Bi;
    Cola C;
    int continuar = 1;
    VaciaBi(&Bi);
    VaciaC(&C);
    while (continuar)
    {
        puts("\n Entrada de datos: [acción: i/d/I/D] ");
        puts("\n i retirar izquierda d retirar derecha ");
        puts("\n I omsertar izquierda D insertar  derecha ");
        puts(" Para terminar la simulación: x");
        do {
            scanf("%c%c",&ch);
        } while(ch != 'i' && ch != 'd' && ch != 'I' && ch != 'D' && ch != 'x');
        if (ch == 'i')
        {
            if (!EsvaciaBi(Bi))
            {
                coche = PrimeroBiP(Bi);
                BorraBiP(&Bi);
                printf("Salida del coche: %s por la izquierda", coche.matricula);
                if (!EsvaciaC(C))
                {
                    // si hay cohes en cola d espera añadirlo
                    coche= PrimeroC(C);
                    BorraC(&C);
                    AnadeBiP(&Bi,coche);
                }
            }
        }
        else if (ch == 'd')
        {
            if (!EsvaciaBi(Bi))
            {
                coche = PrimeroBiF(Bi);
                BorraBiF(&Bi);
                printf("Salida del coche: %s por la derecha", coche.matricula);
                if (!EsvaciaC(C))

```

```

        {
            // si hay coches en cola de espera añadirlo
            coche= PrimeroC(C);
            BorraC(&C);
            AnadeBiF(&Bi,coche);
        }
    }
}
else if (ch == 'I')
{
    printf( " Introduzca matricula: " );
    gets(coche.matricula);
    if (!EstallenaBi(Bi))
        AnadeBiP(&Bi, coche);
    else
        AnadeC(&C, coche); /* No cabe en la bicola ponerlo en cola de espera*/
}
else if (ch == 'D')
{
    printf( " Introduzca matricula: " );
    gets(coche.matricula);
    if (!EstallenaBi(Bi))
        AnadeBiF(&Bi, coche);
    else
        AnadeC(&C, coche); // No cabe en la bicola ponerlo en cola de espera
}
continuar = !(ch == 'x');
}
}

```

11.13. Se dispone de un sistema de computación que funciona de la siguiente forma: existen 2 niveles de prioridad (1,2). A cada nivel N la CPU le asigna un tiempo de 2 y 1 unidades de tiempo respectivamente. La CPU va eligiendo alternativamente trabajos de cada nivel, lógicamente siempre que existan trabajos. En cada nivel ocurre la siguiente situación. Si un trabajo no termina con el tiempo asignado, es el último trabajo de los que han llegado a ese nivel que se le vuelve a dar servicio. Se trata de realizar una emulación del sistema dando la orden de finalización de los trabajos para un conjunto de trabajos que se encuentran almacenados en un archivo de texto *DATOS.DAT*. Cada línea del archivo tiene la siguiente información:

Nº_TRABAJO HORA_LLEGADA TIEMPO_CPU NIVEL

Cada uno de los datos es entero.

HORA_LLEGADA, son los segundos transcurridos desde que se encendió el sistema.

TIEMPO_CPU, tiempo necesario para su computación en segundos.

NIVEL, toma los valores 1,2.

El archivo se encuentra ordenado de modo ascendente por el valor de *HORA_LLEGADA*. El archivo cabe en memoria central. No existen dos trabajos distintos que lleguen a la misma hora.

Análisis

Se usan como estructuras de datos fundamentales:

- Dos colas de prioridad implementadas cada una de ellas con listas enlazadas ordenadas.
- Un array (arreglo) de tiempos que indica los tiempos de CPU de cada uno de los niveles de prioridad.

La solución presentada se ha estructurado con las siguientes funciones:

- `InsertarNivel`, coloca los datos de un registro en la cola de prioridad correspondiente. La prioridad viene dada por el campo `HoraActual`.
- `SacarDeCola`, extrae el primer dato de la cola de prioridad correspondiente.
- `inicializar`, lee el archivo e inserta los datos en las colas de prioridades.
- `Cogertrabajo`, toma el primer trabajo de la cola de prioridad correspondiente, actualiza el reloj del sistema, y dependiendo de que el tiempo que reste al trabajo sea mayor que el tiempo asignado de CPU inserta el trabajo en la cola de prioridad o no lo hace, por supuesto después de haber actualizado el campo `TiempoResta` y `HoraActual` en el primer caso.
- `ElegirNivel`, decide a qué nivel le corresponde tomar el trabajo, teniendo en cuenta todas las posibilidades.

El programa principal inicializa los datos con el archivo, elige el nivel del que sale el primer trabajo, inicializa el reloj del sistema y los tiempos de CPU asignado a cada uno de los niveles. Posteriormente realiza un bucle controlado por el fin de datos en las colas de prioridad en el que llama a los módulos `CogerTrabajo` y `ElegirNivel`.

Codificación (En página web del libro)

PROBLEMAS PROPUESTOS

- 11.1. Con un archivo de texto se quieren realizar las siguientes acciones. Formar una lista enlazada, de tal forma que en cada nodo esté la dirección de una cola que contiene todas las palabras del archivo que empiezan por una misma letra. Una vez formada esta estructura, se quiere visualizar las palabras del archivo, empezando por la cola que contiene las que empiezan por la letra a, luego las de la letra b, y así sucesivamente.
- 11.2. 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 simplemente enlazada. El programa a desarrollar contempla los siguientes puntos: a) Crear una estructura de cola para recoger en ella el número de la seguridad social de cada repartidor y la entidad anunciada en la propaganda para un único día de trabajo. b) 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.
- 11.3. En un archivo de texto se encuentran los resultados de una competición de tiro al plato, de tal forma que en cada línea se encuentra Apellido, Nombre, número de dorsal y número de platos rotos. Se quiere escribir un programa que lea el archivo de la competición y determine los tres primeros. La salida ha de ser los tres ganadores y a continuación los concursantes en el orden en que aparecen en el archivo (utilizar la estructura cola).
- 11.4. 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, según 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 sitúa en la cola de la caja que tiene menos gente, y no se cambia de cola.
 - En el momento en que un cliente paga en la caja, el carro de la compra que tiene queda disponible.
- 11.5. 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 mas de 10 personas esperando se va con una probabilidad del 20%, aumentando en 10 puntos por cada 15 personas mas que haya esperando, hasta un tope del 50%. 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 admite, como máximo, 50 personas. Simular el sistema durante un periodo de 6 horas para determinar: número de personas que llegan a la sala y número de personas que entran y el tiempo medio que debe esperar una persona para entrar en la sala.

- 11.6.** La universidad de La Alcarria dispone de 15 computadoras conectadas a Internet. Se quiere hacer una simulación de la utilización de las computadoras 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 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 10 de la mañana a las 8 de la tarde.

- 11.7.** Diseñar un algoritmo para que dados dos montículo binarios se mezclen formando un único montículo. ¿Qué complejidad tiene el algoritmo diseñado?
- 11.8.** 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`. Diseñe un algoritmo que realice la operación.
- 11.9.** En un montículo minimal diseñar un algoritmo que encuentre el elemento con mayor clave. ¿Qué complejidad tiene el algoritmo diseñado?

Tablas de dispersión y funciones hash

Las tablas de datos permiten el acceso directo a un elemento de una secuencia, indicando la posición que ocupan. La potencia de las tablas *hash* o dispersas radica en la búsqueda de elementos ya que conociendo el campo clave se puede obtener directamente la posición que ocupa, y por consiguiente la información asociada a dicha clave. 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.

12.1. Tablas de dispersión

Las tablas de dispersión son estructuras de datos en las cuales los elementos que se guardan han de estar identificados por un campo clave que permita la realización de las operaciones de inserción, eliminación y búsqueda con una complejidad constante. La organización ideal en este caso es que el campo clave de los elementos se corresponda directamente con el índice de la tabla; no obstante esto no es posible en muchas ocasiones. La **clave**, que ha de ser un campo del propio registro con un significado lógico, no sirve como índice cuando el porcentaje de claves a utilizar es reducido en comparación con el rango en el que pueden oscilar los valores de las mismas o cuando dichas claves son alfanuméricas. La solución puede ser emplear una función, *funcion de transformación de claves* o *funcion hash*, que transforme las claves a números (índices) en un determinado rango. Las funciones *hash* pueden producir que dos registros con claves diferentes les corresponda la misma dirección relativa; es decir que colisionen dos claves. Por ello el estudio del direccionamiento disperso se divide en dos partes: búsqueda de funciones *hash* y resolución de colisiones.

Considerando $h(x)$ la función *hash*, las operaciones del tipo *Tabla dispersa* pueden especificarse:

<i>Buscar</i> (Tabla T , clave x)	devuelve el elemento de la tabla $T[h(x)]$
<i>Insertar</i> (Tabla T , elemento k)	añade el elemento k , $T[h(\text{clave}(k))] \leftarrow k$
<i>Eliminar</i> (Tabla T , clave x)	retira de la tabla el elemento con clave x , $T[h(x)] \leftarrow \text{LIBRE}$

No obstante, todas estas operaciones deben incorporar un proceso de resolución de colisiones ya que no pueden estar dos elementos diferentes en la misma posición.

12.2.Funciones de transformación de clave

Una función de transformación de claves (*hash*) debe reunir las siguientes características:

- Distribuir las claves uniformemente entre las direcciones para producir pocos sinónimos. Existen incluso funciones de transformación de clave que no dan origen a colisiones, como la que se puede aplicar para transformar los números de las habitaciones de un hotel (habitaciones 101, 102, ..., 120, 201, 202, ..., 220, 301, 302, ..., 320) en un rango de direcciones, relativas al comienzo del archivo, que abarque el total de habitaciones del mismo (direcciones: 1, 2 ..., 20, 21, 22, ..., 40, 41, ..., 60).
- No ser una función compleja que pueda ralentizar los cálculos.

Hay que tener en cuenta que las claves no tienen por qué ser numéricas y en ellas podrán aparecer letras. En general, cuando aparecen letras en las claves se suele asociar a cada letra un entero. Recuerde que existe un valor numérico entero asociado a cada carácter, su código ASCII.

Algunas de las funciones *hash* que resultan más fáciles y eficientes son:

Restas sucesivas

Esta función se emplea con claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas.

EJEMPLO 12.1. Supuesto un colegio donde cada curso tiene un máximo de 300 alumnos y cada alumno un $\text{Num_de_matrícula} = \text{año}(\text{aa}) + \text{curso} + \text{num_de_alumno}$. Aplicar el método de restas sucesivas para obtener un índice adecuado.

Se podría pensar en usar el número de matrícula como índice o posición para el correspondiente registro en el archivo, pero éste resultaría enormemente grande cuando en realidad sólo necesita espacio para `300 * el_número_de_cursos` registros. El método de las restas sucesivas obtiene las posiciones aplicando el siguiente algoritmo:

<i>Curso</i>	<i>Clave</i>		<i>Dirección</i>
1º	961001	961001 menos 961000	1
	961002	961002 menos 961000	2
	...		
	961300	961700 menos 961000	300
2º	962001	962001 menos 962000 mas 300	301
	962002	962002 menos 962000 mas 300	302

Aritmética modular

Consiste en efectuar la división entera de la clave por el tamaño del rango del índice y tomar el resto (direcciones desde 0 a N - 1). Por ejemplo, si la clave es un número entero y el tamaño del rango del índice es N se podría crear la siguiente función:

entero función hash(*E entero*: clave)

inicio

devolver (clave *mod* N)

```
// direcciones desde 0 a N-1
```

fin_función

También es frecuente sumarle uno al resto para, así, obtener direcciones en el rango 1 a N. Para lograr una mayor uniformidad en la distribución es conveniente que N sea un número primo.

EJEMPLO 12.2. Se desea elegir el tamaño de la tabla de dispersión para una aplicación en la que se deben almacenar $T = 900$ registros y calcular la posición que ocuparían los elementos cuyos campos clave son: 245643, 245981 y 257135.

Una buena elección de N , en este supuesto, es 997 al ser un número primo próximo y tener como factor de carga (T/N) aproximadamente 0.9 cuando se hayan guardado todos los elementos.

Teniendo en cuenta el valor de N , se aplica la función hash de aritmética modular y se obtienen estas direcciones:

```
hash (245643)= 245643 mod 997 = 381
hash (245981)= 245981 mod 997 = 719
hash (257135)= 257135 mod 997 = 906
```

Mitad del cuadrado

Consiste en elevar al cuadrado la clave y tomar como dirección los dígitos que ocupan una determinada posición, siempre la misma. El número de dígitos a tomar queda determinado por el rango del índice.

EJEMPLO 12.3. *Aplicar el método de Mitad del Cuadrado al registro con clave 245643 para una tabla de 1000 posiciones.*

$245643 \rightarrow 245643^2 \rightarrow 60340483449 \rightarrow$ (dígitos 4, 5 y 6 por la derecha) 483

Truncamiento

Consiste en ignorar parte de la clave y utilizar la parte restante directamente como índice.

EJEMPLO 12.4. *Aplicar el método de truncamiento a la clave 72588495 para una tabla de 1000 posiciones.*

Se asume que las claves son enteros de ocho dígitos, como la tabla tiene mil posiciones, entonces el primero, segundo y quinto dígitos desde la derecha pueden formar la función de conversión y la posición que correspondería a la clave indicada es 895.

Plegamiento

La técnica del plegamiento consiste en la división de la clave en diferentes partes y su combinación en un modo conveniente (a menudo utilizando suma o multiplicación) para obtener el índice. La clave x se divide en varias partes x_1, x_2, \dots, x_n donde cada una, con la única posible excepción de la última, tiene el mismo número de dígitos que la dirección especificada y la función *hash* se define como la suma de todas ellas. En esta operación se desprecian los dígitos más significativos que se obtengan del arrastre o acarreo.

EJEMPLO 12.5. *Aplicar el método de plegamiento para claves de seis dígitos, como 245643, y una tabla de 1000 posiciones.*

Las claves se plegarán en dos grupos de tres y tres dígitos, y la dirección de la clave indicada será $245 + 643 = 888$

Multiplicación

La dispersión de una clave utilizando el método de la multiplicación genera direcciones en tres pasos. Primero, multiplica la clave por una constante real, especialmente seleccionada y comprendida entre 0 y 1, que suele ser la inversa de la razón áurea 0.6180334; en segundo lugar, determina la parte decimal del producto obtenido; y por último, multiplica el tamaño de la tabla, n , por ese número decimal y trunca el resultado para obtener un número entero en el rango $0 \dots n-1$.

EJEMPLO 12.6. *Indique la dirección que correspondería a la clave $x = 245981$ en una tabla de 1000 posiciones utilizando como constante real multiplicativa la razón áurea.*

La dirección sería la 473 y los cálculos efectuados para su obtención:

```
0.6180334 * 245981 → 152024.4738
152024.4738 → ParteEntera (152024.4738) → 0.4738
1000 * 0.4738 → ParteEntera (473.8) → 473
```

12.3. Tratamiento de sinónimos

El empleo de las funciones anteriormente mencionadas puede originar que a dos registros con claves diferentes les corresponda la misma dirección relativa. Cuando a un registro le corresponde una dirección que ya está ocupada se dice que se ha producido una *colisión* o *sinónimo* y se puede optar por:

- Crear una zona especial, denominada *zona de excedentes*, donde llevar exclusivamente estos registros. La *zona de desbordamiento*, excedentes o sinónimos podría encontrarse a continuación de la zona de datos en posiciones que no puedan ser devueltas por la función *hash* empleada.
- Buscar una nueva dirección libre en el mismo espacio donde se están introduciendo todos los registros, zona de datos, para el registro colisionado. Como métodos aplicables en este caso para resolver el problema de las colisiones se detallarán dos: exploración de direcciones y encadenamiento.

EJEMPLO 12.7. *Obtener, en una tabla de rango 997 con la función hash del módulo, las direcciones correspondientes a las claves 245643, 350328:*

$$\begin{aligned}\text{hash}(245643) &= 245643 \bmod 997 = 381 \\ \text{hash}(350328) &= 350328 \bmod 997 = 381\end{aligned}$$

Como se puede observar, generan colisión.

Exploración de direcciones.

La exploración de direcciones resuelve las colisiones mediante la búsqueda, en el mismo espacio donde se están introduciendo todos los registros (zona de datos), de la primera posición libre que siga a aquella donde debiera haber sido colocado el registro (p) y en la que no se pudo situar por encontrarse ocupada. El archivo se considera circular, las primeras posiciones siguen a las últimas como en los *arrays* circulares. La búsqueda de dicha primera posición libre se puede efectuar:

- Por prueba lineal. Se recorren las direcciones $p+1$, $p+2$, $p+3$, ..., $p+i$.
- Por prueba cuadrática. Se recorren las direcciones $p+1$, $p+4$, $p+9$, ..., $p+i^2$.
- Doble direccionamiento *hash*. Se utiliza una segunda función que devolverá un valor p' . Se recorren las direcciones $p+p'$, $p+2*p'$, $p+3*p'$, ..., $p+i*p'$.

El encadenamiento.

Este método para resolver colisiones de claves con la misma dirección *hash* se basa en crear un vector con tantos elementos como valores pueda devolver la función *hash* utilizada. Cada elemento del vector será un puntero a una lista enlazada y cada lista enlazada almacenará los elementos colisionados; es decir aquellos a los que corresponde el mismo valor de función *hash*. En la Figura 12.1 puede verse la estructura de datos necesaria

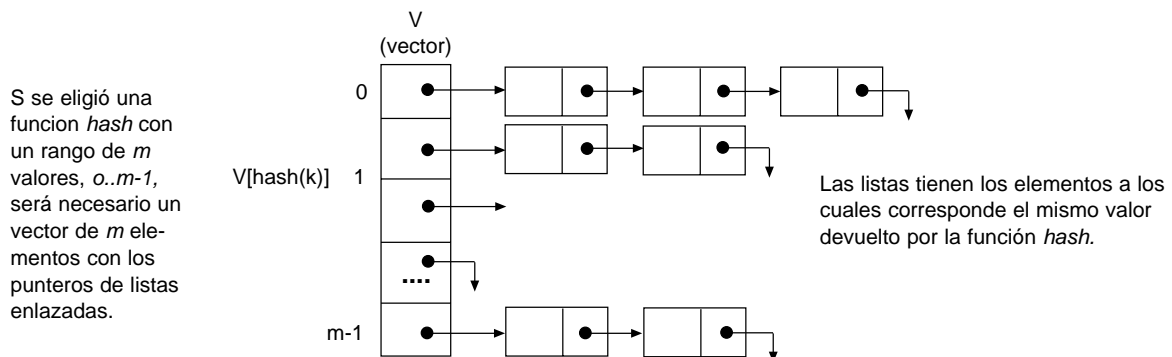


Figura 12.1. Encadenamiento.

La búsqueda de un registro R no es más que la búsqueda de un nodo en una lista enlazada. Con la función $\text{hash}(k)$ se obtendrá la lista que le corresponde, después se busca en dicha lista.

La eliminación de un registro puede consistir en localizar, aplicando la función *hash*, su lista enlazada en el vector de listas y, a continuación, eliminarlo de la misma. Es una baja lógica, físicamente se deja en el archivo.

PROBLEMAS BÁSICOS

- 12.1.** Implementar una tabla dispersa donde almacenar la siguiente información sobre una serie de productos: “código de producto”(tipo cadena), “proveedor”(cadena), “precio”(entero) y “cantidad”(entero). El número máximo de productos sobre los que se considera va a ser necesario guardar información es 100. Utilizar una función hash aritmética modular y el método de exploración lineal para la resolución de colisiones.

Análisis

Crear. Se construye la tabla mediante un vector de punteros con referencias a la estructura que indica el enunciado. El vector se creará con 127 posiciones, por lo que siempre habrá sitio para almacenar los datos, y se comenzará inicializando todas ellas a NULL.

Insertar. Para dar un alta o insertar un nuevo producto, se busca el registro y si no está se procede al alta. Para ello, primero se calcula la dirección mediante la función *hash*. Si está libre, se introducen los datos, si no, se soluciona la colisión buscando el primer registro vacío.

Para *solucionar las colisiones* se utiliza el método de prueba lineal. Desde la posición asignada por la función *hash*, se hace una búsqueda secuencial hasta encontrar el primer registro vacío. Se considerará la tabla como circular, es decir, si en la búsqueda se llega al registro `MaxReg` se pasa al registro número 1 para continuar el proceso.

Consultas. Para resolver un registro, después de leer la clave buscada, se utiliza la función `buscar` para ver si se encuentra en la tabla. Si existe, se utilizará el procedimiento `salidaDatos()` para mostrar el contenido del registro encontrado.

Borrar. Los elementos que se dan de baja van a permanecer en la tabla, por ello se añade un nuevo campo, *esta*, que si se encuentra activo indica que la estructura está dada de alta, en caso contrario se dio de baja.

Codificación

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MaxReg 127

typedef struct
{
    char cod_prod[4], proveedor[31];
    int precio, cantidad;
    int esta;
}Producto;

typedef struct
{
    Producto* tabla[MaxReg];
    int numElementos;
    double factorCarga;
}TablaDispersa;

void crear(TablaDispersa * t)
{
    int j;

    for( j = 0; j < MaxReg ; j++)
        t -> tabla[j] = NULL;
    t -> numElementos = 0;
```

```

    t -> factorCarga = 0.0;
}

int transforma(const char* clave)
{
    int j;
    int d = 0;
    for (j = 0; j < strlen(clave); j++)
    {
        d = d + clave[j];
    }
    return d;
}

int direccion(TablaDispersa t, const char* clave)
{
    int p;
    int d;

    d = transforma(clave);
    // se aplica aritmética modular para obtener dirección base
    p = d % MaxReg;
    while (t.tabla[p] != NULL && t.tabla[p]->esta)
    {
        p = p + 1;
        p = p % MaxReg; // considera array como circular
    }

    return p;
}

Producto* buscar(TablaDispersa t, const char* clave)
{
    Producto* pr;
    int p;
    int d, cont;
    int encontrado;

    d = transforma(clave);
    // se aplica aritmética modular para obtener dirección base
    p = d % MaxReg;
    cont = 1;
    encontrado = 0;
    while (t.tabla[p] != NULL && !encontrado && cont <= MaxReg)
    {
        if (strcmp(t.tabla[p] -> cod_prod, clave) == 0
            && (t.tabla[p] -> esta))
            encontrado = 1;
        else
        {
            p = p + 1;
            p = p % MaxReg; /* considera array como circular */
            cont = cont + 1;
        }
    }
    return encontrado ? t.tabla[p] : NULL;
}

```

```
    }
}
if (encontrado)
    pr = t.tabla[p];
else
    pr = NULL;
return pr;
}

void insertar(TablaDispersa * t, Producto r)
{
    Producto *p;
    int posicion;

    if ((*t).factorCarga < 1)
    {
        p = buscar(*t, r.cod_prod);
        if (p == NULL) //Si no esta
        {
            // se busca una posicion de inserción
            posicion = direccion(* t, r.cod_prod);
            if ((*t).tabla[posicion] == NULL)
                // nunca ha habido elemento en esa posición
                (*t).tabla[posicion] = (Producto*) malloc(sizeof(Producto));
            strcpy((*t).tabla[posicion]->cod_prod, r.cod_prod);
            strcpy((*t).tabla[posicion]->proveedor, r.proveedor);
            (*t).tabla[posicion]->precio = r.precio;
            (*t).tabla[posicion]->cantidad = r.cantidad;
            (*t).tabla[posicion]->esta = 1;
            (*t).numElementos++;
            (*t).factorCarga = ((double)(*t).numElementos)/MaxReg;
            if ((*t). factorCarga > 0.5)
                printf("Factor de    carga = %f \n", (*t).factorCarga);
        }
        else
            printf ("Ya existe\n");
    }
}

void borrar(TablaDispersa * t, const char* clave)
{
    Producto * p;

    p = buscar(*t, clave);
    if (p != NULL)
        p -> esta = 0;
    (* t).numElementos--;
    (* t).factorCarga = ((* t).numElementos)/MaxReg;
}

void salidaDatos(Producto pr)
{

```

```

    printf("Codigo: %s, ", pr.cod_prod);
    printf("Proveedor: %s, ", pr.proveedor);
    printf("Precio: %d, ", pr.precio);
    printf("Cantidad; %d. \n", pr.cantidad);
}

void consulta (TablaDispersa t, const char* clave)
{
    Producto* pr;
    pr = buscar( t, clave);
    if (pr != NULL)
        salidaDatos(*pr);
    else
        printf("No esta\n");
}

void mostrar(TablaDispersa t)
{
    int i;

    for(i = 0 ;i <= MaxReg-1; i++)
        if (t.tabla[i]!= NULL && (t.tabla[i]->esta))
        {
            printf("Posicion %d ",i);
            salidaDatos(*t.tabla[i]);
        }
}

```

- 12.2.** *Implementar las rutinas necesarias para la gestión de un archivo binario para control de personal cuyos registros están formados por los campos DNI (campo clave), nombre del empleado, departamento y sueldo. Se desea acceder a los registros directamente por su clave mediante el empleo de una función hash. El archivo se diseñará para contener 100 registros (más un 27% para colisiones). Utilice una función hash aritmética modular y el método de exploración lineal para la resolución de colisiones.*

Análisis

El ejercicio es análogo al anterior y sólo levemente modificado para que trabaje en memoria externa. El archivo se abre de forma que si no existe se crea y si existe permite la adición de nuevos datos. La creación del archivo implica su apertura en el modo "wb+" y la inicialización de MaxReg posiciones marcando el campo ocupado como libre (L). Puesto que por el método de transformación de claves a registros con claves distintas puede corresponderles la misma posición (colisiones), el campo ocupado será consultado cuando se vayan a efectuar altas con la finalidad de no sobrecribir información existente. Si en la posición devuelta por la función *hash* no se ha escrito nunca, o los datos que allí aparecen están borrados (campo ocupado con una L o una B) se coloca en ella la nueva información, pero si dicha posición está ocupada se efectúa una búsqueda secuencial hasta encontrar una posición válida. Para borrar un registro se marca su campo ocupado con una B. Se incluye un programa principal simplificado para mostrar el funcionamiento del programa.

Codificación

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MaxReg 127

```

```
typedef struct
{
    char dni[9], nombre[31], dep[21];
    int sueldo;
    char ocupado;
}TipoR;

void crear(FILE * f)
{
    int j;
    TipoR r;

    for( j = 0; j < MaxReg ; j++)
    {
        fseek(f, j * sizeof(TipoR), SEEK_SET);
        r.ocupado = 'L';
        fwrite(&r, sizeof(TipoR),1, f);
    }
}

int transforma(const char* clave)
{
    int j;
    int d = 0;
    for (j = 0; j < strlen(clave); j++)
    {
        d = d + clave[j]-'0';
    }
    return d;
}

int direccion(FILE * f, const char* clave)
{
    int p;
    int cont;
    TipoR r;
    int d;

    d = transforma(clave);
    // se aplica aritmética modular para obtener dirección base
    p = d % MaxReg;
    fseek(f, p * sizeof(TipoR), SEEK_SET);
    fread(&r, sizeof(TipoR),1, f);
    cont = 0;

    while (r.ocupado == '0' && cont < MaxReg)
    {
        p = p + 1;
        p = p % MaxReg; // considera el archivo circular
        fseek(f, p * sizeof(TipoR), SEEK_SET);
        fread(&r, sizeof(TipoR),1, f);
        cont++;
    }
}
```



```

    if (cont == MaxReg)
        p = -1;
    return p;
}

int buscar(FILE * f, const char* clave)
{
    TipoR r;
    int p;
    int d;
    int cont;
    int encontrado;

    d = transforma(clave);
    // se aplica aritmética modular para obtener la dirección base
    p = d % MaxReg;
    cont = 1;
    fseek(f, p * sizeof(TipoR), SEEK_SET);
    fread(&r, sizeof(TipoR), 1, f);
    if (strcmp(r.dni, clave) == 0 && (r.ocupado == '0'))
        encontrado = 1;
    else
    {
        encontrado = 0;
        while (r.ocupado != 'L' && !encontrado && cont <= MaxReg)
        {
            fseek(f, p * sizeof(TipoR), SEEK_SET);
            fread(&r, sizeof(TipoR), 1, f);
            if (strcmp(r.dni, clave) == 0 && (r.ocupado == '0'))
                encontrado = 1;
            else
            {
                p = p + 1;
                p = p % MaxReg;                      /* considera el array como circular */
                cont = cont + 1;
            }
        }
    }
    if (! encontrado)
        p = -1;
    return p;
}

void insertar(FILE * f, TipoR r)
{
    int p;
    int posicion;

    p = buscar(f, r.dni);
    if (p == -1) //Si no esta busca una posicion de inserción
    {
        posicion = direccion(f, r.dni);
        if (posicion != -1)

```

```
    {
        fseek(f, posicion * sizeof(TipoR), SEEK_SET);
        r.ocupado = '0';
        fwrite(&r, sizeof(TipoR), 1, f);
    }
    else
        printf ("No hay sitio\n");
    }
    else
        printf ("Ya existe\n");
}

void borrar(FILE * f, const char* clave)
{
    int p;
    TipoR r;

    p = buscar(f, clave);

    if (p != -1)
    {
        r.ocupado = 'B'; // marca el registro con estatus de baja
        fseek(f, p * sizeof(TipoR), SEEK_SET);
        fwrite(&r, sizeof(TipoR), 1, f);
    }
}

void salidaDatos(TipoR r)
{
    printf("DNI: %s\n", r.dni);
    printf("Nombre: %s\n", r.nombre);
    printf("Departamento: %s\n", r.dep);
    printf("Salario: %d\n", r.sueldo);
}

void consulta (FILE * f, const char* clave)
{
    int p;
    TipoR r;

    p = buscar(f, clave);
    if (p != -1)
    {
        fseek(f, p * sizeof(TipoR), SEEK_SET);
        fread(&r, sizeof(TipoR), 1, f);
        salidaDatos(r);
    }
    else
        printf ("No esta\n")
}

void mostrar(FILE * f)
```

```

{
    int i, n;
    TipoR r;

    fseek(f, 0L, SEEK_END);
    n = (int)ftell(f)/sizeof(TipoR);
    for(i = 0 ;i <= n-1; i++)
    {
        fseek(f, i * sizeof(TipoR), SEEK_SET);
        fread(&r, sizeof(TipoR),1, f);
        if (r.ocupado == '0')
        {
            printf("Posicion %d ",i);
            printf("DNI: %s\n", r.dni);
        }
    }
}

int main ()
{
    TipoR r;
    char cod[9];
    char pausa;
    FILE* f;

    f = fopen("Personal.dat","rb+");
    if (f == NULL)
    {
        f = fopen("Personal.dat","wb+");
        if (f == NULL)
        {
            puts("\nError en la operación de abrir archivo.");
            exit(-1);
        }
        crear(f);
    }
    printf("Indique DNI y * para fin ");
    gets(r.dni);
    while (strcmp(r.dni,"*")!= 0)
    {
        printf ("Nombre ");
        gets(r.nombre);
        printf ("Departamento ");
        gets(r.dep);
        printf ("Salario ");
        scanf("%d", &r.sueldo);
        fflush(stdin);
        insertar (f, r);
        printf("\nIndique DNI y * para fin ");
        gets(r.dni);
    }
    mostrar(f);
}

```

```
printf("\nIndique DNI a consultar ");
gets(cod);
consulta(f, cod);
fclose(f);
pausa = getc(stdin);
return 0;
}
```

Ejecución

```
Indique DNI y * para fin 12345678
Nombre Ana Fernandez
Departamento Ventas
Salario 2600
```

```
Indique DNI y * para fin 87654321
Nombre Carlos Lopez
Departamento Ventas
Salario 2600
```

```
Indique DNI y * para fin *
Posicion 36 DNI: 12345678
Posicion 37 DNI: 87654321
```

```
Indique DNI a consultar 12345678
DNI: 12345678
Nombre: Ana Fernandez
Departamento Ventas
Salario 2600
```

PROBLEMAS AVANZADOS

- 12.3.** *Una tienda necesita un programa que le permita almacenar, borrar y consultar la siguiente información sobre los 100 artículos que puede vender: unidades, precio, identificación y fecha de caducidad. El programa deberá efectuar también la creación del archivo. Para mejorar el tiempo de acceso a los registros el archivo se construirá aplicando las técnicas hash y el direccionamiento por encadenamiento para resolver colisiones.*

Análisis

En este problema es necesario crear un vector con tantos elementos como valores pueda devolver la función *hash* utilizada. Cada elemento del vector será un puntero a una lista enlazada, y cada lista enlazada almacenará las diversas posiciones donde se han situado en el archivo de datos los registros a los que corresponde un determinado valor de función *hash*. Los registros, cuando se vayan añadiendo, se colocarán siempre en la primera dirección disponible del archivo. En la Figura 12.2 puede verse la estructura de datos necesaria.

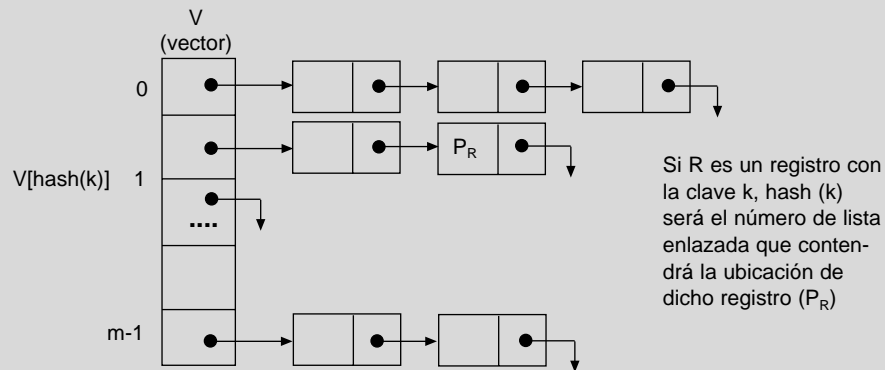


Figura 12.2. Estructura multienlazada.

Al añadir un registro R con clave k , éste se almacenará en la primera posición libre del archivo y, a continuación, se colocará dicha posición en la lista $V[\text{hash}(k)]$. La posición que ocupa el registro R en el archivo de datos no guarda relación el valor devuelto por $\text{hash}(k)$, pues, como se ha dicho, el registro se sitúa siempre en la primera posición libre del archivo.

La búsqueda de un registro R no es más que la búsqueda de un nodo en una lista enlazada. Con la función $\text{hash}(k)$ se obtendrá la lista que le corresponde, después se busca en dicha lista.

La eliminación de un registro puede consistir en localizar, aplicando la función hash , su lista enlazada en el vector de listas y, a continuación, eliminarlo de la misma. Es una baja lógica; físicamente se deja en el archivo.

Al terminar se almacenará en un archivo auxiliar la información necesaria para poder reconstruir el vector de listas la próxima vez que se necesite trabajar con el archivo. Este archivo tiene por tanto todos sus elementos de tipo entero. El primer registro es el número de direcciones hash . En el segundo registro se graba la siguiente posición libre del archivo de datos, para así empezar a grabar a partir de ésta. El resto responden a esta secuencia: índice en el vector, número de registros de la lista y posición de cada registro en el archivo de datos. Repitiéndose esta secuencia por cada índice del vector, que es lo mismo que decir por cada valor de la función hash . Siempre que el trabajo con el archivo de datos termina se sobrescribe este archivo auxiliar y siempre que el proceso se inicie hay que leerlo para reconstruir la estructura multienlazada.

El campo clave es el identificador del artículo, consta de 5 dígitos. La función hash que se aplica responde a la técnica de "mitad del cuadrado": una vez elevado al cuadrado el campo clave, la dirección hash se corresponde con los dos dígitos centrales.

Codificación (Se encuentra en la página Web del libro)

PROBLEMAS PROPUESTOS

- 12.1.** Para comparar los resultados teóricos de la eficiencia de las tablas de dispersión escriba un programa que inserte 1000 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
- 12.2.** Escribir un programa para comparar los resultados teóricos de la eficiencia de las tablas de dispersión con exploración cuadrática. Seguir las pautas marcadas en el problema 12.1.
- 12.3.** 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.
- 12.4.** Se desea almacenar en un archivo los atletas participantes en un Cross popular. El máximo de participantes se tiene establecido en 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, para lo que nos es necesario transformar en valor numérico a la cadena (considerar los caracteres de orden impar, con un máximo de 5). Las colisiones se pueden resolver con el método de prueba lineal.
Las operaciones que contempla el ejercicio son: dar de alta un nuevo registro; modificar datos de un registro; eliminar un registro y búsqueda de un atleta.
- 12.5.** El archivo de texto EMPLEADOS de una empresa con 125 empleados contiene por línea un entero positivo correspondiente al número de nomina, el apellido, el nombre y el sueldo bruto anual. Realizar un programa que lea el archivo y añada los datos de cada empleado a una tabla *hash* con encadenamiento, tomando como clave el número de empleado. El archivo puede tener claves repetidas, en cuyo caso se considerará la que tenga mayor cantidad de ingresos.
- 12.6.** Escribir la función `posicion()`, que tenga como entrada la clave de un elemento que representa los coches de alquiler (carros) 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.
- 12.7.** 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.
- 12.8.** 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 la aritmética modular: $h(x) = x \bmod m$. 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 lo que se crea la lista en la posición m , y los elementos que se encuentran en la lista 0 se dispersan utilizando la función $h_1(x) = x \bmod 2*m$. La segunda ampliación supone que el tamaño de la tabla sea $m+2$, en el rango de 0 a $m+1$, por lo que se crea la lista de índice $m+1$, y los elementos que se encuentran en la lista 1 se dispersan utilizando la función $h_2(x) = x \bmod 2*(m+1)$. Así sucesivamente se va ampliando la estructura que almacena los elementos de una tabla dispersa enlazada.

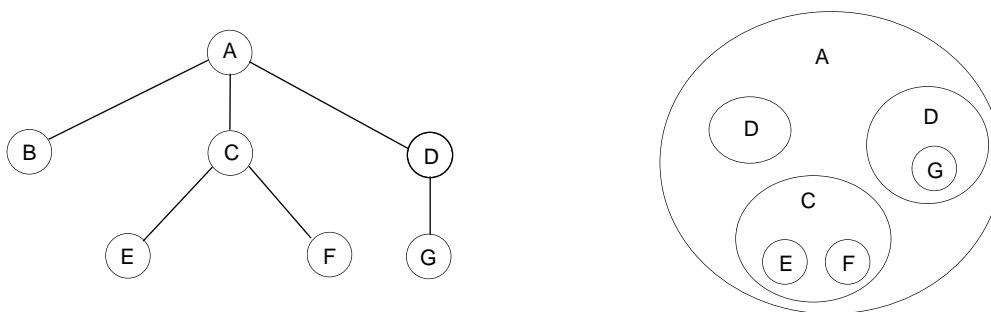
Árboles, árboles binarios y árboles ordenados

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. Los árboles son estructuras no lineales muy utilizados en informática para representar fórmulas algebraicas, como un método eficiente para búsquedas grandes y complejas 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 y proceso de texto.

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.

13.1. Concepto de árbol

Las estructuras de tipo árbol se usan para representar datos con una relación jerárquica entre sus elementos. La definición de árbol implica una naturaleza recursiva, ya que un árbol o es vacío o se considera formado por un nodo raíz y un conjunto disjunto de árboles que se llaman subárboles del raíz. La Figura 13.1 muestra cómo representar gráficamente un árbol.



$A((B) (C((E) (F))) (D((G))))$

Figura 13.1. Representación de árboles.

El concepto de *árbol* requiere una terminología específica, entre la que es preciso destacar la siguiente:

- *Nodos* son los elementos o vértices del árbol.
- Todo árbol que no está vacío tiene un nodo *raíz*, que es aquel del cual derivan o descienden todos los demás elementos del árbol.
- Cada nodo tiene un único antecesor o ascendiente denominado *padre*, excepto el nodo raíz.
- Cualquier nodo, incluido el raíz puede tener varios descendientes, denominados *hijos*, que salen de él.
- Se llama *grado de un nodo* al número de hijos que salen de él y a los nodos con grado 0 se les denomina *nodos terminales* u *hojas*.
- Dos nodos hijos del mismo padre reciben el nombre de *hermanos*.
- Cada nodo de un árbol tiene asociado un número de *nivel* que se determina por el número de antecesores que tiene desde la raíz, teniendo en cuenta que el nivel de la raíz es 1 o bien 0.
- *Profundidad o altura* de un árbol es el máximo de los niveles de los nodos del árbol.
- *Peso* de un árbol es el número de nodos terminales.
- Una colección de dos o más árboles se llama *bosque*.

13.2. Árbol binario

Árbol binario es aquel en el que cada nodo tiene como máximo el grado 2. Un árbol binario es *equilibrado* cuando la diferencia de altura entre los subárboles de cualquier nodo es como máximo una unidad. Cuando los subárboles de todos los nodos tienen todos la misma altura se dice que está perfectamente equilibrado.

Árbol binario *completo* es un árbol equilibrado en el que todos los nodos *interiores*, es decir aquellos con descendientes, tienen dos hijos. Un árbol binario *completo* de profundidad n para cada nivel, del 0 al nivel $n-1$ tiene un conjunto lleno de nodos y todos los nodos hoja a nivel n 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 binario *lleno* tiene todas sus hojas al mismo nivel y sus nodos interiores tienen cada uno dos hijos. Cuando un árbol binario es lleno es, necesariamente, completo. Un árbol binario completo es equilibrado, mientras que un árbol binario lleno es totalmente equilibrado.

13.2.1. CONSTRUCCIÓN DE UN ÁRBOL BINARIO

Los árboles binarios se construyen, al igual que las listas enlazadas, utilizando una serie de elementos con la misma estructura básica. Esta estructura básica es un nodo con un espacio para el almacenamiento de datos y dos punteros que actúan como enlace con sus hijos izquierdo y derecho, estos punteros permiten que cada nodo pueda tener de cero a dos elementos sucesores o siguientes puesto que alguno de ellos o incluso ambos pueden ser NULL. La formación del árbol pasa por la creación de cada uno de los nodos y el establecimiento de los correspondientes enlaces de tal forma que cada nodo, excepto la raíz, tenga un único elemento predecesor y todos los nodos del árbol puedan ser accedidos a través de un puntero que referencia dicho nodo raíz.

13.2.2. RECORRIDOS

Se denomina **recorrido** al proceso que permite acceder una sólo vez a cada uno de los nodos del árbol. Existen diversas formas de efectuar el recorrido de un árbol binario, en anchura y en profundidad.

RECORRIDO EN ANCHURA:

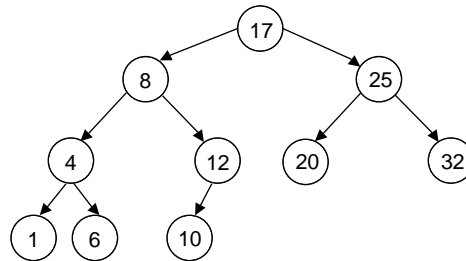
Consiste en recorrer los distintos niveles y, dentro de cada nivel, los diferentes nodos de izquierda a derecha.

Recorrido en profundidad:

- | | |
|------------------|--|
| <i>Preorden</i> | Visitar la raíz, recorrer en <i>preorden</i> el subárbol izquierdo, recorrer en <i>preorden</i> el subárbol derecho. |
| <i>Inorden</i> | Recorrer <i>inorden</i> el subárbol izquierdo, visitar la raíz, recorrer <i>inorden</i> el subárbol derecho. |
| <i>Postorden</i> | Recorrer en <i>postorden</i> el subárbol izquierdo, recorrer en <i>postorden</i> el subárbol derecho, visitar la raíz. |

Estos algoritmos tienen naturaleza recursiva basada en la propia estructura recursiva de los árboles. En cualquiera de ellos recorrer el árbol implica visitar la raíz e invocarse a sí mismo dos veces para efectuar el mismo tratamiento con los subárboles izquierdo y derecho. La condición de salida de la recursividad es que el árbol o subárbol considerado en un determinado momento se encuentre vacío.

Inorden:
1, 4, 6, 8, 10, 12, 17, 20, 25, 32
Preorden:
17, 8, 4, 1, 6, 12, 10, 25, 20, 32
Postorden:
1, 6, 4, 10, 12, 8, 20, 32, 25, 17



Cuando se trata de un árbol binario de búsqueda el recorrido *Enorden* presenta los elementos ordenados. De este tipo de árboles binarios se hablará posteriormente en este mismo capítulo.

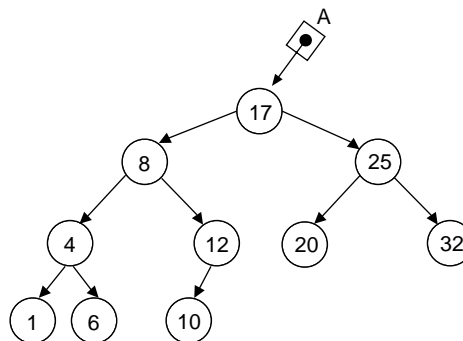
Figura 13.2. Recorridos en profundidad.

Implementación de los recorridos de forma iterativa:

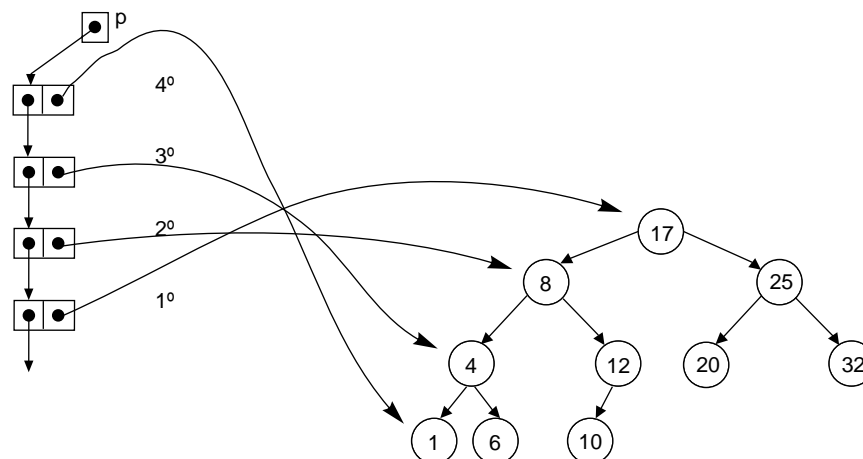
Para implementar el recorrido de un árbol binario en forma no recursiva resulta necesario el uso de estructuras auxiliares, una pila en el caso de los recorridos en profundidad y una cola para el recorrido en anchura, donde se almacenarán punteros a los diversos nodos del árbol.

Recorrido en profundidad. Es posible implementar de forma no recursiva cualquiera de los tres tipos de recorridos en profundidad comentados anteriormente. Para ello se utiliza una pila donde almacenar punteros a los distintos nodos del árbol.

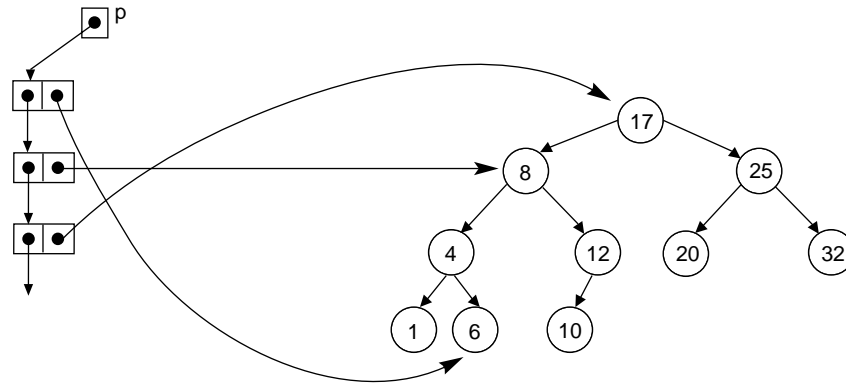
EJEMPLO 13.1. Dado el siguiente árbol, efectuar el seguimiento de su recorrido iterativo en profundidad.



Se van colocando en la pila punteros, a la raíz y los sucesivos hijos izquierdos de cada nodo

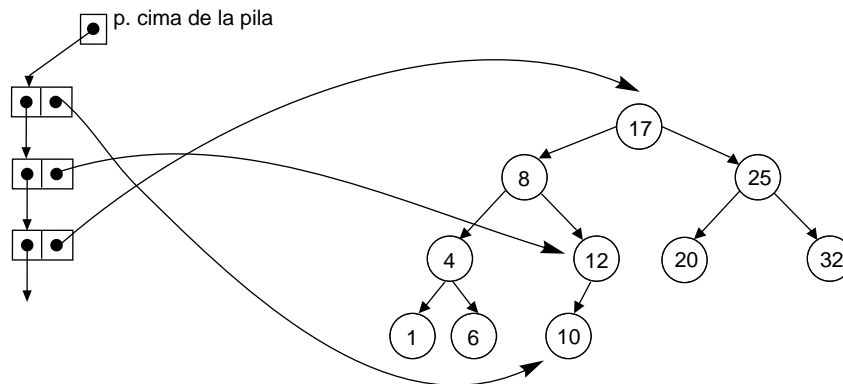


Recupera de la pila y escribe 1. Como no tiene hijo derecho, recupera de la pila y escribe 4.
El hijo derecho de 4 es 6. Pone en la pila el puntero al 6.



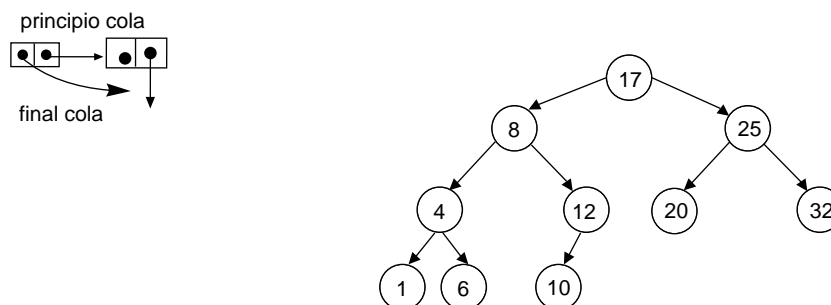
Se recupera de la pila y escribe el 6. Como no tiene hijo derecho, se pasa a recuperar de la pila y escribir el 8.

El 8 tiene un hijo derecho, que se coloca en la pila. Después se coloca en la pila el hijo izquierdo del 12 que será el que se recupere a continuación. El resto del seguimiento se deja como ejercicio al lector.

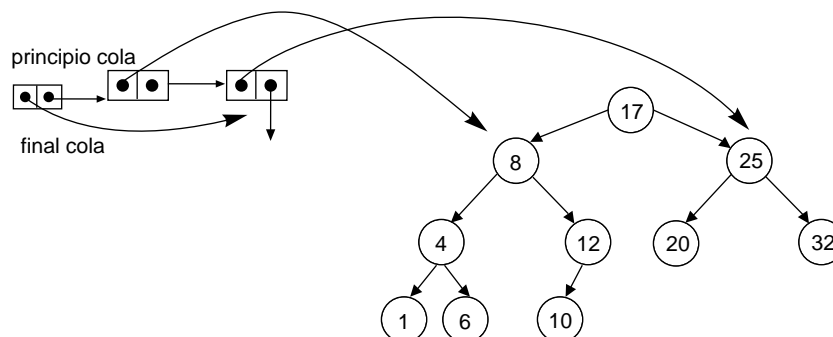


Recorrido en anchura. En el recorrido en anchura se visitan los nodos por niveles, para ello se utiliza una estructura auxiliar tipo cola. El proceso consiste en tomar el puntero a la raíz y ponerlo en la cola, a continuación se repite: quitar el primer elemento de la cola, mostrar el contenido de dicho nodo y almacenar los punteros correspondientes a sus hijos izquierdo y derecho en la cola por el final. De esta forma, al recorrer los nodos de un nivel, mientras se muestra su contenido, se almacenan en la cola los punteros a todos los nodos del nivel siguiente.

EJEMPLO 13.2. Mostrar el seguimiento del recorrido iterativo en anchura para el árbol que se indica a continuación.



Se recupera de la cola el 17 y se colocan en la cola los punteros a sus hijos izquierdo y derecho:



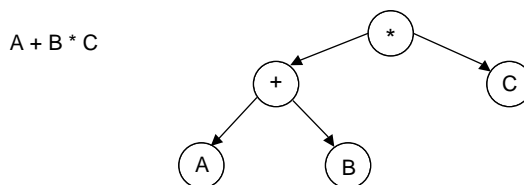
Se recupera de la cola el 8 y se añaden al final de la cola los punteros a sus hijos izquierdo y derecho (4 y 12 respectivamente). Se recupera el primer elemento de la cola, que ahora sería el puntero al 25 y se añaden al final los punteros al 20 y 32. El resto del seguimiento se deja como ejercicio al lector.

13.3. Árboles binarios de expresiones

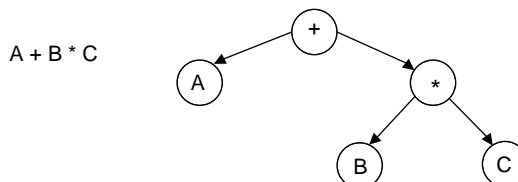
Los árboles binarios se utilizan para almacenar expresiones en memoria. Si se supone que todos los operadores tienen dos operandos, se puede representar una expresión por un árbol binario cuya raíz contiene un operador y cuyos subárboles izquierdo y derecho son los operandos izquierdo y derecho respectivamente. Los operandos podrán ser subexpresiones representadas como subárboles y, los que no son subexpresiones, se almacenan en nodos hojas. En resumen, se puede decir que los árboles de expresión son árboles binarios, cuyas hojas contienen operandos y los otros nodos operadores.

EJEMPLO 13.3. Dibujar un árbol que represente la expresión $(A + B) * C$.

Obsérvese que los paréntesis no se almacenan en el árbol pero están implícitos en la forma del mismo.



Mientras que el árbol dibujado a continuación representa la expresión $A + B * C$.



Las expresiones matemáticas se pueden escribir según diversos tipos de notaciones. La notación *infija* es la empleada habitualmente y requiere el uso de paréntesis, pero únicamente cuando es necesario modificar la prioridad entre los distintos operadores. En la notación prefija los operadores aparecen situados inmediatamente antes de los operandos sobre los cuales actúan y la notación *postfija* a continuación. Los recorridos de un árbol de expresión en *preorden* y *postorden* muestran la expresión en notación *prefija* y *postfija* respectivamente.

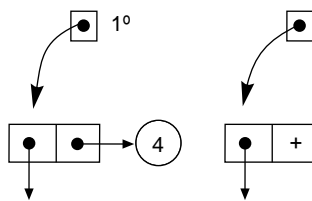
13.3.1. CONSTRUCCIÓN A PARTIR DE UNA EXPRESIÓN EN NOTACIÓN CONVENCIONAL

Para construir un árbol de expresión a partir de una expresión matemática en notación infija se utilizarán, como estructuras de datos auxiliares, una pila capaz de almacenar punteros a los nodos de un árbol y otra donde retener los operadores temporalmente hasta que llegue el momento de incorporarlos al árbol. Los pasos a seguir serían los siguientes:

1. Cuando se lee un operando se crea un árbol de un nodo y se mete el apuntador a él en la correspondiente pila.
2. Cuando se lee un operador se retiene en la pila de operadores. Los operadores se van poniendo en esta pila sucesivamente hasta encontrar uno con menor o igual prioridad que el de la cima, en cuyo caso se sacan los que hubiera en la pila de mayor o igual prioridad y se coloca en ella éste último. Aunque el paréntesis tiene la mayor prioridad sólo se saca cuando aparece un paréntesis derecho. Cuando se acaba la entrada también se saca lo que hubiera en la pila.
3. Al sacar de la pila de operadores uno de ellos, hay que extraer de la de la pila de punteros a nodos, los dos últimos apuntadores. Con éstos tres elementos, un operador y dos punteros, se forma un nuevo árbol cuya raíz almacena el operador y los punteros anteriores. El apuntador a este nuevo árbol se coloca ahora en la pila de apuntadores.
4. El proceso termina cuando se acaba la entrada y la pila de operadores queda vacía.

EJEMPLO 13.4. Dada la expresión: $4 + 5 \wedge (2 * 3) + 8$, donde el operador \wedge significa elevado a, construir el correspondiente árbol.

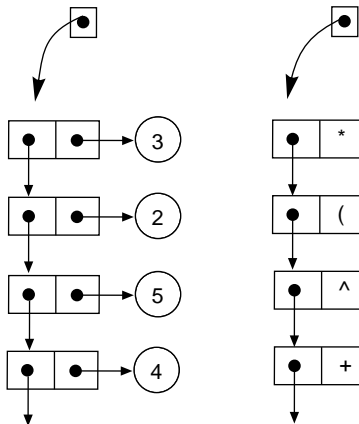
Los pasos a seguir son:



1º

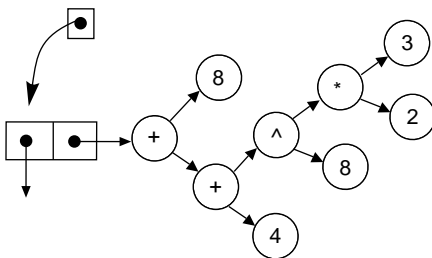
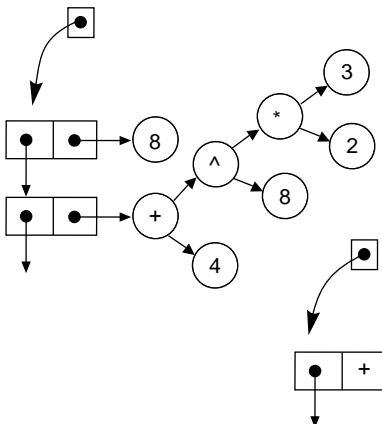
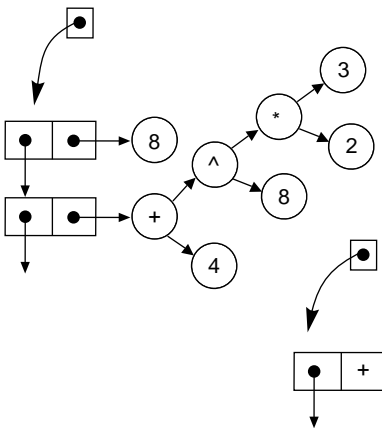
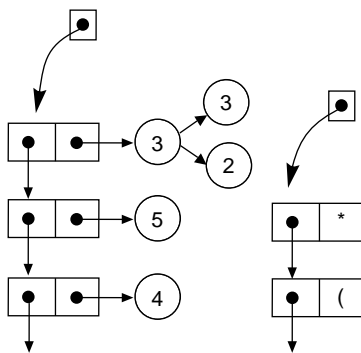
Se lee un operando por lo que se crea un árbol de un nodo y se pone en una pila de árboles.

A continuación se lee un operador que se coloca en una pila de operadores.



2º

Se repiten estas operaciones: los operandos que se van leyendo se colocan en la pila de árboles y los operadores en la pila de operadores.



3º

El paréntesis de cierre saca los operadores de la pila hasta el paréntesis de apertura. Al sacar de la pila de operadores el *, hay que extraer, de la de la pila de árboles, los dos últimos apuntadores. Con estos tres elementos, un operador y dos punteros, se forma un nuevo árbol cuya raíz almacena el operador y los subárboles izquierdo y derecho de dicha raíz serán los punteros anteriores. El apuntador a este nuevo árbol se coloca ahora en la pila de apuntadores.

4º

El + saca de la pila de operadores todos aquellos con mayor o igual prioridad que él, es decir ^ y el otro + y a continuación se coloca él en dicha pila. Cada vez que se saca un operador de la pila de operadores hay que extraer, de la de la pila de árboles, los dos últimos apuntadores, construir un nuevo árbol en la forma anteriormente indicada y colocarlo en la pila de árboles.

5º

El que llega ahora es un operando, el 8, que se coloca en la pila de árboles

6º

Cuando se ha acabado la expresión de entrada se sacan los operadores de su pila hasta que esta queda vacía. Cada vez que se saca un operador de la pila de operadores hay que extraer, de la pila de árboles, los dos últimos apuntadores, construir el nuevo árbol y colocarlo en la pila de árboles.

13.4. Árboles binarios de búsqueda

Un tipo especial de árbol binario es el denominado *árbol binario de búsqueda*, en el que para cualquier nodo su valor es superior a los valores de los nodos de su subárbol izquierdo e inferior a los de su subárbol derecho. Una característica de los árboles binarios de búsqueda es que no son únicos para los datos dados. La Figura 13.3 muestra dos árboles binarios de búsqueda.

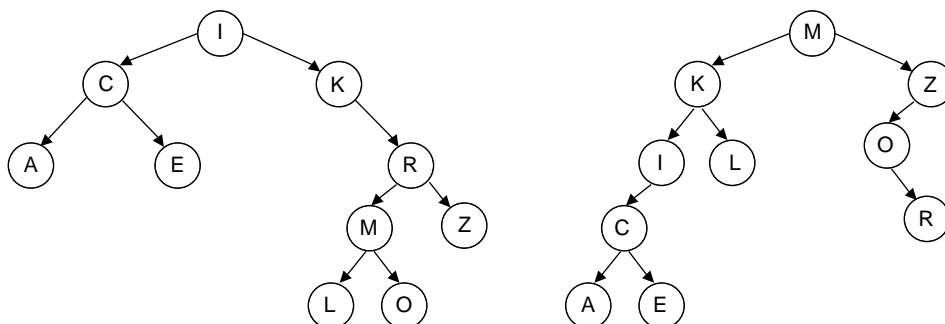


Figura 13.3. Árboles binarios de búsqueda.

La utilidad de este tipo de árbol se refiere a la eficiencia en la búsqueda de un nodo, similar a la búsqueda binaria, y a la ventaja que presentan los árboles sobre los *arrays* en cuanto a que la inserción o borrado de un elemento no requiere el desplazamiento de todos los demás.

La supresión de un nodo requiere la liberación del espacio en memoria ocupado por el mismo. En los ejercicios resueltos aparecen codificados todas estas funciones tanto de forma recursiva como iterativa.

La **búsqueda de un elemento** 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 derecho. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda en el subárbol izquierdo.

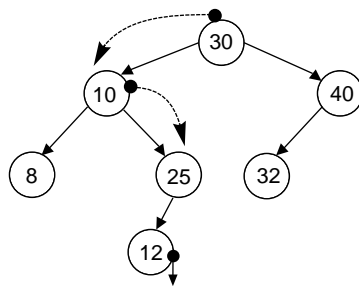
La operación de **inserción de un nodo** es una extensión de la operación de búsqueda y para efectuarla se sigue este algoritmo:

1. Comparar la clave del elemento a insertar con la clave del nodo raíz, si es mayor avanzar hacia el subárbol derecho, si es menor hacia el izquierdo.
2. Repetir el paso anterior hasta encontrar un elemento con clave igual o llegar al final del subárbol donde debiera situarse el nuevo elemento.
3. Cuando se llega al final es porque no se ha encontrado, por tanto se deberá reservar memoria para una nueva estructura nodo, introducir en la parte reservada para los datos los valores del nuevo elemento y asignar nulo a los punteros izquierdo y derecho del mismo. A continuación se colocará el nuevo nodo como hijo izquierdo o derecho del anterior según sea el valor de su clave comparada con la de aquel.

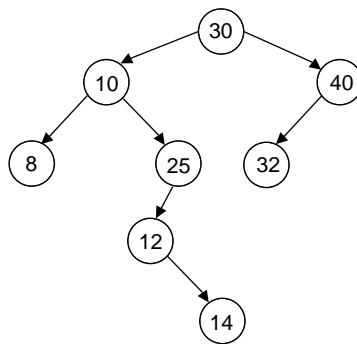
La operación de **eliminación de un nodo** es también una extensión de la operación de búsqueda. El borrado o eliminación de un elemento requerirá, una vez buscado, considerar las siguientes posibilidades:

- Que no tenga hijos, sea hoja. Se suprime, asignando nulo al puntero de su antecesor que antes lo apuntaba a él.
- Que tenga un único hijo. El elemento anterior se enlaza con el hijo del que queremos borrar.
- Que tenga dos hijos. Se sustituye por el elemento más próximo en clave, inmediato superior o inmediato inferior. Para localizar estos elementos debe situarse en el hijo derecho del nodo a borrar y avanzar desde él siguiendo la rama izquierda de cada nodo hasta que a la izquierda ya no haya ningún nodo más, o bien, situarse en el hijo izquierdo y avanzar siguiendo siempre la rama derecha de cada nodo hasta llegar al final.

EJEMPLO 13.5. *Mostrar el camino a seguir para insertar el elemento de clave 14 en el árbol binario de búsqueda a continuación representado.*



El camino seguido se ha marcado sobre el propio árbol inicial y el resultado final es:



PROBLEMAS BÁSICOS

- 13.1.** *Escribir los pseudocódigos para efectuar de manera recursiva el recorrido de un árbol de las siguientes tres formas: **Preorden** (visitar la raíz, recorrer en preorden el subárbol izquierdo, recorrer en preorden el subárbol derecho), **Inorden** (recorrer inorden el subárbol izquierdo, visitar la raíz, recorrer inorden el subárbol derecho), **Postorden** (recorrer en postorden el subárbol izquierdo, recorrer en postorden el subárbol derecho, visitar la raíz).*

(Solución en la página Web del libro)

- 13.2.** *Escriba una función que reciba como parámetro un árbol A y devuelva una copia de él.*

Análisis

Para resolver el ejercicio basta con hacer un recorrido del árbol en **preorden** haciendo una copia de él. El recorrido se implementará de forma recursiva, siguiendo el algoritmo explicado en la parte teórica. A la salida de la recursividad se van construyendo los nodos del nuevo árbol y asignándoles sus valores.

Codificación

```
/* Estructura de un nodo */
```

```
typedef struct Arbol
{
```



```

int info;
struct Arbol* izq;
struct Arbol* der;
}arbol;

void copiar (arbol* a, arbol** acop)
{
    arbol* aiz;
    arbol* ade;
    arbol* acopiz;
    arbol* acopde;
    int e;

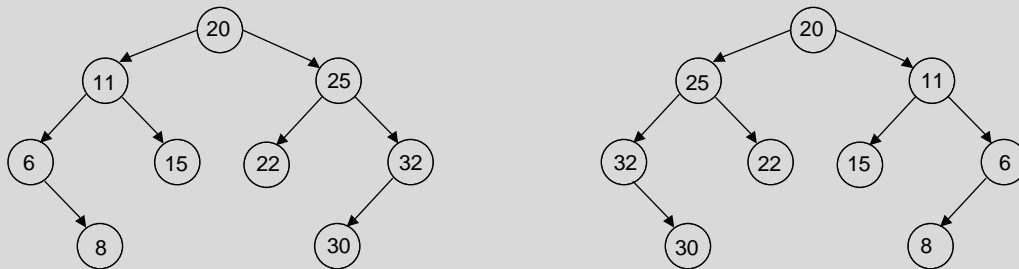
    if (a == NULL)
        *acop = NULL;
    else
    {
        e = a->info;
        aiz = a->izq;
        ade = a->der;
        copiar(aiz, &acopiz);
        copiar(ade, &acopde);
        *acop = (arbol*)malloc(sizeof(arbol));
        (*acop)->info = e;
        (*acop)->izq = aiz;
        (*acop)->der = ade;
    }
}

```

- 13.3.** Escribir una función que reciba como parámetro un árbol A y devuelva como resultado otro árbol Aespejo que es el árbol simétrico de A.

Análisis

Para resolver el ejercicio basta con hacer un recorrido del árbol en preorden haciendo una copia de él, pero intercambiando entre sí los hijos.



Codificación

```

void espejo (arbol* a, arbol** aespejo)
{
    arbol* aiz;
    arbol* ade;
    arbol* aespejoiz;

```

```

arbol* aespejode;
int e;

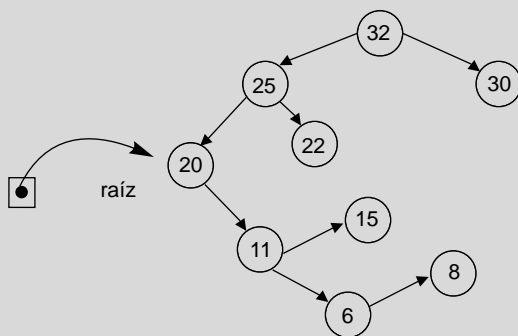
if (a == NULL)
    *aespejo = NULL;
else
{
    e = a->info;
    aiz = a->izq;
    ade = a->der;
    espejo(aiz, &aespejoiz);
    espejo(ade, &aespejode);
    *aespejo = (arbol*)malloc(sizeof(arbol));
    (*aespejo)->info = e;
    (*aespejo)->der = aespejoiz;
    (*aespejo)->izq = aespejode;
}
}

```

Ejecución

	24	
12	3	
		1
Espejo		
		1
	3	
12	24	

13.4. Escribir una función para dibujar un árbol en pantalla de la siguiente forma:

**Análisis**

Para resolver el ejercicio basta con hacer un recorrido del árbol en *inorden*. La columna se obtiene escribiendo los nodos precedidos por una serie de espacios en blanco cuyo número depende del nivel, y la fila se deriva del recorrido recursivo.

Tal y como se ha implementado lo primero que se escribe es el nodo que se encuentra situado lo más a la derecha posible y precedido por los espacios en blanco estipulados para su nivel.

Codificación

```
void dibujar(arbol* a, int nivel)
{
    int i;
    if (a != NULL)
    {
        dibujar(a -> der, nivel+1);
        for (i = 1; i <= nivel; i++) printf("    ");
        printf("%d\n", a -> info);
        dibujar(a -> izq, nivel+1);
    }
}
```

Ejecución

```

      32
    25
  28
    11
      15
      6      8
```

13.5. Diseñar una función que permita comprobar si son iguales dos árboles; es decir, tengan la misma estructura de nodos y éstos el mismo contenido.

Análisis

Se trata de una función recursiva que compara nodo a nodo la información almacenada en ambos árboles. Las condiciones de salida del proceso recursivo serán que:

- se termine de recorrer uno de los dos árboles.
- terminen de recorrerse ambos.
- se encuentre diferente información en los nodos comparados.

Si los árboles terminaron de recorrerse simultáneamente es que ambos tienen el mismo número de nodos y nunca ha sido diferente la información comparada; por tanto, la función devolverá *verdad*, en cualquier otro caso la función devolverá *falso*.

Codificación

```
#define True 1
#define False 0
...
typedef struct Arbol
{
    int info;
    struct Arbol* izq;
    struct Arbol* der;
```

```

} arbol;

/* Implementación de la función */

int iguales(arbol* a, arbol* b)
{
    if (a == NULL)
        if (b == NULL)
            return True;
        else
            return False;
    else
        if (b == NULL)
            return False;
        else
            if (a->info != b->info)
                return False;
            else
                return iguales(a->izq, b->izq) &&
                    iguales(a->der, b->der);
}

```

13.6. Escribir una función que permita contar las hojas de un árbol.

Análisis

Se debe recorrer el árbol, contando, únicamente, los nodos que no tienen hijos. En la llamada inicial, el parámetro `cont` debe valer 0. El tipo `árbol` representa el nodo del árbol (consultar problema 13.5)

Codificación

```

void contarhojas(arbol* a, int* cont)
{
    if (a != NULL)
        if ((a->izq == NULL) && (a->der == NULL))
            *cont = *cont + 1 ;
        else
        {
            contarhojas(a->izq, cont);
            contarhojas(a->der, cont);
        }
}

```

13.7. Diseñar una función que permita determinar la altura de un árbol.

Análisis

Profundidad o altura de un árbol es el máximo de la altura de los nodos del árbol. Esta profundidad se puede obtener utilizando una función que recorra el árbol y en cada nodo seleccione el subárbol de mayor altura y le sume a ésta una nueva unidad. La declaración del tipo `árbol` es la realizada en el problema 13.5.

Codificación

```

int altura(arbol* a)
{
    if (a != NULL)
        return 1 + mayor(altura(a->izq), altura(a->der));
    else
        return 0;
}

int mayor(int n1, int n2)
{
    if (n1 > n2)
        return n1;
    else
        return n2;
}

```

13.8 *Escribir una función que muestre todos los nodos de un árbol que se encuentran en un nivel dado.*

Análisis

Si se considera que la raíz se encuentra en el nivel cero, de un árbol habrá que escribir el nodo cuando el parámetro que indique el nivel sea cero siempre y cuando en cada llamada recursiva disminuya en una unidad el parámetro. La declaración de la estructura `arbol` está escrita en el problema 13.5.

Codificación

```

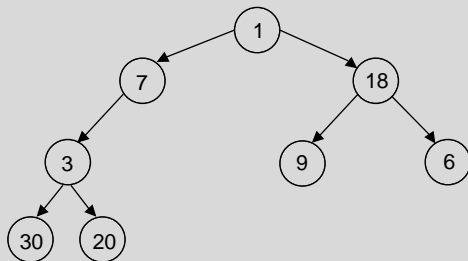
void nodosdeniveldado(arbol* a, int n)
{
    arbol* aiz;
    arbol* ade;
    if (n < 0)
        printf ("error");
    else
        if (n == 0)
        {
            if (a != NULL)
                printf(" %d ", a->info);
        }
        else
            if (a != NULL)
            {
                aiz = a->izq;  ade = a->der;
                nodosdeniveldado(aiz, n - 1);
                nodosdeniveldado(ade, n - 1);
            }
    }

    /* Llamada a la función */

    int nm;
    arbol* c;

```

```
printf("Indique el nivel: ");
scanf(" %d*c ", &nm);
printf("Nodos del nivel %d \n", nm);
nodosdeniveldado(c, nm);
```

Ejecución

- 13.9** Escriba un módulo de programa que muestre todos los nodos de un árbol que estén entre dos niveles dados p y q (ambos inclusive $0 \leq p < q$)

Análisis

Si se considera que la raíz se encuentra en el nivel cero de un árbol, habrá que escribir el nodo cuando el parámetro p sea menor o igual que cero y el q sea mayor o igual que cero, siempre y cuando cada llamada recursiva se disminuya en una unidad los parámetros p y q . Se supone las declaraciones del problema 13.5.

Codificación

```
void arbolentre(arbol* a, int p, int q)
{
    if (a != NULL)
    {
        if (p > 0)
        {
            arbolentre(a->izq, p - 1, q - 1);
            arbolentre(a->der, p - 1, q - 1);
        }
        else
        {
            if ((p <= 0) && (q >= 0))
            {
                printf("%d, ", a->info);
                arbolentre(a->izq, p - 1, q - 1);
                arbolentre(a->der, p - 1, q - 1);
            }
        }
    }
}
```

Ejecución (para el árbol del problema 13.8 y entre el nivel 1 y el 3)

7, 18, 3, 9, 6, 30, 20

- 13.10.** Escriba un pseudocódigo que efectúe el recorrido en anchura de un árbol de forma iterativa, visitando los nodos del mismo por niveles.

Solución (En página Web del libro)

13.11. *Escriba un pseudocódigo que efectúe el recorrido inorden de un árbol de forma iterativa.*

Solución

Implementación de los recorridos de forma iterativa:

Para implementar el recorrido de un árbol binario en forma no recursiva resulta necesario el uso de estructuras auxiliares, una pila en el caso de los recorridos en profundidad y una cola para el recorrido en anchura, donde se almacenarán punteros a los diversos nodos del árbol.

```

tipo
  puntero_a NodoPilaArboles: PilaArboles
  registro: NodoPilaArboles
  Arbol: info
  PilaArboles: sig
  fin_registro

procedimiento EnOrdenIterativo(E Arbol: A)
  var
    PilaArboles: p
    Arbol: b
  inicio
    InicializarPila(p)
    b ← A
  repetir
    mientras <no se alcance el final de la rama izquierda
      que parte del nodo apuntado por b> hacer
        <guardar en la pila los hijos izquierdos que vayan apareciendo en cada
          subárbol, comenzando por el propio b>
    fin_mientras
    si no VacíaPila(p) entonces
      <se desapila un elemento y se le asigna al puntero b, escribiéndose la
        información apuntada por él>
      <se traslada b para que apunte a su propio hijo derecho>
    fin_si
  hasta_que VacíaPila(p) and vacío(b)
fin_procedimiento

```

PROBLEMAS AVANZADOS

13.12. *Elaborar un programa que lea desde teclado una expresión en notación infija y únicamente con los paréntesis mínimos necesarios para su correcta interpretación y genere el árbol de expresión correspondiente. El programa debe además recorrer el árbol y, como resultado de dicho recorrido, presentar por pantalla la expresión, en cualquier tipo de notación (infija, prefija o postfija). Por último debe efectuar su evaluación.*

*Los posibles operadores son: +, -, *, /, ^.*

Análisis

Las estructuras de datos auxiliares necesarias para resolver el problema serán una pila de caracteres donde almacenar los operadores y una pila para almacenar punteros a los nodos del árbol.

La expresión será leída de teclado mediante una variable de tipo cadena que almacenará caracteres alfanuméricos. Dichos caracteres se irán analizando uno por uno y, según su valor, se les considerará operadores u operandos. Se supone que los operandos se encuentran representados mediante letras mayúsculas, los operadores pueden ser $+$, $-$, $*$, $/$, potenciación $^$ y que, dado que la expresión se proporciona en notación infija, podrán aparecer en ella paréntesis de apertura y cierre. Para que se pueda evaluar la expresión es necesario disponer de los valores numéricos de los operandos. El valor numérico de un operando se encuentra en la correspondiente posición del vector `double operandos[26]`, de tal forma que si los operandos son A, D, G, los valores se guardan en las posiciones: `operandos[0]`, `operandos[3]`, `operandos[6]` respectivamente.

Los pasos para construir el árbol de expresiones fueron comentados en la pregunta teórica correspondiente. Allí se veía que, para su construcción, era preciso comparar la prioridad de los operadores y considerar que un operador extrae de la pila de operadores todos aquellos que encuentra con mayor o igual prioridad que él. También se comentó que el paréntesis es un caso especial, pues aunque tiene la mayor prioridad sólo se saca cuando aparece un paréntesis derecho. Esto se puede solucionar considerando prioridades distintas para este operador según esté dentro o fuera de la pila, así dentro de la pila se le asigna el valor más bajo para que ningún otro operador lo pueda extraer y fuera de la pila el valor más alto para que tampoco él pueda extraer a ningún otro. Las prioridades para que todo esto se efectúe del modo adecuado se otorgan en las funciones `prioridadd` y `prioridadf` que gestionan

Operador	Prioridad dentro	Prioridad fuera
'('	-1	5
'^'	3	4
'/', '*',	2	2
+', '-',	1	1

respectivamente la prioridad de los operadores dentro y fuera de la pila, es decir cuando se toma el operador de la expresión. Aunque el enunciado del problema sólo pide una de ellas, el programa recorrerá el árbol para presentar la expresión almacenada en los tres tipos de notaciones infija, prefija y postfija. La forma habitual de escribir operaciones aritméticas es situando el operador entre sus dos operandos, la llamada notación infija. Esta forma de notación obliga en muchas ocasiones a utilizar paréntesis para indicar el orden de evaluación.

Estos paréntesis obligados son los únicos que proporcionará el usuario cuando teclee la expresión. Para mostrar la expresión almacenada en el árbol en notación infija se necesitará efectuar un recorrido en `inorden` y mandar escribir directamente los paréntesis en el lugar adecuado, si las prioridades que indica dicho recorrido difieren de las generales adoptadas por omisión.

La notación en la que el operador se coloca delante de los dos operandos se denomina *prefija* o *polaca* (en honor del matemático polaco que la estudió) y, en ella, no es necesario el uso de paréntesis al escribir la expresión, pues el orden en que se van a realizar las operaciones está determinado por la posición de los operadores y operandos en la expresión. Coincide con el recorrido del árbol en `preorden`.

La notación *postfija* o *polaca inversa* coloca el operador a continuación de sus dos operandos. Efectuando un recorrido `postorden` del árbol se obtiene la expresión en esta notación.

La función `evaluar`, recursiva, opera el resultado de evaluar dos subexpresiones según las reglas establecidas en operar. El algoritmo de evaluación es aplicable para los operadores algebraicos binarios: $+$, $-$, $*$, $/$ y potenciación ($^$).

Codificación (En página Web del libro)

- 13.13.** Implementar las funciones necesarias para manipular un árbol binario de búsqueda cuyos nodos almacenan números enteros. Implementar todos ellos de forma recursiva.

Análisis

Las funciones básicas que se implementarán: inicializar, vacío, insertar, buscar, borrar, eliminar un nodo y recorrer el árbol. Ya han sido descritos anteriormente, por lo que no se repite su explicación. Los subprogramas `construir` y `menor` actúan como auxiliares en estas funciones. Así `construir` se encarga específicamente de crear el nuevo nodo necesario en la inserción y `menor` busca la menor de las claves en el árbol o subárbol que recibe como argumento y se invoca desde la función `eliminar` cuando se desea eliminar un nodo con dos ramas no vacías.

Codificación

```
/* Arbol binario de busqueda. Implementación recursiva */
```



```
#include <stdio.h>
#include <stdlib.h>
#define True 1
#define False 0

typedef struct Arbol
{
    int info;
    struct Arbol* izq;
    struct Arbol* der;
}arbol;

void inicializar(arbol ** a);
int vacio(arbol* a);
void insertar(arbol ** a, int e);
void borrar(arbol ** a, int c);
void buscar(arbol * a, int c, int * encontrado);
void recorreras(arbol * a);
void recorrerdes(arbol * a);
arbol* construir(arbol * a, int e, arbol * b);
void menor(arbol* a, int * e);
void eliminar( arbol ** a);

void inicializar(arbol ** a)
{
    *a = NULL;
};

int vacio(arbol* a)
{
    return a == NULL;
}

void buscar(arbol * a, int c, int * encontrado)
{
    if (vacio(a))
        * encontrado = False;
    else
        if (a->info == c)
            *encontrado = True;
        else
            if (a->info > c)
                buscar(a->izq, c, encontrado);
            else
                buscar(a->der, c, encontrado);
}

arbol* construir(arbol * a, int e, arbol * b)
{
    arbol* nuevo;

    nuevo = (arbol*)malloc(sizeof(arbol));
    nuevo->info = e;
```

```
nuevo->izq = a;
nuevo->der = b;
return nuevo;
}

void insertar(arbol ** a, int e)
{
    if (vacio(*a))
        *a = construir(NULL, e, NULL);
    else
        if ((*a)->info > e)
            insertar(& (*a)->izq, e);
        else
            if ((*a)->info < e)
                insertar(& (*a)->der, e);
}

void menor(arbol* a, int* e)
{
    if (a->izq == NULL)
        *e = a->info;
    else
        menor(a->izq, e);
}

void eliminar( arbol ** a)
{
    arbol* auxi;
    int e;

    if ((*a)->izq == NULL)
    {
        auxi = *a; *a = (*a)->der;
        free (auxi);
    }
    else
        if ((*a)->der == NULL)
        {
            auxi = *a; *a = (*a)->izq;
            free (auxi);
        }
        else
        {
            menor((*a)->der, & e); (*a)->info = e;
            borrar(&(*a)->der, e);
        }
}

void borrar(arbol ** a, int c)
{
    if (*a != NULL )
        if ((*a)->info == c)
            eliminar(a);
        else
```

```

        if ((*a)->info > c)
            borrar(&(*a)->izq, c);
        else
            borrar(&(*a)->der, c);
    }

void recorreras(arbol * a)
{
    if (! vacio(a))
    {
        recorreras(a->izq);
        printf(" %d ", a->info);
        recorreras(a->der);
    }
}

void recorrerdes(arbol * a)
{
    if (! vacio(a))
    {
        recorrerdes(a->der);
        printf(" %d ", a->info);
        recorrerdes(a->izq);
    }
}

/* función main para probar el buen funcionamiento */

int main()
{
    int nm;
    arbol* a;
    char pausa;

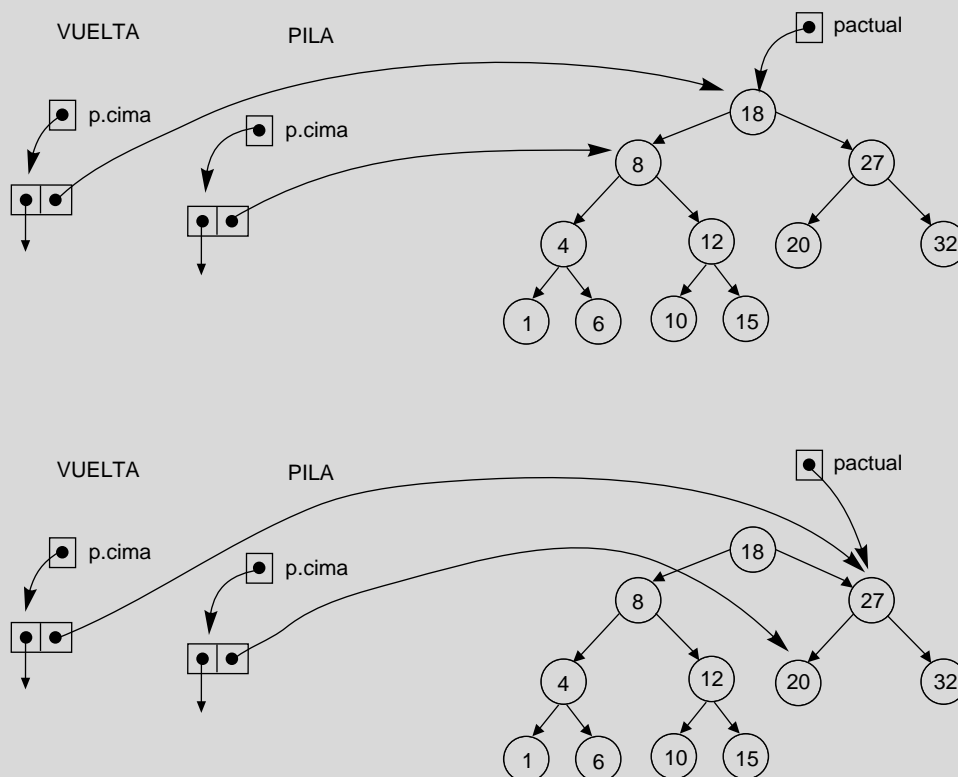
    inicializar (&a);
    printf("Deme numero (0 -> Fin): ");
    scanf("%d%c ", &nm);
    while (nm !=0)
    {
        insertar(&a,nm);
        printf("Deme numero (0 -> Fin): ");
        scanf("%d%c ", &nm);
    }
    recorrerdes(a);
    puts("");
    recorreras(a);
    printf("\n Pulse cualquier tecla para continuar ... ");
    scanf("%c%c",&pausa);
    return 0;
}

```

13.14. Codificar de forma iterativa los procedimientos y funciones necesarios para manipular un árbol binario de búsqueda.

Análisis

Aunque los árboles tienen naturaleza recursiva y en consecuencia las operaciones sobre ellos también lo son, es posible realizarlas de forma iterativa. Los procedimientos de recorrido iterativos preorden y postorden implementados necesitan pilas de árboles auxiliares y el recorrido por niveles una cola de árboles auxiliar. Por ejemplo, el recorrido postorden del siguiente árbol sería: 1, 6, 4, 10, 15, 12, 8, 20, 32, 27, 18. Se implementará haciendo uso de dos pilas auxiliares: *vuelta* y *pila*.



En un principio *pactual* se sitúa en la raíz. Con *pactual* se va recorriendo la rama derecha de cada nodo y cuando no se puede seguir se desapila el último puntero colocado en *pila*, se asigna a *pactual* y se repite el proceso. En *pila* se guardan los correspondientes hijos izquierdos, y en *vuelta* punteros a todos los nodos visitados por *pactual*.

El recorrido que en el ejemplo efectuaría *pactual* es 18, 27, 32, (recuperación de *pila*) 20, (recuperación de *pila*) 8, 12, 15, (recuperación de *pila*) 10, (recuperación de *pila*) 4, 6, (recuperación de *pila*) 1.

Cuando *pila* se queda vacía y *pactual* apunta a NULL se vacía la pila denominada *vuelta* y se muestra.

1, 6, 4, 10, 15, 12, 8, 20, 32, 27, 18.

El recorrido *preorden* sólo necesita una pila auxiliar donde ir guardando los hijos derechos de cada nodo. La función comienza situando *pactual* en la raíz. El puntero *pactual* muestra la información, guarda el hijo izquierdo del nodo visitado en *pila* y se se desplaza al siguiente hijo izquierdo, cuando no puede seguir, se ha llegado al final de esa rama izquierda, se desapila el último puntero colocado en *pila*, se asigna a *pactual* y repite el proceso. El recorrido que efectuaría *pactual* en el ejemplo es: 18, 8, 4, 1, 6, 12, 10, 15, 27, 20, 32.

El recorrido por niveles fue explicado en la parte teórica de este capítulo, por lo que no se detalla ahora.

Codificación

```

/* Arbol binario de busqueda. Implementación iterativa */

#include <stdio.h>
#include <stdlib.h>
#define True 1
#define False 0

typedef struct Arbol
{
    int info;
    struct Arbol* izq;
    struct Arbol* der;
}arbol;

/* Estructura para las pilas auxiliares. Pilas de árboles. */

typedef struct Pilas
{
    arbol* info;
    struct Pilas* cimaant;
}pilas;

/* Estructuras para las cola auxiliar. Cola de árboles. */

typedef struct Nodoc
{
    arbol* info;
    struct Nodoc* sig;
}nodoc;

typedef struct
{
    nodoc* prim;
    nodoc* ulti;
}colas;

/* Rutinas para manipular el árbol*/
void inicializar(arbol ** a);
int vacio(arbol* a);
void insertar(arbol** a, int e);
arbol* construir(arbol* a, int e, arbol* b);
void borrar(arbol** a, int c);
void menor(arbol* a, int* e);
void eliminar( arbol ** a);
void buscar(arbol** a, int c, int* encontrado);
void recorrerar(arbol* a);
void recorrerdes(arbol* a);
void recorreniveles(arbol* a);
void preorden(arbol* a);
void postorden(arbol* a);

/* funciones para el manejo de las pilas auxiliares */

```

```
void inicializarpa( pilas** p);
int vaciapa(pilas* p);
void apilarpa(pilas** p, arbol* e);
void desapilarpa(pilas** p, arbol** e);

/* funciones para el manejo de la cola auxiliar */

void inicializarca(colas* c);
int vaciaca(colas c);
void ponerca(colas* c, arbol* e);
void quitarca(colas* c);
void primeroca(colas c, arbol** e);

/* Implementación de las funciones para manipular el árbol */

void inicializar(arbol ** a)
{
    *a = NULL;
}

int vacio(arbol* a)
{
    return a == NULL;
}

void buscar(arbol** a, int c, int * encontrado)
{
    arbol* pactual;

    pactual = *a; *encontrado = False;
    while ((pactual != NULL) && ! encontrado)
        if (pactual->info == c)
        {
            *encontrado = True;
            printf(" %d ", pactual->info);
        }
        else
            if (pactual->info > c)
                pactual = pactual->izq ;
            else
                pactual = pactual->der;
    }

arbol* construir(arbol* a, int e, arbol* b)
{
    arbol* nuevo;

    nuevo = (arbol*)malloc(sizeof(arbol*));
    nuevo->info = e;
    nuevo->izq = a;
    nuevo->der = b;
    return nuevo;
}
```

```

void insertar(arbol ** a, int e)
{
    arbol* pactual;
    int encontradositio;
    if (vacio(*a))
        *a = construir(NULL, e, NULL);
    else
    {
        pactual = *a; encontradositio = False;
        while (! encontradositio)
            if (pactual->info == e)
                encontradositio = True;
            else
                if (pactual->info > e)
                    if (pactual->izq == NULL)
                    {
                        pactual->izq = construir(NULL, e, NULL);
                        encontradositio = True;
                    }
                    else
                        pactual = pactual->izq;
                else
                    if (pactual->der == NULL)
                    {
                        pactual->der = construir(NULL, e, NULL);
                        encontradositio = True;
                    }
                    else
                        pactual = pactual->der;
    }
}

void menor(arbol* a, int * e)
{
    arbol* pactual;

    pactual = a;
    while (pactual->izq != NULL)
        pactual = pactual->izq;
    *e = pactual->info;
}

void eliminar( arbol ** a)
{
    arbol* auxi;
    int e;

    if ((*a)->izq == NULL)
    {
        auxi = *a; *a = (*a)->der;
        free(auxi);
    }
    else

```

```
    if ((*a)->der == NULL)
    {
        auxi = *a; *a = (*a)->izq;
        free(a);
    }
    else
    {
        menor((*a)->der, &e); (*a)->info = e;
        borrar(&(*a)->der, e);
    }
}

void borrar(arbol** a, int c)
{
    arbol* pactual;
    int buscando;
    if (*a != NULL)
        if ((*a)->info == c)
            eliminar(a);
        else
        {
            pactual = *a; buscando = True;
            while (buscando)
                if (pactual->info > c)
                    if (pactual->izq == NULL)
                        buscando = False;
                    else
                        if (pactual->izq->info == c)
                        {
                            eliminar(& pactual->izq);
                            buscando = False;
                        }
                    else
                        pactual = pactual->izq;
                else
                    if (pactual->der == NULL )
                        buscando = False;
                    else
                        if (pactual->der->info == c)
                        {
                            eliminar(& pactual->der);
                            buscando = False;
                        }
                        else
                            pactual = pactual->der;
        }
    }
}

void recorrerar(arbol* a)
{
    arbol* pactual;
    pilas* pila;
```



```

inicializarpa(&pila);
pactual = a;
while (! vacio(pactual) || ! vaciapa(pila))
    if (pactual == NULL)
    {
        desapilarpa(&pila, & pactual);
        printf(" %d ", pactual->info);
        pactual = pactual->der;
    }
    else
    {
        apilarpa(&pila, pactual);
        pactual = pactual->izq;
    }
}

void recorrerdes(arbol* a)
{
    arbol* pactual;
    pilas* pila;

    inicializarpa(&pila); pactual = a;
    while (! vacio(pactual) || ! vaciapa(pila))
        if (pactual == NULL)
        {
            desapilarpa(& pila, & pactual);
            printf(" %d ", pactual->info);
            pactual = pactual->izq;
        }
        else
        {
            apilarpa(& pila, pactual);
            pactual = pactual->der;
        }
}

void recorreniveles(arbol* a)
{
    arbol* pactual;
    colas cola;

    if (! vacio(a))
    {
        inicializarca(& cola); pactual = a; ponerca(&cola, a);
        while (! vaciaca(cola))
        {
            primeroca(cola, &pactual); quitarca(&cola);
            printf(" %d ", pactual->info);
            if (! vacio(pactual->izq))
                ponerca(&cola, pactual->izq);
            if (! vacio(pactual->der))
                ponerca(&cola, pactual->der);
        }
    }
}

```

```
    }
}

void preorden(arbol* a)
{
    pilas* pila;
    arbol* pactual;

    inicializarpa(&pila);
    pactual = a;
    while (! vaciapa(pila) || ! vacio(pactual))
        if (vacio(pactual))
            desapilarpa(&pila, &pactual);
        else
        {
            printf(" %d ", pactual->info);
            apilarpa(&pila, pactual->der);
            pactual = pactual->izq ;
        }
}

void postorden(arbol* a)
{
    pilas* pila;
    pilas* vuelta;
    arbol* pactual;

    inicializarpa(& pila);
    inicializarpa(& vuelta);
    pactual = a;
    while (! vaciapa(pila) || ! vacio(pactual))
        if (vacio(pactual))
            desapilarpa(& pila, & pactual);
        else
        {
            apilarpa(& vuelta, pactual);
            apilarpa(& pila, pactual->izq);
            pactual = pactual->der;
        };
    while (! vaciapa(vuelta))
    {
        desapilarpa(&vuelta, &pactual);
        printf(" %d ", pactual->info);
    }
}

/* Programa principal para probar el buen funcionamiento */

int main()
{
    int nm;
    arbol* a;
    char pausa;
```

```

    inicializar (&a);
    printf("Deme numero (0 -> Fin): ");
    scanf("%d", &nm);
    while (nm !=0)
    {
        insertar(&a,nm);
        printf("Deme numero (0 -> Fin): ");
        scanf("%d", &nm);
    }
    printf("Preorden: \n");
    preorden(a);
    puts("");
    printf("Postorden: \n");
    postorden(a);
    puts("");
    printf("Niveles: \n");
    recorreniveles(a);
    puts("");
    pausa = getch();
    return 0;
}

/* Pila de árboles. Implementación */

void inicializarpa(pilas** p)
{
    *p = NULL;
}

void desapilarpa(pilas **p, arbol ** a)
{
    pilas *nn;
    if (vacipa(*p))
        exit (1);
    *a = (*p)->info;
    nn =*p;
    *p = nn->cimaant;
    free(nn);
}

int vacipa(pilas *p)
{
    return p == NULL;
}

void apilarpa(pilas** p, arbol* elemento)
{
    pilas * nn;

    nn = (pilas*)malloc(sizeof(pilas*));
    nn->info = elemento;
    nn->cimaant = *p;

```

```
    *p=nn;
}

void inicializarca(colas* c)
{
    (*c).prim = NULL;
    (*c).ulti = NULL;
}

/* Cola de arboles. Implementación */

void ponerca(colas* c, arbol* e)
{
    nodoc * nn;

    nn = (nodoc*)malloc(sizeof(nodoc*));
    nn->info = e;
    nn->sig = NULL;
    if (vaciacac(*c))
        (*c).prim=nn;
    else
        (*c).ulti->sig = nn;
    (*c).ulti = nn;
}

void quitarca(colas* c)
{
    nodoc* nn;

    if (vaciacac(*c))
        exit (1);
    nn = (*c).prim;
    (*c).prim = (*c).prim->sig;
    if ((*c).prim == NULL)
        (*c).ulti = NULL;
    free(nn);
}

int vaciacac(colas c)
{
    return (c.prim == NULL) && (c.ulti == NULL);
}

void primeroca(colas c, arbol** e)
{
    *e = c.prim->info;
}
```

Ejecución

```

Deme numero (0 -> Fin): 18
Deme numero (0 -> Fin): 8
Deme numero (0 -> Fin): 27
Deme numero (0 -> Fin): 4
Deme numero (0 -> Fin): 12
Deme numero (0 -> Fin): 20
Deme numero (0 -> Fin): 32
Deme numero (0 -> Fin): 1
Deme numero (0 -> Fin): 6
Deme numero (0 -> Fin): 10
Deme numero (0 -> Fin): 15
Deme numero (0 -> Fin): 0
Preorden:
18 8 4 1 6 12 10 15 27 20 32
Postorden:
1 6 4 10 15 12 8 20 32 27 18
Niveles:
18 8 27 4 12 20 32 1 6 10 15

```

PROBLEMAS PROPUESTOS

- 13.1.** Dados dos árboles binarios de búsqueda indicar mediante un programa, si los árboles tienen o no elementos comunes.
- 13.2.** 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 correspondientes funciones para gestionar interactivamente un árbol de números enteros. (Comentar el inconveniente de esta representación de cara al máximo y mínimo número de nodos que pueden almacenarse).
- 13.3.** Una matriz de N elementos almacena cadenas de caracteres. Utilizando un árbol binario de búsqueda como estructura auxiliar ordenar ascendentemente la cadena de caracteres.
- 13.4.** En tres árboles binarios de búsqueda (ORO, PLATA, COBRE) están representados los medallistas de cada una de las pruebas de una reunión atlética. Cada nodo tiene

la información: nombre de la prueba, nombre del participante y nacionalidad. El árbol ORO almacena los atletas ganadores de dicha medalla, y así respectivamente con los árboles PLATA y COBRE. El criterio de ordenación de los árboles ha sido el nombre del atleta. Escribir las funciones necesarias para resolver este supuesto:

Dado el nombre de una atleta y su nacionalidad, del cual no se sabe si tiene medalla, encontrar un equipo de atletas de su mismo país, incluyendo a él mismo que tenga una suma de puntos comprendida entre N y M . Hay que tener en cuenta que una medalla de oro son 10 puntos, plata 5 puntos y cobre 2 puntos.

- 13.5.** 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 un algoritmo que decida si dos árboles binarios A y B son “parecidos”.
- 13.6.** Construir una función recursiva para escribir todos los nodos de un árbol binario de búsqueda cuyo campo clave sea mayor que un valor dado y menor que otro valor dado (el campo clave es de tipo entero).

- 13.7.** Crear un archivo de datos en el que cada línea contenga la siguiente información:

Nombre	30 caracteres
Número de la Seguridad Social	10 caracteres
Dirección	24 caracteres

Escribir un programa que lea cada registro de datos del archivo y lo inserte en un árbol binario de búsqueda según el número de seguridad social, de modo que cuando el árbol se recorra en orden los números de la Seguridad Social se almacenen en orden ascendente. Imprimir una cabecera “DATOS DE EMPLEADOS ORDENADOS ACUERDO AL NÚMERO DE LA SEGURIDAD SOCIAL” y a continuación imprimir los datos del árbol con el formato Columnas:

1-10	Número de la Seguridad Social
20-50	Nombre
55-79	Dirección

- 13.8.** Dado un árbol binario de búsqueda diseñar una función que liste los nodos del árbol que cumplan la condición de ser elegantes. Un nodo (de clave entera) se dice que es elegante si cumple la condición de ser capicúa.
- 13.9.** Una matriz de N elementos almacena cadenas de caracteres. Utilizando un árbol binario de búsqueda como estructura auxiliar ordenar ascendentemente la cadena de caracteres, de tal manera que todas las cadenas que comiencen por un carácter se almacenen en un nodo que tenga una lista enlazada con todas las cadenas ordenadas ascendentemente.
- 13.10.** Se dice que dos árboles binarios de búsqueda son medio iguales, si tienen el mismo número de nodos y coinciden al menos la mitad de las claves de los nodos de los árboles. Escribir una función que decida si dos árboles binarios de búsqueda son medio iguales.

Árboles binarios equilibrados

Un **árbol binario equilibrado** es aquel en el que la altura de los subárboles izquierdo y derecho de cualquier nodo nunca difiere en más de una unidad. Para determinar si un árbol binario 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 binario se determina por la siguiente fórmula: $Fe = h_D - h_I$

Los conceptos de árbol binario equilibrado y árbol binario de búsqueda equilibrado así como los algoritmos de manipulación son el motivo central de este capítulo. Los métodos para su tratamiento fueron descritos en 1962 por los matemáticos G. M. Adelson - Velskii y E. M. Landis.

14.1. Árbol binario equilibrado, árboles AVL

La eficiencia de los árboles binarios de búsqueda para la localización de una clave varía entre $O(n)$ y $O(\log(n))$ dependiendo de su estructura (véase el capítulo 2). Por ejemplo, si se añaden una serie de claves a un árbol binario de búsqueda mediante los algoritmo expuesto en el capítulo anterior, la estructura del árbol dependerá del orden en que éstas hayan sido añadidas. Así, si todos los elementos se insertan en orden creciente o decreciente, el árbol va a tener todas las ramas izquierda o derecha, respectivamente, vacías, y la búsqueda en dicho árbol será totalmente secuencial. Para que las búsquedas resulten eficientes

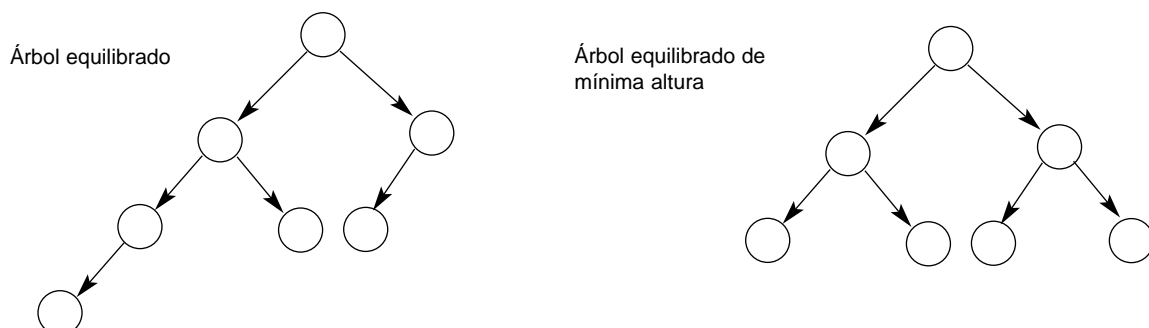


Figura 14.1. Árboles equilibrados.

interesa que los árboles binarios de búsqueda sean equilibrados y mejor aún si además son de mínima altura.

Un árbol binario equilibrado es aquel que la altura de los subárboles izquierdo y derecho de cualquier nodo nunca difiere en más de una unidad, no obstante esta definición permite que árboles bastante asimétricos sean considerados como equilibrados. El árbol binario de búsqueda ideal será un árbol binario de búsqueda equilibrado y de mínima altura, en el que, debido a su estructura, las comparaciones máximas necesarias para localizar cualquier elemento no excedan $\log(n)$, siendo n el número de nodos del árbol.

Los árboles equilibrados con los que habitualmente se trabaja son los denominados árboles AVL. Los AVL, son árboles binarios de búsqueda equilibrados, aunque no necesariamente de mínima altura, que reciben su nombre en honor a Adelson-Velskii y Landis, primeros científicos que estudiaron esta estructura. En un árbol AVL de altura h , los subárboles del nodo raíz tendrán las alturas $h-1$ y $h-2$ o bien $h-1$ y $h-1$, por tanto, si se quiere saber el menor número de nodos que pueden dar origen a un árbol con dicha altura h , habrá que escoger la primera suposición de entre las dos anteriores y esto mismo se repetirá para cada uno de dichos subárboles. Los árboles que siguen esta regla son los AVL menos densos y se llaman árboles de Fibonacci, en ellos el número de nodos para un árbol de altura h viene dado por:

$$N_0 = 0, \quad N_1 = 1, \quad N_h = N_{h-1} + 1 + N_{h-2} \quad \text{para todo } h \geq 2$$

14.2. Inserción en árboles AVL

La inserción de un elemento en un árbol AVL utiliza el algoritmo usual de inserción de un nuevo elemento en un árbol binario modificado con la finalidad de conseguir que en ningún momento la altura de los subárboles izquierdo y derecho de un nodo difiera en más de una unidad. Para poder determinar esto con facilidad, cada uno de los nodos de un AVL suele tener un campo donde almacenar su *factor de equilibrio*. El *factor de equilibrio* de un nodo es la diferencia entre las alturas de sus subárboles derecho e izquierdo y debe oscilar entre -1 , 0 y 1 , pues cualquier otro valor implicaría la necesaria reestructuración del árbol.

El proceso de inserción consistirá en:

- Comparar el elemento a insertar con el nodo raíz, si es mayor avanzar hacia el subárbol derecho, si es menor hacia el izquierdo y repetir la operación de comparación hasta encontrar un elemento igual o llegar al final del subárbol donde debiera estar el nuevo elemento. El camino recorrido desde la raíz hasta el momento en el que se terminan las comparaciones sin encontrar al elemento constituye el camino de búsqueda al que en reiteradas ocasiones se hará referencia.
- Cuando se llega al final es porque no se ha encontrado, entonces se crea un nuevo nodo donde se coloca el elemento y, tras asignarle un cero como factor de equilibrio, se añade como hijo izquierdo o derecho del nodo anterior según corresponda por la comparación de sus campos de clasificación. En este momento también se activará un interruptor sw para indicar que el subárbol ha crecido en altura.
- Regresar por el camino de búsqueda y si sw está activo calcular el nuevo factor de equilibrio del nodo que está siendo visitado. Deben distinguirse los siguientes casos:

1. *Las ramas izquierda y derecha del mencionado nodo tenían anteriormente la misma altura.*

Al insertar un elemento en la rama izquierda su altura se hará mayor que la de la derecha.

Al insertar un elemento en la rama derecha ésta se hará más alta que la izquierda.

2. *La rama derecha era más alta que la izquierda.*

Un nuevo elemento en la rama izquierda consigue que las dos adquieran la misma altura. El subárbol ha dejado de crecer y sw se desactiva.

Un nuevo elemento en la rama derecha rompe el equilibrio del árbol y hace que sea necesaria su reestructuración. La reestructuración anula el crecimiento en altura de la rama en la que se encuentra y, cuando se ejecuta, hay que conmutar el valor de la variable sw para que no se sigan recalculando factores de equilibrio.

3. *La rama izquierda era más alta que la derecha.*

Un nuevo elemento en la rama derecha consigue que las dos adquieran la misma altura. El subárbol ha dejado de crecer y sw se desactiva.

Un nuevo elemento en la rama izquierda rompe el equilibrio del árbol y hace que sea necesaria su reestructuración. La reestructuración anula el crecimiento en altura de la rama en la que se encuentra y, cuando se ejecuta, hay que conmutar el valor de la variable sw para que no se sigan recalculando factores de equilibrio.

14.3. Rotaciones en la inserción

Durante el proceso de inserción, al actualizar los factores de equilibrio, si un nodo deja de cumplir el criterio de equilibrio (factor de equilibrio entre -1 y 1) el árbol deberá ser reestructurado. Reestructurar un árbol significa rotar los nodos del mismo. En la inserción las rotaciones podrán ser de los siguientes tipos: simple (derecha-derecha e izquierda-izquierda) y doble (derecha-izquierda, izquierda-derecha).

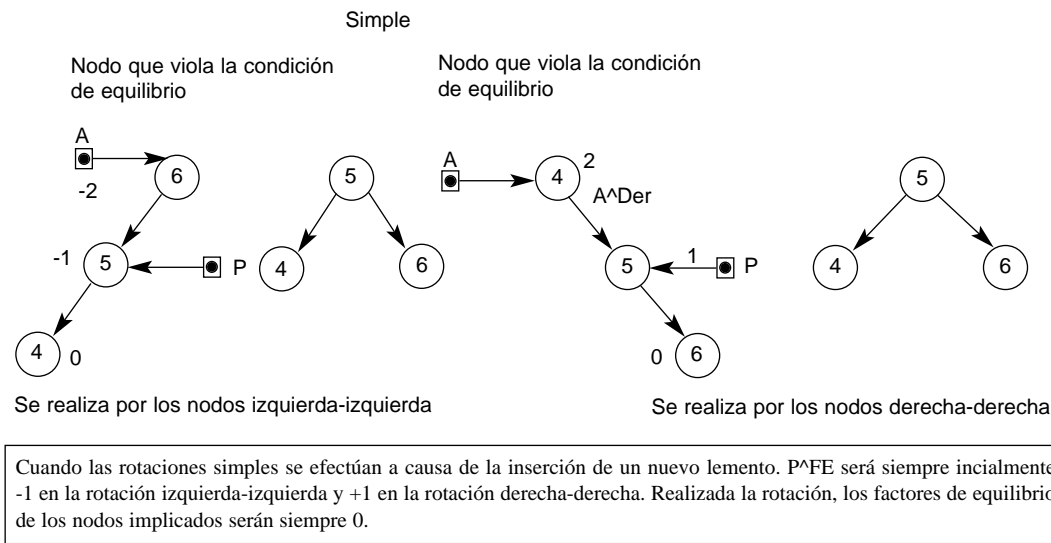


Figura 14.2. Tipos de rotación simple.

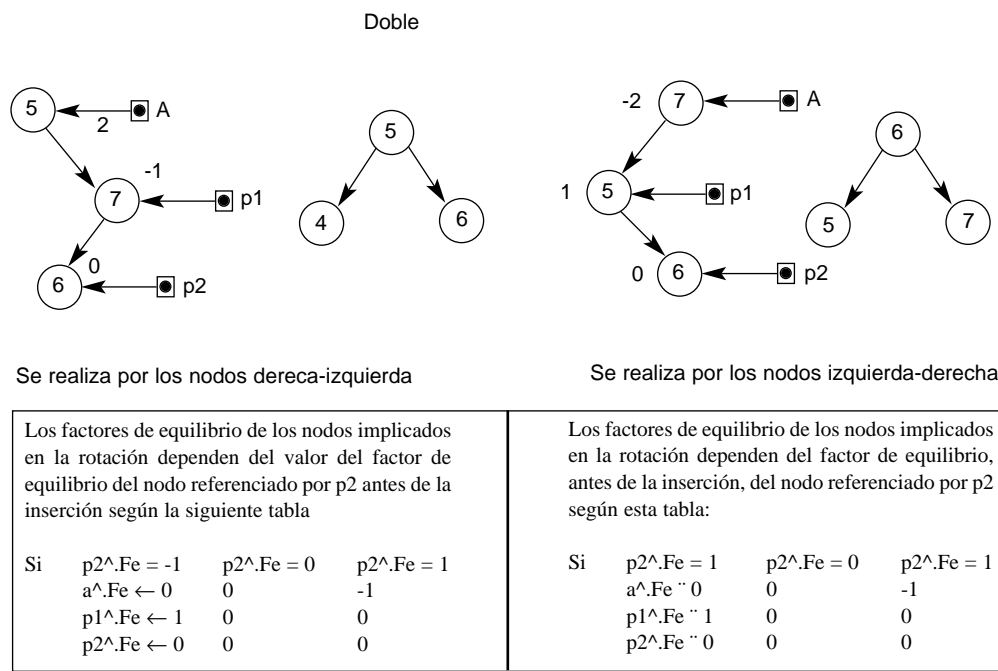


Figura 14.3. Tipos de rotación doble

La rotación simple izquierda-izquierda se produce cuando el nodo actual se ha desequilibrado, pretende adquirir el factor de equilibrio -2, y su descendiente izquierdo o descendiente en la rama por la que se ha efectuado la inserción tiene como factor de equilibrio -1. Por su parte, la rotación simple derecha-derecha se produce cuando el nodo actual se ha desequilibrado y pretende adquirir el factor de equilibrio 2 y su descendiente derecho o descendiente en la rama por la que se ha efectuado la inserción tiene como factor de equilibrio 1.

EJEMPLO 14.1. Rotación izquierda-izquierda.

En el árbol de la Figura 14.4 al insertar un 6 el desequilibrio se produce en el 15 y los elementos anteriormente visitados son el 7 y el 5 (izquierda-izquierda). La situación es que el penúltimo, el 7 ($15 > 7 > 5$), puede actuar como padre de los otros dos. Las figuras 14.4 y 14.5 muestran este tipo de rotación.

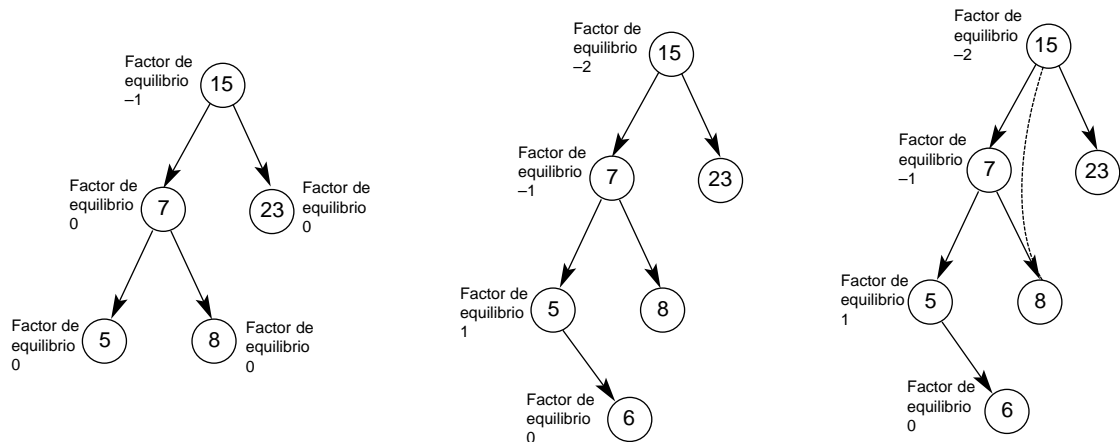


Figura 14.4. Inserción de un elemento y actualización de los factores de equilibrio.

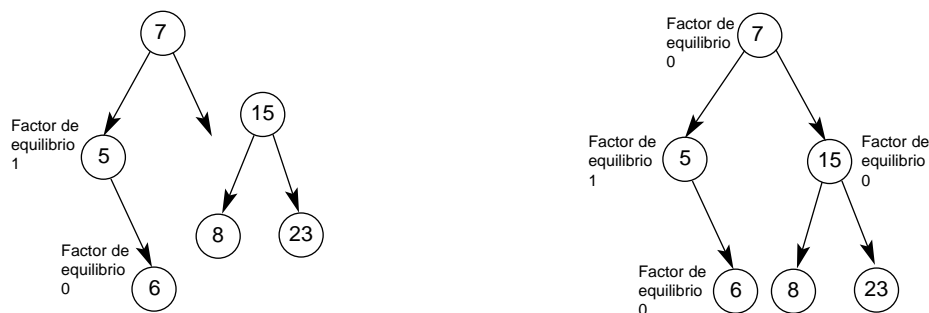


Figura 14.5. Rotación izquierda-izquierda.

Las rotación doble izquierda-derecha se produce cuando el nodo actual se desequilibra y pretende adquirir el factor de equilibrio -2 y su descendiente en la rama por la que se está efectuando la inserción (izquierdo) tiene como factor de equilibrio 1. Como se puede deducir, la rotación doble derecha-izquierda implica que el nodo desequilibrado pretende adquirir el factor de equilibrio 2 y su descendiente en la rama por la que se está efectuando la inserción (derecho) tiene como factor de equilibrio -1.

EJEMPLO 14.2. Rotación derecha-izquierda.

Como se observa en la figura 14.6 al insertar el nodo con clave 6 en el árbol mostrado, se sigue el camino derecha de 4 e izquierda de 7 para terminar insertando por la derecha de 5. Al regresar por el camino de búsqueda los factores de

equilibrio se incrementan en 1 si se fue por la rama derecha y se decrementan en uno cuando se fue por la rama izquierda. En el nodo 4 el equilibrio se rompe, por lo que será necesario efectuar una rotación derecha-izquierda con la finalidad de restablecerlo. El nodo que ahora puede actuar como raíz es el antepenúltimo visitado en el camino de vuelta, es decir el 5 (Figura 14.7)

Nodo que viola la condición de equilibrio

Fe anterior -1

Fe después de la inserción -2

Fe anterior -0

Fe después de la inserción -1

Fe anterior -0

Fe después de la inserción -1

Fe asignado -0

Nodo que se acaba de insertar

Figura 14.6. Inserción de un elemento y actualización de los factores de equilibrio.

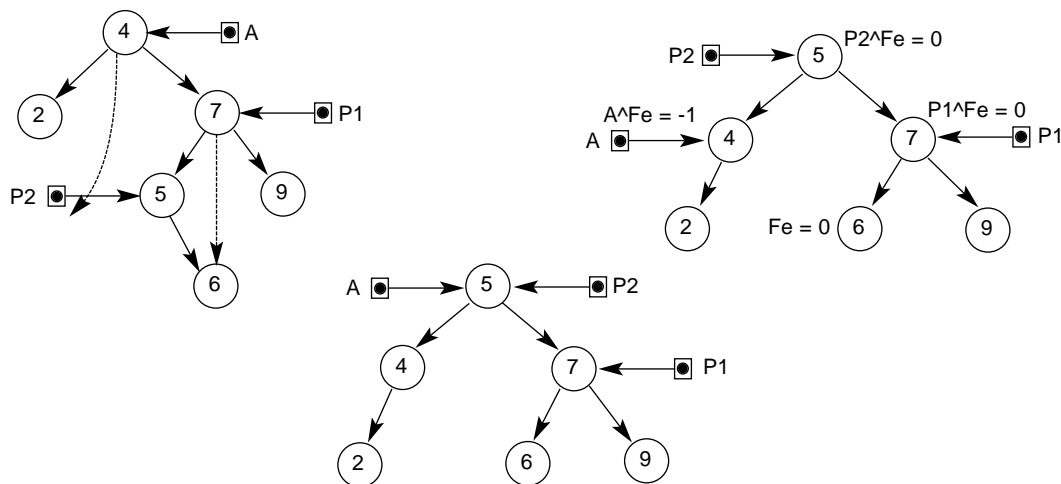


Figura 14.7. Rotación derecha-izquierda.

14.4. La eliminación (o borrado en árboles AVL)

El borrado de un nodo en un árbol AVL consiste en la eliminación del mismo sin que el árbol deje de ser de búsqueda ni equilibrado. En principio el algoritmo de Eliminación en un AVL sigue casi los mismos pasos que el borrado en árboles de búsqueda binarios, la diferencia está en que, a las operaciones habituales, hay que añadir ahora las de cálculo de los factores de equilibrio y reestructuración del árbol (rotaciones de nodos simples, o dobles) cuando el equilibrio ha sido alterado.

En un árbol AVL, la eliminación de un nodo implica la activación de una variable, sw , que, en este caso, indica ha disminuido la altura del subárbol considerado. Por tanto, una vez eliminado un nodo se activa sw y se regresa por el camino de búsqueda calculando, mientras sw esté activo, los nuevos factores de equilibrio (Fe) de los nodos visitados. Hay que tener en cuenta que, cuando se regresa por el camino de búsqueda con sw activo, el *factor de equilibrio* del nodo visitado disminuye en 1 si la eliminación se efectuó por la rama derecha y aumenta en 1 cuando se hizo por la izquierda. Si alguno de los nodos pierde la condición de equilibrio, ésta debe ser restaurada mediante rotaciones.

La variable sw en el borrado representa que decrece la rama que se está considerando y por lo tanto sólo se desactiva cuando se verifica que la eliminación del nodo ha dejado de repercutir en la altura del subárbol. Así como en la inserción una vez efectuada una rotación sw siempre conmutaba y los restantes nodos mantenían su factor de equilibrio, en el borrado las rotaciones no siempre paran el proceso de actualización de los factores de equilibrio. Esto implica que puede producirse más de una rotación en el retroceso realizado por el camino de búsqueda hacia la raíz del árbol.

EJEMPLO 14.3. Desactivación de la variable sw en el borrado.

La Figura 14.8 muestra por qué razón la variable se desactiva, puesto que la eliminación del nodo no ocasiona una disminución en altura del árbol.

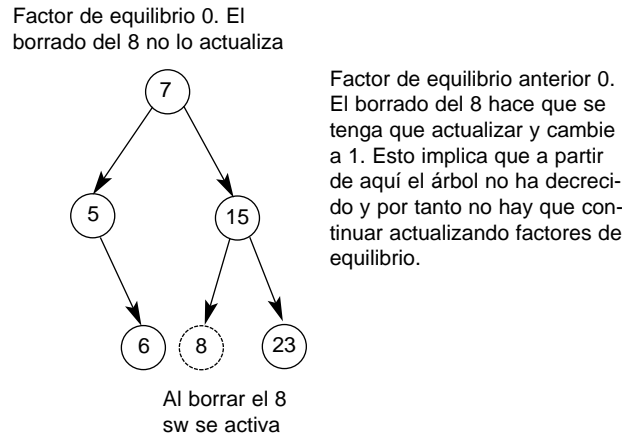


Figura 14.8. Los factores de equilibrio tras el borrado de un nodo.

14.5. Rotaciones en la eliminación

Las rotaciones a efectuar a causa de la eliminación de un nodo son las mismas que en la inserción, aunque con la adición de dos subtipos más a los que se suele denominar: rotación simple derecha-derecha especial y rotación simple izquierda-izquierda especial. La rotación simple izquierda-izquierda especial se aplica cuando el nodo desequilibrado pretende adquirir el factor de equilibrio -2 y su descendiente por la rama contraria a aquella en la que se ha efectuado el borrado tiene 0 como factor de equilibrio y la rotación derecha-derecha especial cuando el nodo desequilibrado pretende adquirir el factor de equilibrio 2 y su descendiente por la rama contraria a aquella en la que se ha efectuado el borrado tiene 0 como factor de equilibrio. En estas clases especiales de rotación derecha-derecha e izquierda-izquierda, tras realizarse la rotación, los factores de equilibrio de los nodos implicados no son 0 lo que implica que la rama ha dejado de decrecer y sw se desactiva.

EJEMPLO 14.4. La figura 14.9 muestra las rotaciones derecha-derecha especial del borrado.

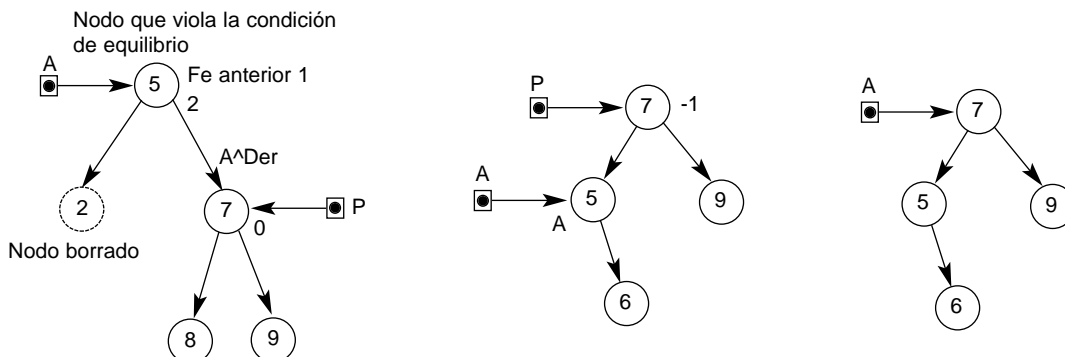
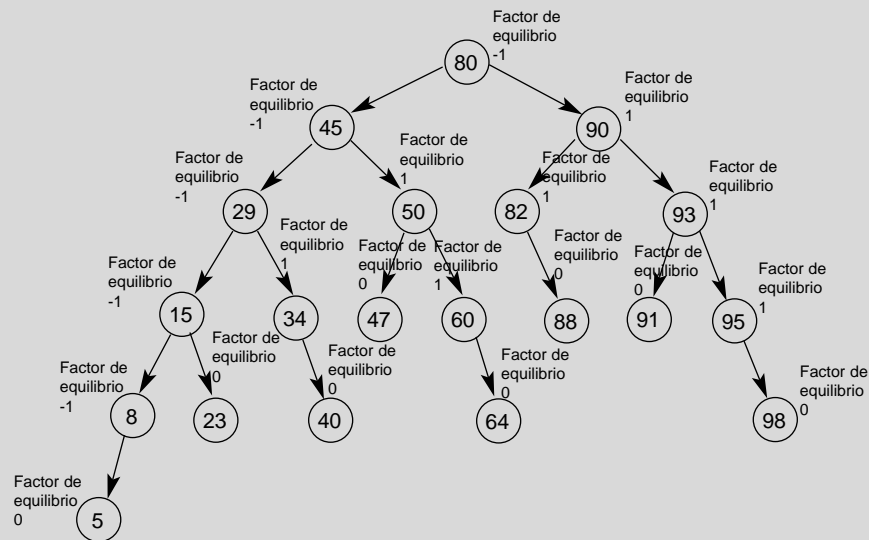


Figura 14.9. La rotación derecha-derecha especial del borrado.

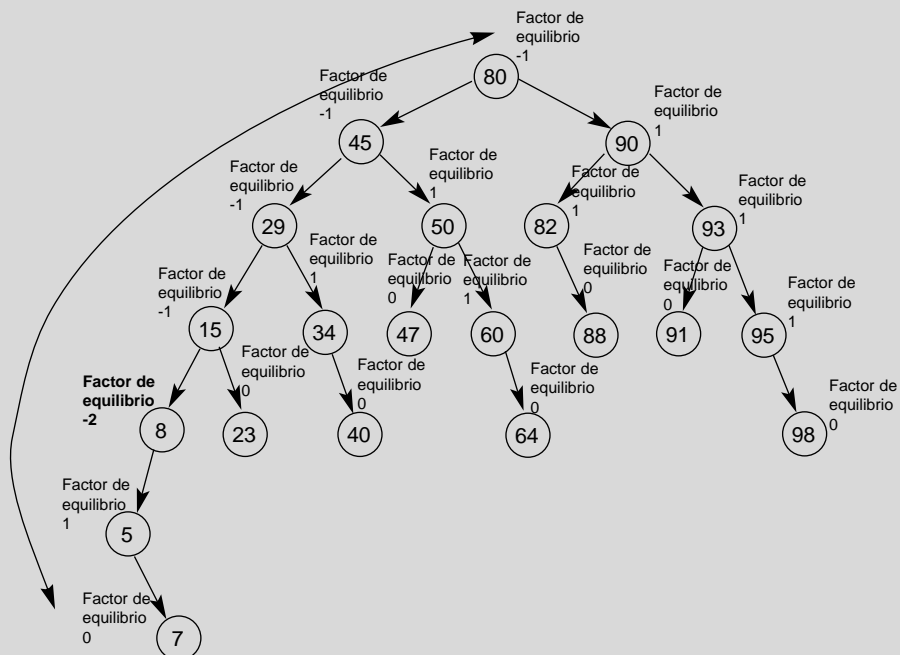
PROBLEMAS DE SEGUIMIENTO

14.1. Describir paso a paso el proceso de inserción de un nodo con clave 7 en el árbol que se muestra a continuación.

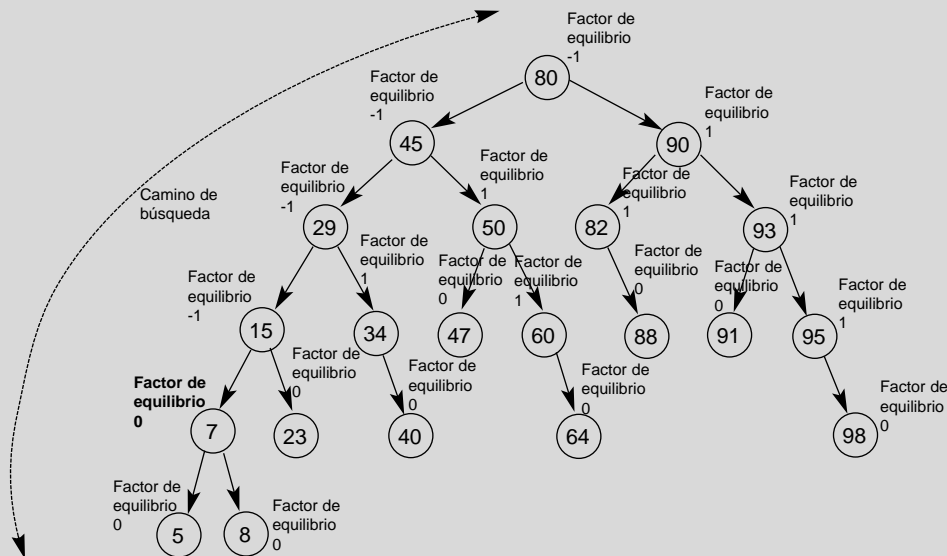


Solución

En primer lugar se baja buscando si el elemento existe o, en caso de que no exista, la posición de inserción. Cuando se descubre que la clave 7 no existe se crea un nuevo nodo y se inserta. Tras insertar el 7, se sube por el camino seguido para localizar la posición de inserción actualizando los factores de equilibrio de los diferentes nodos según van siendo visitados.



Al llegar al nodo con valor 8 se observa que toma como factor de equilibrio -2 y, puesto que no está permitido que un nodo tenga como factor de equilibrio -2, se hace necesaria una reestructuración. La clase de rotación la determinan los factores de equilibrio del 8 y del 5 (elemento anterior en rama donde se ha efectuado la inserción) y será una rotación izquierda-derecha.



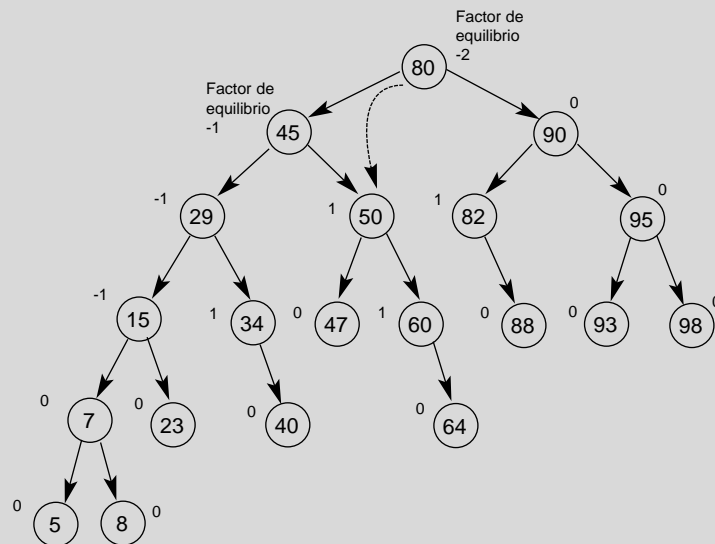
Tras la reestructuración ya no hace falta actualizar los factores de equilibrio de los nodos que ocupan posiciones precedentes en el camino de búsqueda. Es decir, a partir de ese momento, se ha conseguido que la subrama no crezca en altura y por tanto ya no puede originar la modificación del factor de equilibrio de ningún nodo que se encuentre en posición superior ni se van a producir más rotaciones.

- 14.2.** Considerar ahora la adición de un nuevo elemento con clave 25 al árbol anterior e indique los valores para el factor de equilibrio en cada nodo al terminar el proceso.

Solución

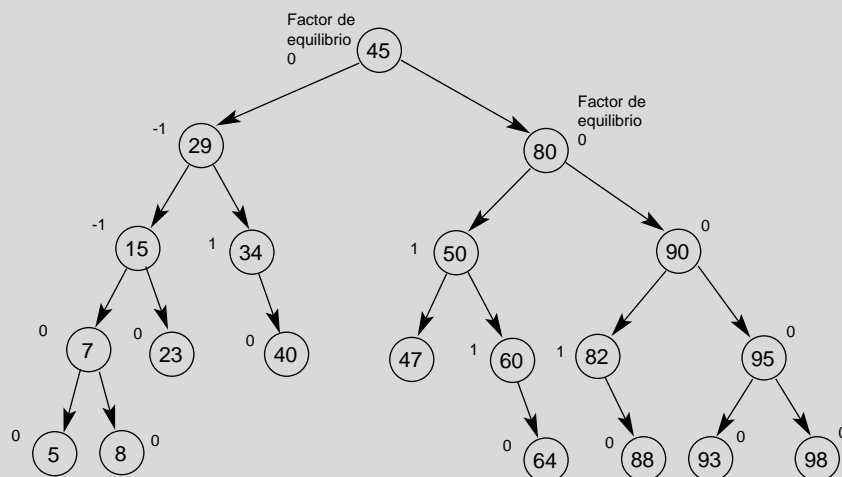
Inicialmente el factor de equilibrio para el nuevo nodo será 0 y sw se activa. Al retroceder por el camino de búsqueda el factor de equilibrio del 23 se convertirá en 1 y sw seguirá estando activo. Sin embargo, al llegar al 15 su factor de equilibrio se convierte en 0, esto indica que la subrama izquierda del 29, cuya raíz es el 15, no ha cambiado su altura y por lo tanto la adición del 25 ya no puede afectar el factor de equilibrio de los restantes nodos e implica la desactivación de sw . Los factores de equilibrio de los restantes nodos no se modifican.

14.3. Describir el proceso de eliminación del nodo con clave 91 del siguiente árbol.

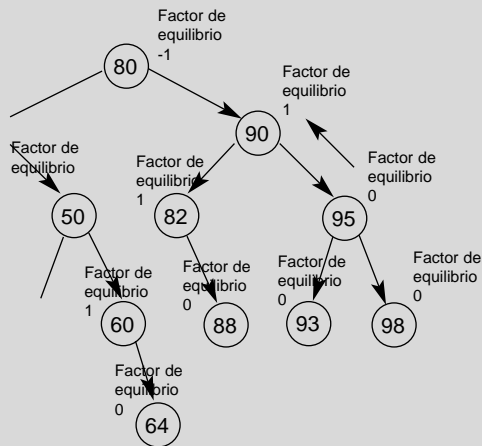


Solución

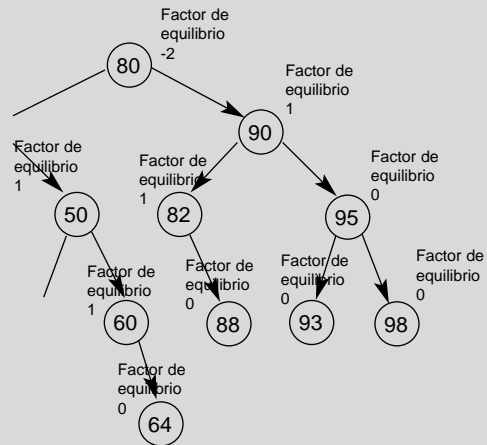
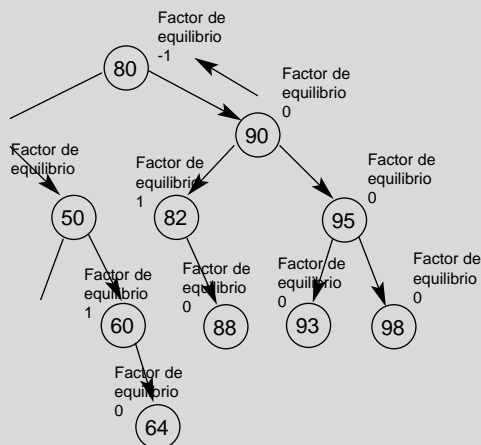
En primer lugar se busca el elemento. Una vez localizado, como es una hoja, se borra, se activa `sw` y se retrocede por el camino de búsqueda actualizando los factores de equilibrio. Como el borrado se ha efectuado por la izquierda al factor de equilibrio del 93 habría que sumarle una unidad y pasaría a ser 2, situación no permitida que requiere una rotación. La clase de rotación la definen los factores de equilibrio del 93 y 95 (su hijo por la rama contraria a aquella en la que se está efectuando el borrado) y resulta ser una rotación simple derecha-derecha.



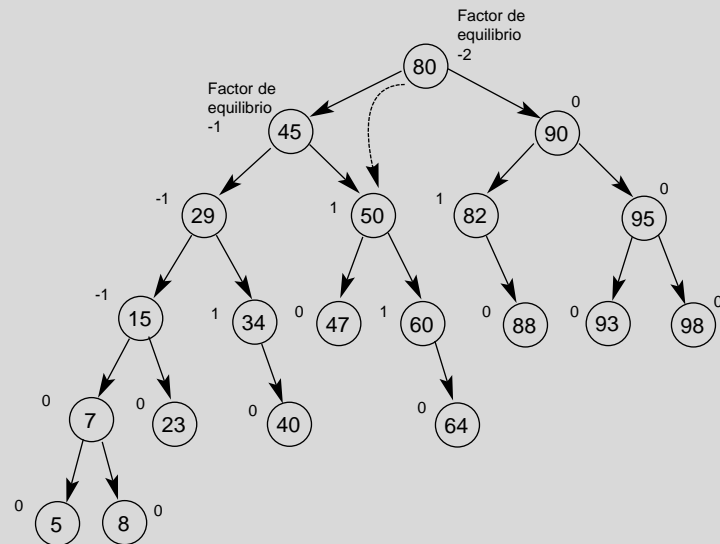
La rotación derecha-derecha pone los factores de equilibrio del 93 y 95 a cero. La variable `sw` no conmuta, pues como se puede observar el subárbol sigue decreciendo.



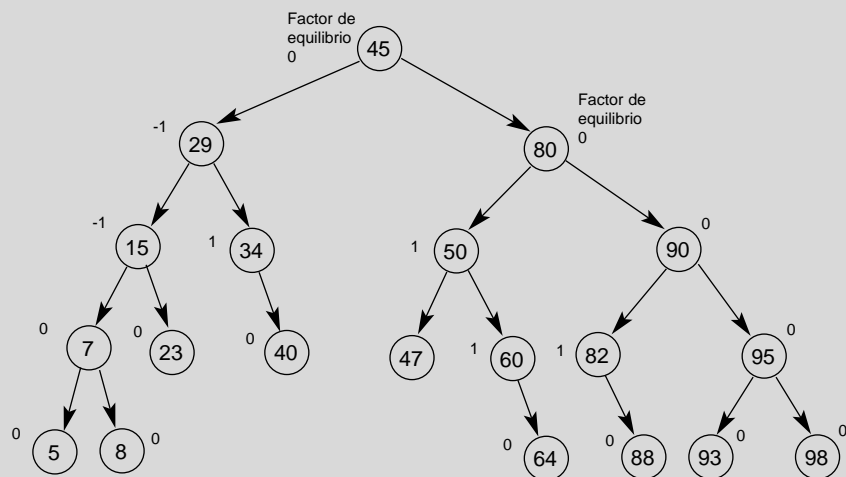
El proceso de actualización de los factores de equilibrio continúa. Como el borrado se ha efectuado en la rama derecha del 90 a su factor de equilibrio se le resta una unidad. La variable `sw` sigue sin conmutar y se pasa a actualizar el factor de equilibrio del 80. Para el 80 el borrado se ha efectuado por su rama derecha y por tanto a su factor de equilibrio también hay que restarle una unidad.



Se produce nuevamente un factor de equilibrio no permitido que obliga a la reestructuración. La clase de rotación la determinan los factores de equilibrio del 80 (nodo desequilibrado) y su hijo por la rama contraria a aquella en la que se está efectuando el borrado, es decir el 45 y resulta ser una rotación izquierda-izquierda. La línea de puntos en la figura marca donde tiene que situarse el nodo 80.



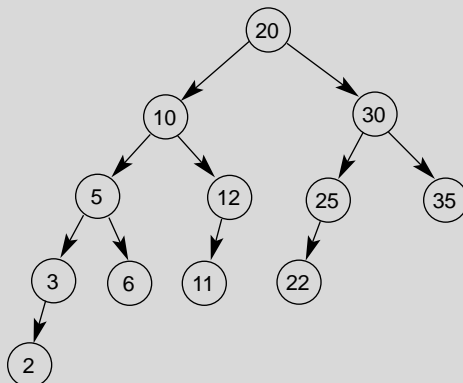
Esta segunda rotación es ya la última, puesto que se ha llegado a nivel de la raíz y el árbol resultante del borrado del 91 es pues el siguiente:



- 14.4.** Dada la secuencia de claves enteras: 20, 10, 30, 5, 25, 12, 3, 35, 22, 11, 6, 2. Representar gráficamente el árbol AVL correspondiente e indicar en qué momento se efectuó una rotación.

Solución

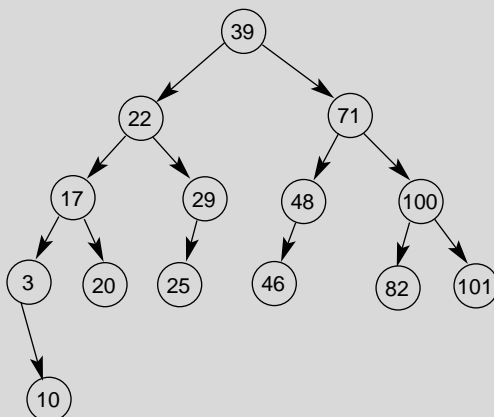
No se efectuó rotación en ningún momento.



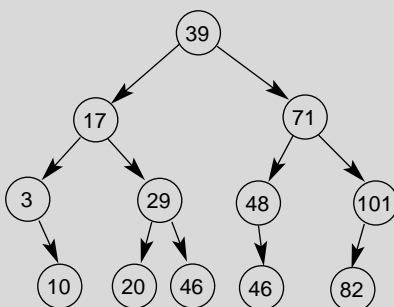
- 14.5.** Dada la secuencia de claves enteras: 100, 29, 71, 82, 48, 39, 101, 22, 46, 17, 3, 20, 25, 10. Representar gráficamente el árbol AVL correspondiente. Eliminar claves consecutivamente hasta encontrar un desequilibrio y dibujar la estructura del árbol tras efectuarse la oportuna restauración.

Solución

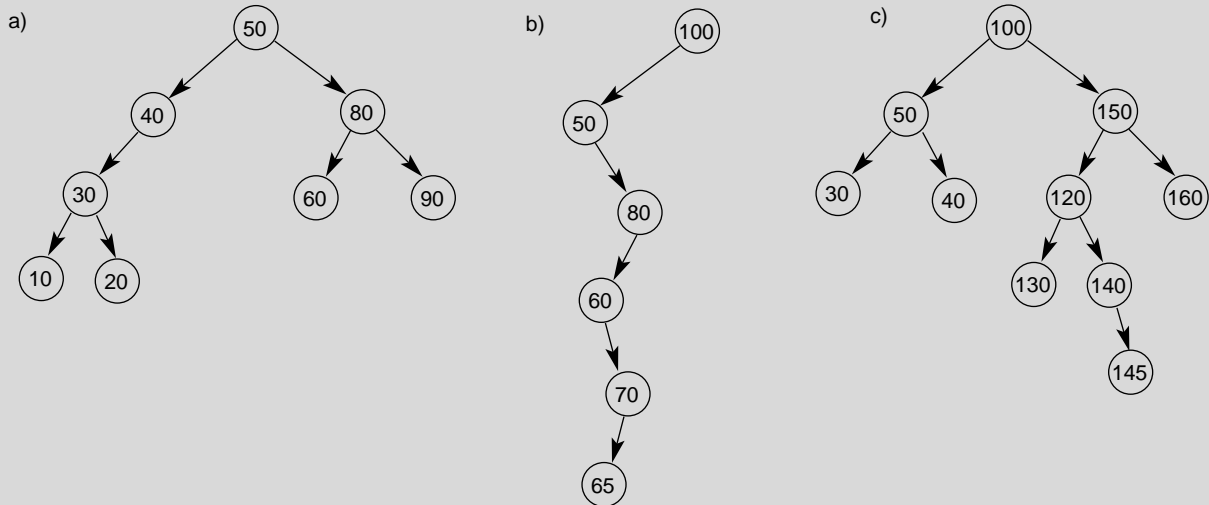
El árbol AVL que resulta es:



Se elimina el 100 y a continuación el 29. Al suprimir el 29 es necesario efectuar una restauración del tipo rotación simple y el árbol queda con la siguiente estructura.



14.6. Determinar cuáles de los siguientes árboles de búsqueda binaria son AVL. En el caso de que no lo sean encuentre todos los nodos que violen los requerimientos de AVL.



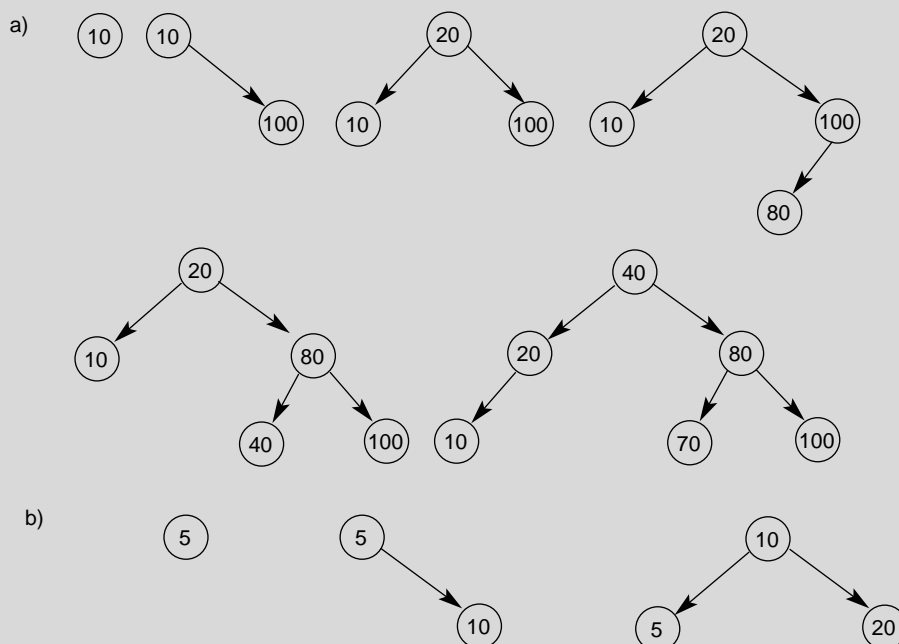
Solución

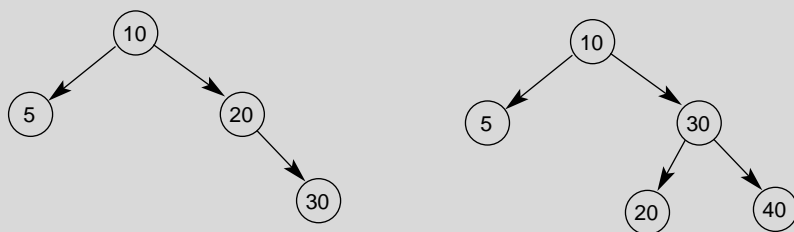
Se marcaron con * los que hacen que no sean AVL.

14.7. Insertar las claves en el orden indicado a fin de incorporarlas a un árbol AVL.

- a) 10, 100, 20, 80, 40, 70
 b) 5, 10, 20, 30, 40, 50, 60
 c) 50, 100, 40, 5, 110, 20, 60, 65
 d) 10, 100, 20, 90, 30, 80, 40, 70, 50, 60

Solución (apartado c y d se encuentra en la página Web del libro)

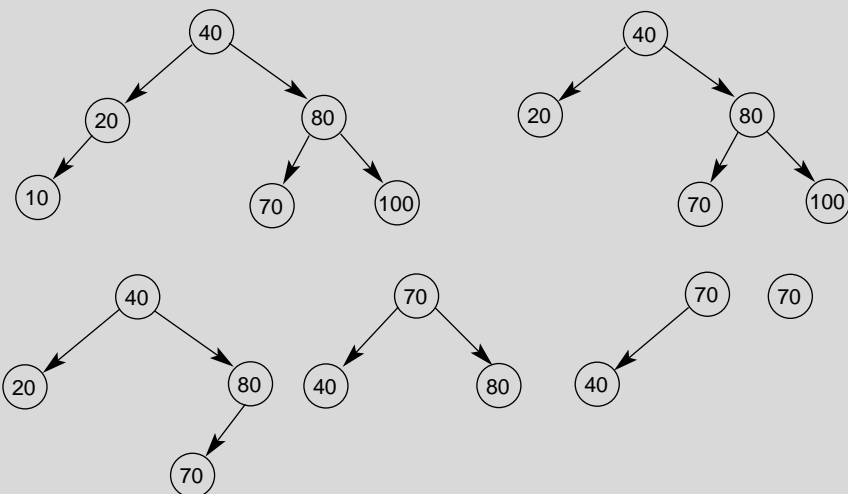




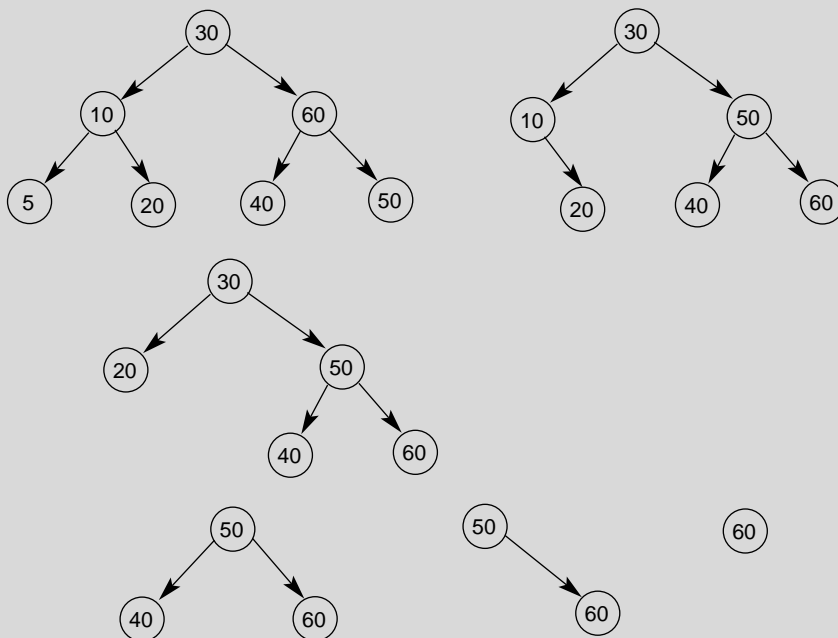
14.8. Eliminar las claves de los árboles construidos en el ejercicio 14.7 en el orden primero en entrar primero en salir.

Solución (apartado c y d se encuentra en la página Web del libro)

a)



b)



PROBLEMAS BÁSICOS

14.9. *Escribir un pseudocódigo que describa el proceso de inserción de un elemento en un árbol AVL.*

Solución

```

procedimiento insertar(E/S arbol: a; E <tipo_info_nodo>: e; E/S logico: sw)
  inicio
    si a = nulo entonces
      a ← construir(nulo, e, nulo)
      a↑.Fe ← 0
      sw ← verdad          /* inicialmente se considera que el subárbol ha crecido */
    si_no
      si a↑.info > e entonces
        insertar(a↑.izq, e, sw)
        si sw entonces
          actualizarizq(a, sw)
        fin_si
      si_no
        si a↑.info < e entonces
          insertar(a↑.der, e, sw)
          si sw entonces
            actualizarder(a, sw)
          fin_si
        si_no
          sw ← falso
        fin_si
      fin_si
    fin_si
  fin_procedimiento

arbol: funcion construir(E arbol: a; E <tipo_info_nodo>: e; E arbol: b)
  var
    arbol: nuevo
  inicio
    reservar(nuevo)
    nuevo↑.info ← e
    nuevo↑.izq ← a
    nuevo↑.der ← b
    devolver(nuevo)
  fin_funcion

procedimiento actualizarizq(E/S arbol: a; E/S logico: sw)
  inicio
    según_sea a↑.Fe hacer
      1: a↑.Fe ← 0
        sw ← falso
      0: a↑.Fe ← -1
      -1: si a↑.izq↑.Fe = -1 entonces
          rotacioniisimple(a)
        si_no
          rotacioniddoble(a)

```

```

        fin_si
        sw ← falso
    fin_según
fin_procedimiento

procedimiento rotacioniisimple(E/S arbol: a)
var
    arbol: p
inicio
    p ← a↑.izq
    a↑.izq ← p↑.der
    p↑.der ← a
    p↑.Fe ← 0
    a↑.Fe ← 0
    a ← p
fin_procedimiento

procedure rotacioniddoble(E/S arbol: a)
var
    arbol: p1, p2
inicio
    p1 ← a↑.izq
    p2 ← p1↑.der
    p1↑.der ← p2↑.izq
    p2↑.izq ← p1
    a↑.izq ← p2↑.der
    p2↑.der ← a
    si p2↑.Fe = 1 entonces
        p1↑.Fe ← - 1
    si_no
        p1↑.Fe ← 0
    fin_si
    si p2↑.Fe = -1 entonces
        a↑.Fe ← 1
    si_no
        a↑.Fe ← 0
    fin_si
    p2↑.Fe ← 0
    a ← p2
fin_procedimiento

procedimiento actualizar(E/S arbol: a; E/S logico: sw)
inicio
    según_sea a↑.Fe hacer
        -1: a↑.Fe ← 0
            sw ← falso
        0: a↑.Fe ← 1
        1: si a↑.der↑.Fe = 1 entonces
            rotacionddsimple(a)
        si_no
            rotaciondidoble(a)
    fin_si
    sw ← falso

```

```

    fin_según
fin_procedimiento

procedimiento rotacionddsimple(E/S árbol: a)
var
    árbol: p
inicio
    p ← a↑.der
    a↑.der ← p↑.izq
    p↑.izq ← a
    p↑.Fe ← 0
    a↑.Fe ← 0
    a ← p
fin_procedimiento

procedimiento rotaciondidoble(E/S árbol: a)
var
    árbol: p1, p2
inicio
    p1 ← a↑.der
    p2 ← p1↑.izq
    p1↑.izq ← p2↑.der
    p2↑.der ← p1
    a↑.der ← p2↑.izq
    p2↑.izq ← a
    si p2↑.Fe = 1 entonces
        a↑.Fe ← -1
    si_no
        a↑.Fe ← 0
    fin_si
    si p2↑.Fe = -1 entonces
        p1↑.Fe ← 1
    si_no
        p1↑.Fe ← 0
    fin_si
    p2↑.Fe ← 0
    a ← p2
fin_procedimiento

```

14.10. Escribir un pseudocódigo que describa el proceso de borrado de un elemento en un árbol AVL.

Solución

```

procedimiento borrar(E/S árbol: a; E <tipo_info_nodo>: c; E/S logico: sw)
inicio
    si a = nulo entonces
        sw ← falso
    si_no
        si a↑.info = c entonces
            eliminar(a, sw)
        si_no
            si a↑.info > c entonces
                borrar(a↑.izq, c, sw)

```



```

        si sw entonces
            actualizarbi(a, sw)
        fin_si
    si_no
        borrar(a↑.der, c, sw)
        si sw entonces
            actualizarbd(a, sw)
        fin_si
    fin_si
fin_si
fin_si
fin_procedimiento

```

procedimiento eliminar(E/S arbol: a; E/S logico: sw)

```

var
    arbol: a
    <tipo_info_nodo>: e
inicio
    si a↑.izq = nulo entonces
        auxi ← a
        a ← a↑.der
        liberar(auxi)
        sw ← verdad /* inicialmente se considera que el árbol ha decrecido en altura */
    si_no
        si a↑.der = nulo entonces
            auxi ← a
            a ← a↑.izq
            liberar(auxi)
            sw ← verdad
        si_no
            menor(a↑.der, e)
            a↑.info ← e
            borrar(a↑.der, e, sw)
            si sw entonces
                actualizarbd(a, sw)
            fin_si
        fin_si
    fin_si
fin_procedimiento

```

procedimiento menor(E arbol: a; E/S <tipo_info_nodo>: e)

```

inicio
    si a↑.izq = nulo entonces
        e ← a↑.info
    si_no
        menor(a↑.izq, e)
    fin_si
fin_procedimiento

```

procedimiento actualizarbi(E/S arbol: a; E/S logico: sw)

```

inicio
    según_sea a↑.Fe hacer
        -1: a↑.Fe ← 0

```

```

0: a↑.Fe ← 1
   sw ← falso
1: según_sea a↑.der↑.Fe hacer
   1: rotacionddsimple(a)
   -1: rotaciondidoble(a)
   0: rotaciondd2(a)
   sw ← falso
   fin_según
fin_según
fin_procedimiento

procedimiento actualizarbd(E/S arbol: a; E/S logico: sw)
inicio
  según_sea a↑.Fe hacer
    1: a↑.Fe ← 0
    0: a↑.Fe ← -1
    sw ← falso
    -1: según_sea a↑.izq↑.Fe hacer
        -1: rotacioniisimple(a)
        1: rotacioniddoble(a)
        0: rotacionii2(a)
        sw ← falso
    fin_según
  fin_según
fin_procedimiento

procedimiento rotaciondd2(E/S arbol: a)
var
  arbol: p
inicio
  p ← a↑.der
  a↑.der ← p↑.izq
  p↑.izq ← a
  p↑.Fe ← -1
  a↑.Fe ← 1
  a ← p
fin_procedimiento

procedimiento rotacionii2(E/S arbol: a)
var
  arbol: p
inicio
  p ← a↑.izq
  a↑.izq ← p↑.der
  p↑.der ← a
  p↑.Fe ← 1
  a↑.Fe ← -1
  a ← p
fin_procedimiento

```

```

procedimiento rotaciondd2(E/S arbol: a)
var
  arbol: p
inicio
  p ← a↑.der
  a↑.der ← p↑.izq
  p↑.izq ← a
  p↑.Fe ← -1
  a↑.Fe ← 1
  a ← p
fin_procedimiento

```

```

procedimiento rotacionii2(E/S arbol: a)
var
  arbol: p
inicio
  p ← a↑.izq
  a↑.izq ← p↑.der
  p↑.der ← a
  p↑.Fe ← 1
  a↑.Fe ← -1
  a ← p
fin_procedimiento

```

14.11. *Escribir un programa que lea de un fichero de texto el número de nodos que tiene un árbol, a continuación, lea también del mismo archivo las claves del árbol, de tipo entero y ordenadas ascendentemente, y construya un árbol binario de búsqueda, equilibrado y de mínima altura.*

Análisis

Si se conoce el total de claves que va a tener el árbol basta con poner en el subárbol izquierdo la mitad de las claves, en la raíz una y en subárbol derecho el resto. Por tanto basta con construirlo en el orden izquierda-raíz-derecha recursivamente.

Codificación

// árbol binario de búsqueda, equilibrado y de mínima altura

```

#include <stdio.h>
#include <stdlib.h>

typedef struct Arbol
{
    int e;
    struct Arbol* izq;
    struct Arbol* der;
}arbol;

int vacio(arbol* a);
void recorrerar(arbol * a, int n);
void arbolr(int n, arbol** a, FILE* f);
int main()
{
    int n;
    char pausa;
    arbol* a;
    FILE * f;

    if ((f = fopen ("arminimo.dat", "rt")) == NULL)
    {
        puts ("Error de apertura para lectura ");
        exit(1);
    }
    //el primer entero es el número de nodos
    fscanf (f, "%d", &n);
    arbolr(n, &a, f);

```

```
    fclose(f);
    printf("Valores      Nivel\n");
    recorrer(a, 1);
    pausa = getchar();
    return 0;
}

void arbolr(int n, arbol** a, FILE* f)
{
    int nizq;
    int nder;

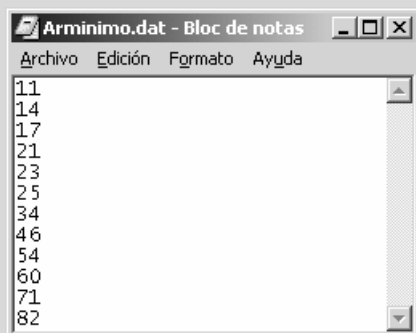
    if (n == 0)
        *a = NULL;
    // no existen datos el árbol es vacío
    else
    {
        nizq = n / 2; // nodos que van a la izquierda
        nder = n - nizq - 1; // nodos que van a la derecha
        *a = (arbol*)malloc(sizeof(arbol));
        arbolr(nizq, &(*a)->izq, f);
        fscanf(f, "%d ", &(*a)->e);
        arbolr(nder, &(*a)->der, f);
    }
}

int vacio(arbol* a)
{
    return a == NULL;
}

void recorrer(arbol *a, int n)
{
    if (!vacio(a))
    {
        recorrer(a->izq, n+1);
        printf(" %d          %d\n", a->e, n);
        recorrer(a->der, n+1);
    }
}
```

Ejecución

Supuesto el siguiente archivo de datos



El resultado de la ejecución es:

Valores	Nivel
14	4
17	3
21	2
23	4
25	3
34	1
46	4
54	3
60	2
71	4
82	3

PROBLEMAS AVANZADOS

14.12. *Crear un programa con los procedimientos básicos necesarios para la manipulación de un árbol AVL, a saber: creación del árbol, inserción y borrado de nodos, así como para efectuar listados ordenados ascendente o descendente por pantalla de la información almacenada en el mismo. El programa debe mostrar un dibujo del árbol cada vez que se realice la inserción o el borrado de un elemento.*

Análisis

Se corresponde con la teoría anteriormente expuesta. Hay que tener en cuenta que el parámetro formal `sw` tiene distinto significado según el tipo de operación que se esté realizando. En las inserciones se activa cuando el subárbol aumenta en altura, mientras que en las operaciones de borrado su activación se corresponde con una disminución en la altura del subárbol.

También hay que destacar que en el borrado de nodos se utilizarán cuatro procedimientos para implementar rotaciones simples; es decir, además de los usados en la inserción, otros dos más a los que se denominará `rotacionii2` (cuando $a^{\wedge}.Fe = -1$ y $p^{\wedge}.Fe = 0$) y `rotaciondd2` (cuando $a^{\wedge}.Fe = 1$ y $p^{\wedge}.Fe = 0$). Estos nuevos procedimientos representan una ligera modificación de `rotacioniisimple` y `rotacionddsimple` debido a que, ahora, los factores de equilibrio de los nodos implicados no terminan siendo 0.

Codificación (La implementación de las funciones se encuentran en la página Web)

```
#include <stdio.h>
#include <stdlib.h>
#define True 1
#define False 0

typedef struct Arbol
{
    int info;
    struct Arbol* izq;
    struct Arbol* der;
    int Fe;
}arbol;

void inicializar(arbol ** a);
```

```

int vacio(arbol* a);
void insertar(arbol ** a, int e, int* sw);
void borrar(arbol ** a, int c, int* sw);
void buscar(arbol * a, int c, int* encontrado);
void recorrerar(arbol * a);
void recorrerdes(arbol * a);
arbol* construir(arbol * a, int e, arbol* b);
void menor(arbol* a, int * e);
void eliminar( arbol ** a, int * sw);
void rotacioniisimple(arbol** a);
void rotacionddsimple(arbol** a);
void rotacioniddoble(arbol** a);
void rotaciondidoble(arbol** a);
void actualizarizq(arbol** a, int* sw);
void actualizarder(arbol** a, int* sw);
void rotacionii2(arbol** a);
void rotaciondd2(arbol** a);
void actualizarbd(arbol** a, int* sw);
void actualizarbi(arbol** a, int * sw);
void dibujar(arbol* a, int h);

//Programa principal

int main()
{
    int nm;
    char pausa;
    arbol* a;
    int sw;

    inicializar (&a);
    printf("Deme numero (0 -> Fin): ");
    scanf("%d%c ", &nm);
    while (nm != 0)
    {
        insertar(&a,nm,&sw);
        dibujar (a, 0);
        puts("");
        printf("Deme numero (0 -> Fin): ");
        scanf("%d%c ", &nm);
    }
    printf("Deme numero a borrar(0 -> Fin): ");
    scanf("%d%c ", &nm);
    while (nm != 0)
    {
        borrar(&a,nm,&sw);
        dibujar (a, 0);
        puts("");
        printf("Deme numero a borrar(0 -> Fin): ");
        scanf("%d%c ", &nm);
    }
    pausa = getchar();
    return 0;
}

```

Ejecución

La ejecución muestra las rotaciones que se producen cuando se inserta en el árbol la siguiente serie de números:

1, 2, 3, 4, 5, 6 y 7

Después pide los números a borrar y muestra las rotaciones originadas en cada caso:

ROTACIONES

```

Deme numero <0 -> Fin>: 1
1

Deme numero <0 -> Fin>: 2
1

Deme numero <0 -> Fin>: 3
1
2
1

Deme numero <0 -> Fin>: 4
4
3
2
1

Deme numero <0 -> Fin>: 5
5
4
3
2
1

Deme numero <0 -> Fin>: 6
6
5
4
3
2
1

Deme numero <0 -> Fin>: 7
7
6
5
4
3
2
1

```

PROBLEMAS PROPUESTOS

- 14.1.** Realizar estudios prácticos y empíricos para estimar el número promedio de rotaciones que se necesitan para insertar un elemento en un árbol AVL, así como para suprimirlo.
- 14.2.** Realizar una comparación práctica de los rendimientos de inserción y borrado en árboles binarios de búsqueda comparados con los binarios de búsqueda AVL.
- 14.3.** Escribir un procedimiento no recursivo que inserte un nodo en un árbol AVL.
- 14.4.** Una empresa de servicios tiene tres departamentos, comercial(1), explotación(2) y comunicaciones(3). Cada empleado está adscrito a uno de ellos. Se ha realizado una redistribución del personal entre ambos departamentos. El archivo EMPRESA contiene en cada registro los campos Número-Idt, Origen, Destino. El campo Origen toma los valores 1, 2, 3 dependiendo del departamento inicial al que pertenece el empleado. El campo Destino toma los mismos valores, dependiendo del nuevo departamento asignado al empleado. El archivo no está ordenado.
- Escribir un programa que almacene los registros del archivo EMPRESA en tres árboles AVL uno por cada departamento origen y realice el intercambio de registros en los árboles.
- 14.5.** Un archivo F contiene los nombres que formaban un árbol binario de búsqueda perfectamente equilibrado A, y que fueron grabados en F en el transcurso de un recorrido en anchura de A. Escribir un programa que realice las siguientes tareas:
- Leer el archivo F para reconstruir el árbol A.
 - Buscar un nombre determinado en A. En caso de encuentro mostrar la secuencia de nombres contenidos entre la raíz de A y el nodo donde figura el nombre buscado.
- 14.6.** Encontrar una expresión para indicar el número máximo y mínimo de nodos de un árbol AVL de altura h . ¿Cuál es el número máximo y mínimo de nodos que puede almacenar un árbol AVL de altura 10?
- 14.7.** En un archivo se han 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 vector de n elementos. Cada elemento tiene el nombre del pueblo y la raíz de un árbol AVL en el que se va a guardar 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.
- 14.8.** Dibujar un árbol AVL de altura n ($n=2, 3, 4, 5, 6, 7, 8$) con el criterio del *peor de los casos*, es decir, en el cada nodo tenga como factor de equilibrio ± 1 .
- 14.9.** En los árboles equilibrados del Ejercicio 14.8 eliminar una de las hojas menos profundas. Representar las operaciones necesarias para restablecer el equilibrio.
- 14.10.** Un árbol binario de búsqueda se dice que es *casi AVL* si cumple la siguiente condición: la diferencia de alturas de los hijos de cada nodo es como máximo de 2. Escriba una función que reciba un árbol binario de búsqueda como parámetro y decida si el árbol es casi AVL.

Árboles B

En los árboles de binarios de búsqueda equilibrados la localización de una clave, en el mejor de los casos, tiene una complejidad de $O(\log n)$, que cuando todas las claves están en memoria principal es considerada eficiente. No obstante, cuando se tiene un conjunto de datos masivo, por ejemplo “1.000.000 de clientes de un banco”, los registros no pueden estar en memoria principal, se ubicarán 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*, estos 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 número de iteraciones y, por tanto, los accesos a disco de la operación de búsqueda, dependen 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 que se estudian en este capítulo tienen muy variadas aplicaciones. Por ejemplo, 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 en la gestión del sistema de archivos de determinados 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.

15.1. Árboles B

Los árboles B (Figura 15.2) son árboles de orden M , $M > 2$, equilibrados, de búsqueda y mínima altura, propuestos por Bayer y McCreight que han de cumplir las siguientes características:

- El nodo raíz tiene entre 2 y M ramas descendientes.
- Todos los nodos (excepto la raíz) tienen entre $\lceil M/2 \rceil$ y M ramas descendientes.
- Todos los nodos (excepto la raíz) tienen entre $\lceil M/2 \rceil$ y M claves.
- El número de claves en cada nodo es siempre una unidad menor que el número de sus ramas.
- Todas las ramas que parten de un determinado nodo tienen exactamente la misma altura.
- En los nodos las claves se encuentran clasificadas y además, a su vez, clasifican las claves almacenadas en los nodos descendientes. Es costumbre denominar a los nodos de un árbol B, *páginas*. La estructura de una página de un árbol B de orden 5 puede representarse como muestra la Figura 15.1.

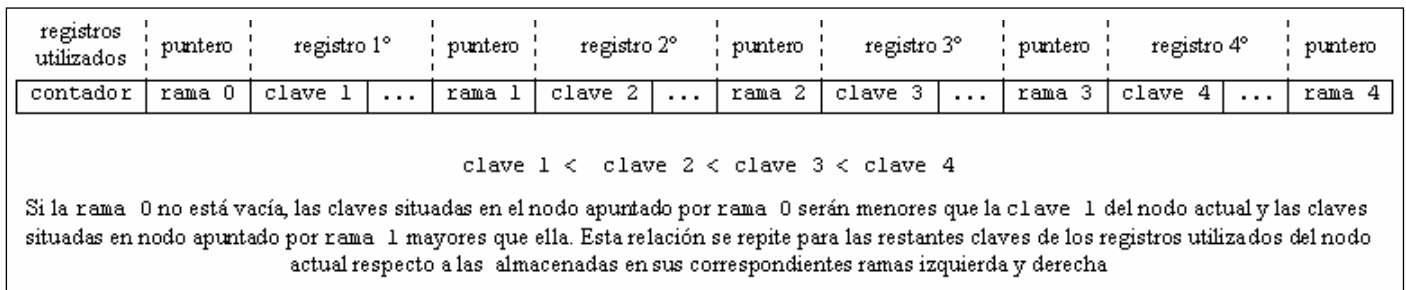


Figura 15.1. Estructura de una página.

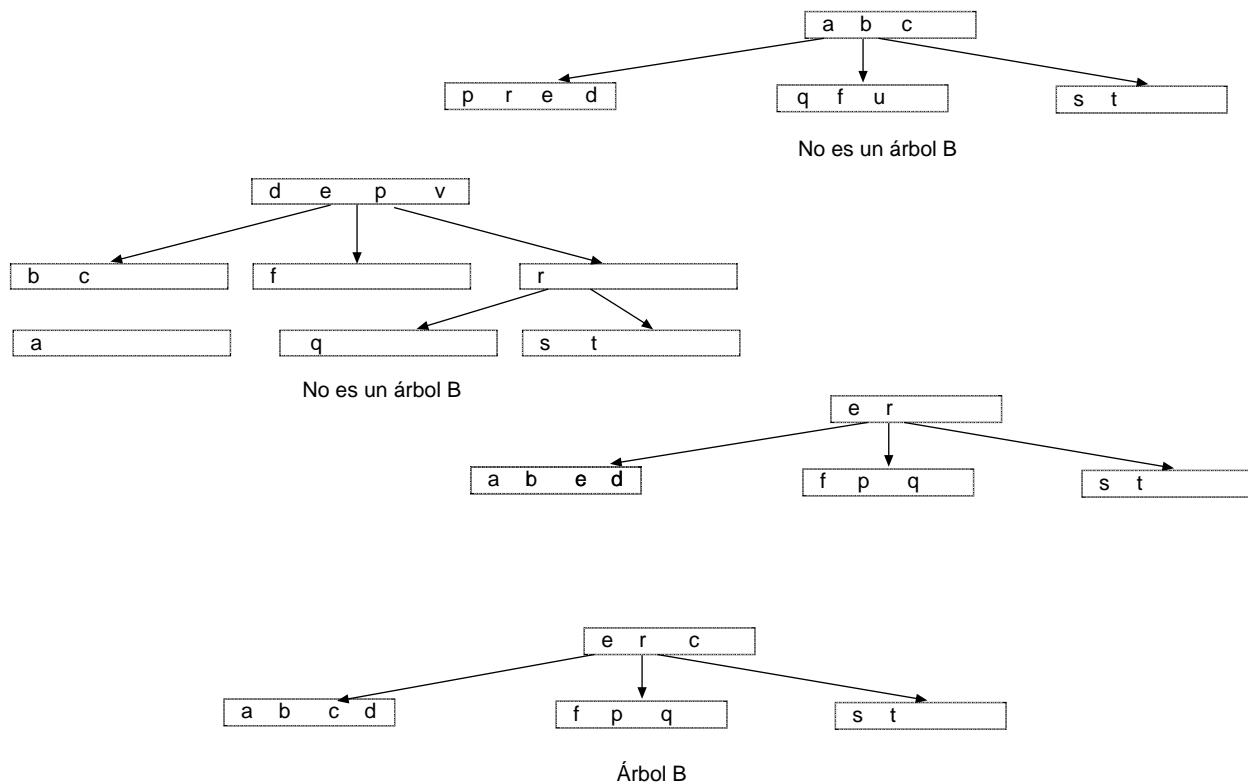


Figura 15.2. Características estructurales de los árboles B.

15.1.1 BÚSQUEDA DE UNA CLAVE

La búsqueda de una clave en un árbol B comienza en la página apuntada por la raíz. Si el puntero a la página es NULL el proceso de búsqueda ha terminado y la clave no está. Cuando el puntero no es NULL se compara la clave buscada (`Clb`) con las existentes en la página y si es igual a alguna de ellas la búsqueda ha terminado (clave encontrada), si no es igual a ninguna de éstas claves la búsqueda continúa en la página apuntada por la rama correspondiente:

- si $Clb < Clave[1]$, por la Rama[0]
- si $Clave[p-1] < Clb < Clave[p]$, por la Rama[p-1] (p puede ser cualquier valor entre 2 y el contador de registros utilizados, Cont)
- si $Clb > Clave[Cont]$, por la Rama[Cont].

Las operaciones descritas se repiten con la nueva página hasta que la clave se encuentre o el puntero a la página sea NULL y, por tanto, se pueda dar por descartado el encontrarla.

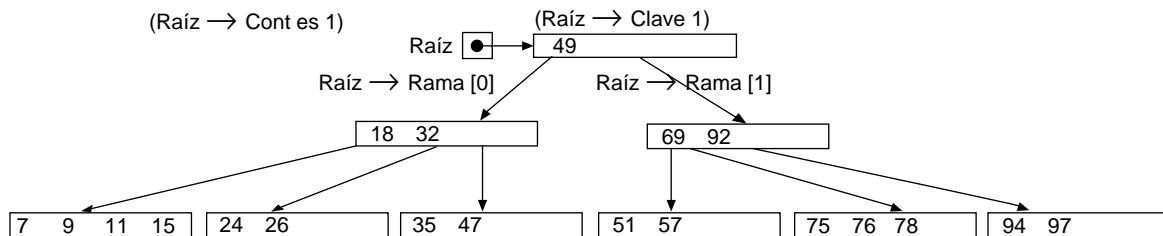


Figura 15.3. Árbol B. Denominaciones de los elementos del nodo raíz.

15.1.2. INSERCIÓN DE INFORMACIÓN

Los árboles B crecen de abajo hacia arriba, es decir desde las hojas hacia la raíz. Los pasos a seguir para insertar un nuevo registro en un árbol B son:

- Si el árbol no está vacío buscar la página y el lugar donde corresponde insertar el nuevo registro. Esta página siempre será una hoja, y el lugar donde colocar el registro en la hoja será a la derecha del puntero por donde se ha bajado buscando la clave por última vez y al encontrar que apunta a NULL ha permitido determinar que la clave y, por tanto, el registro que dicha clave identifica no está.
- Si el número de elementos de la mencionada página hoja es menor que el máximo de claves permitido, el nuevo registro se inserta en el lugar que le corresponde, añadiendo una nueva rama derecha para él con el valor NULL. En caso contrario, es decir si la página está llena y el número de registros utilizados indicados por el contador es igual al máximo de claves o registros permitidos, se crea una nueva página y toda la información, incluido el nuevo registro y las ramas correspondientes, se distribuye equitativamente entre las dos, pasando a la nueva página los registros con claves de mayor valor. A continuación se quita el registro con clave de valor intermedio entre ambas páginas para subirlo a la página antecesora o a un nuevo nodo a crear cuando el desbordamiento se produce a nivel de la raíz.
- Si el desbordamiento no se ha producido a nivel de la raíz, el registro que sube, si cabe, se insertará en la página actual a la derecha del puntero por donde se bajó buscando la clave y tendrá como rama derecha un puntero a la nueva página creada. Si la página actual se desbordara nuevamente se repite el proceso descrito para tal situación en el segundo apartado.
- Si el árbol está vacío o el nodo raíz se desborda será necesario crear un nuevo nodo y colocar en su primera posición el registro a insertar (árbol B vacío) o el que sube tras tratar el desbordamiento del nodo raíz, la rama izquierda de este registro será la antigua raíz y la rama derecha NULL o un puntero a la nueva página que, a causa del desbordamiento, se creó. Por último, la raíz pasará a apuntar al nuevo nodo y la altura del árbol se incrementará en una unidad.

EJEMPLO 15.1. Estudiar el proceso de inserción de la clave 8 en el árbol B de orden 5 que muestra la Figura 15.4

La Figura 15.4 presenta las fases de la inserción del elemento en el árbol B. La clave del elemento a insertar es 8 y la página donde ha de efectuarse la inserción, que en este caso además es el nodo raíz, se encuentra llena. Los pasos para la inserción son: (1) búsqueda de la posición de inserción y (2) como el nuevo elemento no cabe en la página se crea una nueva página y en este caso también una nueva raíz.

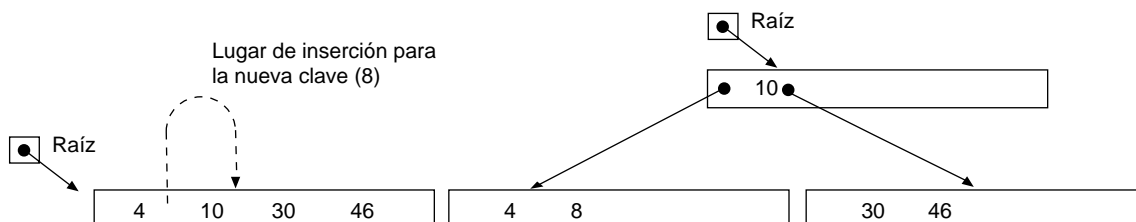


Figura 15.4. Árbol B. Inserción de un nuevo registro con clave 8.

15.1.3. BORRADO FÍSICO DE UN REGISTRO

La operación de borrado físico consiste en quitar un registro o clave del árbol sin que éste pierda sus características de árbol B. Para efectuar esta operación se busca en primer lugar la clave, y si se encuentra hay que considerar dos casos posibles:

- La clave del registro a suprimir se encuentra en una hoja y se puede eliminar inmediatamente.
- La clave del registro a suprimir no está en una página hoja, si no en un nodo interior. Este segundo caso conduce al primero, ya que si el registro a borrar no está en una hoja debe sustituirse por su sucesor o predecesor más inmediato y a continuación mandar eliminar la clave sucesora o predecesora para que la información correspondiente no aparezca duplicada. El sucesor más inmediato de un registro es aquel que se encuentra más a la izquierda en el subárbol derecho más próximo a dicho registro, mientras que el predecesor más inmediato es el que se encuentra lo más a la derecha posible en el subárbol izquierdo más cercano al mencionado registro. Como los registros y las claves predecesora y sucesora siempre están en una hoja, al final, también termina eliminándose una clave en una hoja.

Al borrar un registro en una hoja que no es la raíz hay que tener en cuenta que si el número de elementos de la página (hoja) donde se efectúa el borrado se convierte en inferior a la mitad del máximo de claves permitidas el nodo no satisface la condición de ocupación requerida por los árboles B y se ocasiona un problema que es necesario solucionar. Para ello, a la salida de la recursividad y desde los nodos padre se pregunta por la ocupación de los hijos desde los que se retrocede y si el grado de ocupación es correcto no se hace nada mientras que en caso contrario se aplican diferentes soluciones:

- Si el hermano izquierdo de aquel donde se ha efectuado el borrado tiene más claves que las mínimas necesarias, envía su último registro al padre y el padre envía el registro con la clave anteriormente divisionaria a su descendiente derecho. Para que además se sigan cumpliendo las condiciones de búsqueda la rama más derecha del hermano izquierdo pasa a convertirse en la rama 0 del derecho.
- Si no tiene hermano izquierdo, pero el hermano derecho de aquel donde se ha borrado la clave tiene más claves que las mínimas necesarias, éste envía al padre su primer registro y el padre manda el registro con la clave anteriormente divisionaria a su descendiente izquierdo. Para que además se sigan cumpliendo las condiciones de búsqueda, se debe enviar la rama 0 del hermano derecho al izquierdo. Ahora ya se cumplen las condiciones de árbol B.
- Si el hermano consultado tiene únicamente el número de claves mínimas necesarias se efectúa una fusión. Para ello se baja, desde el padre al hijo izquierdo, el registro con la clave divisionaria entre ambos hermanos y luego se mueven los registros y ramas del nodo derecho al izquierdo y se libera el derecho. La acción de bajar el registro del padre implica la eliminación del mismo en su página, por lo que puede ocurrir que ésta quede con un número de claves inferior al mínimo necesario y al subir otro nivel y retornar al punto donde se pregunta por la condición de ocupación se repita la necesidad de considerar nuevamente los diferentes casos expuestos como solución a este tipo de problema. Los procesos de fusión podrán pues continuar hasta la raíz del árbol.

Si el árbol tiene un nodo raíz con un único registro que se borra o las fusiones llegan hasta la raíz y bajan el único registro de este nodo, la raíz pasa a ser la rama 0 del actual nodo raíz que, a continuación se libera, y la altura del árbol disminuye en una unidad.

EJEMPLO 15.2. Estudiar el proceso de borrado del elemento con clave 4 del árbol B que muestra la Figura 15.5.

La figura 15.5 muestra los pasos de la operación de borrado de un elemento en un árbol B. El orden del árbol es 5 y la clave del elemento a borrar 4. A consecuencia de la eliminación del 4 el nodo queda con un número de claves menor que el mínimo necesario. Como el nodo derecho no puede prestar claves a su hermano izquierdo, se origina una fusión que alcanza la raíz y disminuye la altura del árbol.

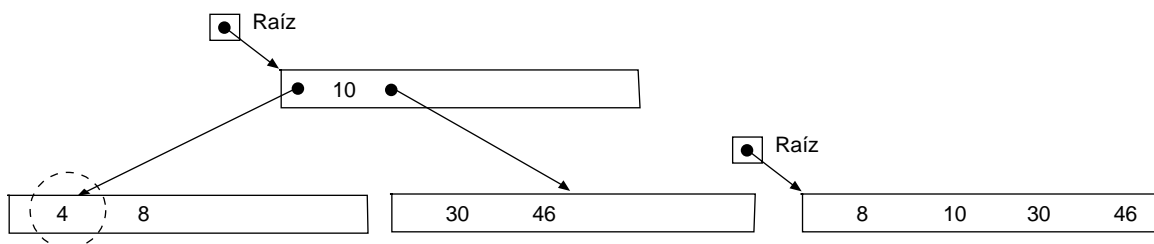


Figura 15.5. Árbol B. Borrado del registro con clave 4, a consecuencia del borrado disminuye la altura del árbol.

15.2. Realización de un árbol B en memoria externa

Los árboles B estudiados hasta el momento trabajan en memoria principal, pero existen ocasiones en que resulta conveniente organizar los datos con estructura de árbol B directamente sobre el dispositivo de almacenamiento externo.

Los archivos con estructura de árbol B tratan de minimizar el número de accesos a disco que, en el peor de los casos, pueden llegar a ser necesarios para localizar una clave. Estos archivos son de tipo directo y se encuentran formados por registros, que son bloques de información que se leen cuando se accede al disco. Los bloques o registros del archivo, en realidad, constituyen las páginas o nodos del árbol B y, para un árbol de orden M_r , deben tener capacidad para agrupar:

- $N (N \leq M_r)$ elementos, variables de tipo entero que actúan como punteros, capaces de almacenar la posición de las páginas hijas en el disco.
- $N - 1$ registros con claves que actúan clasificando la información almacenada en las páginas o nodos descendientes.

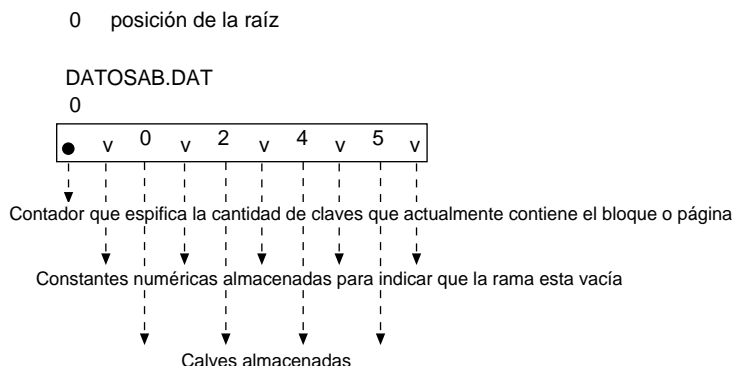
Una vez diseñados los nodos de esta forma, los procesos de búsqueda, inserción, eliminación y recorrido del archivo seguirán las mismas reglas y resultarán análogos a los empleados para realizar esas mismas operaciones cuando se trabaja con un árbol B en memoria interna. No obstante, habrá que considerar que para crear un nuevo nodo (página) habrá que buscar un hueco libre en el archivo, en lugar en la memoria. Los nuevos nodos en un principio son nuevos registros de archivo que se añaden al final y aumentan el tamaño del mencionado archivo, pero más adelante, con las operaciones de borrado, pueden quedar bloques libres en posiciones intermedias del fichero que hay que aprovechar y, para ello, se crea y utiliza un archivo auxiliar donde guardar la posición en el disco del nodo raíz y las de los bloques libres entre una ejecución del programa y otra. Guardar la raíz es imprescindible pues el nodo raíz no ocupa una posición fija en el archivo de datos y, sin embargo, su lectura inicial es la que permite el acceso a toda la estructura.

EJEMPLO 15.3. Puesto que se desea guardar una serie de datos en disco en un archivo con estructura de árbol B de orden 5 (es decir cuyo máximo número de claves en un nodo es 4) explicar las operaciones necesarias.

Los archivos que se van a crear son dos:

HUECOS.DAT que, almacenará la raíz y posiciones libres intermedias del archivo de datos.
DATOSAB.DAT para contener los datos.

Suponga que los archivos se encuentran inicialmente vacíos y se introducen las claves 2, 4, 0, 5. El contenido del primer registro del archivo de datos al terminar de introducir dicha serie de números y su posición relativa al comienzo del mismo es:



Imagine que a continuación se añaden las claves 9 y 3. La adición de la clave 9 da origen a la creación de una nueva página y a la formación de una nueva raíz (segunda nueva página). Las nuevas páginas se sitúan al final del archivo de datos y la raíz ya no coincide con el primer registro de dicho archivo.

Al terminar el trabajo y cerrar el archivo será necesario almacenar la posición de la raíz en HUECOS.DAT, pues sin esta información cuando se abra el archivo de nuevo no se podrá acceder al resto de la estructura.

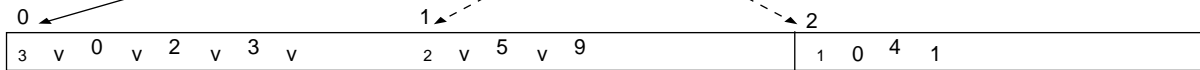
HUECOSAB.DAT

2

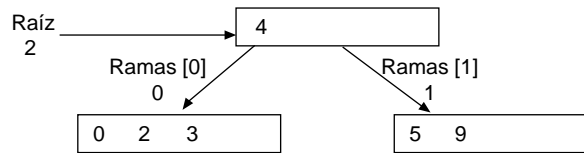
 Posición de la Raíz en DATOS.DAT

DATOSAB.DAT

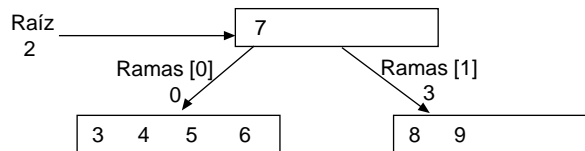
(posición de los registros del archivo de datos. páginas o nodos)



La situación se podrá representar de forma análoga a como se efectúa con los árboles B almacenados en memoria interna.



Si se abren nuevamente los archivos, se añaden las claves 6, 7 y 8, se borra la 0 y la 2 y se vuelven a guardar, el resultado final será:



y el contenido de los archivos:

HUECOSAB.DAT

Raíz hueco
libre

2	1
---	---

DATOSAB.DAT



15.3. Árboles B*

Constituyen una mejora de los árboles B con la finalidad de aumentar el promedio de utilización de los nodos. La inserción de nuevas claves en el árbol B* supone que si el nodo que le corresponde está lleno mueve las claves a uno de sus hermanos (de manera similar a como se mueven en la eliminación cuando hay que restaurar el número de claves de un nodo), con lo cual se pospone la división del nodo hasta que ambos hermanos estén completamente llenos y, entonces, estos pueden dividirse en tres nodos, cada uno de los cuales estará lleno en sus dos terceras partes.

15.4. Árboles B+

Los árboles B+ constituyen otra mejora sobre los árboles B, pues conservan la propiedad de acceso aleatorio rápido y permiten además un recorrido secuencial rápido. En un árbol B+ todas las claves se encuentran en hojas, duplicándose en la raíz y nodos

interiores aquellas que resulten necesarias para definir los caminos de búsqueda. Para facilitar el recorrido secuencial rápido las hojas se pueden vincular, obteniéndose, de esta forma, una trayectoria secuencial para recorrer las claves del árbol. Los árboles B+ ocupan algo más de espacio que los árboles B, debido a la mencionada duplicidad en algunas claves. En los árboles B+ las claves de las páginas raíz e interiores se utilizan únicamente como índices.

Búsqueda en un árbol B+.

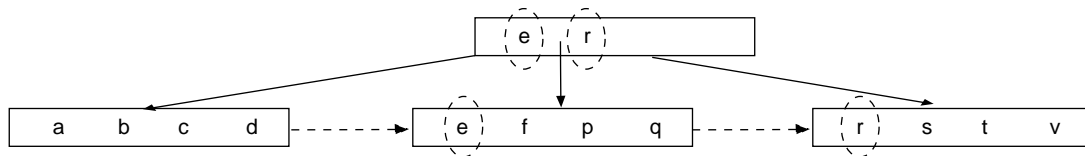
En este caso la búsqueda no debe detenerse cuando se encuentre la clave en la página raíz o en una página interior, si no que debe proseguir en la página apuntada por la rama derecha de dicha clave.

Inserción en un árbol B+.

Su diferencia con el proceso de inserción en árboles B consiste en que cuando se inserta una nueva clave en una página llena, ésta se divide también en otras dos, pero ahora la clave que se sube como separador entre ambas no se retira o elimina de la hojas y lo que se sube a la página antecesora es una copia de la clave central.

EJEMPLO 15.4. *Mostrar el árbol B+ de orden 5 resultante de la inserción secuencial de las siguientes claves p, v, d, e, b, c, s, a, r, f, t, q.*

La Figura 15.6 muestra el árbol B+ de grado 5 que se originaría por la introducción secuencial de las siguientes claves p, v, d, e, b, c, s, a, r, f, t, q.



El orden de inserción de los diversos elementos fue: p v d e b c s e r t v q

Figura 15.6. Árbol B+.

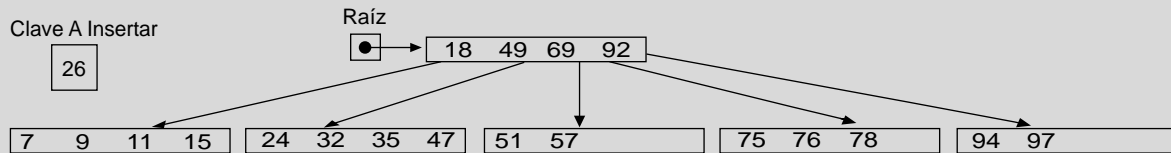
Borrado en un árbol B+.

La operación de borrado debe considerar:

- Si al eliminar la clave (siempre en una hoja) el número de claves es mayor o igual al mínimo permitido el proceso ha terminado. Las claves de las páginas raíz o internas no se modifican aunque sean una copia de la eliminada, pues siguen constituyendo un separador válido entre las claves de las páginas descendientes.
- Si al eliminar la clave el número de ellas que quedan en la página es inferior al mínimo tolerado será necesaria una fusión y redistribución de las mismas tanto en las páginas hojas como en el índice.

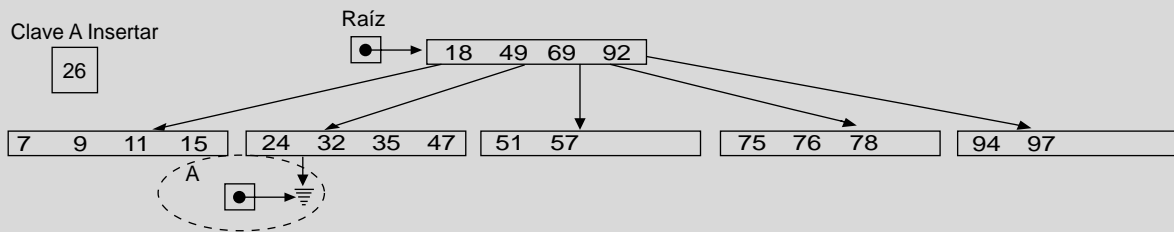
PROBLEMAS DE SEGUIMIENTO

15.1. Describir paso a paso el proceso de inserción de la clave 26 en el siguiente árbol B de orden 5.



Solución

El primer paso será bajar buscando la clave desde la raíz hasta comprobar que dicha clave no existe.



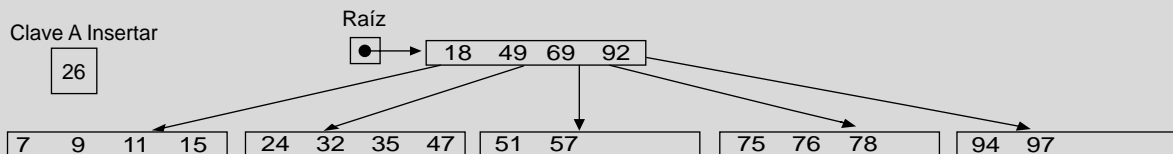
Efectivamente, la clave 26 no se encuentra en el árbol y deberá ser insertada.

HUECOSAB.DAT

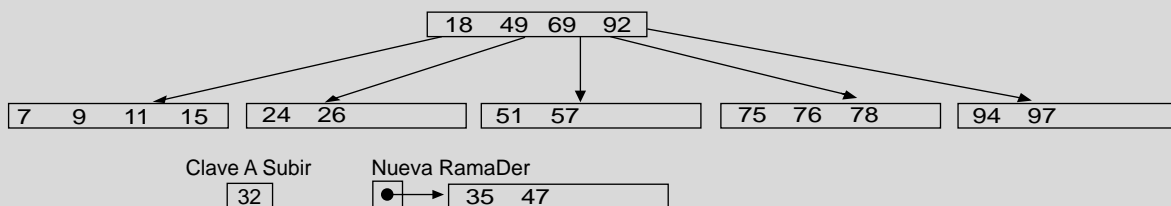
Raíz hueco
libre

2	1
---	---

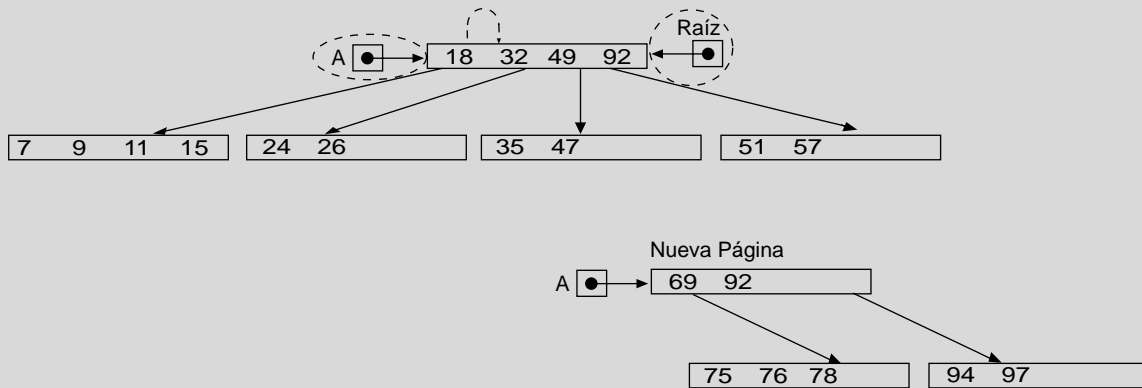
Como no cabe en la página, se crea una nueva página y se distribuye la información entre ambas.



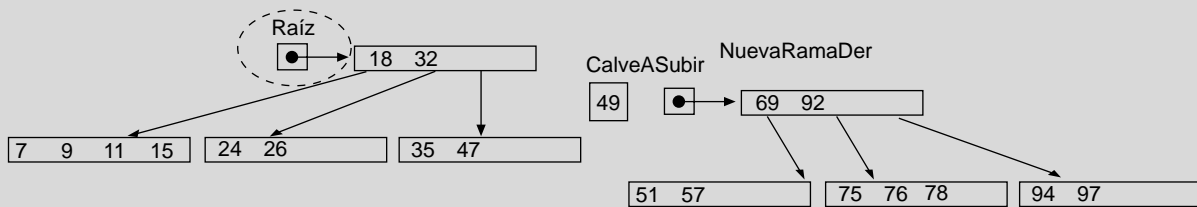
La clave con valor intermedio o clave a subir es 32 y la Nueva RamaDer queda como se indica, tras trasladar las claves y ramas necesarias a la nueva página.



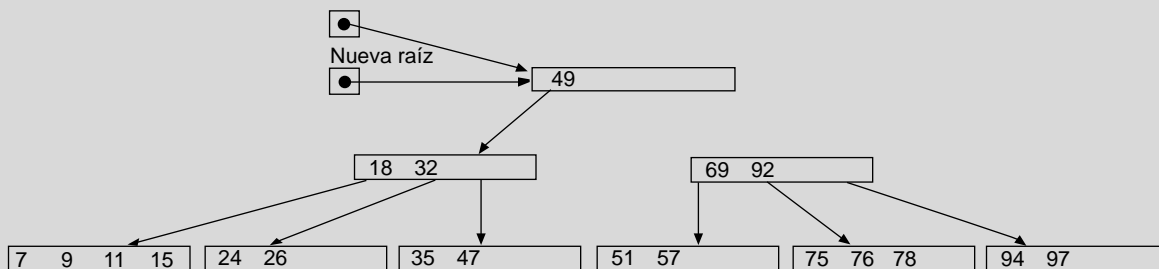
Como la clave a subir tampoco cabe en la página donde debe ser insertada, se crea una nueva página y se distribuye la información entre ambas.



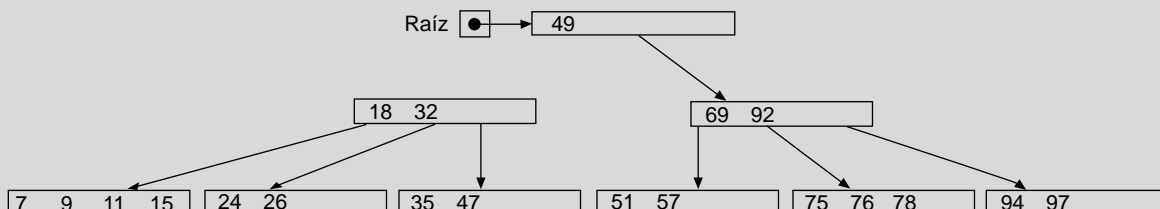
La clave con valor intermedio o clave a subir es 49 y la NuevaRamaDer queda como se indica, tras terminar de trasladar todas las ramas necesarias a la nueva página.



Las sucesivas divisiones se han propagado hasta la raíz, y se necesita crear un nuevo nodo donde se colocan la antigua Raíz, la ClaveASubir y NuevaRamaDer. El puntero a este nodo se convierte en la nueva Raíz.

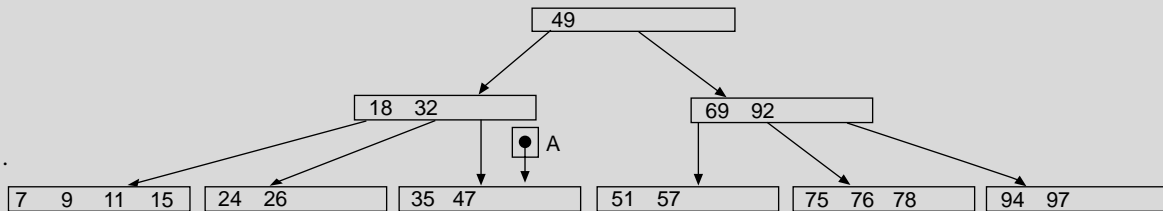


Describir paso a paso el proceso de borrado de la clave 47 en el siguiente árbol B de orden 5 obtenido en el problema anterior, 15.1

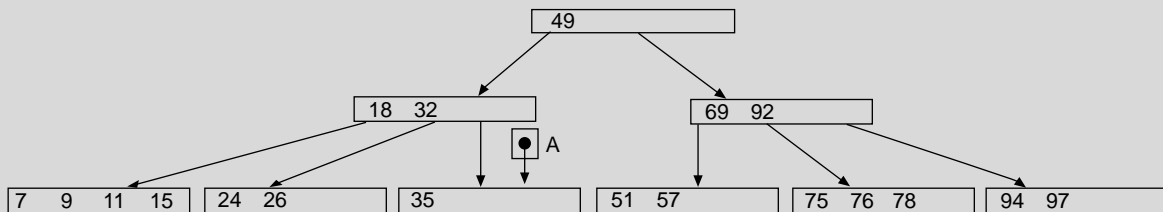


Solución

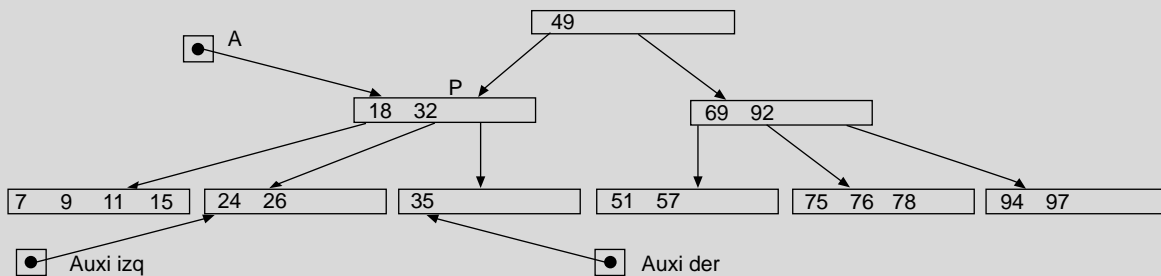
En primer lugar se baja buscando en el árbol hasta que se encuentra la clave o se determina que no está. En este caso la clave está en una hoja.



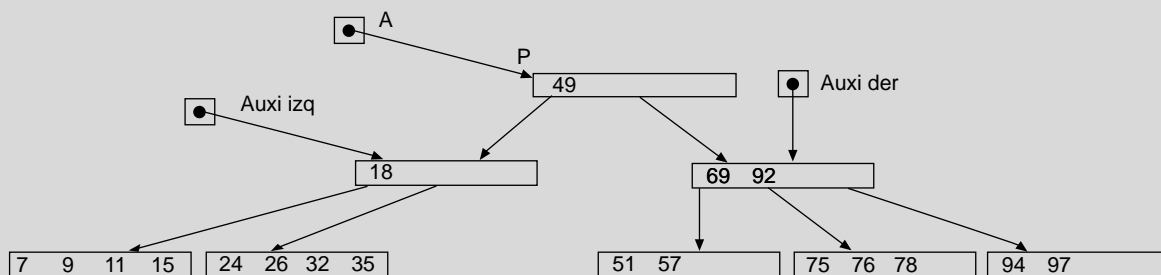
A continuación se quita la clave y se retrocede por el camino de búsqueda.



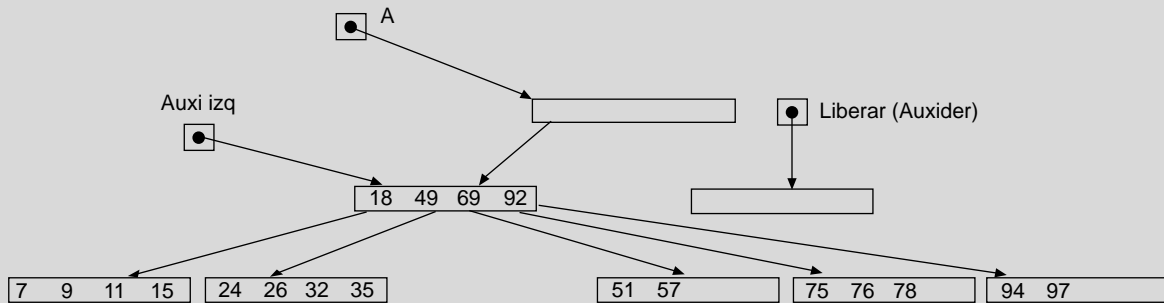
Como $A \rightarrow \text{Rama}[p]$ ha quedado con un número menor de claves que el mínimo permitido, tiene hermano izquierdo y éste no posee suficientes claves como para cederle una y quedarse con el mínimo aceptable, se efectúa la fusión o combinación de ambos nodos



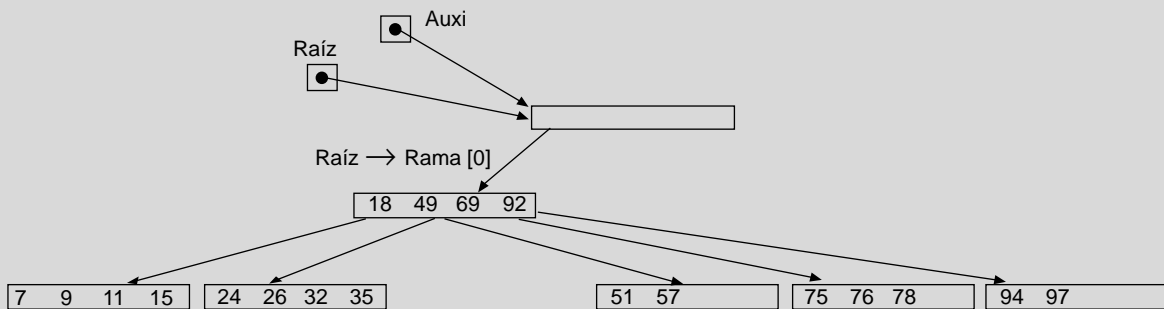
y se continúa retrocediendo por el camino de búsqueda.



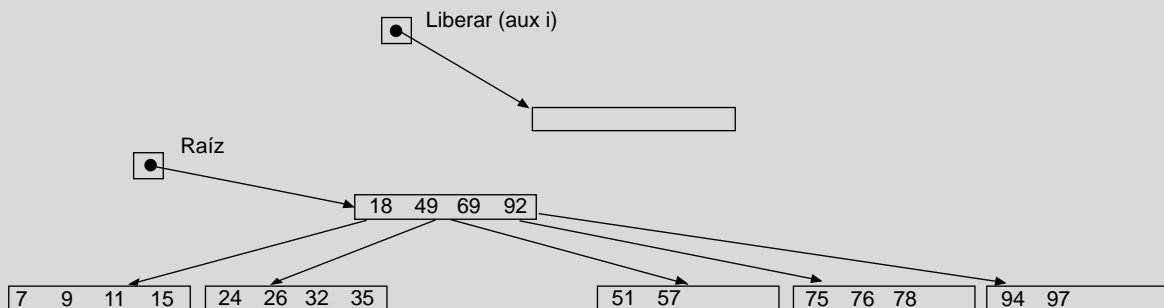
Como $A \rightarrow \text{Rama}[0]$ ha quedado con un número menor de claves que el mínimo permitido, sólo tiene hermano derecho, y éste no posee suficientes claves como para cederle una y quedarse con el mínimo aceptable, se efectúa la fusión o combinación entre $A \rightarrow \text{Rama}[0]$ y $A \rightarrow \text{Rama}[1]$.



Como se ha bajado la última clave de la raíz del árbol, ésta debe ser sustituida por $\text{Raíz} \rightarrow \text{Rama}[0]$.

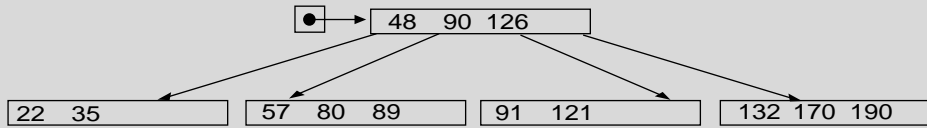


Por último, se libera el antiguo nodo raíz (Auxi). Como puede observarse el árbol ha decrecido en altura.



- 15.3.** Dada la secuencia de claves enteras: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80; dibujar el árbol B de orden 5 cuya raíz es R, que se corresponde con dichas claves.

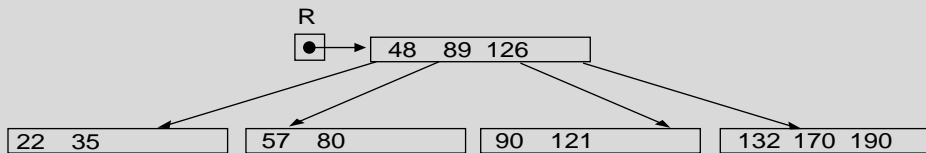
Resultado



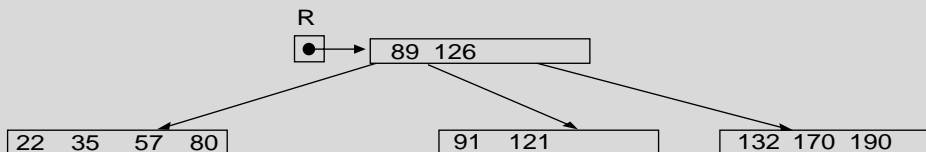
- 15.4.** En el árbol R del problema anterior, eliminar la clave 91 y dibujar el árbol resultante. Eliminar, ahora la clave 48. Dibujar el árbol resultante, ¿ha habido reducción en el número de nodos?

Resultado

Tras eliminar la clave 91:



Tras eliminar la clave 48:



Como se puede observar hay reducción en el número de nodos.

PROBLEMAS BÁSICOS

- 15.5.** Escribir el pseudocódigo del proceso de inserción de un registro en un árbol B.

Solución

```

algoritmo insertar(E/S ArbolB: Raíz; E <tipo_clave>: ClaveAInsertar)
var
  lógico: SubirClave
  <tipo_clave>: ClaveASubir
  ArbolB: NuevaRamaDer, Nueva
inicio
  BuscarClavePropagarDivisión(Raíz, ClaveAInsertar, SubirClave, ClaveASubir, NuevaRamaDer)
  
```

```

si SubirClave entonces
  /*
  Indica que el árbol estaba vacío o que la raíz ha necesitado dividirse y es necesario crear
  una nueva raíz y subir a ella la clave intermedia de la anterior
  */
  reservar(Nueva)
  Nueva^.Cont ← 1
  Nueva^.Clave[1] ← ClaveASubir
  Nueva^.Rama[1] ← NuevaRamaDer
  Nueva^.Rama[0] ← Raíz
  Raíz ← Nueva
Fin_si
Fin_algoritmo

algoritmo BuscarClavePropagarDivisión(E ArbolB: A; E <tipo_clave>: Cl;
                                     S lógico: SubirClave;
                                     <tipo_clave>: ClaveASubir;
                                     ArbolB: NuevaRamaDer)
                                     S
                                     S

var
  entero: P                               /*número de rama por la que se continúa la búsqueda*/
  lógico: Encontrada                      /*si ya existe no se inserta*/
inicio
  si A = nulo entonces
    /*termina el proceso recursivo y retorna para realizar la inserción*/
    SubirClave ← verdad
    ClaveASubir ← Cl
    NuevaRamaDer ← nulo
  si_no
    <Se recorre la página apuntada por raíz comparando sus claves con la que se desea insertar
    (Cl), de esta forma, o se localiza la clave o se obtiene la posición (P) la de rama por donde
    debiera continuar el proceso de búsqueda>
    si Encontrada entonces
      /*la clave ya existe y no se puede insertar*/
      SubirClave ← falso
    si_no
      BuscarClavePropagarDivisión(A^.Rama[p], Cl, SubirClave, ClaveASubir, NuevaRamaDer)
      /*Las llamadas recursivas devuelven el control a este punto, por tanto
      siempre se ejecuta la sentencia siguiente*/
      si SubirClave entonces
        si A^.Cont < m entonces
          InsertarEnPágina(A, ClaveASubir, NuevaRamaDer, P)
          SubirClave ← falso
        si_no
          /*Como no hay espacio llama a DividirPágina que recibe a través de ClaveASubir y
          NuevaRamaDer la clave y rama a insertar. DividirPágina crea una nueva página y copia
          en ella las claves y ramas adecuadas. Además, el procedimiento selecciona entre ambas
          páginas la apropiada e inserta la clave y rama recibidas en la posición que les corres-
          ponde dentro de la página. Por último, envía la clave más a la derecha de la página
          izquierda y el puntero a la página creada, mediante ClaveASubir y NuevaRamaDer, hacia
          arriba para una inserción posterior en otra página*/
          DividirPágina(A, ClaveASubir, NuevaRamaDer, P, ClaveASubir, NuevaRamaDer)
        fin_si
      fin_si

```

```

    fin_si
  fin_si
fin_procedimiento

```

15.6. *Escribir el pseudocódigo del proceso de borrado de físico de un registro con una determinada clave en un árbol B.*

Solución

```

algoritmo borrar(E/S ArbolB: Raíz; E <tipo_clave>: ClaveABorrar)
  var
    lógico: Encontrada
    ArbolB: Aux1
  inicio
    BuscarClavePropagarFusion(Raíz, ClaveABorrar, Encontrada);
    si no Encontrada entonces
      /*La clave no está*/
    si_no
      si (Raíz^.Cont=0) entonces
        /*La raíz se ha quedado vacía*/
        <Se sustituye por Raíz^.Rama[0] y se libera el nodo apuntado anteriormente
        por ella>
        fin_si
      fin_si
    fin_algoritmo

algoritmo BuscarClavePropagarFusion(E ArbolB: A; E <tipo_clave>: Cl; S lógico: Encontrada)
  var
    entero: P
    <tipo_clave>: sucesora
  inicio
    si Arbolvacio(A) entonces
      Encontrada ← falso
    si_no
      <Se recorre la página apuntada por Raíz comparando sus claves con la buscada, de esta forma,
      o se localiza la clave o se obtiene el número de la rama, P, por donde debiera continuar el
      proceso de búsqueda>
      si <se encuentra la clave> entonces
        si <está en una hoja> entonces
          <se quita>
        si_no
          <se busca la clave menor en la rama derecha de la actual, es decir la que sigue en valor
          a la actual. Se copia en la clave a borrar dicha clave menor y se manda eliminar la
          duplicada en su nodo, el procedimiento se llama a sí mismo para eliminar la clave en la
          hoja>
          fin_si
        si_no
          <el procedimiento se llama a sí mismo para intentar eliminar la clave en la rama obtenida>
        fin_si
      /*Las llamadas recursivas devuelven el control a este punto del procedimiento*/
      si <el nodo hijo no es una hoja y tiene un número de claves menor que el mínimo
      necesario> entonces
        /*la página de la rama está desocupada y habrá que arreglarla modificando la clave en el padre

```

```

    y añadiendo elementos a la desocupada de la página de la izquierda, de la de la derecha o bien
    fusionar dos páginas. El proceso de fusión puede propagarse hacia arriba y llegar a disminuir
    la altura del árbol en una unidad*/
    Arreglar(A, P)
    fin_si
  fin_si
fin_algoritmo

```

15.7. *Escribir una función que efectúe la búsqueda de una clave en un árbol B de forma iterativa.*

Análisis

Se utiliza un puntero auxiliar que, en un principio, se coloca apuntando a la raíz del árbol. Mientras el valor de dicho puntero sea distinto de NULL y la clave no haya sido encontrada se compara la clave buscada con las existentes en la página, de esta forma o se localiza la clave o se halla la rama por la que debe continuar la búsqueda y el puntero pasa a apuntar a la nueva página. Se consideran las claves de tipo entero.

Codificación

```

void BuscarEnArbol (Pagina* Raiz, TipoClave ClaveBuscada, int* Encontrada,
                   Pagina** Pag, int* Posic)
{
    *Pag = Raiz;
    *Encontrada = False;
    while (*Pag != NULL && !(*Encontrada))
    {
        if (ClaveBuscada < (*Pag)->Clave[1])
        {
            *Encontrada = False;
            *Posic = 0;
        }
        else
        {
            *Posic = (*Pag)->Cont;
            while ((*Posic > 1) && (ClaveBuscada < (*Pag)->Clave[*Posic]))
                *Posic = *Posic - 1;
            *Encontrada = (ClaveBuscada == (*Pag)->Clave[*Posic]);
        }
        if (!*Encontrada)
            *Pag = (*Pag)->Rama[*Posic];
    }
}

```

15.8. *Escribir un módulo de programa que reciba como parámetro un árbol B y muestre sus claves en Preorden.*

Análisis

El recorrido se implementa de forma similar al de su mismo tipo en un árbol binario. Se usa un bucle para acceder a las claves de un nodo, cada clave divide a sus descendientes a la forma de árbol binario.

Codificación

```

void Preorden(Pagina* Raiz)

```



```

{
    int j;

    if (Raiz != NULL)
    {
        for (j = 1; j <= Raiz->Cont; j++)
        {
            printf("%d  ",Raiz->Clave[j]);
            Preorden(Raiz->Rama[j-1]);
        }
        Preorden(Raiz->Rama[Raiz->Cont]);
    }
}

```

15.9. *Escribir un módulo de programa que reciba como parámetro un árbol B y presente por pantalla sus claves en Postorden.*

Análisis

Con una pequeña variación que permita recorrer las claves de cada nodo, se puede aplicar a los árboles B el algoritmo utilizado para efectuar dicho recorrido en árboles binarios.

Codificación

```

void Postorden(Pagina* Raiz)
{
    int j;

    if (Raiz != NULL)
    {
        Postorden(Raiz->Rama[0]);
        for (j = 1; j <= Raiz->Cont; j++)
        {
            Postorden(Raiz->Rama[j]);
            printf("%d  ",Raiz->Clave[j]);
        }
    }
}

```

15.10. *Implementar de forma iterativa el recorrido Inorden de un árbol B.*

Análisis

El procedimiento será análogo al explicado en los árboles binarios. La diferencia principal radica en la pila auxiliar que es necesario utilizar y que en el caso de los árboles B debe almacenar, además de la dirección del nodo, el índice de la rama.

Codificación

```

/*
    Declaraciones del TAD Pila
*/

```

```

/* Implementación del recorrido en inorden de manera iteativa */

void InordenIt (Pagina* Raiz)
{
    int I;
    Pagina* P;
    Pila* P1;

    VacíaP(&P1);
    P = Raiz;
    do
    {
        I = 0;
        while (! Arbolvacío(P))
        {
            AnadeP(&P1, P, I);
            P = P->Rama[I];
        }
        if (! EsVacíaP(P1))
        {
            PrimeroP(P1, &P, &I);
            BorrarP(&P1);
            I = I + 1;
            if (I <= P->Cont)
            {
                printf("%d  ",P->Clave[I]);
                if (I < P->Cont)
                    AnadeP(&P1, P, I);
                P = P->Rama[I];
            }
        }
    }while (!EsVacíaP(P1) || !Arbolvacío(P));
}

```

15.11. Escribir un procedimiento que dibuje en pantalla un árbol B.

Análisis

Para dibujar un árbol B (pequeño) en pantalla basta con introducir un parámetro que indique la distancia relativa a la que aparecerá cada nodo del árbol del margen izquierdo de la pantalla. De esta forma si se realizan llamadas recursivas aumentando en una unidad el valor del parámetro, se tendrán determinados los desplazamientos de los hijos de un nodo. Será necesario escribir la mitad de los hijos antes que la raíz, después escribir las claves almacenadas en ella dando un salto de línea y terminar presentando el resto de los hijos del nodo.

Codificación

```

void Escribearchol(Pagina* A, int H)
{
    int I, J;
    if (A != NULL)
    {
        Escribearchol(A->Rama[A->Cont], H+1);
        for (I = A->Cont; I >= 1; --)

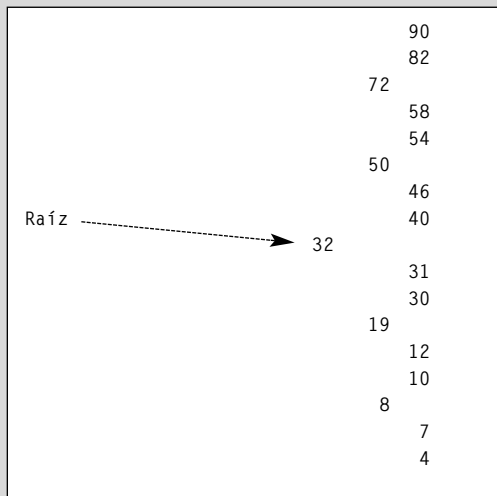
```

```

    {
        for (J = 1; J<= H; J++)
            printf("    ");
        //desplazamiento que se estima para el valor relativo H
        printf("%4d\n",A->Clave[I]);
        Escribeárbol(A->Rama[I-1], H+1);
    }
}
}

```

Ejecución



PROBLEMAS AVANZADOS

15.12. Implementar las operaciones primitivas para el manejo de un árbol B y un programa principal que las utilice.

Análisis

Es una codificación de la teoría anteriormente expuesta. Los detalles se explicarán mediante comentarios introducidos en el código. Se considera que las claves son de tipo entero.

Codificación (la implementación de las funciones del árbol B se encuentran en la página Web del libro)

```

#include <stdio.h>
#include <stdlib.h>
#define mr 5 //Orden del árbol
/* En un árbol B de orden mr, el número de ramas de una página es uno
   más que el de sus claves

```

```

*/
#define m 4 //Numero maximo de claves en un nodo
#define True 1
#define False 0

typedef int TipoClave;

typedef struct pagina
{
    int Cont;
    TipoClave Clave[mr];
    struct pagina* Rama[mr];
    /* Las claves que se consideran son de la 1 a la m
       y las ramas de la 0 a la m
    */
}Pagina;

//Procedimiento auxiliar del programa principal

void Menu(int* Opcion);

//Primitivas para el manejo del árbol

void Inicializar(Pagina** Raiz);
int Arbolvacio(Pagina* Raiz);
void BuscarEnArbol (Pagina* Raiz, TipoClave ClaveBuscada, int* Encontrada,
                   Pagina** PunteroAPaginaConClave, int* Posicion);
void BuscarEnPagina (Pagina* A, TipoClave ClaveBuscada, int * Encontrada,
                   int* Posicion);

void Insertar(Pagina** Raiz, TipoClave ClaveAInsertar);
void BuscarClavePropagarDivision(Pagina* A, TipoClave Cl, int* SubirClave,
                                TipoClave* ClaveASubir, Pagina** NuevaRamaDer);
void InsertarEnPagina(Pagina* A, TipoClave ClaveAInsertar,
                    Pagina* RamaDer, int Posicion);
void DividirPagina(Pagina* A, TipoClave Clave, Pagina* RamaD, int Posicion,
                  TipoClave* ClaveASubir, Pagina** NuevaRamaDer);
void Borrar (Pagina** Raiz, TipoClave ClaveABorrar);
void BuscarClavePropagarFusion(Pagina* A, TipoClave Cl, int* Encontrada);
TipoClave Menor(Pagina* A);
void Arreglar(Pagina* A, int P);
void Quitar(Pagina* A, int Posicion);
void Combina(Pagina* A, int P);
void MoverADrcha(Pagina* A, int P);
void MoverAIzqda(Pagina* A, int P);
void Inorden(Pagina* Raiz);

// Programa principal

int main ()
{
    Pagina* Raiz, * A;
    TipoClave Cl;

```

```

int Opcion, P, Esta;
char ch;

Inicializar(&Raiz);
do
{
    Menu(&Opcion);
    switch (Opcion)
    {
        case 1:
            printf("\nIndique la clave a insertar: ");
            scanf("%d", &Cl);
            Esta = False;
            BuscarEnArbol(Raiz, Cl, &Esta, &A, &P );
            if (!Esta)
                Insertar(&Raiz, Cl);
            else
                printf("La clave ya está");
            break;
        case 2:
            printf("\nIndique la clave a eliminar: ");
            scanf("%d", &Cl);
            Borrar(&Raiz, Cl);
            break;
        case 3:
            Inorden(Raiz);
            break;
    }
    printf("\nPulse una tecla para continuar\n");
    ch = getchar();
    putchar(ch);
}while (Opcion != 4);
printf("FIN");
ch = getchar();
return 0;
}

void Menu(int* Opcion)
{
    printf("\n1. Insertar una clave\n");
    printf("2. Eliminar una clave\n");
    printf("3. Listar\n");
    printf("4. Fin\n\n");
    do
    {
        printf("Opción ?: ");
        scanf("%d", Opcion);
    }while ((*Opcion < 1) || *Opcion > 4);
}

```

15.13. Se define un árbol B de orden m de la siguiente forma:

- 1) Todas las hojas están a un mismo nivel.
- 2) Todos los nodos internos menos la raíz tienen a lo sumo $m+1$ hijos y como mínimo $(m+1) \div 2$.
- 3) El número de claves en cada nodo interno es uno menos que el de sus hijos, y estas claves dividen a las de los hijos a manera de un árbol de búsqueda.
- 4) La raíz tiene como máximo $m+1$ hijos, pero puede llegar a tener hasta 2 si no es una hoja y ninguna si el árbol consta de raíz solamente.

Se pide:

- a) Estudiar asintóticamente el número n de claves que puede almacenar un árbol B de orden m y altura h.
- b) Calcular el número máximo y mínimo de claves que se puede almacenar en función de h y m.
- c) Suponga que $m = 9$ y que se quieren almacenar un millón de claves. ¿Qué altura tendrá como máximo y mínimo el árbol B? Utilice 2^{20} , que es aproximadamente igual a un millón de claves. Compárelo con la altura de un árbol binario de búsqueda de mínima altura.
- d) Haga una estimación de h para que el árbol B pueda almacenar 100 millones de claves (más del doble del número de habitantes estimado que tiene España) para $m = 9$. Compárelo con la altura de un árbol binario de búsqueda.
- e) Como ya sabe cada vez que se da una orden de lectura en memoria externa se leen 512 Bytes, ó 2^{512} Bytes, ó 4^{512} Bytes dependiendo del sistema operativo. Suponga que el árbol B se almacena en memoria externa (como así ocurre en la práctica) y que cada vez que se accede a la memoria externa se lea un nodo del árbol. Estime m cuando las claves a almacenar son enteras (2 Bytes) y estime la altura mínima y máxima que tendrá el árbol B para almacenar 100 millones de claves en alguno de los casos expuestos (por ejemplo para la altura máxima use 4^{512} y para la altura mínima use 2^{512}).

Solución

- a) El número n de claves que pueden almacenarse en las distintas alturas en función de m y h para el caso de almacenamiento máximo es:

Altura 1	m
Altura 2	$m^*(m+1)$
Altura 3	$m^*(m+1)^2$
.....
.....
Altura h	$m^*(m+1)^{h-1}$

Teniendo en cuenta que los resultados obtenidos forman parte de los términos de una progresión geométrica de razón $m+1$ y conociendo que la fórmula de la suma de los h primeros términos de una progresión geométrica es:

$$s = \frac{m(m+1)^{h-1}(m+1) - m}{m+1 - 1}$$

después de simplificar queda la siguiente igualdad: $n = (m+1)^h - 1$ y se deduce que n es $O((m+1)^h)$. Es decir, el número de claves crece asintóticamente de acuerdo con una función potencial cuya base es el orden del árbol B más una unidad, y cuyo exponente es la altura del árbol B, independientemente del número exacto de claves que se almacene en cada nodo.

- b) A partir de la igualdad obtenida previamente en el apartado a que relaciona n con m y con h se tiene que el número máximo y mínimo de claves vienen dados respectivamente por:

$$n = (m+1)^h - 1$$

$$n = \frac{m/2}{(m+1)/2 - 1} ((m+1)/2)^h - 1 \approx ((m+1)/2)^h - 1$$

En la última igualdad se ha tenido en cuenta que $m \div 2 \approx (m+1) \div 2 - 1$.

- c) Para el caso que se nos plantea $m = 9$ (poner $m = 9$ es para que $m+1$ valga 10 y sea bastante sencillo poner las igualdades), con lo que para almacenar un millón de claves basta con establecer la siguiente igualdad para el caso de cada altura.

$$1.000.000 = (9+1)^h - 1 \quad \text{y} \quad 1.000.000 = ((9+1) \div 2)^h - 1$$

Entonces la altura h será respectivamente para los casos mínimo y máximo de:

$$h = \log(1.000.000) = 6 \quad h = \log_5(1.000.000) = 6 * 1,430676 \approx 9$$

Si se compara con la altura dada por un árbol binario de búsqueda y considerando que un árbol de estas características almacena un total de $n = 2^h - 1$ claves se obtiene que la altura h es aproximadamente igual a 20, ya que

$$2^{20} = 2^{10*2} \approx 1.000 * 1.000 = 1.000.000.$$

Obsérvese la diferencia entre el número de accesos a nodos en el caso de un árbol B y un árbol binario; pasa de estar entre 6 y 9 a 20, si bien ambas siguen siendo logarítmicas.

- d) Para este análisis, de nuevo $m = 9$, con lo que para almacenar un total de cien millones de claves basta con establecer la siguiente igualdad para el caso de cada un de las alturas:

$$100.000.000 = (9+1)^h - 1 \quad \text{y} \quad 100.000.000 = ((9+1) \div 2)^h - 1 = 5^h - 1$$

por consiguiente la altura h será, respectivamente, para los casos mínimo y máximo de:

$$h = \log(100.000.000) = 8 \quad h = \log_5(100.000.000) = 8 * 1,430676 \approx 12$$

Si se compara con la altura dada por un árbol binario de búsqueda y teniendo en cuenta que un árbol de estas características almacena un total de $n = 2^h - 1$ claves se obtiene que la altura h es aproximadamente igual a:

$$20 + \log_2(100) \approx 27, \text{ ya que } 100 * 2^{20} = 100 * 2^{10*2} \approx 100 * 1.000 * 1.000 = 100.000.000$$

Observe que con sólo altura entre 8 y 9 se puede almacenar más del doble de habitantes que tiene España.

- e) Considere el caso de $4 * 512$ Bytes. Teniendo en cuenta que un entero ocupa 2 Bytes y que hay que almacenar 2 enteros correspondientes a la clave y la dirección (se ignora que el número de direcciones en un nodo de un árbol B es una más que de claves) y que se quiere que la altura sea mínima, entonces el número total de claves a almacenar será aproximadamente igual $4 * 512 / 4$ que son 512 con lo que la altura mínima será $\log_{512}(100.000.000) \approx 3$.

Para el caso de $2 * 512$ Bytes y teniendo en cuenta las consideraciones anteriores, la altura máxima vendrá dada por $\log_{512}(100.000.000) \approx 3$.

Tenga en cuenta este último resultado, para observar la importancia que tiene el árbol B cuando realmente se implementa en un disco, ya que el número real de accesos a disco para la realización de una operación de búsqueda es sólo de tres.

PROBLEMAS PROPUESTOS

- 15.1.** 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 tiene números consecutivos y en el rango de las unidades de 1.000. (así a Madrid le corresponde del 1 al 1.000, a Toledo del 1.001 al 2.00).

Escribir un programa que permita gestionar la información indicada, formando una estructura en memoria de árbol B con un máximo de 6 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.

- 15.2.** En el problema 15.1 cuando se termina la ejecución se pierde toda la información. Modificar el programa 1 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 él la estructura de árbol B. La estructura puede experimentar modificaciones: nuevos centros, eliminación de alguno existente, por lo que al terminar la ejecución debe de escribirse de nuevo el árbol en el archivo.

- 15.3.** Se quiere dar más contenido a la información tratada en el problema 15.1. 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.

- 15.4.** Dada la siguiente secuencia de claves: 7, 25, 27, 15, 23, 19, 14, 29, 10, 50, 18, 22, 46, 17, 70, 33 y 58; dibuje árbol B+ de orden 5 cuya raíz es R, que se corresponde con dichas claves. (*Solución se encuentra en la página web del libro*).

- 15.5.** Comparar experimentalmente el rendimiento de los algoritmos de inserción y borrado en árboles binarios, árboles equilibrados AVL y árboles B que almacenen en cada nodo un máximo de dos claves.

- 15.6.** Dado un árbol B de orden $m = 5$ que contiene un total de $k = 4000000$, calcular el número máximo de compara-

ciones necesarias para decidir si una clave se encuentra en el árbol.

- 15.7.** Un árbol B+ es un caso particular de árbol B que permite un recorrido secuencial rápido. En un árbol B+ todas las claves se encuentran en nodos hoja, duplicando las claves que se encuentran en el nodo raíz y en los nodos interiores. Por consiguiente, la principal característica de un árbol B+ es que todas las claves de búsqueda se encuentran en nodos hoja. La operación de insertar una nueva clave sigue los pasos de la misma operación en un árbol B. Cuando el nodo está lleno se divide en dos, el primero se queda con $m/2$ y el segundo con $1+m/2$ y asciende un duplicado de la clave mediana. En cuanto a la operación de borrado de una clave, hay que tener en cuenta que se borran las claves de los nodos hoja, no de los nodos interiores, sirven para determinar el camino de búsqueda. Escribir las modificaciones en las funciones correspondientes a la operación de inserción y borrado para implementar este tipo de árboles.

- 15.8.** Una empresa de servicios tiene cuatro departamentos. Cada empleado de la empresa está adscrito a uno de ellos. Se ha realizado una redistribución del personal entre los distintos departamentos. El archivo EMPRESA contiene en cada registro los campos Número-Idt, Origen, Destino. El campo Origen toma los valores 1, 2, 3, 4 dependiendo del departamento inicial al que pertenece el empleado. El campo Destino toma los mismos valores, dependiendo del nuevo departamento asignado al empleado. Escribir un programa que almacene los registros del archivo EMPRESA en cuatro árboles B de orden m (dato ≤ 6) uno por cada departamento origen y realice el intercambio de registros en los árboles.

- 15.9.** Escribir una función no recursiva para insertar un elemento en un árbol B.

- 15.10.** En el árbol B+ del problema 15.4 eliminar las claves: 46, 50, 58 y dibujar el árbol resultante. (*Solución se encuentra en la página web del libro*).

- 15.11.** Se desea colocar cada uno de los elementos almacenados en una cola C dentro de una lista doblemente enlazada D, que implementa una cola de prioridad (Lista doblemente enlazada ordenada por el campo prioridad y dentro de la misma prioridad, su orden de aparición es el de la cola inicial). Para conocer las prioridades de las claves almacenadas en la cola C, se ha de consultar un Árbol B, ya creado, que contiene todas las claves enteras de la

cola C. La prioridad de una clave entera de la cola se obtiene de la siguiente forma: a la altura total del árbol B se le resta la altura del nodo en el que se encuentra en el árbol B. Es decir las claves de la raíz tiene prioridad cero, las de sus hijos inmediatos tienen prioridad 1. etc. Cada nodo de la cola tiene los campos: información (cadena de 10) y clave (entero).

El árbol B es de orden 5 (almacena 4 claves enteras). Los nodos de la lista doblemente enlazada tendrán los mismos campos que la cola.

Se pide:

- Declarar todos los tipos de datos necesarios para gestionar las tres estructuras de datos.
- Implementar el TAD Cola.
- Implementar las primitivas que necesite para tratar la lista doblemente enlazada de acuerdo con las especificaciones;

no puede tener ni nodo cabecera ni final, y no es circular.

- Escribir una función que calcule la prioridad de una clave entera que se le pase como parámetro, usando el árbol B. (suponga que todas las claves están en el árbol B).
- Escribir una definición que ponga en la cola de prioridad dada por la lista enlazada D los elementos de la cola C.

15.12. Implementar las operaciones para el manejo de un árbol B y un programa principal que las utilice en memoria externa. Con esta implementación los nodos del árbol B se deben almacenar en un archivo; ahora, con esta representación, los punteros deben ser posiciones relativas en el archivo. (*Solución se encuentra en la página web del libro*).

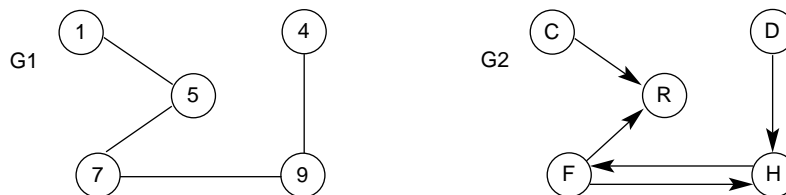
Grafos I: representación y operaciones

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. También se estudian operaciones importantes y algoritmos de grafos que son significativos en informática.

16.1. Conceptos y definiciones

Un **grafo** se define como un conjunto de vértices o nodos y un conjunto de arcos. Se dice que hay un arco (v_1, v_2) que va del vértice v_1 al vértice v_2 , cuando el nodo v_1 apunta a otro nodo v_2 . Un grafo se dice que es no dirigido si se cumple la condición de que si (v_1, v_2) es un arco del grafo, entonces también lo es el arco (v_2, v_1) . Para representarlos gráficamente se dibuja un conjunto de círculos que simbolizan los vértices y un conjunto de segmentos entre ellos que representan los arcos. Si se trata de un arco dirigido los segmentos serán flechas. Un grafo valorado es aquel que tiene asociado un factor de peso, es decir cuyas aristas o arcos están ponderadas.

EJEMPLO 16.1. Definir los grafos G_1 y G_2 .



G_1 es un grafo no dirigido $G_1 = \{V_1, A_1\}$, formado por el conjunto de vértices $V_1 = \{1, 4, 5, 7, 9\}$, y el conjunto de arcos $A_1 = \{(1, 4), (5, 1), (7, 9), (7, 5), (4, 9), (4, 1), (1, 5), (9, 7), (5, 7), (9, 4)\}$. El grafo dirigido $G_2 = \{V_2, A_2\}$, está formado por el conjunto de vértices $V_2 = \{C, D, E, F, H\}$, y el conjunto de arcos $A_2 = \{(C, E), (E, D), (F, E), (F, H), (H, F), (D, H)\}$.

En un grafo no dirigido si la arista (v_1, v_2) pertenece al conjunto de las aristas del grafo se dice que los vértices v_1 y v_2 son adyacentes. En un grafo dirigido si el arco (v_1, v_2) pertenece al conjunto de vértices del grafo se dice que v_1 es adyacente con v_2 y v_2 es adyacente desde v_1 .

Grado de un vértice es el número de arcos incidentes a dicho vértice. En los grafos dirigidos se considera grado de entrada, número de arcos que llegan a un determinado vértice y grado de salida como el número de arcos que salen de él.

Un camino entre dos vértices v_i, v_j es una sucesión ordenada de vértices del grafo que cumplen las condiciones siguientes: el primer vértice es v_i ; el último vértice es v_j ; cualquier par de vértices consecutivos v_p, v_q cumplen la condición de que (v_p, v_q) es un arco del grafo. Un camino simple entre dos vértices v_i, v_j es un camino en el que todos los vértices son distintos excepto posiblemente el primero y el último. Longitud del camino simple es el número de arcos que lo forman. Si el grafo es valorado se define la longitud como la suma de los valores de los arcos que lo forman.

Un vértice v_j se dice que es alcanzable desde un vértice v_i , si existe un camino desde v_i a v_j . También se dice que v_j es descendiente del vértice v_i . Un vértice v_j se dice que es un antecesor del vértice v_i si v_i es descendiente de v_j . Un ciclo es un camino simple en el que coinciden los vértices primero y último.

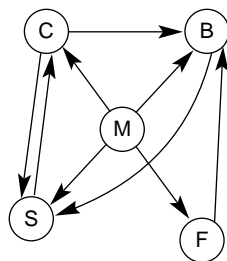
Una cadena simple entre dos vértices v_i, v_j es una sucesión ordenada de vértices del grafo que cumplen las condiciones siguientes: el primer vértice es v_i ; el último vértice es v_j ; cualquier par de vértices consecutivos v_p, v_q cumplen la condición de que o bien (v_p, v_q) o bien (v_q, v_p) es un arco del grafo y todos los vértices son distintos excepto posiblemente el primero y el último.

Un *grafo dirigido*, se dice que es conexo si para cada par de vértices del grafo existe una cadena simple que los une. Un grafo dirigido, se dice que es fuertemente conexo si para cada par de vértices del grafo hay un camino que los une. Un grafo G_1 es un subgrafo de G si el conjunto de vértices de G_1 está contenido en el conjunto de vértices de G y el conjunto de aristas de G_1 también está contenido en el conjunto de aristas de G .

16.2. Representación de los grafos

Hay que considerar que el grafo lo constituyen un cierto número de vértices y un cierto número de arcos; por tanto podrá ser representado utilizando una matriz, denominada **matriz de adyacencia**, donde las filas y columnas son los vértices del grafo y los elementos indican si entre un determinado par de vértices existe arco que los una. En los grafos no dirigidos la matriz de adyacencia es simétrica. Los grafos con factor de peso, grafos valorados, pueden representarse de tal forma que si existe arco, el elemento $A[i, j]$ es el factor de peso, la no existencia de arco supone que $A[i, j]$ es 0 ó ∞ dependiendo de su uso. A esta matriz se la denomina matriz valorada.

EJEMPLO 16.2. Mostrar la matriz de adyacencia correspondiente al grafo de la figura.



$X = S$

Array de
Vértices V

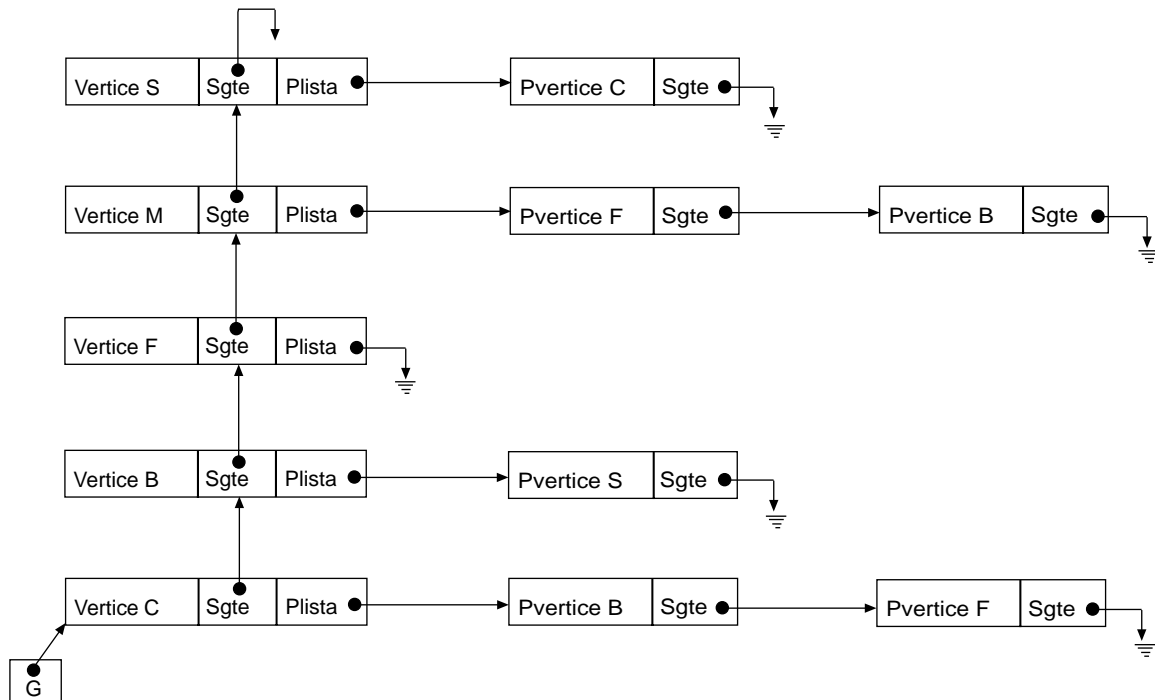
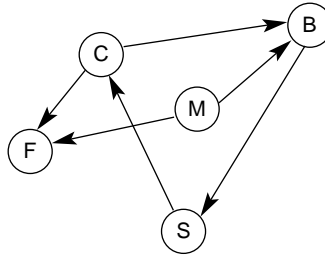
	1	2	3	4	5
C	B	M	S	F	

Matriz de
Adyacencia
A

	1	2	3	4	5
0	1	1	1	0	
0	0	0	1	0	
0	1	0	1	1	
1	0	0	0	0	
0	1	0	0	0	

Cuando el grafo tiene pocos arcos, y por tanto la matriz de adyacencia tiene muchos ceros (*matriz sparse*) resulta poco eficiente la matriz de adyacencia ya que el espacio que ocupa es el mismo que si el grafo tuviera muchos arcos. En estos casos se representa el grafo mediante listas enlazadas que se denominan listas de adyacencia. Una **lista de adyacencia** es una estructura multienlazada formada por una lista directorio, cuyos nodos representan los vértices del grafo, y, además, de cada uno de estos nodos, puede emerger otra lista enlazada cuyos nodos, a su vez, representan los arcos cuyo vértice origen es el del nodo de la lista directorio. Cuando se trate de un grafo valorado también almacena el peso.

EJEMPLO 16.3. *Mostrar la lista de adyacencia correspondiente al grafo de la figura.*



16.3. Tipo Abstracto de Datos Grafo

La especificación del TAD grafo no dirigido se puede encontrar en la página Web del libro.

16.4. Recorrido de un grafo

Hay dos formas de recorrer un grafo: recorrido en anchura y recorrido en profundidad. El **recorrido en anchura** visita el vértice de partida, para a continuación visitar todos los adyacentes que no estuvieran ya visitados y así sucesivamente. Utiliza una cola como estructura auxiliar donde colocar los vértices adyacentes hasta que les corresponda ser procesados y una lista con todos los vértices del grafo ya visitados. Una codificación del recorrido en anchura de un grafo puede consultarse en el ejercicio resuelto 16.3.

EJEMPLO 16.4. *El recorrido en anchura de grafo de la figura 16.1 a partir del vértice M: F,C,B,S,J.*

El **recorrido en profundidad** empieza visitando el vértice de partida y a continuación pasa a visitar en profundidad cada vértice adyacente no visitado. Puede realizarse de forma recursiva o bien iterativa eliminando la recursividad. Una codificación recursiva del recorrido en profundidad puede consultarse en el ejercicio resuelto 16.4, y una codificación iterativa del recorrido en el ejercicio resuelto 16.5.

EJEMPLO 16.5. El recorrido en profundidad de grafo de la figura 16.1 a partir del vértice M : F, S, J, C, B.

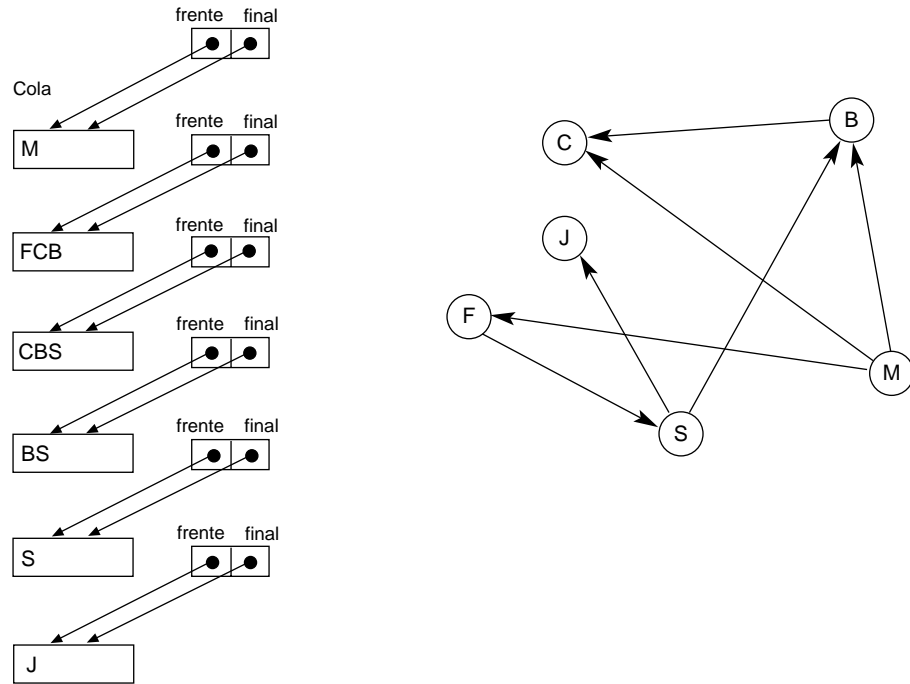


Figura 16.1. Grafo y seguimiento de su recorrido en anchura.

16.5. Componentes conexas

La determinación de si un grafo **no dirigido** es o no conexo, se hace realizando un recorrido en anchura o bien en profundidad del grafo. El grafo es conexo si el conjunto de vértices visitados coincide con todos los vértices del grafo (Figura 16.2)

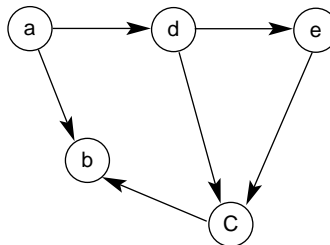


Figura 16.2. Grafo conexo.

Para el caso de grafos **dirigidos** la determinación sobre si un grafo G es conexo pasa por definir un grafo G' que contiene los mismos vértices que G pero todos sus arcos han pasado a ser aristas.

16.6. Componentes fuertemente conexas

Un grafo dirigido fuertemente conexo es aquel que cumple la condición de que para todo par de vértices existe un camino que los une. Un algoritmo que determina las componentes fuertemente conexas de un grafo es el siguiente:

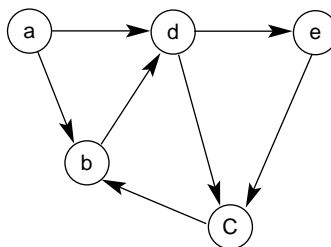
- Obtener el conjunto de descendientes de un vértice de partida v , $D(v)$, incluido el propio vértice v . (Para obtener $D(v)$ basta con hacer un recorrido en anchura o profundidad del grafo a partir de v).
- Obtener el conjunto de ascendientes de v , $A(v)$, incluido el propio vértice v . (Para obtener $A(v)$ basta con cambiar el sentido de todos los arcos del grafo y hacer un recorrido en anchura o profundidad del grafo).

La componente fuertemente conexa que contiene al vértice v es $D(v) \cap A(v)$. El grafo será fuertemente conexo, si la componente obtenida coincide con los vértices del grafo.

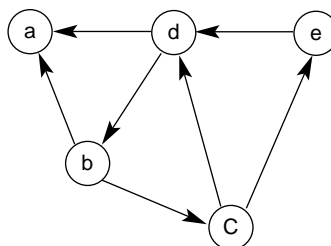
EJEMPLO 16.6. Determinar si el grafo de la figura 16.1 es fuertemente conexo.

El grafo de la figura 16.1 no es fuertemente conexo y sus componentes fuertemente conexas son: $\{C, B, S\}$ $\{F\}$ $\{M\}$

EJEMPLO 16.7. Mostrar la sistemática seguida para determinar las componentes fuertemente conexas en el siguiente grafo.



Al recorrerlo en profundidad a partir del vértice d , el conjunto de vértices que se alcanza es $\{d, c, b, e\}$ y repitiendo el recorrido en el grafo inverso (representado a continuación) se obtienen los vértices ascendientes: $\{d, b, c, e\}$. Los vértices comunes (en este ejemplo d, b, c, e) forman una componente fuertemente conexa.



PROBLEMAS RESUELTOS BÁSICOS

16.1. Implementación de las operaciones primitivas para el manejo de un grafo mediante matriz de adyacencia.

Análisis

Se implementa el TAD Grafo estableciendo su estructura un conjunto de vértices y un conjunto de arcos. La implementación de los conjunto de vértices y arcos se realiza con un *array* lógico (los valores son `True` o `False`) para los vértices y una matriz Boolean para los arcos. Se supone que los vértices toman valores comprendidos entre 0 y $n-1$, siendo n una constante previamente definida. Todas las funciones se implementan comprobando que los vértices implicados están en el grafo. La función `AnadeArco` añade los vértices del arco previamente. La función `BorraVértice`, además de borrar el vértice, borra todos los arcos del grafo que incidan en él.

Codificación

```
#include <stdio.h>
#define <stdlib.h>
#define n 10
#define True 1
#define False 0
struct Grafo
{
    int V[n], A[n][n];
};
struct Arco
{
    int u,v;
};

void Inicializa (Grafo *G)
{ /* no tiene ni vértices ni arcos */
    int i,j;
    for(i = 0; i < n; i++)
    {
        G->V[i] = False;
        for(j = 0; j < n; j++)
            G->A[i][j] = False;
    }
}

void AnadeVertice(Grafo *G, int v)
{
    /*Añade el vértice v al grafo */
    if ((0 <= v) && (v < n))
        G->V[v] = True;
}

void AnadeArco( Grafo *G, Arco Arc)
{
    /* Añade un arco al grafo, para lo cual debe añadir por si no están primeramente los dos
    vértices del arco*/
    AnadeVertice(G, Arc.u);
    AnadeVertice(G,Arc.v);
}
```

```

        G->A[Arc.u][Arc.v] = True;
    }

void BorraArco(Grafo *G, Arco Arc)
{
    /* elimina el arco del grafo */
    G->A[Arc.u][Arc.v] = False;
}

void BorraVertice( Grafo *G, int v)
{
    /* elimina el vértice v y todos los arcos que lo tengan de origen o destino*/
    int i;
    if((0 <= v) && (v < n))
    {
        G->V[v] = False;
        for ( i = 0; i < n; i++)
        {
            G->A[i][v] = False;
            G->A[v][i] = False;
        }
    }
}

int PerteneceVertice( Grafo G, int v)
{
    /* Decide si un vértice está en el grafo
    if(( 0 <= v) && (v < n))
        return (G.V[v]) ;
    else
        return(False);
}

int PerteneceArco( Grafo G, Arco Arc)
{
    /* decide si un arco está en el grafo */
    int sw;
    sw = PerteneceVertice(G,Arc.u)&& PerteneceVertice(G,Arc.v);
    if(sw)
        return (G.A[Arc.u][Arc.v]);
    else
        return(False);
}

```

16.2. Implementar las operaciones primitivas para el manejo de un grafo mediante lista de adyacencia.

Análisis

Se implementa el TAD Grafo estableciendo su estructura multienlazada de la siguiente forma. Una lista enlazada general *ListaG* mantiene la información de todos los vértices del grafo. Cada nodo de esta lista contiene un puntero *sig* al siguiente nodo de la lista y un campo *el* que es un registro que contiene un campo *v* donde almacena la información del vértice y un puntero *Ady* (lista de adyacencia) que apunta a una lista enlazada donde se almacenan los distintos arcos adyacentes al vértice. Cada lista de adyacencia se implementa como una lista enlazada, cuyos nodos contienen los campos *sig* que apun-

ta al siguiente nodo, y el campo `el` que a su vez es un registro que contiene el campo `v` para alacénar el otro vértice del arco y el campo `valor` que contiene el valor del arco.

Las siguientes funciones se encargan de manejar la lista de adyacencia:

- La función `anadeConjuntoAdy` añade un nodo de la lista de adyacencia de un vértice.
- La función `PerteneceConjuntoAdy` decide si un vértice se encuentra en la lista de adyacencia de otro vértice.
- La función `BorraConjuntoAdy` borra un nodo de la lista de adyacencia si se encuentra en ella.
- La función `BorraListaAdy` elimina todos los nodos de una lista de adyacencia.
- La función `EncuentraPosAdy` encuentra dos punteros, uno al nodo que contiene un vértice y otro al nodo inmediatamente anterior.

Las funciones de tratamiento general del grafo son:

- `Inicializa` crea el grafo vacío poniendo la lista general a `NULL`.
- `AnadeVertice` añade un vértice a la lista general de vértices.
- `AnadeArco` añade un arco, para ello se asegura de que los vértices que lo componen estén en la lista general añadiéndolos previamente.
- `BorraArco` borra un arco eliminando el vértice destino de la lista de adyacencia que emerge del vértice fuente.
- `BorraVertice` borra un vértice de la lista general así como todos los arcos que llegan o emergen de él.
- `PertenceVertice` decide si un vértice está en el grafo comprobando su pertenencia a la lista general.
- `PertenceArco` decide si un arco está en el grafo.

Codificación

```
#include <stdlib.h>
#define False 0
#define True 1
struct Arco
{
    int u,v;
    float valor;
};
struct ItemAdy
{
    int v;
    float valor;
};
struct ListaAdy
{
    ItemAdy el;
    struct ListaAdy* sig;
};
struct ItemG
{
    int v;
    ListaAdy *Ady;
};
struct ListaG
{
    ItemG el ;
    ListaG *sig;
};
```

```
void Inicializa(ListaG **Primero)
{
    (*Primero)=NULL;
}

int PerteneceConjuntoAdy (ListaAdy* Primero, ItemAdy dato)
{
    ListaAdy *ptr;
    for (ptr = Primero; ptr != NULL; ptr = ptr ->sig )
        if (ptr-> el.v == dato.v)
            return True;
    return False;
}

void AnadeConjuntoAdy(ListaAdy** Primero, ItemAdy dato)
{
    /* añade vértice a la lista de adyacencia */
    ListaAdy *nuevo ;
    if(!PerteneceConjuntoAdy(*Primero,dato))
    {
        nuevo = (ListaAdy*)malloc(sizeof(ListaAdy));
        nuevo -> el = dato;
        nuevo -> sig = *Primero;
        *Primero= nuevo;
    }
}

int PertenceVertice(ListaG* Primero, ItemG dato)
{
    ListaG *ptr;
    for (ptr = Primero; ptr != NULL; ptr = ptr ->sig )
        if (ptr-> el.v == dato.v)
            return True;
    return False;
}

void AnadeVertice(ListaG** Primero, ItemG dato)
{
    ListaG *nuevo ;
    if(!PertenceVertice(*Primero,dato))
    {
        nuevo = (ListaG*)malloc(sizeof(ListaG));
        nuevo -> el = dato;
        nuevo -> sig = *Primero;
        *Primero= nuevo;
    }
}

void EncuentraPosAdy(ListaAdy *Primero, ItemAdy dato, ListaAdy **Ant,
                    ListaAdy **Pos)
/* en Ant y Pos retorna la dirección del nodo anterior y la posición donde se encuentra.
   Si no se encontrara, retorna en Pos el valor de NULL*/
{

```

```

ListaAdy* ptr, *ant;
int enc = False;
ptr =Primero;
ant = NULL;
while (( !enc) && (ptr != NULL))
{
    enc = (ptr->el.v == dato.v);
    if (!enc)
    {
        ant = ptr;
        ptr = ptr -> sig;
    }
}
*Ant = ant;
*Pos = ptr;
}

void BorraConjuntoAdy (ListaAdy** Primero, ItemAdy dato)
{
    ListaAdy* ptr, *ant;
    EncuentraPosAdy(*Primero, dato, &ant, &ptr);
    if (ptr != NULL)
    {
        if (ptr == *Primero)
            *Primero = ptr->sig;
        else
            ant -> sig = ptr->sig;
        free(ptr);
    }
}

void EncuentraPosGrafo(ListaG *Primero, ItemG dato, ListaG **Ant, ListaG **Pos)
/* en Ant y Pos retorna la dirección del nodo anterior y la posición donde se encuentra. Si no se
   encontrara, retorna en Pos el valor de NULL */
{
    ListaG* ptr, *ant;
    int enc = False;
    ptr = Primero;
    ant = NULL;
    while (( ! enc) && (ptr != NULL))
    {
        enc = (ptr->el.v == dato.v);
        if (!enc)
        {
            ant = ptr;
            ptr = ptr -> sig;
        }
    }
    *Ant = ant;
    *Pos = ptr;
}

void BorraVertice (ListaG** Primero, ItemG dato)

```

```

{
    ListaG* ptr, *ant;
    ItemAdy dato1;
    EncuentraPosGrafo(*Primero, dato, &ant,&ptr);
    if (ptr != NULL)
    {
        if (ptr == *Primero)
            *Primero = ptr->sig;
        else
            ant -> sig = ptr->sig;
        BorraListaAdy(&(ptr->el.Ady));           /* borra toda la deadyacencia*/
        free(ptr);
        ptr = *Primero;
        dato1.v = dato.v;
        while (ptr)                             /*borra todos los arcos que llegan al vértice v*/
        {
            BorraConjuntoAdy(&(ptr->el.Ady), dato1);
            ptr = ptr->sig;
        }
    }
}

void BorraListaAdy(ListaAdy ** Primero)
{
    ListaAdy *l, *l1;
    l = *Primero;
    while(l != NULL)
    {
        l1 = l;
        l = l->sig;
        free(l1);
    }
}

void AnadeArco(ListaG **Primero,Arco arc)
{
    ItemG dato;
    ItemAdy dato1;
    ListaG *Ant,*Pos;
    dato.v = arc.u;
    dato.Ady = NULL;
    AnadeVertice( Primero, dato);
    dato.v=arc.v;
    AnadeVertice( Primero, dato);
    dato.v = arc.u;
    EncuentraPosGrafo(*Primero, dato,&Ant,&Pos);
    dato1.v = arc.v;
    dato1.valor = arc.valor;
    AnadeConjuntoAdy(&(Pos->el.Ady), dato1);
}

void BorraArco(ListaG **Primero,Arco arc)
{

```

```

    ItemG dato;
    ItemAdy dato1;
    ListaG *Ant,*Pos;
    dato.v = arc.u;
    dato.Ady = NULL;
    EncuentraPosGrafo(*Primero, dato,&Ant,&Pos);
    dato1.v = arc.v;
    dato1.valor=arc.valor;
    BorraConjuntoAdy(&(Pos->el.Ady), dato1);
}

int PertenceArco(ListaG *Primero, Arco arc)
{
    ItemG dato;
    ItemAdy dato1;
    ListaG *Ant,*Pos;
    ListaAdy *Ant1, *Pos1;
    dato.v = arc.u;
    dato.Ady=NULL;
    EncuentraPosGrafo(Primero, dato,&Ant,&Pos);
    dato1.v = arc.v;
    dato1.valor = arc.valor;
    if (Pos != NULL)
    {
        EncuentraPosAdy(Pos->el.Ady,dato1, &Ant1,&Pos1);
        return (Pos1 != NULL);
    }
    return(False);
}

```

16.3. *Codificar una función realice el recorrido en anchura del grafo G partir del vértice v.*

Análisis

La función `RecorridoEnAnchura` se codifica de acuerdo con la teoría y usa las primitivas de gestión de colas implementadas con un registro con dos punteros `Frente` y `Final` a una lista enlazada, y las declaraciones y algunas funciones definidas en el Ejercicio 16.2. La base del algoritmo es llevar un conjunto de `Visitados`, y una cola para el recorrido. Unicialmente se añade a la cola el vértice `v`, y al conjunto de `visitados`. Posteriormente se implementa el siguiente algoritmo general:

```

Mientras la cola no esté vacía hacer
  Sacar un vértice u de la cola
  desde cada vártice v adyacentes a u hacer
    si el vértice v no está en el conjunto de visitados entonces
      añadirlo al conjunto de visitados y a la cola
    fin si
  fin desde
fin mientras

```

Codificación

```

void RecorridoEnAnchura(ListaG *G, int v, Cola *C)
{

```

```

Cola C1;
ItemG dato;
ListaAdy *Ady;
ListaG *Visitados, *Ant,*Pos;
VacíaC(C);
VacíaC(&C1);
AnadeC(&C1,v);
AnadeC(C,v);
Inicializa(&Visitados);
dato.v = v;
AnadeVertice(&Visitados,dato);
while (!EsVacíaC(C1))
{
    dato.v = PrimeroC(C1);
    BorrarC(&C1);
    EncuentraPosGrafo(G,dato,&Ant,&Pos);
    if(Pos != NULL)
    {
        Ady = Pos->el.Ady;
        while (Ady != NULL)
        {
            dato.v = Ady->el.v;
            if(!PerteneceVertice(Visitados,dato))
            {
                AnadeC(&C1,dato.v);
                AnadeC(C,dato.v);
                AnadeVertice(&Visitados,dato);
            }
            Ady = Ady->sig;
        }
    }
}
}

```

16.4. Codificar una función recursiva que realice el recorrido en profundidad del grafo G a partir del vértice v .

Análisis

El recorrido en profundidad se implementa utilizando la técnica recursiva, ya que la propia definición se adapta íntegramente a la definición recursiva del recorrido. En este caso la lista de vértices `visitados` y la cola `C` son variables globales y se inicializa “fuera” de la función `RecorridoEnProfundidadRec`. Se usan las declaraciones y algunas funciones del Ejercicio 16.2.

Codificación

```

ListaG *Visitados; Cola *C;

void RecorridoEnProfundidadRec(ListaG *G,int v)
{
    ItemG dato;
    ListaAdy *Ady;
    ListaG *Ant,*Pos;
    AnadeC(C,v);

```

```

    dato.v = v;
    AnadeVertice(&Visitados,dato);
    EncuentraPosGrafo(G,dato,&Ant,&Pos);
    if(Pos != NULL)
    {
        Ady = Pos->el.Ady;
        while (Ady != NULL)
        {
            dato.v = Ady->el.v;
            if(!PerteneceVertice(Visitados,dato))
                RecorridoEnProfundidadRec(G,dato.v);
            Ady = Ady->sig;
        }
    }
}

```

16.5. Codificar una función iterativa que realice el recorrido en profundidad del grafo G a partir del vértice v .

Análisis

La función `RecorridoEnProfundidad` usa las primitivas de gestión de colas, pilas y las declaraciones y algunas funciones definidas en el Ejercicio 16.2. La codificación es parecida a la del recorrido en anchura, excepto que para simular la recursividad del recorrido en profundidad se usa una pila P (implementada con una lista enlazada). Se introducen los vértices en la lista de visitados así como a la cola C después de salir de la pila P . Para que el recorrido en profundidad del Ejercicio 16.4 y el que se codifica en este ejercicio sean el mismo se usa una pila auxiliar $P1$ cuyo único objetivo es que los vértices de la lista de Adyacencia Ady aparezcan en la pila P en el mismo orden en el que son tratados en el recorrido en profundidad recursivo.

Codificación

```

void RecorridoEnProfundidad(ListaG *G,int v, Cola *C)
{
    Pila *P,*P1;
    ItemG dato;
    ListaAdy *Ady;
    ListaG *Visitados, *Ant, *Pos;
    VaciaC(C);
    VaciaP(&P);
    AnadeP(&P,v);
    Inicializa(&Visitados);
    while (!EsVacíaP(P))
    {
        dato.v = PrimeroP(P);
        BorrarP(&P);
        AnadeC(C,dato.v);
        AnadeVertice(&Visitados,dato);
        EncuentraPosGrafo(G,dato,&Ant,&Pos);
        if(Pos != NULL)
        {
            Ady = Pos->el.Ady;
            while (Ady != NULL)
            {

```

```

        dato.v = Ady->el.v;
        VacíaP(&P1);
        if(!PerteneceVertice(Visitados,dato))
            AnadeP(&P1,dato.v);
        Ady = Ady->sig;
    }
    while(!EsVacíaP(P1))
    {
        dato.v = PrimeroP(P1);
        BorrarP(&P1);
        AnadeP(&P,dato.v);
    }
}
}
}

```

16.6. Escribir un programa que compruebe si un grafo leído del teclado tiene circuitos mediante el siguiente algoritmo:

- 1) Obtener los sucesores de todos los nodos.
- 2) Buscar un nodo sin sucesores y tachar ese nodo donde aparezca.
- 3) Se continúa este proceso hasta que sea posible.
- 4) Si todos los nodos pueden tacharse el grafo no tiene circuito.

Análisis

Se leen los arcos de la entrada hasta que se lea un arco con valores $(-1, -1)$. Se implementa el grafo con un *array* de listas de adyacencias ordenadas crecientemente, que es otra posible forma de implementar un grafo. Se usan las funciones *InsertarArco* que inserta un arco en el grafo, y *BorrarTodos* que borra un vértice de todas las listas de adyacencias del grafo.

Codificación

Sólo se escribe la función *main()* y las dos funciones mas interesantes: *InsertarArco()* y *BorrarTodos()*.

```

void main(void)
{
    /* implementa el algoritmo definido en el ejercicio*/
    int Visita[n], PuedoSeguir, i; Arco a;
    for (i = 0; i < n; i++)
    {
        Lg[i] = NULL;
        Visita[i] = False;
    }
    while (LeeArco(&a.x,&a.y))
        InsertarArco(&(Lg[a.x]), a.y);
    PuedoSeguir = True;
    while (PuedoSeguir)
    {
        i = 0;
        PuedoSeguir = False;
        while ((i < n) && (!PuedoSeguir))
            if((Lg[i] == NULL)&& (!Visita[i]))
            {
                Visita[i] = True;
            }
        }
    }
}

```



```

        BorrarTodos(i);
        PuedoSeguir = True;
    }
    else
        i ++;
}
if (! EsVacioGrafo)
    printf(" no hay circuitos");
else
{
    printf(" hay circuito ");
    for (i = 0; i < n; i++)
        Visita[i] = False;
    I = 0;
    while (Lg[i] == NULL)
        i++;
    do
    {
        printf(" %d",i);
        Visita[i] = True;
        i = Lg[i]->Adyacente;
    }
    while (! Visita[i]);
    printf("%d",i);
}
}

void InsertarArco( ListaAdy **Inicio, int Ver)
{
    /* inserta en la lista de adyacencia ordenada dada por inicio el nuevo nodo Ver */
    int Encontrado;
    ListaAdy *Nuevo, *p, *q;
    Nuevo = (ListaAdy*)malloc(sizeof(ListaAdy));
    Nuevo -> Adyacente = Ver;
    p = *Inicio;
    Encontrado = False;
    while ((p != NULL) && (! Encontrado))
        if (p->Adyacente > Ver)
            Encontrado = True;
        else
        {
            q = p;
            p = p->Sig;
        }
    if (p == *Inicio)
    { /* se inserta al comienzo de la lista. Puede que se vacía*/
        Nuevo->Sig = *Inicio;
        *Inicio = Nuevo;
    }
    else
    { /* Se inserta en el centro o al final de la lista*/
        Nuevo->Sig = p;

```

```

        q->Sig = Nuevo;
    }
}
void BorrarTodos(int v)
{
    /* Elimina el vértice v de la lista de sucesores de todos los nodos */
    ListaAdy *p, *L;
    int i, Encontrado;
    for (i = 0; i < n; i++)
        if (i != v)
        {
            L = Lg[i];
            if (L != NULL)
            {
                Encontrado= False;
                while ((L != NULL) && (! Encontrado))
                {
                    if (L->Adyacente == v)
                        Encontrado = True;
                    else
                    {
                        p = L;
                        L = L->Sig;
                    }
                }
                if (Encontrado)
                { /* eliminar vértice*/
                    if (L == Lg[i])
                        Lg[i] = L->Sig;                /* se elimina el primero de la lista*/
                    else
                        p->Sig = L->Sig;                /* Se elimina en el centro o al final*/
                    free(L);
                }
            }
        }
}
}

```

- 16.7.** Escribir un módulo de programa que de un camino mínimo si existe para ir de v a $v1$ en un grafo no valorado. El camino mínimo de un grafo no valorado es cualquier camino cuyo número de arcos para ir de v a $v1$ sea mínimo.

Análisis

Para encontrar el camino mínimo para ir del vértice v al $v1$, basta con aplicar un recorrido en anchura desde el vértice v y parar cuando se visite $v1$. Si no se consigue visitar el vértice $v1$ no hay camino. Una codificación puede verse en el Ejercicio 16.3. El bucle `while (!EsVacíaC(C1))` debe cambiarse de la siguiente forma: `while (!EsVacíaC(C1)&&!Fin)`, tal y como expresa la codificación siguiente:

Codificación

```

Fin = False;
while (!EsVacíaC(C1) && !Fin)
{
    if(!PerteneceVertice(Visitados,dato))
    {
        AnadeC(&C1,dato.v);
    }
}

```

```

    AnadeC(C,dato.v);
    AnadeVertice(&Visitados,dato);
    if(dato.v == v1)
        fin = True;
    }
}

```

16.8. *Escribir un algoritmo para determinar las componentes conexas de un grafo G no dirigido.*

Análisis

Para calcular las componentes conexas puede usarse entre otro el siguiente algoritmo:

1. Realizar un recorrido del grafo a partir de cualquier vértice w (se usa en la implementación un recorrido en anchura). Los vértices visitados son guardados en el conjunto `Conj2`.
2. Si el conjunto `Conj2` es el conjunto de todos los vértices del grafo, entonces el grafo es conexo, y sólo tiene una componente conexa.
3. Si el grafo no es conexo, `Conj2` es una componente conexa.
4. Se toma un vértice cualquiera no visitado, v , y se realiza de nuevo el recorrido del grafo a partir de v . Los vértices visitados en `Conj2` forman otra componente conexa.
5. El algoritmo termina cuando todos los vértices del grafo han sido visitados.

Se parte de un grafo G implementado mediante listas de adyacencia, según se realiza en el Ejercicio 16.2. Se usa el recorrido en anchura implementando en el Ejercicio 16.3. Se usa además una implementación de una Cola y de un Conjunto. La función `PasaVerticesConjunto` pone todos los vértices del grafo en un Conjunto `ConjG`. La función `Conexa` encuentra y escribe una componenete conexa a partir de un vértice v . La función `Componetesconexas`, itera hasta que consigue visitar todos los vértices del grafo. Se usa además la función `PasaVerticesConjunto` que añade los vértices del grafo al conjunto `Conj`.

Codificación

```

void PasaVerticesConjunto(ListaG *G, Conjunto *Conj)
{
    int v;
    ListaG *G1;
    VaciaConj(Conj);
    G1 = G;
    while(G1 != NULL)
    {
        v = G1->el.v;
        AnadeConj(v,Conj);
    }
}

```

```

void Conexa(ListaG *G, int v, Conjunto *Conj)
{
    Cola C ;
    int u;
    RecorridoEnAnchura(G,v,&C);
    VaciaConj(Conj);
    printf("Componente conexa\n");
    while (! EsVacíaC(C))
    {
        u = PrimeroC(C);

```

```

        BorrarC(&C);
        printf(" %d", u);
        AnadeConj(u, Conj);
    }
}

void ComponentesConexas(ListaG *G)
{
    Conjunto ConjG, Conj1, Conj2;
    int v;
    PasaVerticesConjunto(G, &ConjG);
    VaciaConj(&Conj1);
    while (DistintoConj(Conj1, ConjG))
    {
        v = 0;
        while (PerteneceConj(v, Conj1))
            v++;
        Conexa(G, v, &Conj2);
        UnionConj(Conj1, Conj2, &Conj1);
    }
}

```

16.9. Escribir un algoritmo para encontrar las componentes fuertemente conexas de un grafo dirigido.

Análisis

Para calcular las componentes fuertemente conexas de un grafo se puede usar entre otros el siguiente algoritmo:

1. Obtener el conjunto de descendientes (sucesores) de un vértice de partida v , que se denomina $D(v)$.
2. Obtener el conjunto de ascendientes (predecesores) de v , que se denomina $A(v)$.
3. Los vértices comunes que tiene $D(v)$ y $A(v)$ es el conjunto de vértices de la componente fuertemente conexas a la que pertenece el vértice v .
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 los pasos 1, 2, y 3 hasta obtener todas las componentes fuertes del grafo.

En la implementación del algoritmo en el paso 1 se realiza un recorrido en anchura del grafo G a partir del vértice v .

Para el paso 2 hay que proceder en primer lugar a construir otro grafo dirigido $G_{invertido}$ que sea el resultante de invertir las direcciones (sentidos) de todos los arcos de G , y a continuación proceder como en el paso 1. Se codifican las funciones `InvierteGrafo`, `FuertementeConexa`, y `ComponentesFuertes` que resuelven el problema. Se usa el Ejercicio 16.8 y una constante entera n tal que los vértices del grafo varían entre 0 y $n-1$.

Codificación

```

void InvierteGrafo(ListaG *G, ListaG **Ginvertido)
{
    ItemG dato;
    Arco arc;
    ListaG *Gaux, *GauxInvertido; ListaAdy *Ady;
    Gaux = G;
    GauxInvertido = NULL;
    while (Gaux != NULL)
    {

```

```

    dato.v = Gaux->el.v;
    dato.Ady = NULL;
    AnadeVertice(&GauxInvertido,dato);
    Gaux = Gaux->sig;
}
Gaux = G;
while(Gaux != NULL)
{
    Ady = (Gaux->el.Ady);
    arc.v = Gaux->el.v;
    while(Ady != NULL)
    {
        arc.u = Ady->el.v;
        arc.valor = Ady->el.valor;
        AnadeArco(&GauxInvertido,arc);
        Ady = Ady->sig;
    }
}
*Ginvertido = GauxInvertido;
}

void FuertementeConexa(ListaG *G, ListaG *Ginvertido, int v, Conjunto *Conj)
{
    Conjunto Conj1, Conj2;
    int i;
    Conexa(G,v,&Conj1);
    Conexa(Ginvertido,v,&Conj2);
    InterseccionConj(Conj1,Conj2,Conj);
    for(i = 0; i < n; i++)
        if(PerteneceConj(i,*Conj) )
            printf("%d",i);
}

void ComponentesFuentes(ListaG *G)
{
    Conjunto ConjG, Conj1, Conj2;
    ListaG *Ginvertido;
    int v;
    PasaVerticesConjunto(G,&ConjG);
    VacíaConj(&Conj1);
    InvierteGrafo(G,&Ginvertido);
    while (DistintoConj(Conj1,ConjG))
    {
        v = 0;
        while(PerteneceConj(v,Conj1))
            v++;
        FuertementeConexa(G,Ginvertido,v,&Conj2);
        UnionConj(Conj1,Conj2,&Conj1);
    }
}

```

16.10. Escribir una función que reciba como parámetro la matriz de adyacencia de un grafo y devuelva el número de caminos de longitud k que hay entre cualquier par de vértices.

Análisis

Para resolver el problema basta con calcular la matriz A^k , ya que esta matriz como se sabe (véase la teoría) contiene el resultado pedido, siempre y cuando la matriz A de adyacencia tenga como valores 1 si están conectados los vértices correspondientes con un arco y cero si no lo están. La función `ProductoK_esimo` devuelve en A^k la matriz que contiene el número de caminos de longitud k entre cualquier par de vértices.

Codificación (Se encuentra en la página Web)

16.11. Escribir una función que calcule la matriz de caminos de un grafo G .

Análisis

La matriz de adyacencia es de tipo lógico y los productos y las sumas son productos y sumas lógicos, es decir `and` y `or`. La matriz de caminos es la que obtiene la función `suma`.

Codificación (Se encuentra en la página Web del libro)

PROBLEMAS RESUELTOS AVANZADOS

16.12. Durante el recorrido en profundidad de un grafo dirigido, cuando se recorren ciertos arcos, se llega a vértices aún sin visitar. Los arcos que llevan a vértices nuevos se conocen como arcos de árbol y forman un bosque abarcador en profundidad para el grafo dirigido dado. Además de los arcos del árbol existen otros tipos de arcos diferentes que se llaman arcos de retroceso, arcos de avance y arcos cruzados:

- Un arco se dice que es de retroceso si va de un nodo del árbol a otro que es su predecesor.
- Un arco se dice que es de avance si va de un nodo del árbol a otro nodo del árbol ya construido pero que es un descendiente de él.
- Un arco se dice que es cruzado si va de un nodo del árbol a otro que no está relacionado por la relación jerárquica definida en el árbol.

Escribir un programa que lea un grafo y calcule el bosque abarcador, y los arcos de avance, retroceso y cruzado.

Análisis

Para distinguir entre cada uno de estos arcos basta con numerar los nodos del árbol según se va realizando el recorrido en profundidad. Si se codifica recursivamente entonces si `numero[v]=cont` basta con hacer `cont++` cada vez que se realiza una llamada recursiva. A esto se llama numeración en profundidad. Los arcos de avance van de vértices de baja numeración a vértices de alta numeración, siempre que estén en el mismo árbol. Los arcos de retroceso van de los vértices de alta numeración a vértices de baja numeración siempre que estén dentro del mismo árbol. Los arcos cruzados relacionan vértices que no están relacionados por la relación de jerarquía definida por los árboles del bosque abarcador. Por lo tanto:

- Un arco (i, j) es de retroceso, si i es sucesor de j en el bosque abarcador, y además `numero[i] > numero[j]` y el arco (i, j) no está en el bosque abarcador.
- Un arco (i, j) es de avance, si j es sucesor de i en el bosque abarcador, y además (i, j) no es arco del bosque abarcador.
- Son arcos cruzados aquellos que son arcos del grafo y sus vértices no son ni antecesores ni descendientes entre sí en el bosque abarcador.

Para construir el árbol abarcado a partir de un vértice *v*, basta con hacer un recorrido en profundidad almacenando el resultado en un cola *C* y numerando los vértices *Numerop* de acuerdo con el valor de *cont*, para posteriormente, ir sacando los vértices de la cola y creando los arcos del árbol en la matriz *A1*. Este trabajo lo realiza la función *ArbolAbarcador*. La función *main*, se encarga de construir el bosque abarcador llamando a la función anterior, iterativamente hasta que se marquen todos los vértices. Los vértices marcados se almacenan en un Conjunto de *Visitados* cuyas funciones de tratamiento se suponen ya implementadas, así como todas las primitivas para manejar pilas y colas.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#define False 0
#define True 1
#define n 10
#define Max -100

void LeeGrafo(float A[n][n]); /* Lee el grafo, se deja como ejercicio */
void EscribeMatriz(float A[n][n]); /* Lee el grafo, se deja como ejercicio */
void RecorridoEnProfundidad(int v)
{
    int j;
    /*Realiza un recorrido en profundidad del a partir del vertice v*/
    AnadeC(&C, v);
    AnadeConj(v,&Visitados);
    Cont ++;
    Numerop[v] = Cont;
    for (j = 0; j < n; j++)
        if (A[v][j] != Max)
            if (!PerteneceConj(j,Visitados))
                RecorridoEnProfundidad(j);
}

void ArbolAbarcador(int v, float A1[n][n])
{
    int e1, e, e2;
    Cola C;
    Pila *P;
    VaciaC(&C);
    Cont = 0;
    RecorridoEnProfundidad(v);
    e1 = PrimeroC(C);
    BorrarC(&C);
    VaciaP(&P);
    AnadeP(&P, e1);
    while (!EsVaciaC(C))
    {
        e = PrimeroC(C);
        if (A[e1][e] != Max)
        {
            AnadeP(&P, e);
            A1[e1][e] = A[e1][e];
            e1 = e;
        }
    }
}
```

```

        else
        {
            do
            {
                BorrarP(&P);
                e2 = PrimeroP(P);
            }
            while ( A[e2][e] == Max);
            AnadeP(&P, e);
            A1[e2][e] = A[e2][e];
            e1 = e;
        };
        BorrarC(&C);
    }
}

int UesSucesordeV(int u,int v,float B[n][n])
{
    /*nos dice si u esta entre los sucesores de v, usando la matriz B*/
    int z[n],i, j, Exito;
    Pila *P;
    Exito = False;
    VacíaP(&P);
    for(i = 0; i < ; i++)
        z[i] = False;
    i = v;
    AnadeP(&P, i);
    z[i] = True;
    while ((! EsVacíaP(P)) && ( ! Exito))
    {
        i = PrimeroP(P);
        BorrarP(&P);
        for ( j = 0; j < n; j++)
            if ((B[i][j] != Max)&& (! z[j]))
                if (u == j)
                    Exito = True;
                else
                {
                    AnadeP(&P, j);
                    z[j] = True;
                }
    }
    return (Exito);
}

void ArcosDeRetroceso(float A[n][n],float A1[n][n],float A2[n][n])
{
    /*un arco (i,j) es de retroceso , si i es sucesor de j en el bosque abarcador, y
    además numero[i] > número[j]*/
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {

```



```

        A2[i][j] = Max;
        if ((A[i][j] != Max) && (A1[i][j]==Max) && (Numerop[i] > Numerop[j])
            && ( UesSucesordeV(i,j,A1)))
            A2[i][j] = A[i][j];
    }
}

void ArcosDeAvance( float A[n][n],float A1[n][n],float A2[n][n])
{
    /*un arco (i,j) es de avance , si j es sucesor de i en el bosque abarcador,
    y además (i,j) no es arco del bosque abarcador*/
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            A2[i][j] = Max;
            if ((A[i][j] != Max)&&(A1[i][j] == Max) && ( UesSucesordeV(j,i,A1)))
                A2[i][j] = A[i][j];
        }
}

void ArcosCruzados(float A[n][n],float A1[n][n], float A2[n][n])
{
    /*Son arcos cruzados aquellos que son arcos del grafo y no son ni antecesoros ni descendientes
    entre sí en el bosque abarcador*/
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            A2[i][j] = Max;
            if ((A[i][j] != Max) && (A1[i][j] == Max) && ( !UesSucesordeV(j,i,A1))
                && (!UesSucesordeV(i,j,A1)))
                A2[i][j] = A[i][j];
        }
}

void main (void)
{
    int i, j;
    Conjunto Universo, visitados;
    LeeGrafo(A);
    printf(" matriz de distancias \n"); EscribeMatriz(A);
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            A1[i][j] = Max;
    VacíaConj(&Visitados);
    VacíaConj(&Universo);
    for (i = 0; i < n; i++)
        AnadeConj(i,&Universo);
    while (DistintoConj(Visitados,Universo))
    {
        i= 0;
        while (PerteneceConj(i,Visitados))

```

```

        i++;
        ArbolAbarcador(i, A1);
    }
    printf(" matriz de bosque abarcador\n");
    EscribeMatriz(A1);
    ArcosDeRetroceso(A, A1, A2);
    printf("  matriz de arcos de retroceso \n");
    EscribeMatriz(A2);
    ArcosDeAvance(A, A1, A2);
    printf(" matriz de arcos de avance \n");
    EscribeMatriz(A2);
    ArcosCruzados(A, A1, A2);
    printf("  matriz de arcos cruzados \n");
    EscribeMatriz(A2);
}

```

16.13. *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 e imprimir todos sus caminos hamiltonianos, si los hay.*

Análisis

Cada grafo se da en forma de lista de sus arcos, y cada arco viene indicado por una pareja de valores (nodo inicial, nodo final). Los nodos se designan por números enteros de rango 0..n. El final de datos del grafo se determina mediante la pareja (-1, -1). El programa hamiltoniano funciona a base a un subprograma recursivo *Ensayar* que, a partir del camino recorrido hasta ese momento:

- Determina si es Hamiltoniano, y lo escribe.
- Intenta prolongarlo, añadiendo un vértice, de todas las maneras posibles, y con cada una de ellas se llama recursivamente a sí mismo para completarlo si es posible.

Se utiliza el tipo abstracto de datos *Conjunto* como estructura auxiliar en la que se almacenan los nodos del grafo. Las operaciones de entrada del grafo: *LeeArco()* y *LeerDatos()* se dejan como ejercicio. Se usan las variables globales *A[n][n]*, como matriz de adyacencia, *Camino[n]* que almacena el camino en construcción y *Encontrado* que decide si existe algún camino Hamiltoniano.

Codificación

```

#define False 0
#define True 1
#define n 10
int A[n][n], Camino[n], Encontrado;

int LeeArco (int *i, int* j);
void LeerDatos(Conjunto *Nodos);

void Ensayar(Conjunto Nodos, int k)
{
    int i, Ult;
    if (EsVacioConj(Nodos)) /* antes de empezar como terminar*/
    {
        printf("Un camino hamiltoniano de este grafo es:\n");
        for (i = 0; i <= k; i++)
            printf("%3d", Camino[i]);
    }
}

```

```

        printf("\n"); Encontrado = True;
    }
    else
    {
        /*prolongación del camino si se puede*/
        Ult = Camino[k];
        k ++;
        for (i = 0; i < n; i++)
            if (PerteneceConj(i, Nodos ) && A[Ult][i])
                { /* es aceptable */
                    Camino[k] = i;
                    BorraConj(i,&Nodos);
                    Ensayar(Nodos, k);
                    AnadeConj(i,&Nodos);
                }
    }
}

void main (void)
{
    int j;
    Conjunto Nodos;
    LeerDatos(&Nodos);
    Encontrado = False;
    for (j = 0; j < n; j++)
        if (PerteneceConj(j,Nodos))
        {
            Camino[0] = j;
            BorraConj(j,&Nodos);
            Ensayar(Nodos, 0);
            AnadeConj(j,&Nodos);
        }
    if (! Encontrado)
        printf(" no hay caminos hamiltonianos \n");
}

```

PROBLEMAS PROPUESTOS

- 16.1.** 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:
- Dibujar el grafo correspondiente.
 - Representar el grafo mediante listas de adyacencia.

$$M = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

- 16.2.** 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.
- Escribir un programa que represente el grafo en listas de adyacencia.
 - Realizar un recorrido en anchura desde un vértice dado.
- 16.3.** Se quiere formar un grafo de manera aleatoria con los siguientes requisitos: consta de 10 vértices que son

números enteros de 11 a 99. Dos vértices x y y están relacionados si $x+y$ es múltiplo de 3.

- Escribir un programa para representar el grafo descrito mediante una matriz de adyacencia
- Determinar la matriz de caminos utilizando las potencias de la matriz de adyacencia.

16.4. 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.

- Escribir un programa que represente el grafo descrito mediante listas de adyacencia.
- ¿Tiene fuentes y sumideros?

16.5. Dado el grafo descrito en el problema 16.3. Escribir 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.

16.6. Dado un grafo dirigido en el que los vértices son números enteros positivos y el par (x,y) es un arco si $x-y$ es múltiplo de 3.

- Escribir un programa para representar el grafo mediante listas de adyacencia de tal forma que cada lista sea circular.
- Una vez que el grafo esté en memoria determinar el grado de entrada y el grado de salida de cada nodo.

16.7. Un grafo, en el que los vértices son regiones y los arcos tienen 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 necesarios para que partiendo de la representación mediante listas se obtenga la representación mediante la matriz de pesos.

16.8. Representar la siguiente información sobre un Centro de enseñanza:

- Nombre del centro, ubicación, nombre del Director.
- Alumnos divididos en clases.
- Enseñanza Primaria: n grupos de un máximo de 25 alumnos.
- Enseñanza Secundaria: m grupos de un máximo de 30 alumnos.
- Bachillerato: b grupos de un máximo de 40 alumnos.
- Profesores que pueden ser:
 - De Enseñanza Primaria se asigna un profesor a cada clase.
 - De Enseñanza Secundaria y Bachillerato, hay un profesor por cada asignatura del curso.
- Asignaturas: En cada curso hay un máximo de max asignaturas.

Se pide :

1. Lectura de los alumnos de cada uno de los grupos.
2. Lectura del profesorado de cada uno de los grupos.
3. Lectura de las notas de una clase.
4. Clasificación del alumnado de dicha clase en alumnos aprobados, suspensos y de muy bajo rendimiento (3 asignaturas suspensas o más).
5. Ordenación alfabética de una clase.
6. Ordenación alfabética del alumnado del colegio.

Nota: Evitar la presencia de información redundante. Utilice estructuras de datos dinámicas.

16.9. Escribir un algoritmo que determine el mínimo número de aristas que se debe eliminar en un grafo no dirigido para que el grafo resultante sea acíclico.

16.10. Escribir un programa que encuentre en un grafo no dirigido y conexo un camino que vaya a través de todas las aristas exactamente una vez en cada dirección.

16.11. Un grafo no dirigido conexo tiene la propiedad de ser biconexo si no hay ningún vértice que al suprimirlo del grafo haga que este se convierta en no conexo. Escribir una función que decida si un grafo es biconexo.

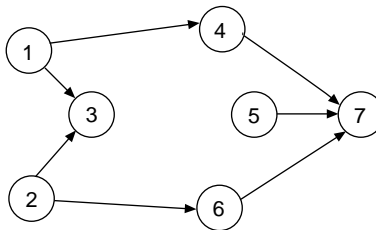
Grafos II: Algoritmos

La resolución de problemas que se pueden modelar mediante grafos 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. 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; también se estudia uno de los problemas típicos de flujos, el *flujo máximo*. Algunos algoritmos notables han sido desarrollados por grandes investigadores y en su honor se conocen por sus nombres: algoritmos de Dijkstra, Warshall, Prim, Kruscal, Ford-Fulkerson.

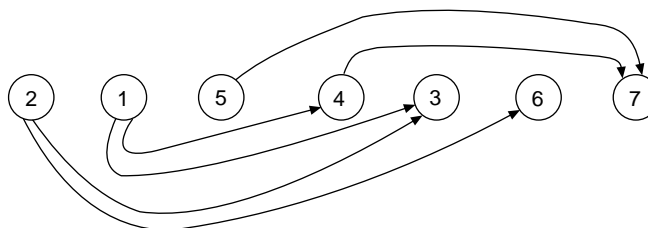
17.1. Ordenación topológica

Un grafo G dirigido y sin ciclos se denomina *gda* o *grafo acíclico* (también conjunto parcialmente ordenado). Una ordenación topológica T de G es una ordenación lineal de los nodos de G que preserve la ordenación parcial. Si hay un camino del vértice u al vértice v , entonces u va delante de v en la ordenación lineal ($u < v$). Se puede demostrar que cualquier grafo dirigido finito y sin ciclos, tiene una ordenación topológica. Una codificación en C del algoritmo aparece en el ejercicio 17.15.

EJEMPLO 17.1. *Mostrar la ordenación topológica del grafo de la figura.*



La ordenación topológica es:



17.2. Matriz de caminos: Algoritmo de Warshall

Warshall propone un algoritmo eficiente de programación dinámica para calcular la matriz de caminos (también llamado cierre transitivo). Para todo $k = 0, 1, 2, \dots, n$ se define la matriz P_k de la siguiente forma: $P_k[i, j] = \text{true}$ si hay camino del vértice i al j que use a lo sumo como vértices intermedios el $1, 2, \dots, k$; $P_k[i, j] = \text{false}$ en otro caso.

De esta forma $P_0[i, j] = A[i, j]$, donde A es la matriz de adyacencia del grafo.

Warshall hace la siguiente observación: para que $P_k[i, j] = \text{true}$ debe ocurrir uno de estos dos casos:

1. Ya existe un camino simple del vértice i al j que usa los vértices $1, 2, \dots, k-1$, en cuyo caso $P_{k-1}[i, j] = \text{true}$.
2. Hay un camino simple de i al k que usa los vértices $1, 2, \dots, k-1$ y otro camino simple de k al j que usa los vértices $1, 2, \dots, k-1$. Por lo tanto debe cumplirse $P_{k-1}[i, k] = \text{true}$ y $P_{k-1}[k, j] = \text{true}$. De esta forma la relación para encontrar los elementos de P_k se puede expresar: $P_k[i, j] = P_{k-1}[i, j]$ or $(P_{k-1}[i, k] \text{ and } P_{k-1}[k, j])$.

17.3. Problema de los caminos más cortos con un sólo origen: algoritmo de Dijkstra

Se parte de un grafo dirigido y valorado de forma positiva, por lo que cada arco del grafo (i, j) tiene asociado un coste $c_{ij} \geq 0$, de tal forma que el coste de un camino entre dos vértices viene dado por la suma de los costes de cada uno de los arcos que componen el camino. Lo que se pretende encontrar es el camino de coste mínimo que va del vértice i al vértice j . El algoritmo de Dijkstra, encuentra el camino de longitud mínima de un vértice origen al resto de los vértices del grafo. En este algoritmo voraz clásico donde los candidatos son los vértices del conjunto C y S es el conjunto de los vértices ya escogidos. Se denomina camino especial a un camino que parte del vértice origen y que tiene todos los vértices dentro de S excepto posiblemente el último. Se define D como el vector de distancias que mantiene la longitud del camino especial más corto desde el origen a cualquier vértice de G . A es la matriz de costes (pesos).

El algoritmo se puede plantear de la forma siguiente:

- Inicialmente C contiene todos los vértices excepto el origen (el 1) y S contendrá el vértice origen (el 1). Así $D[j] = A[1, j]$ para $j = 2 \dots n$.
- Sea i el vértice de C con $D[i]$ mínimo tal que i esté en C .
- Se elimina i de C y se pone en S .
- Para cada vértice j de C se actualiza $D[j]$ con el mínimo entre $D[j]$ y $D[i] + A[i, j]$.

Si el esquema anterior se repite $n-1$ veces y n es el número de vértices, se tiene en D la longitud de los caminos mínimos que partiendo del vértice 1 llega a cada uno de los restantes vértices. Para comprobar la afirmación anterior se observa que, si $D[j]$ era mínimo antes de añadir a C el vértice i , después de la actualización de $D[j]$, se tiene que $D[j]$ también debe ser mínimo, ya que si el camino especial mínimo pasa por el vértice i el valor mínimo será $D[i] + A[i, j]$, y si no pasa por el vértice i no hace falta actualizar $D[j]$. (Observe que la afirmación no es correcta si los arcos pudieran ser negativos). Para recuperar el camino de longitud mínima en el algoritmo de Dijkstra, basta con añadir al algoritmo un *array* auxiliar P de tal manera que $P[j]$ contiene siempre el predecesor inmediato del camino especial mínimo que va del vértice 1 al vértice j . Una codificación del algoritmo se encuentra en el Ejercicio 17.16.

17.4. Problema de los caminos más cortos entre todos los pares de vertices: algoritmo de Floyd

El problema se puede resolver por medio del algoritmo de Dijkstra, aplicándolo a cada uno de los vértices, pero hay otra alternativa más directa, que es el algoritmo de Floyd. Sea G un grafo dirigido valorado, $G = (V, A)$. Se supone que los vértices están numerados de 1 a n ; la matriz A es en este caso la matriz de pesos, de tal forma que todo arco (i, j) tiene asociado un peso $c_{ij} \geq 0$; si no existe arco (i, j) se supone que $c_{ij} = \infty$. Ahora se quiere encontrar la matriz D de $n \times n$ elementos tal que cada elemento $D_{i,j}$ contenga el coste mínimo de los caminos que van del vértice i al vértice j . El proceso que sigue el algoritmo de Floyd tiene los mismos pasos que el algoritmo de Warshall para encontrar la matriz

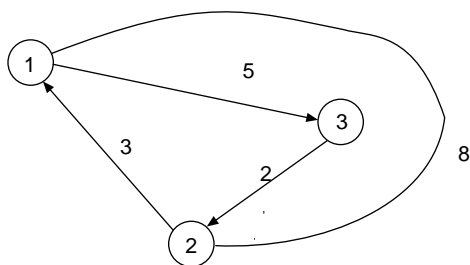
de caminos. Se generan iterativamente la secuencia de matrices $D_0, D_1, D_2, \dots, D_k, \dots, D_n$ cuyos elementos tienen el significado: $D_0[i, j] = C[i, j]$ coste (peso) del arco de i a j . Para todo $k = 1, 2, \dots, n$, $D_k[i, j]$ contiene la longitud de un camino mínimo para ir del vértice i al vértice j usando los vértices $1, 2, \dots, k$. Para calcular $D_k[i, j]$ basta con observar que el camino mínimo para ir del vértice i al vértice j o bien no usa el vértice k en cuyo caso.

$$D_k[i, j] = D_{k-1}[i, j] \text{ o bien lo usa en cuyo caso } D_k[i, j] = D_{k-1}[i, k] + D_{k-1}[k, j].$$

Es decir $D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$. La matriz D_n será la matriz de caminos mínimos del grafo.

Al igual que se hace en el algoritmo de Dijkstra por cada vértice se guardará el índice del último vértice que ha conseguido que el camino sea mínimo del i al j en caso de que el camino sea directo tiene un cero. Para ello se usa una matriz de vértices predecesores P . Una codificación del algoritmo se encuentra en el Ejercicio 17.17.

EJEMPLO 17.2. El ejemplo aplica el algoritmo de Floyd al siguiente grafo y muestra las matrices que se obtienen al considerar los diferentes vértices como auxiliares.



A_0	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

A_1	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

A_2	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

A_3	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

17.5. Concepto de flujo. Algoritmo de Ford Fulkerson

Se pueden modelar muchas situaciones mediante una red en la cual se considera que los arcos tienen una capacidad de limitar la cantidad de un producto que se puede enviar a través del arco. En estas situaciones, frecuentemente se desea transportar la máxima cantidad de flujo desde un punto de partida (llamado fuente s) hacia un punto final (llamado sumidero t). Tales problemas se llaman problemas de flujo máximo. Existen varios algoritmos especializados para resolver el problema del flujo máximo. El método que nosotros analizaremos es el de Ford-Fulkerson (1962). Sea V el conjunto de nodos del grafo. Sea s el nodo inicial del flujo y t el nodo sumidero. Sean i, j vértices de un grafo, $e(i, j)$ arcos del grafo. Si U_{ij} es la capacidad del arco (i, j) , se denominará flujo a una función F_{ij} definida en A (arcos) que verifica las propiedades:

$$1. F_{ij} \geq 0 \quad \forall (i, j) \in A \quad 2. \sum_i F_{ij} - \sum_i F_{ji} = 0 \quad i \in V, i \neq s, i \neq t \quad 3. F_{ij} \leq U_{ij} \quad \forall (i, j) \in A$$

La segunda condición impone la conservación de la cantidad total de flujo. Las condiciones primera y tercera imponen la cota superior e inferior de los valores del flujo. El problema del flujo máximo consiste en:

$$\text{maximizar } \sum_t F_{it} \text{ sujeto a las condiciones } 1. 2. 3.$$

ALGORITMO DEL AUMENTO DEL FLUJO: ALGORITMO DE FORD Y FULKERSON

La idea básica de este algoritmo es partir de una función de flujo cero, e iterativamente ir mejorando el flujo. La mejora se da en la medida que el flujo de s hasta t aumenta, teniendo en cuenta las condiciones que ha de cumplir la función de flujo que en lenguaje natural son: flujo que entra a un nodo ha de ser igual al flujo que sale; en todo momento el flujo no puede superar la capacidad del arco.

Los arcos de la red se clasifican en tres categorías:

- No modificables: arcos cuyo flujo no puede aumentarse ni disminuirse; por tener capacidad cero o tener un coste prohibitivo.

- Incrementables: arcos cuyo flujo puede aumentarse, transportan un flujo inferior a su capacidad.
- Reducibles: arcos cuyo flujo puede ser reducido.

Con estas categorías se pueden establecer las siguientes mejoras de la función de flujo desde s a t :

1ª. *Forma de mejora*: Encontrar un camino P del vértice fuente s al sumidero t tal que el flujo a través de cada arco del camino (todos los arcos incrementables) es menor que la capacidad:

$$F_{ij} < U_{i,j} \quad \forall (i,j) \in P \quad \text{Entonces el flujo se puede mejorar en las unidades: } \text{Minimo} \{(U_{i,j} - F_{i,j}), \quad \forall (i,j) \in P\}$$

2ª. *Forma de mejora*: Encontrar un camino P' del sumidero t a la fuente s formado por arcos reducibles, entonces es posible reducir el flujo de t a s y por tanto aumentar en las mismas unidades de flujo de s a t en la cantidad:

$$\text{Minimo} \{F_{i,j}, \quad \forall (i,j) \in P'\}$$

3ª. *Forma de mejora*: Existe una cadena $P1$ desde s a t con arcos incrementables, o reducibles en este caso el flujo puede ser incrementado en el mínimo de las dos cantidades siguientes:

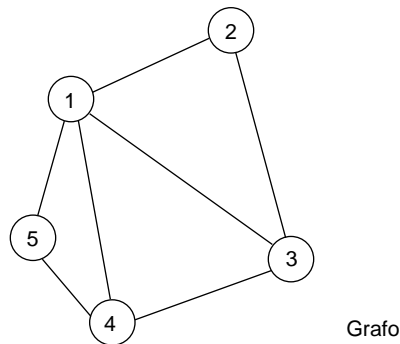
$$\text{Minimo} \{(U_{i,j} - F_{i,j}) \text{ de los arcos incrementables}\}$$

$$\text{Minimo} \{F_{i,j} \text{ de los arcos reducibles}\}$$

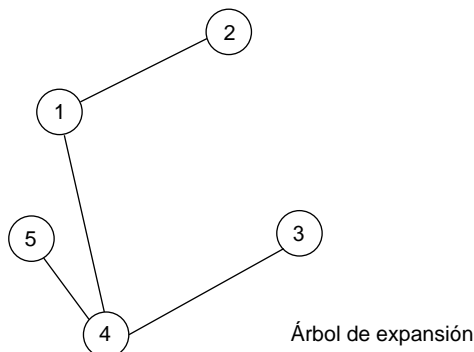
17.6. Problema del árbol de expansión de coste mínimo

Sea G un grafo no dirigido y tal que todos sus arcos sean positivos. Un árbol en una red es un subgrafo G' del grafo G que es conectado y sin ciclos. Los árboles tienen dos propiedades importantes: todo árbol de n vértices contiene exactamente $n-1$ arcos.; si se añade un arco a un árbol de expansión entonces resulta un ciclo. Un árbol de expansión de coste mínimo, es un árbol que contiene a todos los vértices de una red y tal que la suma de los pesos de sus arcos es mínima.

EJEMPLO 17.3. *Mostrar el árbol de expansión del Grafo G .*



Un árbol de expansión de un Grafo G es un subgrafo de G con su mismo número de nodos, que además es un árbol.



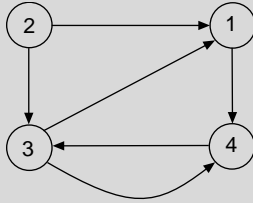
17.7. Algoritmo de Prim y algoritmo de Kruskal

El **algoritmo de Prim** encuentra el árbol de expansión de coste mínimo. Es un algoritmo clásico que se incluye dentro de la categoría de voraces. El punto de partida es un grafo $G = (V, A)$ modelo red y sea $c_{i,j} \geq 0$ el peso o coste asociado al arco (i, j) . Si se supone $V = \{1, 2, 3, \dots, n\}$ 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 en cada iteración otro vértice v de $V - W$ tal que si u es un vértice de W , el arco (u, v) es el mas corto de entre todos los arcos que itenen un vértice u en W y otro v en $V - W$. El proceso termina cuando $V = W$. Se observa que en todo momento el conjunto de nodos que forma W constituyen una componente conexa sin ciclos. Puede consultarse una codificación del algoritmo en el Ejercicio 17.19.

Algoritmo de Kruskal. De nuevo se tiene un grafo conexo no dirigido valorado $G = (V, A)$ y una función de coste definida $c_{i,j} \geq 0$ en los arcos de A . Kruskal propone otra estrategia para encontrar el árbol de expansión de coste mínimo. El algoritmo comienza con un grafo con los mismos vértices V pero sin arcos. Se puede decir que cada vértice es una componente conexa en sí mismo. Para construir componente conexas cada vez mayores, se examinan los arcos de A , en orden creciente del coste. Si el arco conecta dos vértices que se encuentran en dos componentes conexas distintos, entonces se añade el arco a T . Se descartarán los arcos si conectan dos vértices contenidos en el mismo componente, ya que pueden provocar un ciclo si se le añadiera al árbol de expansión para esa componente conexa. Cuando todos los vértices están en una misma componente ya se ha obtenido el árbol de expansión de coste mínimo del grafo G . Una codificación del algoritmo se encuentra en el ejercicio 17.20.

PROBLEMAS RESUELTOS BÁSICOS

17.1. Aplicar el algoritmo de Warshall al grafo de la figura adjunta:

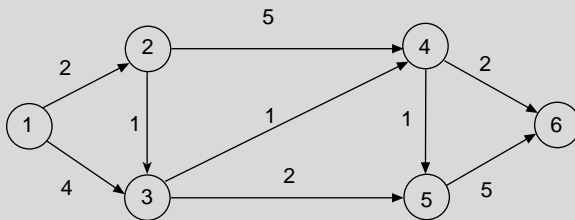


Solución

Si se realiza un seguimiento del algoritmo descrito en la teoría se obtiene en los sucesivos pasos las siguientes matrices. La solución de Warshall viene dada por la matriz del paso 4.

Inicialización:	Paso 1	Paso 2	Paso 3	Paso 4
$\begin{bmatrix} F & F & F & T \\ T & F & T & F \\ T & F & F & T \\ F & F & T & F \end{bmatrix}$	$\begin{bmatrix} F & F & F & T \\ T & F & T & T \\ T & F & F & T \\ F & F & T & F \end{bmatrix}$	$\begin{bmatrix} F & F & F & T \\ T & F & T & T \\ T & F & F & T \\ F & F & T & F \end{bmatrix}$	$\begin{bmatrix} F & F & F & T \\ T & F & T & T \\ T & F & F & T \\ T & F & T & T \end{bmatrix}$	$\begin{bmatrix} T & F & T & T \\ T & F & T & T \\ T & F & T & T \\ T & F & T & T \end{bmatrix}$

17.2. Aplicar el algoritmo de Dijkstra al siguiente grafo:



Solución

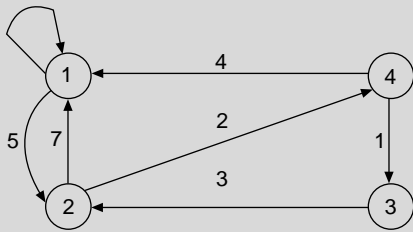
Al aplicar el algoritmo descrito en la teoría al grafo de la figura se tiene:
(La inicialización viene dada por el paso cero)

Paso	C	S	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
0	[2,3,4,5,6]	[1]	0	2	4	∞	∞	∞
1	[3,4,5,6]	[1,2]	0	2	3	6	∞	∞
2	[4,5,6]	[1,2,3]	0	2	3	4	5	∞
3	[5,6]	[1,2,3,4]	0	2	3	4	5	∞
4	[6]	[1,2,3,4,5]	0	2	3	4	5	6
5	[]	[1,2,3,4,5,6]	0	2	3	4	5	6

Al final del algoritmo el vector de predecesores P viene dado por:

P[1]	P[2]	P[3]	P[4]	P[5]	P[6]
0	1	2	3	3	4

17.3. Aplicar el algoritmo de Floyd al siguiente grafo.

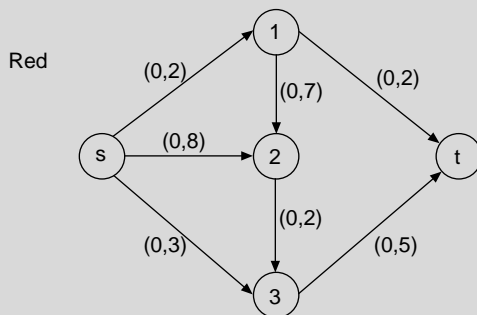


Solución

Si se aplica el algoritmo de Floyd descrito en la teoría la matriz D se va modificando en los sucesivos pasos tal y como se indica posteriormente. La matriz de predecesores final es la dada.

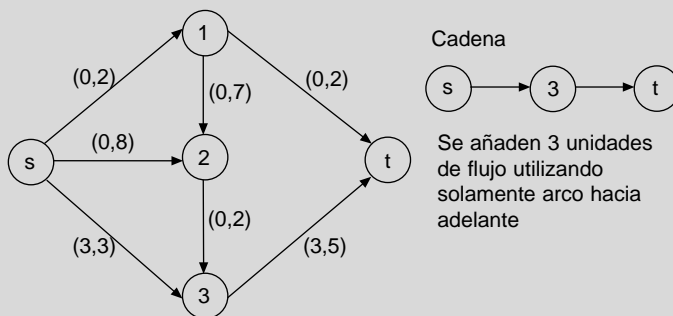
Inicialización:	Matriz D					Matriz final de predecesores P
	Paso 1	Paso 2	Paso 3	Paso 4		
7 5 ∞ ∞	7 5 ∞ ∞	7 5 ∞ 7	7 5 ∞ 7	7 5 8 7		0 0 4 2
7 ∞ ∞ 2	7 12 ∞ 2	7 12 ∞ 2	7 12 ∞ 2	7 6 3 2		0 4 4 0
∞ 3 ∞ ∞	∞ 3 ∞ ∞	10 3 ∞ 5	10 3 ∞ 5	9 3 6 5		4 0 4 2
4 ∞ 1 ∞	4 9 1 ∞	4 9 1 11	4 1 1 6	4 4 1 6		0 3 0 3

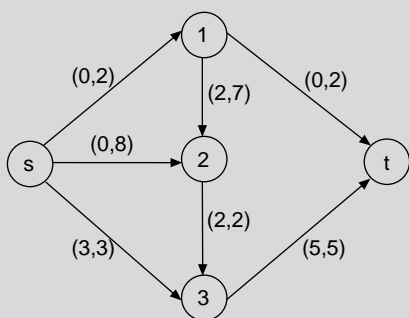
17.4. Aplicar el algoritmo de Ford_Fulkerson al siguiente grafo.



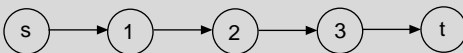
Solución

Aplicando el algoritmo al grafo, los sucesivos pasos son

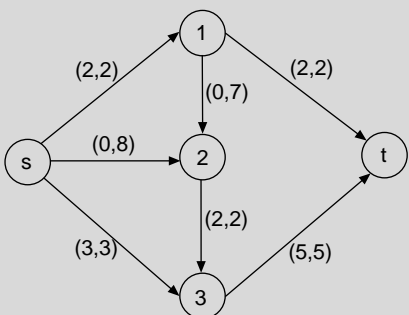




Cadena



Se añaden 2 unidades de flujo utilizando solamente arcos hacia delante.

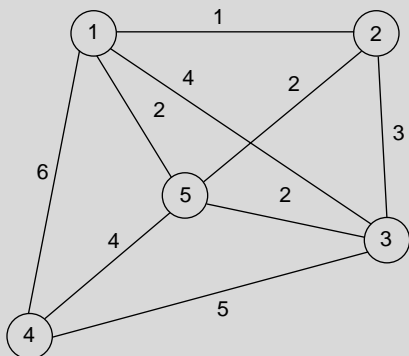


Cadena



Se añaden 2 unidades de flujo utilizando el arco hacia atrás (1,3).
Se obtiene un flujo máximo de 7.

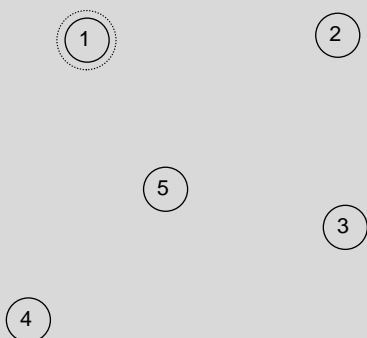
17.5. Aplique el algoritmo de Prim al siguiente grafo.



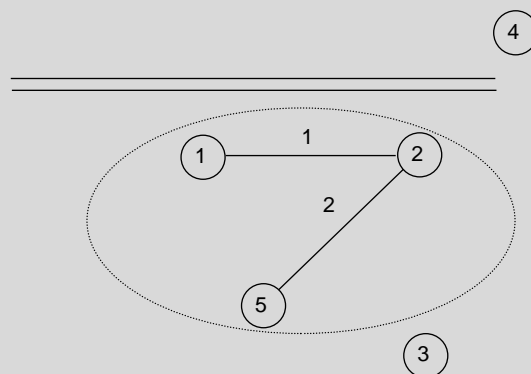
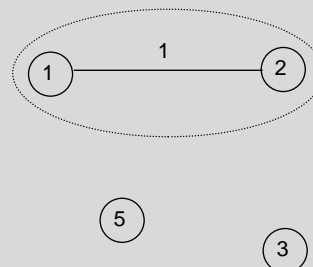
Solución

Los sucesivos pasos del algoritmo comenzando por el vértice 1 son:

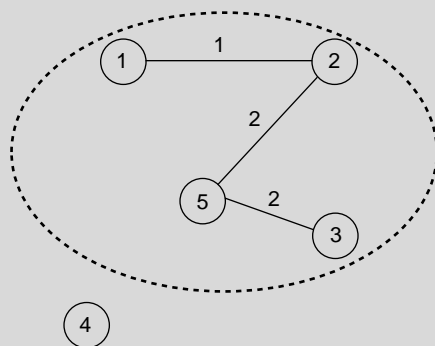
Paso 0



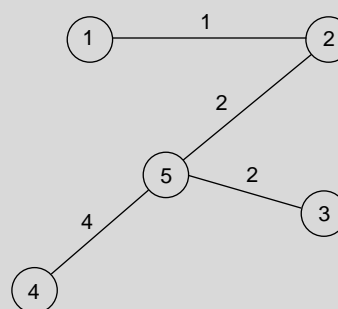
Paso 1



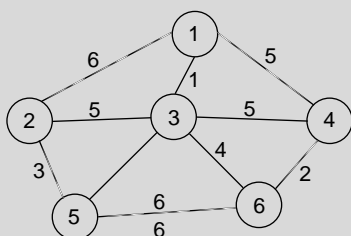
Paso 3



Paso 4

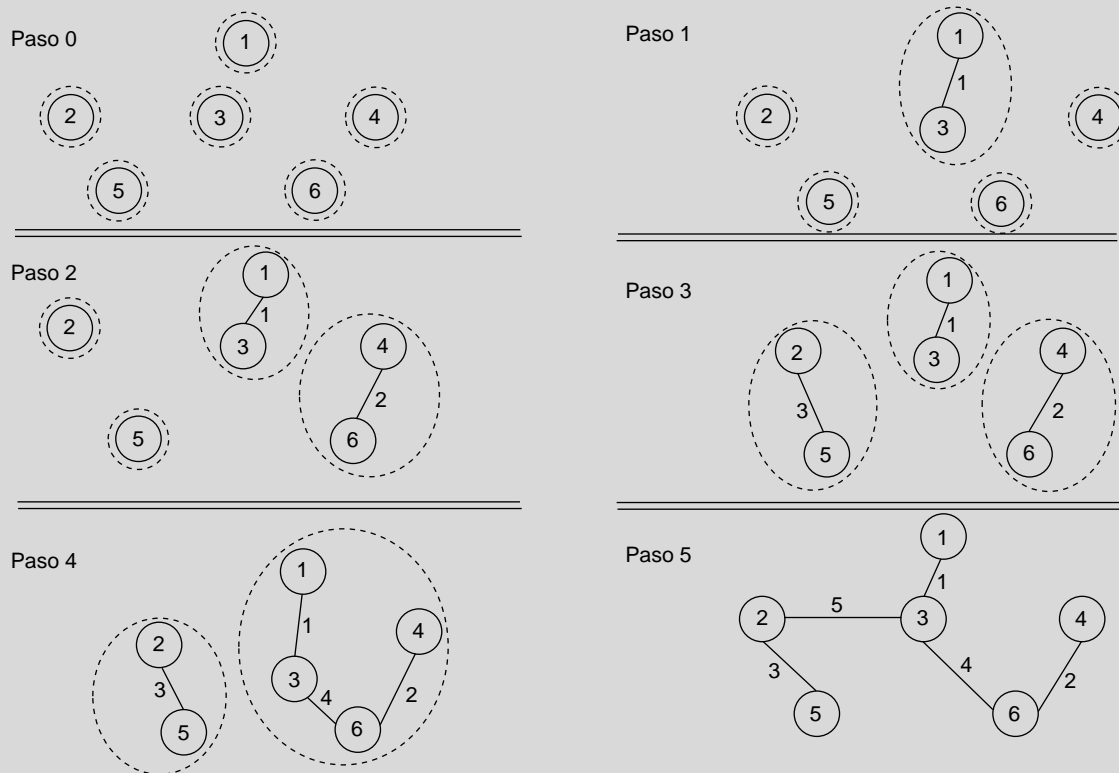


17.6. Aplicar el algoritmo de Kruskal al siguiente grafo:

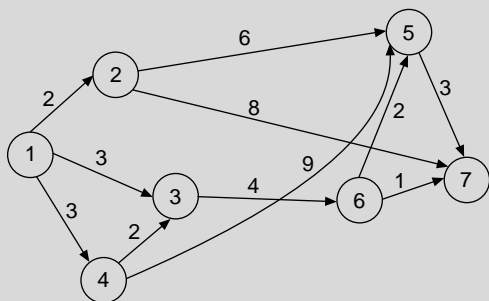


Solución

Al aplicar el algoritmo se originan los siguientes pasos



17.7. Dada la siguiente red: Se pide encontrar una ordenación topológica.

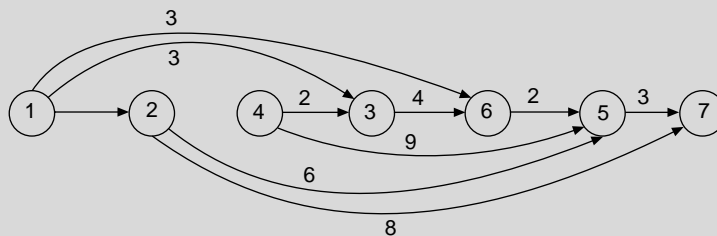


Solución

Aplicando el algoritmo de la ordenación topológica a la red:

	1	2	3	4	5	6	7	Cola	Salida
Paso	GRADOS DE ENTRADA								
0	0	1	2	1	3	1	3	1	
1	X	0	1	0	3	1	3	2,4	1
2	X	X	1	0	2	1	2	4	1,2
3	X	X	0	X	1	1	2	3	1,2,4
4	X	X	X	X	1	0	2	6	1,2,4,3
5	X	X	X	X	0	X	1	5	1,2,4,3,6
6	X	X	X	X	X	X	0	7	1,2,4,3,6,5
7	X	X	X	X	X	X	X	X	1,2,4,3,6,5,7

Una ordenación topológica es:



- 17.8.** En la red del Ejercicio 17.7 los arcos representan actividades y el factor de peso representa el tiempo necesario para realizar dicha actividad (un Pert). Cada vértice v de la red representa el tiempo que tardan todas las actividades representadas por los arcos que terminan en v . El ejercicio consiste en asignar a cada vértice v de la red 17.7 el tiempo necesario para que todas las actividades que terminan en v se puedan realizar; este lo denominamos $Tn(v)$.

Análisis

Una forma de hallarlo es la siguiente: asignar tiempo 0 a los vértices sin predecesores; si a todos los predecesores de un vértice v se les ha asignado tiempo, entonces $Tn(v)$ es el máximo, para cada predecesor, de la suma del tiempo del predecesor con el factor de peso del arco desde ese predecesor hasta v .

Solución

Aplicando el algoritmo indicado al grafo anterior se obtiene:

Nodos	1	2	3	4	5	6	7
Tn	0	2	5	3	12	9	15

- 17.9.** Tomando de nuevo la red del Ejercicio 17.7 y teniendo en cuenta el tiempo de cada vértice $Tn(v)$ calculado en el ejercicio 17.8, ahora se quiere calcular el tiempo límite en que todas las actividades que terminan en el vértice v pueden ser completadas sin atrasar la terminación de todas las actividades, a este tiempo lo llamamos $tl(v)$.

Análisis

Un algoritmo conocido para resolver el problema es el siguiente: asignar $tn(v)$ a todos los vértices v sin sucesores. Si todos los sucesores de un vértice v tienen tiempo asignado, entonces $tl(v)$ es el mínimo de 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' .

Solución

Si se aplica el algoritmo indicado al grafo se tiene:

Nodos	1	2	3	4	5	6	7
Tl	0	6	6	3	12	10	15

- 17.10.** Una ruta crítica de una red es un camino desde un vértice que no tiene predecesores hasta un 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 del ejercicio 17.7.

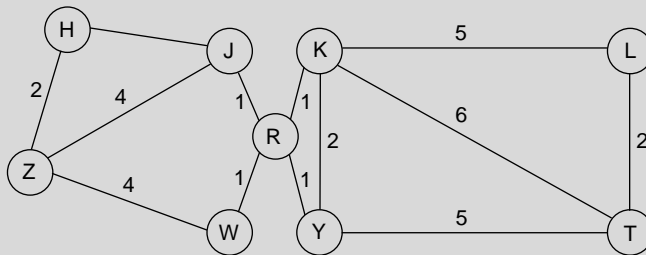
Solución

De acuerdo con los dos ejercicios anteriores se tiene:

Nodos	1	2	3	4	5	6	7
Tn	0	2	5	3	12	9	15
Nodos	1	2	3	4	5	6	7
Tl	0	6	6	3	12	10	15

Por lo tanto sólo hay una ruta crítica, y viene dada por los nodos 1, 4, 5, 7

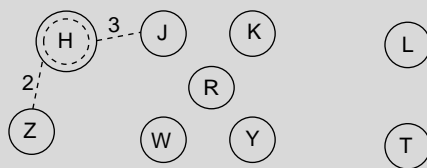
- 17.11.** Dado el grafo de la figura siguiente, encontrar un árbol de expansión de coste mínimo mediante el algoritmo de Prim.



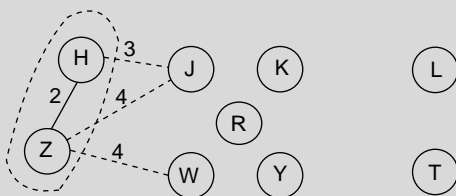
Solución

Si se aplica el algoritmo de Prim en los sucesivos pasos se tiene:

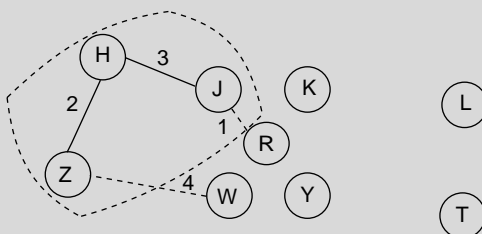
Paso 0



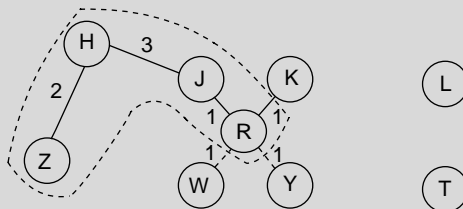
Paso 1
Elige
vértice



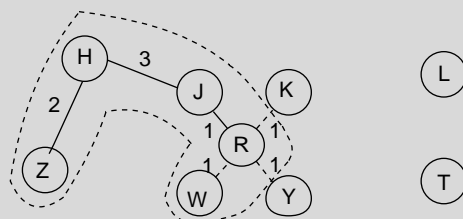
Paso 2
Elige
vértice J



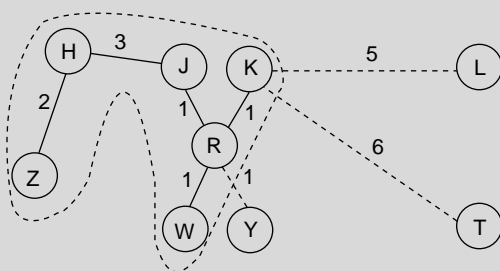
Paso 3
Elige
vértice R



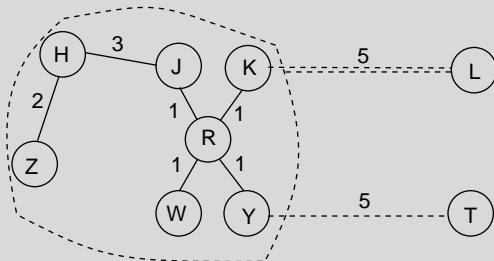
Paso 4
Elige
vértice W



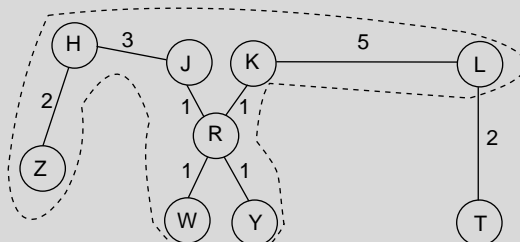
Paso 5
Elige
vértice K



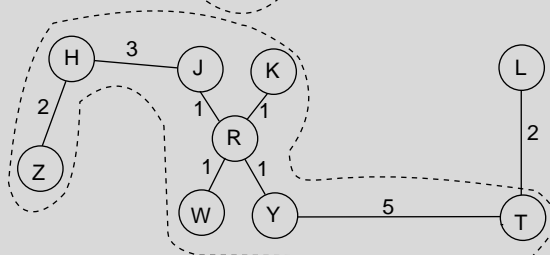
Paso 6
Elige
vértice Y



Paso 7
Elige
vértice L

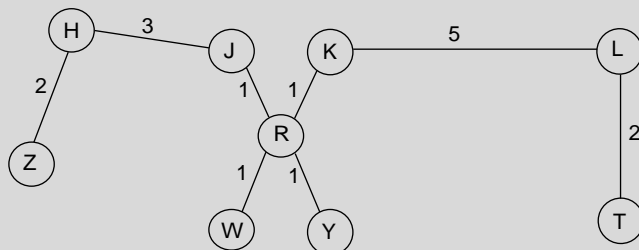


Paso 7'
Elige
vértice T



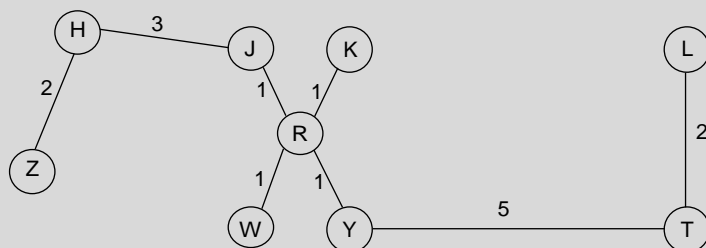
Paso 8
Elige
vértice T

SOLUCIÓN 1



Paso 8'
Elige
vértice L

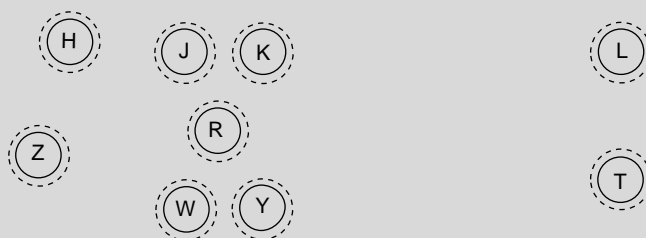
SOLUCIÓN 2



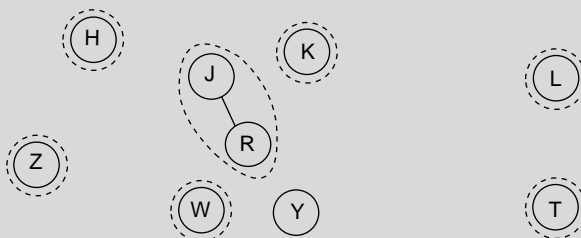
17.12. Dado el grafo del ejercicio 17.11 encuentre un árbol de expansión de coste mínimo con el algoritmo de Kruskal.

Solución

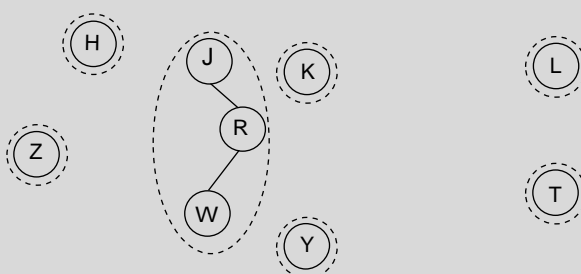
Paso 0



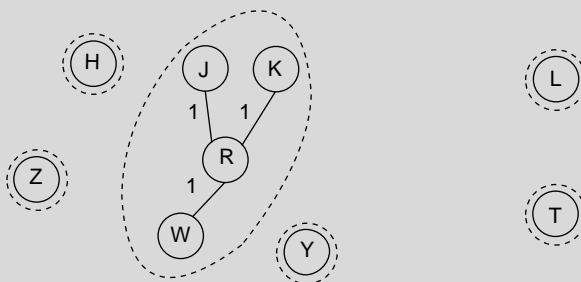
Paso 1
Elige R con J



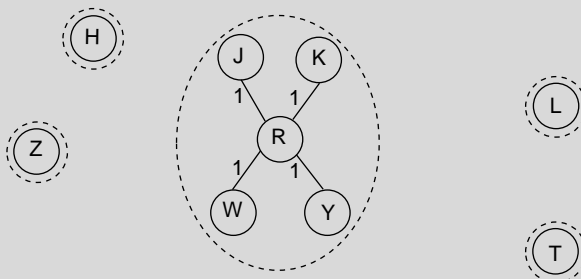
Paso 2
Elige R con W



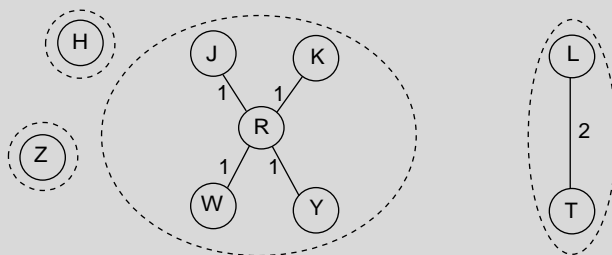
Paso 3
Elige R con K



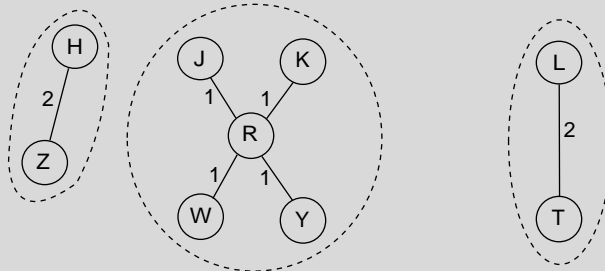
Paso 4
Elige R con Y



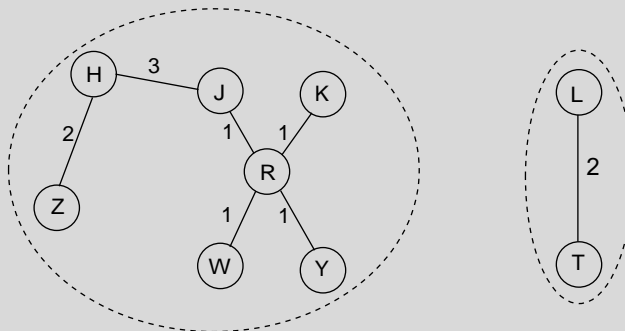
Paso 5
Elige L con T



Paso 6
Elige H con Z



Paso 7
Elige H con J

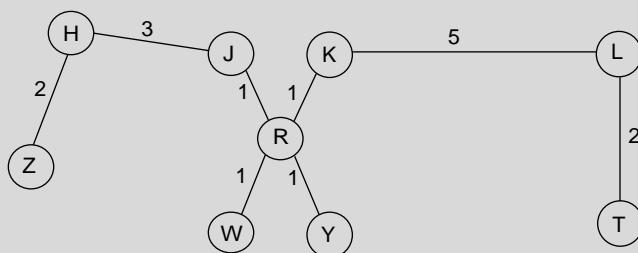


Paso 8 Elige Z con W y se rechaza, forma ciclo

Paso 9 Elige Z con J y se rechaza, forma ciclo

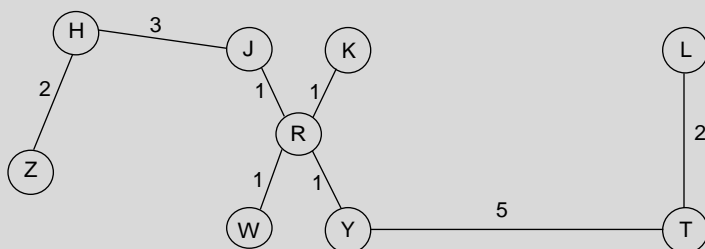
Paso 10
Elige K con L

SOLUCIÓN 1

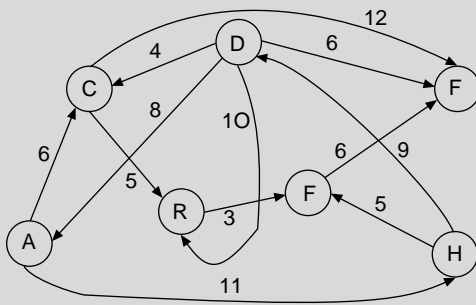


Paso 10
Elige Y con T

SOLUCIÓN 2



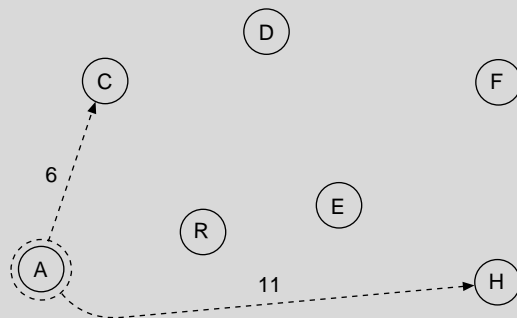
17.13. En el grafo dirigido con factor de peso de la figura siguiente:



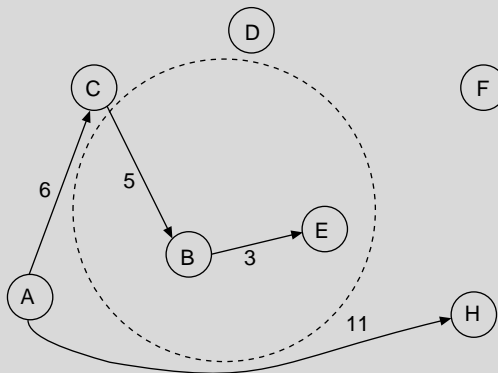
Encuentrar el camino más corto desde el vértice A a todos los demás vértices del grafo usando el algoritmo de Dijkstra.

Solución

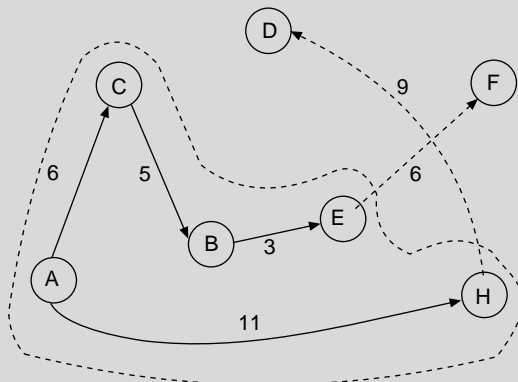
Paso 0



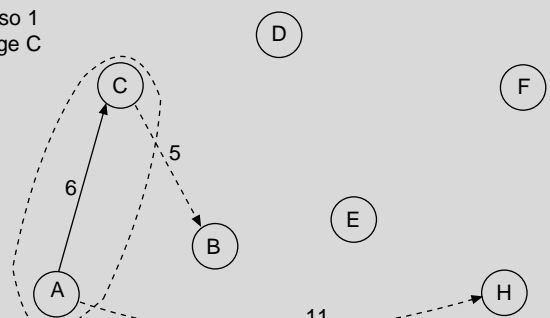
Paso 2
Elige B



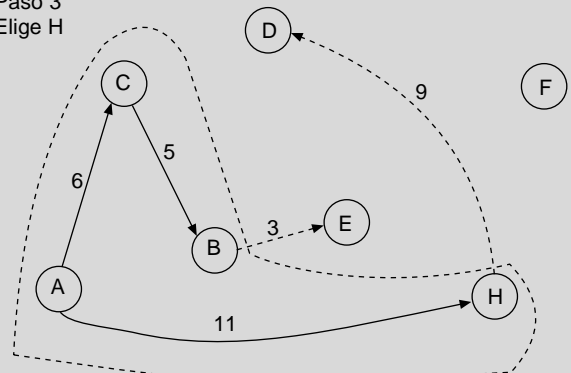
Paso 4
Elige E



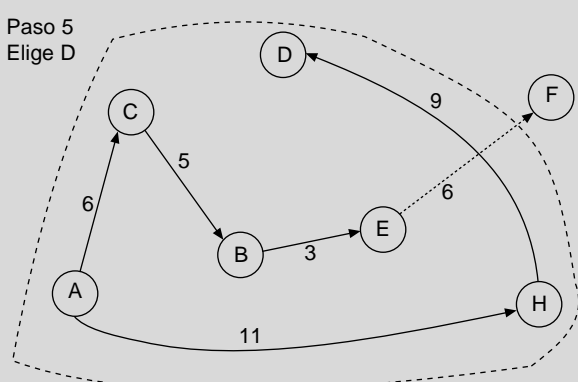
Paso 1
Elige C



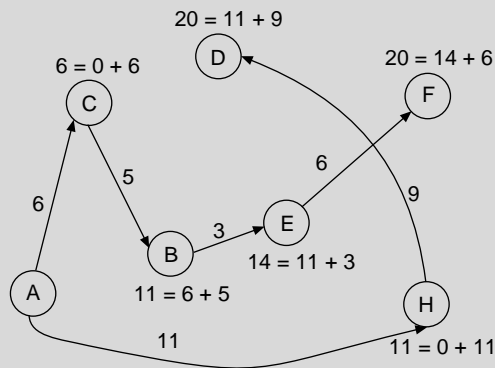
Paso 3
Elige H



Paso 5
Elige D



Paso 6
Elige F



Aplicando el algoritmo de Dijkstra se tiene:

Distancia desde A a cada uno de los vértices

A	B	C	D	E	F	H
0	11	6	20	14	20	11

17.14. En el grafo del Ejercicio 17.13 encontrar el camino más corto desde el vértice D a todos los demás vértices del grafo siguiendo el algoritmo de Dijkstra.

PROBLEMAS BÁSICOS

17.15. Codificar el algoritmo que realice la ordenación topológica de un grafo dirigido y sin ciclos.

Análisis

En el archivo Cola.h está implementada una cola. Este archivo es incluido para realizar la ordenación topológica. Se supone que el grafo viene dado por la matriz de pesos y que entre dos vértices no existe arco cuando el valor de la entrada de la matriz de pesos es -100.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#include <Cola.h>
#define Max -100
#define n 10
void MostrarCola(Cola C);
void PoIterativo (float a[][n], int GradodeEntrada[n], Cola *C);
void LeeGrafo(float a[][n])
{
```

```
    /*Lectura de la matriz del grafo*/
}

void EscribeMatriz(float a[][n])
{
    /*Escribe la matriz de pesos del grafo*/
}

void GradosdeEntrada (float a[][n], int GradodeEntrada[n])
{
    /*Determina los grados de entrada de cada uno de los vértices del grafo*/
    int i, j;
    for(j = 0; j < n; j++)
    {
        GradodeEntrada[j] = 0;
        for (i = 0; i < n; i++)
            if (a[i][j] != Max)
                GradodeEntrada[j]++;
    }
}

void PoIterativo (float a[][n], int GradodeEntrada[n], Cola *C)
{
    /*Realiza la ordenación topológica iterativamente*/
    Cola C1;
    int i,j;
    VaciaC(&C1);
    VaciaC(C);
    for (i = 0; i < n; i++)
        if (GradodeEntrada[i] == 0)
            AnadeC(&C1, i);
    while (! EsVacíaC(C1))
    {
        i = PrimeroC(C1);
        BorrarC(&C1);
        AnadeC(&C, i);
        for( j = 0; j < n; ++j)
            if (a[i][j] != Max)
                if (GradodeEntrada[j] != 0)
                {
                    GradodeEntrada[j]--;
                    if (GradodeEntrada[j] == 0)
                        AnadeC(&C1, j);
                }
    }
}

void MostrarCola(Cola C)
{
    int i;
    while (! EsVacíaC(C))
    {
        i = PrimeroC(C );
    }
}
```



```

        BorrarC(&C);
        printf("%d",i);
    }
}

void main (void)
{
    int GradodeEntrada[n];
    float a[n][n];
    Cola C;
    LeeGrafo(a);
    EscribeMatriz(a);
    printf(" Comienza P0\n");
    GradosdeEntrada(a, GradodeEntrada);
    PoIterativo(a, GradodeEntrada, &C);
    MostrarCola(C);
    printf(" final de Ordenación topológica\n");
}

```

17.16. *Escribir los tipos de datos y una función para codificar el algoritmo de Dijkstra.*

Análisis

Se usa el valor de `Infinito = 1000` para indicar que entre los vértices i , y j no existe arco. El grafo viene dado por la matriz de pesos M . Se da la distancia mínima del vértice 0 al resto de los vértices en el vector $D[j]$ de tal forma que $D[j]$, es la distancia del camino más corto para ir del vértice 0 al vértice j . El predecesor inmediato para la decodificación del camino mínimo se dará en el vector $P[j]$. Se usan las funciones `PertenceConjunto` que decide si un vértice i está en un conjunto C . `VaciaConj` que crea el conjunto vacío, y `AnadeConj` que añade a un conjunto C un vértice, y el tipo de datos `Conjunto`.

Codificación

```

#define Infinito 1000
#define n 5
int Minimo(float D[n], Conjunto C)
{
    int i, aux1;
    float aux;
    aux = Infinito;
    aux1 = 0;
    for (i = 1; i < n; i++)
        if ((aux > D[i]) && (!(PertenceConj(C,i)))
        {
            aux = D[i];
            aux1 = i;
        }
    return(aux1);
}

void Dijkstra(float M[n][n], float D[n], int P[n])
{
    int i, j, k;
    Conjunto C;
}

```

```

VacíaConj(&C);
AnadeConj(&C,0);
for (i = 0; i < n; i++)
{
    D[i] = M[0][i];
    P[i] = 0;
}
for(i = 0; i < n - 1; i++)
{
    j = Minimo(D, C);
    AnadeConj(&C,j);
    for (k = 1; k < n; k++)
        if (! PerteneceConj(C,k))
            if (D[k] > D[j] + M[j][k])
            {
                D[k] = D[j] + M[j][k];
                P[k] = j;
            }
}
}

```

17.17. Codificar el algoritmo de Floyd.

Análisis

Si la matriz de pesos viene dada por A , se calcula la distancia mas corta entre todos los pares de nodos i, j del grafo dando el resultado en la matriz D . Además devuelve el camino codificado en la matriz P de tal manera que $P[i, j]=0$ si el camino más corto para ir de i a j es directo $P[i, j] = k$ si k es el índice que permite calcular el valor más pequeño de $D[i, j]$ y para ir del vértice i al vértice j , hay que ir primero del vértice i al k para posteriormente ir del vértice k al j . Se considera que 1.000 es la distancia infinito, y que el grafo tiene 5 vértices.

Codificación

```

#define Infinito 1000
#define n 5
#define Max -100

void Floyd (float A[n][n], float D[n][n], int P[n][n])
{
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            if (A[i][j] != Max)
                D[i][j] = A[i][j];
            else
                D[i][j] = Infinito;
            P[i][j] = 0;
        }
    for (k = 0; k < n; k++)
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                if (D[i][j] > D[i][k] + D[k][j])

```

```

    {
        D[i][j] = D[i][k] + D[k][j];
        P[i][j] = k
    }
}

```

17.18. Codificar el algoritmo de Ford-Fulkerson.

Análisis

Los pasos del algoritmo de aumento de flujo son:

1. Parte de un flujo inicial, $F_{ij} = 0$.
2. Marcar el vértice origen s .
3. Repetir hasta que sea marcado el vértice sumidero t , o bien no sea posible marcar más vértices.
 - 3.1. Si un vértice i está marcado, existe el arco (i, j) y es tal que (i, j) es arco incrementable marcar vértice j .
 - 3.2. Si un vértice i está marcado, existe el arco (j, i) y es tal que (j, i) es un arco reducible marcar el vértice j .
4. Si ha sido marcado el vértice sumidero t , se tiene una cadena de aumento de flujo. Aumentar el flujo al máximo aumentando el flujo permitido por la cadena. Actualizar los flujos de cada arco con las unidades de aumento o disminución. Borrar todas las marcas. Volver al paso 2.
5. Si no ha sido marcado el vértice t finalizar la aplicación del algoritmo, no es posible enviar más flujo desde s hasta t .

Se consideran S y T los vértices fuente y sumidero respectivamente. Cap es una Array que contiene la capacidad máxima que admite cada uno de los arcos del grafo. F es un array que contendrá al final del programa el flujo total que circula a través de los distintos arcos de la red. $FjoTotal$ es el flujo total que circulará por la red del vértice fuente S al sumidero T . La codificación se estructura de la siguiente forma:

- La función *Imprime* se encarga de escribir el flujo total de la red.
- La función *Cadena*, escribe una cadena de aumento de flujo del vértice S al vértice T . Para ello usa un array de predecesores.
- La función *alguno*, encuentra un vértice que se encuentre en el array *Finvia*.
- La función *Maxflujo* codifica el algoritmo de Ford-Fulkerson. *Precede* es un array que indica a cada vértice el predecesor en una posible cadena de aumento de flujo. *Mejora* indica en cuanto puede mejorarse una cadena de aumento de flujo del vértice S . *Finvia* indica si un vértice es fin de una posible cadena incrementable de aumento de flujo. *Adelante* indica si el arco involucrado en la cadena, es de aumento o de disminución de flujo. *Encadena* indica si un vértice ya ha sido usado en alguna cadena de aumento de flujo.
- La función *main* lee las capacidades de los arcos de la red y llama a la función *Maxflujo*.

Codificación (Se encuentra en la página Web del libro).

17.19. Codificar algoritmo de Prim.

Análisis

Se usa el valor de $Infinito = 1000$ para indicar que entre los vértices i , y j no existe arco. El grafo viene dado por la matriz de pesos M de tal manera que si entre los vértices i y j no existe arista entonces $M[i, j] = M[j, i] = infinito$. El árbol de expansión se dará en una matriz A , con las mismas características que la matriz M . Se parte del vértice 0 , y de un conjunto C de tal manera que cuando un vértice esté en C ya se encuentra en el árbol de expansión en construcción. Un vector de distancias D mantiene la distancia mínima de los vértices que no estén en C al árbol de expansión en construcción. Se implementa una función *Mínimo* que encontrará el índice j de D (vértice del grafo) que hace que $D[j]$ sea mínimo de entre todos los vértices que no estén en el conjunto C . El predecesor inmediato para la decodificación del camino mínimo se dará en el vector $P[j]$. Se usan las funciones *PertenceConjunto* que decide si un vértice i está en un conjunto C . *VaciaConj* que crea el conjunto *Vacio*, y *AnadeConj* que añade a un conjunto C un vértice. El tipo *Conjunto* se supone ya declarado.

17.20. Codificar algoritmo de Kruskal.**Análisis**

Se supone que en el archivo `Conj.h` se encuentra una implementación de Conjuntos que sirve para almacenar las distintas componentes conexas del grafo. En este archivo se define el tipo `Conjunto` y las funciones `void VacíaConj(Conjunto *C)` que vacía un conjunto `C`; la función `void UnionConj(Conjunto C1, Conjunto C2, Conjunto &C3)` une los conjuntos `C1` y `C2` en `C3`; la función `int PerteneceConj(int v, Conjunto C)` que decide si el vértice `v` está en el conjunto `C`; la función `void AnadeConj(int v, Conjunto *C)` añade el vértice `v` al conjunto `C`. El resto del programa se estructura de la siguiente forma: la función `OrdenaQuickSort` ordena las aristas del grafo en orden creciente. Estas aristas son almacenadas en una lista de aristas del grafo; la función `Inicializar` se encarga de inicializar la lista de aristas del grafo, tomándola de la matriz de pesos y posteriormente llama a la función `OrdenaQuickSort` para ordenarlas crecientemente; La función `Combina` combina dos componentes conexas del grafo; la función `NumComponConexa` decide la componente conexa en la que se encuentra un vértice; la función `Kruskal` aplica el algoritmo de Kruskal descrito en la teoría; la función `LeerGrafo` lee los arcos del grafo que no tienen valor infinito.

Codificación

```
#include <Conj.h>
#define Infinito 1000
#define n 10
#define True 1
#define False 0
struct Arco
{
    int u, v;
    float Coste;
};

void OrdenaQuickSort( Arco Lars[n*n], int iz, int de); /* ordena los arcos */
void Inicializar(float C[n][n],Arco T[n*n],int *Ka)
{
    /*De la matriz de costes, obtiene los arcos en orden ascendente de peso*/
    int i,j;
    /*Primero son almacenados los arcos. Para después ordenar según el coste*/
    *Ka = -1;
    for ( i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (C[i][j] != Infinito) /* arco <i,j> */
            {
                (*Ka) ++;
                T[*Ka].u = i;
                T[*Ka].v = j;
                T[*Ka].Coste = C[i][j];
            }
    OrdenaQuickSort(T,0, *Ka);
}

void Combina(int Cu, int Cv , Conjunto Cx[n])
{
    /*Une las componentes conexas Cu, Cv. La unión queda en la posición Cu;
    la componente de la posición Cv queda vacía*/
    Conjunto Caux;
```

```

    UnionConj(Cx[Cu], Cx[Cu], &Caux);
    Cx[Cu] = Caux;
    VaciaConj(&Cx[Cv]);
}

int NumComponConexa(int v, Conjunto Cx[n])
{
    /*Obtiene el número de componente donde se encuentra el vértice v*/
    int Nc, Comx;
    /* El número de componentes es n, aunque puede haber componentes vacías*/
    Nc = -1;
    Comx = False;
    while ((Nc < n - 1) && (! Comx))
    {
        Nc ++ ;
        Comx = PerteneceConj(v, Cx[Nc]);
    }
    if (Comx)
        return( Nc);
    else
        return(n);
}

void Kruskal(float C[n][n], Arco T[n*n], float *Ct)
{
    Arco ListArcos[n*n], Min_Ar;
    Conjunto CompConx[n];
    int v, C_u, C_v, Comp_n;
    int Kn, Ka, K;
    Kn = -1;
    *Ct = 0;
    /* Número de arcos en T */
    /*Coste del árbol de expansión */
    Inicializar(C, ListArcos, &Ka);
    /*obtiene los Ka arcos en orden creciente de costes*/
    for (v = 0; v < n; v ++ )
    {
        VaciaConj(&(CompConx[v]));
        AnadeConj(v, &(CompConx[v]));
    }
    /*Obtiene los n componentes conexos iniciales, con cada vértice*/
    Comp_n = n - 1;
    K = 0;
    /*Número de componentes conexas*/
    while (Comp_n > 0)
    {
        K ++;
        Min_Ar = ListArcos[K];
        C_u = NumComponConexa(Min_Ar.u, CompConx);
        /* Arco mínimo actual */
        /*Componente donde se encuentra el vértice u del arco mínimo*/
        C_v = NumComponConexa(Min_Ar.v, CompConx);
        if (C_u != C_v)
            /*Conecta dos componentes distintos*/
            {
                (*Ct) += Min_Ar.Coste;
                printf("Arco %d X %d ", Min_Ar.u, Min_Ar.v);
                Combina(C_u, C_v, CompConx);
            }
    }
}

```

```

        /*Une componentes u y v.Nueva componente queda en u*/
        Comp_n --;
        T[Kn] = Min_Ar;
        Kn ++;
    }
}
}

```

17.21. Escribir un programa que represente en memoria una red (un *Pert*: grafo dirigido sin ciclos y factor de peso) mediante la matriz de adyacencia (matriz de pesos) y calcule:

- El tiempo $tn(v)$.
 - El tiempo $tl(v)$.
 - Además encuentre los nodos que puedan formar parte de rutas críticas de la red.
- (Nota: Ver los problemas de seguimiento resueltos 17.7, 17.8, 17.9.).

Análisis

El programa que se presenta se descompone en los siguientes funciones.

- Función `leeGrafo`, lee los valores de los arcos. Si se lee -1 indicará que no hay arco.
- Función `escribeMatriz`, se encarga de escribir la matriz de costes.
- Función `GradoDeEntrada`, calcula el grado de entrada de cada vértice del grafo.
- Función `GradoDeSalida`, calcula el grado de salida de cada vértice del grafo.
- Función `CalculaTn`, calcula los valores de T_n para cada vértice del grafo de acuerdo con el algoritmo descrito en el ejercicio 17.7.
- Función `CalculaTl`, calcula los valores de T_l para cada vértice del grafo de acuerdo con el algoritmo descrito en el ejercicio 17.8.
- Función `CalculaNodosRuta`, calcula los vértices del grafo que pueden formar parte de una ruta crítica de acuerdo con el algoritmo descrito en el ejercicio 17.9.

Codificación

```

#include <stdio.h>
#include <stdlib.h>
#include <Cola.h>
#define Max -1
#define Infinito 1000
#define n 7

void LeeGrafo(float M[n][n]);
void EscribeMatriz(float M[n][n]);
void GradosdeEntrada (float a[][n], int GradodeEntrada[n])
{
    /*Determina los grados de entrada de cada uno de los vértices del grafo*/
    ...
}

void GradoDeSalida(float M[n][n],int GdS[n])
{
    /*Determina los grados de salida de cada uno de los vértices del grafo*/
    ...
}

```

```

void CalculaTn(float M[n][n], int Tn[n])
{
    Cola C;
    int GdE[n], i, j, k, Maximo;
    GradosdeEntrada(M, GdE);
    VacíaC(&C);
    for(i = 0; i < n; i++)
        if (GdE[i] == 0)
        {
            AnadeC(&C, i);
            Tn[i] = 0;
        }
    while (! EsVacíaC(C))
    {
        k=PrimeroC(C);
        BorrarC(&C);
        for( j = 0; j < n; j++)
            if (M[k][j] != Max)
            {
                GdE[j] --;
                if (GdE[j] == 0)
                {
                    AnadeC(&C, j);
                    Maximo = -1;
                    for (i = 0; i < n; i++)
                        if ((M[i][j] != Max) && (Maximo < (Tn[i] + M[i][j])))
                            Maximo = Tn[i] + M[i][j];
                    Tn[j]= Maximo;
                }
            }
        }
    printf("\n");
    for (i = 0; i < n; i++)
        printf("%5d",Tn[i]);
    printf(" valores de tn\n");
}

void CalculaTl(float M[n][n], int Tn[n], int Tl[n])
{
    Cola C;
    int GdS[n],i, j, k, Minimo;
    GradoDeSalida(M, GdS);
    VacíaC(&C);
    for(i = 0; i < n; i++)
        if (GdS[i] == 0)
        {
            AnadeC(&C, i);
            Tl[i] =Tn[i];
        }
    while (! EsVacíaC(C))
    {
        k = PrimeroC(C);
        BorrarC(&C);
    }
}

```



```

    for( i = 0; i < n; i++)
        if (M[i][k] != Max)
        {
            GdS[i] --;
            if (GdS[i] == 0)
            {
                AnadeC(&C, i);
                Minimo = Infinito;
                for (j = 0; j < n; j++)
                    if ((M[i][j] != Max) && (Minimo > (Tl[i] + M[i][j])))
                        Minimo = Tl[i] - M[i][j];
                Tn[i] = Minimo;
            }
        }
    }
    for (i = 0; i < n; i++)
        printf("%5d",Tl[i]);
    printf(" valores de t1\n");
}

void CalculaNodosRuta(int Tn[n], int Tl[n])
{
    int i;
    printf("nodos con Tn = Tl\n");
    for(i = 0; i < n; i++)
        if (Tl[i] == Tn[i])
            printf("%5d",i);
}

void main (void)
{
    int Tn[n],Tl[n];
    float M[n][n];
    LeeGrafo(M);
    EscribeMatriz(M);
    CalculaTn(M, Tn);
    CalculaTl(M, Tn, Tl);
    CalculaNodosRuta(Tn, Tl)
}

```

PROBLEMAS AVANZADOS

17.22. Se quiere cablear con fibra todas las 10 ciudades de una provincia; para tal fin se dispone de un archivo llamado `arcos.dat` de texto, en el que figuran las conexiones entre las ciudades y los metros de cable de fibra óptica necesario para unirlos. En cada línea figura una serie de números con el siguiente significado: cada tres números, los dos primeros indican dos ciudades y el tercero los metros de cable necesarios para unirlos. El objetivo es gastar la menor cantidad posible de cable, de tal modo que queden todas las ciudades unidas por cable óptico. Se pide realizar un programa que genere el grafo (mediante listas de adyacencia), y posteriormente obtenga en la salida, las ciudades que se han de unir y los metros de cable utilizados para conseguir el objetivo anterior.

Análisis

Por las condiciones del problema, se debe unir todas las ciudades para que se puedan comunicar entre ellas a un coste mínimo; es decir con el menor número de metros. Es un problema de árbol de recubrimiento de coste mínimo, cuya solución viene dada por el algoritmo de Kruskal o de Prim. Se elige Kruskal. La solución se estructura de la siguiente forma:

- La función `grafovacio` crea el grafo vacío, implementado con una *array* de vértices de tal manera que cada entrada del *array* es una lista en la que figuran los vértices adyacentes y el coste de sus arcos.
- La función `insertarenlista` inserta en una lista enlazada un nodo.
- La función `creargrafo` lee el grafo del fichero y crea la estructura de datos que representa el grafo.
- La función `ordenar` almacena todos los arcos del grafo en un *array* de arcos los ordena crecientemente por el método del montículo implementado mediante la función `monton` que a su vez usa la función `hundir`, la explicación del funcionamiento de estas funciones puede consultarse en el capítulo de ordenación.
- La función `nohayarcos` decide cuando se han terminado los arcos del grafo.
- La función `eliminar` elimina la arista mas corta del *array* de aristas y nos dice la arista y su peso.
- La función `inicializar` se encarga de inicializar *n* conjuntos a un sólo vértice. Estos conjuntos se representan mediante un *array* de padres. El padre no apunta a su hijo, sino que es el hijo el que apunta a su padre. El nodo raíz apunta a `-1`, no tiene padre. Cada nodo esta en un árbol que se identifica por el nodo raíz.
- La función `encontrar` decide si dos vértices se encuentran en un mismo conjunto y en caso de que no lo esté, fusiona los conjuntos, cambiando el padre de la raíz de uno de ellos.
- La función `Kruskal` implementa el algoritmo usando para almacenar las componentes conexas los árboles descritos anteriormente.
- La función `main` se encarga de llamar a las distintas funciones.

Codificación

```
#include <stdio.h>
#include <stdlib.h>
#define Infinito 1000
#define n 10
#define True 1
#define False 0
struct Arista
{
    int x,y;
};
typedef struct Nodo
{
    int v, valor;
    struct Nodo* sig;
}Lista;
struct registrokruskal
```

```

{
    Arista a;
    int coste;
};

struct Tipokruskal
{
    int conteo;
    registrokruskal arcos[n];
};

Lista *g[n];
void grafovacio ()
{ /*vacía el grafo g*/
    int i;
    for(i=0;i<n;i++)
        (g[i]) = NULL;
}

void insertarenlista (Lista **l,int x, int coste)
{ /*inserta un nodo como primer elemento de la lista*/
    Lista *nuevo;
    nuevo=(Lista*)malloc(sizeof (Lista));
    nuevo->v = x;
    nuevo->valor= coste;
    nuevo->sig = *l;
    *l = nuevo ;
}

void creargrafo()
{
    FILE *f;
    int a1, a2, coste;
    if ((f=fopen("arcos.dat","rt"))==NULL)
    { printf(" error en archivo texto");
      exit(1);
    }
    while (!feof(f))
    {
        fscanf(f,"%d %d %d ",&a1, &a2,&coste);
        insertarenlista(&g[a1],a2,coste);
    }
    fclose(f);
}

void monton ( Tipokruskal *a)
{ /* ordena por el método de ordenación del montón*/
}

void ordenar (Tipokruskal *conjuntoarcos)
{
    int i,c;
    Lista *l;
    conjuntoarcos->conteo = -1;
    for (i=0;i<n;i++)

```

```

        if (g[i] !=NULL)
        {
            l = g[i];
            while (l !=NULL)
            {
                conjuntoarcos->conteo++; c=conjuntoarcos->conteo;
                conjuntoarcos->arcos[c].a.x = i;
                conjuntoarcos->arcos[c].a.y = l->v;
                conjuntoarcos->arcos[c].coste=l->valor;
                l= l->sig;
            }
        }
        monton(conjuntoarcos);
    }

    int nohayarcos ( Tipokruskal t)
    {
        /* Decide si el grafo está vacío */
        return ( t.conteo == -1);
    }

    void eliminar ( Tipokruskal *t, Arista *a, int *m)
    {
        /* elimina de la lista de arcos el primero */
        int i;
        *a = t->arcos[0].a; *m = t->arcos[0].coste;
        for (i = 0; i < t->conteo; i++)
            t->arcos[i] = t->arcos[i+1];
        t->conteo--;
    }

    void inicializar(int p[n])
    {
        int i;
        for (i = 0 ; i < n; i++)
            p[i] = -1 ;
    }

    int encontrar (int p[n],int x,int y)
    {
        while (p[x] > -1 ) x = p[x];
        while (p[y] > -1 ) y = p[y];
        if (x!=y)
        {
            p[y]=x; return(True);
        }
        else
            return(False);
    }

    void kruskal ( Tipokruskal *arbol, int *hay)
    {
        int p[n],i=-1,n;
        Arista a;
        Tipokruskal conjuntoarcos;
    }

```

```

inicializar(p);
/*t inicialmente no tiene arcos y p son solo los vértices*/
ordenar ( &conjuntoarcos);
/*ordenar el grafo g por coste de las aristas en conjuntoarcos*/
*hay = True;
while (i < n - 2)
{
    if (nohayarcos(conjuntoarcos))
    {
        *hay = False;
        i = n;
    }
    else
        eliminar(&conjuntoarcos, &a, &m);
    if (encontrar(p,a.x, a.y))
    { /*Obtiene el número de componente donde se encuentra el vértice v*/
        i++;
        arbol->arcos[i].a = a;
        arbol->arcos[i].coste = m;
        arbol->conteo = i;
    }
}
}

void main (void)
{
    int i,hay, cos;
    Tipokruskal t; creargrafo ();
    /*el grafo es no dirigido, pero a efectos del problema solo se pone un arco*/
    kruskal ( &t, &hay);
    /*Salida de resultados */
    if (hay)
    {
        cos = 0;
        for (i =0;i<=t.conteo;i++)
        {
            printf("Metros:%d %d =%d\n",t.arcos[i].a.x,t.arcos[i].a.y,t.arcos[i].coste);
            cos +=t.arcos[i].coste;
        }
        printf("Metros  totales = %d\n", cos);
    }
    else
        printf("No hay arbol. El  grafo no es conexo");
}

```

17.23. Dado un grafo G dirigido y valorado positivamente con n vértice. Se pide realizar un programa recursivo que encuentre y escriba (si existe) un camino de longitud l de una ciudad a a otra b .

Análisis

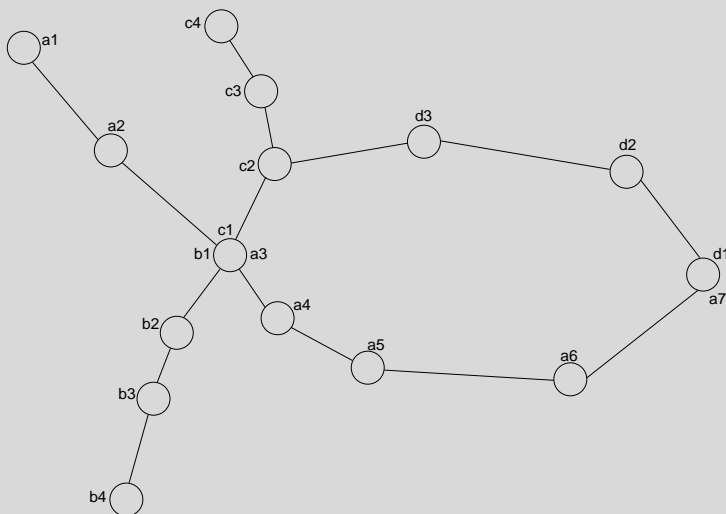
Se implementa el grafo mediante un *array* de un número máximo de nodos de listas de adyacencia. La solución del problema usa las funciones `grafovacio`, `insertarenlista` y `creargrafo` implementadas en el ejercicio 17.22. La función `EncontrarCamino` implementa la recursividad de una forma sencilla: corta la recursividad cuando se encuentra una solución

$k=0$, o cuando se está seguro de que no la hay $k<0$, y prueba con todos los sucesores de cada nodo en otro caso. La función `main` se encarga de hacer las correspondientes llamadas.

Codificación (Se encuentra en la página Web del libro)

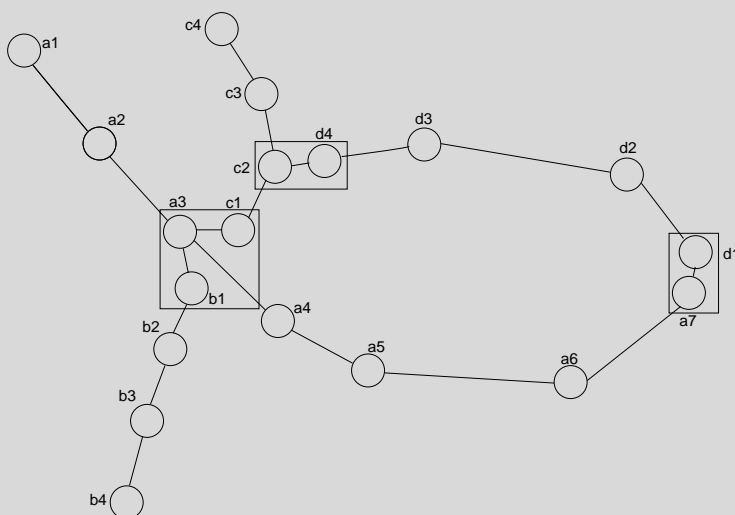
17.24. En la figura siguiente se representa una porción de la red del metro de una ciudad imaginaria. Sabiendo que entre dos estaciones de la misma línea, el tiempo que se tarda, es de 5 minutos y que el tiempo de transbordo de una línea a otra es de 8 minutos, se pide:

- Modelizar mediante un grafo dicha porción de plano. Realizar las declaraciones necesarias para representarlo en memoria.
- Escribir un programa que calcule el tiempo mínimo entre cada par de estaciones de metro así como los recorridos correspondientes a dichos tiempo.



Análisis

El problema que se presenta para modelizar el grafo es el de representación de los transbordos entre líneas. Esto se resuelve representando cada estación de transbordo en otras dos, con un arco que las una que será el tiempo de transbordo (8 minutos), de la siguiente forma:



- Para la codificación del algoritmo se usan las variables globales $D[n][n]$ para la decodificación de los caminos $Q[n][n]$, $A[n][n]$ y almacenar la matriz de Floyd y los pesos de los arcos. La matriz $V[n][n]$ almacena los nombres de las estaciones. Las funciones `Floyd DecodificaCamino`, y `DecodificaCaminoRecursivamente`, son el algoritmo de Floyd y la decodificación de los caminos tal y como se explica en teoría. La función `Inicializa` se encarga de inicializar la matriz D y las matrices A y V según el enunciado. La función `EscribeTodosCamino`, se encarga de escribirlos.

Codificación (Se encuentra en la página Web del libro)

17.25. *Se quiere almacenar en un grafo los pueblos de una determinada comarca y la red de carreteras que los unen. Codificar un programa que cumpla este propósito y que determine el pueblo más alejado y el más cercano de la capital de la región así como una decodificación de los caminos.*

Análisis

Para resolver el problema, basta con calcular la distancia mínima de la capital a todos los pueblos, mediante el algoritmo de Dijkstra. El pueblo más cercano será el primer pueblo que “se coma” el algoritmo voraz de Dijkstra y el más alejado será el último que “se coma” el citado algoritmo.

17.26. *Sea G un grafo dirigido y j un vértice del grafo. Se define la excentricidad del vértice j respecto del grafo G como el máximo de las longitudes de los caminos mínimos que van desde cualquier otro vértice i del grafo al vértice j . Se define el Centro del grafo G como el vértice que tiene mínima excentricidad. Supongamos que se tiene implementado un grafo G a través de la matriz de pesos. Escribir un segmento de programa que:*

- *Calcule la excentricidad de cada uno de los vértices del grafo y nos la presente en pantalla.*
- *Calcule el centro del grafo y nos presente en pantalla la distancia mínima de cada uno de los vértices del grafo al centro.*

Análisis

Para resolver el problema planteado basta con aplicar el algoritmo de Floyd. De esta forma se calcula la distancia mínima entre todos los vértices del grafo. Para calcular la excentricidad basta con encontrar el máximo de cada columna. El centro del grafo es el vértice que le corresponda el mínimo de las excentricidades. La solución que se plantea, parte de que se tiene calculada la matriz de distancias mínimas Q en una variable global. Las excentricidades de cada vértice se almacenan en la variable global Ex .

Codificación

```
void Excentricidad()
{
    int i, j; float Maximo;
    printf(" excentricidad de los vértices\n");
    for (j = 0; j < n; j++)
    {
        Maximo = Q[0][j];
        for (i = 1; i < n; i++)
            if (Q[i][j] > Maximo)
                Maximo = A[i][j];
        Ex[j] = Maximo;
        printf(" La excentricidad de %d es %f\n", j, Maximo);
    };
}

void CentrodelGrafo()
{
    int i, Centro;
    float Minimo;
    Minimo = Ex[0];
    Centro = 0;
    for(i=1; i<n; i++)
        if (Ex[i] < Minimo)
        {
            Minimo = Ex[i];
            Centro = i;
        }
}
```

```

    };
    printf(" El centro del grafo es %d\n", Centro);
}

```

PROBLEMAS PROPUESTOS

- 17.1.** Escribir un procedimiento para implementar el algoritmo de Dijkstra con esta modificación: en la búsqueda del camino mínimo desde el origen a cualquier vértice puede haber más de un camino del mismo camino mínimo, entonces seleccionar aquel con el menor número de arcos.
- 17.2.** El algoritmo de Dijkstra resuelve el problema de hallar los caminos mínimos desde un único vértice origen a los demás vértices. Escribir una función para resolver el problema de que dado un grafo G representado por su matriz de pesos encuentre los caminos mínimos desde todo vértice V a un mismo vértice destino D .
- 17.3.** 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 todo vértice tiene iguales sus grado de entrada y de salida. Implementar un algoritmo para encontrar, si existe, un circuito de Euler.
- 17.4.** Un grafo está representado en memoria mediante listas de adyacencia. Escribir las rutinas necesarias para determinar la matriz de caminos.
(Nota: Seguir la estrategia expuesta en el algoritmo de Warshall pero sin utilizar la matriz de adyacencia).
- 17.5.** Se quiere escolarizar una zona rural compuesta de 4 poblaciones: Lupiana, Centenera, Atanzón y Pinilla. Para ello se va a construir un centro escolar en la población que mejor coste de desplazamiento educativo tenga (mínimo de la función Z_i). $Z_i = \sum p_j d_{ij}$; donde p_j es la población escolar de la población j y d_{ij} es la distancia mínima del pueblo j al pueblo i . Las distancias entre los pueblos en Kilómetros:

Lup	Cen	Atn	Pin
Lup	7	11	4
Cen		5	12
Atn			8

La población escolar de cada pueblo: 28, 12, 24, 8 respectivamente de Lupiana, Centenera, Atanzón y Pinilla. Codificar un programa que tenga como entrada los datos expuestos y determine la población donde conviene situar el centro escolar.

- 17.6.** Un grafo G representa una red de centros de distribución. Cada centro dispone de una serie de artículos y un stock de ellos, representado mediante un estructura lineal ordenada respecto al código del artículo. Los centros están conectados aunque no necesariamente bidireccionalmente.

Escribir un programa en el que se represente el grafo como un grafo dirigido ponderado (el factor de peso que represente la distancia en kilómetros entre dos centros). En el programa debe estar la opción de que un centro H no tenga el artículo z y lo requiera, entonces el centro más cercano que disponga de z se lo suministra.

- 17.7.** Europa consta de n capitales de cada uno de sus estados; cada par de ciudades está conectada, o no, por vía aérea. En caso de estar conectadas (se entiende por vuelo directo) se sabe las millas de la conexión y el precio del vuelo. Las conexiones no tienen por qué ser bidireccionales, así puede haber vuelo Viena-Roma y no Roma-Viena.

Escribir un programa que represente la estructura expuesta y resuelva el problema: disponemos de una cantidad de dinero D , deseamos realizar un viaje entre dos capitales, $C1$ - $C2$, y queremos información sobre la ruta más corta que se ajuste a nuestro bolsillo.

- 17.8.** La ciudad dormitorio de Martufa está conectada a través de una red de carreteras, que pasa por poblaciones intermedias, con el centro de la gran ciudad. Cada conexión entre dos nodos soporta un número máximo de vehículos a la hora. Escribir un programa para simular la salida de vehículos de la ciudad dormitorio y llegada por las diversas conexiones al centro de la ciudad. La entrada de datos ha de ser los nodos de que consta la red (incluyendo ciudad-dormitorio y centro-ciudad) y la capacidad de cada conexión entre dos nodos. El programa de calcular el máximo de vehículos/h que pueden llegar al centro y cómo se distribuyen por las distintas calzadas.

17.9. Dados los siguientes valores de un grafo.

∞	5	∞	∞	∞	∞	∞	∞
∞	∞	7	∞	8	3	∞	∞
∞	∞	∞	2	∞	∞	9	∞
∞	∞	3	∞	∞	∞	∞	12
6	∞	∞	∞	∞	4	∞	∞
∞	∞	∞	∞	∞	∞	10	∞
∞	∞	∞	∞	∞	6	∞	7
∞	∞	∞	∞	∞	∞	∞	4

Se pide:

- ¿El grafo es dirigido o no dirigido? Indicar por qué.
- Un camino de longitud cuatro entre los vértices 2 y 4.
- Indique si tiene bucles el grafo y en su caso, cuál o cuáles.
- Indicar si existe un circuito y en su caso, escríbalo.
- Indicar si el grafo es fuertemente conexo, y si no lo es escribir sus componentes fuertemente conexas.

17.10. Modificar los algoritmos de Prim y de Kruskal para encontrar el árbol de expansión máximo.

17.11. Un multigrafo es un grafo en el cual se permiten aristas múltiples entre pares de vértices. ¿Qué algoritmos de los capítulos 16 y 17 funcionan para multigrafos? Proponst modificaciones para los algoritmos que no funcionen.

17.12. Escribir un programa en el que dado un grafo *valorado*, con ciertos pesos de aristas negativos, determine los caminos más cortos desde un vértice origen al resto de los vértices.

17.13. En un centro de análisis de datos se dispone de tres computadores que pueden realizar una serie de tareas. El centro debe realizar cuatro tareas distintas, pero no todos los ordenadores pueden realizar todas las tareas. Se A la matriz dada, donde $A_{ij}=1$ si la tarea i la puede realizar el ordenador j , y $A_{ij}=0$ en caso contrario. Se trata de resolver el problema de asignación de tareas planteado, teniendo en cuenta que el ordenador 1 sólo puede realizar una tarea y que los ordenadores 2 y 3 un máximo de 2. Hacer una representación gráfica del problema para poder resolverlo mediante un algoritmo del flujo máximo. Resolver el problema planteado manualmente, dando un resultado final, así como los pasos intermedios

EJERCICIOS DE EXAMEN

- 1.1. Escribir un algoritmo que triangule una matriz cuadrada y hallar su determinante. En las matrices cuadradas el valor del determinante coincide con el producto de los elementos de la diagonal de la matriz triangulada, multiplicado por -1 tantas veces como hayamos intercambiado filas al triangular la matriz.

Análisis

El proceso de triangulación y cálculo del determinante tiene las siguientes operaciones:

- Inicializar `signo` a 1
- Desde `k` igual a 0 hasta `n-2` hacer:
 - a) Localizar entre las filas, desde `k` a `n-1`, aquella donde se encuentra el pivote, que será el elemento mayor existente en la columna `k`.
 - b) Si la fila `k` no es la que tiene el mayor elemento en la columna `k`, intercambiar las filas. Cuando se intercambia se multiplica por -1 el valor de la variable `signo`.
 - c) Utilizar el pivote para convertir en cero los elementos situados por debajo en esa misma columna. El determinante de una matriz no cambia si se sustituye una fila por esa misma fila a la que se suma otra de las filas multiplicada por un factor cualquiera.
- Asignar al determinante el valor de `signo` por el producto de los elementos de la diagonal de la matriz triangulada.

Codificación (Se encuentra en la página web del libro)

- 1.2. Utilizar el método de las aproximaciones sucesivas para resolver el problema del juego de la vida de Conway. Este problema consiste en imaginar una rejilla rectangular en la que cada una de sus casillas puede estar ocupada o no por un microorganismo vivo y, siguiendo las reglas de reproducción y muerte para los microorganismos que se indicarán a continuación, averiguar el estado de la población al cabo de un cierto número de generaciones. Las leyes para la reproducción y muerte de los microorganismos son:

- Si en una celda hay un microorganismo vivo y a su alrededor sólo hay otro vivo o ninguno, muere de soledad.
- Los que tienen dos vecinos se mantienen como están.
- En las celdas con 3 vecinos nace un microorganismo en la próxima generación.
- En aquellas que tengan 4, 5, 6, 7 u 8 vecinos el ser vivo que hubiera en ellas muere por hacinamiento.
- Todos los nacimientos y muertes tienen lugar al mismo tiempo.

Análisis

Para resolver el ejercicio se puede utilizar un `array` cuyas celdas se marquen con el valor `vivo` (celda con microorganismo vivo) o `muerto` (celda en la que no existe microorganismo). En cada generación se recorrerán para cada una de las celdas todas las que la rodean y se contarán las vivas, si el número de vivas existentes a su alrededor es 2 se deja la celda como está, si es tres se marca como viva y en los demás casos se marca como muerta. Las marcas se efectúan en un `array` auxiliar, *Nuevo Mapa*, que posteriormente se vuelca sobre el original.

MAPA

	*			
	*			
				*
			*	*

NUEVO MAPA

			*	*
			*	*

El planteamiento general podría ser

Inicio

<Leer mapa y número de generaciones>

<Escribir el mapa>

desde <la 1ª> hasta <la última generación> hacer

desde <la 1ª> hasta <la última celda> hacer

/* este bucle se traducirá en dos, uno que recorre las filas y otro que recorre las columnas */

<Calcular vecinos de una celda visitando las que la rodean. Hay que tener en cuenta que cada celda está rodeada por otras 8 excepto las situadas en posiciones extremas de la tabla>

<Asignar valores al nuevo mapa>

fin_desde

<Asignar el nuevo mapa al mapa>

<Escribir el mapa>

fin_desde

fin

Este método no es muy bueno pues obliga a recorridos innecesarios, ya que, en las sucesivas generaciones, el número de vecinos sólo cambia en las celdas situadas alrededor de otras en las que nacen o mueren microorganismos. Resulta por tanto más eficaz el siguiente planteamiento:

MAPA

	*			
	*			
				*
			*	*

NÚMERO DE VECINOS

2	1	2	0	0
2	1	2	0	0
1	1	1	1	1
0	0	1	3	2
0	0	1	2	2

tabla auxiliar donde se registra el número de vecinos de cada celda

LISTA DE VIVOS (los que les corresponde nacer)

1	
4	4

LISTA DE MUERTOS (los que les corresponde morir)

2	
1	2

SIGUIENTES VIVOS

0 (número de elementos de la lista)
Lista celdas (fila y columna de cada una)

SIGUIENTES MUERTOS

0
...

Variables con estructura análoga, inicialmente vacías. Almacenan las celdas donde deben nacer o morir microorganismos en la siguiente generación).

```

inicio
<Leer el número de generaciones>
<Leer el mapa y obtener y almacenar el número de vecinos de cada celda>
<Almacenar las celdas sin microorganismos cuyo número de vecinos sea 3 en una lista de vivos(cel-
das que contendrán vivos) y las que tienen microorganismos vivos con un número de vecinos infe-
rior a 2 o superior a 3 en una lista de muertos(celdas que contendrán muertos)>
<Escribir el mapa>
desde <la 1ª generación> hasta <la última> hacer
    <Recorrer la lista de vivos creando microorganismos en aquellas celdas del mapa que no los
    tenían ya y cuyo número de vecinos es 3. Quitar de la lista de vivos las repeticiones>
    <Recorrer la lista de muertos destruyendo en el mapa aquellos que están vivos y les
    corresponde morir por tener un número de vecinos inferior a 2 o superior a 3. Quitar de la
    lista de muertos las repeticiones>
    <Escribir el mapa>
    <Sumar vecinos alrededor de las casillas reflejadas en la lista de vivos y añadir las que
    alcancen 3 vecinos a una lista de siguientes vivos y las que adquieran 4 a otra de
    siguientes muertos>
    <Sustraer vecinos alrededor de las casillas reflejadas en la lista de muertos y añadir las
    que obtengan 3 vecinos a la lista de siguientes vivos y las que se queden con 1 a la de
    siguientes muertos>
    <Copiar siguientes vivos en la lista de vivos>
    <Copiar siguientes muertos en la lista de muertos>
fin_desde
fin

```

Codificación (Se encuentra en la página web del libro)

- 1.3. *Suponga que dispone de una colección de productos químicos y una serie de cajas con capacidad suficiente para almacenarlos; no obstante, algunos de estos productos, pueden originar violentas reacciones cuando se guardan juntos, por lo que los denominaremos incompatibles. Diseñar un algoritmo que permita mediante “backtracking” determinar el mínimo número de cajas necesarias para almacenar los mencionados productos teniendo en cuenta que sólo se almacenarán en la misma caja los que sean compatibles.*

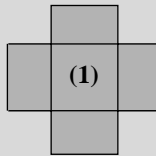
Análisis

Para resolver el problema se supone que el máximo número de productos y de cajas es 10.

- Las incompatibilidades entre unos productos y otros se representarán mediante una matriz, donde filas y columnas representan los productos de tal manera que si un producto i es incompatible con otro j , la celda correspondiente de la matriz almacenará *verdad* y cuando sean compatibles almacenará *falso*.
- La solución será almacenada en un registro con dos campos, uno de tipo *array* con tantos elementos como productos donde se guardará la caja correspondiente a cada producto, y otro de tipo entero donde se almacenará el número máximo de cajas.
- El problema se divide en pasos, donde se considera cada uno de los productos y en cada paso se analizan las distintas posibilidades. Las posibilidades para un determinado producto i son las distintas cajas. La función *aceptable*, se encarga de decidir si es posible guardar el producto i en una determinada caja. Será factible si la caja es distinta de la que tengan todos productos incompatibles con él, en cuyo caso se anota, es decir se asigna al producto su caja (borrar anotación, es marcarlo como sin caja). La función *Distinta*, se encarga de decidir si la caja en la que se va a almacenar un producto es una nueva caja, es decir una caja donde aun no se ha almacenado nada, por tanto el número de cajas necesarias para almacenar los productos se incrementará en una unidad. Para encontrar la solución óptima hay que hallar todas las soluciones, es decir todas formas de almacenamiento posibles sin que ocurran reacciones peligrosas, y quedarse con aquella que requiera un menor número de cajas.
- Otras funciones interesantes son, la función *inicializar*, encargada de almacenar la información sobre productos incompatibles, así como inicializar una solución cuyo número de cajas sea infinito y la función *EscribeSolución*, que escribe la solución óptima.

Codificación (Ver página web del libro)

- 1.4. En “Futurlandia” se prevee una invasión extraterrestre; ante este peligro imagine el país como una matriz de 4 por 4 y utilice *backtracking* para determinar el mínimo número de soldados que serían necesarios para proteger el territorio. Las casillas que puede proteger cada soldado, en relación con el lugar en que está situado, se indican en el siguiente gráfico.



(1) Posición donde se encuentra situado el soldado.

Análisis

Para proteger todo el territorio una solución inmediata es colocar un soldado en cada casilla del tablero. Las estructuras de datos y variables que usa el problema son:

- Dos *arrays* unidimensionales paralelos *a*, *b* para guardar la información sobre las casillas que pueden proteger los soldados.
- Cuatro *arrays* bidimensionales *h*, *h1*, *s*, *s1*. El *array h* sirve para indicar el número de veces que ha sido protegida una casilla en la solución en curso. El *array h1* sirve para guardar el óptimo hasta el momento. El *array* bidimensional *s* sirve para guardar el soldado asignado a una determinada casilla en la solución en curso. Si una casilla no tiene asignado soldado el valor de *s* será cero. La tabla *s1* almacena la solución óptima hasta el momento.
- La variable *p* indica el número de casillas protegidas en la solución en curso. De tal manera que cuando *p* valga $n \times n$ se habrá obtenido una solución.
- La variable *sold* indica el número de soldados que se han colocado en la solución en curso, y la variable *soldopt* el número de soldados de la solución encontrada óptima hasta el momento.

El programa se estructura en las siguientes funciones:

1. La función *solución*, escribe la solución encontrada.
2. La función *ensayarn* resuelve recursivamente el problema. El parámetro *sold* indica el número de soldado que se va a poner en la solución en curso. Las posibilidades son todas las casillas del tablero. Es aceptable una posición cuando el valor del *array s* es cero. A la hora de anotar hay que marcar en el *array s* el número de soldado, y marcar todas las posiciones que se protejan desde la posición del soldado en el *array h*, de tal manera que si hay alguna que sea marcada de nuevo, el contador *p* se debe incrementar en una unidad. Estaremos en una solución cuando el número de casillas protegidas sea todo el tablero, o lo que es lo mismo, cuando la variable *p* valga $n \times n$. Cuando se encuentra una solución hay que comprobar si es mejor que la óptima hallada hasta el momento.

Codificación (se encuentra en la página web del libro)

- 1.5. Crear un programa con las funciones necesarias para la creación e inserción de elementos en un nuevo tipo de árbol AVL2 (AVL con factor de equilibrio 2). En un árbol AVL con factor de equilibrio 2 la altura entre los subárboles izquierdo y derecho de cualquier nodo no podrá diferir en más de 2 unidades..

Análisis

La inserción de un elemento en el árbol AVL2 utilizará el algoritmo usual de inserción de un nuevo elemento en un árbol binario modificado con la finalidad de conseguir que en ningún momento la altura de los subárboles izquierdo y derecho de un nodo difiera en más de dos unidades. Para poder determinar esto con facilidad cada uno de los nodos de un AVL tendrá un campo donde almacenar su *factor de equilibrio*. El *factor de equilibrio* de un nodo debe oscilar entre -2, -1, 0, 1 y 2, y cualquier otro valor implicaría la reestructuración del árbol.

El proceso de inserción consistirá en:

- Comparar el elemento a insertar con el nodo raíz; si es mayor avanzar hacia el subárbol derecho, si es menor hacia el izquierdo y repetir la operación de comparación hasta encontrar un elemento igual o llegar al final del subárbol donde debiera estar el nuevo elemento.
- Cuando se llega al final es porque no se ha encontrado, por lo que se crea un nuevo nodo donde se coloca el elemento y, tras asignarle un cero como factor de equilibrio, se añade como hijo izquierdo o derecho del nodo anterior según corresponda por la comparación de sus campos de clasificación. En este momento también se activará un interruptor sw para indicar que el subárbol ha crecido en altura.
- Regresar por el camino de búsqueda y si sw está activo calcular el nuevo factor de equilibrio del nodo que está siendo visitado.

Los casos a considerar cuando se actualiza el factor de equilibrio (Fe) de un nodo al que se llega tras haber efectuado una inserción en su rama izquierda son:

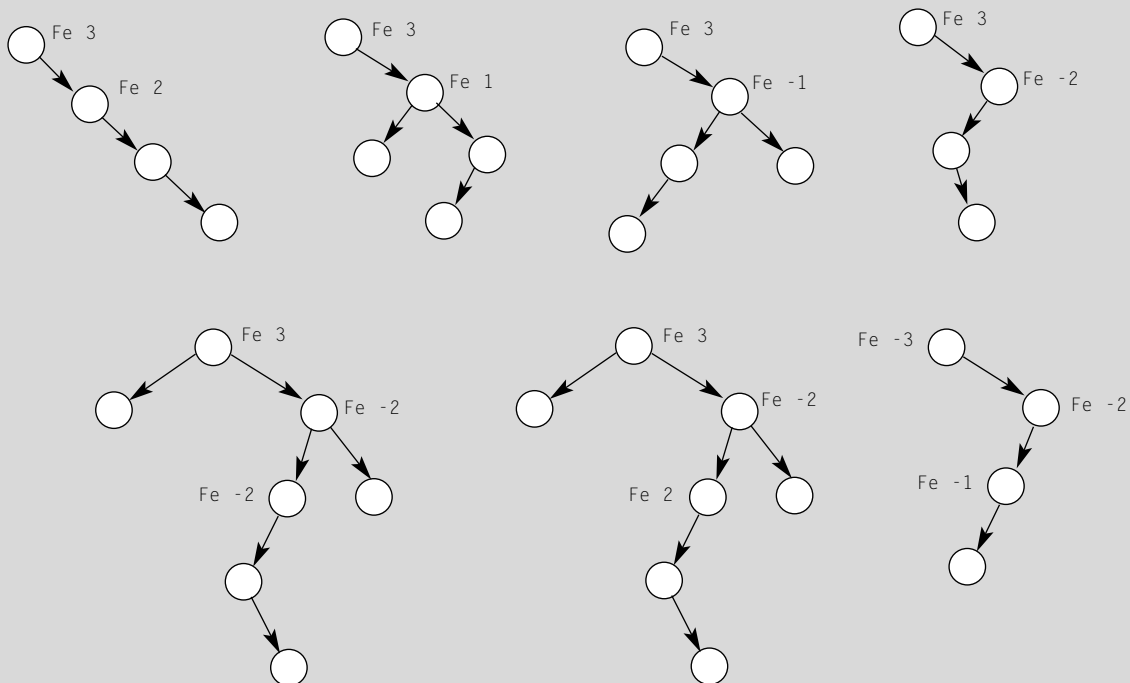
- La rama derecha es más alta que la izquierda en una o dos unidades, por tanto un nuevo elemento en la rama izquierda no consigue que el subárbol crezca, la altura del subárbol sigue siendo la de su rama derecha que es la mayor y sw se desactiva.
- Las ramas izquierda y derecha del nodo considerado tenían anteriormente la misma altura, o bien la rama izquierda era más alta que la derecha en una unidad, en ambos casos la inserción de un nuevo elemento en la rama izquierda incrementa la altura del subárbol izquierdo.
- La rama izquierda era más alta que la derecha en dos unidades y un nuevo elemento en la rama izquierda rompe el equilibrio del árbol y hace que sea necesaria su reestructuración. El subárbol reestructurado no ha crecido y por tanto no puede modificar la diferencia de alturas entre las ramas derecha e izquierda de los nodos superiores, sw se desactiva.

La actualización del factor de equilibrio de un nodo al que se llega tras haber efectuado una inserción en su rama derecha son similares:

- La rama izquierda era más alta que la derecha, así que un nuevo elemento en la rama derecha no consigue que el subárbol crezca y sw se desactiva.
- Las ramas izquierda y derecha del nodo considerado tenían anteriormente la misma altura o bien la rama derecha era más alta que la izquierda en una unidad, por lo que, al insertar un elemento en la rama derecha, la altura del subárbol se incrementa.
- La rama derecha era más alta que la izquierda en dos unidades y un nuevo elemento en la rama derecha rompe el equilibrio del árbol y hace que sea necesaria su reestructuración. La reestructuración consigue que la rama donde se ha insertado el elemento no crezca, entonces sw se desactiva.
- Durante el proceso de inserción, al actualizar los factores de equilibrio, si un nodo deja de cumplir el criterio de equilibrio (factor de equilibrio entre -2 y 2) el árbol deberá ser reestructurado. Reestructurar un árbol significa rotar los nodos del mismo.
- Cuando la inserción se ha efectuado en la rama derecha las distintas situaciones que pueden presentarse para el nodo desequilibrado son:

Fe del nodo que requiere reestructuración (valor no permitido)	3	3	3	3
Fe del nodo anterior	2	1	-1	-2

Por ejemplo:



Se originarán pues rotaciones derecha-derecha y derecha izquierda. En las rotaciones derecha izquierda para el cálculo del factor de equilibrio es necesario tener en cuenta también la situación del antepenúltimo nodo visitado.

Fe del nodo que requiere reestructuración	3	3	3	3	3	3	3	3
Fe del nodo anterior	-1	-1	-1	-1	-2	-2	-2	-2
Fe del nodo anterior al anterior	1	2	-1	-2	1	2	-1	-2

Las inserciones en la rama izquierda del nodo que se está considerando plantean situaciones simétricas.

Fe del nodo que requiere reestructuración (valor no permitido)	-3	-3	-3	-3
Fe del nodo anterior	2	1	-1	-2

Por ejemplo, dado el siguiente árbol.

```

90 (Fe=0)
80 (Fe=-2)
    60 (Fe=0)
    50 (Fe=1)
    40 (Fe=0)
    30 (Fe=-1)
    20 (Fe=0)

Deme numero (0 -> Fin): 70

```

al insertar el 70 se requiere una rotación izquierda-derecha,

Fe del nodo que requiere reestructuración	-3
Fe del nodo anterior	1
Fe del nodo anterior al anterior	2

y el resultado que se obtiene es:

90 (Fe=0)
80 (Fe=-1)
70 (Fe=0)
60 (Fe=1)
50 (Fe=0)
40 (Fe=-2)
30 (Fe=-1)
20 (Fe=0)

Si el árbol hubiera sido este otro

90 (Fe=0)
80 (Fe=-2)
50 (Fe=-1)
45 (Fe=0)
40 (Fe=0)
30 (Fe=-1)
20 (Fe=0)
Deme numero (0 -> Fin): 42

y se inserta el 42 la situación que se produce es:

Fe del nodo que requiere reestructuración	-3
Fe del nodo anterior	1
Fe del nodo anterior al anterior	-2

lo que también se arregla con rotación izquierda-derecha.

90 (Fe=0)
80 (Fe=-1)
50 (Fe=-1)
45 (Fe=-1)
42 (Fe=0)
40 (Fe=0)
30 (Fe=-1)
20 (Fe=0)

Codificación (Se encuentra en la página web del libro)

- 1.6. Diseñar un programa que genere la matriz de incidencia de un grafo a partir de la información almacenada en un archivo de texto. La información almacenada en las líneas de elementos del archivo de texto es: “Nodo Inicio” “Nodo Final” “Valor”, por ejemplo:

```
1 2 1k
1 3 1k
2 5 10k
..
```

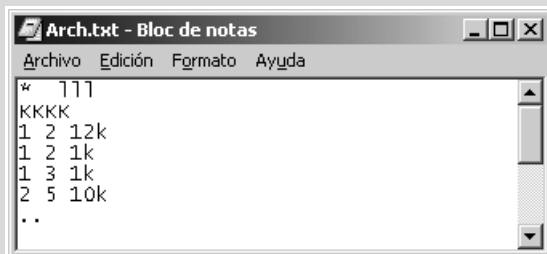
El archivo podrá tener además otras líneas de cabecera o comentarios, pero éstas nunca empezarán por un dígito. Se admite la posibilidad de que diferentes elementos conecten los mismos vértices, pero no la de elementos que conecten un vértice consigo mismo.

Análisis

La matriz de incidencia tendrá una columna para cada elemento y una fila por nodo y, puesto que inicialmente se desconoce el número de filas y columnas, su creación se hará de forma dinámica. Las celdas tendrán un +1 si el elemento sale del nodo y un -1 si llega. Las celdas con valor 0 indican que el elemento no tiene relación con el nodo. Por ejemplo:

	1 (0-1)	2 (1-2)	3 (2-3)	4 (3-4)	5 (4-5)	6 (3-6)	7 (2-4)	8 (0-6)	9 (0-5)
0	1	0	0	0	0	0	0	1	1
1	-1	1	0	0	0	0	0	0	0
2	0	-1	1	0	0	0	1	0	0
3	0	0	-1	1	0	1	0	0	0
4	0	0	0	-1	1	0	-1	0	0
5	0	0	0	0	-1	0	0	0	-1
6	0	0	0	0	0	-1	0	-1	0

Respecto a la lectura de los datos, supuesto un archivo similar al siguiente:



la lectura del archivo deberá reconocer las líneas con información de interés, esto es las líneas de elementos. El primer paso será pues efectuar una lectura secuencial del archivo seleccionando las líneas adecuadas para encontrar el nodo de mayor valor y contar el número de elementos. A continuación se crea la matriz y se vuelve a leer el archivo para asignar los valores a las celdas.

Codificación (Se encuentra en la página web del libro)

¡Estudia a tu propio ritmo y aprueba tu examen con Schaum!

Los Schaum son la herramienta esencial para la preparación de tus exámenes.
Cada Schaum incluye:

- Teoría de la asignatura con definiciones, principios y teoremas claves.
- Problemas resueltos y totalmente explicados, en grado creciente de dificultad.
- Problemas propuestos con sus respuestas.

Hay un mundo de Schaum a tu alcance...¡BUSCA TU COLOR!



www.mhe.es/joyanes

www.mcgraw-hill.es

The McGraw-Hill Co