Algoritmos y estructuras de datos

Una perspectiva en C

• Luis Joyanes Aguilar

• Ignacio Zahonero Martínez

McGraw Hill

ALGORITMOS Y ESTRUCTURAS DE DATOS

Una perspectiva en C

ALGORITMOS Y ESTRUCTURAS DE DATOS

Una perspectiva en C

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

Mc Graw Hill

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

CONTENIDO

Prólogo xv

PARTE I
ANÁLISIS DE ALGORITMOS Y ESTRUCTURAS DE DATOS BÁSICAS

Capítulo 1. Algoritmos, estructuras de datos y programas 3

1.1. Tipos de datos 4
1.1.1. Tipos de datos primitivos 4
1.1.2. Tipos de datos compuestos y agregados 6
1.2. La necesidad de las estructuras de datos 7
1.2.1. Etapas en la selección de una estructura de datos 8
1.3. Abstracción y tipos abstractos de datos 9
1.3.1. Composición de un TAD 10
1.3.2. Modelo de tipo abstracto de dato 11
1.4. Algoritmos y programas 12
1.4.1. Propiedades de los algoritmos 12
1.4.2. Programas 13
1.5. Representación de los algoritmos: el pseudocódigo 14
1.5.1. Codificación de un programa 16
1.6. Eficiencia y exactitud 17
1.6.1. Eficiencia de un algoritmo 18
1.6.2. Formato general de la eficiencia 19
RESUMEN 27
EJERCICIOS 28

Capítulo 2. Análisis de algoritmos 31

2.1. La medida de la eficiencia de algoritmos 32
2.1.1. Análisis del orden de magnitud 32
2.1.2. Consideraciones de eficiencia 33
2.1.3. Análisis de rendimiento 33
2.1.4. Tiempo de ejecución 34
2.2. Notación O-grande 35
2.2.1. Descripción de tiempos de ejecución con la notación O 35
2.2.2. Definición conceptual 36
2.3. Complejidad de funciones 37
2.3.1. Funciones tipo monomio 37
2.3.2. Órdenes de funciones polinómicas 38

2.3.3. Órdenes exponenciales y logarítmicas 38
2.3.4. Bases de logaritmos distintas 39
2.3.5. Orden de una función general 39
2.3.6. Propiedades de la notación *O-grande* 40
2.4. Inconvenientes de la notación *O-grande* 41
2.5. Complejidad de las sentencias de una función C 42
2.5.1. Complejidades más comunes en algoritmos 44
2.6. Análisis de algoritmos de ordenación 45
2.6.1. Análisis de la ordenación por selección 45
2.6.2. Análisis de la ordenación por inserción 46
2.7. La eficiencia de los algoritmos de búsqueda 47
2.7.1. Búsqueda secuencial 47
2.7.2. Búsqueda binaria 48
RESUMEN 49
EJERCICIOS 50
PROBLEMAS 53

Capítulo 3. Arrays (listas y tablas) en C 55

3.1. Arrays (arreglos) 56
3.1.1. Declaración de un array 56
3.1.2. Subíndices de un array 57
3.1.3. Almacenamiento en memoria de los arrays 58
3.1.4. El tamaño de los arrays 59
3.1.5. Verificación del rango del índice de un array 60
3.2. Inicialización de un array 60
3.3. Arrays de caracteres y cadenas de texto 62
3.4. Arrays multidimensionales 64
3.4.1. Inicialización de arrays multidimensionales 66
3.4.2. Acceso a los elementos de los arrays bidimensionales 67
3.4.3. Lectura y escritura de elementos de los arrays bidimensionales 67
3.4.4. Acceso a elementos mediante bucles 68
3.4.5. Arrays de más de dos dimensiones 69
3.4.6. Una aplicación de un array tridimensional 70
3.5. Utilización de arrays como parámetros 71
3.5.1. Precauciones 74
3.5.2. Paso de cadenas como parámetros 76
RESUMEN 80
EJERCICIOS 80
PROBLEMAS 82

Capítulo 4. Estructuras y uniones 87

4.1. Estructuras 88
4.1.1. Declaración de una estructura 89
4.1.2. Definición de variables de estructuras 89
4.1.3. Uso de estructuras en asignaciones 91
4.1.4. Inicialización de una declaración de estructuras 91
4.1.5. El tamaño de una estructura 92
4.2. Acceso a estructuras 93
4.2.1. Almacenamiento de información en estructuras 93
4.2.2. Lectura de información de una estructura 95
4.2.3. Recuperación de información de una estructura 95
4.3. Estructuras anidadas 96
4.3.1. Ejemplo de estructuras anidadas 97
4.4. Arrays de estructuras 101
4.4.1. Arrays como miembros 101
4.5. Utilización de estructuras como parámetros 103

4.6. Uniones 104
4.7. Enumeraciones 106
4.7.1. sizeof de tipos de datos estructurados 108
4.7.2. typedef 109
4.8. Campos de bits 111
RESUMEN 115
EJERCICIOS 115
PROBLEMAS 116

PARTE II
DISEÑO DE ALGORITMOS, RECURSIVIDAD, ORDENACIÓN Y BÚSQUEDA

Capítulo 5. Recursividad 121

5.1. Naturaleza de la recursividad 122
5.2. Funciones recursivas 125
5.2.1. Recursividad indirecta: funciones mutuamente recursivas 126
5.2.2. Condición de terminación de la recursión 128
5.3. Recursión *versus* iteración 128
5.3.1. Directrices en la toma de decisión iteración/recursión 130
5.4. Recursión infinita 130
5.5. Algoritmos *divide* y *vencerás* 132
5.5.1. Torres de Hanoi 132
5.5.2. Búsqueda binaria recursiva 137
5.6. Ordenación por mezclas: *mergesort* 139
5.6.1. Algoritmo *mergesort* en C 139
5.7. *Backtracking*, algoritmos de vuelta atrás 141
5.7.1. Problema del salto del caballo 142
5.7.2. Problema de las ocho reinas 145
5.8. Resolución de problemas con algoritmos de *vuelta atrás* 147
5.8.1. Generar las distintas variaciones $V_{m,n}$ 147
5.8.2. Objetos que suman un peso 150
5.9. Selección óptima 152
5.9.1. Problema del viajante 154
RESUMEN 157
EJERCICIOS 158
PROBLEMAS 160

Capítulo 6. Algoritmos de ordenación y búsqueda 165

6.1. Ordenación 166
6.2. Algoritmos de ordenación básicos 167
6.3. Ordenación por intercambio 168
6.4. Ordenación por selección 171
6.4.1. Algoritmo de selección 172
6.4.2. Codificación en C del algoritmo de selección 172
6.5. Ordenación por inserción 173
6.5.1. Algoritmo de ordenación por inserción 173
6.5.2. Codificación en C del algoritmo de ordenación por inserción 173
6.6. Ordenación por burbuja 174
6.6.1. Algoritmo de la burbuja 174
6.6.2. Codificación en C del algoritmo de la burbuja 176
6.6.3. Análisis del algoritmo de la burbuja 177
6.7. Ordenación Shell 178
6.7.1. Algoritmo de ordenación Shell 178

6.7.2. Codificación en C del algoritmo del método Shell 179
16.8. Ordenación rápida (*quicksort*) 180
6.8.1. Algoritmo *quicksort* 181
6.8.2. Codificación en C del algoritmo *quicksort* 184
6.8.3. Análisis del algoritmo *quicksort* 185
6.9. Ordenación Binsort y Radixsort 186
6.9.1. Método de Binsort 186
6.9.2. Método de Radixsort 188
6.10. Búsqueda en listas: búsquedas secuencial y binaria 190
6.10.1. Búsqueda secuencial 190
6.10.2. Búsqueda binaria 190
6.10.3. Algoritmo y codificación de la búsqueda binaria 191
6.10.4. Análisis de los algoritmos de búsqueda 193
RESUMEN 195
EJERCICIOS 195
PROBLEMAS 196

Capítulo 7. Algoritmos de ordenación de archivos 199

7.1. Archivos en C 200
7.1.1. Flujos 200
7.1.2. Puntero (apuntador) FILE 201
7.1.3. Apertura de un archivo 202
7.1.4. Modos de apertura de un archivo 203
7.1.5. Macros `NULL` y `EOF` 204
7.1.6. Cierre de archivos 204
7.2. Lectura y escritura de archivos de texto 205
7.2.1. Escribir caracteres: `putc ()`, `fputc ()` 205
7.2.2. Lectura de caracteres: funciones `getc ()` y `fgetc ()` 206
7.2.3. Escribir y leer cadenas de caracteres: funciones `fputs ()` y `fgets ()` 206
7.2.4. Entrada y salida de datos de diferente tipo: `fprintf ()` y `fscanf ()` 208
7.2.5. Fin de archivo: `feof ()` y `rewind ()` 209
7.3. Archivos binarios en C 210
7.3.1. Escribir y leer un buffer, funciones `fwrite ()` y `fread ()` 210
7.3.2. Funciones para acceso aleatorio 212
7.4. Ordenación externa: métodos de ordenación 213
7.5. Método de mezcla directa (mezcla simple) 214
7.5.1. Codificación del algoritmo de mezcla directa 215
7.6. Fusión natural 219
7.6.1. Algoritmo de la fusión natural 220
7.7. Mezcla equilibrada múltiple 221
7.7.1. Algoritmo de la mezcla equilibrada múltiple 222
7.7.2. Tipos de datos para la mezcla equilibrada múltiple 222
7.7.3. Cambio de finalidad de un archivo: *entrada* ↔ *salida*. 222
7.7.4. Control del número de tramos 223
7.7.5. Codificación del algoritmo de mezcla equilibrada múltiple 223
7.8. Método polifásico de ordenación externa 229
7.8.1. Mezcla polifásica con `m = 3` archivos 229
7.8.2. Distribución inicial de tramos 231
7.8.3. Algoritmo de la mezcla 233
7.8.4. Mezcla polifásica *versus* mezcla múltiple 234
RESUMEN 235
EJERCICIOS 235
PROBLEMAS 236

PARTE III
ESTRUCTURAS DE DATOS FUNDAMENTALES (ABSTRACCIÓN DE DATOS, LISTAS, PILAS, COLAS Y TABLAS HASH

Capítulo 8. Tipos abstractos de datos y objetos **239**

8.1. El papel (el rol) de la abstracción 240
8.1.1. La abstracción como un proceso natural mental 240
8.1.2. Historia de la abstracción del software 240
8.1.3. Procedimientos 241
8.1.4. Módulos 242
8.1.5. Tipos abstractos de datos 242
8.1.6. Objetos 243
8.2. Modularidad 243
8.2.1. La estructura de un módulo 244
8.2.2. Reglas de modulación 244
8.3. Diseño de módulos 246
8.3.1. Acoplamiento de módulos 246
8.3.2. Cohesión de módulos 247
8.4. Tipos de datos 248
8.5. Abstracción en lenguajes de programación 249
8.5.1. Abstracciones de control 249
8.5.2. Abstracciones de datos 250
8.6. Tipos abstractos de datos 251
8.6.1. Ventajas de los tipos abstractos de datos 252
8.6.2. Implementación de los TAD 252
8.7. Especificación de los TAD 253
8.7.1. Especificación informal de un TAD 253
8.7.2. Especificación formal de un TAD 254
8.8. Tipos abstractos de datos en C 256
8.8.1. Implementación del TAD `conjunto` en C 257
8.8.2. Aplicación del tipo abstracto de dato `conjunto` 259
8.9. Orientación a objetos 261
8.9.1. Abstracción 262
8.9.2. Encapsulación 262
8.9.3. Modularidad 263
8.9.4. Jerarquía 263
8.9.5. Polimorfismo 263
8.9.6. Otras propiedades 264
8.9.7. Lenguajes de programación orientados a objetos 265
8.10. Reutilización de software 265
8.11. Desarrollo tradicional frente al orientado a objetos 267
RESUMEN 268
EJERCICIOS 269
PROBLEMAS 270

Capítulo 9. Listas, listas enlazadas **271**

9.1. Fundamentos teóricos de listas enlazadas 272
9.2. Clasificación de las listas enlazadas 273
9.3. Tipo abstracto de datos (TAD) lista 274
9.3.1. Especificación formal del TAD Lista 274
9.4. Operaciones en listas enlazadas 275
9.4.1. Declaración de un nodo 275
9.4.2. Puntero de cabecera y cola 276

9.4.3. El puntero nulo 277
9.4.4. El operador → de selección de un miembro 278
9.4.5. Construcción de una lista 279
9.5. Inserción de un elemento en una lista 281
9.5.1. Insertar un nuevo elemento en la cabeza de una lista 281
9.5.2. Inserción de un nodo al final de la lista 284
9.5.3. Inserción de un nodo entre dos nodos de la lista 285
9.6. Búsqueda en listas enlazadas 287
9.7. Eliminación de un nodo en una lista 289
9.8. Lista doblemente enlazada 292
9.8.1. Declaración de una lista doblemente enlazada 293
9.9. Insertar un elemento en una lista doblemente enlazada 294
9.9.1. Insertar un nuevo elemento en la cabeza de una lista doble 294
9.9.2. Inserción de un nuevo nodo que no está en la cabeza de lista 294
9.10. Eliminación de un elemento en una lista doblemente enlazada 295
9.11. Listas circulares 299
9.11.1. Insertar un elemento en una lista circular 300
9.11.2. Eliminar un elemento en una lista circular 300
RESUMEN 306
EJERCICIOS 306
PROBLEMAS 307

Capítulo 10. Pilas y sus aplicaciones 311

10.1. Concepto de pila 312
10.1.1. Especificaciones del tipo abstracto de datos Pila 313
10.2. El tipo `Pila` implementado con arrays 313
10.2.1. Especificación del tipo abstracto `Pila` 315
10.2.2. Implementación de las operaciones sobre pilas 317
10.2.3. Operaciones de verificación del estado de la pila 319
10.3. El tipo `Pila` implementado como una lista enlazada 321
10.3.1. Tipos de datos para una pila con listas enlazadas 321
10.3.2. Implementación de las operaciones de pilas con listas enlazadas 322
10.4. Evaluación de expresiones aritméticas mediante pilas 325
10.4.1. Notación prefija y notación postfija de una expresión aritmética 325
10.4.2. Evaluación de una expresión aritmética 327
RESUMEN 335
EJERCICIOS 336
PROBLEMAS 336

Capítulo 11. Colas 339

11.1. Concepto de cola 340
11.1.1. Especificaciones del tipo abstracto de datos Cola 340
11.2. Tipo Cola implementada con arrays 341
11.2.1. *TAD Cola* con un array lineal 342
11.3. Definición del *TAD Cola* con arrays circulares 345
11.3.1. Codificación del *TAD Cola* con un array circular 347
11.4. Realización del *TAD Cola* con una lista enlazada 350
11.4.1. Declaración de tipos de datos y operaciones 351
11.4.2. Codificación de las funciones del *TAD Cola* con listas 351
11.4.3. Realización del *TAD Cola* con una lista circular 353
11.5. Bicolas: colas de doble entrada 356
11.5.1. Representación de una bicola con listas dobles 357
RESUMEN 363
EJERCICIOS 363
PROBLEMAS 364

Capítulo 12. Colas de prioridades y montículos **367**

12.1. Cola de prioridades 368
12.1.1. Declaración del *TAD Cola de prioridad* 368
12.2. Construcción de una cola de prioridades 368
12.2.1. Definición del *TAD Cola de prioridades* 369
12.3. Montículos 371
12.3.1. Definición de montículo 371
12.3.2. Representación de un montículo 373
12.3.3. Propiedad de ordenación: condición de montículo 373
12.3.4. Operaciones definidas en un montículo 374
12.3.5. Operación `insertar` 374
12.3.6. Operación `buscar mínimo` 376
12.3.7. Operación `eliminar mínimo` 377
12.4. Ordenación por montículos (*HeapSort*) 379
12.4.1. Algoritmo 380
12.4.2. Codificación 381
12.4.3. Análisis del algoritmo de ordenación por montículos 383
RESUMEN 384
EJERCICIOS 384
PROBLEMAS 385

Capítulo 13. Tablas de dispersión, funciones hash **387**

13.1. Tablas de dispersión 388
13.1.1. Definición de una tabla de dispersión 388
13.1.2. Operaciones de una tabla de dispersión 389
13.2. Funciones de dispersión 390
13.2.1. Aritmética modular 391
13.2.2. Plegamiento 391
13.2.3. Mitad del cuadrado 392
13.2.4. Método de la multiplicación 393
13.3. Colisiones y resolución de colisiones 395
13.4. Exploración de direcciones 396
13.4.1. Exploración lineal 397
13.4.2. Exploración cuadrática 398
13.4.3. Doble dirección dispersa 399
13.5. Realización de una tabla dispersa 399
13.5.1. Creación de una tabla dispersa 400
13.5.2. Posición de un elemento 400
13.5.3. Insertar un elemento en la tabla 401
13.5.4. Búsqueda de un elemento 402
13.5.5. Dar de baja un elemento 402
13.6. Direccionamiento enlazado 403
13.6.1. Operaciones de la tabla dispersa enlazada 403
13.6.2. Análisis del direccionamiento enlazado 404
13.7. Realización de una tabla dispersa encadenada 404
13.7.1. Dar de alta un elemento en una tabla dispersa encadenada 406
13.7.2. Eliminar un elemento de la tabla dispersa encadenada 406
13.7.3. Buscar un elemento de la tabla dispersa encadenada 407
RESUMEN 408
EJERCICIOS 408
PROBLEMAS 409

PARTE IV
ESTRUCTURAS DE DATOS AVANZADAS (ÁRBOLES Y GRAFOS)

Capítulo 14. Árboles. Árboles binarios y árboles ordenados **415**
14.1. Arboles generales 416
14.1.1. Representación gráfica de un árbol 419
14.1.2. Resumen de definiciones 420
14.2. Arboles binarios. TAD árbol binario 421
14.2.1. Equilibrio 422
14.2.2. Árboles binarios completos 423
14.2.3. Especificación del TAD árbol binario 425
14.3. Estructura de un árbol binario 425
14.3.1. Representación de un nodo 426
14.3.2. Creación de un árbol binario 427
14.4. Operaciones en árboles binarios 428
14.5. Árboles de expresión 429
14.5.1. Reglas para la construcción de árboles de expresión 431
14.6. Recorrido de un árbol 432
14.6.1. Recorrido *preorden* 433
14.6.2. Recorrido *enorden* 434
14.6.3. Recorrido *postorden* 435
14.6.4. Profundidad de un árbol binario 437
14.6.5. Evolución de un árbol de expresión 438
14.7. Árbol binario de búsqueda 439
14.7.1. Creación de un árbol de búsqueda 439
14.7.2. Nodo de un árbol binario de búsqueda 441
14.8. Operaciones en árboles binarios de búsqueda 442
14.8.1. Búsqueda 442
14.8.2. Insertar un nodo 443
14.8.3. Función `insertar ()` 444
14.8.4. Eliminación 445
14.8.5. Función `eliminar ()` 446
RESUMEN 449
EJERCICIOS 450
PROBLEMAS 451

Capítulo 15. Árboles equilibrados de búsqueda **455**
15.1. Eficiencia de la búsqueda en un árbol binario ordenado 456
15.2. Árbol binario equilibrado, árboles AVL 456
15.2.1. Altura de un árbol equilibrado, árbol AVL 457
15.3. Inserción en árboles de búsqueda equilibrados. Rotaciones 461
15.3.1. Proceso de inserción de un nuevo nodo 462
15.3.2. Rotación simple 463
15.3.3. Movimientos de punteros en la rotación simple 464
15.3.4. Rotación doble 465
15.3.5. Movimiento de punteros en la rotación doble 466
15.4. Realización en C de la inserción con balanceo y rotaciones 467
15.5. Eliminación de un nodo en un árbol equilibrado 472
15.5.1. Algoritmo para eliminar un nodo de un árbol equilibrado 473
15.5.2. Realización en C de la operación de eliminación en un árbol equilibrado 475
RESUMEN 478
EJERCICIOS 479
PROBLEMAS 479

Capítulo 16. Árboles B **481**
16.1. Definición de un árbol B 482

16.2. *TAD árbol B* y su representación 483
16.2.1. Representación de un nodo de árbol B 483
16.2.2. Tipo abstracto árbol B 484
16.3. Proceso de formación de un árbol B 485
16.3.1. Creación de un árbol B de orden 5 485
16.4. Búsqueda de una clave en un árbol B 487
16.4.1. Función `buscarNodo ()` 487
16.4.2. Función `buscar ()` 488
16.5. Operación de inserción en un árbol B 488
16.5.1. Función `insertar ()` 489
16.5.2. Función `empujar ()` 489
16.5.3. Función `meterHoja ()` 491
16.5.4. Función `dividirNodo ()` 491
16.6. Eliminación de una clave en un árbol B 492
16.7. Operación de eliminación de una clave en un árbol B 494
16.7.1. Función `eliminar ()` 495
16.7.2. Función `eliminarRegistro ()` 496
16.7.3. Función `quitar ()` 497
16.7.4. Función `sucesor ()` 497
16.7.5. Función `restablecer ()` 497
16.7.6. Funciones `moverDrcha ()` y `moverIzqda ()` 498
16.7.7. Función `combina ()` 499
16.8. Listado de las claves de un árbol B 500
RESUMEN 503
EJERCICIOS 503
PROBLEMAS 505

Capítulo 17. Grafos I: representación y operaciones 507

17.1. Conceptos y definiciones 508
17.1.1. Grado de entrada, grado de salida de un nodo 509
17.1.2. Camino 509
17.2. Representación de los grafos 510
17.2.1. Matriz de adyacencia 510
17.2.2. Listas de adyacencia 512
17.3. *TAD grafo* 514
17.3.1. Realización de las operaciones con listas de adyacencia 515
17.4. Recorrido de un grafo 517
17.4.1. Recorrido en anchura 517
17.4.2. Recorrido en profundidad 520
17.5. Conexiones en un grafo 522
17.5.1. Componentes conexas de un grafo 522
17.5.2. Componentes fuertemente conexas de un grafo 523
17.6. Matriz de caminos: cierre transitivo 526
17.6.1. Implementación en C 528
17.6.2. Matriz de caminos y cierre transitivo 529
17.7. Puntos de articulación de un grafo 530
17.7.1. Búsqueda de los puntos de articulación 530
17.7.2. Implementación en C 532
RESUMEN 533
EJERCICIOS 534
PROBLEMAS 537

Capítulo 18. Grafos II: algoritmos 539

18.1. Ordenación topológica 540
18.1.1. Algoritmo para obtener una ordenación topológica 541
18.1.2. Implementación en C del algoritmo de ordenación topológica 541

18.2. Matriz de caminos: algoritmo de Warshall 543
18.2.1. Implementación del algoritmo de Warshall 545
18.3. Caminos mínimos con un solo origen: algoritmo de Dijkstra 546
18.3.1. Algoritmo de Dijkstra 546
18.3.2. Codificación del algoritmo de Dijkstra 548
18.4. Caminos mínimos para todo vértice: algoritmo de Floyd 549
18.4.1. Codificación del algoritmo de Floyd 552
18.4.2. Recuperación de los vértices del camino 552
18.5. Árbol de expansión de coste mínimo 553
18.5.1. Algoritmo de Prim 554
18.5.2. Codificación del algoritmo de Prim 556
18.5.3. Algoritmo de Kruskal 557
18.5.4. Codificación del algoritmo de Kruskal 558
18.6. Grafos para representar redes de flujo 560
18.6.1. Planteamiento del problema 561
18.6.2. Formulación matemática 562
18.6.3. Algoritmo del aumento del flujo: algoritmo de Ford-Fulkerson 562
18.6.4. Ejemplo de mejora de flujo 564
18.6.5. Pasos que sigue el algoritmo 565
18.6.6. Algoritmo Ford-Fulkerson en seudocódigo 567
RESUMEN 568
EJERCICIOS 569
PROBLEMAS 572

APÉNDICES

A. Lenguaje algorítmico UPSAM versión 2.0 ..... **577**

B. Guía de sintaxis C **595**

C. Palabras reservadas de C y C++ **623**

BIBLIOGRAFÍA **647**

ÍNDICE **649**

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, Algoritmos y estructuras de datos. El estudio de estructuras de datos y de algoritmos es tan antiguo como el nacimiento de la programación y se ha convertido en estudio obligatorio en todos los currículos desde finales de los años setenta 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.

¿Por qué 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, 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 decisión del 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, incluso 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, orientada a la programación en Web, etc.

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 complejo. En este caso el paradigma orientado a objetos es una herramienta de programación y organizativa muy poderosa, 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

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

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 son menos importantes ni dejan de tener gran incidencia en la formación en esta materia. Por ejemplo, algunas de las funciones de Entrada/Salida (tan importantes en programación) son más fáciles en C++ que en C (véase el caso de números enteros); otros tipos de datos tales como cadenas y números reales se pueden formatear más fácilmente en C. Otro factor importante para los principiantes es el conjunto de mensajes de error y advertencias proporcionadas por un compilador durante el desarrollo del programa.

Este libro no es un libro clásico de programación[2] sino más bien la continuación natural de estos estudios introductorios. Se estudian estructuras de datos con un objetivo fundamental: aprender a escribir programas más eficientes. También cabe aquí hacerse la pregunta de 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 las 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 tienen 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 inversas, 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 muchas técnicas de representación de datos. El contexto las mismas se engloba 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

[2] Los lectores interesados en este nivel tienen a su disposición numerosos y excelentes obras, algunas de las cuales podrá consultar en el apéndice final de bibliografía. En nuestro caso, puede consultar, si lo desea, nuestra obra *Fundamentos de programación*, 3.ª edición (McGraw-Hill, 2003).

para adaptarse 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. ej., 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 puedan manejar grandes cantidades de entradas se vuelve más crítica y su eficiencia aumenta a medida que estos programas puedan manipular más y mejores organizaciones de datos. Analizando un algoritmo antes de que se codifique realmente, los estudiantes pueden decidir si una determinada solución será factible y rigurosa. Por ejemplo, se pueden ver cómo diseños e implementaciones cuidadas pueden reducir los costes en tiempo y memoria de algoritmos. Por esta razón, se dedican dos capítulos, 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 texto de referencia universitaria y profesional

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

En el caso de España, los actuales planes de estudios y los futuros, contemplados en la Declaración de Bolonia, 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 currículos de carreras de Ingeniería de Sistemas y Licenciaturas en Informática. ACM, la organización profesional norteamericana más prestigiosa del mundo, incluye en las recomendaciones de sus diferentes currículos

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 el último currículo publicado, *Computing Curricula* 2001, se incluyen en el área **PF** de Fundamentos de Programación (Programming Fundamentals, PF1 a PF4), **AL** de Algoritmos y Complejidad (*Algorithms and Complexity*, AL1 a AL3). En este libro se ha incluido los descriptores más importantes tales como Algoritmos y Resolución de Problemas, Estructuras de datos fundamentales, Recursión, Análisis de algoritmos básicos y estrategias de algoritmos. Además se incluyen un estudio de algoritmos de estructuras discretas tan importantes como Árboles y Grafos.

Organización del libro

Este libro, pese a su amplitud, está concebido como libro de texto y no como una enciclopedia o libro de consulta, aunque dado el carácter didáctico que hemos pretendido y por nuestra experiencia en este campo, podría servir para estas últimas tareas pero una vez terminado el aprendizaje y formación básica en estructuras de datos, objetivo fundamental de la obra, bien en cursos avanzados o bien en la actividad profesional de programación.

Se pretende enseñar los principios 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 consiguiente, hemos diseñado esta obra como un libro de texto, para cursos reglados o pensando en autodidactas que ya tienen formación básica de programación, y por ello contempla la mayoría de los algoritmos y estructuras de datos más notables y sobresalientes, a la par que los más utilizados. Los lectores deben tener conocimientos a nivel de iniciación o nivel medio en programación. Es deseable haber cursado al menos un curso de un semestre de introducción a los algoritmos y a la programación, con ayuda de alguna herramienta de programación, preferentemente pseudocódigo; y se sacará el mayor rendimiento si además se tiene conocimiento de un lenguaje estructurado tal como Pascal o C. De cualquier forma y pensando en los lectores que puedan tener conocimientos básicos de estas materias, se incluyen los anexos, una guía de sintaxis de un lenguaje algorítmico (UPSAM 2.0) basado en pseudocódigo, junto con una guía de sintaxis del lenguaje C y una guía de palabras reservadas en C/C++, de modo que pueda servir al lector para consulta o como referencia.

El libro contiene gran número de ejercicios y problemas resueltos, implementados en pseudocódigo (lenguaje algorítmico en «español» de sintaxis similar a C) aunque en su gran mayoría todos están posteriormente implementados en lenguaje C. Normalmente los programas contienen una o varias líneas de comentarios que pretenden ayudar al entendimiento y seguimiento de los mismos.

La estructura de cada capítulo contiene una lista de palabras clave (reservadas) relativas al tema central del capítulo, una propuesta de ejercicios junto con la otra propuesta de problemas. La diferencia, en general, reside en el nivel de complejidad de la cuestión propuesta a resolver. Se pretende que el lector practique y adquiera destrezas en el diseño y programación de algoritmos y estructuras de datos preferentemente implementando problemas correspondientes a situaciones reales dadas.

El libro está pensado para un curso completo anual o bien dos semestres y una carga lectiva de 4/6 horas semanales, incluyendo teoría, prácticas en cuaderno y en computadora con compiladores en C. Se ha estructurado el contenido en cuatro partes para facilitar al profesor/maestro la selección que considere más oportuna y a los alumnos/lectores tener disponibles los algoritmos y estructuras de datos básicos y avanzados más empleados en el mundo de la programación y en el de base de datos e ingeniería de software. La estructuración del libro permite al lector seguir cualquier capítulo una vez que ha leído y comprendido los cuatro primeros capítulos, siempre que disponga de una formación básica en programación y en C. Caso de no tener estos conocimientos o bien desea refrescar los que ya tiene, le recomendamos comience leyendo los Apéndices A y B, donde incluimos unos cursos básicos a modo de guía de sintaxis tanto del lenguaje algorítmico UPSAM 2.0 como de C y palabras reservadas en C.

Contenido

El contenido del libro sigue los programas clásicos de las disciplinas Estructura de Datos y/o Estructuras de Datos y de la Información respetando las directrices emanadas de los currículos de 1991 y las actualizadas del 2001 de ACM/IEEE, así como de los planes de estudio de Ingeniero Informático e Ingenieros Técnicos en Informática de España y los de Ingenieros de Sistemas y Licenciados en Informática de muchas universidades latinoamericanas.

La Parte I: *Análisis de algoritmos y estructuras de datos básicas* describe el importante concepto de análisis de un algoritmo y las diferentes formas de medir su complejidad y eficiencia; así mismo se describen las estructuras de datos más simples tales como arrays, cadenas o estructuras. La Parte II: *Diseño de algoritmos (recursividad, ordenación y búsqueda)* examina los algoritmos más utilizados en la construcción de cualquier programa tales como los relativos a búsqueda y ordenación, así como las potentes técnicas de manipulación de la recursividad. La Parte III: *Estructuras de datos lineales (abstracción de datos, listas, pilas, colas y tablas hash)* constituyen una de las partes avanzadas del libro y que suele formar parte de cursos de nivel medio/alto en organización de datos. Por último, la Parte IV: *Estructuras de datos avanzadas (árboles y grafos)* constituyen también una de las partes avanzadas del libro; su conocimiento y manipulación permitirán al programador obtener el máximo aprovechamiento en el diseño y construcción de sus programas. La descripción más detallada de los capítulos correspondientes se reseñan a continuación.

Capítulo 1. 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) 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.

Capítulo 4. Estructuras y uniones. 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 conforman 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 5. 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 más conocidos para resolver problemas de naturaleza recursiva. Entre ellos se destacan: *las Torres de Hanoi, Backtraking, Salto del Caballo, las Ocho Reinas* o *el Problema de la Selección Óptima.* Se analizan en detalle los algoritmos y sus correspondientes codificaciones en C. En algún caso se utilizan conceptos que se explicarán con más detalles en los capítulos de grafos y árboles, pero se han utilizado en este capítulo para guardar consistencia con el título de este capítulo y que todos los conceptos importantes se describieran juntos, aunque en algún caso pueden ser necesarios conceptos de otros capítulos, que por otra parte se señalan explícitamente.

Capítulo 6. Algoritmos de ordenación y búsqueda. La ordenación y la búsqueda, como ya se ha comentado anteriormente, son dos de las operaciones más frecuentemente utilizadas en programa-

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 currículos de 1991 y las actualizadas del 2001 de ACM/IEEE, así como de los planes de estudio de Ingeniero Informático e Ingenieros Técnicos en Informática de España y los de Ingenieros de Sistemas y Licenciados en Informática de muchas universidades latinoamericanas.

La Parte I: *Análisis de algoritmos y estructuras de datos básicas* describe el importante concepto de análisis de un algoritmo y las diferentes formas de medir su complejidad y eficiencia; así mismo se describen las estructuras de datos más simples tales como arrays, cadenas o estructuras. La Parte II: *Diseño de algoritmos (recursividad, ordenación y búsqueda)* examina los algoritmos más utilizados en la construcción de cualquier programa tales como los relativos a búsqueda y ordenación, así como las potentes técnicas de manipulación de la recursividad. La Parte III: *Estructuras de datos lineales (abstracción de datos, listas, pilas, colas y tablas hash)* constituyen una de las partes avanzadas del libro y que suele formar parte de cursos de nivel medio/alto en organización de datos. Por último, la Parte IV: *Estructuras de datos avanzadas (árboles y grafos)* constituyen también una de las partes avanzadas del libro; su conocimiento y manipulación permitirán al programador obtener el máximo aprovechamiento en el diseño y construcción de sus programas. La descripción más detallada de los capítulos correspondientes se reseñan a continuación.

Capítulo 1. 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) 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.

Capítulo 4. Estructuras y uniones. 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 conforman 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 5. 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 más conocidos para resolver problemas de naturaleza recursiva. Entre ellos se destacan: *las Torres de Hanoi, Backtraking, Salto del Caballo, las Ocho Reinas* o *el Problema de la Selección Óptima.* Se analizan en detalle los algoritmos y sus correspondientes codificaciones en C. En algún caso se utilizan conceptos que se explicarán con más detalles en los capítulos de grafos y árboles, pero se han utilizado en este capítulo para guardar consistencia con el título de este capítulo y que todos los conceptos importantes se describieran juntos, aunque en algún caso pueden ser necesarios conceptos de otros capítulos, que por otra parte se señalan explícitamente.

Capítulo 6. Algoritmos de ordenación y búsqueda. La ordenación y la búsqueda, como ya se ha comentado anteriormente, son dos de las operaciones más frecuentemente utilizadas en programa-

ción. Los algoritmos más reconocidos y más eficientes se analizan y describen con detalle en este capítulo. Así, por ejemplo, se describen e implementan en C, algoritmos de ordenación tan populares como *Burbuja, Selección, Inserción, Shell, QuickSort* junto con otros más avanzados y no tan populares como *MergeSort, RadixSort o BinSort.* También se describen los métodos de búsqueda lineal o secuencial y binaria, junto con la búsqueda binaria recursiva.

Capítulo 7. Algoritmos de ordenación de archivos. 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, etc.) constituyen este importante capítulo.

Capítulo 8. Tipos abstractos de datos 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 9. Listas, 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. Las listas doblemente enlazadas y circulares son variantes de las listas enlazadas, también, muy empleadas en el importante campo de la programación.

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. 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.

Capítulo 12. Colas de prioridades y montículos. Un tipo especial de cola, la cola de prioridades, utilizado en situaciones especiales, para resolución de problemas, junto con el concepto de montículo (*heap*, en inglés) se analizan detalladamente, junto con un método de ordenación por montículos muy eficientes, sobre todo en situaciones complejas y difíciles.

Capítulo 13. Tablas de dispersión, funciones hash. Las tablas aleatorias hash junto con los problemas de resolución de colisiones y los diferentes tipos de direccionamiento conforman este capítulo.

Capítulo 14. Árboles. Árboles binarios, árboles ordenados. 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 15. Árboles equilibrados de búsqueda. Un tipo especial de árbol de búsqueda, no por ello menos importante, es el árbol equilibrado de búsqueda. 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 16. 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 17. 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. El conocimiento profundo de la teoría de grafos junto con los algoritmos de implementación es fundamental para conseguir el mayor rendimiento de las operaciones con datos, sobre todo si éstos son complejos en su organización. Un programador de alto nivel no puede dejar de conocer en toda su profundidad la teoría de grafos y sus operaciones

Capítulo 18. 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., 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, etc., o algoritmos clásicos como Prim, Kruskal o Warshall.

Apéndice A. Lenguaje algorítmico UPSAM 2.0. El estudio de algoritmos y su implementación práctica es usual se realice con un lenguaje algorítmico basado en pseudocódigo. En nuestro caso se describe la guía de sintaxis UPSAM 2.0[3], es una versión de lenguaje algorítmico cuyo lenguaje de implementación es el pseudocódigo en español. Este lenguaje y su correspondiente compilador ha sido construido en el Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software de la Facultad de Informática de la Universidad Pontificia de Salamanca. Sus primeras versiones se publicaron en la segunda mitad de los años ochenta y la versión tratada en esta obra se publicó el año 2003.

Apéndice B. Guía de sintaxis C. Descripción de los elementos básicos, tipos de datos, sentencias, funciones, estructuras de datos básicos, etc., se incluyen en este apéndice. Esta guía busca facilitar el aprendizaje de la programación del lector no iniciado en C y proporcionar reglas y recordatorios de sintaxis para los lectores ya iniciados que les ayude en su tarea de aprendizaje y formación.

Apéndice C. Palabras reservadas de C y C++. El buen uso de la sintaxis de las palabras clave (reservadas) de C es vital en el aprendizaje de la programación y posteriormente en el diseño y construcción de algoritmos y programas.

Código C disponible

Los códigos en C de todos los programas importantes de este libro están disponibles en la Web (Internet) —en formato Word para que puedan ser utilizados directamente y evitar su «tecleado» en el caso de los programas largos, o bien simplemente, para seleccionar, recortar, modificar, etc., 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.macgraw-hill.es/ayedenC`.

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*,

[3] La guía de sintaxis fue creada por el profesor Joyanes, pero las últimas versiones han sido obra de un equipo que el citado profesor ha coordinado. Entre los autores de esta guía, además de los autores de este libro se pueden destacar nombres de profesores tales como Luis Rodríguez y Matilde Fernández y muchos otros que han colaborado tales como Víctor Martín, Joaquín Abeger, Antonio Muñoz, etc. (en *Fundamentos de programación*, 3.ª ed. McGraw-Hill, 2003) se enumeran todos los profesores que han contribuido a la creación del lenguaje. En lo relativo al compilador, la profesora María Luisa Díez Plata y el profesor Enrique Torres Fernández del mismo departamento y facultad, lideran el equipo de trabajo que han desarrollado el compilador UPSAM 2.0 utilizado en prácticas de programación y de procesadores de lenguajes.

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 nuestro/as colegas —y sin embargo amigo/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 anónimos de universidades e institutos tecnológicos y politécnicos de Latinoamérica que siempre apoyan nuestras obras y a los que desgraciadamente no podemos agradecer individualmente; al menos que conste en este humilde homenaje, nuestro eterno agradecimiento y reconocimiento por ese cariño que siempre prestan a nuestros obras. Como saben aquellos que nos conocen siempre estamos a su disposición en la medida que, físicamente, nos es posible. Gracias a todos, esta obra es posible, en un porcentaje muy alto, por vuestra ayuda y colaboración.

Y como no, a los estudiantes, a los lectores autodidactas y no autodidactas, que siguen nuestras obras. Su apoyo es un gran acicate para seguir nuestra 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 Concepción Fernández (Concha), coordinadora editorial de McGraw-Hill, nuestra gran amiga y editora desde «ha muchos años», a Amelia Nieta de McGraw-Hill, que en esta edición me ha asesorado y aconsejado en todo momento. Simplemente, gracias.

Madrid, febrero de 2004

PARTE I

ANÁLISIS DE ALGORITMOS Y ESTRUCTURAS DE DATOS BÁSICAS

1

ALGORITMOS, ESTRUCTURAS DE DATOS Y PROGRAMAS

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Revisar los conceptos básicos de tipos de datos.
- Introducirse en las ideas fundamentales de estructuras de datos.
- Comprender el significado de un Tipo Abstracto de Datos (TAD) y un modelo representativo.
- Revisar el concepto de algoritmo y programa.
- Conocer y entender la utilización de la herramienta de programación conocida por «pseudocódigo».
- Entender los conceptos de análisis, verificación y eficiencia de un algoritmo.

CONTENIDO

1.1. Tipos de datos.
1.2. La necesidad de las estructuras de datos.
1.3. Abstracción y tipos abstractos de datos.
1.4. Algoritmos y programas.
1.5. Representación de los algoritmos: el pseudocódigo.
1.6. Eficiencia y exactitud.

RESUMEN

EJERCICIOS

CONCEPTOS CLAVE

- Tipo de dato.
- Estructura de datos.
- Algoritmo.
- Eficiencia.
- Pseudocódigo.
- Tipo Abstracto de Datos (TAD).
- Precondición y poscondición.
- Invariante.

INTRODUCCIÓN

La representación de la información es fundamental en ciencias de la computación y en informática. El propósito principal de la mayoría de los programas de computadora 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 datos, abstracción, algoritmos y programas, así como los primeros criterios relativos a análisis y eficiencia de algoritmos.

1.1. TIPOS DE DATOS

Los lenguajes de programación tradicionales, tales como Pascal y C, proporcionan *tipos de datos* para clasificar diversas clases de datos. La ventajas de utilizar tipos en el desarrollo de software [TREMBLAY 2003] son:

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

Ejemplo 1.1

Detección y prevención de errores mediante lenguajes tipificados (tipeados) tales como Pascal y ALGOL, en determinadas situaciones.

La expresión aritmética

```
6 + 5 + "Ana la niña limeña"
```

Contiene un uso inapropiado del literal cadena. Es decir, se ha utilizado un objeto de un cierto tipo (datos enteros) en un contexto inapropiado (datos tipo cadena de caracteres) y se producirá un error con toda seguridad en el momento de la compilación y naturalmente, en consecuencia, en la ejecución del programa del cual forme parte.

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

Definición 1: Un *tipo de dato* es un conjunto de valores y operaciones asociadas a esos valores.

Definición 2: Un *tipo de dato* consta de dos partes: un conjunto de datos y las operaciones que se pueden realizar sobre esos datos.

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

1.1.1. Tipos primitivos de datos

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

Para recordar

Los tipos de datos primitivos son tipos cuyos valores son elementos de datos indivisibles. Incluyen tipos numéricos, carácter y lógicos (*booleanos*).

Ejemplo 1.2

El número 0810 es un tipo primitivo de dato ya que si bien se puede descomponer en cuatro cifras o dígitos, éstas ya no tienen el mismo significado.

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

Tipo de dato atómico

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

Ejemplo 1.3

Diferentes tipos de datos atómicos:

Enteros:

```
valores        -, ....., -3, -2, -1, 0, 1, 2, 3, .....,
operaciones    *, +, -, /, %,++,--, ...
```

Coma flotante:

```
valores        -, ....., 0.0, .....
operaciones    *, +, -, %, /, ...
```

Carácter:

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

Lógico:

```
valores        verdadero, falso
operaciones    and, or, not,... (y, o, no)
```

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

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

El tipo de dato ***boolean*** **(lógico)** suele considerarse como el tipo más simple debido a que sólo tiene dos valores posibles: `verdadero` (*true*) y `falso` (*false*). El formato sintáctico de estas constantes lógicas puede variar de un lenguaje de programación a otra. Algunos lenguajes ofrecen un conjunto más rico de operaciones booleanas que otros. Por ejemplo, algunos lenguajes tienen

construcciones que permiten a los programadores especificar operaciones condicionales o en corto-circuito. Por ejemplo, en Java en los operadores booleanos `&&` y `||`, sólo se evalúa el segundo operando si su valor se necesita para determinar el valor de la expresión global.

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

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

Los códigos de caracteres más utilizados son: **EBCDIC** (utilizado por las primeras máquinas de IBM) y **ASCII** (el código universal más extendido). La aparición del lenguaje Java trajo consigo la representación de caracteres en **Unicode**, un conjunto internacional de caracteres que unifica a muchos conjuntos de caracteres, incluyendo inglés, español, italiano, griego, francés, alemán, latín, hebreo o indio. En Java, el tipo `char` se representa como un valor de 16 bits. Por consiguiente, el rango de `char` está entre `0` y `65.536`, lo que permite representar prácticamente la totalidad de los alfabetos y formatos numéricos y de puntuación universales. En C, el tipo `char` se representa con 8 bits, por lo que el rango es de 0 a 255 que se corresponde con la secuencia de caracteres ASCII.

Una generalización simple de los tipos anteriores es el tipo **enumerado.** Este tipo se introdujo con el lenguaje Pascal y consta de un número finito de valores. Este conjunto se especifica por un listado de valores con un límite inferior y otro superior, un rango de valores. Los tipos enteros, carácter y boolean son *tipos enumerados predefinidos*. Sin embargo, los tipos enumerados son algo más. Un tipo enumerado puede representar cualquier lista finita de nombres o etiquetas únicos que se utilizan con frecuencia en sentencias condicionales o de decisión tales como `case` o `switch`, disponibles en Pascal, C o Java; sin embargo, Java no soporta tipos de datos enumerados.

1.1.2. Tipos de datos compuestos y agregados

Los **datos compuestos** son el tipo opuesto a los tipos de datos atómicos. Los datos compuestos se pueden romper en subcampos que tengan significado. Un ejemplo sencillo es el número de su teléfono celular `51199110101`. Realmente, este número consta de varios campos, el código del país `(51, Perú)`, el código del área `(1, Lima)` y el número propiamente dicho (suponiendo que los teléfonos celulares y/o fijos de línea fuesen de 8 dígitos).

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

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

```
array de enteros: [4, 6, 8, 35, 46, 0810]
```

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

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

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

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

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

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

1.2. LA NECESIDAD DE LAS ESTRUCTURAS DE DATOS

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

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

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

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

Estructura de datos

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

La Tabla 1.1 recoge las definiciones de dos estructuras de datos clásicas: arrays y registros.

Tabla 1.1. Ejemplos de estructuras de datos

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

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

Las estructuras de datos requieren espacio suficiente para almacenar colecciones de datos y, posteriormente se requiere realizar operaciones sobre esos datos, tales como búsqueda del dato mayor, del dato central, imprimir datos, ordenar los elementos en un orden determinado, etc. Eso significa que será necesario que la elección de una estructura de datos adecuada permita la posibilidad de poder realizar operaciones sobre dichas estructuras.

Los programas se deben escribir para resolver problemas, pero la elección de la estructura de datos adecuadas redundará en una mayor eficiencia del programa y, sobre todo, en una mejor resolución del problema en cuestión. Una elección inadecuada de la estructura de datos puede conducir a programas lentos, largos y poco eficientes. Por el contrario, la adopción de una representación compleja para mejorar un programa puede conducir a una complejidad innecesaria ya que podría haber sido suficiente un diseño sencillo.

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

El **coste** de una solución es la cantidad de recursos que la solución consume. Normalmente, el coste se mide en término de recursos clave, especialmente el tiempo.

1.2.1. Etapas en la selección de una estructura de datos

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

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

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

Las restricciones de recursos sobre ciertas operaciones clave tales como búsqueda, inserción de registros de datos y eliminación de registros de datos, normalmente conducen el proceso de selección de las estructuras de datos.

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

- ¿Todos los datos se insertan en la estructura de datos al principio o se entremezclan con otras operaciones?
- ¿Se pueden eliminar los datos?
- ¿Los datos se procesan en un orden bien definido o se permite el acceso aleatorio?

Resumen

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

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

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

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

1.3. ABSTRACCIÓN Y TIPOS ABSTRACTOS DE DATOS

Se suele decir que la «ciencia informática» o «ciencias de la computación» (*computer sciences*) es la ciencia de la abstracción. Pero, ¿qué es exactamente la abstracción? Se puede pensar en el tamaño de un objeto sin conocer cómo está constituido ese objeto. De modo similar se puede saber conducir un automóvil sin conocer el tipo de modelo o cómo está fabricado. La abstracción se utiliza para suprimir detalles irrelevantes, mientras se enfatiza en los relevantes o significativos.

El beneficio principal de la abstracción es que facilita al programador pensar acerca del problema a resolver. Uno de los principios importantes del diseño de software es el de la abstracción y ocultación de la información. La abstracción procedimental ha estado soportada, virtualmente, en todos los lenguajes de programación, incluyendo los más antiguos como FORTRAN y COBOL. Durante la década de los años setenta, se introdujo el concepto de tipo abstracto de dato. La idea básica que subyace dentro de un tipo abstracto de datos es la separación del uso del tipo de datos de su implementación (las operaciones que actúan sobre los valores de esos datos). Aparece el concepto de especificación del tipo de dato que puede expresarse de modo independiente de su implementación.

El uso del paradigma de programación orientada a objetos para desarrollar software, además de su uso en la implementación de software, requiere que los desarrolladores traten con objetos en todas las fases del ciclo de vida del software. El objeto se representa en un primer nivel por los tipos abstractos de datos, TAD (*Abstract Data Type*, ADT).

Un tipo abstracto de dato define un tipo de dato aisladamente en términos de un tipo y un conjunto de operaciones sobre ese tipo. Cada operación se define por sus entradas y sus salidas. La definición de un TAD no especifica cómo se implementan el tipo de dato. Estos detalles están ocultos en el uso del TAD. El proceso de ocultación de los detalles de la implementación se conoce como **encapsulación.** Una estructura de datos, en esencia, es la implementación física de un TAD.

En resumen, con un TAD, los usuarios no están preocupados en *cómo* se hace la tarea sino *con qué* o *cómo* se puede hacer esa tarea. En otras palabras, el TAD consta de un conjunto de definiciones que permiten a los programadores utilizar las funciones mientras ocultan las implementaciones. Esta generalización de las operaciones con implementaciones no especificadas se conoce como **abstracción.** Se abstraen la esencia del proceso y se dejan ocultos los detalles de la implementación.

Nota

El concepto de abstracción significa:

1. Se conoce lo que un tipo de dato *puede hacer*.
2. *El cómo* se hace está oculto.

Una estructura de datos es la implementación física de un TAD. Cada operación asociada con el TAD se implementa por una o más subrutinas. El término «estructura de datos» se refiere, frecuentemente, a los datos almacenados en la memoria principal de la computadora, mientras que el término *estructura archivo* se refiere, normalmente, a la organización de los datos en un almacenamiento periférico, tal como una unidad de cinta, disco, CD, DVD o similar.

Definamos formalmente un TAD. Un **tipo abstracto de datos** es una declaración de datos empaquetada junto con las operaciones que son significativas para el tipo de dato. En otras palabras, encapsulamos los datos y las operaciones sobre esos datos y ocultamos estos detalles a la vista del usuario.

En los lenguajes orientados a objetos, las **clases** son las soluciones ideales a los TAD. En los lenguajes que no soportan orientación a objetos como Pascal y C será necesario recurrir a las declaraciones y especificaciones de modo convencional como aquí se está explicando.

Tipo abstracto de dato

1. Declaración del dato.
2. Declaración de las operaciones.
3. Encapsulación (encapsulamiento) de datos y operaciones.

1.3.1. Composición de un TAD

Un **TAD** consta de dos partes: *especificación* e *implementación* (Tabla 1.2).

Tabla 1.2. Tipo abstracto de datos

Tipo abstracto de datos			
Especificación		**Implementación**	
Sintaxis (signatura)	Semántica	Representación	Algoritmo

La parte de especificación, sintaxis y semántica, describe *qué* hace el TAD, idealmente especificada, independientemente de *cómo* lo hace. La sección de implementación, representación y algoritmos, define una posible implementación de un TAD. Desde la perspectiva del cliente no es necesario ni deseable conocer la implementación de un TAD sino *qué* es lo que hace el TAD. La idea principal que subyace en la palabra **abstracto** de **tipo abstracto de datos** es la separación de la especificación del tipo de dato de su implementación.

La especificación de un TAD consta de dos partes: sintaxis y semántica. La *sintaxis* de un TAD especifica los formatos de sus operaciones, denominada también *signatura*. La sintaxis de una operación consta de su nombre, número y tipo de sus operandos, y el tipo de valor devuelto por la operación. La *semántica* de un TAD especifica para cada operación cuál es la salida que debe

producirse para cada posible entrada. También, cualquier valor de entrada para los que una operación que no esté definida deba ser especificada.

La implementación de un TAD consta de una representación y sus algoritmos asociados. Una representación especifica como los valores TAD se almacenan en memoria. Los algoritmos especifican cómo las operaciones de un TAD se implementan basándose en las representaciones elegidas. Un TAD puede tener varias implementaciones, cada una de las cuales tiene sus propios requerimientos de tiempo y espacio. Las implementaciones apropiadas para utilizar en un contexto dado se determina por las prestaciones y por los tipos o volúmenes de entrada de datos.

La separación del uso de un TAD de su implementación ayuda también a los desarrolladores en la producción de software fiable y correcto. Ocultando los detalles de la implementación de un TAD inaccesibles a sus usuarios, los errores que se deban al mal diseño de los algoritmos se pueden evitar. Este tipo de protección de la información se conoce, como ya se ha comentado anteriormente, como: *ocultación de la información* o *encapsulamiento.*

1.3.2. Modelo de tipo abstracto de dato

Un modelo típico de TAD se muestra en la Figura 1.1 [GILFOR 2001][2]. El área sombreada irregular representa el modelo. Dentro del área abstracta se representan dos aspectos diferentes del modelo: la estructura de datos y las funciones operaciones. Ambas están totalmente contenidas en el modelo y están fuera del ámbito del usuario. Sin embargo, la estructura de datos está disponible a todas las operaciones que necesite el TAD, y una operación puede llamar a otras funciones para ejecutar su tarea. En otras palabras, las estructuras de datos y las funciones están mutuamente dentro del mismo ámbito.

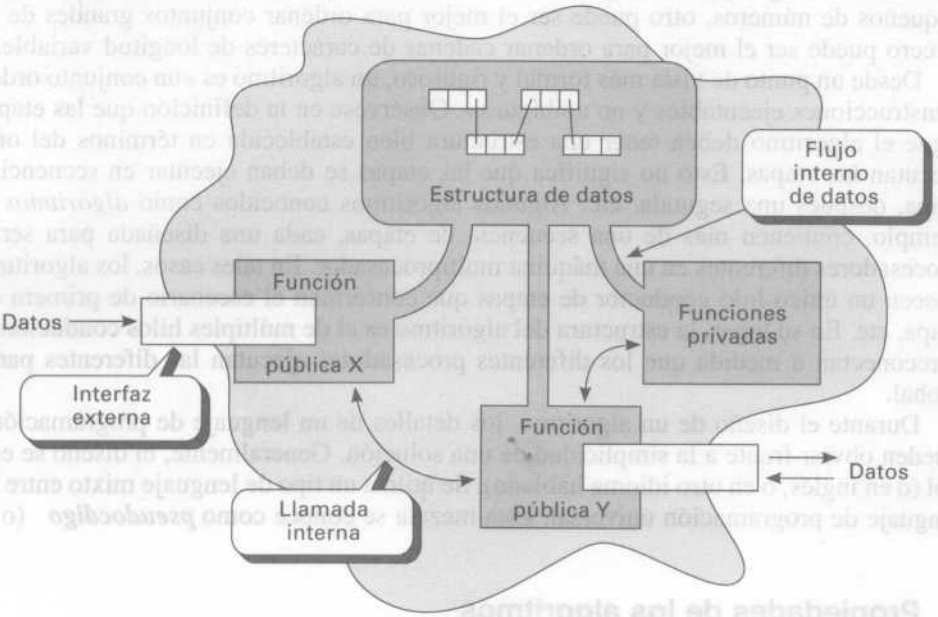


Figura 1.1. Modelo de tipo abstracto de dato (Gilfor, 2001).

Los datos se introducen, acceden, se modifican y se suprimen mediante los interfaces operaciones dibujados como rectángulos fuera de la estructura. Para cada cabecera de la operación hay un

[2] [GILFOR 2001] Richard F. Gilberg y Behrouz A. Forouzan. *Data Structures. A pseudo o de approach with C++.* VIA: Brooks/Cole, 2001, pp. 11-13. Excelente libro, cuya lectura y consulta recomendamos.

algoritmo que ejecuta la operación específica. Sólo el nombre de la operación y sus parámetros son visibles al usuario y proporcionan el único interfaz al TAD. Se pueden crear operaciones adicionales para cumplir requisitos específicos.

Ejemplo 1.4

Operaciones que debe incluir, como mínimo, un TAD que represente una lista de enteros.

- Insertar un nuevo entero al final de la lista.
- Visualizar o imprimir los enteros en orden a medida que aparecen en la lista.
- Devolver *verdadero* si un entero determinado está en la lista.
- Eliminar un entero de una posición determinada de la lista.

1.4. ALGORITMOS Y PROGRAMAS

Un **algoritmo** es un método con un conjunto de instrucciones utilizadas para resolver un problema específico. Un problema puede ser resuelto mediante varios algoritmos. Un algoritmo dado correcto resuelve un problema definido y determinado (p. ej., calcular una función determinada). En este libro se explican muchos algoritmos y para algunos problemas se proponen diferentes algoritmos, como es el caso del problema típico de ordenación de listas.

La ventaja de conocer varias soluciones a un problema es que las diferentes soluciones pueden ser más eficientes para variaciones específicas del problema, o para diferentes entradas del mismo problema. Por ejemplo, un algoritmo de ordenación puede ser el mejor para ordenar conjuntos pequeños de números, otro puede ser el mejor para ordenar conjuntos grandes de números, y un tercero puede ser el mejor para ordenar cadenas de caracteres de longitud variable.

Desde un punto de vista más formal y riguroso, un algoritmo es «un conjunto ordenado de pasos o instrucciones ejecutables y no ambiguas». Obsérvese en la definición que las etapas o pasos que sigue el algoritmo deben tener una estructura bien establecida en términos del orden en que se ejecutan las etapas. Esto no significa que las etapas se deban ejecutar en secuencia, una primera etapa, después una segunda, etc. Algunos algoritmos conocidos como *algoritmos paralelos,* por ejemplo, contienen más de una secuencia de etapas, cada una diseñada para ser ejecutada por procesadores diferentes en una máquina multiprocesador. En tales casos, los algoritmos globales no poseen un único hilo conductor de etapas que conforman el escenario de primera etapa, segunda etapa, etc. En su lugar, la estructura del algoritmo es el de múltiples hilos conductores que bifurcan y reconectan a medida que los diferentes procesadores ejecutan las diferentes partes de la tarea global.

Durante el diseño de un algoritmo, los detalles de un lenguaje de programación específico se pueden obviar frente a la simplicidad de una solución. Generalmente, el diseño se escribe en español (o en inglés, o en otro idioma hablado). Se utiliza un tipo de lenguaje mixto entre el español y un lenguaje de programación universal. Esta mezcla se conoce como ***pseudocdigo*** (o *seudocódigo*).

1.4.1. Propiedades de los algoritmos

Un algoritmo debe cumplir diferentes propiedades[3]:

1. *Especificación precisa de la entrada.* La forma más común del algoritmos es una transformación que toma un conjunto de valores de entrada y ejecuta algunas manipulaciones para

[3] En [BUDD 1998], [JOYANES 2003], [BROOKSHEAR 2003] y [TREMBLAY 2003] más las referencias incluidas al final del libro en la bibliografía, puede encontrar ampliación sobre teoría y práctica de algoritmos.

producir un conjunto de valores de salida. Un algoritmo debe dejar claros el número y tipo de valores de entrada y las condiciones iniciales que deben cumplir esos valores de entrada para conseguir que las operaciones tengan éxito.

2. *Especificación precisa de cada instrucción.* Cada etapa de un algoritmo debe ser definida con precisión. Esto significa que no puede haber *ambigüedad* sobre las acciones que se deban ejecutar en cada momento. Las ambigüedades de los idiomas propios de los algoritmos (pseudocódigo en inglés, español, etc.) no pueden afectar al algoritmo en sí.
3. *Exactitud, corrección.* Un algoritmo debe ser *exacto, correcto.* Se debe poder demostrar que el algoritmo resuelve el problema. Con frecuencia, esto se plasma en el formato de un argumento, lógico o matemático, al efecto de que ***si*** las condiciones de entrada se cumplen y se ejecutan los pasos del algoritmo ***entonces*** se producirá la salida deseada. En otras palabras, se debe calcular la función deseada, convirtiendo cada entrada a la salida correcta. Un algoritmo se espera resuelva un problema.
4. *Etapas bien definidas y concretas.* Un algoritmo se compone de una serie de *etapas concretas.* Concreta significa que la acción descrita por esa etapa está totalmente comprendida por la persona o máquina que debe ejecutar el algoritmo. Cada etapa debe ser ejecutable en una cantidad finita de tiempo. Por consiguiente, el algoritmo nos proporciona una «receta» para resolver el problema en etapas y tiempos concretos.
5. *Número finito de pasos.* Un algoritmo se debe componer de un número *finito* de pasos. Si la descripción del algoritmo se compone de un número infinito de etapas, nunca se podrá implementar como un programa de computador. La mayoría de los lenguajes que describen algoritmos (español, inglés o pseudocógio) proporcionan un método para ejecutar acciones repetidas, conocidas como iteraciones que controlan las salidas de bucles o secuencias repetitivas.
6. *Un algoritmo debe terminar.* En otras palabras, no puede entrar en un bucle infinito.
7. *Descripción del resultado o efecto.* Por último, debe estar claro cuál es la tarea que el algoritmo debe ejecutar. La mayoría de las veces, esta condición se expresa con la producción de un valor como resultado que tenga ciertas propiedades. Con menor frecuencia, los algoritmos se ejecutan para un *efecto lateral*, tal como imprimir un valor en un dispositivo de salida. *En cualquier caso la salida esperada debe estar especificada completamente.*

Ejemplo 1.5

¿Es un algoritmo la siguiente instrucción?

```
Escribir una lista de todos los enteros positivos
```

Es imposible ejecutar la instrucción anterior dado que hay infinitos enteros positivos. Por consiguiente, cualquier conjunto de instrucciones que implique esta instrucción no es un algoritmo.

1.4.2. Programas

Normalmente, se considera que un programa de computadora es una representación concreta de un algoritmo en un lenguaje de programación. En este libro casi todos los algoritmos están representados en términos de programas, o como partes de programas. Naturalmente, hay muchos programas *que son ejemplos del mismo algoritmo, dado que cualquier lenguaje de programación moderno se* puede utilizar para implementar cualquier algoritmo (aunque algunos lenguajes de programación facilitarán su tarea al programador mejor que otros). Para simplificar la presentación a través del texto, nosotros utilizamos de modo intercambiable los conceptos de «programa» y «algoritmo», pese a que como ya hemos comentado son conceptos totalmente distintos. Por definición un algorit-

mo debe proporcionar suficiente detalle para que se pueda convertir en un programa cuando se necesite.

El requisito de que un algoritmo «debe terminar» significa que no todos los programas de computadora son algoritmos. Su sistema operativo es un programa en tal sentido. Sin embargo, si piensa en las diferentes tareas de un sistema operativo (cada una con sus entradas y salidas asociadas) como problemas individuales, cada una resuelta por un algoritmo específico implementado por una parte del programa sistema operativo, cada una de las cuales termina cuando se produce su correspondiente salida.

Para recordar

1. Un problema es una función o asociación de entradas con salidas.
2. Un algoritmo es una receta para resolver un problema cuyas etapas son concretas y no ambiguas.
3. El algoritmo debe ser correcto, finito y debe terminar para todas las entradas.
4. Un programa es una «ejecución» (*instanciación*) de un algoritmo en un lenguaje de programación de computadora.

El diseño de un algoritmo para ser implementado por un programa de computadora debe tener dos características principales:

1. Fácil de entender, codificar y depurar.
2. Capaz de conseguir la mayor eficiencia a los recursos de la computadora.

Idealmente, el programa resultante debería ser el más eficiente. ¿Cómo medir la eficiencia de un algoritmo o programa? El método correspondiente se denomina *análisis de algoritmos* y permite medir la dificultad inherente a un problema. En este capítulo se introducirá el concepto, y se desarrollará y profundizará en el siguiente y en posteriores capítulos.

1.5. REPRESENTACIÓN DE LOS ALGORITMOS: EL PSEUDOCÓDIGO

Aunque desde los años sesenta se han definido numerosas herramientas para representar algoritmos (diagramas de flujo, diagramas N-S, lenguaje natural, lenguajes de iconos, etc.) uno de los más utilizados es el **pseudocódigo**[4]. El pseudocódigo (en su origen, nació como una representación textual en inglés para definir algoritmos) es una representación en el idioma elegido del código requerido por un algoritmo. Sin embargo, desde mediados de la década de los ochenta, numerosas universidades españolas, entre ellas la nuestra (Pontificia de Salamanca, España), comenzaron a impartir las clases de algoritmos mediante unas primitivas versiones de pseudocódigo en español. Estas versiones fueron evolucionando y mejorando, y si bien existen diferentes modelos, nosotros lógicamente seguimos la versión realizada en nuestra universidad y que en estos momentos se encuentra en la versión 2.0 y se conoce con el nombre de **UPSAM 2.0.**

En realidad el pseudocódigo es mitad sintaxis en español (o en inglés, si sigue este idioma por su semejanza a la sintaxis de los lenguajes de programación) y mitad reglas de codificación. En esencia, la parte de código consta de una versión ampliada de las construcciones algorítmicas básicas —secuencia, selección e iteración— junto con construcciones de tipos de datos, funciones y procedimientos, estructuras de datos, etc.

[4] En [JOYANES 2003 *a*, *b*] podrá encontrar amplia información sobre el pseudocódigo junto con una gran cantidad de ejercicios y problemas de programación resueltos con esta técnica algorítmica.

A tener en cuenta

El pseudocódigo es una de las herramientas utilizadas en la definición de algoritmos. Se compone de una parte en español y de otra parte de código estructurado.

El pseudocódigo es *un lenguaje de especificación de algoritmos*. El uso de tal lenguaje hace el paso de codificación final (esto es, la traducción a un lenguaje de programación) relativamente fácil. El lenguaje APL se utiliza a veces como un lenguaje de especificación de algoritmos.

El pseudocódigo nació como un lenguaje similar al inglés y era un medio de representar básicamente las estructuras de control de programación estructurada que se verán en capítulos posteriores. Se considera un *primer borrador*, dado que el pseudocódigo tiene que traducirse posteriormente a un lenguaje de programación. El pseudocódigo no puede ser ejecutado por una computadora. La *ventaja del pseudocódigo* es que en su uso en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico. Es también fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica del programa, mientras que en muchas ocasiones suele ser difícil el cambio en la lógica una vez que está codificado en un lenguaje de programación. Otra ventaja del pseudocódigo es que puede ser traducido fácilmente a lenguajes como Pascal, COBOL, C, FORTRAN 77, C++, Java o C#.

El pseudocódigo utiliza palabras reservadas en inglés para representar las acciones sucesivas —similares a sus homónimas en los lenguajes de programación—, tales como **start, end, stop, if-then-else, while-end, repeat-until,** etc. La escritura en pseudocódigo exige normalmente la *indentación* o *sangrado* (sangría en el margen izquierdo) de diferentes líneas.

La representación en pseudocódigo de un algoritmo que calcule el salario de un trabajador contratado por horas es la siguiente:

```
start
   {cálculo de impuestos y salarios}
   read nombre, horas, precioHora
   salarioBruto ← horas * precioHora
   tasas ← 0.25 * salarioBruto
   salarioNeto ← salarioBruto - tasas
   write nombre, salarioBruto, tasas, salarioNeto
end
```

El algoritmo comienza con la palabra **start** y finaliza con la palabra **end,** en inglés (en español, **inicio, fin**). Entre estas palabras, sólo se escribe una instrucción o acción por línea.

La línea encerrada entre llaves (`{...}`) se denomina *comentario*. Es una información al lector del programa y no realiza ninguna instrucción ejecutable, sólo tiene efecto de documentación interna del programa. Algunos autores suelen utilizar corchetes en lugar de llaves (`[...]`).

No es recomendable el uso de apóstrofos o simples comillas, como representaban algunos lenguajes de programación primitivos de los comentarios, ya que este carácter es representativo de apertura o cierre de cadenas de caracteres en lenguajes como Pascal o FORTRAN, y daría lugar a confusión.

Otro ejemplo aclaratorio en el uso del pseudocódigo podría ser un sencillo algoritmo del arranque matinal de un coche clásico (antiguo).

```
inicio
   {arranque matinal de un coche}
   introducir la llave de contacto
   tirar del estrangulador del aire
```

```
        girar la llave de contacto
        pisar el acelerador
        oír el ruido del motor
        pisar de nuevo el acelerador
        esperar unos instantes a que se caliente el motor
        llevar el estrangulador de aire a su posición
fin
```

Por fortuna, aunque el pseudocódigo nació como un sustituto del lenguaje de programación y, por consiguiente, sus palabras reservadas se conservaron o fueron muy similares a las de dichos lenguajes, prácticamente el inglés, el uso del pseudocódigo se ha extendido en la comunidad hispana con términos en español como `inicio`, `fin`, `parada`, `leer`, `escribir`, `si-entonces-sino`, `mientras`, `fin_mientras`, `repetir`, `hasta_que`, etc. Sin duda, el uso de la terminología del pseudocódigo en español ha facilitado y facilitará considerablemente el aprendizaje y uso diario de la programación. En esta obra, al igual que en otras nuestras[5], utilizaremos el pseudocódigo en español y daremos en su momento las estructuras equivalentes en inglés, al objeto de facilitar la traducción del pseudocódigo al lenguaje de programación seleccionado.

Así, pues, en los pseudocódigos citados anteriormente deberían ser sustituidas la palabras `start`, `end`, `read`, `write`, por `inicio`, `fin`, `leer`, `escribir` respectivamente.

Ejemplo 1.6

Calcular el valor de la suma 1 + 2 + 3 + ... + 100.

```
Algoritmo
```

Se utiliza una variable `Contador` como un contador que genere los sucesivos números enteros, y `Suma` para almacenar las sumas parciales 1, 1 + 2, 1 + 2 + 3, ...

```
1.  Establecer Contador a 1
2.  Establecer Suma a 0
3.  mientras Contador ≤ 100 hacer
        Sumar Contador a Suma
        Incrementar Contador en 1
    fin_mientras
4.  Visualizar Suma
```

1.5.1. Codificación de un programa

Codificación es la escritura en un lenguaje de programación de la representación del algoritmo desarrollada en las etapas precedentes. Dado que el diseño de un algoritmo es independiente del lenguaje de programación utilizado para su implementación, el código puede ser escrito con igual facilidad en un lenguaje o en otro.

Para realizar la *conversión* del algoritmo en programa se deben sustituir las palabras reservadas en español por sus homónimos en inglés, y las operaciones/instrucciones indicadas en lenguaje natural expresarlas en el lenguaje de programación correspondiente.

---

[5] Joyanes, *Metodología de la Programación*. McGraw-Hill, 1986.

```
/*
 Este programa obtiene una tabla de depreciaciones acumuladas y
 valores reales de cada año de un determinado producto
*/
#include <stdio.h>
void main()
{
 double coste, depreciacion,
        valorRecuperacion,
        valorActual,
        acumulado,
        valorAnual;
 int anio, vidaUtil;

 puts("Introduzca coste, valor recuperación y vida útil");
 scanf("%lf %lf %lf",&coste,&valorRecuperacion,&vidaUtil);
 puts("Introduzca año actual");
 scanf("%d",&anio);
 valorActual = coste;
 depreciacion = (coste - valorRecuperacion)/vidaUtil;
 acumulado = 0;
 puts("Año Depreciación Dep. Acumulada");
 while (anio < vidaUtil)
 {
  acumulado = acumulado + depreciacion;
  valorActual = valorActual - depreciacion;
  printf("Anio: %d, Depreciacion:%.2lf, %.2lf Acumulada\n",
         anio, depreciacion, acumulado);
  anio = anio + 1;
 }
}
```

1.6. EFICIENCIA Y EXACTITUD

De las características antes analizadas que deben cumplir los algoritmos destacan dos por su importancia en el desarrollo de algoritmos y en la construcción de programas: *eficiencia* y *exactitud*, que se examinarán y utilizarán amplia y profusamente en los siguientes capítulos.

Existen numerosos enfoques a la hora de resolver un problema. ¿Cómo elegir el más adecuado entre ellos? Entre las líneas de acción fundamentales en el diseño de computadoras se suelen plantear dos objetivos (a veces conflictivos y contradictorios entre sí) [SHAFFER 1997]:

1. Diseñar un algoritmo que sea fácil de entender, codificar y depurar.
2. Diseñar un algoritmo que haga un uso eficiente de los recursos de la computadora.

Idealmente el programa resultante debe cumplir ambos objetivos. En estos casos, se suele decir que tal programa es «elegante». Aunque los algoritmos y programas escritos en este libro buscan cumplir esta propiedad, la misma no es uno de los objetivos fundamentales del mismo. Más bien, se puede decir, esta tarea cae dentro de la disciplina de Ingeniería de Software. Por el contrario, entre los objetivos centrales de este libro, sí que está la medida de la eficiencia de algoritmo, así como su diseño correcto o exacto.

1.6.1. Eficiencia de un algoritmo

Raramente existe un único algoritmo para resolver un problema determinado. Cuando se comparan dos algoritmos diferentes que resuelven el mismo problema, normalmente, se encontrará que un algoritmo es un orden de magnitud más eficiente que el otro. En este sentido lo importante, es que el programador sea capaz de reconocer y elegir el algoritmo más eficiente.

Entonces, ¿qué es eficiencia? La ***eficiencia*** de un algoritmo es la propiedad mediante la cual un algoritmo debe alcanzar la solución al problema en el tiempo más corto posible y/o utilizando la cantidad más pequeña posible de recursos físicos, y que sea compatible con su exactitud o corrección. Un buen programador buscará el algoritmo más eficiente dentro del conjunto de aquellos que resuelven con exactitud un problema dado.

¿Cómo medir la eficiencia de un algoritmo o programa informático? Uno de los métodos más sobresalientes es el ***análisis de algoritmos***, cuyos fundamentos se estudian en el siguiente capítulo y se desarrollan a lo largo de todo el libro. El análisis de algoritmos permite también medir la dificultad inherente de un problema. Los restantes capítulos utilizan con frecuencia la técnica de análisis de algoritmos siempre que éstos se diseñan. Esta característica le permitirá comparar algoritmos para la resolución de problemas en términos de eficiencia.

Aunque las máquinas actuales son capaces de ejecutar millones de instrucciones por segundo, la eficiencia permanece como un reto o preocupación a resolver. Con frecuencia, la elección entre algoritmos eficientes e ineficientes pueden mostrar la diferencia entre una solución práctica a un problema y una no práctica. En los primeros tiempos de la informática moderna (de los años sesenta a los ochenta) las computadoras eran muy lentas y tenían pequeña capacidad de memoria. Los programas tenían que ser diseñados cuidadosamente para hacer uso de los recursos escasos, tales como almacenamiento y tiempo de ejecución. Los programadores gastaban horas intentando recortar radicalmente segundos a los tiempos de ejecución de sus programas o intentando comprimir los programas en un pequeño espacio en memoria utilizando todo tipo de tecnologías de comprensión y reducción de tamaño. La eficiencia de un programa se medía en aquella época como un factor dependiente del binomio *espacio-tiempo*.

Hoy, la situación ha cambiado radicalmente. Los costes del hardware han caído drásticamente mientras que los costes humanos han aumentado considerablemente. El tiempo de ejecución y el espacio de memoria ya no son factores críticos como lo fueron anteriormente. Hoy día el esfuerzo considerable que se requería para conseguir la eficiencia máxima, no es tan acusado, excepto en algunas aplicaciones, como por ejemplo, aplicaciones de sistemas en tiempo real con factores críticos de ejecución. Pese a todo, la eficiencia sigue siendo un factor decisivo en el diseño de algoritmos y posterior construcción de programas.

Existen diferentes métodos con los que se trata de medir la eficiencia de los algoritmos; entre ellos, están los que se basan en el número de operaciones que debe efectuar un algoritmo para realizar una tarea; otros métodos se centran en tratar de medir el tiempo que se emplea en llevar a cabo una determinada tarea ya que lo importante al usuario final es que ésta se efectúe de forma correcta y en el menor tiempo posible. Sin embargo, estos métodos presentan varias dificultades ya que cuando se trata de generalizar la medida hecha, ésta depende de factores como la máquina en que se efectuó, el ambiente del procesamiento y el tamaño de la muestra, entre otros factores.

Brassard y Bratley acuñaron, en 1988, el término **algoritmia** (*algorithmics*)[6] que definía como «el estudio sistemático de las técnicas fundamentales utilizadas para diseñar y analizar algoritmos eficientes». Este estudio fue ampliado posteriormente en 1997[7] con la consideración de que la determinación de la eficiencia de un algoritmo se podía expresar en el tiempo requerido para realizar la tarea en función del tamaño de la muestra e independiente del ambiente en que se efectúe.

---

[6] Giles Brassard y Paul Bratley. *Algorithmics. Theory and Practice.* Englewood Cliffs, N. N.: Prentice-Hall, 1988.

[7] *Fundamental of algorithmics* (Prentice-Hall, 1997). Este libro fue traducido al español, y publicado también en Prentice-Hall (España), por un equipo de profesores de la Universidad Pontificia de Salamanca, dirigidos por el co-autor de este libro, profesor Luis Joyanes.

El estudio de la eficiencia de los algoritmos se centra fundamentalmente en el análisis de la ejecución de bucles ya que en el caso de funciones lineales —no contienen bucles— la eficiencia es función del número de instrucciones que contiene. En este caso su eficiencia depende de la velocidad de las computadoras y generalmente no es un factor decisivo en la eficiencia global de un programa.

Debido a los avances en hardware y software, los creadores de programas que se ejecutan sólo una vez con pequeñas cantidades de datos no necesitan estar preocupados con consideraciones de eficiencia. En tales casos, el programa más simple se debe formular, depurar y ejecutar con los datos de entrada y descartar otras consideraciones. Por el contrario, al crear programas que se ejecutan muchas veces a lo largo de su vida y/o tienen grandes cantidades de datos de entrada, las consideraciones de eficiencia, no se pueden descartar. Además, existen en la actualidad un gran número de aplicaciones informáticas que requieren características especiales de hardware y software en las cuales los criterios de eficiencia deben ser siempre tenidas en cuenta.

Por otra parte, las consideraciones espacio-tiempo se han ampliado con los nuevos avances en tecnologías de hardware de computadoras. Las consideraciones de espacio implican hoy diversos tipos de memoria: principal, caché, *flash*, archivos, discos duros USB, y otros formatos especializados. Así mismo con el uso creciente de redes de computadoras de alta velocidad, existen muchas consideraciones de eficiencia a tener en cuenta en entornos de informática o computación distribuida.

Nota

La eficiencia como factor espacio-tiempo debe estar estrechamente relacionada con la buena calidad, el funcionamiento y la facilidad de mantenimiento del programa.

1.6.2. Formato general de la eficiencia

En general, el formato se puede expresar mediante una función:

$f(n) = eficiencia$

Es decir, la eficiencia del algoritmo se examina como una función del número de elementos a ser procesados.

Bucles lineales

En los bucles se repiten las sentecias del *cuerpo del bucle* un número determinado de veces, que determina la eficiencia del mismo. Normalmente en los algoritmos los bucles son el término dominante en cuanto a la eficiencia del mismo.

Ejemplo 1.7

¿Cuántas veces se repite el cuerpo del bucle en el siguiente código?

```
1 i = 1
2 iterar (i ≤ n)
   1 código de la aplicación
   2 i = i + 1
3 fin_iterar
```

Si *n* es un entero, por ejemplo de valor 100, la respuesta es 100 veces. El número de iteraciones es directamente proporcional al factor del bucle, `n`. Como la eficiencia es directamente proporcional al número de iteraciones la función que expresa la eficiencia es:

```
f(n) = n
```

Ejemplo 1.8

¿Cuántas veces se repite el cuerpo del bucle en el siguiente código?

```
1 i = 1
2 iterar (i ≤ n)
    1 código de la aplicación
    2 i = i + 2
3 fin_iterar
```

La respuesta no siempre es tan evidente como en el ejercicio anterior. Ahora el contador `i` avanza de 2 en 2, por lo que la respuesta es `n/2`. En este caso el factor de eficiencia es:

```
f(n) = n/2
```

Bucles algorítmicos

Consideremos un bucle en el que su variable de control se multiplique o divida dentro de dicho bucle. ¿Cuántas veces se repetirá el cuerpo del bucle en los siguientes segmentos de programa?

```
1 i = 1
2 mientras (i < 1000)
    { código de la aplicación }
    i = i * 2
3 fin_mientras
```

```
1 i = 1000
2 mientras (i ≥ 1)
    { código aplicación }
    i = i/2
3 fin_mientras
```

La Tabla 1.3 contiene las diferentes iteraciones y los valores de la variable `i`.

Tabla 1.3. Análisis de los bucles de multiplicació y divisió

Bucle de multiplicar		Bucle de dividir	
Iteración	Valor de i	Iteración	Valor de i
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
salida	1024	*salida*	0

En ambos bucles se ejecutan 10 iteraciones. La razón es que en cada iteración el valor de `i` se dobla en el bucle de multiplicar y se divide por la mitad en el bucle de división. Por consiguiente, el número de iteraciones es una función del multiplicador o divisor, en este caso 2.

Bucle de multiplicar	$2^{\text{iteraciones}} < 1000$
Bucle de división	$1000/2^{\text{iteraciones}} \geq 1$

Generalizando el análisis, se puede decir que las iteraciones de los bucles especificados se determinan por la siguiente fórmula:

$$f(n) = [\log_2 n]$$

Bucles anidados

En el caso de bucles anidados (bucles que contienen a otros bucles) se determinan cuántas iteraciones contiene cada bucle. El total es, entonces, el producto del número de iteraciones del bucle interno y el número de iteraciones del bucle externo.

```
iteraciones : iteraciones del bucle externo × iteraciones bucle interno
```

Existen tres tipos de bucles anidados: *lineal logarítmico, dependiente cuadrático y cuadrático,* que con ejemplos y análisis similares a los anteriores nos conducen a ecuaciones de eficiencia contempladas en la Tabla 1.4.

Tabla 1.4. Fórmulas de eficiencia

Lineal logarítmica	$f(n) = [n \log_2 n]$
Dependiente cuadrática	$f(n) = \dfrac{n(n+1)}{2}$
Cuadrática	$f(n) = n^2$

Notación O grande

La alta velocidad de las computadoras actuales (frecuencias del procesador iguales o superiores a 3 GHz ya son usuales en computadoras comerciales) hace que la medida exacta de la eficiencia de un algoritmo no sea una preocupación vital en el diseño, pero sí el orden general de magnitud de la misma. Si el análisis de dos algoritmos muestra que uno ejecuta 25 iteraciones mientras que otro ejecuta 40, la práctica muestra que ambos son muy rápidos, entonces ¿cómo se valoran las diferencias? Por otra parte, si un algoritmo realiza 25 iteraciones y otro 2.500 iteraciones, entonces debemos estar preocupados, por la rápidez o la lentitud de uno u otro.

Se ha comentado anteriormente que el número de sentencias ejecutadas en la función para n datos es función del número de elementos y se expresa por la fórmula $f(n)$. Aunque la ecuación derivada para obtener una función puede ser compleja, el factor dominante que se debe considerar para determinar el orden de magnitud del resultado, es el denominado factor de eficiencia. Por consiguiente, no se necesita determinar la medida completa de la eficiencia, basta con calcular el factor que determina la magnitud. Este factor se define como **«*O* grande»**, que representa `"está en el orden de"` y se expresa como $O(n)$, es decir, `"en el orden de n"`.

Para recordar

La expresión eficiencia de un algoritmo se simplifica con la función *O* grande.

Si un algoritmo es cuadrático, se dice entonces que su eficiencia es $O(n^2)$. Esta función expresa cómo crece el tiempo de proceso del algoritmo, de tal forma que una eficiencia $O(n^2)$ muestra que crece con el cuadrado del número de entradas.

Regla

La notación *O* grande se puede obtener de $f(n)$ utilizando los siguientes pasos:

1. En cada término, establecer el coeficiente del término en 1.
2. Mantener el término mayor de la función y descartar los restantes. Los términos se ordenan de menor a mayor:

$$\log_2 n \quad n \quad n \log_2 n \quad n^2 \quad n^3 \quad \ldots \quad n^k \quad 2^n \quad n!$$

Ejemplo 1.9

Calcular la función *O* grande de eficiencia de las siguientes funciones:

a) $f(n) = n\ (n + 1) = 1/2\ n^2 + 1/2\ n$

b) $f(n) = a_j n^k + a_{j-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$

Caso a:

1. Se eliminan todos los coeficientes y se obtiene

   $n^2 + n$

2. Se eliminan los factores más pequeños

   n^2

3. La notación *O* correspondiente es

   $O(f(n)) = O(n^2)$

Caso b:

1. Se eliminan todos los coeficientes y se obtiene

   $f(n) = n^k + n^{k-1} + \ldots + n^2 + n + 1$

2. Se eliminan los factores más pequeños y el término de exponente mayor es el primero

   n^k

3. La notación *O* correspondiente es

   $O(f(n)) = O(n^k)$

Medidas estándar de eficiencia

La eficiencia de los algoritmos se puede clasificar en siete categorías [GILFOR 01][8] que se pueden observar en la Tabla 1.5 y en la Figura 1.2 en orden decreciente de eficiencia.

Tabla 1.5. Medidas de eficiencia[9]

Eficiencia	O grande	Interaciones	Tiempo estimado[a]
Loarítmica	$O(\log_2 n)$	14	Microsegundos
Lineal	$O(n)$	10.000	0,1 segundos
Logarítmica lineal	$O(n(\log_2 n))$	140.000	2 segundos
Cuadrática	$O(n^2)$	10.000^2	15-20 minutos
Polinómica	$O(n^k)$	10.000^k	Horas
Exponencial	$O(c^n)$	$2^{10.000}$	Inmedible
Factorial	$O(n!)$	10.000!	Inmedible

[a] Para n = 10.000.

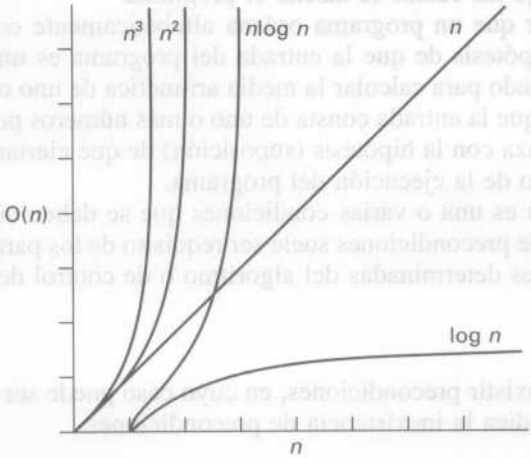


Figura 1.2. Rangos de *O* grande.

Ejemplo 1.10

Considerar la situación en la que se pueda resolver un problema de tres formas posibles: lineal, lineal logarítmica y cuadrática, y evaluar la eficiencia para el caso de un problema que contenga 10.000 elementos.

Según la Tabla 1.5 se observa que en el caso de 10.000 elementos, la eficiencia lineal requiere una fracción de una décima de segundo, mientras que la solución cuadrática requiere 15-20 minutos y la lineal logarítmica 2 segundos.

Verificación

Una de las etapas fundamentales en el desarrollo de un algoritmo es la evaluación de la solución exacta y su potencial como herramienta para la resolución de problemas.

[8] Gilberg y Forouzan (2001), *op. cit.*, pp. 20-21.
[9] Tabla reproducida de [GILFOR 2001], *op. cit.*, p. 21.

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. Si en un algoritmo para la puesta en marcha de un aparato electrónico se omite el paso de verificar el voltaje, evidentemente puede ocurrir una sobrecarga, un cortocircuito, la fundición del fusible de protección, etc., y en consecuencia el aparato no podría ponerse en marcha hasta tanto se repare la avería.

Los algoritmos y, en consecuencia, el software debe verificarse para comprobar su correcto funcionamiento. La verificación del software es una etapa importante en su desarrollo y la búsqueda de técnicas importantes de verificación constituye un campo activo de investigación en informática y en ciencias de la computación.

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 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.

Al igual que una prueba formal matemática se basa en axiomas (las pruebas geométricas se basan normalmente en los axiomas de la geometría euclediana, mientras que otras pruebas se pueden basar en la teoría de conjuntos), una prueba formal de la exactitud de un programa se basa en las especificaciones bajo las cuales se diseñó el programa.

Así, para probar que un programa ordena alfabéticamente con fidelidad (exactitud), se debe comenzar con la hipótesis de que la entrada del programa es una lista de nombres; o bien si el programa está diseñado para calcular la media aritmética de uno o más números positivos, se debe suponer, de hecho, que la entrada consta de uno o más números positivos. En resumen, una prueba de exactitud comienza con la hipótesis (suposición) de que ciertas condiciones *precondiciones,* se cumplen al principio de la ejecución del programa.

La *precondición* es una o varias condiciones que se deben cumplir antes de la ejecución del algoritmo. La lista de precondiciones suele ser requisito de los parámetros de entrada en las funciones o en las variables determinadas del algoritmo o de control de bucles.

Casos

1. Pueden no existir precondiciones, en cuyo caso puede ser de interés insertar una sentencia *nada* que indica la inexistencia de precondiciones.

   ```
   Pre nada
   ```

2. Si existen varios parámetros de entrada, entonces se deben mostrar las precondiciones de cada uno de ellos.

Una *poscondición* es la condición o condiciones que deben cumplir parámetros de salida, determinadas variables o los resultados de salida. Las poscondiciones identifican cualquier acción que se realiza y el estado de todos los parámetros de salida.

Ejemplo 1.11

Escribir la cabecera de un posible algoritmo de búsqueda de un dato en un array de elementos enteros.

```
Algoritmo (val lista: array,
           val argumento:  entero,
           ref posición:   índice)
           // val, parámetro por valor
           // ref, parámetro por referencia
           // búsqueda en el array un elemento determinado
```

```
                // y devuelve la posición o índice en la lista
Pre             lista contiene un array de dato
                argumento, contiene un dato a localizar en la lista
Post            El argumento posición contiene el índice de un elemento
                de igual valor que está en la lista o se visualiza
                «no encontrado» si no existe.
```

El algoritmo de búsqueda tiene dos parámetros que se pasan por valor (*val*) y uno por referencia (*ref*). La precondición especifica que los dos parámetros de entrada, `lista` y `argumento`, se deben inicializar. Si, por ejemplo, la búsqueda utilizada fuera binaria, la precondición sería que el array debería estar ordenado.

Las poscondiciones identifican cualquier acción realizada y el estado de los parámetros de salida. En el algoritmo la poscondición contiene dos partes. Primero, se indica qué se ha de imprimir o visualizar en el informe. Segundo, el parámetro de referencia, en el Ejemplo 1.11 `numPagina` (número de página), contiene el número actualizado de páginas del informe. En el algoritmo de búsqueda sólo hay una poscondición que puede ser unos de dos diferentes valores.

También puede existir una condición de retorno para el caso de que se devuelva un valor. En el caso del algoritmo de búsqueda se devuelve el número de líneas impresas, y un valor lógico `verdadero` si se encontró el elemento buscador, `falso` si no se encontró dicho elemento.

Ejemplo 1.12

El algoritmo TestArchivo lee un archivo e imprime un informe.

```
Algoritmo TestArchivo(ref numPagina: entero)
// El algoritmo lee un archivo y visualiza o imprime un registro

Pre        Inicializar numPagina
Post       Informe impreso. numPagina contiene número de páginas
           del informe
Retorno    Se devuelve número de líneas impresas

Abrir archivo
líneas = 0
numPagina = 0
Mientras (no sea final de archivo)
  Leer archivo
  Si (página completa)
    Avanzar página
    numPágina = numPágina + 1
  escribiir cabecera de página
  fin_si
  escribir línea del informe
  líneas = líneas + 1
fin_mientras
cerrar archivos
Devolver número delíneas
Fin_TestArchivo
```

Aserciones

El siguiente paso en una prueba de exactitud es considerar cómo las consecuencias de las precondiciones se propagan a través del programa. Para verificar la influencia de estas propagaciones y como una prueba más para medir la corrección de un programa, se insertan en diferentes puntos del programa *aserciones* o *asertos*[10]. El resultado es una colección de aserciones, cada una de las cuales es una consecuencia de las precondiciones del programa, y la secuencia de instrucciones que conducen al punto del programa en que está establecida la aserción. Si la aserción establecida al final del programa se corresponde a las especificaciones de salida deseadas se dice que el programa es correcto o fiable.

Ejemplo 1.13

Aserciones asociadas con una estructura típica `mientras (while)`.

La Figura 1.3 representa una estructura típica de bucle **mientras.** Supongamos que como consecuencia de las precondiciones dadas en el punto *A*, se puede establecer que una aserción específica es verdadera cada vez que el test de terminación se realiza (punto *B*) durante el proceso iterativo. Esta aserción en el interior de un bucle se conoce como ***invariante del bucle.*** [GLEEN 03][11] Si la repetición termina siempre, la ejecución se mueve al punto *C*, y se deduce que se debe a contener la invariante del bucle y la condición de terminación.

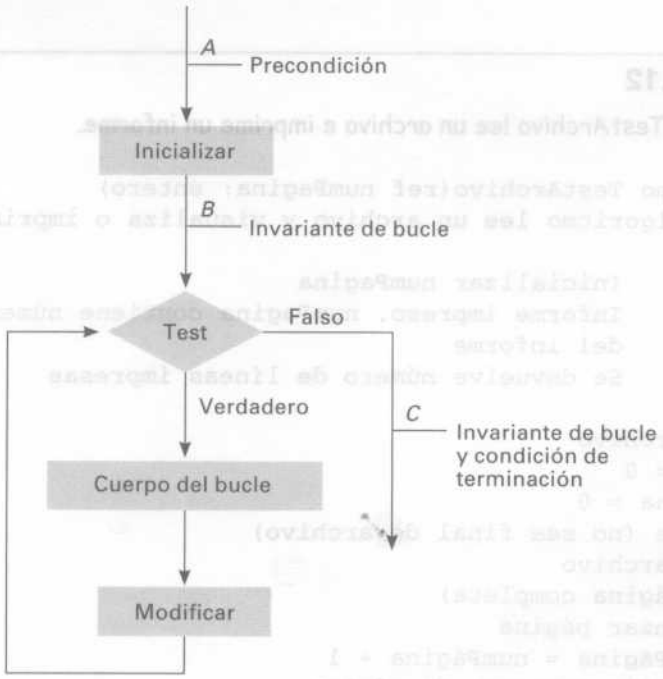


Figura 1.3. Estructura del bucle condicional `Mientras (while)`.

[10] La 22.ª edición del *Diccionario de la Lengua Española* editado por la Real Academia Española (Madrid: Espasa, 2001) define *aserción* como: «Acción y efecto de afirmar o dar por cierto algo» y también «Proposición en que se afirma o da por cierto algo»; *aserto*, se define como «afirmación de la certeza de algo».

[11] J. Glenn Brookeshean. *Computer Science: an overview*, 7.ª edición, Boston: Addison-Wesley, 2003, pp. 195-196.

Las aserciones ayudan a escribir bucles correctos, actividades de programación propensa a errores. Las causas típicas de error son:

- Situaciones que suceden la primera y última ejecución o iteración de un bucle. Posibilidad de que el cuerpo de un bucle no se ejecute nunca.
- Errores por bucles infinitos.
- Errores de condiciones compuestas. Por ejemplo, considere condiciones de salida tal como

```
actual ≤ longTabla && !encontrado
```

Se debe incluir un comprobación para ver qué sucedería si `actual` *nunca se hace mayor que* `longTabla` y lo que sucede si `encontrado` nunca se pone a verdadero dentro del cuerpo del bucle.

Un medio para evitar un bucle infinito es utilizar una *variante de bucle* en cada bucle. Una variante de bucle es una expresión entera que siempre es no negativa y que se decrementa en al menos uno después de cada ejecución del cuerpo del bucle. Dado que la secuencia de los valores de las variantes de bucles es una secuencia monolíticamente decreciente cuyo último elemento es siempre no negativo, esta acción garantiza que un bucle termina.

Regla

La variante del bucle no es una aserción sino una expresión entera.

Una variante de bucle es una expresión entera relacionada con el bucle que debe estar en los siguientes momentos:

- Inmediatamente antes de la primera ejecución del bucle.
- Inmediatamente después de la última iteración del bucle.
- Inmediatamente después de cada una de las iteraciones.

RESUMEN

- Una de las herramientas típicas más utilizadas para definir algoritmo es el **pseudocódigo.**
- El pseudocódigo es una representación en español (o en inglés, brasileiro, etc.) del código requerido para *un algoritmo.*
- Los datos atómicos, son datos simples que no se pueden descomponer.
- Un tipo de dato atómico es un conjunto de datos atómicos con propiedades idénticas.
- Los tipos de datos atómicos se definen por un conjunto de valores y un conjunto de operaciones que actúan sobre esos valores.
- Una estructura de datos es un agregado de datos atómicos y datos compuestos en un conjunto con relaciones bien definidas.
- Un tipo abstracto de datos (**TAD**) es una declaración de datos donde se empaquetan juntos los datos y las operaciones que actúan sobre esos datos.
- La eficiencia de un algoritmo se define generalmente en función del número de elementos a procesar y el tipo de bucle que se va a utilizar.
- Las eficiencias de los diferentes bucles son:

— *Bucle lineal:*	`f(n) = n`
— *Bucle logarítmico:*	`f(n) = log n`
— *Bucle logarítmico lineal:*	`f(n) = n * log n`
— *Bucle cuadrático dependiente*	`f(n) = n(n+1)/2`
— *Bucle cuadrático dependiente*	$f(n) = n^2$
— *Bucle cúbico*	$f(n) = n^3$

EJERCICIOS

1.1. 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
```

Justifique su respuesta.

1.2. El siguiente segmento de programa está diseñado para calcular el producto de dos enteros negativos `x` e `y` por acumulación de la suma de copias de `y` (es decir, 4 por 5 se calcula acumulando la suma de cuatro cinco veces). ¿Es correcto el programa? Justifique su respuesta.

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

1.3. Suponiendo la precondición de que 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 + ... n`.

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

1.4. Reescribir el siguiente segmento de programa utilizando una estructura **`mientras`** en lugar de una estructura **`repetir`**. Asegúrese que la nueva versión imprima los mismos valores que la original.

```
cuenta ← 1;
repetir
  // imprimir el valor asignado a cuenta y cuenta ← cuenta + 1
hasta_que (cuenta = 5)
```

1.5. ¿Qué debe hacer para traducir un bucle *postest* expresado de la forma

```
repetir (...) hasta_que (...)
```

en un bucle postest equivalente expresado en la forma

```
hacer (...) mientras (...)
```

1.6. 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 una vez en la cadena `ababa`.

1.7. Diseñar un algotimo que determine el día de la semana de cualquier fecha desde enero de 1700. Por ejemplo, el 17 de agosto de 2001 era viernes.

1.8. ¿Cuál es la diferencia entre especificación sintáctica y semántica?

1.9. Diseñar un algoritmo para determinar si un número n es primo. (Un número primo sólo puede ser divisible por él mismo y por la unidad.)

1.10. *Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura* (S = $\frac{1}{2}$ Base × Altura).

1.11. Calcular y visualizar la longitud de la circunferencia y el área de un círculo de radio dado.

1.12. Escribir un algoritmo que indique si una palabra leída del teclado es un palíndromo. Un *palíndromo* es una palabra que se lee igual en ambos sentidos, como «radar».

1.13. 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".

1.14. Muchos bancos y cajas de ahorro calculan los intereses de las cantidades depositadas por los clientes diariamente en base a las siguientes premisas. Un capital de 1.000 pesetas, 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 pesetas 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 período de tiempo.

1.15. Calcular la eficiencia de los siguientes algoritmos:

a)
```
i = 1
mientras (i ≤ n)
  j = 1
  mientras (j ≤ n)
    // código fuente
    j = j * 2
  fin_mientras
  i = i + 1
fin_mientras
```

b)
```
i = 1
mientras (i ≤ n)
  j = 1
  mientras (j ≤ i)
    // código fuente
    j = j + 1
  fin_mientras
  i = i + 1
fin_mientras
```

c)
```
i = 1
mientras (i ≤ 10)
  j = 1
  mientras (j ≤ 10)
    // código fuente
    j = j + 1
  fin_mientras
  i = i + 1
fin_mientras
```

2

ANÁLISIS DE ALGORITMOS

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Conocer los factores que intervienen en el tiempo de ejecución.
- Aprender nociones básicas sobre la complejidad de los algoritmos.
- Conocer las propiedades matemáticas de la *notación O*.
- Obtener la complejidad de las estructuras de control de un programa.
- Realizar análisis sencillos relativos a la complejidad de algoritmos.
- Definir la complejidad de las sentencias básicas de todo programa C.

CONTENIDO

2.1. La medida de la eficiencia de algoritmos.
2.2. Notación *O-grande*.
2.3. Complejidad de funciones.
2.4. Inconvenientes de la notación *O-grande*.
2.5. Complejidad de las sentencias de una función C.
2.6. Análisis de algoritmos de ordenación.
2.7. La eficiencia de los algoritmos de búsqueda.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Tiempo de ejecución.
- *Rendimiento de un programa.*
- Complejidad.
- Notación asintótica.
- Operación en memoria interna.
- Operación de entrada/salida.
- Complejidad lineal.
- Complejidad exponencial.
- Sentencias de control.

INTRODUCCIÓN

Este capítulo muestra cómo analizar la eficiencia de algoritmos. Las técnicas matemáticas básicas de análisis de algoritmos son fundamentales para el tratamiento de temas avanzados de computación y permiten formalizar el concepto de eficiencia de un algoritmo. El análisis de algoritmos es una parte muy importante de las ciencias de la computación, de modo que la medida de la eficiencia de un algoritmo es uno de los factores fundamentales en su tratamiento. Es importante analizar los requisitos de tiempo y espacio de un algoritmo y medir su complejidad y eficiencia.

2.1. LA MEDIDA DE LA EFICIENCIA DE ALGORITMOS

El coste de un programa de computadora está constituido por diversos componentes. Algunos componentes implican el coste del tiempo humano —el tiempo de las personas que desarrollan, mantienen y usan el programa—. Otros componentes implican el coste de la ejecución del programa —es decir, la *eficiencia del programa*—, medida por la cantidad de tiempo y espacio de computadora que requiere el programa en su ejecución.

Es difícil realizar un análisis simple de un algoritmo que determine la cantidad exacta de tiempo requerida para ejecutarlo. La primera complicación es que la cantidad exacta de tiempo dependerá de la implementación del algoritmo y de la máquina real en que se ejecuta. El análisis normalmente debe ser independiente del lenguaje o máquina que se utilice para implementar el algoritmo. La tarea de calcular el tiempo exacto requerido suele ser bastante laborioso. Para realizar este cálculo se necesitan las instrucciones exactas ejecutadas por el hardware y el tiempo requerido para cada instrucción. El análisis del algoritmo pretende obtener el orden de magnitud de tiempo requerido para la ejecución del mismo. Cada algoritmo tendrá un coste computacional diferente.

La eficiencia es un criterio que se debe utilizar en la selección de un algoritmo y su posterior implementación. Una comparación de algoritmos se debe centrar en diferencias significativas en eficiencia. ¿Cómo comparar la eficiencia de tiempo de dos algoritmos que resuelvan el mismo problema? Un posible enfoque es implementar los dos algoritmos en local y ejecutar los programas. Existen al menos tres interrogantes fundamentales con este enfoque:

1. ¿Cómo se codifican los algoritmos?
2. ¿Qué computadora se utilizará?
3. ¿Qué datos debe utilizar el programa?

El análisis de la eficiencia debe ser independiente de las implementaciones específicas, de las computadoras y de los datos específicos que manipulan.

Para recordar

El *Análisis de Algoritmos* es una rama de las ciencias de la computación que estudia la *eficiencia* de los algoritmos. Frecuentemente lleva consigo un análisis matemático complicado.

2.1.1. Análisis del orden de magnitud

Las técnicas matemáticas analizan algoritmos con independencia de implementaciones específicas, computadoras o datos. El enfoque matemático considera un enfoque de tiempo del algoritmo como una función del tamaño del problema. El medio para medir el tamaño de un problema depende de la aplicación; así por ejemplo el tamaño de un array en cuanto al número de elementos, el número de nodos de una lista enlazada, el número de registros de un archivo, el número de elementos de una lista ordenada o el grado de un polinomio.

A medida que crece el tamaño del programa, generalmente, crece el tiempo de ejecución. Observando cómo varía el tiempo de ejecución con el tamaño de la entrada, se puede determinar la *tasa de crecimiento* del programa o algoritmo, expresado normalmente en términos de n, donde n es una medida del tamaño de la entrada o complejidad de la entrada.

La *tasa de crecimiento* de un programa es una medida importante de la eficiencia ya que predice cuánto tiempo se requerirá para entradas muy grandes.

A tener en cuenta

El tiempo de ejecución de un algoritmo no se puede medir como tiempo de reloj, no es una medida exacta. El enfoque matemático considera el tiempo de un algoritmo como una función del tamaño del problema. El concepto *tamaño del problema* depende de la aplicación —*ejemplos típicos son el número n de elementos de un array, el exponente entero de una* potencia, el número, *n*, de nodos de una lista enlazada, el número de registros de un archivo o el grado de un polinomio.

¿Qué se necesita conocer sobre los requisitos de tiempo de un algoritmo? Un sistema es tomar el número de sentencias ejecutadas en una traza del algoritmo como medida de los requisitos de tiempo de ejecución del programa correspondiente. De modo similar, se toman el número de variables en una traza del algoritmo como una medida de los requisitos de memoria principal del programa. Ambas medidas se representarán como funciones del «tamaño» del problema. Por ejemplo, el tamaño de un problema de ordenación es el número de valores a ordenar. Normalmente, un problema con n registros de entrada se dice que es «de tamaño n».

2.1.2. Consideraciones de eficiencia

Las consideraciones de eficiencia fundamentales son: *el tiempo* y *el espacio*. El programador normalmente debe optimizar ambos conceptos, aunque a menudo se producirá la optimización de un concepto a costa del otro. Al considerar el tiempo necesario para ordenar un archivo de tamaño n, no son interesantes las unidades de tiempo real, porque éstas varían de unas máquinas a otras, de un programa a otro y de un conjunto de datos a otro. Lo que realmente interesa es el cambio correspondiente en la cantidad de tiempo para ordenar un archivo, inducido por un cambio en el tamaño del archivo n. Así, por ejemplo, si n es el tamaño del problema, se puede decir que un algoritmo requerirá un tiempo proporcional a n, n^2, 2^n o $\log n$.

Considérese un algoritmo `Alg` del que se conoce que requiere *un tiempo proporcional a* n^2 ¿Qué puede significar? ¿Cuál es la cantidad exacta de tiempo que `Alg` requiere para resolver un problema de tamaño n? La afirmación es válida también tanto si `Alg` requiere n^2 segundos, $5 \cdot n^2$ segundos, o $n^2/10$ segundos para resolver un problema de tamaño n.

2.1.3. Análisis de rendimiento

La medida del rendimiento o prestaciones se consigue mediante la complejidad del tiempo y espacio de un programa.

La *complejidad del espacio* de un programa es la cantidad de memoria que se necesita para ejecutar hasta la compleción (*terminación*). La *complejidad del tiempo* de un programa es la cantidad de tiempo de computadora que se necesita para ejecutar hasta la compleción (completado totalmente).

Para recordar

El rendimiento de un programa se mide en dos variables: la memoria ocupada y el tiempo de ejecución. Es la *complejidad del espacio* y la *complejidad del tiempo*.

El espacio ocupado por un programa se determina por el número de variables, el tamaño de cada una de ellas y las estructuras de datos que necesita el algoritmo que implementa el programa. El avance tecnológico proporciona hoy en día memoria abundante, por esta razón el análisis de algoritmos se centra fundamentalmente en el tiempo de ejecución.

2.1.4. Tiempo de ejecución

Es conveniente utilizar una función $T(n)$ para representar el número de unidades de tiempo tomadas por un programa o un algoritmo de cualquier entrada de tamaño n. $T(n)$ es el tiempo de ejecución de un programa. Por ejemplo, un programa puede tener un tiempo de ejecución $T(n) = c \cdot n$, donde c es una constante. El tiempo de ejecución de un programa es linealmente proporcional al tamaño de la entrada sobre la que se ejecuta. Tal programa o algoritmo se dice que es *tiempo lineal* o simplemente *lineal*.

$T(n)$ es, en realidad, el número de sentencias del programa C ejecutadas por el programa; se puede ver como la longitud de tiempo tomada para ejecutar el programa en una computadora estándar.

Con frecuencia, el tiempo de ejecución de un programa depende de una entrada particular, no del tamaño de la entrada. El tiempo medio de ejecución suele ser una medida más realista de lo que las prestaciones verán en la práctica, aunque con frecuencia es mucho más difícil de calcular que el tiempo de ejecución en el caso peor. El concepto de «tiempo medio de ejecución» implica también que todas las entradas de tamaño n son igualmente probables, que puede o no ser cierto en una situación dada.

Otra medida del rendimiento común es $T_{\text{media}}(n)$ (tiempo medio de ejecución del programa sobre todas las entradas de tamaño n). El tiempo medio de ejecución es, a veces, una medida más realista de lo que el rendimiento será en la práctica, pero es, normalmente, mucho más difícil de calcular que el tiempo de ejecución en el caso peor. La noción de un tiempo de ejecución «medio» implica también que todas las entradas de tamaño n son igualmente probables, que puede o no ser verdadero en una situación dada.

El tiempo de ejecución de una función se puede determinar como el tiempo requerido para cada una de las sentencias, multiplicadas por el número de veces que se ejecutan. Consideremos, por ejemplo, la función:

```
double serie(double x, int n)
{
  double s;
  int i;
  s = 0.0;                /* tiempo t1  */
  for (i=1; i<=n; i++)    /* tiempo t2  */
  {
   s += i*x;              /* tiempo t3  */
  }
  return s;               /* tiempo t4  */
}
```

El tiempo de ejecución de esta función es:

$$T(n) = t_1 + nt_2 + nt_3 + t_4$$

y para una computadora particular se calcula respecto a los valores de t_1, t_2, t_3 y t_4. Es inmediato que este tiempo crece según crece el tamaño de n, por esa razón es preferible expresar el tiempo de ejecución de tal forma que indique el comportamiento que va a tener la función con respecto al valor de n.

2.2. NOTACIÓN *O-GRANDE*

Supongamos que se ha escrito y depurado un programa en C y se ha seleccionado la entrada específica para considerar su ejecución. El tiempo de ejecución del programa para esta entrada dependerá de dos factores.

1. La computadora en la que se ejecuta el programa. Algunas computadoras ejecutan instrucciones más rápidamente que otras. Así, la relación de las velocidades de las supercomputadoras más rápidas y las computadoras personales más lentas es aproximadamente 1.000 a 1.
2. El compilador C específico utilizado para generar instrucciones máquina para esta computadora. El número de instrucciones máquinas utilizadas para ejecutar una sentencia dada variará de un compilador a otro compilador.

En consecuencia no podemos decir relativo a un programa en C y sus entradas que «esta tarea llevará 4.51 segundos el tiempo», a menos que se conozca cuál es la máquina y cuál es el compilador.

Por estas razones, el tiempo de ejecución de un programa se expresa normalmente utilizando la notación asintótica: «notación *O*, *O-grande*». Con la notación *O* se dispone de un medio para expresar la cota superior asintótica de una función. En cuanto al tiempo de ejecución, la notación *O* indica la cota superior asíntótica. Está diseñada para expresar factores constantes tales como:

1. El número medio de instrucciones máquina que genera un compilador determinado.
2. El número medio de instrucciones máquina por segundo que ejecuta una computadora específica.

Así, en lugar de decir que un algoritmo determinado emplea un tiempo de $4m - 1$ en un array de longitud m, se dirá que emplea un tiempo $O(m)$, que se lee «*O-grande* de m», o bien «*oh de m*», y que informalmente significa «algunos tiempos constantes m».

La noción de «algunos tiempos constantes m» no sólo permite ignorar la constante desconocida asociada con el compilador y la máquina, sino que permite hacer algunas hipótesis para simplificar.

> Con la notación *O* se expresa una aproximación de la relación entre el tamaño de un problema y la cantidad de proceso necesario para hacerlo. Por ejemplo, si
>
> `f(n) = n² - 2n + 3` entonces `f(n)` es $O(n^2)$.

2.2.1. Descripción de tiempos de ejecución con la notación *O*

En esta sección se examinará la notación más típica para describir tiempos de ejecución, como funciones de la cantidad de datos implicados.

Supongamos que $T(n)$ es el tiempo de ejecución de algún programa, medido como una función de la entrada de tamaño n. Esta función que mide el tiempo de ejecución de un programa supondrá que:

1. El argumento n es un entero no negativo.
2. $T(n)$ es no negativo para todos los argumentos n.

Si $f(n)$ es una función definida sobre enteros no negativos, se dice «$T(n)$ es $O(f(n))$» si $f(n)$ acota superiormente a $T(n)$. Esta característica se puede expresar diciendo que $T(n)$ es como máximo una constante de tiempo $f(n)$, excepto posiblemente para algunos valores pequeños de n.

De modo más riguroso, se dice: $T(n)$ es $O(f(n))$ si existe un entero n_0 y una constante $c > 0$ tal que para todos los enteros $n \geqslant n_0$ se tendrá que $T(n) \leqslant cf(n)$.

Se puede aplicar la definición de «O-grande» para probar que $T(n)$ es $O(f(n))$ para unas funciones específicas T y f. Para ello se eligen un par de constantes, n_0 y c, y a continuación se prueba que $T(n) \leqslant cf(n)$, suponiendo sólo que n es un entero no negativo y que n es al menos tan grande como nuestra opción n_0.

Ejemplo 2.1

Sea la función $f(n) = n^3 + 3n + 1$, encontrar la complejidad asintótica.

Para valores de $n \geqslant 1$ se puede demostrar que

$$f(n) = n^3 + 3n + 1 \leqslant n^3 + 3n^3 + 1n^3 = 5n^3$$

Entonces, escogiendo la constante $c = 5$ y $n_0 = 1$ se satisface la desigualdad $f(n) \leqslant 5\,n^3$, de tal forma que se puede asegurar que

$$f(n) = O(n^3)$$

Ahora bien, también se puede asegurar que $f(n) = O(n^4)$ y que $f(n) = O(n^5)$, y así sucesivamente, con potencias mayores de n. Sin embargo, lo que realmente interesa es la cota superior más ajustada que informa de la tasa de crecimiento de la función con respecto a n.

2.2.2. Definición conceptual

Específicamente, la notación $f(x) = O(g(x))$ significa que existen constantes c y x_0 tales que $-Cg(x) \leqslant f(x) \leqslant Cg(x)$, para todo $x \geqslant x_0$. En otras palabras, $f(x) = O(g(x))$ significa que $|f(x)| \leqslant c|g(x)|$, para $x \geqslant x_0$. Por consiguiente, $|g(x)|$ es un límite superior para $f(x)$, como se ilustra en la Figura 2.1.

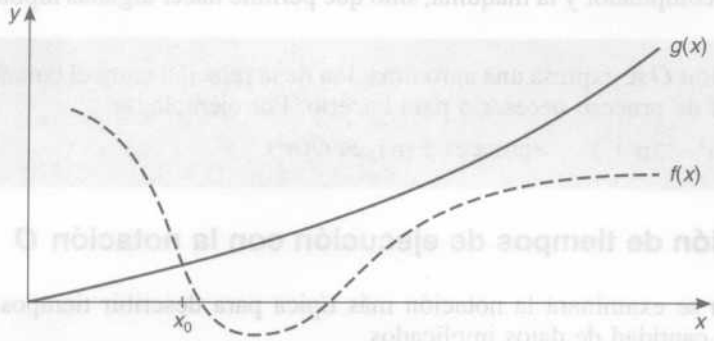


Figura 2.1. $f(x) = O(g(x))$ si existen números c y x_0 tales que $|f(x)| \leqslant C \cdot g(x)$.

Supongamos que $f(x) = 2x^2 + 3x - 1$, entonces:

$$f(x) \leqslant 3{,}5x^2 \quad \text{para} \quad x \geqslant 1{,}6$$

$$f(x) \leqslant 2x^3 \quad \text{para} \quad x \geqslant 1{,}8$$

$$f(x) \leqslant x^4 \quad \text{para} \quad x \geqslant 1{,}9$$

Por consiguiente, $f(x) = O(x^2)$, $f(x) = O(x^3)$ y $f(x) = O(x^4)$ y, naturalmente, se continúa hasta potencias de nivel superior. Los múltiplos de todas estas potencias son cotas superiores de $f(x)$ si x es suficientemente grande.

Sin embargo, se suele considerar la potencia de nivel inferior que puede actuar como un límite superior, de modo que, en general, $f(x) = 2x^2 + 3x - 1 = O(x^2)$.

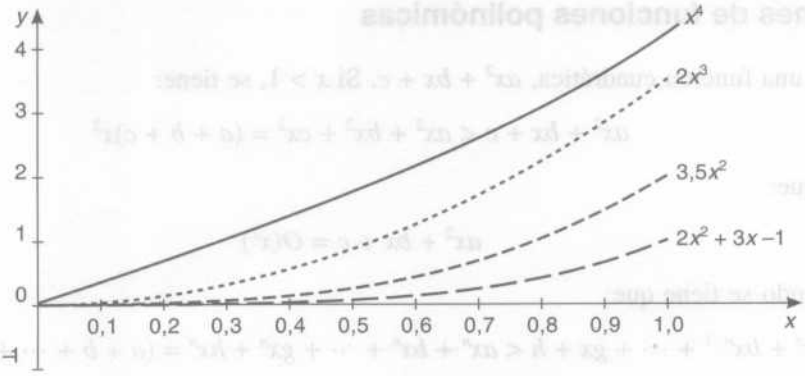


Figura 2.2. Las funciones individuales $3,5x^2$, $2x^3$ y x^4 residen por encima de $f(x) = 2x^2 + 3x - 1$ para x bastante grande.

Una función $f(x)$ puede estar acotada superiormente por un número indefinido de funciones a partir de ciertos valores x_0,

```
f(x) ⩽ g(x) ⩽ h(x) ⩽ k(x) ...
```

La complejidad asintótica de la función $f(x)$ se considera que es la cota superior más ajustada:

```
f(x) = O(g(x))
```

2.3. COMPLEJIDAD DE FUNCIONES

El orden, en cuanto a la complejidad que tienen las funciones con las que habitualmente se expresan los tiempos de ejecución de un programa, se deduce en los siguientes apartados.

2.3.1. Funciones tipo monomio

Supongamos que $1 < x$, multiplicando repetidamente en ambos lados de la desigualdad por el positivo x, quedará $x < x^2$, $x^2 < x^3$, $x^3 < x^4$, ...

Por inducción, se deduce que $1 = x^0 < x^1 < x^2 < x^3 < ...$
En otras palabras: $x^n < x^{n+1}$, para todo $x > 1$, $n \geqslant 0$.
Por consiguiente: $x^n = O(x^n) = O(x^{n+1}) = O(x^{n+2}) = ...$
En general, casi siempre se considera el límite superior más bajo para una función dada; para cualquier constante k se tendrá:

$$k = O(1)$$

$$kx = O(x)$$

$$kx^2 = O(x^2), ...$$

Se puede concluir que la complejidad de $f(x) = ax^n = O(x^n)$, para cualquier constante a.

2.3.2. Órdenes de funciones polinómicas

Considere una función cuadrática, $ax^2 + bx + c$. Si $x > 1$, se tiene:

$$ax^2 + bx + c < ax^2 + bx^2 + cx^2 = (a + b + c)x^2$$

de modo que:

$$ax^2 + bx + c = O(x^2)$$

Generalizando se tiene que:

$$ax^n + bx^{n-1} + \cdots + gx + h < ax^n + bx^n + \cdots + gx^n + hx^n = (a + b + \cdots + g + h)x^n$$

de modo que:

$$ax^n + bx^{h+1} + \cdots + gx + h = O(x^n)$$

El orden de cualquier función polinómica de grado n de x es $O(x^n)$ para $x > 1$, $n \geqslant 0$.

2.3.3. Órdenes exponenciales y logarítmicas

La función exponencial e^x se puede expresar como una serie infinita de potencias de x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

De este modo, una función exponencial tiene un orden mayor que cualquier orden monomial:

$$ke^x = O(e^x) > O(x^n), \quad \text{para todo } n$$

De modo similar es posible probar que una función logarítmica tiene un orden menor que cualquier orden monomial positivo.

$$k \log_b x = O(\log_b x) < O(x^n), \quad \text{para } b > 1 \text{ y cualquier entero } n \geqslant 1$$

La Figura 2.3 muestra el gráfico de e^x ln (x) y x^n.

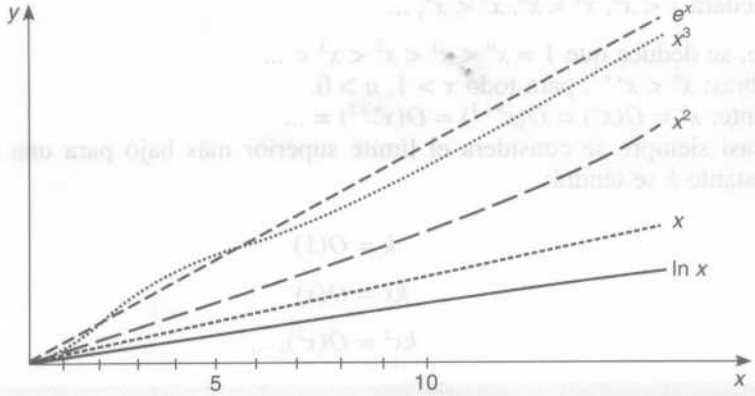


Figura 2.3. Para valores grandes de x, la función e^x se sitúa por encima de la función potencia de $x(x^n)$, y la función ln x se sitúa por debajo.

2.3.4. Bases de logarítmos distintas

Los logaritmos de bases diferentes del mismo número son proporcionales. Supongamos que b y B son bases diferentes. Los logaritmos de un número x de definen matemáticamente así:

$$x = b^{\log_b x} \quad \text{y} \quad x = B^{\log_B x}$$

Si se hace el logaritmo en base B de estas ecuaciones, resulta:

$$\log_B x = \log_B (b^{\log_b x}) = (\log_b x)(\log_B b)$$
$$\log_B x = \log_B (B^{\log_B x}) = (\log_B x)(\log_B B) = \log_B x$$

de modo que

$$\boxed{\log_b x = \frac{\log_B x}{\log_B b}}$$

Así, para el caso de las bases 2 y 10, se tiene que:

$$\log_2 x = \frac{\log_{10} x}{\log_{10} 2} = 3{,}3219281 \log_{10} x \qquad \text{y} \qquad \log_2 x = \frac{\ln x}{\ln 2} = 1{,}4426950 \ln x$$

En consecuencia, las funciones logarítmicas de todas las bases tienen el mismo orden:

$$O(\log_{10} x) = O(\ln x) = O(\log_2 x) = O(\log x)$$

Las funciones logarítmicas son de *orden logarítmico*, independientemente de la base del logaritmo.

2.3.5. Orden de una función general

La medida de *O-grande* de una función depende de su variable independiente y su orden es el correspondiente al término que crece más rápidamente. Así por ejemplo, en las siguientes funciones resulta:

$$f(n) = 2n^3 - 8n + 5 = O(n^3)$$
$$f(r) = (r^2 - 3r + 5)/(r - 1) = O(r)$$
$$f(n) = 15 = O(1)$$
$$f(p) = \log_{10} p + 5p = O(p)$$
$$f(x) = \log_2 x + (5/x) = O(\log x)$$

Ejemplo 2.2

Sea $f(n) = n \cdot (n + 1) + 4$, para $n = 1, 2, \ldots$ Demostrar que f es $O(n^2)$.

Demostración

Se necesita encontrar constantes c y n_0 tales que

$$f(n) \leqslant c \cdot n^2 \quad \text{para todo } n \geqslant n_0$$

Dado que $n \cdot (n + 1) + 4 = n^2 + n + 4$ para todos los enteros positivos n, conocemos que $f(n) \leqslant n^2 + n^2 + 4 \cdot n^2$ siempre que $n \geqslant 1$, y así sucesivamente para $c = 6$ y $n_0 = 1$, $|f(n)| \leqslant c \cdot n^2$, para todo $n \geqslant n_0$. Por consiguiente, f es $O(n^2)$.

En general, si f es un polinomio de la forma $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$, se puede establecer que f es $O(n^k)$ eligiendo $N = 1$:

$$c = |a_k| + |a_{k-1}| + \cdots + |a_1| + |a_0|$$

Ejemplo 2.3

Sean a y b constantes positivas. Demostrar que si f es $O(\log_a n)$, entonces f es también $O(\log_b n)$.

Demostración

Supongamos que f es $O(\log_a n)$. Por consiguiente, existen constantes positivas c y n_0 tales que para todo $n \geqslant n_0$

$$|f(n)| \leqslant c \cdot \log_a n$$

Recordemos una prioridad fundamental de los logaritmos:

$$\log_a n = (\log_a b) \cdot (\log_b n), \quad \text{para todo} \quad n > 0$$

Sea

$$c_1 = c \cdot \log_a b$$

entonces para todo $n \geqslant n_0$ se tiene que

$$|f(n)| \leqslant c \cdot \log_a n = c \cdot \log_a b \cdot \log_b n = c_1 \cdot \log_b n$$

de modo que f es $O(\log_b n)$.

2.3.6. Propiedades de la notación *O-grande*

A partir de la definición conceptual de la notación O y del orden de las funciones más habituales analizadas anteriormente en este apartado, se deducen las siguientes propiedades de la notación O.

1. Siendo c una constante,

$$c \cdot O(f(n)) = O(f(n))$$

   Por ejemplo, si `f(n) = 3n⁴`, entonces `f(n) = 3 * O(n⁴) = O(n⁴)`.

2. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$. Por ejemplo, si `f(n) = 2eⁿ` y `g(n) = 2 n³`, entonces

```
O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(2eⁿ + 2 n³) = O(eⁿ)
```

   La última igualdad está basada en el Apartado 2.3.3, en el que se demuestra el mayor orden de la función exponencial frente a la monomial.

3. *Máximo*$(O(f(n)), O(g(n))) = O($*Máximo*$(f(n), g(n))$. Por ejemplo,

```
Maximo(O(log(n)), O(n) = O(Maximo(log(n), n)) = O(n)
```

4. $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$. Por ejemplo, si `f(n) = 2n³` y `g(n) = n`, entonces

```
O(f(n)) * O(g(x)) = O(f(x) * g(x)) = O(2n³ * n) = O(n⁴)
```

5. $O(\log_a(n)) = O(\log_b(n))$, para $a, b > 1$. Las funciones logarítmicas tienen todas el mismo orden, independientemente de la base del logaritmo.

6. $O(\log(n!)) = O(n \cdot \log(n))$.

7. Siendo k un constante mayor que uno, $k > 1$ entonces:

$$O\left(\sum_{i=1}^{n} i^k\right) = O(n^{k+1})$$

8. $O\left(\sum_{i=2}^{n} \log(i)\right) = O(n \log(n))$

2.4. 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 $O(n^2 + 5n + 3)$, $O(n^3)$ y $O(n^{10} + 3)$. Siempre que sea posible, se elige el elemento más pequeño de la siguiente jerarquía de órdenes:

$$O(1),\ O(\log n),\ O(n),\ O(n \log n),\ O(n^2),\ O(n^3),\ \ldots,\ O(2^n),\ \ldots,\ O(n^n)$$

Ejemplo

Si $f(n) = n + 7$, para $n = 1, 2, \ldots$, se dice que f es $O(n)$. Algunas funciones de muestra en la jerarquía de órdenes son:

Orden	**Funcin de muestra**
$O(1)$	$f(n) = 5$
$O(\log n)$	$f(n) = (n \cdot \log_2(n + 1) + 5)/(n + 1)$
$O(n)$	$f(n) = c \log_2 n + n$
$O(n \log n)$	$f(n) = \log_2 n^n$
$O(n^2)$	$f(n) = n \cdot (n + 1)/2$

Otro *inconveniente* de la notación O-grande es que aproxima el comportamiento de una función sólo para argumentos arbitrarios grandes. Por ejemplo, supongamos que el número de sentencias ejecutadas en una traza del algoritmo $Alg1$ viene dada de entrada por la función T_1 en términos del número de valores de entrada n para $n = 1, 2, \ldots$

$$T_1(n) = 6n \log_2(n + 1) + 15n + 25 \quad \text{sentencias ejecutadas}$$

De modo similar para el algoritmo $Alg2$, se supone que la función correspondiente viene dada por:

$$T_2(n) = n^2/3 - 7n \quad \text{sentencias ejecutadas}$$

En estos casos T_1 es $O(n \log n)$ y T_2 es $O(n^2)$, lo que conduce a pensar que T_1 es «más pequeño» que T_2, lo que no es cierto para $n \leqslant 200$, en donde T_1 es mayor que T_2.

La notación *O-grande* es una herramienta muy útil, pero ha de ser utilizada con prudencia.

2.5. COMPLEJIDAD DE LAS SENTENCIAS DE UNA FUNCIÓN C

A la hora de analizar la complejidad de una función no recursiva escrita en C, de un algoritmo, se han de aplicar las propiedades de la notación O y las siguientes consideraciones relativas al orden que tienen las sentencias, fundamentalmente a las estructuras de control.

- Las sentencias de asignación son de orden constante $O(1)$.
- La complejidad de una sentencia de selección es el máximo de las complejidades del bloque `then` y del bloque `else`.
- La complejidad de una sentencia de selección múltiple (`switch`) es el máximo de las complejidades de cada uno de los bloques `case`.
- Para calcular la complejidad de un bucle, condicional o automático, se ha de estimar el número máximo de iteraciones para el peor caso; entonces la complejidad del bucle es el producto del número de iteraciones por la complejidad de las sentencias que forman el cuerpo del bucle.
- La complejidad de un bloque se calcula como la suma de las complejidades de cada sentencia del bloque.
- La complejidad de la llamada a una función se considera de orden 1, complejidad constante. Hay que considerar la complejidad que tenga la función invocada.

Ejemplo 2.4

Determinar la complejidad de la función:

```
double mayor(double x, double y)
{
   if (x > y)
     return x;
   else
     return y;
}
```

Esta función consta de una sentencia de selección, cada alternativa tiene complejidad constante, $O(1)$. Como conclusión, la complejidad es $O(1)$.

Ejemplo 2.5

Determinar la complejidad de la función:

```
void entrada(double x[], int n)
{
   int j;
   for (j = 0; j < n; n++)
   {
     scanf("%lf",&x[j]);
   }
}
```

La función consta de una sentencia repetitiva, un bucle que se ejecuta n veces, $O(n)$. El cuerpo del bucle consta de una sentencia simple, llamada a la función `scanf()`, complejidad constante $O(1)$. Como conclusión, la complejidad es:

```
O(n) * O(1) = O(n)
```

Ejemplo 2.6

Determinar la complejidad de la función:

```
float suma(float d[], int n)
{
   int k ;
   k= s = 0;
   while (k < n)
   {
      s += d[k];
      if (k == 0)
        k = 2;
      else
        k = 2*k;
   }
   return s;
}
```

La función `suma()` consta de una sentencia de asignación múltiple, $O(1)$, de un bucle condicional y una última sentencia que devuelve control de complejidad constante, $O(1)$. Por consiguiente, la complejidad de la función va a ser la del bucle.

Se tiene que determinar el número de iteraciones que va a realizar el bucle y la complejidad del cuerpo del bucle. El número de iteraciones es igual al número de veces que el algoritmo multiplica por dos a la variable `k`. Supongamos que $k = 2^t$, donde t es el número de veces que k se puede multiplicar hasta alcanzar el valor de n que hace que termine el bucle.

$0, 2, 2^2, 2^3, \ldots, 2^t \geqslant n$. Tomando logaritmos, $t \geqslant \log_2 n$ y es inmediato que el máximo de iteraciones es $t = \log_2 n$.

El cuerpo del bucle tiene complejidad constante $O(1)$, al tener una sentencia simple y un sentencia de selección de complejidad $O(1)$. Por lo que se puede afirmar que la complejidad del bucle y de la función, $O(\log_2 n) * O(1) = O(\log_2 n)$, complejidad logarítmica que se expresa: $O(\log n)$.

Ejemplo 2.7

Determinar la complejidad de la función:

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

En este ejemplo la función consta de dos bucles `for` anidados. El bucle interno tiene un bloque de tres sentencias de complejidad constante, $O(1)$. Por ello la complejidad de la función va a ser la de los dos bucles. El bucle externo siempre realiza $n - 1$ veces el bucle interno. A su vez, el bucle interno hace k veces su bloque de sentencias, donde k va desde 1 a $n - 1$, de modo que el número total de iteraciones es:

$$C = \sum_{k=1}^{n-1} k$$

y en consecuencia el desarrollo anterior produce la siguiente expresión:

$$(n-1) + (n-2) + \cdots + 1 = \frac{n \cdot (n-1)}{2}$$

La suma es cuadrática, por lo que el número de iteraciones es:

$$\frac{n^2}{2} - \frac{n}{2}$$

y enseguida se deduce, teniendo en cuenta las propiedades de la notación O, que la complejidad de la función es $O(n^2)$. El término que domina en el tiempo de ejecución es n^2, por ello se dice que la complejidad es cuadrática.

2.5.1. Complejidades más comunes en algoritmos

Las complejidades que aparecen más frecuentemente en el análisis de algoritmos, se exponen en orden creciente y de menor complejidad a mayor.

- *Complejidad constante*, se expresa como $O(1)$. Se encuentra en algoritmos sin bucles; por ejemplo, el intercambio de dos variables.
- *Complejidad logarítmica*, se expresa como $O(\log n)$. Es una complejidad óptima; por ejemplo, el algoritmo de búsqueda binaria tiene esta complejidad.
- *Complejidad lineal*, expresada como $O(n)$. Se considera una buena complejidad, se encuentra en los bucles simples; por ejemplo, el cálculo del mínimo de un vector.
- *Complejidad* $O(n \cdot \log n)$. Es una buena complejidad; los mejores algoritmos de ordenación tienen esta complejidad.
- *Complejidad cuadrática*, expresada como $O(n^2)$. Es una complejidad que aparece en el tratamiento de matrices, generalmente con dos bucles anidados.
- *Complejidad cúbica*, expresada como $O(n^3)$. Para n grande la complejidad crece notablemente. Esta complejidad se encuentra en algoritmos con tres bucles anidados; por ejemplo, el algoritmo de Floyd, también al procesar arrays tridimensionales.
- *Complejidad exponencial*, $O(2^n)$. Es evidente que la tasa de crecimiento de los algoritmos con esta complejidad crece muy rápidamente con el tamaño n. Es la complejidad de algunos algoritmos recursivos, como las Torres de Hanoi.
- *Complejidad* $O(k^n)$. Esta complejidad se manifiesta en algoritmos recursivos, como el del *Salto del caballo*, o las *Ocho reinas*, que tienen un bucle de k llamadas recursivas.

2.6. ANÁLISIS DE ALGORITMOS DE ORDENACIÓN

La **ordenación** es un proceso que organiza una colección de datos en un orden ascendente o descendente. La necesidad de ordenación se requiere en muchas situaciones. Se puede desear ordenar una colección de datos antes de ser visualizados para el consumo humano. Se debe realizar una ordenación como una etapa previa para ciertos algoritmos; por ejemplo, la búsqueda de datos, que es una de las tareas más frecuentes realizadas por las computadoras. Cuando la colección de datos, que se busca es grande, un método eficiente de búsqueda es deseable. El algoritmo de búsqueda binaria es el óptimo, pero tiene el inconveniente de que los datos han de estar ordenados. Por consiguiente, la ordenación es una etapa que debe preceder a una búsqueda binaria en una colección de datos que no estén ordenados.

La eficiencia de los algoritmos de ordenación es un parámetro importante a considerar en cualquier problema de tratamiento de datos.

2.6.1. Análisis de la ordenación por selección

Una función C que realiza una ordenación por selección de un array (lista) a de n elementos puede ser éste:

```
void ordenarSeleccion(double a[], int n)
{
   int i,j,jMax;
   for (i = n-1; i >= 1; i--)
   {
      jMax = 0;
      for (j = 1; j <= i; j++) /* encontrar clave mayor */
        if (a[j] > a[jMax])
        {
            jMax = j;
        }
     intercambio (a[i], a[jMax]);
   }
}
```

Como primera etapa en el análisis de algoritmos se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. La función `ordenarSeleccion()` intercambia dos elementos del array al final de cada vuelta a través del bucle `for`, sin importar el orden en que estén originalmente los elementos. Por consiguiente, la función siempre realiza $n - 1$ intercambios (n, número de elementos del array). El bucle interno es ejecutado $n - 1$ veces. Cada bucle se ejecuta una vez menos, $n - 2$, etc. Por consiguiente se cuentan dos casos en ordenación: el número de comparaciones que se han de hacer y el número de datos que se deben mover. El bucle interno hace $i - 1$ comparaciones cada vez; el bucle externo va desde 2 a n. De modo que el número total de comparaciones es:

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

y en consecuencia el desarrollo anterior produce la siguiente expresión:

$$(n - 1) + (n - 2) + \cdots + 1 = \frac{n \cdot (n - 1)}{2}$$

La suma es cuadrática, por lo que el orden de la función es $O(n^2)$.

Ha de observarse que el algoritmo no depende de la disposición inicial de los datos; ello supone una ventaja en un algoritmo de selección, pero dado que $O(n^2)$ es cuadrático, tiende a crecer rápidamente.

2.6.2. Análisis de la ordenación por inserción

La siguiente función C realiza una ordenación por inserción en un array de n elementos de tipo `double` puede ser:

```
void ordenarInsercion (double a[], int n)
{
    int k, i;
    int orden;
    double tem;
    for (k = 1; k < n; k++)
    {
        orden = 0;
        tem = a[k];
        i = k-1;
        while ((i >= 0) && (!orden))
        {
            if (a[i] > tem)
            {
              a[i+1] = a[i];
              i = i-1;
            }
            else
              orden = 1;
        }
        a[i+1] = tem;
    }
}
```

El bucle `for` de la función `ordenarInsercion()` se ejecuta $n - 1$ veces. Dentro de este bucle existe un bucle `while` que se ejecuta a lo más el valor de $i - 1$ 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:

$$\sum_{i=1}^{n-1} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

es decir,

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

Por otra parte, el algoritmo mueve los datos como máximo el mismo número de veces. Existen, en el peor de los casos, las siguientes comparaciones:

$$n \cdot (n-1) = n^2 - n$$

Por consiguiente, el algoritmo de ordenación por inserción es $O(n^2)$ en el caso peor. Para listas de datos pequeñas —menos de 50 elementos— la sencillez del algoritmo de inserción es una elección adecuada. Para listas grandes, sin embargo, el algoritmo por inserción no suele ser una buena elección por su ineficiencia.

2.7. LA EFICIENCIA DE LOS ALGORITMOS DE BÚSQUEDA

Una aplicación típica del análisis de algoritmos es medir la eficiencia de dos algoritmos clásicos de búsqueda en una colección, lista o vector (*array*) de datos: búsqueda secuencial y búsqueda binaria.

2.7.1. Búsqueda secuencial

El algoritmo de búsqueda secuencial en una lista (*array*) de *n* elementos, recorre cada elemento por turno, comenzando con el primero y en secuencia hasta que, o bien se encuentra el elemento deseado o bien se alcanza el final de la colección de datos. La función escrita en C:

```
int busquedaLineal (int a[], int x, int n)
{
   int encontrado;
   int k, posicion;
   posicion = -1;
   encontrado = 0;
   k = 0;
   while ((k < n) && (!encontrado)) do
   {
      if (a[k] == x)
      {
         posicion = k;
         encontrado = 1;
      }
      else
        k++;
   }
   return posicion;
}
```

En el mejor caso, el elemento deseado es el primero que se examina, de modo que sólo se necesita una comparación, y en este mejor caso, una búsqueda secuencial es $O(1)$. En el peor caso, el elemento deseado es el último que se examina, de modo que se necesitan *n* comparaciones, y en este peor de los casos, el elemento deseado es el último que se examina, de modo que son necesarias *n* comparaciones; por consiguiente, en el peor de los casos, el algoritmo es $O(n)$. En el caso medio, se encontrará el elemento deseado en el centro de la colección, haciendo $n/2$ comparaciones y en este caso el algoritmo es $O(n)$ en el caso medio.

Búsqueda secuencial	
• Caso mejor	$O(1)$
• Caso medio	$O(n)$
• Caso peor	$O(n)$

2.7.2. Búsqueda binaria

La primera pregunta que cabe hacerse es: ¿cuál es el método más eficiente de búsqueda? El algoritmo de búsqueda binaria busca en una lista (*array*) ordenada un elemento específico dividiendo repetidamente el array en dos mitades. El código de la función en C:

```
int busquedaBinaria (int a[], int x, int n)
{
   int encontrado;
   int primero, ultimo, central;
   encontrado = 0;
   primero = 0;
   ultimo = n-1;
   while ((primero <= ultimo) && (!encontrado))
   {
      central = (primero+ultimo)/2;
      if (a[central] == x)
      {
        encontrado = 1;
      }
      else
      {
         if (a[central] > x)
           primero = central+1;
         else
           ultimo = central-1;
         }
   }
   if (!encontrado)
     return -1;
   else
     return central;
}
```

El algoritmo determina en qué mitad está el elemento —si es que está dentro de la lista— y descarta la otra mitad. Por consiguiente, el algoritmo de búsqueda binaria busca sucesivamente arrays de tamaños más pequeños. El tamaño de un array es aproximadamente una mitad del tamaño del array buscado anteriormente.

En cada división, el algoritmo hace una comparación. ¿Cuántas comparaciones hace el algoritmo cuando busca en un array de n elementos? La respuesta exacta depende, naturalmente, de donde esté situado el elemento deseado. Sin embargo, se puede calcular el número máximo de comparaciones que requiere una búsqueda binaria; es decir, el peor caso. El número de comparaciones es igual al número de veces que el algoritmo divide el array por la mitad. Supongamos que $n = 2^k$, donde k es el número de veces que n se puede dividir hasta tener un elemento encuadrado. La búsqueda requiere las siguientes etapas:

1. Inspeccionar el elemento central de un array de tamaño n.
2. Inspeccionar el elemento central de un array de tamaño $n/2^2$.
3. ...

Si se realizan divisiones sucesivas hasta que sólo existe un elemento en la partición, se habrán ejecutado k divisiones. Este hecho es cierto ya que $n/2^k = 1$ (se ha supuesto que $n = 2^k$). En el caso

peor, el algoritmo realiza k divisiones, por consiguiente, k comparaciones. Debido a que si $n = 2^k$, k es el logaritmo en base 2 de n ($k = \log_2 n$).

$$k = \log_2 n$$

Se puede decir que k es la parte entera de $\log_2 n$. Por consiguiente el algoritmo es $O(\log_2 n)$ en el peor de los casos cuando $n = 2^k$, es la llamada complejidad logarítmica, que simplemente se expresa $O(\log n)$: ¿Qué sucede si n no es una potencia de 2? Se puede encontrar fácilmente el k más pequeño tal que

$$2^{k-1} < n < 2^k$$

(Por ejemplo, si n es 31, entonces $k = 5$ ya que $2^4 = 16 < 31$ y $2^5 = 32 > 31$.)

El algoritmo requerirá como máximo k divisiones para obtener un subarray con un elemento; es decir:

$$k - 1 < \log_2 n < k$$
$$k < 1 + \log_2 n < k + 1$$
$$k = 1 + \log_2 n \quad \text{redondeado por defecto}$$

En consecuencia, la función O del algoritmo es $O(\log_2 n)$ en el peor caso cuando $n \neq 2^k$. En general, el algoritmo es $O(\log n)$ en el caso peor para cualquier base.

La pregunta inmediata a realizar es: *¿Una búsqueda binaria es mejor que una búsqueda secuencial?* Mucho mejor es la respuesta, excepto algunos casos puntuales. Por ejemplo, en una búsqueda secuencial de una lista de un millón de elementos ordenados puede requerir un millón de comparaciones mientras que en una búsqueda binaria de iguales comparaciones requerirá $\log_2 1.000.000 + 1 \approx 20$. La búsqueda binaria tiene una enorme ventaja sobre una búsqueda secuencial. Sin embargo, el orden y el mantenimiento de un array ordenado requiere también un coste suplementario que será preciso evaluar.

Tabla 2.1. Comparación de funciones de tiempo de ejecución

n	$T(n) = n \log_2 n$	$T(n) = n^2$
2	2	4
8	24	64
10	40	100
50	300	2.500
100	700	10.000
1.000	10.000	1.000.000

RESUMEN

El análisis de algoritmos es el área de la ciencia informática que proporciona herramientas para contrastar la eficiencia de los diferentes algoritmos. Para comparar la eficiencia en cuanto al tiempo de dos algoritmos, se han de considerar al menos las siguientes premisas:

1. ¿Cómo se codifican los algoritmos?
2. ¿Qué computadora se va a utilizar para ejecutarlos?
3. ¿Qué datos de entrada y de salida utilizan los algoritmos?

El análisis de algoritmos debe ser independiente de la implementación específica que se realice. Mediante el análisis de algoritmos se puede establecer la calidad de un programa y compararlo con otros programas sin necesidad de resolver el mismo problema. De este modo, para evaluar un algoritmo se tendrán en cuenta tres aspectos:

1. El algoritmo cumple la especificación dada y funciona para cualquier posible valor de los datos que manipula.
2. El algoritmo es fácil de codificar y depurar, incluso para personas distintas al diseñador.
3. No existe otro algoritmo que resuelva el mismo problema utilizando menos recursos.

Los recursos que miden la eficiencia de un algoritmo son el *tiempo* que tarda en ejecutarse, y la *memoria* que consume durante ese tiempo.

Hoy en día el avance del hardware ha hecho que la eficiencia en lo relativo al binomio *tiempo* × *memoria* no sea tan relevante como hace años. Cada día la potencia y la memoria de las computadoras crece y crece sin un límite aparente. Este crecimiento incita a la resolución de problemas cada vez más complejos y ello obliga a seguir considerando la cuidadosa administración de los recursos de la computadora y conlleva la vigencia del estudio de la eficiencia de los algoritmos que dan solución a un problema.

La evaluación de la memoria no suele ser muy compleja; en algunos algoritmos es preciso considerar la memoria estática y la memoria dinámica. La *memoria estática* se mide mediante el cálculo del tamaño de las variables de tipos simples o escalares, campos de registros y los componentes de vectores y matrices. La memoria dinámica, memoria reservada en tiempo de ejecución, dependerá de la cantidad de datos y del funcionamiento del programa.

Para la evaluación de tiempo se suele suponer una computadora ideal. La cantidad de tiempo será una función $T(n)$, donde n hace referencia a una medida de los datos. $T(n)$ se define como el tiempo empleado por el algoritmo A en procesar una entrada de tamaño n y producir una solución al problema. Así, cuando se dice que la complejidad de un algoritmo es una función $T(n)$ (por ejemplo, n^2) se supone que el tiempo de ejecución del algoritmo es proporcional a $T(n^2)$. La constante de proporcionalidad depende de la computadora.

La notación *O-grande* permite analizar la eficiencia de un algoritmo. Se dice que una función $f(n)$ es $O(g(n))$ si al aumentar el número de datos que se deben procesar, el tiempo del algoritmo va a crecer como lo hace g en relación a n. La complejidad de los algoritmos se expresa normalmente utilizando la notación *O grande*; así se dice que un algoritmo emplea un tiempo $O(n)$, se lee *O de n*, que informalmente significa *tiempos que crecen con n*. Algunas de las notación O más habituales en algoritmos son éstas: $O(n)$ complejidad lineal; $O(n^2)$ complejidad cuadrática; $O(\log n)$ complejidad logarítmica.

EJERCICIOS

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

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

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

c) $x^2 + (1/x)$ es $O(x^2)$

d) $x^2 + \ln x$ es $O(x^2)$

e) $r + r \log r$ es $O(r \log r)$

2.2. Probar que:

$$O(f(x) + k) = O(f(x)) \quad \text{donde } k \text{ es una constante}$$

$$O(k(f(x)) = O(f(x)) \quad \text{donde } k \text{ es una constante}$$

$$O(f(x)) + O(g(x)) = O(g(x)) + O(f(x))$$

$$O(f(x) + O(g(x)) = O(g(x)) \quad \text{si } O(f(x)) \leqslant O(g(x))$$

2.3. Expresar el orden de las siguientes funciones en notación *O-grande:*

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

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

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

d) $x^2 + 5x \ln x$

e) $x^4(x^2 - 1)$

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

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

h) $x^2 \ln x + e^x - 6e^8$

2.4. Demostrar que cualquier función polinómica $f(x) = C_k x^k + C_{k-1}x^{k-1} + \cdots + C_1 x + C_0$ es $O(x^k)$.

2.5. Analizar la complejidad, expresándola en términos de notación *O-grande*, de los siguientes segmentos de programas:

a)
```
z = 0;
for (i = 0; i < n; i++)
  z += 1;
```

b)
```
i = 1;
x = 0;
while (i <= n)
{
  i += 3;
  x++;
}
```

c)
```
i = 1;
x = 0;
while (i <= n)
{
  i *= 3;
  x++;
}
```

d)
```
i = 1;
x = 0;
do {
  j = 1;
  while (j <= n)
  {
    j *= 2;
    x++;
  }
  i++;
} while (i <= n);
```

e)
```
i = 2;
x = 0;
do {
  j = 1;
  while (j <= i)
  {
    j *= 2;
    x++;
  }
  i++;
} while (i <= n);
```

f)
```
z = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    z += 1;
```

g)
```
z = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      z += 1;
```

h)
```
z = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < j; k++)
      z += 1;
```

i)
```
z = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < i; j++)
    for (k = 0; k < j; k++)
      z += 1;
```

2.6. Un algoritmo tarda 5 segundos en resolver un caso de un determinado problema de tamaño $n = 10.000$. Determinar el tiempo que tardará en resolver un caso de tamaño $n = 30.000$ en el supuesto de que la complejidad del algoritmo sea:

a) $O(n^2)$

b) $O(n^5)$

c) $O(n \log n)$

d) $O(2^n)$

e) $O(5^n)$

2.7. Un algoritmo A resuelve un problema de tamaño n en un tiempo n^2 minutos. Otro algoritmo B resuelve el mismo problema de tamaño n en n^3 milisegundos. Determinar el umbral n_0 a partir del cual el algoritmo A es mejor que el B.

2.8. Analizar la complejidad, expresándola en términos de notación *O-grande*, de las siguientes funciones recursivas:

a)
```
int funcion1(int n)
{
  if (n <= 1)
    return 1;
  else
    return n * funcion1(n-1);
}
```

b)
```
int funcion2(int n)
{
  if (n <= 1)
    return 1;
  else
    return n * funcion2(n % 2);
}
```

c)
```
int funcion3(int n)
{
  int x, i;
  if (n <= 1)
    return 1;
  else
  {
    x = funcion3(n % 2);
    for (i = 1; i <= n; i++)
      x++;
  }
}
```

d)
```
int funcion4(int n)
{
  int x, i;
  if (n <= 1)
    return 1;
  else
  {
    x = funcion4(n - 1);
    for (i = 1; i <= n; i++)
     x++;
  }
}
```

e)
```
int funcion5(int n)
{
  if (n <= 1)
    return 1;
  else
    return funcion5(n - 1) + funcion5(n - 1);
}
```

f)
```
int funcion6(int n)
{
  int x, i, j;
  if (n <= 1)
    return 1;
  else
  {
    x = funcion6(n % 2) + funcion6(n % 2);
    for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
        x++;
  }
}
```

PROBLEMAS

2.1. Escribir las funciones en C necesarias para resolver los siguientes problemas y calcular la complejidad, expresada con la *notación O*, de cada una de las funciones.

a) Calcular el máximo común divisor de dos números enteros positivos.
b) Encontrar el menor elemento de un vector de n elementos.
c) Calcular el número combinatorio:

$$\frac{m!}{n!(m-n)!}$$

2.2. Sean x, y dos números enteros positivos grandes de n dígitos (*tamaño n*).

a) Escribir un algoritmo que calcule $z = x + y$. Determinar la complejidad del algoritmo.
b) Escribir un algoritmo que calcule $z = x \cdot y$. Determinar la complejidad del algoritmo.
c) Escribir un algoritmo que calcule $z = x \cdot y$ que tenga la complejidad $O(n \log n)$.

2.3. Escribir las funciones necesarias para resolver los siguientes problemas y calcular la complejidad, expresada con la *notación O*, de cada uno de las funciones.

a) Evaluar un polinomio de grado n para $x = x_0$.
b) Calcular la suma de dos matrices cuadradas de orden $n \cdot n$.
c) Calcular el producto de dos matrices cuadradas de orden $n \cdot n$.

2.4. Responder a los siguientes supuestos:

a) Escribir un algoritmo que encuentre el elemento mayor de un vector de tamaño n en un tiempo $O(n)$.
b) Escribir un algoritmo que encuentre el elemento menor de un vector de tamaño n en un tiempo $O(n)$.
c) Escribir un algoritmo que encuentre el elemento mayor y menor de un vector de tamaño n en un tiempo $3/2n + c$, siendo c una constante.

2.5. Supóngase un vector de n elementos desordenado, se quiere encontrar el elemento central, es decir, aquel que, si se considera la magnitud de los elementos, hay igual número de elementos menores que de mayores en el vector.

a) Escribir un algoritmo que resuelva el problema en tiempo $O(n \log n)$.
b) Escribir un algoritmo que resuelva el problema en tiempo esperado $O(n)$ aunque en el peor caso pueda ser $O(n^2)$.

2.6. El problema matemático de calcular el determinante de una matriz cuadrada $n \cdot n$ se puede resolver mediante un algoritmo recursivo, o bien por el método de Gauss.

a) Escribir un algoritmo recursivo que calcule el determinante de una matriz y determinar su complejidad.

b) Aplicar el método de Gauss de resolución de sistemas de ecuaciones lineales para calcular el determinante y determinar la complejidad.

2.7. Determinar la complejidad de los siguientes problemas de cálculo numérico.

a) Escribir un algoritmo recursivo que calcule la inversa de una matriz cuadrada de orden n.

b) Aplicar el método de Gauss-Jordan de resolución de sistemas de ecuaciones lineales para calcular la matiz inversa de una matriz cuadrada de orden n.

3

ARRAYS (LISTAS Y TABLAS) EN C

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Conocer los tipos de datos estructurados en C.
- Conocer la forma de definir arrays en C.
- Acceder a los elementos de un array con variables subindicadas.
- Acceder a los elementos con la aritmética de punteros.
- Transmitir arrays, como argumentos, a las funciones escritas en C.
- Resolver problemas que implican una secuencia de elementos con arrays.

CONTENIDO

3.1. Arrays (arreglos).
3.2. *Inicialización de un array.*
3.3. Arrays de caracteres y cadenas de texto.
3.4. Arrays multidimensionales.
3.5. Utilización de arrays como parámetros.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Array (arreglo).
- Vector.
- Lista, tabla de elementos.
- Inicialización.
- Cadena de caracteres.
- Índice.
- Matriz.
- Array multidimensional.
- Parámetros de tipo array.

INTRODUCCIÓN

En los lenguajes de programación se distingue entre tipos de datos básicos o simples (carácter, entero y coma flotante) y tipos de datos estructurados. Los *arrays*[1] son tipos de datos estructurados; en una variable array se agrupan elementos todos del mismo tipo, por lo que a los arrays se les considera estructuras de datos homogéneas. En este capítulo se examina el tipo *array* (lista o tabla) y se aprenderá el concepto y tratamiento de los arrays. Un array almacena muchos elementos del mismo tipo, tales como veinte enteros, cincuenta números de coma flotante o quince caracteres. El array es muy importante por diversas razones; una muy importante es almacenar secuencias o *cadenas* de texto.

[1] En España se suele utilizar el término en inglés, y a veces se traduce por «lista, vector o tabla». En Latinoamérica está muy extendido el término «arreglo» como traducción del inglés *array*.

3.1. ARRAYS (ARREGLOS)

Un *array* es una secuencia de datos del mismo tipo. Los datos se llaman elementos del array y se numeran consecutivamente `0, 1, 2, 3...` El tipo de elementos almacenados en el array puede ser cualquier tipo de dato de C, incluyendo estructuras definidas por el usuario, como se describirá más tarde. Normalmente el array se utiliza para almacenar tipos tales como `char`, `int` o `float`.

Un array puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o el número de personas que residen en cada una de las diecisiete comunidades autónomas españolas. Cada ítem del array se denomina *elemento*.

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

Si el nombre del array es `a`, entonces `a[0]` es el nombre del elemento que está en la posición `0`, `a[1]` es el nombre del elemento que está en la posición `1`, etc.; en general, el elemento *i-ésimo* está en la posición $i - 1$, de modo que si el array tiene n elementos, sus nombres son `a[0], a[1],...,a[n - 1]`. En la Figura 3.1 se representa un array con seis elementos.

a	25.1	34.2	5.25	7.45	6.09	7.54
	0	1	2	3	4	5

Figura 3.1. Array de seis elementos.

El array `a` tiene 6 elementos: `a[0]` contiene `25.1`, `a[1]` contiene `34.2`, `a[2]` contiene `5.25`, `a[3]` contiene `7.45`, `a[4]` contiene `6.09` y `a[5]` contiene `7.54`. El diagrama de la Figura 3.1 representa realmente una región de la memoria de la computadora, ya que un array se almacena siempre con sus elementos en una secuencia de posiciones contiguas de memoria.

En C los índices de un array siempre tienen como límite inferior 0 como índice superior el tamaño del array menos 1.

3.1.1. Declaración de un array

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

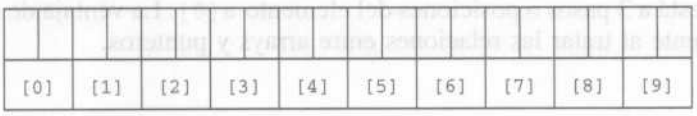
```
tipo nombreArray[numeroDeElementos];
```

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

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

Esta declaración hace que el compilador reserve espacio suficiente para contener diez valores enteros. En C los enteros ocupan, normalmente, 2 bytes, de modo que un array de diez enteros ocupa 20 bytes de memoria. La Figura 3.2 muestra el esquema de un array de diez elementos; cada elemento puede tener su propio valor.

Array de datos enteros: a



Un array de enteros se almacena en bytes consecutivos de memoria. Cada elemento utiliza dos bytes. Se accede a cada elemento de array mediante un índice que comienza en cero. Así, el elemento quinto (`a[4]`) del array ocupa los bytes 9.º y 10.º.

Figura 3.2. Almacenamiento de un array en memoria.

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

```
printf("%d \n",numeros[4]);
```

visualiza el valor del elemento 5 del array. Los arrays siempre comienzan en el elemento 0. Así, pues, el array `numeros` contiene los siguientes elementos individuales:

```
numeros[0]   numeros[1]   numeros[2]   numeros[3]
numeros[4]   numeros[5]   numeros[6]   numeros[7]
numeros[8]   numeros[9]
```

Si por ejemplo, se quiere crear un array de números reales cuyo tamaño sea una constante representada por un parámetro, las sentencias serán:

```
#define N 20
float vector[N];
```

Para acceder al elemento 3 y leer un valor de entrada:

```
scanf("%f ",&vector[2]);
```

Precaución

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

3.1.2. Subíndices de un array

El índice de un array se denomina, con frecuencia, *subíndice del array*. El término procede de las matemáticas, donde un subíndice se utiliza para representar un elemento determinado.

$numeros_0$	*equivale a*	`numeros[0]`
$numeros_3$	*equivale a*	`numeros[3]`

El método de numeración del elemento *i-ésimo* con el índice o subíndice $i - 1$ se denomina *indexación basada en cero*. Su uso tiene el efecto de que el índice de un elemento del array es

siempre el mismo que el número de «pasos» desde el elemento inicial `a[0]` a ese elemento. Por ejemplo, `a[3]` está a 3 pasos o posiciones del elemento `a[0]`. La ventaja de este método se verá de modo más evidente al tratar las relaciones entre arrays y punteros.

Ejemplos

`int edad[5];`	*Array* `edad` *contiene 5 elementos: el primero,* `edad[0]` *y el último,* `edad[4]`.
`int pesos[25], longitudes[100];`	*Declara 2 arrays de enteros.*
`float salarios[25];`	*Declara un array de 25 elementos* `float`.
`double temperaturas[50];`	*Declara un array de 50 elementos* `double`.
`char letras[15];`	*Declara un array de caracteres.*
`#define MX 120` `char buffer[MX+1];`	*Declara un array de caracteres de tamaño* `MX+1`, *el primer elemento es* `buffer[0]` *y el último* `buffer[MX]`.

En los programas se pueden referenciar elementos del array utilizando fórmulas para los subíndices. Como el subíndice se evalúa a un entero, se puede utilizar una constante, una variable o una expresión para el subíndice. Así, algunas referencias individuales a elementos son:

```
edad[4]
ventas[total+5]
bonos[mes]
salario[mes[i]*5]
```

3.1.3. Almacenamiento en memoria de los arrays

Los elementos de los arrays se almacenan en bloques contiguos. Así, por ejemplo, los arrays

```
int edades[5];
char codigos[5];
```

se representan gráficamente en memoria en la Figura 3.3.

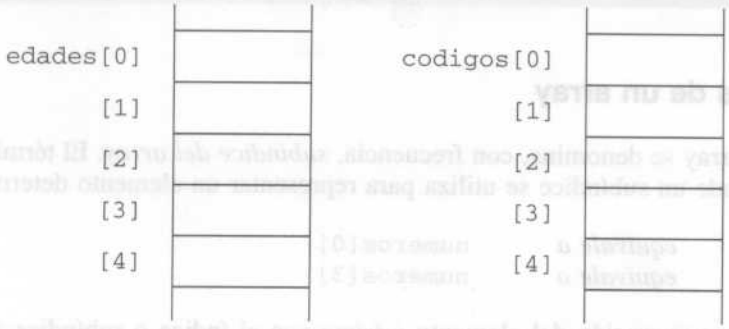


Figura 3.3. Almacenamiento en memoria de arrays.

Nota de programación

Todos los subíndices de los arrays en C comienzan con 0.

Precaución

C permite asignar valores fuera de rango a los subíndices. Se debe tener cuidado en no hacer esta acción, debido a que se sobreescribirían datos o código.

Los arrays de caracteres funcionan de igual forma que los arrays numéricos, partiendo de la base de que cada carácter ocupa normalmente un byte. Así, por ejemplo, un array llamado `nombre` se puede representar en la Figura 3.4.

```
char nombre[] = "Cazorla"
char zona[10] = "La Betica"
```

nombre	
C	[0]
a	[1]
z	[2]
o	[3]
r	[4]
l	[5]
a	[6]
	[7]

Figura 3.4. Almacenamiento de un array de caracteres en memoria.

Debe tenerse en cuenta que en las cadenas de caracteres el sistema siempre inserta un último carácter (nulo) para indicar fin de cadena.

3.1.4. El tamaño de los arrays

El operador `sizeof` devuelve el número de bytes necesarios para contener su argumento. Si se usa `sizeof` para solicitar el tamaño de un array, esta función devuelve el número de bytes reservados para el array completo.

Por ejemplo, supongamos que se declara un array de enteros de 100 elementos denominado `edades`; si se desea conocer el tamaño del array, se puede utilizar una sentencia similar a:

```
n = sizeof(edades);
```

donde `n` tomará el valor 200. Si se desea solicitar el tamaño de un elemento individual del array, tal como

```
n = sizeof(edades[6]);
```

`n` almacenará el valor 2 (número de bytes que contienen un entero).

3.1.5. Verificación del rango del índice de un array

C, al contrario que otros lenguajes de programación —por ejemplo, Pascal—, no verifica el valor del índice de la variable que representa al array. En el caso de Pascal si se define un array `a[]` con índices 0 a 5, entonces `a[6]` hará que el programa se «rompa» en tiempo de ejecución. Por el contrario, en C, `a[6]` no provoca que el programa se «rompa» en tiempo de ejecución, sin embargo los resultados del programa no se corresponderá con la solución buscada.

Ejemplo 3.1

Protección frente a errores en el intervalo (rango) de valores de una variable de índice que representa un array.

```
double suma(double a[], int n)
{
  int i;
  double s = 0.0;
  if (n * sizeof(double) > sizeof(a))
    return 0;
  for (i = 0; i < n; i++)
    s += a[i];
  return s;
}
```

3.2. INICIALIZACIÓN DE UN ARRAY

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

```
precios[0] = 10;
precios[1] = 20;
precios[3] = 30;
precios[4] = 40;
...
```

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

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

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

El array `numeros` tiene 6 elementos, `n` tiene 3 elementos y el array `c` tiene 4 elementos.

En C los arrays de caracteres, las cadenas, se caracterizan por tener un carácter final que indica el fin de la cadena, es el carácter nulo `'\0'`. Lo habitual es inicializar un array de caracteres (una variable cadena) con una constante cadena.

```
char s[] = "Puesta del Sol";
```

Nota de programación

C puede dejar los corchetes vacíos, sólo cuando se asignan valores al array, tal como

```
int cuenta[] = {15, 25, -45, 0, 50};
```

El compilador asigna automáticamente cinco elementos a cuenta.

El método de inicializar arrays mediante valores constantes después de su definición es adecuado cuando el número de elementos del array es pequeño. Por ejemplo, para inicializar un array de 10 enteros a los valores 10 a 1, y a continuación visualizar dichos valores en un orden inverso, se puede escribir:

```
int cuenta[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
for (i = 9; i >= 0; i--)
  printf("\n cuenta descendente %d = %d",i,cuenta[i]);
```

Pueden asignarse valores a un array utilizando un bucle `for` o `while/do-while`, y éste suele ser el sistema más empleado normalmente. Por ejemplo, para inicializar todos los valores del array `numeros` al valor 0, se puede utilizar la siguiente sentencia:

```
for (i = 0; i <= 5; i++)
  numeros[i] = 0;
```

debido a que el valor del subíndice *i* varía de 0 a 5, cada elemento del array `numeros` se inicializa y establece a cero.

Ejemplo 3.2

El siguiente programa lee un número determinado de valores enteros; a continuación visualiza el total de los valores que son positivos.

La cantidad de enteros a leer se establece en una constante simbólica. Esta constante se utiliza para definir el array y para el bucle de lectura. El beneficio que tiene definir la constante simbólica es que para cambiar el número de elementos, el tamaño del array y toda referencia a ese tamaño, sólo hay que modificar la sentencia que da valor a la constante simbólica.

```
#include <stdio.h>

#define NUM 18

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

  for (i = 0; i < NUM; i++)
  {
    printf("Por favor, introduzca el número: ");
    scanf("%d",&nums[i]);
  }
  printf("\nLista de números: ");
```

```
    for (i = 0; i < NUM; i++)
    {
      total += ((nums[i]>0 ? nums[i] : 0);
    }

    printf("\nLa suma de los números es %d",total);
    return 0;
  }
```

Las variables globales que representan arrays se inicializan por defecto a 0. En el ejemplo

```
int Notas[5];
void main()
{
  static char Nombres[5];
}
```

el array de enteros se ha definido globalmente y el array de caracteres como un array local estático de `main()`. Si se ejecuta ese segmento de programa, se obtendrán las siguientes asignaciones a los elementos de los arrays:

Notas		Nombres	
[0]	0	[0]	'\0'
[1]	0	[1]	'\0'
[2]	0	[2]	'\0'
[3]	0	[3]	'\0'
[4]	0	[4]	'\0'

Nota de programación

Si se define un array globalmente o un array estático y no se proporciona ningún valor de inicialización, el compilador inicializará el array con un valor por defecto (cero para arrays de elementos enteros y reales —coma flotante— y carácter nulo para arrays de caracteres). En cualquier caso, es buena práctica de programación inicializar los arrays explícitamente y no esperar que lo haga el sistema

3.3. ARRAYS DE CARACTERES Y CADENAS DE TEXTO

Una cadena de texto es un conjunto de caracteres, tales como `"ABCDEFG"`. C soporta cadenas de texto utilizando un array de caracteres que contenga una secuencia de caracteres:

```
char cadena[] = "ABCDEFG";
```

Es importante comprender la *diferencia* entre un array de caracteres y una cadena de caracteres o, simplemente, una *cadena*. Las *cadenas* contienen un carácter nulo `('\0')` al final del array de 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.

Nota

Las cadenas se deben almacenar en arrays de caracteres, pero no todos los arrays de caracteres contienen cadenas.

Examine la Figura 3.5, donde se muestra una cadena de caracteres y un array de caracteres.

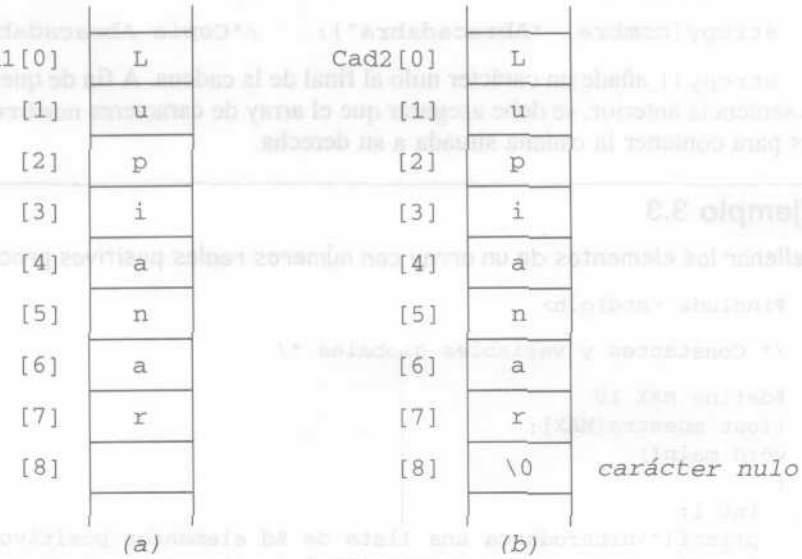


Figura 3.5. (*a*) Array de caracteres; (*b*) cadena.

Las cadenas se señalan incluyendo un carácter al final de la cadena: el *carácter nulo* (`\0`), cuyo valor en el código ASCII es 0. El medio más fácil de inicializar un array de caracteres es hacer la inicialización de la declaración:

```
char Cadena[7] = "ABCDEF";
```

El compilador inserta automáticamente un carácter nulo al final de la cadena, de modo que la secuencia real sería:

```
char Cadena[7] = "ABCDEF";
```

Cadena	A	B	C	D	E	F	\0

La asignación de valores a `Cadena` se puede hacer del modo siguiente:

```
Cadena[0] = 'A';
Cadena[1] = 'B';
Cadena[2] = 'C';
Cadena[3] = 'D';
Cadena[4] = 'E';
Cadena[5] = 'F';
Cadena[6] = '\0';
```

Sin embargo, no se puede asignar una cadena a un array del siguiente modo:

```
Cadena = "ABCDEF";
```

Para copiar una constante cadena o copiar una variable de cadena a otra variable de cadena se debe utilizar la función de la biblioteca estándar `strcpy()` («copiar cadenas»). `strcpy()` permite

copiar una constante de cadena en una cadena. Para copiar el nombre `"Abracadabra"` en el array `nombre`, se puede escribir

```
strcpy(nombre, "Abracadabra");    /*Copia Abracadabra en nombre */
```

`strcpy()` añade un carácter nulo al final de la cadena. A fin de que no se produzcan errores en la sentencia anterior, se debe asegurar que el array de caracteres `nombre` tenga elementos suficientes para contener la cadena situada a su derecha.

Ejemplo 3.3

Rellenar los elementos de un array con números reales positivos procedentes del teclado.

```
#include <stdio.h>

/* Constantes y variables globales */

#define MAX 10
float muestra[MAX];
void main()
{
  int i;
  printf("\nIntroduzca una lista de %d elementos positivos.\n",MAX);
  for (i = 0; i < MAX; muestra[i]>0 ? ++i : i)
    scanf("%f",&muestra[i]);
}
```

En el bucle principal, sólo se incrementa `i` si `muestra[i]` es positivo: `muestra[i]>0 ? ++i : i`. Con este incremento condicional se consigue que todos los valores almacenados sean positivos.

Ejemplo 3.4

Visualizar los elementos del array `muestra[]` *que están en el rango de valores* `[5,10]`.

Un sencillo bucle permite recorrer todos los elementos del array; la selección pedida se realiza con una expresión que comprueba si el elemento del array está en el rango. Los límites del rango se declan con una constante simbólica.

```
#define LIM_INF 5.0
#define LIM_SUP 10.0

printf("\n Elementos en el rango: ");
for ( i = 0, i < MAX; ++i)
  if (muestra[i]>=LIM_INF && muestra[i]<=LIM_SUP)
    printf("%f\t",muestra[i]);
```

3.4. ARRAYS MULTIDIMENSIONALES

Los arrays vistos anteriormente se conocen como arrays *unidimensionales* (una sola dimensión) y se caracterizan por tener un sólo subíndice. Estos arrays se conocen también por el término *listas*. 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.

Un array de dos dimensiones equivale a una tabla con múltiples filas y múltiples columnas (Figura 3.6).

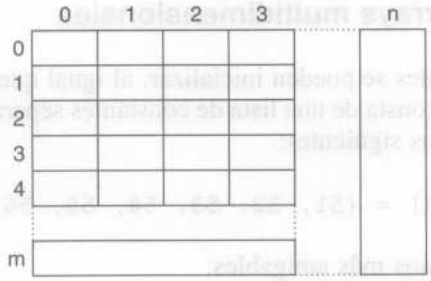


Figura 3.6. Estructura de un array de dos dimensiones.

Obsérvese que en el array bidimensional de la Figura 3.6, si las filas se etiquetan de `0` a `m` y las columnas de `0` a `n`, el número de elementos que tendrá el array será el resultado del producto `(m+1) * (n+1)`. El sistema de localizar un elemento será por las coordenadas representadas por su número de fila y su número de columna `(a, b)`. La sintaxis para la declaración de un array de dos dimensiones es:

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

Algunos ejemplos de declaración de tablas:

```
char pantalla[25][80];
int puestos[6][8];
int equipos[4][30];
int matriz[4][2];
```

Atención

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

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

Los elementos de los arrays se almacenan en memoria de modo que el subíndice más próximo al nombre del array es la fila y el otro subíndice, la columna. En la Tabla 3.1 se representan todos los elementos y sus posiciones relativas en memoria del array `int tabla[4][2]` y se supone que cada entero ocupa 2 bytes.

Tabla 3.1. Un array bidimensional

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

3.4.1. Inicialización de arrays multidimensionales

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

1. `int tabla1[2][3] = {51, 52, 53, 54, 55, 56};`

   o bien en los formatos más amigables:

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

2. `int tabla2[3][4] = {`

```
                    {1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 10, 11, 12}
                  };
```

Tabla[2][3]

		0	1	2	Columnas
Filas	0	51	52	53	
	1	54	55	56	

Tabla[3][4]

		0	1	2	3	Columnas
Filas	0	1	2	3	4	
	1	5	6	7	8	
	2	9	10	11	12	

Figura 3.7. Tablas de dos dimensiones.

Consejo

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

```
float ventas[3][4] = {0.,0.,0.,0., 0.,0.,0.,0., 0.,0.,0.,0.};
```

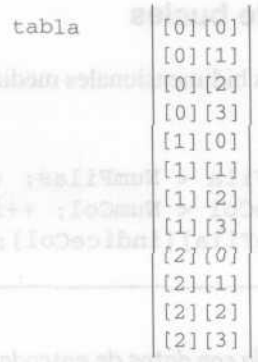


Figura 3.8. Almacenamiento en memoria de tabla [3] [4].

3.4.2. Acceso a los elementos de los arrays bidimensionales

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

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

```
inserción de elementos
  <nombre array>[indice fila][indice columna] = valor elemento;

extracción de elementos
  <variable> = <nombre array>[indice fila][indice columna];
```

Algunos ejemplos de inserciones:

```
Tabla[2][3] = 4.5;
Resistencias[2][4] = 50;
AsientosLibres[5][12] = 5;
```

y de extracción de valores:

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

3.4.3. Lectura y escritura de elementos de arrays bidimensionales

Las funciones de entrada o salida se aplican de igual forma a los elementos de un array bidimensional. Por ejemplo,

```
int tabla[3][4];
double resistencias[4][5];

scanf("%d",&tabla[2][3]);
printf("%4d",tabla[1][1]);
scanf("%lf",&resistencias[2][4]);

if (asientosLibres[3][1])
  puts("VERDADERO");
else
  puts("FALSO");
```

3.4.4. Acceso a elementos mediante bucles

Se puede acceder a los elementos de arrays bidimensionales mediante bucles anidados. Su sintaxis es:

```
int indiceFila, indiceCol;

for (indiceFila = 0; indiceFila < NumFilas; ++indiceFila)
  for (indiceCol = 0; indiceCol < NumCol; ++indiceCol)
    Procesar elemento[indiceFila][indiceCol];
```

Ejemplo 3.5

Definir una tabla de discos; rellenar la tabla con datos de entrada y mostrar dichos datos por pantalla.

```
float discos[2][4];
int fila, col;

for (fila = 0; fila < 2; fila++) {
  for (col = 0; col < 4; col++)
  {
    scanf("%f",&discos[fila][col]);
  }
}

/* Visualizar la tabla */
for (fila = 0; fila < 2; fila++)
{
  for (col = 0; col < 4; col++)
  {
    printf("\n Pts %.1f \n",discos[fila][col]);
  }
}
```

Ejercicio 3.1

Lectura y visualización de un array de dos dimensiones.

La función `leer()` lee un array (una tabla) de dos dimensiones y la función `visualizar()` presenta la tabla en la pantalla.

```
#include <stdio.h>
/* prototipos */
void leer(int a[][5]);
void visualizar(const int a[][5]);

int main()
{
  int a[3][5];
  leer(a);
  visualizar(a);
  return 0;
}

void leer(int a[][5])
{
  int i,j;
```

```
  puts("Introduzca 15 números enteros, 3 por fila");
  for (i = 0; i < 3; i++)
  {
    printf("Fila %d: ", i);
    for (j = 0; j < 5; j++)
      scanf("&d",&a[i][j]);
  }
}
void visualizar (const int a[][5])
{
  int i,j;

  for (i = 0; i < 3; i++)
  {
    for (j = 0; j < 5; j++)
      printf(" %d",a[i][j]);
    printf("\n");
  }
}
```

3.4.5. Arrays de más de dos dimensiones

C proporciona la posibilidad de almacenar varias dimensiones, aunque raramente los datos del *mundo real requieren más de dos o tres dimensiones. El medio más fácil de dibujar un array de tres* dimensiones es imaginar un cubo tal como se muestra en la Figura 3.9.

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

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

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

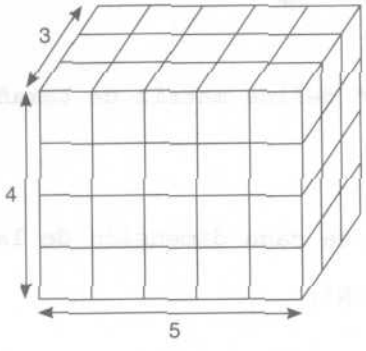


Figura 3.9. Un array de tres dimensiones (4 × 5 × 3).

3.4.6. Una aplicación de un array tridimensional

El array `libro` tiene tres dimensiones, `[PAGINAS] [LINEAS] [COLUMNAS]`, que definen el tamaño del array. El tipo de datos del array es `char`, ya que los elementos son caracteres.

¿Cómo se puede acceder a la información del libro? El método más fácil es mediante bucles anidados. Dado que el libro se compone de un conjunto de páginas, el bucle más externo será el bucle de página; y el bucle de columnas, el bucle más interno. Esto significa que el bucle de líneas se insertará entre los bucles página y columna. El código siguiente permite procesar el array.

```
int pagina, linea, columna;

  int libro[PAGINAS][LINEAS][COLUMNAS];

  for (pagina = 0; pagina < PAGINAS; ++pagina)
    for (linea = 0; linea < LINEAS; ++linea)
      for (columna = 0; columna < COLUMNAS; ++columna)
        <procesar Libro[pagina][linea][columna]>
```

Ejercicio 3.2

Comprobar si una matriz de números enteros es simétrica respecto a la diagonal principal.

La matriz se genera internamente, con la función `random()` y argumento `N(8)` para que la matriz tenga valores de 0 a 7. El tamaño de la matriz se pide como dato de entrada, aunque tiene que ser como máximo la constante `N`. La función `simetrica()` analiza si la matriz es simétrica. La función `main()` genera matrices hasta encontrar una que sea simétrica y la escribe en pantalla.

```
/*
    Determina si una matriz es simétrica. La matriz se genera con
    números aleatorios de 0 a 7. El programa itera hasta encontrar una
    matriz simétrica.
*/

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

void genMat(int a[][N], int n);
int simetrica(int a[][N], int n);
void escribeMat(int a[][N], int n);

int main(void)
{
  int a[N][N];      /* define matriz de tamaño máximo N */
  int n,i,j;
  int esSim;
  randomize();
  do {
    printf("\nTamaño de cada dimensión de la matriz, máximo %d: ",N);
    scanf("%d",&n);
    }while (n<2 && n>N);

  do {
    genMat(a,n);
```

```
      esSim = simetrica(a,n);

    if (esSim)
    {

      puts("\n\Encontrada matriz simétrica.\n");
      escribeMat(a,n);
      }

    }while (!esSim);

    return 0;
  }

  void genMat(int a[][N], int n)
  {
    int i,j;

    for (i=0; i<n; i++)
      for (j=0; j<n; j++)
      a[i][j]= random(N);
  }

  int simetrica(int a[][N], int n)
  {
    int i,j;
    int 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 escribeMat(int a[][N], int n)
  {
    int i,j;
    puts("\tMatriz analizada");
    puts("\t----- ---------\n");
    for (i=0; i<n; i++)
    { putchar('\t');
      for (j=0; j<n; j++)
      printf("%d %c",a[i][j],(j==n-1 ?'\n':' '));
    }
}
```

3.5. UTILIZACIÓN DE ARRAYS COMO PARÁMETROS

En C *todos los arrays se pasan por referencia* (dirección). Esto significa que cuando se llama a una función y se utiliza un array como parámetro, se debe tener cuidado de no modificar los arrays en 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 Figura 3.10 ayuda a comprender el mecanismo.

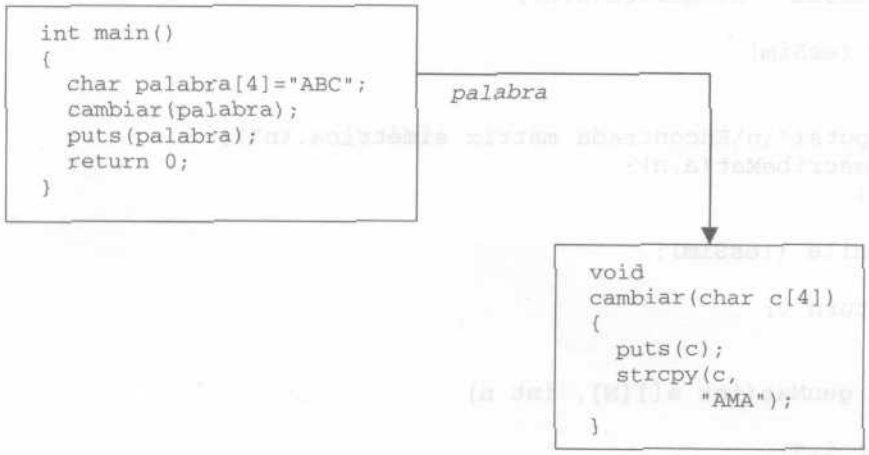


Figura 3.10. Paso de un array por dirección.

Dadas las declaraciones

```
#define MAX 100
double datos[MAX];
```

se puede declarar una función que acepte un array de valores `double` como parámetro. Así un posible prototipo de la función `SumaDeDatos()`:

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

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

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

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

```
printf("\nSuma = %lf",SumaDeDatos(datos, MAX));
```

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

```
double SumaDeDatos(double datos[], int n)
{
  double suma = 0;

  while (n > 0)
    suma += datos[--n];

  return suma;
}
```

El código que se utiliza para pasar un array a una función incluye el tipo de elemento del array y su nombre. El siguiente ejemplo incluye dos funciones que procesan arrays. En ambas listas de parámetros, el array `a[]` se declara en la lista de parámetros tal como

```
double a[]
```

El número real de elementos se pasan mediante una variable entera independiente. Cuando se pasa un array a una función, realmente se pasa *sólo* la dirección de la celda de memoria donde comienza el array. Este valor se representa por el nombre del array `a`. La función puede cambiar entonces el contenido del array accediendo directamente a las celdas de memoria en donde se almacenan los elementos del array. Así, aunque el nombre del array se pasa por valor, sus elementos se pueden cambiar como si se hubieran pasado por referencia.

Ejemplo 3.6

Paso de arrays a funciones. En el ejemplo se lee un array y se escribe.

El array se define con un tamaño máximo, `L`, aunque el número real de elementos que se van a procesar se obtiene en la función de entrada `leerArray()`. Por ello el segundo argumento de dicha función es un puntero a entero; de esa forma se transmite por referencia y es posible obtener el número de elementos del array.

```
#include <stdio.h>
#define L 100
void leerArray(double a[], int* num );
void imprimirArray (const double [], int n);

int main()
{
  double a[L];
  int n;

  leerArray(a, &n);
  printf("El array a tiene %d elementos, estos son: \n ",n);
  imprimirArray(a, n);

  return 0;
}

void leerArray(double a[], int* num)
{
  int n = 0;
  puts("Introduzca datos. Para terminar pulsar 0.\n");
  for (; n < L; n++)
  {
    printf("%d: ",n);
    scanf("%lf",&a[n]);
    if (a[n] == 0) break;
  };
  *num = n;
}

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

3.5.1. Precauciones

Cuando se utiliza una variable array como argumento, la función receptora puede no conocer cuántos elementos existen en el array. Sin su conocimiento una función no puede utilizar el array. Aunque la variable array puede apuntar al comienzo de él, no proporciona ninguna indicación de dónde termina el array.

Observe el siguiente segmento de un programa; en él la función `SumaDeEnteros()` suma los valores de todos los elementos de un array y devuelve el total:

```
int SumaDeEnteros(int *ArrayEnteros)
{
   ...
}

int main()
{
   int lista[5] = {10, 11, 12, 13, 14};
   SumaDeEnteros (lista);
   ...
}
```

Aunque `SumaDeEnteros()` conoce dónde comienza el array, no conoce cuántos elementos hay en el array; en consecuencia, no sabe cuántos elementos hay que sumar.

Nota de programación

Se pueden utilizar dos métodos alternativos para permitir que una función conozca el número de elementos asociados con un array que se pasa como argumento de una función:

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

Todas las cadenas de caracteres utilizan el primer método ya que terminan en el carácter nulo. Una segunda alternativa es pasar el número de elementos del array siempre que se pasa el array como un argumento. El array y el número de elementos se convierten entonces en una pareja de argumentos que se asocian con la función llamada. La función `SumaDeEnteros()`, por ejemplo, se puede actualizar así:

```
int SumaDeEnteros(int arrayEnteros[], int numElementos)
{
   ...
}
```

El segundo argumento, `numElementos`, es un valor entero que indica a la función `SumaDeEnteros()` cuántos elementos se procesarán en el array `arrayEnteros`.

Este método suele ser el utilizado para arrays de elementos que no son caracteres.

Ejemplo 3.7

Este programa introduce una lista de 10 números enteros y calcula su suma y el valor máximo.

```
#include <stdio.h>
int SumaDeEnteros(const int arrayEnteros[], int numElementos);
int maximo(const int arrayEnteros[], int numElementos);

int main()
{
  int items[10];
  int total, i;

  puts("Introduzca 10 números, seguidos por return");
  for (i = 0; i < 10; i++)
    scanf("%d",&items[i]);

  printf("Total = %d \n",SumaDeEnteros(items,10));
  printf("Valor máximo: %d \n",maximo(items,10));
  return 0;
}

int SumaDeEnteros(const int arrayEnteros[], int numElementos)
{
  int i, total = 0;
  for (i = 0; i < numElementos; i++)
    total += arrayEnteros[i];

  return total;
}
int maximo(const int arrayEnteros[], int numElementos)

{
  int mx;
  int i;

  mx = arrayEnteros[0];
  for (i = 1; i < numElementos; i++)
    mx = (arrayEnteros[i]>mx ? arrayEnteros[i]: mx);

  return mx;
}
```

El siguiente programa muestra cómo se pasa un array de enteros a una función de ordenación, `ordenar()`. En este caso el tamaño del array se determina con el operador `sizeof`.

```
#include <stdio.h>
void ordenar(int v[],int);      /*prototipo de ordenar*/

int main()
{
  int ListaEnt[ ] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 10};
  int i;
  int LongLista = sizeof(ListaEnt) / sizeof(int);
```

```
    ordenar(ListaEnt,LongLista);
    for (i = 0; i < LongLista; i++)
      printf("%d ",ListaEnt[i]);
    return 0;
  }

  void ordenar(int lista[],int numElementos)
  {
    /*cuerpo de la función ordenar el array */
  }
```

Como C trata las cadenas como arrays de caracteres, las reglas para pasar arrays como argumentos a funciones se aplican también a cadenas. El siguiente ejemplo de una función de cadena que convierte los caracteres de sus argumentos a mayúsculas, muestra el paso de parámetros tipo cadena.

```
void convierte_mayus(char cad[])
{
  int i = 0;
  int intervalo = 'a'-'A ';
  while (cad[i])
  {
    cad[i] = (cad[i]>='a' &&cad[i]<='z') ? cad[i]-intervalo: cad[i];
    i++;
  }
}
```

La función `convierte_mayus()` recibe una cadena, un array de caracteres cuyo último carácter es el nulo `('\0')`. El bucle termina cuando se alcance el fin de cadena (nulo, condición *false*). La expresión condicional del operador ternario determina si el carácter es minúscula, en cuyo caso resta a dicho carácter el intervalo que hay entre las minúsculas y las mayúsculas.

3.5.2. Paso de cadenas como parámetros

La técnica de pasar arrays como parámetros se utiliza para pasar cadenas de caracteres a funciones. Las cadenas terminadas en nulo utilizan el primer método dado anteriormente para controlar el tamaño de un array. Las cadenas son arrays de caracteres. Cuando una cadena se pasa a una función, por ejemplo, a `strlen(const char* cad)`, esta función *conoce* que se ha almacenado en el array el carácter *fin de cadena* como último elemento, razón por la cual itera por cada carácter hasta encontrar dicho carácter.

Las cadenas utilizan siempre un 0 para indicar que es el último elemento del array de carácteres. Este 0 es el carácter nulo del código de caracteres ASCII.

Considérese las declaraciones de una constante y de una función que acepta un parámetro tipo cadena y otro parámetro entero, el valor de su longitud.

```
#define MAXLON 128
void funcDemo(char s[],int longitud);
```

El parámetro `s` es un array de caracteres de longitud no especificada; el parámetro `longitud` indica a la función cuántos bytes ocupa (que puede ser diferente del número de caracteres almacenados en `s`). En estas declaraciones:

```
char presidente[MAXLON] = "Manuel Martínez";
funcDemo(presidente, MAXLON);
```

la primera línea declara e inicializa un array de caracteres llamado `presidente`, capaz de almacenar hasta `MAXLON-1` caracteres más un byte de terminación, carácter nulo. La segunda línea, pasa la cadena a la función.

Ejemplo 3.8

El siguiente programa busca todas las ocurrencias de un elemento y la posición que ocupa en una matriz. La posición viene dada por fila y columna; la matriz se genera con números aleatorios de 0 a 49.

La función de búsqueda devuelve `0` si no encuentra al elemento, `1` si lo encuentra. Tiene el argumento de la matriz y dos parámetros para devolver la fila y columna que deberán ser de tipo puntero para poder devolver dicha información. La búsqueda se hará a partir de la fila y columna de la última coincidencia.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define F 8
#define C 10
#define N 50

void escribe_mat(int a[][C]);
void gen_mat(int a[][C]);
int buscar(int a[][C],int* fila,int* col,int elemento);

int main()
{
  int a[F][C];
  int item, nf,nc,esta;
  int veces = 0;

  randomize();
  gen_mat(a);
  printf("\n Elemento a buscar: ");
  scanf("%d",&item);

  do {
    esta = buscar(a,&nf,&nc,item);
    if (esta)
    {
      veces = veces+1;
      printf("\n coincidencia %d: Fila %d, Columna %d\n",veces,nf,nc);
    }
  }while (esta);
  escribe_mat(a);
  printf("\nNúmero de coincidencias del elemento %d :%d",item,veces);
  return 0;
}

/* Busqueda lineal en toda la matriz */
int buscar(int a[][C],int* fila,int* col,int elemento)
{
  static int x = 0, y = -1;
  int i,j,encontrado;

  /* avanza al siguiente elemento(fila,columna) */
```

```
    if (y == C-1)        /* ultima columna */
    {
      y = 0;
      x = x+1;
    }
    else
      y = y+1;
    encontrado = 0;
    while (!encontrado &&(x<F))
    {
      encontrado = (a[x][y] == elemento);
      if (!encontrado)        /* avanza a siguiente elemento */
        if (y == C-1)
        {
          y = 0;
          x = x+1;
        }
        else
          y = y+1;
    }
                                /* ultimo valor de x e y */
    *fila = x;
    *col = y;
    return encontrado;
  }

  void gen_mat(int a[][C])
  {
    int i,j;

    for (i=0; i<F; i++)
      for (j=0; j<C; j++)
        a[i][j]= random(N);
  }

  void escribe_mat(int a[][C])
  {
    int i,j;
    puts("\t\tMatriz analizada");
    puts("\t\t----- ---------\n");
    for (i=0; i<F; i++)
    { putchar('\t');
      for (j=0; j<C; j++)
        printf("%d %c",a[i][j],(j==C-1 ?'\n ':' '));
    }
  }
```

Ejemplo 3.9

En este programa se desea buscar la fila de una matriz de números reales que tiene la propiedad de ser máxima la suma de sus elementos en valor absoluto. La matriz se genera con números aleatorios, las dimensiones de la matriz se establecen con una constante predefinida.

Para determinar la suma de una fila se define la función `sumar()`, a la cual se pasa la dirección del primer elemento de la fila al objeto de tratar cada fila como un array unidimensional.

La generación de números aleatorios de tipo real se puede hacer dividiendo el número que devuelve la función `rand()` entre `100.0`

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define F 6
#define C 10
#define V 100.0

void escribe_mat(float mt[][C]);
void gen_mat(float mt[][C]);
float sumar(float v[]);
int maximo(float mt[][C]);

int main()
{
  float mat[F][C];
  int fila;

  randomize();
  gen_mat(mat);
  escribe_mat(mat);

  fila = maximo(mat);
  printf("\n\nFila cuya suma de elementos es mayor: %d",fila);

  return 0;
}

void gen_mat(float mat[][C])
{
  int i,j;
  for (i=0; i<F; i++)
    for (j=0; j<C; j++)
      mat[i][j]= rand()/V;
}

void escribe_mat(float mat[][C])
{
  int i,j;
  puts("\n\t\tMatriz analizada\n");
  puts("\t\t----- ---------\n");
  for (i=0; i<F; i++)
  {
    for (j=0; j<C; j++)
      printf("%.2f%c",mat[i][j],(j==C-1 ?'\n ':' '));
  }
}

float sumar(float v[])
{
  int i;
  float s;
  for (s=0.0,i=0; i<C; i++)
    s += v[i];
  return s;
}
```

```
int maximo(float mt[][C])
{
  float mx;
  int i,f;

  mx = sumar(&mt[0][0]); /* dirección de primera fila */
  printf("\nSuma fila %d %.2f",0,mx);
  for (f=0,i=1; i<F; i++)
  {
    float t;
    t = sumar(&mt[i][0]);
    printf("\nSuma fila %d %.2f",i,t);
    if (t > mx)
    {
      mx = t;
      f = i;
    }
  }
  return f;
}
```

RESUMEN

En este capítulo se analizan los *arrays* como tipos agregados de C. Después de leer este capítulo debe tener un buen conocimiento de los conceptos fundamentales de los tipos agregados.

Se describen y analizan los siguientes conceptos:

- Un *array* es un tipo de dato estructurado que se utiliza para localizar y almacenar elementos de un tipo de dato dado.
- Existen arrays de una dimensión, de dos dimensiones... y multidimensionales.
- En C, los arrays se definen especificando el tipo de dato del elemento, el nombre del array y el tamaño de cada dimensión del array. Para acceder a los elementos del array se deben utilizar sentencias de asignación directas, sentencias de lectura/escritura o bucles (mediante las sentencias `for`, `while` o `do-while`).

```
int totalMeses[12];
```

- Los arrays de caracteres contienen cadenas de textos. En C se terminan las cadenas de caracteres situando un carácter nulo (`'\0'`) como último byte de la cadena.
- C soporta arrays multidimensionales.

```
int ventasTotales[12][50];
```

EJERCICIOS

Para los ejercicios 1 a 6 suponer las declaraciones:

```
int i,j,k;
int Primero[21], Segundo[21];
int Tercero[6][12];
```

Determinar la salida de cada segmento de programa (en los casos que se necesite, se indica debajo el archivo de datos de entrada correspondiente).

3.1.
```
for (i=1; i<=6; i++)
  scanf("%d",&Primero[i]);
for(i= 3; i>0; i--)
  printf("%4d",Primero[2*i]);
.................
3   7   4   -1   0   6
```

3.2.
```
scanf("%d",&k);
for(i=3; i<=k;)
  scanf("%d",&Segundo[i++]);
j= 4;
printf("%d %5d\n",Segundo[k],Segundo[j+1]);
.............
6   3   0   1   9
```

3.3.
```
for(i= 0; i<10;i++)
  Primero[i] = i + 3;
scanf("%d %d",&j,&k);
for(i= j; i<=k;)
  printf("%d\n",Primero[i++]);
............
7   2   3   9
```

3.4.
```
for(i=0, i<12; i++)
  scanf("%d",&Primero[i]);
for(j=0; j<6;j++)
  Segundo[j]=Primero[2*j] + j;
for(k=3; k<=7,k++)
  printf("%d %d \n",Primero[k+1],Segundo [k-1]);
...........
2   7    3   4   9   -4
6   -5   0   5   -8   1
```

3.5.
```
for(j= 0; j<7; )
  scanf("%d",&Primero[j++]);
i = 0;
j = 1;
while ((j< 6) &&(Primero[j-1]<Primero[j]))
{
    i++,j++;
}
for(k= -1; k<j+2; )
  printf("%d",Primero[++k]);
..........
20   60   70   10   0   40   30   90
```

3.6.
```
for(i= 0; i< 3; i++)
  for(j= 0; j<12; j++)
    Tercero[i][j] = i+j+1;
for(i= 0; i< 3;i++)
{
  j = 2;
  while (j < 12)
  {
```

```
        printf("%d \n",i,j,Tercero[i][j]);
        j+=3;
    }
}
```

3.7. Escribir un programa que lea el array

```
4   7   1   3   5
2   0   6   9   7
3   1   2   6   4
```

y lo escriba como

```
4   2   3
7   0   1
1   6   2
3   9   6
5   7   4
```

3.8. Dado el array

```
4   7   -5   4    9
0   3   -2   6   -2
1   2    4   1    1
6   1    0   3   -4
```

Escribir un programa que encuentre la suma de todos los elementos que no pertenecen a la diagonal principal.

3.9. Escribir una función que intercambie la fila *i-ésima* por la *j-ésima* de un array de dos dimensiones, $m \times n$.

PROBLEMAS

Nota: Todos los problemas propuestos se deben realizar descomponiendo dicho problema en módulos (funciones en C).

3.1. 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

3.2. Escribir un programa que permita visualizar el triángulo de Pascal:

```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
  1   5  10  10   5   1
1   6  15  20  15   6   1
```

En el triángulo de Pascal cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando un array de una sola dimensión.

3.3. Escribir una función que invierta el contenido de *n* números enteros. El primero se vuelve el último; el segundo, el penúltimo, etc.

3.4. Escribir una función a la cual se le proporcione una fecha (día, mes, año), así como un número de días a añadir a esta fecha. La función calcula la nueva fecha y se visualiza.

3.5. Un número entero es primo si ningún otro número primo más pequeño que él es divisor suyo. A continuación, escribir un programa que rellene una tabla con los 80 primeros números primos y los visualice.

3.6. Escribir un programa que visualice un cuadrado mágico de orden impar *n* comprendido entre 3 y 11; el usuario debe elegir el valor de *n*. Un cuadrado mágico se compone de números enteros comprendidos entre 1 y *n*. La suma de los números que figuran en cada fila, columna y diagonal son iguales.

Ejemplo

```
8  1  6
3  5  7
4  9  2
```

Un método de generación consiste en situar el número 1 en el centro de la primera fila, el número siguiente en la casilla situada por encima y a la derecha, y así sucesivamente. El cuadrado es cíclico: la línea encima de la primera es, de hecho, la última y la columna a derecha de la última es la primera. En el caso de que el número generado caiga en una casilla ocupada, se elige la casilla situada encima del número que acaba de ser situado.

3.7. El juego del ahorcado se juega con dos personas (o una persona y una computadora). Un jugador selecciona una palabra y el otro jugador trata de adivinar la palabra adivinando letras individuales. Diseñar un programa para jugar al ahorcado. *Sugerencia:* almacenar una lista de palabras en un array y seleccionar palabras aleatoriamente.

3.8. Escribir un programa que lea las dimensiones de una matriz, lea y visualice la matriz y a continuación encuentre el mayor y menor elemento de la matriz y sus posiciones.

3.9. Si x representa la media de los números $x_1, x_2, \ldots x_n$, entonces la *varianza* es la media de los cuadrados de las desviaciones de los números de la media.

$$Varianza = \frac{1}{n} \sum_{i=1}^{n} (x_i - x)^2$$

Y la *desviación estándar* es la raíz cuadrada de la varianza. Escribir un programa que lea una lista de números reales, los cuente y a continuación calcule e imprima su media, varianza y desviación estándar. Utilizar funciones para calcular la media, varianza y desviación estándar.

3.10. Escribir un programa que lea una matriz A y forme la matriz transpuesta de A. El programa debe escribir ambas matrices.

3.11. Escribir 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.

3.12. Los resultados de las últimas elecciones a alcalde en el pueblo *x* han sido los siguientes:

Distrito	Candidato A	Candidato B	Candidato C	Candidato D
1	194	48	206	45
2	180	20	320	16
3	221	90	140	20
4	432	50	821	14
5	820	61	946	18

Escribir un programa que realice las siguientes tareas:

a) Imprimir la tabla anterior con cabeceras incluidas.
b) Calcular e imprimir el número total de votos recibidos por cada candidato y el porcentaje del total de votos emitidos. Asimismo, visualizar el candidato más votado.
c) Si algún candidato recibe más del 50 por 100 de los datos, el programa imprimirá un mensaje declarándole ganador.
d) Si ningún candidato recibe más del 50 por 100 de los datos, el programa debe imprimir el nombre de los dos candidatos más votados, que serán los que pasen a la segunda ronda de las elecciones.

3.13. Escribir un programa que lea una colección de cadenas de caracteres de longitud arbitraria. Por cada cadena leída, su programa hará lo siguiente:

a) Imprimir la longitud de la cadena.
b) Contar el número de ocurrencia de palabras de cuatro letras.
c) Sustituir cada palabra de cuatro letras por una cadena de cuatro asteriscos e imprimir la nueva cadena.

3.14. Una agencia de venta de vehículos automóviles distribuye quince modelos diferentes y tiene en su plantilla diez vendedores. Se desea un programa que escriba un informe mensual de las ventas por vendedor y modelo, así como el número de automóviles vendidos por cada vendedor y el número total de cada modelo vendido por todos los vendedores. Asimismo, para entregar el premio al mejor vendedor, necesita saber cuál es el vendedor que más coches ha vendido.

vendedor \ modelo	1	2	3	4.....15
1	4	8	1	4
2	12	4	25	14
3	15	3	4	7
⋮				
10				

3.15. 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).

3.16. Escribir un programa que lea una cadena de caracteres y la visualice en un cuadro.

3.17. Escribir un programa que lea una frase, sustituya todas las secuencias de dos o más blancos por un solo blanco y visualice la frase restante.

3.18. Escribir un programa que lea una frase y a continuación visualice cada palabra de la frase en columna, seguido del número de letras que compone cada palabra.

3.19. Escribir un programa que desplace una palabra leída del teclado desde la izquierda hasta la derecha de la pantalla.

3.20. Escribir un programa que lea una línea de caracteres, y visualice la línea de tal forma que las vocales sean sustituidas por el carácter que más veces se repite en la línea.

3.21. Escribir un programa que calcule la frecuencia de aparición de las vocales de un texto proporcionado por el usuario. Esta solución debe presentarse en forma de histograma.

3.22. Escribir un programa que lea una serie de cadenas, a continuación determine si la cadena es un identificador válido según la sintaxis de C. *Sugerencias:* utilizar las siguientes funciones: longitud (tamaño del identificador en el rango permitido); primero (determinar si el nombre comienza con un símbolo permitido); restantes (comprueba si los restantes son caracteres permitidos).

3.23. Escribir una función `sort` que ordene un conjunto de *n* cadenas en orden alfabético.

3.24. Diseñar un programa que determine la media del número de horas trabajadas durante todos los días de la semana para cada uno de los empleados de la Universidad.

3.25. Escribir una función que ordene los *n* primeros elementos de un array de cadenas basado en las longitudes de las cadenas. Por ejemplo, 'bibi' vendrá antes que 'Ana'.

3.26. Se introduce una frase por teclado. Se desea imprimir cada palabra de la frase en líneas diferentes y consecutivas.

3.27. Escribir un programa que determine si una frase o una palabra es un palíndromo. Un palíndromo es una cadena de caracteres que se leen de igual forma en ambos sentidos; por ejemplo: ana.

3.28. Escribir un programa que tenga como entrada una palabra y *n* líneas. Se quiere determinar el número de veces que se encuentra la palabra en las *n* líneas.

3.29. 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 calcular la posición de un *punto de silla* (si es que existe).

3.30. 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.

4 ESTRUCTURAS Y UNIONES

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Declarar nuevos tipos de datos.
- Agrupar campos de tipo diferente.
- Declarar registros en C.
- Designar objetos con los mismos nombres simbólicos de la vida cotidiana.
- Identificar con un único identificador dos o más variables diferentes.
- Realizar operaciones con los bits de una palabra.

CONTENIDO

4.1. Estructuras.
4.2. Acceso a estructuras.
4.3. Estructuras anidadas.
4.4. Arrays de estructuras.
4.5. Utilización de estructuras como parámetros.
4.6. Uniones.
4.7. Enumeraciones.
4.8. Campos de bit.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Estructura.
- `struct`.
- `union`.
- Estructuras anidadas.
- Selector de campos.
- `sizeof`.
- `typedef`.
- Enumeración.

INTRODUCCIÓN

Este capítulo describe estructuras, uniones, enumeraciones y tipos definidos por el usuario que permiten a un programador crear nuevos tipos de datos. La capacidad para crear nuevos tipos es una característica importante y potente de C y libera a un programador de restringirse al uso de los tipos ofrecidos por el lenguaje. Una *estructura* contiene múltiples variables, que pueden ser de tipos diferentes. La estructura es importante para la creación de programas potentes, tales como bases de datos u otras aplicaciones que requieran grandes cantidades de datos. Por otra parte, se analizará el concepto de *unión*, otro tipo de dato no tan importante como las estructuras y los arrays, pero sí necesarias en algunos casos.

Un tipo de dato enumerado es una colección de miembros con nombre que tienen valores enteros equivalentes. Un tipo definido por el usuario, `typedef`, es de hecho no un nuevo tipo de dato sino simplemente un sinónimo de un tipo existente.

4.1. ESTRUCTURAS

Los arrays son estructuras de datos que contienen un número determinado de elementos (su tamaño), todos ellos del mismo tipo de datos. Esta característica —estructura de datos homogénea— supone una gran limitación cuando se requieren grupos de elementos con tipos diferentes de datos cada uno. Por ejemplo, si se dispone de una lista de temperaturas, es muy útil un array; sin embargo, si se necesita una lista de información de clientes que contengan elementos tales como el nombre, la edad, la dirección, el número de la cuenta, etc., los arrays no son adecuados. La solución a este problema es utilizar un *tipo de dato registro*, en C denominado *estructura*.

Los componentes individuales de una estructura se llaman *miembros*. Cada miembro (elemento) de una estructura puede contener datos de un tipo diferente de otros miembros. Por ejemplo, se puede utilizar una estructura para almacenar diferentes tipos de información sobre una persona, tal como nombre, estado civil, edad y fecha de nacimiento. Cada uno de estos elementos se denomina *nombre del miembro*.

Para recordar

Una **estructura** es una colección de uno o más tipos de elementos denominados **miembros**, cada uno de los cuales puede ser de un tipo de dato diferente.

Una estructura puede contener cualquier número de miembros, cada uno de los cuales tiene un nombre único, denominado *nombre* del miembro. Supongamos que se desea almacenar los datos de una colección de discos compactos (CD) de música. Estos datos pueden ser:

- título
- artista
- número de canciones
- precio
- fecha de compra

La estructura CD contiene cinco miembros. Tras decidir los miembros, se debe decidir cuáles son los tipos de datos para utilizar por los miembros. Esta información se representa en la tabla siguiente:

Nombre miembro	Tipo de dato
Título	Array de caracteres de tamaño 30.
Artista	Array de caracteres de tamaño 25.
Número de canciones	Entero.
Precio	Coma flotante.
Fecha de compra	Array de caracteres de tamaño 8.

La Figura 4.1 contiene la estructura CD, mostrando gráficamente los tipos de datos dentro de la misma. Obsérvese que cada miembro es un tipo de dato diferente.

Título	Ay, ay, ay, cómo se aleja el sol.
Artista	No me pises las sandalias.
Número de canciones	10
Precio	20.5
Fecha de compra	8-10-2003

Figura 4.1. Representación gráfica de una estructura CD.

4.1.1. Declaración de una estructura

Una estructura es un tipo de dato definido por el usuario, que se debe declarar antes de que se pueda utilizar. El formato de la declaración es:

```
struct <nombre de la estructura>
{
  <tipo de dato miembro1> <nombre miembro1>
  <tipo de dato miembro2> <nombre miembro2>
  ...
  <tipo de dato miembron> <nombre miembron>
}
```

La declaración de la estructura `CD` es:

```
struct coleccion_CD
{
  char titulo[30];
  char artista[25];
  int num_canciones;
  float precio;
  char fecha_compra[8];
};
```

Ejemplo

```
struct complejo
{
  float parteReal, parteImaginaria;
};
```

En este otro ejemplo se declara el tipo estructura `venta`:

```
struct venta
{
  char vededor[30];
  unsigned int codigo;
  int unids_articulos;
  float precio_unit;
};
```

4.1.2. Definición de variables de estructuras

Al igual que a los tipos de datos enumerados, a una estructura se accede utilizando una variable o variables que se deben definir después de la declaración de la estructura. Del mismo modo que sucede en otras situaciones, en C existen dos conceptos similares a considerar: *declaración* y *definición*. La diferencia técnica es la siguiente, una *declaración* especifica simplemente el nombre y el formato de la estructura de datos, pero no reserva almacenamiento en memoria; la *declaración* especifica un nuevo tipo de dato: `struct <nombreEstructura>`. Por consiguiente, cada definición de variable para una estructura dada crea un área en memoria en donde los datos se almacenan de acuerdo al formato estructurado declarado.

Las variables de estructuras se pueden definir de dos formas:

1. Listándolas inmediatamente después de la llave de cierre de la declaración de la estructura.
2. Listando el tipo de la estructura creado seguida por las variables correspondientes en cualquier lugar del programa antes de utilizarlas.

La definición y declaración de la estructura `coleccionesCD` se puede hacer por cualquiera de los dos métodos siguientes:

1.
```
struct coleccionesCD
{
  char titulo[30];
  char artista[25];
  int num_canciones;
  float precio;
  char fecha_compra[8];
} cd1, cd2, cd3;
```

2.
```
struct coleccionesCD cd1, cd2, cd3;
```

Otros ejemplos de definición/declaración

Considérese un programa que gestione libros y procese los siguientes datos: título del libro, nombre del autor, editorial y año de publicación. Una estructura `infoLibro` podría ser:

```
struct infoLibro
{
  char titulo[60];
  char autor[30];
  char editorial[30];
  int anyo;
};
```

La definición de la estructura se puede hacer así:

1.
```
struct infoLibro
{
  char titulo[60];
  char autor[30];
  char editorial[30];
  int anyo;
} libro1, libro2, libro3;
```

2.
```
struct infoLibro libro1, libro2, libro3;
```

Ahora se nos plantea una aplicación de control de los participantes en una carrera popular, cada participante se representa por los datos: nombre, edad, sexo, categoría, club y tiempo. El registro se representa con la estructura `corredor`:

```
struct corredor
{
  char nombre[40];
  int edad;
```

```
  char sexo;
  char categoria[20];
  char club[26];
  float tiempo;
};
```

La definición de variables estructura se puede hacer así:

```
struct corredor v1, s1, c1;
```

4.1.3. Uso de estructuras en asignaciones

Como una estructura es un tipo de dato similar a un `int` o un `char`, se puede asignar una estructura a otra. Por ejemplo, se puede hacer que `libro3`, `libro4` y `libro5` tengan los mismos valores en sus miembros que `libro1`. Por consiguiente, sería necesario realizar las siguientes sentencias:

```
libro3 = libro1;
libro4 = libro1;
libro5 = libro1;
```

De modo alternativo se puede escribir:

```
libro5 = libro4 = libro3 = libro1;
```

4.1.4. Inicialización de una declaración de estructuras

Se puede inicializar una estructura de dos formas: dentro de la sección de código de su programa, o bien como parte de la definición. Cuando se inicializa una estructura como parte de la definición, se especifican los valores iniciales, entre llaves, después de la definición de variables estructura. El formato general en este caso:

```
struct <tipo> <nombre variable estructura> =
{ valor miembro1,
  valor miembro2,
    ...
  valor miembrom
};

struct infoLibro
{
  char titulo[60];
  char auto[30];
  char editorial[30];
  int anyo;
}libro1 = {"Maravilla del saber", "Lucas Garcia",
                                  ,"McGraw-Hill", 2003};
```

Otro ejemplo podría ser inicilizar la estructura coleccionCD:

```
struct coleccionCD
{
  char titulo[30];
  char artista[25];
  int num_canciones;
  float precio;
  char fecha_compra[8];
} cd1 = {
         "El humo nubla tus ojos",
         "Col Porter",
         15,
         12.45,
         "02/6/03"
        };
```

Un último ejemplo con la estructura corredor:

```
struct corredor v1 = {
  "Salvador Rapido",
   29,
   'V',
   "Senior",
   "Independiente",
    0.0
};
```

4.1.5. El tamaño de una estructura

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 siguiente programa ilustra el uso del operador sizeof para determinar el tamaño de una estructura:

```
#include <stdio.h>

/* declarar una estructura Persona */
struct persona
{
  char nombre[30];
  int edad;
  float altura;
  float peso;
};

void main()
{
  struct persona mar;
  printf("sizeof(persona): %d \n",sizeof(mar));
}
```

Al ejecutar el programa se produce la salida:

```
sizeof(persona) : 40
```

El resultado se obtiene determinando el número de bytes que ocupa la estructura:

Persona	Miembros dato	Tamañ (bytes)
nombre[30]	char(1)	30
edad	int(2)	2
altura	float(4)	4
peso	float(4)	4
Total		40

4.2. ACCESO A ESTRUCTURAS

Cuando se accede a una estructura, o bien se almacena información en la estructura o se recupera la información de la estructura, se puede acceder a los miembros de una estructura de una de estas dos formas:

1. Utilizando el operador punto (.).
2. Utilizando el operador puntero ->.

4.2.1. Almacenamiento de información en estructuras

Se puede almacenar información en una estructura mediante inicialización, asignación directa o lectura del teclado. El proceso de inicialización ya se ha examinado, veamos ahora la asignación directa y la lectura del teclado.

Acceso a una estructura de datos mediante el operador punto

La asignación de datos a los miembros de una variable estructura se hace mediante el operador punto. La sintaxis en C es:

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

Algunos ejemplos:

```
strcpy(cd1.titulo,"Granada");
cd1.precio = 3450.75;
cd1.num_canciones = 7;
```

El operador punto proporciona el camino directo al miembro correspondiente. Los datos que se almacenan en un miembro dado deben ser del mismo tipo que el tipo declarado para ese miembro.

En el siguiente ejemplo se lee del teclado los datos de una variable estructura corredor:

```
struct corredor cr;

printf("Nombre: ");
gets(cr.nombre);
printf("edad: ");
scanf("%d",&cr.edad);
printf("Sexo: ");
scanf("%c",&cr.sexo);
printf("Club: ");
gets(cr.club);
if (cr.edad <= 18)
  strcpy(cr.categoria,"Juvenil");
else if (cr.edad <= 40)
  strcpy(cr.categoria,"Senior");
else
  strcpy(cr.categoria,"Veterano");
```

Acceso a una estructura de datos mediante el operador puntero

El operador puntero, ->, sirve para acceder a los datos de la estructura a partir de un puntero. Para utilizar este operador se debe definir primero una variable puntero para apuntar a la estructura. A continuación, utilice simplemente el operador puntero para apuntar a un miembro dado. La asignación de datos a estructuras utilizando el operador puntero tiene el formato:

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

Así, por ejemplo, una estructura `estudiante`

```
struct estudiante
{
  char nombre[41];
  int numEstudiante;
  int anyo_de_matricula;
  float nota;
};
```

se puede definir `ptrEst` como un puntero a la estructura

```
struct estudiante *ptrEst;
struct estudiante mejor;
```

A los miembros de la estructura `estudiante` se les puede asignar datos como sigue (siempre y cuando la estructura ya tenga creado su espacio de almacenamiento, por ejemplo, con `malloc();` o bien, tenga la dirección de una variable estructura).

```
ptrEst = &mejor;   /* ptrEst tiene la dirección(apunta a) mejor */
strcpy(ptrEst -> nombre,"Pepe Alondra");
ptrEst -> numEstudiante = 3425;
ptrEst -> nota = 8.5;
```

Precaución

Antes de acceder a los miembros de una estructura con una variable puntero y el operador -> es necesario crear espacio de almacenamiento en memoria, por ejemplo, con la función `malloc()`; o bien, que tenga la dirección de una estructura ya creada, por ejemplo, `ptrEst = &mejor`.

4.2.2. Lectura de información de una estructura

Si ahora se desea introducir la información en la estructura basta con acceder a los miembros de la estructura con el operador punto o flecha (puntero). Se puede introducir la información desde el teclado o desde un archivo, o asignar valores calculados.

Por ejemplo, si z es una variable de tipo estructura complejo, se lee parte real, parte imaginaria y se calcula el módulo:

```
struct complejo
{
  float pr;
  float pi;
  float modulo;
 };
struct complejo z;

printf("\nParte real: ");
scanf("%f",&z.pr);
printf("\nParte imaginaria: ");
scanf("%f",&z.pi);
/* calculo del módulo */
z.modulo = sqrt(z.pr*z.pr + z.pi*z.pi);
```

4.2.3. Recuperación de información de una estructura

Se recupera información de una estructura utilizando el operador de asignación o una sentencia de salida (`printf()`, `puts()` ...). Igual que antes, se puede emplear el operador punto o el operador flecha (puntero). El formato general toma uno de estos formatos:

1. *<nombre variable>* =
   <nombre variable estructura>.<nombre miembro>;

   o bien

   <nombre variable> =
   <puntero de estructura> -> <nombre miembro>;

2. Para salida:
   printf(" ",*<nombre variable estructura>.<nombre miembro>*);

   o bien

   printf(" ",*<puntero de estructura> -> <nombre miembro>*);

Algunos ejemplos del uso de la estructura `complejo`:

```
float x,y;
struct complejo z;
struct complejo *pz;
pz = &z;
...
x = z.pr;
y = z.pi;

printf("\nNúmero complejo (%.1f,%.1f), módulo: %.2f",
        pz->pr,pz->pi,pz->modulo);
```

4.3. ESTRUCTURAS ANIDADAS

Una estructura puede contener otras estructuras llamadas *estructuras anidadas*. Las estructuras anidadas ahorran tiempo en la escritura de programas que utilizan estructuras similares. Se han de definir los miembros comunes sólo una vez en su propia estructura y a continuación utilizar esa estructura como un miembro de otra estructura.

Consideremos las siguientes dos definiciones de estructuras:

```
struct empleado
{
  char nombre_emp[30];
  char direccion[25];
  char ciudad[20];
  char provincia[20];
  long int cod_postal;
  double salario;
};

struct cliente
{
  char nombre_cliente[30];
  char direccion[25];
  char ciudad[20];
  char provincia[20];
  long int cod_postal;
  double saldo;
};
```

Estas estructuras contienen muchos datos diferentes, aunque hay datos que están solapados. Así, se podría disponer de una estructura, `info_dir`, que contuviera los miembros comunes.

```
struct info_dir
{
  char direccion[25];
  char ciudad[20];
  char provincia[20];
  long int cod_postal;
};
```

Esta estructura se puede utilizar como un miembro de las otras estructuras, es decir, *anidarse*.

```
struct empleado
{
  char nombre_emp[30];
  struct info_dir direccion_emp;
  double salario;
};

struct cliente
{
  char nombre_cliente[30];
  struct info_dir direccion_clien;
  double saldo;
};
```

Gráficamente se podrían mostrar estructuras anidadas en la Figura 4.2.

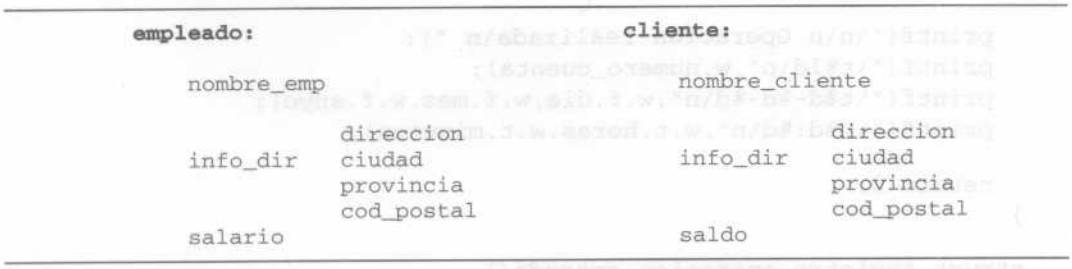


Figura 4.2. Estructuras anidadas.

4.3.1. Ejemplo de estructuras anidadas

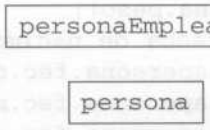
Se desea diseñar una estructura que contenga información de operaciones financieras. Esta estructura debe constar de un número de cuenta, una cantidad de dinero, el tipo de operación (depósito = 0, retirada de fondos = 1, puesta al día = 2 o estado de la cuenta = 3) y la fecha y hora en que la operación se ha realizado. A fin de realizar el acceso correcto a los campos día, mes y año, así como el tiempo (la hora y minutos) en que se efectuó la operación, se define una estructura `fecha` y una estructura `tiempo`. La estructura `registro_operacion` tiene como miembros una variable (un campo) de tipo `fecha`, otra variable del tipo `tiempo` y otras variables para representar los otros campos. La representación del tipo de operación se hace con una variable entera, aunque el tipo apropiado es un tipo enumerado (descrito en siguientes apartados). A continuación se declara estos tipos, se escribe una función que lee una operación financiera y devuelve la operación leída. La fecha y hora es capturada del sistema.

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

struct fecha
{
  unsigned int mes, dia, anyo;
};
```

```
struct tiempo
{
  unsigned int horas, minutos;
};
struct registro_operacion
{
  long numero_cuenta;
  float cantidad;
  int tipo_operacion;
  struct fecha f;
  struct tiempo t;
};

struct registro_operacion entrada();

int main()
{
  struct registro_operacion w;

  w = entrada();

  printf("\n\n Operación realizada\n ");
  printf("\t%ld\n",w.numero_cuenta);
  printf("\t%d-%d-%d\n",w.f.dia,w.f.mes,w.f.anyo);
  printf("\t%d:%d\n",w.t.horas,w.t.minutos);

  return 0;
}

struct registro_operacion entrada()
{
  struct time t;
  struct date d;
  struct registro_operacion una;
  printf("\nNúmero de cuenta: ");
  scanf("%ld",&una.numero_cuenta);
  puts("\n\tTipo de operación");
  puts("Deposito(0)");
  puts("Retirada de fondos(1)");
  puts("Puesta al dia(2)");
  puts("Estado de la cuenta(3)");
  scanf("%d",&una.tipo_operacion);
      /* Fecha y tiempo del sistema */
  gettime(&t);
  una.t.horas = t.ti_hour;
  una.t.minutos = t.ti_min;

  getdate(&d);
  una.f.anyo = d.da_year;
  una.f.mes = d.da_mon;
  una.f.dia = d.da_day;
  return una;
}
```

Ejercicio 4.1

Se desea registrar una estructura `PersonaEmpleado` que contenga como miembros los datos de una persona empleado que a su vez tenga los datos de la fecha de nacimiento. En un programa se muestra el uso de la estructura, se define una función para dar entrada a los datos de la estructura y otra función para dar salida a los datos de una estructura persona. A la función de entrada se transmite por dirección (&p) la variable estructura, por lo que el argumento correspondiente tiene que ser un puntero (*p) y el acceso a los campos se hace con el selector ->.




```
#include <stdio.h>

struct fecha
{
  unsigned int dia, mes, anyo;
};

struct persona {
  char nombre[20];
  unsigned int edad;
  int altura;
  int peso;
  struct fecha fec;
};

struct personaEmpleado
{
  struct persona unapersona;
  unsigned int salario;
  unsigned int horas_por_semana;
};

/* prototipos de funciones */

void entrada(struct personaEmpleado *p);
void muestra(struct personaEmpleado up);

void main()
{
  struct personaEmpleado p;

    /* En la llamadase transmite la dirección de la estructura */
  entrada(&p);

    /* salida de los datos almacenados */
  muestra(p);
}
```

```
void entrada(struct personaEmpleado *p)
{
  printf("\nIntroduzca su nombre: ");
  gets(p->unapersona.nombre);
  printf("Introduzca su edad: ");
  scanf("%d",&p->unapersona.edad);
  printf("Introduzca su altura: ");
  scanf("%d",&p->unapersona.altura);
  printf("Introduzca su peso: ");
  scanf("%d",&p->unapersona.peso);
  printf("Introduzca su fecha de nacimiento: ");
  scanf("%d %d %d",&p->unapersona.fec.dia,
                   &p->unapersona.fec.mes,
                   &p->unapersona.fec.anyo);
  printf("Introduzca su salario:");
  scanf("%d",&p->salario);
  printf("introduzca numero de horas:");
  scanf("%d",&p->horas_por_semana);
}

void muestra(struct personaEmpleado up)
{
  puts("\n\n\tDatos de un empleado");
  puts("\n\n\t----- -- -- --------");
  printf("Nombre: %s \n",up.unapersona.nombre);
  printf("Edad: %d ¨n",up.unapersona.edad);
  printf("fecha de nacimiento: %d-%d-%d\n",
                      up.unapersona.fec.dia,
                      up.unapersona.fec.mes,
                      up.unapersona.fec.anyo);
  printf("Altura: %d \n",up.unapersona.altura);
  printf("Peso: %d \n",up.unapersona.peso);
  printf("Numero de horas: %d \n",up.horas_por_semana);
}
```

Nota de programación

El acceso a miembros dato de estructuras anidadas requiere el uso de múltiples operadores punto. Ejemplo: acceso al día del mes de la fecha de nacimiento de un empleado.

```
up.unapersona.fec.dia
```

Las estructuras se pueden anidar a cualquier grado. También es posible inicializar estructuras anidadas en la definición. El siguiente ejemplo inicializa una variable `Luis` de tipo `struct persona`.

```
struct persona Luis { "Luis", 25, 1940, {40, 12, 1, 70}};
```

4.4. ARRAYS DE ESTRUCTURAS

Se puede crear un *array de estructuras* tal como se crea un array de otros tipos. Los arrays de estructuras son idóneos para almacenar en memoria un archivo completo de empleados, un archivo de inventario, o cualquier otro conjunto de datos que se adapte a un formato de estructura. Mientras que los arrays proporcionan un medio práctico de almacenar diversos valores del mismo tipo, los arrays de estructuras permiten almacenar juntos diversos valores de diferentes tipos, agrupados como estructuras.

Muchos programadores de C utilizan arrays de estructuras como un método para almacenar los datos de un archivo de disco. Se pueden introducir y calcular sus datos en arrays de estructuras, y una vez realizado el proceso requerido con las estructuras almacenar esas estructuras en memoria externa, en disco.

La declaración de un array de estructuras `info_libro` se puede hacer de un modo similar a cualquier array; es decir,

```
struct info_libro libros[100];
```

asigna un array de 100 elementos denominado `libros`. Para acceder a los miembros de cada uno de los elementos estructura se utiliza la notación de índices de los arrays. Para inicializar el primer elemento de `libros`, por ejemplo, su código debe hacer referencia a los miembros de `libros[0]` de la forma siguiente:

```
strcpy(libros[0].titulo, "C a su alcance");
strcpy(libros[0].autor, "Luis Joyanes");
strcpy(libros[0].editorial, "McGraw-Hill");
libros[0].anyo = 1999;
```

También puede inicializarse un array de estructuras en el punto de la declaración encerrando la lista de inicializadores entre llaves, `{ }`. Por ejemplo,

```
struct info_libro libros[3] = { "C a su alcance", "Luis Joyanes",
     "McGraw-Hill", 1999, "Estructura de datos", "Luis Joyanes",
     "McGraw-Hill", 1999, "Problemas en Pascal", "Angel Hermoso",
     "McGraw-Hill", 1997};
```

En el siguiente ejemplo se declara una estructura que representa a un número racional, un array de números racionales es inicializado con valores al azar.

```
struct racional
{
  int n,
  int d;
};
struct racional rs[4] = { 1,2, 2,3, -4,7, 0,1};
```

4.4.1. Arrays como miembros

Los miembros de las estructuras puede ser así mismo arrays. En este caso, será preciso extremar las precauciones cuando se accede a los elementos individuales del array.

Considérese la definición de estructura `nomina`. Esta sentencia declara un array de cien estructuras; cada estructura contiene información de datos de empleados de una compañía.

```
struct nomina
{
   char nombre[30];
   int dependientes;
   char departamento[10];
   float horas_dias[7];        /* array de tipo float */
   float salario;
} empleado[100];               /* Un array de 100 empleados */
```

Ejemplo 4.1

Una librería desea catalogar su inventario de libros. El siguiente programa crea un array de 100 estructuras, donde cada estructura contiene diversos tipos de variables, incluyendo arrays.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

struct inventario
{
  char titulo[25];
  char fecha_pub[20];
  char autor[30];
  int num;
  int pedido;
  float precio_venta;
};

int main()
{
  struct inventario libro[100];
  int total = 0;
  char resp, b[21];

  do {
    printf("Total libros %d \n",(total+1));
    printf("¿Cuál es el título?: ");
    gets(libro[total].titulo);

    printf("¿Cuál es la fecha de publicación?: ");
    gets(libro[total].fecha_pub);

    printf("¿Quién es el autor?");
    gets(libro[total].autor);

    printf("¿Cuántos libros existen?: ");
    scanf("%d",&libro[total].num);

    printf("¿Cuántos ejemplares existen pedidos?: ");
    scanf("%d",&libro[total].pedido);
```

```
        printf("¿Cuál es el precio de venta?: ");
        gets(b);
        libro[total].precio_venta = atof(b); /* conversión a real */
        fflush(stdin);

        printf("\n ¿Hay más libros? (S/N)");
        scanf("%c",&resp);
        fflush(stdin);
        resp = toupper(resp);
        /* convierte a mayúsculas */
        if (resp == 'S')
        {
          total++;
          continue;
        }
      } while (resp == 'S');
      return 0;
    }
```

4.5. UTILIZACIÓN DE ESTRUCTURAS COMO PARÁMETROS

C permite pasar estructuras a funciones, bien *por valor* o bien *por referencia*, utilizando el operador &. Si la estructura es grande, el tiempo necesario para copiar un parámetro `struct` a la pila puede ser prohibitivo. En tales casos, se debe considerar el método de pasar la dirección de la estructura.

El listado siguiente muestra la llamada a una función para entrada de datos que pasa la dirección de una estructura. La misma variable estructura se pasa por valor a otra función para salida de los campos.

```
/* Define el tipo estructura info_persona */

struct info_persona {
  char nombre[20];
  char calle[30];
  char ciudad[25];
  char provincia[25];
  char codigopostal[6];
};

    /* prototipos de funciones */

void entradaPersona(struct info_persona* pp);
void verInfoPersona(struct info_persona p);

    /* define variable de tipo estructura */
struct info_persona reg_dat;

    /* Pasa por referencia la variable */
entradaPersona(&reg_dat);

    /* Pasa por valor */
verInfoPersona(reg_dat);
```

A tener en cuenta

Si desea pasar la estructura por referencia, necesita situar un operador de referencia `&` antes de `reg_dat` en la llamada a la función `entradaPersona()`. El parámetro correspondiente debe de ser tipo puntero `struct info_persona* pp`. El acceso a miembros dato de la estructura a partir de un puntero requiere el uso del selector `->`.

4.6. 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 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. El tamaño ocupado por una unión se determina analizando el tamaño de cada variable de la unión; el tamaño mayor de variable será el tamaño de la unión.

La sintaxis de una unión es la siguiente:

```
union nombre {
   tipo1 miembro1;
   tipo2 miembro2;
   ...
};
```

Un ejemplo:

```
union PruebaUnion
{
  float Item1;
  int Item2;
};
```

La cantidad de memoria reservada para una unión es igual a la longitud de la variable más grande. En el tipo `union`, cada uno de los miembros dato comparten memoria con los otros miembros de la unión. La cantidad total de memoria utilizada por la unión `comparte` es de 8 bytes, ya que el elemento `double` es el miembro dato mayor de la unión.

```
union comparte
{
  char letra;
  int elemento;
  float precio;
  double z;
};
```

Una razón para utilizar una unión es ahorrar memoria. En muchos programas se deben tener varias variables, pero no necesitan utilizarse todas al mismo tiempo. Considérese la situación en

que se necesitan tener diversas cadenas de caracteres de entrada. Se pueden crear varios arrays de cadenas de caracteres, tales como las siguientes:

```
char linea_ordenes[80];
char mensaje_error[80];
char ayuda[80];
```

Estas tres variables ocupan 240 bytes de memoria. Sin embargo, si su programa no necesita utilizar las tres variables simultáneamente, ¿por qué no permitirle compartir la memoria utilizando una unión? Cuando se combinan en el tipo `union frases`, estas variables ocupan un total de sólo 80 bytes.

```
union frases {
  char linea_ordenes[80];
  char mensaje_error[80];
  char ayuda[80];
} cadenas, *pc;
```

Para referirse a los miembros de una unión, se utiliza el operador punto `(.)`, o bién el operador `->` si se hace desde un puntero a unión. Así:

```
cadenas.ayuda;
cadenas.mensaje_error;
pc -> mensaje_error;
```

En este ejemplo se considera el supuesto de que una pequeña biblioteca tiene dos tipos de objetos, libros y revistas. Se puede definir una estructura `libro`, una estructura `revista` y una unión con ambas denominada `elemento`. En un momento dado la unión va a representar a un libro, en otro a una revista.

```
struct libro
{
  char titulo[41];
  char autor[30];
  char editorial[51];
  int numPagina;
};
struct revista
{
  char nombre[41];
  char tema[30];
  struct fecha d;
  float precio;
};
union elemento
{
  struct libro lb;
  struct revista r;
};
```

4.7. ENUMERACIONES

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

Formato

1.
```
enum
{
  enumerador1, enumerador2, ...enumeradorn
};
```

2.
```
enum nombre
{
  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:

3.
```
enum nombre
{
  enumerador1 = expresión_constante1,
  enumerador2 = expresión_constante2,
  ...
  enumeradorn = expresión_constantenn
};
```

Ejemplo 4.2

Usos típicos de `enum`.

```
enum Interruptor
{
  ENCENDIDO,
  APAGADO
};
```

define las constantes `ENCENDIDO` y `APAGADO` de valores iguales a 0 y 1.

```
enum Boolean
{
  FALSE,
  TRUE
};
```

define dos constantes para representar las constantes lógicas (*true* y *false*) con valores iguales a 0 y 1.

```
enum colores
{
  ROJO, VERDE, AZUL, NEGRO, AMARILLO, BLANCO
};
```

define seis constantes de valores iguales a 0, 1 ... 5 respectivamente. Los miembros datos de un enum se llaman enumeradores y la constante entera por defecto del primer enumerador de la lista de los miembros datos es igual a 0. Obsérvese que, al contrario que **struct** y **union**, los miembros de un tipo **enum** se separan por el operador coma. El ejemplo anterior es equivalente a la definición de las seis constantes tal como:

```
const int ROJO = 0;
const int VERDE = 1;
const int AZUL = 2;
...
const int BLANCO = 5;
```

En la siguiente declaración de tipo enumerado se proporciona un nombre al tipo. Además, la primera constante tiene valor 1 y las demás los siguientes valores enteros.

```
enum dias_semana
{
  LUNES = 1, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
};
```

Una variable de tipo `enum dias_semana` puede tomar los valores especificados en la declaración del tipo. El siguiente bucle está controlado por una variable del tipo enumerado.

```
enum dias_semana dia;

for (dia = LUNES; dia <= DOMINGO; dia++)
{
  ...
  printf("%d ",dia);
}
```

La ejecución del bucle escribiría en pantalla: `1 2 3 4 5 6 7`.

A los enumeradores se pueden asignar valores constantes o expresiones constantes durante la declaración:

```
enum hexaedro
{
  VERTICES = 8,
  LADOS = 12,
  CARAS = 6
};
```

Ejercicio 4.2

El siguiente programa muestra el uso de la enumeración `Boolean`. El programa lee un texto y cuenta las vocales leídas. La función `vocal()` devuelve `TRUE` (verdad) si el carácter de entrada es vocal.

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

```
enum Boolean
{
  FALSE, TRUE
};

enum Boolean vocal(char c);

void main()
{
  char car;
  int numvocal = 0;

  puts("\nIntroduce un texto. Para terminar: INTRO.");
  while ((car = getchar())!='\n')
  {
    if (vocal(tolower(car)))
      numvocal++;
  }
  printf("\n Total de vocales leidas: %d\n",numvocal);
}

enum Boolean vocal(char c)
{
  switch (c)
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
      return TRUE;
    default:
      return FALSE;
  }
}
```

4.7.1. `sizeof` de tipos de datos estructurados

El tamaño en bytes de una estructura, de una unión o de un tipo enumerado se puede determinar con el operador `sizeof`.

El siguiente programa extrae el tamaño de una estructura (`struct`) de una unión (`union`) con miembros dato idénticos y de un tipo enumerado.

```
union tipo_union
{
  char c;
  int i;
  float f;
  double d;
};
```

```
struct tipo_estructura
{
  char c;
  int i;
  float f;
  double d;
};
enum monedas
{
  PESETA,
  DURO,
  CINCODUROS,
  CIEN
};
...

printf("\nsizeof(tipo_estructura): %d\n",
                    sizeof(struct tipo_estructura));
printf("\nsizeof(tipo_union): %d\n",
                    sizeof(union tipo_union));
printf("\nsizeof(monedas): %d\n",
                    sizeof(enum monedas));
```

La salida que se genera con estos datos:

```
sizeof(tipo_estructura):15
sizeof(tipo_union): 8
sizeof(monedas): 2
```

4.7.2. typedef

Un operador **typedef** permite a un programador crear un sinónimo de un tipo de dato definido por el usuario o de un tipo ya existente. La sintaxis:

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 los tipos de datos.

Ejemplo

Uso de `typedef` *para declarar un nuevo nombre,* `Longitud`*, de tipo de dato double.*

```
typedef double Longitud;
...
Longitud distancia (Punto p1, Punto p2)
{
  ...
  Longitud longitud = sqrt(r-cua);
  return longitud;
}
```

Otros ejemplos:

```
typedef char* String;

typedef const char* string;
```

Normalmente los programadores utilizan sinónimos de tipos de datos estructura para simplificar los tipos y hacerlos más *amigables*. En primer lugar se declara el tipo estructura, lo mismo con el tipo unión, y a continuación se asocia el tipo estructrura a un nombre con `typedef`.

Ejemplo

Declaración del tipo de dato complejo y asociación a `complex`.

```
struct complejo
{
  float x,y;
};

typedef struct complejo complex;

/* definición de un array de complejos */
complex v[12];
```

También, en una sola sentencia se puede declarar el tipo estructura y la asociación a un nombre con `typedef`.

Ejemplo

Declaración del tipo de dato que representa a un pasajero y asociación a `Pasajero`.

```
typedef struct
{
  char apellido[31];
  char nombre[21];
  int edad;
  char pasaporte[26];
  char sexo;
} Pasajero;

/* prototipo de función para dar entrada de datos */
Pasajero entrada();
```

A tener en cuenta

La ventaja de `typedef` es que permite dar nombres a tipos de datos mas acordes con lo que representan en una determinada aplicación.

4.8. CAMPOS DE BIT

El lenguaje C permite realizar operaciones con los bits de una palabra mediante un conjunto de operadores de manejo de bits: >>, <<, ... Con los campos de bit, C permite acceder a un número de bits de una palabra entera. Un **campo de bits** es un conjunto de bits adyacentes dentro de una palabra entera.

La sintaxis para declarar campos de bits se basa en la declaración de estructuras. El formato general es:

```
struct identificador_campo {
   tipo nombre1: longitud1;
   tipo nombre2: longitud2;
   tipo nombre3: longitud3;
      .
      .
      .
   tipo   nombren:   longitudn;
};
```

`tipo` ha de ser entero, `int`; generalmente `unsigned int`
`longitud` es el número de bits consecutivos que se toman.

Ejemplo 4.3

Declarar un campo de bits para representar en formato comprimido el día, mes, año (los dos últimos dígitos) y si el año es bisiesto.

```
struct fecha {
   unsigned dia: 5;
   unsigned mes: 4;
   unsigned anyo: 7;
   unsigned bisiesto: 1;
};
```

Ejemplo 4.4

El siguiente ejemplo muestra cómo pueden utilizarse campos de bits para representar si están o no conectados diversos componentes eléctricos. Cada componente se representa con un *flag* (un bit): si está puesto a cero es que no está conectado; si está puesto a uno está conectado.

```
struct componentes {
   unsigned diodo: 1;
   unsigned resistencia: 1;
   unsigned amperimetro: 1;
   unsigned transistor: 1;
   unsigned condensador: 1;
   unsigned inductancia: 1;
};
```

Los campos individuales se referencian como cualquier otro miembro de una estructura: selector punto (.). Por ejemplo,

```
struct componentes ct;
ct.diodo = 1;
if (ct.amperimetro)
{
}
```

Al declarar campos de bits, la suma de los bits declarados puede exceder el tamaño de un entero; en ese caso se emplea la siguiente posición de almacenamiento entero. No está permitido que un campo de bits solape los límites entre dos tipos `int`.

Al declarar una estructura puede haber miembros que sean variables y otros, campos de bits. La siguiente estructura tiene esta característica:

```
struct reg_estudiante{
  char nombre[33];
  char apell1[33];
  char apell2[33];
  unsigned masculino: 1;
  unsigned femenino: 1;
  unsigned curso: 3;
};
```

Los campos de bits se utilizan para rutinas de encriptación de datos y fundamentalmente para ciertos interfaces de dispositivos externos.

Los campos de bits tienen ciertas restricciones: no se puede tomar la dirección de una variable campo de bits; no puede haber arrays de campos de bits; no se puede solapar fronteras de `int`. Depende del procesador el hecho de que los campos de bits se alineen de izquierda a derecha o de derecha a izquierda (conviene hacer una comprobación para cada procesador, utilizar para ello un `union` con una variable entera y campos de bits).

Ejercicio 4.3

Se tiene la función `peticion_acceso()` *capaz de direccionar una posición de memoria de 8 bits si recibe como argumento una variable llamada* `ochobits`. *Con esta variable se controla a través de cada bit las peticiones de acceso a cada uno de los ocho periféricos distintos con los cuales trabaja; eventos externos son los que se encargan de cargar la variable* `ochobits`.

Se quiere escribir una función que determine cuántos accesos se producen por cada periférico en un bucle de 1.000 llamadas a la función `peticion_acceso()`. *Se supone que cada llamada sólo activa un periférico.*

El tipo de la variable `ochobits` se va a declarar como una estructura de campos de bits, cada campo con longitud 1 y puede tener dos estados, 0 o 1, para indicar no acceso o sí acceso.

Un array de 8 elementos (uno por periférico) se utiliza para contar los accesos a cada periférico.

```
/* Tipo estructura de campos de bits */

stuct perifericos {
  unsigned perf1: 1;
```

```
    unsigned perf2: 1;
    unsigned perf3: 1;
    unsigned perf4: 1;
    unsigned perf5: 1;
    unsigned perf6: 1;
    unsigned perf7: 1;
    unsigned perf8: 1;
};

/* Prototipo de función peticion_acceso() */
void peticion_acceso(const struct perifericos ochobits);

/* Función que contabiliza los accesos a cada periférico. */

void accesos_perf(int accper[])
{
  int i;
  const struct perifericos ochobits;
  const int neventos=1000;

  for (i=0; i<8; )
    accper[i++]= 0;

  /* Bucle principal de 1000 llamadas */
  for (i=0; i<neventos; i++)
  {
    peticion_acceso(ochobits);
    if (ochobits.perf1)
      ++accper[0];
    elseif (ocho.bits.perf2)
      ++accper[1];
    elseif (ocho.bits.perf3)
      ++accper[2];
    elseif (ocho.bits.perf4)
      ++accper[3];
    elseif (ocho.bits.perf5)
      ++accper[4];
    elseif (ocho.bits.perf6)
      ++accper[5];
    elseif (ocho.bits.perf7)
      ++accper[6];
    elseif (ocho.bits.perf8)
      ++accper[7];
  }
}
```

Ejercicio 4.4

Haciendo uso de una estructura de campo de bits y de una union, se puede escribir un programa para visualizar la decodificación en bits de cualquier carácter leído por teclado.

Se declara un tipo estructura campo de bits, con tantos campos como bits tiene un byte, que a su vez es el almacenamiento de un carácter. La decodificación es inmediata declarando una `union` entre una variable carácter y una variable campo de bits del tipo indicado.

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

#define mayus(ch) ((ch>='a' &&ch<='z') ? (ch+'A'-'a') : ch)

struct byte {
  unsigned int a: 1;
  unsigned int b: 1;
  unsigned int c: 1;
  unsigned int d: 1;
  unsigned int e: 1;
  unsigned int f: 1;
  unsigned int g: 1;
  unsigned int h: 1;
};
union charbits
{
  char ch;
  struct byte bits;
}caracter;
void decodifica(struct byte b);

void main()
{
  puts("Teclea caracteres. Para salir carácter X");
  do {
    caracter.ch = getche();
    printf(" : ");
    decodifica(caracter.bits);
  }while mayusc(caracter.ch)!='X');
}

void decodifica(struct byte b)
{
  /* Los campos de bits se alinean de derecha a izquierda, por esa
     razón se ecriben los campos en orden inverso */

  printf("%2u%2u%2u%2u%2u%2u%2u%2u \n",
         b.h, b.g, b.f, b.e, b.d, b.c, b.b, b.a);
}
```

RESUMEN

Una estructura permite que los miembros dato de iguales o diferentes tipos se encapsulen en una única implementación, al contrario que los arrays que son agregados de un tipo de dato simple. Los miembros dato de una estructura son accesibles con el operador punto `(.)`, o con el operador flecha `(->)`.

- Una estructura es un tipo de dato que contiene elementos de tipos diferentes.

```
struct empleado {
  char nombre[30];
  long salario;
  char num_telefono[10];
};
```

- Para crear una variable estructura se escribe:

```
struct empleado pepe;
```

- Para determinar el tamaño en bytes de cualquier tipo de dato C utilizar el operador `sizeof`.
- El tipo de dato `union` es similar a **`struct`** de modo que es una colección de miembros dato de tipos diferentes o similares, pero al contrario que una definición **`struct`**, que asigna memoria suficiente para contener todos los miembros dato, una **`union`** puede contener sólo un miembro dato en cualquier momento dado y el tamaño de una unión es, por consiguiente, el tamaño de su miembro mayor.
- Utilizando un tipo `union` se puede hacer que diferentes variables coexistan en el mismo espacio de memoria. La unión permite reducir espacio de memoria en sus programas.
- Un `typedef` no es nuevo tipo de dato sino un sinónimo de un tipo existente.

```
typedef struct empleado regempleado;
```

EJERCICIOS

4.1. Encontrar los errores en la siguiente declaración de estructura y posterior definición de variable.

```
struct hormiga
{
  int patas;
  char especie[41];
  float tiempo;
}
hormiga colonia[100];
```

4.2. Declarar un tipo de datos para representar las estaciones del año.

4.3. Escribir un función que devuelva la estación del año que se ha leído del teclado. La función debe ser del tipo declarado en el Ejercicio 4.2.

4.4. Declarar un tipo de dato enumerado para representar los meses del año, el mes enero debe de estar asociado al dato entero 1, y así sucesivamente los demás meses.

4.5. Encontrar los errores del siguiente código:

```
#include <stdio.h>
void escribe(struct fecha f);
```

```
int main()
{
  struct fecha
  {
    int dia;
    int mes;
    int anyo;
    char mes[];
    }ff;
    ff = {1,1,2000,"ENERO"};
    escribe(ff);
    return 1;
}
```

4.6. Con `typedef`, ¿se declaran nuevos tipos de datos, o bien se permite cambiar el nombre de tipos de datos ya declarados?

4.7. Declarar un tipo de dato estructura para representar a un alumno; los campos que contiene son: nombre, curso, edad, dirección y notas de diez asignaturas. Declarar otro tipo estructura para representar a un profesor, cuyos campos sean: nombre, asignaturas que imparte y dirección. Por último, declarar una estructura que pueda representar a un profesor o a un alumno.

4.8. Definir tres variables correspondientes a los tres tipos de datos declarados en el ejercicio anterior y asignarles un nombre.

4.9. Escribir una función que devuelva el nombre de un profesor o un alumno cuyos datos se introducen por teclado.

4.10. Escribir la misma función que en el ejercicio anterior pero pasando la estructura como argumento a la función.

4.11. Escribir una función que tenga como entrada una estructura, profesor o alumno, y escribir sus campos por pantalla.

PROBLEMAS

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

4.2. Escribir 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. El programa debe generar a los diversos clientes.

4.3. Modificar el programa de facturación de clientes de tal modo que se puedan obtener los siguientes listados:

- Clientes en estado *moroso*.
- Clientes en estado *pagado* con factura mayor de una determinada cantidad.

4.4. 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.

4.5. Un número racional se caracteriza por el numerador y denominador. Escribe un programa para operar con números racionales. Las operaciones a definir son: suma, resta, multiplicación y división. Escribir también una función para simplificar cada número racional.

4.6. 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 cogidos.
- Nombre del mejor anotador de triples.
- Número de triples del mejor triplista.

Para los equipos de fútbol 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.

Escribir un programa que introduzca la información para todos los equipos integrantes en ambas ligas.

4.7. Modificar el programa 4.6 para obtener los siguientes informes o datos:

- Listado de los mejores triplistas de cada equipo.
- Máximo goleador de la liga de fútbol.
- Equipo ganador de la liga de fútbol. Suponiendo que se consiguen tres puntos por partido ganado y un punto por partido empatado.
- Equipo ganador de la liga de baloncesto.

4.8. Un punto en el plano se puede representar mediante una estructura con dos campos. Escribir un programa que realice las siguientes operaciones en el plano:

- Dados dos puntos calcular la distancia entre ellos.
- Dados dos puntos determinar la ecuación de la recta que pasa por ellos.
- Dados tres puntos, que representan los vértices de un triángulo calcular su área.

4.9. 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 empleado (cinco empleados) responsable de realizar las diferentes producciones. Los campos de la estructura empleado: número de identificación, nombre, producción asignada, nombre y horas trabajadas.

PARTE II

DISEÑO DE ALGORITMOS. RECURSIVIDAD, ORDENACIÓN Y BÚSQUEDA

5 RECURSIVIDAD

OBJETIVOS

Una vez que se haya leído y estudiado este capítulo usted podrá:

- Entender los pasos que siguen los lenguajes de programación al llamar a un subprograma.
- Hacer el seguimiento de una función que hace una llamada a sí mismo.
- Diseñar subprogramas (funciones) recursivos.
- Distinguir entre recursividad directa y recursividad indirecta.
- Conocer técnicas algorítmicas aplicando recursividad.
- Aplicar la técnica *divide y vencerás* para la resolución de problemas.
- Resolver problemas matemáticos con programas recursivos basados en la técnica *divide y vencerás*.
- Comparar la resolución de un mismo problema por iteración o por recursión.
- Conocer los algoritmos resueltos por la técnica de *backtracking*.
- Aplicar la técnica de *backtracking* al problema de la selección óptima.

CONTENIDO

5.1. Naturaleza de la recursividad.
5.2. Funciones recursivas.
5.3. Recursión *versus* iteración.
5.4. Recursión infinita.
5.5. Algoritmos *divide y vencerás*.
5.6. Ordenación por mezclas: *mergesort*.
5.7. *Backtracking:* algoritmos de vuelta atrás.
5.8. Resolución de problemas con algoritmos de *vuelta atrás*.
5.9. Selección óptima.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Recursividad.
- *Backtracking*.
- Iteración *versus* recursión.
- Búsqueda exhaustiva.
- *Divide y vencerás*.
- Mejor selección.
- Torres de Hanoi.
- Complejidad.

INTRODUCCIÓN

La recursividad (recursión) es aquella propiedad que posee una función por la cual dicha función puede llamarse a sí misma. 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. Por esta causa, la recursión es una herramienta poderosa e importante en la resolución de problemas y en programación.

5.1. NATURALEZA DE LA RECURSIVIDAD

Los programas examinados hasta ahora, generalmente estructurados, se componen de una serie de funciones que se llaman unas a otras de modo disciplinado. En algunos problemas es útil disponer de funciones que se llamen a sí mismas. Una función *recursiva* es aquella que se llama a sí misma bien directamente o bien a través de otra función. La recursividad es un tópico importante examinado frecuentemente en cursos y seminarios de resolución de algoritmos y de estructuras de datos.

En este libro se dará una importancia especial a las ideas conceptuales que soportan la recursividad. En matemáticas existen numerosas funciones que tienen carácter recursivo; de igual modo, numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo.

Una función que tiene sentencias entre las cuales se encuentra al menos una que llama a la propia función se dice que es *recursiva*. Supongamos se dispone de dos funciones: `funcion1` y `funcion2`, la organización de un programa no recursivo adoptaría una forma similar a ésta:

```
funcion1(...)
{
  ...
}
funcion2(...)
{
  ...
  funcion1();     /* llamada a la funcion1 */
  ...
}
```

Con una organización recursiva, se tendría esta situación:

```
funcion1(...)
{
  ...
  funcion1();
  ...
}
```

Ejemplo 5.1

El factorial de un entero negativo *n*, escrito *n*! (y pronunciado *n* factorial), es el producto:

```
n! = n · (n - 1) · (n - 2), ..., 1
```

en el cual:

```
0! = 1
1! = 1
2! = 2 · 1
3! = 3 · 2 · 1
...
```

de modo que una definición recursiva del método factorial *n* es:

```
n! = n · (n - 1)!
```

El algoritmo que resuelve la solución iterativa de un entero *n*, mayor o igual que 0, se puede calcular utilizando un bucle `for`:

```
factorial = 1;
for (contador = n; contador >= 1; contador --)
  factorial *= contador;
```

En el caso de implementar una función se requerirá una sentencia de retorno que devuelva el valor del factorial, tal como:

```
return (factorial)
```

El algoritmo que resuelve el factorial de modo *recursivo* ha de tener presente una condición de salida. Así, en el caso del cálculo de `6!`, la definición es `6! = 6 × 5!` y `5!` de acuerdo a la definición es `5 x 4!`. Este proceso continúa hasta que `1! = 1 × 0!` por definición. En matemáticas la definición de una función en términos de sí misma se denomina **inductiva** y conduce naturalmente a una implementación recursiva. El caso base de `0! = 1` es esencial dado que se detiene, potencialmente, una cadena de llamadas recursivas. Esta condición base o condición de salida deben fijarse en cada caso de una solución recursiva.

El algoritmo que resuelve `n!` de modo recursivo se apoya en la definición siguiente:

```
n! = 1          si    n = 0
n! = n*(n - 1)!  si    n > 0
```

en consecuencia, el algoritmo mencionado que calcula el factorial será:

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

Ejemplo 5.2

Deducir la definición recursiva del producto de números naturales.

El producto $a \cdot b$, donde a y b son enteros positivos, tiene dos soluciones:

Solucíá iterativa

$$a * b = \underbrace{a + a + a + \cdots + a}_{b \text{ veces}}$$

Solución recursiva

```
a * b = a                 si    b == 1
a * b = a * (b - 1) + a   si    b > 1
```

Así por ejemplo, 7 x 3 será :

```
7 * 3 = 7 * 2 + 7 = 7 * 1 + 7 + 7 = 7 + 7 + 7 = 21
```

Se ha transformado el problema de multiplicar dos enteros a, b en otro ploblema más pequeño: multiplicar a por (b - 1); en este caso la condición de salida se produce cuando b es 1 y el producto es a. La codificación en C del problema es inmediata:

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

Ejemplo 5.3

Definir la naturaleza de la serie de Fibonacci: 0, 1, 1, 2, 3, 5, 8 ,13, 21, ..., y codificar la función C que obtenga el n-ésimo término de la serie.

Esta serie comienza con los términos 0 y 1, y se observa que cada elemento, a partir del tercero, es la suma de los dos elementos anteriores:

```
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
...
```

Entonces se puede decir que:

```
fibonacci(0) = 0
fibonacci(1) = 1
...
fibonacci(n)= fibonacci(n - 1) + fibonacci(n - 2)
```

y la definición recursiva será:

```
fibonacci(n) = n                                        si n = 0 o n = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)      si n > = 2
```

Obsérvese que la definición recursiva de los números de fibonacci es diferente de las definiciones recursivas del factorial de un número y del producto de dos números. Así por ejemplo,

```
fibonacci(6) = fibonacci(5) + fibonacci(4)
```

o lo que es igual, fibonacci(6) ha de aplicarse en modo recursivo dos veces, y así sucesivamente.

Un algoritmo iterativo equivalente sería:

```
if (n <=1)
  return(n);
fibinf = 0;
fibsup = 1;
for (i = 2; i <= n; i++)
{
  int x;
  x = fibinf;
  fibinf = fibsup;
  fibsup = x + fibinf;
}
return (fibsup);
```

y la función recursiva:

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

5.2. FUNCIONES RECURSIVAS

Una función **recursiva** es una función que se invoca a sí mismo de forma directa o indirecta. En **recursión directa** el código de la función `f()` contiene una sentencia que invoca a `f()`, mientras que en **recursión indirecta** `f()` invoca a la función `g()` que invoca a su vez a la función `p()`, y así sucesivamente hasta que se invoca de nuevo a la función `f()`.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia la definición recursiva debe incluir un **componente base** (*condición de salida*) en el que $f(n)$ se defina directamente (es decir, no recursivamente) para uno o más valores de n.

Debe existir una «forma de salir» de la secuencia de llamadas recursivas. Así en la función de `Fibonacci` la condición de salida o base es $f(n) = 1$ para $n \leqslant 1$.

$$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2},\quad \text{para } n > 1$$

$F_0 = 0$ y $F_1 = 1$ constituyen el componente base o condiciones de salida y $F_n = F_{n-1} + F_{n-2}$ es el componente recursivo. Un método recursivo correcto debe incluir un componente base o condición de salida ya que en caso contrario se produce una recursión infinita.

A tener en cuenta

Una función es recursiva si es llamada a sí misma, directa o indirectamente, a través de otra función `g()`. Es necesario contemplar un caso base o condición de salida, para dejar de hacer llamadas recursivas.

Problema 5.1

Realizar un programa que escriba los *n* primeros términos de la serie de Fibonacci, siendo *n* un número requerido al usuario.

La componente base, e incluso la función, se escribió anteriormente, ahora simplemente se escribe el programa, que consiste en introducir un entero mayor que 1 y llamar iterativamente a la función `fibonacci()`.

```
#include <stdio.h>
long fibonacci(int n);
int main()
{
   int n, k;

   do {
     printf("Introduzca el número de términos: ");
     scanf("%d", &n);
   }while (n <= 1);
   puts("\tSerie números de fibonacci");
   printf("0 1 ");                    /* componente base de la serie */
   for (k=2; k <= n; k++)
   {
     printf("%ld%c", fibonacci(k),(k%7==0 ?'\n':' '));
   }
   return 0;
}

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

5.2.1. Recursividad indirecta: funciones mutuamente recursivas

La *recursividad indirecta* se produce cuando una función llama a otra, que eventualmente terminará llamando de nuevo a la primera función. El programa `alfabeto.c` visualiza el alfabeto utilizando *recursión mutua* o *indirecta*.

```
#include <stdio.h>
void funcionA(char c);
void funcionB(char c);
int main()
{
   printf("\n");
   funcionA('Z');
   printf("\n");
   return 0;
```

```
}

void funcionA(char c)
{
  if (c > 'A')
    funcionB(c);
  printf("%c", c);
}

void funcionB(char c)
{
  funcionA(--c);
}
```

El programa principal llama a **funcionA()** con el argumento `'Z'` (la última letra del alfabeto). *Esta función examina su parámetro c, si c está en orden alfabético, después que* `'A'`, *llama a* **funcionB()**, que, a su vez, inmediatamente invoca a **funcionA()** pasándole un parámetro predecesor de `c`. Esta acción hace que **funcionA()** vuelva a examinar *c*, y nuevamente una llamada a **funcionA()** *hasta que c sea igual a* `'A'`. *En este momento, la recursión termina ejecutando* **printf()** veintiséis veces y visualizando el alfabeto, carácter a carácter.

Ejemplo 5.4

Determinar si un número entero positivo es par o impar, con dos funciones que se llaman mutuamente.

Para resolver el ejercicio se escriben dos funciones, `par()` e `impar()`, que se llaman mutuamente. El programa principal invoca a `par()` pasando el entero `n` que se desea determinar si es par o impar. Desde la función `par()` se invoca a `impar()` pasando `n-1`, y lo mismo desde `impar()` a `par()`. El *caso base* es que `n==0`; si esto ocurre en el método `par()` es que el entero es par ya que el número de llamadas ha sido múltiplo de 2; y de igual forma si el *caso base* ocurre en `impar()` es que el entero es impar por lo que `par()` debe devolver `false(0)`.

```
#include <stdio.h>
#include <string.h>
int par(int n);
int impar(int n);
int main()
{
   int n;
   char rslt[12];
   do {
      printf("Introduzca un número: ");
      scanf("%d", &n);
   }while (n < 0);
   if (par(n))
      strcpy(rslt, " es par");
   else
      strcpy(rslt, " es impar");
   printf("\t %d %s", n, rslt);
   return 0;
```

```
}

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

int impar(int n)
{
  if (n == 0)
    return 0;              /* no es par */
  else
    return par(n-1);
}
```

5.2.2. Condición de terminación de la recursión

Cuando se implementa una función recursiva será preciso considerar una condición de terminación, ya que en caso contrario la función continuaría indefinidamente llamándose a sí misma y llegaría un momento en que la memoria se agotaría. En consecuencia, sería necesario establecer en cualquier función recursiva la *condición de parada*, terminación o salida que termine las llamadas recursivas y evitar indefinidamente las llamadas. Así, por ejemplo, en el caso de la función `factorial()`, definida anteriormente, la condición de salida puede ser cuando el número sea 1 o 0, ya que en ambos casos el factorial es 1.

En el caso de la recursión mutua entre `par()` e `impar()` la condición de salida es que `n==0`. Además, hay que considerar que cada llamada recursiva a la función se acerque a la condición de salida. Esto se observa claramente en los ejemplos anteriores en los que `n` disminuye en 1 cada vez y así se puede asegurar que `n` será 0 en el caso de `par()` o `impar()`, o bien 1 en el caso del factorial.

5.3. RECURSIÓN *VERSUS* ITERACIÓN

En las secciones anteriores se han estudiado varias funciones que se pueden implementar fácilmente o bien de modo recursivo o bien de modo iterativo. En esta sección comparamos los dos enfoques y examinamos las razones por las cuales el programador puede elegir un enfoque u otro según la situación específica.

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura repetitiva y* la recursión utiliza una estructura de selección. La iteración y la recursión implican ambas repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas a funciones. La iteración y recursión implican ambas un test de terminación (*condición de salida*). La iteración termina cuando la condición del bucle no se cumple mientras que la recursión termina cuando se reconoce un caso base o se alcanza la condición de salida.

La recursión tiene algunas desventajas. Se invoca repetidamente al mecanismo de llamadas a funciones y en consecuencia se necesita un tiempo suplementario para realizar cada llamada.

Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada recursiva produce la realización de una nueva creación y copia de las variables de la

función; esta acción puede consumir considerable memoria e incrementar el tiempo de ejecución. Por el contrario, la iteración se produce dentro de una función de modo que las operaciones suplementarias de las llamadas a la función y asignación de memoria adicional son omitidas.

En consecuencia, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. Sin embargo, en condiciones críticas de tiempo y de memoria (es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema), la solución a elegir debe ser, normalmente, la iterativa.

Nota de ejecución

Cualquier problema que se puede resolver recursivamente, se puede resolver también iterativamente (no recursivamente). Un enfoque recursivo se elige normalmente con preferencia a un enfoque iterativo cuando el enfoque recursivo es más natural para la resolución del problema y produce un programa más fácil de comprender y depurar. Otra razón para elegir una solución recursiva es que una solución iterativa puede no ser clara ni evidente.

Consejo de programación

Se ha de evitar utilizar recursividad en situaciones de rendimiento crítico o exigencia de altas prestaciones en tiempo y memoria, ya que las llamadas recursivas emplean tiempo y consumen memoria adicional.

Ejemplo 5.5

Dado un número natural *n*, obtener la suma de sus dígitos. Esta aplicación es un ejemplo claro de comparación entre la resolución de modo iterativo y de modo recursivo.

La suma de los dígitos de un número se puede expresar con las siguientes fórmulas:

$$suma = suma\ (n/10) + modulo\ (n,10) \quad \text{para} \quad n > 9$$
$$suma = n \quad \text{para} \quad n \leqslant 9 \text{ caso base}$$

Un ejemplo práctico, $n = 259$:

```
suma = suma(259/10) + modulo (259,10)      → 2 + 5 + 9 = 16
          ↓
suma = suma(25/10)  + modulo (25,10)       → 2 + 5 ↑
          ↓
suma = suma(2/10)   + modulo (2,10)        → 2 ↑
```

Solución recursiva

```
int sumaRecursiva(int n)
{
  if (n <= 9)
    return n;
  else
    return sumaRecursiva(n/10) + n%10;
}
```

Solución iterativa

La solución iterativa para hallar la suma de los dígitos del número `n` se construye con un bucle `mientras`, repitiendo la acumulación del resto de dividir `n` por 10 y actualizar `n` en el cociente. La condición de salida del bucle es que `n` sea menor o igual que 9

```
int sumaIterativa(int n)
{
  int suma = 0;
  while (n > 9)
  {
    suma = suma + n%10;
    n /= 10;
  }
  return (suma+n);
}
```

5.3.1. Directrices en la toma de decisión iteración/recursión

1. Considérese una solución recursiva sólo cuando una solución iterativa *sencilla* no sea posible.
2. Utilícese una solución recursiva sólo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables considerando las limitaciones del sistema.
3. Si son posibles las dos soluciones, iterativa y recursiva, la solución recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales a las funciones.
4. En ciertos problemas, la recursión conduce naturalmente a soluciones que son mucho más fiables de leer y comprender que su correspondiente iterativa. En estos casos los beneficios obtenidos con la claridad de la solución suelen compensar el coste extra (en tiempo y memoria) de la ejecución de un programa recursivo.

Una función recursiva que tiene una llamada recursiva como última sentencia (*recursión final*) puede transformarse fácilmente en iterativa reemplazando la llamada con un bucle cuya condición chequea el caso base.

5.4. RECURSIÓN INFINITA

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el caso base o condición de salida.

En realidad la **recursión infinita** significa que cada llamada recursiva produce otra llamada recursiva y ésta a su vez otra llamada recursiva y así para siempre. En la práctica dicho método se ejecutará hasta que la computadora agote la memoria disponible y se produzca una terminación anormal del programa.

El flujo de control de una función recursiva requiere tres condiciones para una terminación normal:

- Un test para detener (o continuar) la recursión (*condición de salida* o *caso base*).
- Una llamada recursiva (para continuar la recursión).
- Un caso final para terminar la recursión.

Problema 5.2

Deducir cuál es la condición de salida del método `mcd()` que calcula el mayor común denominador de dos números enteros `b1` y `b2` (el mcd, máximo común divisor, es el entero mayor que divide a ambos números) y un programa que llame a la función con enteros requeridos.

Supongamos dos números 6 y 124 ; el procedimiento clásico de obtención del **mcd** es la obtención de divisiones sucesivas entre ambos números (124 entre 6) si el resto no es 0, se divide el número menor (6, en el ejemplo) por el resto (4, en el ejemplo) y así sucesivamente hasta que el resto sea 0.

```
124 | 6
04  | 20
  6 | 4        2 | 1
  2 | 1        0 | 2      (mcd = 2)
```

	20	1	2
124	6	4	2
4	2	0	

$mcd = 2$

En el caso de 124 y 6, el **mcd** es 2. En consecuencia la condición de salida es que el resto sea cero. El algoritmo que calcula **mcd** entre dos números *m* y *n* es:

- `mcd(m, n)` es `n` si `n <= m` y `n` divide a `m`
- `mcd(m, n)` es `mcd (n, m)` si `m < n`
- `mcd(m, n)` es `mcd (n, resto de m divido por n)` en caso contrario.

Los pasos anteriores significan: el **mcd** es `n` si `n` es el número más pequeño y `n` divide a `m`; si `m` es el número más pequeño, entonces la determinación del `mcd` se debe ejecutar con los argumentos transpuestos; por último si `n` no divide a `m`, el resto se obtiene encontrando el `mcd` de `n` y el resto de `m` divido por `n`.

El programa que incorpora la función recursiva `mcd()` tiene como datos de entrada los dos números enteros:

```
#include <stdio.h>
int mcd(int m, int n);
int main()
{
   int m,n;
   do {
      printf("Introduzca dos enteros positivos: ");
      scanf("%d %d",&m, &n);
   }while (n <= 0)||(m <= 0);
   printf("\nEl máximo común divisor es %d: ", mcd(m,n));
   return 0;
}

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);
}
```

Al ejecutarse el programa se produce la siguiente salida:

```
Introduzca dos enteros positivos
1265
2970
El máximo común divisor es : 55
```

5.5. ALGORITMOS *DIVIDE Y VENCERÁS*

Una de las técnicas más importantes para la resolución de muchos problemas en computadora es la denominada *divide y vencerás*. El diseño de algoritmos basados en esta técnica consiste en transformar (dividir) un problema de tamaño *n* en problemas más pequeños, de tamaño menor que *n* pero similares al problema original, de modo que resolviendo los subproblemas y combinando las soluciones se pueda construir fácilmente una solución del problema completo (*vencerás*).

Normalmente, el proceso de división de problema en otros de tamaño menor va a dar lugar a problemas unitarios, *caso base*, cuya solución es inmediata. A partir de la obtención de la solución del problema para el caso base, se combinan soluciones que amplían el tamaño del problema resuelto, hasta que el problema original queda resuelto.

Por ejemplo, se plantea el problema de dibujar un segmento en el plano que está conectado por los puntos `(x1,y1)` y `(x2,y2)`. El problema puede descomponerse así: determinar el punto medio del segmento, dibujar dicho punto y *dibujar los segmentos obtenidos al dividir el segmento original por el punto mitad*. El tamaño del problema se ha reducido a la mitad, el hecho de dibujar un segmento se ha transformado en dibujar dos segmentos de tamaño justamente la mitad. Sobre cada segmento mitad se vuelve aplicar el mismo procedimiento, de tal forma que llega un momento en que a base de dividir el segmento se alcanza uno de longitud próxima a cero: se ha llegado al caso unitario. Cada tarea consiste en realizar las mismas acciones, por tanto se puede plantear con llamadas recursivas al proceso de dibujar el segmento pero cada vez con un tamaño menor.

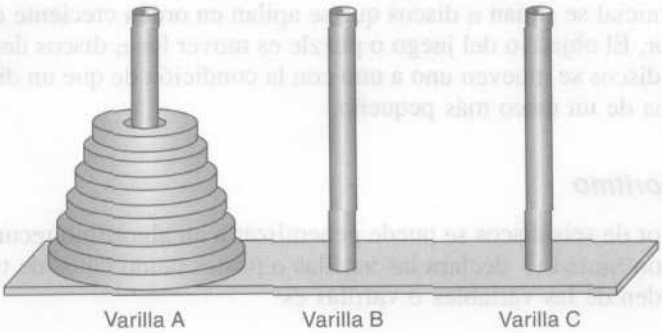
Un algoritmo *divide y vencerás* puede ser definido de manera recursiva, de tal modo que se llama a sí mismo sobre un conjunto menor de elementos. La condición para dejar de hacer llamadas es, normalmente, la obtención de un solo elemento que se denomina *caso base*.

> Un algoritmo *divide y vencerás* divide el problema original en subproblemas más pequeños. Cuando el subproblema es suficientemente pequeño, se resuelve directamente y empieza a combinarse soluciones de subproblemas hasta que queda resuelto el problema.

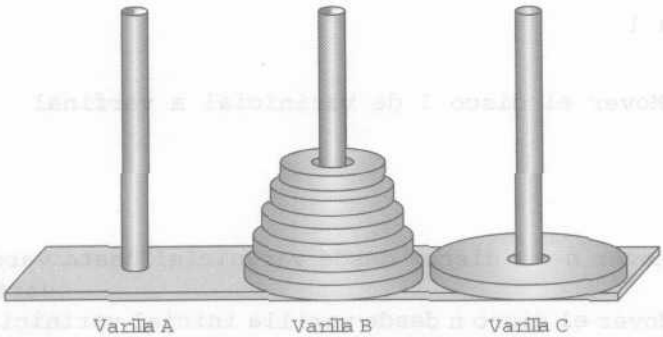
En esta sección se describen problemas clásicos resueltos mediante recursividad. Entre ellos se destacan problemas matemáticos tales como el algoritmo de las Torres de Hanoi y métodos de búsqueda binaria, ordenación rápida, ordenación por mezclas, etc.

5.5.1. Torres de Hanoi

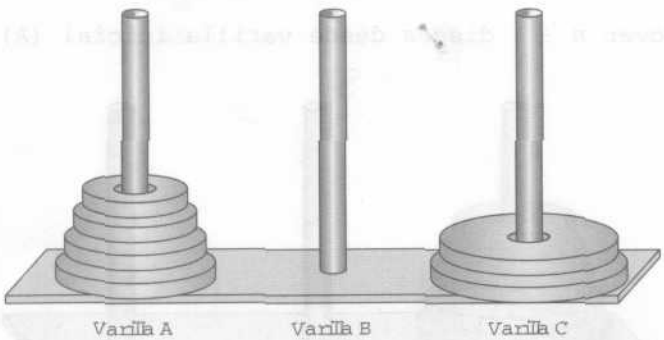
Este juego (un algoritmo clásico) tiene sus orígenes en la cultura oriental y en una leyenda sobre el Templo de Brahma, cuya estructura simulaba una plataforma metálica con tres varillas y discos en su interior. El problema en cuestión suponía la existencia de 3 varillas o postes (A, B y C) en los que se alojaban discos (*n* discos) que se podían trasladar de una varilla a otra libremente pero con una condición: «cada disco era ligeramente inferior en diámetro al que estaba justo debajo de él».



Ilustremos el problema con tres varillas que contengan seis discos en una varilla A, y se desea trasladar a la varilla C conservando la condición de que cada disco sea ligeramente inferior en diámetro al que tiene situado debajo de él. Un primer movimiento puede ser cambiar cinco discos de golpe de la varilla A a la varilla B, y luego el disco más grande a la varilla C. Ya ha habido una transformación del problema en otro de menor tamaño: se ha divido el problema original.



Ahora el problema se centra en pasar los cinco discos de la varilla B a la varilla C y se utiliza un método similar al anterior, pasar los cuatro discos superiores de la varilla B a la varilla A y a continuación se pasa el disco de mayor tamaño de la varilla B a la varilla C y así sucesivamente. El proceso continúa de modo recursivo hasta que finalmente se queda un disco en la varilla B que es *el caso base* y a su vez la condición de parada.



Este problema es claramente recursivo y se puede describir así: *el juego consta de tres varillas (alambres) denominados varilla inicial, varilla central y varilla final.*

En la varilla inicial se sitúan n discos que se apilan en orden creciente de tamaño con el mayor en la parte inferior. El objetivo del juego o puzzle es mover los n discos desde la varilla inicial a la varilla final. Los discos se mueven uno a uno con la condición de que un disco mayor nunca puede ser situado encima de un disco más pequeño.

Diseño del algoritmo

El ejemplo anterior de seis discos se puede generalizar a un algoritmo recursivo con n discos y tres varillas. La función `hanoi()` declara las varillas o postes como datos de tipo `char`. En la lista de parámetros, el orden de las variables o varillas es:

```
varinicial      varcentral      varfinal
```

lo que implica que se están moviendo discos desde la varilla inicial a la final utilizando la varilla central como auxiliar para almacenar los discos. Si $n = 1$ se tiene la condición de parada, el caso base, se puede resolver directamente moviendo el único disco desde la varilla inicial a la varilla final. El algoritmo sería el siguiente:

```
1. Si n es 1

      1.1 Mover el disco 1 de varinicial a varfinal

2. Si no

      1.2 Mover n - 1 discos desde varinicial hasta varcentral utilizando
                                                varfinal como auxiliar
      1.3 Mover el disco n desde varilla inicial varinicial a varfinal
      1.4 Mover n - 1 discos desde la varilla auxiliar varcentral a varfinal
                                          utilizando como auxiliar varinicial
```

Es decir si n es 1, se alcanza la condición de salida o terminación del algoritmo. Si n es mayor que 1, las etapas recursivas `1.2`, `1.3` y `1.4` son tres subproblemas más pequeños, uno de los cuales es la condición de salida.

Las Figuras 5.1 a 5.6 muestran el algoritmo anterior:

Etapa 1: `Mover n - 1 discos desde varilla inicial (A)`

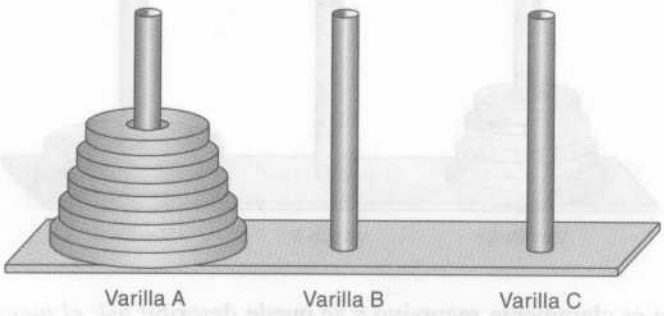


Figura 5.1. Situación inicial.

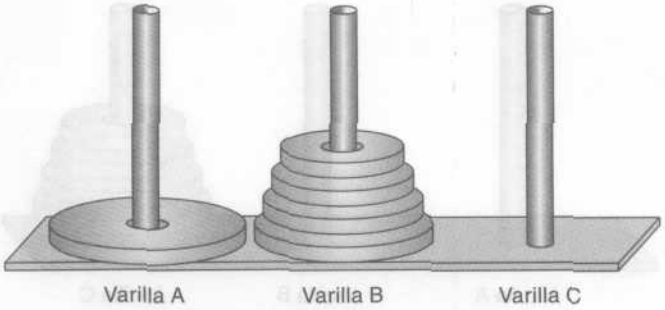


Figura 5.2. Después del movimiento.

Etapa 2: `Mover un disco desde A a C:`

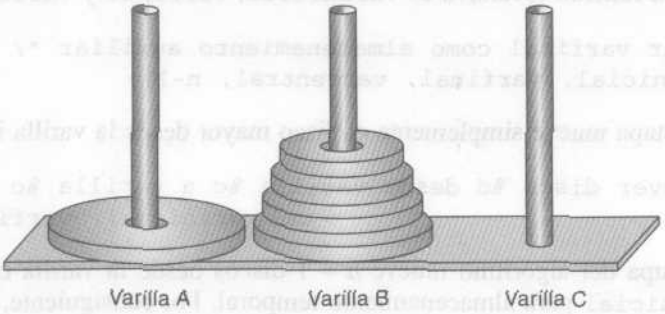


Figura 5.3. Situación de partida.

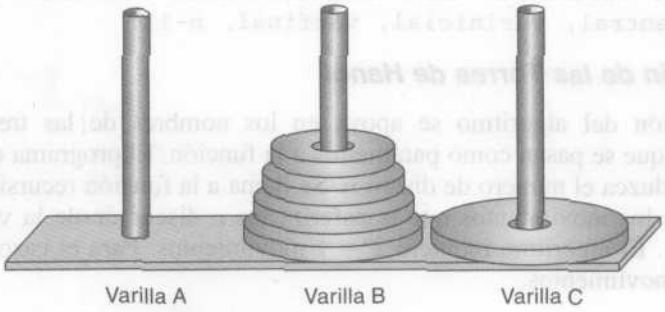


Figura 5.4. Después de la etapa 2.

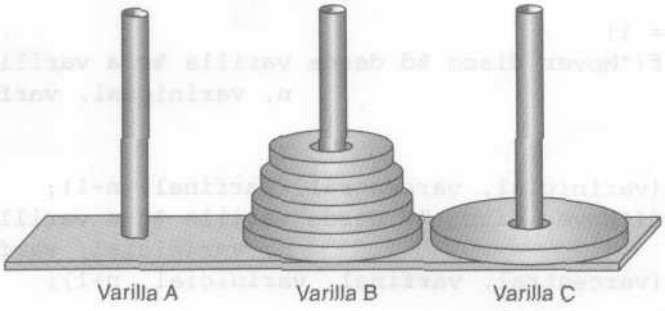


Figura 5.5. Antes de la etapa 3.

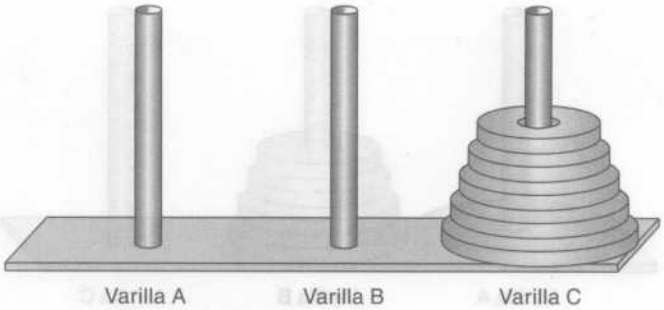


Figura 5.6. Después de la etapa 3.

La primera etapa en el algoritmo mueve $n - 1$ discos desde la varilla inicial a la varilla central utilizando la varilla final como almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada a la función recursiva es `varinicial`, `varfinal` y `varcentral`.

```
/* utilizar varfinal como almacenamiento auxiliar */
hanoi(varinicial, varfinal, varcentral, n-1);
```

La segunda etapa mueve simplemente el disco mayor desde la varilla inicial a la varilla final:

```
printf("Mover disco %d desde varilla %c a varilla %c \n",
                                n, varinicial, varfinal);
```

La tercera etapa del algoritmo mueve $n - 1$ discos desde la varilla central a la varilla final utilizando `varinicial` para almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada a la función recursiva es: `varcentral`, `varinicial` y `varfinal`

```
/* utilizar varinicial como almacenamiento auxiliar */
hanoi(varcentral, varinicial, varfinal, n-1);
```

Implementación de las Torres de Hanoi

La implementación del algoritmo se apoya en los nombres de las tres varillas o alambres: `'A'`, `'B'` y `'C'`, que se pasan como parámetros a la función. El programa comienza solicitando al usuario que introduzca el número de discos `n`. Se llama a la función recursiva `hanoi()` para obtener un listado de los movimientos que transferirá los `n` discos desde la varilla inicial, `'A'`, a la varilla final, `'C'`. El algoritmo requiere $2^n - 1$ movimientos. Para el caso de 10 discos, el juego requerirá 1.023 movimientos.

```
/* Solución recursiva del problema de las Torres de Hanoi */
void hanoi(char varinicial, char varfinal, char varcentral, int n)
{
   if (n == 1)
     printf("Mover disco %d desde varilla %c a varilla %c \n",
                                n, varinicial, varfinal);
   else
   {
     hanoi(varinicial, varcentral, varfinal, n-1);
     printf("Mover disco %d desde varilla %c a varilla %c \n",
                                n, varinicial, varfinal);
     hanoi(varcentral, varfinal, varinicial, n-1);
   }
}
```

Una ejecución de el método para el caso de mover tres discos desde las varillas `A` a `C` tomando la varilla `B` como varilla central o auxiliar, se puede seguir con la siguiente sentencia:

```
hanoi ('A', 'C', 'B', 3);
```

que resuelve el problema de 3 discos desde `A` a `C`. La salida generada sería:

```
Mover disco 1 desde varilla A a varilla C
Mover disco 2 desde varilla A a varilla B
Mover disco 1 desde varilla C a varilla B
Mover disco 3 desde varilla A a varilla C
Mover disco 1 desde varilla B a varilla A
Mover disco 2 desde varilla B a varilla C
Mover disco 1 desde varilla A a varilla C
```

Consideraciones de eficiencia en las Torres de Hanoi

Es de destacar que el método `hanoi()` resolverá el problema de las Torres de Hanoi para cualquier número de discos. El problema de 3 discos se resuelve en un total de 7 ($2^3 - 1$) llamadas al método `Hanoi` mediante 7 movimientos de disco. El problema de 5 discos se resuelve con 31 ($2^5 - 1$) llamadas y 31 movimientos. En general, como ya se ha expresado anteriormente, el número de movimientos, requeridos para resolver el problema de n discos es $2^n - 1$. Cada llamada a la función requiere la asignación e inicialización de un área local de datos en la memoria, por lo que el tiempo de computadora se incrementa exponencialmente con el tamaño del problema. Por estas razones, la ejecución de la función con un valor de n mayor que 10 requiere gran cantidad de prudencia para evitar desbordamientos de memoria y ralentización de tiempo.

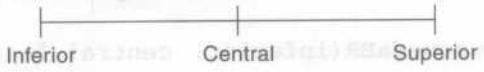
Nota de ejecución

La complejidad del algoritmo que resuelve el problema de las Torres de Hanoi es exponencial. Por consiguiente, a medida que crece n, crece exponencialmente el tiempo de ejecución de la función.

5.5.2. Búsqueda binaria recursiva

La búsqueda binaria es un método de localización de una clave especificada dentro de una lista o array ordenado de n elementos que realiza una exploración de la lista hasta que se encuentra o se decide que no se encuentra en la lista. El algoritmo de búsqueda binaria se puede describir recursivamente aplicando la técnica *divide y vencerás*.

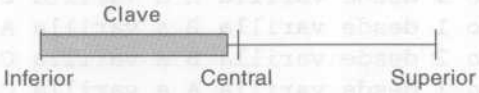
Supóngase que se tiene una lista ordenada `a[]` con un límite inferior y un límite superior. Dada una clave (valor buscado), se comienza la búsqueda en la posición central de la lista (índice central).



`central = (inferior + superior)/2` Comparar `a[central]` y `clave`

Si se produce coincidencia (se encuentra la clave), se tiene la condición de terminación que permite detener la búsqueda y devolver el índice central. Si no se produce la coincidencia (no se encuentra la clave), dado que la lista está ordenada, se centra la búsqueda en la «sublista inferior» (a la izquierda de la posición central) o en la «sublista derecha» (a la derecha de la posición central); el problema de la búsqueda se ha dividido en la mitad, el tamaño de la lista donde buscar es la mitad del tamaño anterior. Cada vez, el tamaño de la lista se reduce a la mitad, así hasta que se encuentre el elemento, o bien la lista resultante esté vacía.

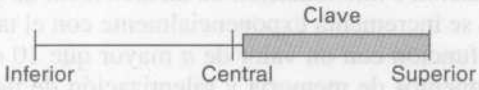
1. Si `clave < a[central]`, el valor buscado sólo puede estar en la mitad izquierda de la lista con elementos en el rango `inferior` a `central - 1`.



Búsqueda en sublista izquierda

```
[inferior .. central - 1]
```

2. Si `clave > a[central]`, el valor buscado sólo puede entrar en la mitad derecha de la lista con elementos en el rango de índices, `central + 1` a `Superior`.



3. El proceso recursivo continúa la búsqueda en sublistas más y más pequeñas. La búsqueda termina o con éxito (*aparece la clave buscada*) o sin éxito (*no aparece la clave buscada*), situación que ocurrirá cuando el límite superior de la lista sea más pequeño que el límite inferior. La condición `inferior > superior` será la condición de salida o terminación y el algoritmo devuelve el índice –1.

La siguiente función expresa la búsqueda binaria, el array o lista `a[]` es de tipo `int`. La función devuelve el índice donde se encuentra el elemento buscado, o bien –1 si no se encuentra:

```
int busquedaBR(int inferior, int superior, int clave)
{
   int central;
   if (inferior > superior)              /* no encontrado */
      return -1;
   else
   {
      central = (inferior + superior)/2;
      if (a[central] == clave)          /* encontrado */
         return central;
      else if (a[central] < clave)
         return busquedaBR(central+1, superior, clave);
      else
         return busquedaBR(inferior, central-1, clave);
   }
}
```

5.6. ORDENACIÓN POR MEZCLAS: *MERGESORT*

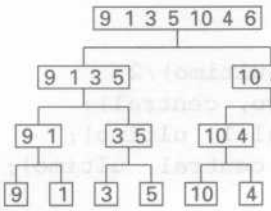
La idea básica de este método de ordenación es la mezcla (*merge*) de listas ya ordenadas. Este algoritmo sigue la estrategia típica de los algoritmos *divide y vencerás*. Los pasos que sigue se basan en dividir el problema de ordenar *n* elementos en dos subproblemas más pequeños, y una vez resueltos combinar la solución (mezclar) para así resolver el problema original. Con más detalle: *se ordena la primera mitad de la lista, se ordena la segunda mitad de la lista y una vez ordenadas su mezcla da lugar a una lista de elementos ordenada; a su vez, la ordenación de la sublista mitad sigue los mismos pasos, ordenar la primera mitad, ordenar la segunda mitad y mezclar.* La sucesiva división de la lista actual en dos hace que el problema (número de elementos) cada vez sea más pequeño; así hasta que la lista actual tenga 1 elemento y, por tanto, se considera ordenada, es el caso base y a partir de dos sublistas de un número mínimo de elementos, se mezclan dando cada vez lugar a listas ordenadas de cada vez más elementos hasta alcanzar la lista total. El Ejemplo 5.6 muestra este proceso de división y mezcla.

Ejemplo 5.6

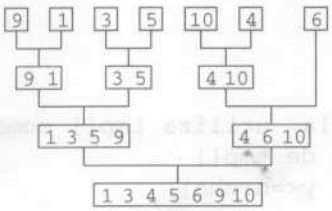
Seguir la estrategia del algoritmo *mergesort* para ordenar la lista:

9 1 3 5 10 4 6

Se representa el proceso con las siguientes figuras en las que aparecen las divisiones.



La mezcla comienza con las sublistas de un solo elemento, que dan lugar a otra sublista del doble de elementos ordenados. El proceso continúa hasta que se construye un única lista ordenada. A continuación se muestra la creación de las sublistas ordenadas:



5.6.1. Algoritmo *mergesort* en C

Este algoritmo de ordenación se diseña fácilmente con ayuda de las llamadas recursivas para dividir las listas en dos mitades; posteriormente, invocar a la función de mezcla de dos listas ordenadas. La delimitación de las dos listas se puede hacer con tres índices: `primero`, `central` y `ultimo`, que apuntan a los elementos del array de iguales significados que los identificadores. Así si se tiene una lista de 10 elementos, los valores de los índices son:

```
primero = 0; ultimo = 9;   central = (primero+ultimo)/2 = 4
```

La primera sublista comprende los elementos $a_0 \ldots a_4$; y la segunda los elementos siguientes $a_{4+1} \ldots a_9$. Los pasos del algoritmo para el array a:

```
mergesort(a, primero, ultimo)
  1. Si (primero < ultimo) Entonces
     1.1 central = (primero+ultimo)/2
     1.2 mergesort(a, primero, central);   ordena primera mitad de la lista
     1.3 mergesort(a, central+1, ultimo);  ordena segunda mitad de la lista
     1.4 mezcla(a, primero, central, ultimo);  fusiona las dos sublista
                                               ordenadas, delimitadas por los
                                               extremos
     fin
```

La codificación en C consta de la función `mergesort()` y la función que fusiona dos sublistas ordenadas, `mezcla()`. Se da por supuesto que el máximo de elementos está representado por la constante `M`.

```
#define M 20
void mezcla(double a[], int izda, int medio, int drcha);
void mergesort(double a[], int primero, int ultimo)
{
   int central;
   if (primero < ultimo)
   {
      central = (primero+ultimo)/2;
      mergesort(a, primero, central);
      mergesort(a, central+1, ultimo);
      mezcla(a, primero, central, ultimo);
   }
}
void mezcla(double a[], int izda, int medio, int drcha)
{
   double tmp[M];
   int x, y, z;
   x = z = izda;
   y = medio+1;
     /* bucle de mezcla, utiliza tmp[] como array auxiliar,
       z es el índice de tmp[]                                  */
   while (x<=medio && y<=drcha)
   {
     if (a[x]<=a[y])
       tmp[z++] = a[x++];
     else
       tmp[z++] = a[y++];
   }
     /* bucle para mover elementos que quedan de sublistas */
   while (x <= medio)
     tmp[z++] = a[x++];
   while (y <= drcha)
     tmp[z++] = a[y++];
     /* Copia de elementos de tmp[] al array a[] */
```

```
    for (z = izda; z <= drcha; z++)
        a[z] = tmp[z];
}
```

5.7. *BACKTRACKING*, ALGORITMOS DE VUELTA ATRÁS

La resolución de algunos problemas exige probar sistemáticamente todas las posibilidades que pueden existir para encontrar la solución. Los **algoritmos de vuelta atrás** utilizan la recursividad para probar cada una de las posibilidades de encontrar la solución.

Este método de resolución de problemas recurre a realizar una búsqueda exhaustiva, sistemática de una posible solución al problema planteado. Estos algoritmos descomponen el proceso de búsqueda o tanteo de una solución en tareas parciales. Cada tarea parcial realiza las mismas acciones que la tarea anterior y por eso se expresa frecuentemente en forma recursiva.

Así por ejemplo, en el problema de determinar los sucesivos saltos que debe de hacer un caballo (con su forma típica de moverse) para que pase por todas las casillas de un tablero de ajedrez, la tarea parcial es realizar un salto válido, comprobando que dicho salto esté dentro de las coordenadas del tablero y que no haya pasado ya por la casilla destino. En este problema, se resuelve posteriormente, todas las posibilidades son los ocho saltos que puede realizar desde una casilla dada.

El proceso general de los algoritmos de *vuelta atrás* se contempla como un método de prueba o búsqueda, que gradualmente construye tareas básicas y las inspecciona para determinar si conducen a la solución del problema. Si una tarea no conduce a la solución, prueba con otra tarea básica hasta agotar todas las posibilidades. Es una prueba sistemática hasta llegar a la solución, o bien determinar que no hay solución por haberse agotado todas las opciones que probar.

Una de las características principales de los algoritmos de *vuelta atrás* es la búsqueda exhaustiva, con todas las posibilidades, de soluciones parciales que conducen a la solución del problema. Otra característica es la *vuelta atrás*; es el sentido de que si una solución o tarea

```
    } while ((x < 1) || (x > N) ||
             (y < 1) || (y > N));
                        /* Valores iniciales del tablero */
    for(i = 1; i<= N; i++)
      for(j = 1; j<= N; j++)
        tablero[i][j] = 0;
    tablero[x][y] = 1;
        /* Llamada con coordenadas iniciales y primer salto */
    solucion = 0;
    saltoCaballo(2, x, y, &solucion);
    if (solucion)
    {
      puts("\n\tEl problema tiene solucion:\n");
      escribeTablero();
    }
    else
      puts("¡¡ No se ha encontrado solucion al problema !! ");
    return 0;
}

void saltoCaballo(int i,int x,int y,int* exito)
{
    int nx, ny;
    int k;
    *exito = 0;
    k = 0;      /* inicializa el conjunto posible de movimientos */
    do {
      k++;
      nx = x+d[k-1][0];
      ny = y+d[k-1][1];
            /* determina si nuevas coordenadas son aceptables */
      if ((nx >= 1) && (nx <= N) &&
          (ny >= 1) && (ny <= N) &&
          (tablero[nx][ny] == 0))
      {
          tablero[nx][ny]= i;        /* anota movimiento */
          if (i < N*N)
          {
            saltoCaballo(i+1,nx,ny,exito);
                  /* se analiza si se ha completado la solución */
            if (!(*exito))
            {                   /* no se alcanzó la solución */
              tablero[nx][ny] = 0;        /* se borra anotación */
          }
            }
            else
              *exito = 1 ;        /* caballo ha cubierto el tablero */
      }
    } while ((k < 8) && !(*exito));
}

void escribeTablero()
{
    int i,j;
```

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

5.7.2. Problema de las ocho reinas

El juego de colocar ocho reinas en un tablero de ajedrez sin que se «ataquen» entre sí es un ejemplo del uso de los métodos de búsqueda sistemática y de los algoritmos de *vuelta atrás*.

El problema se plantea de la forma siguiente: «dado un tablero de ajedrez (8 × 8 casillas), hay que situar ocho reinas de forma que ninguna de ellas pueda actuar (''comer'') a cualquiera de las otras». Hay que recordar, en primer lugar, la regla del ajedrez respecto de los movimientos de la reina: «puede moverse a lo largo de la columna, fila y diagonales donde se encuentra».

Antes de empezar a aplicar los pasos que siguen estos algoritmos, se van a *podar* posibles movimientos de las reinas. Cada columna puede contener una y sólo una reina, la razón de esta restricción es inmediata: si por ejemplo en la columna 1 se encuentra la reina 1 y en esta columna se sitúa otra reina entonces se *atacan* mutuamente. De las `n` × `n` casillas que puede ocupar una reina, se limita su ubicación a las 8 casillas de la columna en que se encuentra. De tal forma que si se numeran las reinas de 1 a 8, entonces la reina `i` se va a situar en alguna de las casillas de la columna `i`.

El *primer paso* para aplicar el algoritmo de vuelta atrás, la tarea básica que exhaustivamente se va a tantear es colocar la reina `i`, el número de alternativas que se tiene son las 8 posibles filas de la columna `i`. La comprobación de que una selección es valida tiene que hacerse investigando que en dicha fila y en las dos diagonales no haya otra reina colocada anteriormente. En cada paso se amplía el número de reinas colocadas.

El *segundo paso* es analizar la *vuelta atrás*, ¿qué ocurre si no se alcanza la solución, si no se es capaz de colocar las ocho reinas?: «se borra la ubicación, la fila donde se ha colocado, y se ensaya con la siguiente fila válida».

La realización de cada tarea supone ampliar el número de reinas colocadas, hasta llegar a la solución completa, o bien determinar que no es posible colocar la reina. Entonces, en el retroceso se coloca la reina en otra fila válida para realizar un nuevo tanteo.

Representación de los datos

En cuanto a los tipos de datos para representar las reinas en el tablero, como lo que nos interesa es determinar en qué fila se sitúa la reina que está en la columna `i`: se define un vector entero. El contenido de una posición del vector va a ser 0, o bien el índice de fila donde se sitúa la reina. Por esta coincidencia, el parámetro `i` sirve de índice de columna dentro de la cual podremos colocarla en los ocho posibles valores de fila.

Como los arrays en C siempre tienen como índice inferior 0, se prefiere reservar una posición más y así la numeración de las reinas coincide con el índice del vector.

```
#define N 8
#define n (N+1)
int reinas[n];
```

Parámetros de la llamada recursiva

La función recursiva va a tener los dos parámetros necesarios para situar una nueva reina: el número de reina y una variable que cambie de valor (un *switch* o interruptor) cuando se haya completado el problema. El vector con las reinas colocadas va a tener ámbito global.

Codificación del algoritmo de las 8 reinas

Es importante señalar cómo se determina que es válida la ubicación de la reina actual. Esta tarea se realiza en una función auxiliar que inspecciona si no se *ataca* (en la fila y dos diagonales) con las reinas colocadas hasta ese momento. Para esto, un bucle con tantas iteraciones como reinas situadas `(j-1)` comprueba que no coincidan las filas ni las diagonales. En el elemento `reinas[r]` se encuentra la fila donde se ubica la reina número `r`, y domina dos diagonales: `(reinas[r] + r)` y `(reinas[r] - r)`.

```
#include <stdio.h>
#define N 8
#define n (N+1)
void colocaReina(int i, int* solucion);
int valido(int j);
void escribirUbicacion();
int reinas[n];
int main()
{
    int tieneSolucion;
                                    /* En la llamada se pasa la reina 1 */
    colocaReina(1,&tieneSolucion);
    if (tieneSolucion)
        escribirUbicacion();
    return 0;
}
void colocaReina(int i, int* solucion)
{
    int k;
    *solucion = 0;
    k = 0;                          /* inicializar posibles movimientos */
    do {
        k++;
        reinas[i] = k;          /* tentativa de colocar reina i en fila k,
                                    a la vez queda anotado movimiento */
        if (valido(i))
        {
            if (i < N)              /* no completado el problema */
            {
                colocaReina(i+1,solucion);  /* vuelta atrás */
                if (!(*solucion))
                    reinas[i] = 0;
            }
            else                    /* todas las reinas colocadas */
                *solucion = 1;
        }
    } while(!(*solucion) && (k < 8));
}
int valido(int j)
{
    /* Inspecciona si la reina de la columna j es atacada por alguna
       reina colocada anteriormente */
    int r;
```

```
    int v;                  /* switch(interruptor) toma valor 1 si es válido */
    v = 1;
    for(r = 1; r <= j-1; r++)
    {
      v = v && (reinas[r]!= reinas[j]);        /* no esté en la misma fila
                                                  y no esté en alguna de las
                                                  dos diagonales */
      v = v && ((reinas[j] + j) != (reinas[r] + r));
      v = v && ((reinas[j] - j) != (reinas[r] - r));
    }
    return v;
}
void escribirUbicacion()
{
    int k;
    puts("\n Número de fila donde se colocan cada reina\n");
    for ( k = 1; k <= N; k++)
      printf("\t Reina %d en fila %d \n",k,reinas[k]);
}
```

5.8. RESOLUCIÓN DE PROBLEMAS CON ALGORITMOS DE *VUELTA ATRÁS*

A continuación se resuelven unos supuestos en los que se aplican los pasos que siguen los algoritmos basados en la búsqueda exhaustiva de todas las posibilidades. En alguno de ellos no se siguen los mismos pasos que los realizado en el *salto del caballo* o las *ocho reinas*, sino que se adapta para aprovechar el hecho de agotar todas las posibilidades.

5.8.1. Generar las distintas variaciones $V_{m,n}$

Se desea escribir los distintos subconjuntos de n elementos que se pueden obtener con un conjunto de m elementos.

Por ejemplo, sea C = {a, b, c}, los distintos subconjuntos de dos elementos que se pueden formar, diferenciándose uno de otro por el orden que ocupan o bien por ser distintos son:

$V_{3,2}$ = { (a,b), (a,c), (b,a), (b,c), (c,a), (c,b)}

Se puede observar que no se repiten los elementos en un mismo subconjunto; no es válido, por ejemplo, (a,a).

Algoritmo para resolver el problema

Se va a seguir el esquema general de los algoritmos de *vuelta atrás*, aunque se han de introducir algunos cambios respecto de los dos últimos problemas. Ahora interesa obtener todas las soluciones posibles, todos los subconjuntos, y no detener el proceso cuando se alcanza una solución. Como *consecuencia de lo anterior, el bucle ya no va a ser condicional sino un bucle que itere para cada* uno de los elementos:

```
desde i = 1 hasta m hacer
  si <elemento_i no forma parte del subconjunto> entonces
     <incorporar elemento_i al subconjunto>
```

El índice del bucle se corresponde con el elemento que pasa a formar parte del subconjunto o variación actual. La tarea básica en este problema es incorporar un nuevo elemento a la variación actual. Un subconjunto, variación, termina de formarse cuando el número de elementos es n, ($V_{m,n}$). Entonces el algoritmo es:

```
Ensayar(p:entero, C: conjunto)
desde i = 1 hasta m hacer
  si <elemento_i no forma parte del subconjunto> entonces
     <incorporar elemento_i al subconjunto>
     si p < n entonces
       Ensayar(p+1,C-elemento_i )
     sino
       escribir(subconjunto)
     fin-si
  fin-si
fin-desde
```

La variable p lleva la cuenta del número de elementos de la variación.

Representación de los datos

En cuanto a la representación de los datos, se supone que el conjunto sobre el que se obtienen las variaciones son las letras del alfabeto:

```
char cnj[] = {'a','b','c','d','e', ...};
```

Esta secuencia de letras se hacen corresponder con una secuencia paralela de valores enteros, en la cual se toma como base 0 con el fin de determinar si una letra está o no en la selección actual:

```
int c[] = {0, 1, 2, 3, 4, ...};
```

Para guardar los elementos de la variación actual se utiliza un array `(int v[] )`. Como hay una correspondencia directa entre el índice y el elemento, este array almacena únicamente los índices. En el momento de escribir por pantalla la variación, los índices guardados en `v[]` se transforman en los elementos. Por ejemplo, si se están formando variaciones de 2 elementos y se tiene que `v[]={0,4}`, se escriben `cnj[v[0]],cnj[v[1]]`, que se corresponden con `cnj[0],cnj[4] = = 'a','e'`.

Codificación del algoritmo

La entrada del programa es el par de valores `m,n` que forman `Vm,n`. El valor de `m` y `n` no puede superar el máximo de letras del conjunto que se considera. Para no aumentar el tamaño del problema se toma un máximo: las 12 primeras letras del alfabeto. Cuando se incorpora un elemento `i` a la variación actual, este se retira del conjunto de índices `c[]`, y para ello se asigna en la posición `i-1` un valor clave, que se elige `-1`. En la *vuelta atrás* hay que reponer el elemento y se hace asignando en la posición `i-1` el valor `i-1`.

Para comprobar si un elemento no está ya en la variación actual, simplemente se pregunta si el índice del elemento está en `c[]`; en el caso de estar, es que no se incorporó anteriormente.

En la codificación se debe tener en cuenta que al indexarse los arrays en C con el índice inferior `0`, el elemento `i` se corresponde con la posición `i-1`.

```
#include <stdio.h>
#define M 12
```

```
#define CLAVE -1
int m,n;
char cnj[M]= {'a','b','c','d','e','f','g','h','i','j','k','l'};
int c[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
int v[M];
void variaciones(int p, int c[]);
void escribir(int v[]);
int main()
{
   do {
      printf("\n m = "); scanf("%d", &m);
      printf("  n = "); scanf("%d", &n);
   } while((m > M) || (n > m));
   printf("\n \tVariaciones (%d,%d)\n", m, n);
   variaciones(1, c);
   return 0;
}
void variaciones(int p, int c[])
{
   int i;
   for (i = 1; i <= m; i++)
   {
      if (c[i-1] == i-1)          /* i no está en la variación actual */
      {
         v[p-1] = i-1;            /* p-1 debido a índice base 0 */
         if (p < n)
         {
           c[i-1] = CLAVE;
           variaciones(p+1,c);
           c[i-1] = i-1;          /* vuelta atrás: repone el índice */
         }
         else
         {
           escribir(v);
           v[p-1] = 0;
         }
      }
   }
}

void escribir(int v[])
{
   int k;
   printf(" (");
   for (k = 1; k <= n; k++)
   {
      printf("%c", cnj[v[k-1]]);
      printf("%c", (k < n ? ',': ')'));
   }
}
```

5.8.2. Objetos que suman un peso

Se puede plantear este problema: *Se tiene un conjunto de objetos de pesos* `p1, p2, p3, ..., pn`, *y se desea estudiar, como objetivo, si existe una selección de dichos objetos que totalice exactamente un peso v dado.*

Por ejemplo, si el objetivo `v = 12` y los pesos son `4, 3, 6, 2, 1`; se pueden elegir el primero, el tercero y el cuarto, ya que `4 + 6 + 2 = 12`.

Algoritmo para resolver el problema

El algoritmo para solucionar el problema tiene como tarea básica añadir un peso nuevo, probar si con ese peso se alcanza el peso objetivo, o se avanza en el sentido de alcanzar la solución.

De nuevo hay una búsqueda sistemática, exhaustiva, de la solución dando pasos hacia adelante. Si se llega a una situación en la que no se consigue el peso objetivo porque es superado, entonces se retrocede, vuelta atrás, para eliminar el objeto añadido y probar con otro para inspeccionar si a partir de él se consigue el objetivo. El proceso de selección termina en el momento de totalizar el peso objetivo, o bien se haya ensayado con todos los objetos.

El número de posibilidades con las que se pueden ensayar son los `n` objetos. Es inmediato que un objeto ya seleccionado no se puede volver a seleccionar.

Representación de los datos

La secuencia de objetos que forman el problema se representa en un vector de tipo entero, cada posición del vector almacena el peso del objeto; el vector se agrupa con el número de objetos en una estructura.

```
struct objetos
{
  int n;
  int objs[M];
};
```

La bolsa donde se meten los objetos que forman la selección actual se puede representar por un tipo conjunto, un estructura con dos campos: `k` (número de objetos) y un array donde se encuentran los objetos. Realmente, en vez de guardar los objetos se guarda los índices de dichos objetos para posteriormente ser recuperados a partir de dicho índice.

```
struct bolsa
{
  int k;
  int b[M];
};
```

Codificación del algoritmo

Los datos de entrada que va a tener el programa son los objetos y el peso objetivo. La salida es el conjunto de objetos que forma la selección que totaliza el peso objetivo, o bien un literal informando de que no se encuentra la selección.

El hecho de añadir un peso, la anotación, se realiza acumulando el peso y metiendo el índice que le corresponde en la bolsa. Para borrar la anotación, en la *vuelta atrás*, se opera a la inversa.

Al codificar este programa se utilizan todas las posiciones del array, desde la posición `0` a la última `(M-1)`. Es por ello por lo que el `candidato == 1`, por ejemplo, se guarda como `candidato - 1` para que así se corresponda con la posición donde se guarda el peso del objeto.

```
#include <stdio.h>
#define M 12
typedef struct
{
  int n;
  int objs[M];
} objetos;
objetos p;
typedef struct
{
  int k;
  int b[M];
} bolsa;
bolsa bs;
void seleccion(int obj, int candidato, int suma, int* encontrado);
void entrada();
void escribirSeleccion();
int main()
{
   int solucion;
   int acumulado, objetivo;
   puts("\n\t Objetos disponibles");
   entrada();
   printf("\tPeso que se quiere obtener: ");
   scanf("%d", &objetivo);
   bs.k = 0;          /* inicializa la bolsa de pesos a vacía */
   seleccion(objetivo, 0, 0, &solucion);
   if (solucion)
     escribirSeleccion();
   else
     puts("No hay selección que totalice el peso objetivo");
   return 0;
}

void seleccion(int obj, int candidato, int suma, int* encontrado)
{
  /* obj: peso objetivo;
     candidato: índice del peso a añadir;
     suma: suma parcial de pesos      */
  *encontrado = 0;
  while ((candidato < p.n) && !(*encontrado))
  {
     candidato++;
     if ((suma + p.objs[candidato - 1]) <= obj)
     {
       bs.k++;
       bs.b[bs.k-1] = candidato - 1 ;   /* el objeto es anotado */
       suma += p.objs[candidato - 1];
```

```
            if (suma < obj)          /* se ensaya con siguiente objeto */
            {
              seleccion(obj, candidato, suma, encontrado);
              if (!(*encontrado))        /* vuelta atrás, se extrae
                                            objeto */
              {
                bs.k--;
                suma -= p.objs[candidato - 1];
              }
            }
            else
              *encontrado = 1;
          }
        }
      }
      void entrada()
      {
        int j;
        do {
          printf("\tNúmero de objetos: ");
          scanf("%d", &p.n);
        } while ((p.n < 1) || (p.n > M));
        for (j = 1; j <= p.n; j++)
        {
          printf("\tObjeto %d , peso = ", j);
          scanf("%d", &p.objs[j-1]);
        }
      }
      void escribirSeleccion()
      {
        int i;
        printf("\Selección de objetos que totalizan el objetivo\n");
        for (i = 1; i <= bs.k; i++)
        {
          printf(" Objeto %d cuyo peso es %d\n",
                                  bs.b[i-1] + 1, p.objs[bs.b[i-1]]);
        }
      }
```

5.9. SELECCIÓN ÓPTIMA

En los problemas del *salto de caballo*, *ocho reinas* y *selección de objetos* se ha aplicado la estrategia de *vuelta atrás* para encontrar una única solución. Aplicando la misma estrategia se puede hacer una ampliación y encontrar así todas las soluciones. Por ejemplo, encontrar todas las maneras distintas de colocar la reinas en el tablero sin que se «coman», o encontrar todas las selecciones posibles de objetos que suman un peso objetivo dado.

Las variaciones en el algoritmo para encontrar todas las soluciones son mínimas, si se quita la variable que toma el estado 1 (`true`) cuando se encuentra la solución, y cada vez que se encuentre una solución llamar a la función que la escriba, tendremos el problema resuelto. En el Ejemplo 5.7 se modifica el programa ya planteado.

Ejemplo 5.7

Dado un conjunto de pesos, se desea escribir todas las combinaciones de ellos que totalicen un peso dado.

El problema se resuelve probando con todas las combinaciones posibles de los objetos disponibles. Cada vez que la suma de los pesos de una combinación sume el peso dado, se escriben los objetos que forman dicha combinación.

Realmente, el problema se resuelve modificando la función `seleccion()` del Apartado 5.8.2. La variable `encontrado` ya no es necesaria, el bucle ha de iterar con todos los objetos, cada vez que `suma` sea igual al peso objetivo, `obj`, y se llama a `escribirSeleccion()`. Ahora, en la *vuelta atrás* siempre se borra de la «bolsa» la anotación del objeto realizada, para así probar con los `n` objetos.

```
void seleccion(int obj, int candidato, int suma)
{
while (candidato < p.n)
{
   candidato++;
   if ((suma + p.objs[candidato - 1]) <= obj)
   {
      bs.k++;
      bs.b[bs.k-1] = candidato - 1 ;        /* el objeto es anotado */
      suma += p.objs[candidato - 1];
      if (suma < obj)        /* se ensaya con siguiente objeto */
      {
        seleccion(obj, candidato, suma);
      }
      else
        escribirSeleccion();
                        /* vuelta atrás, se extrae objeto */
      bs.k--;
      suma -= p.objs[candidato - 1];
   }
}
}
```

En los problemas de tipo selección óptima no se trata de encontrar una situación fija o un valor predeterminado, sino de encontrar la óptima del conjunto de soluciones, según unas condiciones establecidas. También hay que probar con todas las posibilidades para encontrar de entre todas las soluciones la que cumpla una segunda condición, la *solución óptima*.

La selección óptima implica probar con todos los posibles elementos de que se disponga de modo que entre todas las configuraciones que cumplan una primera condición se selecciona la más próxima a una segunda condición que será la selección óptima.

El esquema general del algoritmo que permite encontrar la selección óptima es:

```
subprograma ensayarSeleccion(objeto i)
inicio
  <inicializar cuenta de posibles selecciones>
  repetir
    <tomar siguiente selección (tarea)>
    <determinar si selección es valida>
    si válido entonces
      <anotar selección >
      si < es solución> entonces
        si <mejor(solucion)> entonces
           <guardar selección>
        fin_si
      fin-si
        ensayarSeleccion(objeto i+1)
      <borrar anotación> {el bucle se encarga de probar con otra
                          selección }
    fin_si
  hasta (<no más posibilidades>)
fin
```

5.9.1. Problema del viajante

El ejemplo del problema del viajante es tal vez el que mejor explica la selección óptima:

> *El viajante (piense en un representante de joyería) ha de confeccionar su maleta seleccionando entre* n *artículos, aquellos cuyo valor sea máximo (lo óptimo es que la suma de valores sea máximo) y su peso no exceda de una cantidad, aquella que pueda aguantar (resistir) la maleta.*

Si por ejemplo, la maleta de seguridad para transportar joyas es capaz de almacenar 1,5 kg y los objetos se representan por una pareja de datos `(peso, valor)`: `(115gr, 100€)`, `(90gr, 110€)`, `(50gr, 60€)`, `(120gr, 110€)` ... se pretende hacer una selección, que no sobrepase los 1,5 kg y que el valor asociado sea máximo.

Para resolver el problema se sigue aplicando la estrategia de *vuelta atrás* para generar todas las soluciones posibles, y cada vez que se alcance una solución se guarda si es mejor que las soluciones anteriores según la condición definida.

```
si solucion entonces
  si mejor(solucion) entonces
     optimo = solucion
```

El objetivo del problema es que el valor que se pueda conseguir sea máximo; por esta razón se considera que el valor máximo que se puede alcanzar es la suma de los valores de cada objeto; posteriormente, al excluir un objeto de la selección se ha de restar el valor que tiene.

La tarea básica en esta búsqueda sistemática es investigar si un objeto `i` es adecuado incluirlo en la selección actual (será adecuado si no sobrepasa el peso acumulado al peso de la maleta) y continuar la búsqueda con el siguiente objeto. En el caso de que sea necesario excluir al objeto `i` (*vuelta atrás*) de la selección actual, el criterio para seguir con el proceso de selección es que el valor total todavía alcanzable después de esta exclusión no sea menor que el valor óptimo (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. Cada tarea realiza las mismas acciones que la tarea anterior, por lo que puede expresarse recursivamente.

La búsqueda de la mejor selección se hace probando con todos los objetos de que dispone; una vez que se alcance el último, el objeto `n`, entonces se determina si la selección es óptima.

A tener en cuenta

En el proceso de selección óptima se prueba con la inclusión de un objeto `i` y con la exclusión de `i`. Para hacer más efectivo el algoritmo se hace una *poda* de aquellas selecciones que no van a ser mejores; por ello, antes de probar con la exclusión, se determina si la selección en curso puede alcanzar un mejor valor; en el caso de que no, ¿para qué ir por esa rama si no se va a conseguir una mejor selección?

Representación de los datos

Se supone que el viajante tiene `n` objetos. Cada objeto se representa con el par `<peso, valor>`, que es un registro, una estructura, con dos campos.

La selección actual y la selección óptima se pueden considerar conjuntos de objetos; realmente con conocer el índice de esos objetos es suficiente; por ello se va a utilizar un tipo que permita, fácilmente, realizar la operación de incluir y de excluir elementos de un conjunto.

```
typedef struct
{
  int k;              /* número de elementos */
  int b[M];           /* elementos del conjunto */
} conjunto;
```

El valor máximo que pueden alcanzar los objetos es la suma de los valores de cada uno; está representado por la variable `totalValor`. El valor óptimo alcanzado en el proceso transcurrido está en la variable `mejorValor`.

Los parámetros del procedimiento recursivo son los necesarios para realizar una nueva tarea: `i`, número del objeto a probar su inclusión; `pt`, peso de la selección actual; `va`, valor máximo alcanzable por la selección actual; y `mejorValor`, que es el valor máximo obtenido en el proceso transcurrido.

La entrada de datos es la información asociada con cada objeto. El peso máximo que puede ser transportado también va a ser un dato de entrada.

Codificación

```
#include <stdio.h>
#define M 12
typedef struct
{
  int pso;
  int val;
} objeto;

struct
```

```
{
  int n;
  objeto obj[M];
} a;        /* array de objetos */

typedef struct
{
  int k;
  int b[M];
} conjunto;

conjunto actual, optimo;
int pesoMaximo;
void probarObjeto(int i, int pt, int va, int* mejorValor);
void entradaObjetos();
void escribirOptimo(int mejor);

int main()
{
  int i;
  int maxValor, mejorValor;
  entradaObjetos();
  printf("Peso máximo = ");
  scanf("%d", &pesoMaximo);
                          /* Suma de los valores de cada objeto */
  for (maxValor = i = 0; i < a.n; i++)
    maxValor += a.obj[i].val;
  mejorValor = 0;
  actual.k = optimo.k = 0;
  probarObjeto(1, 0, maxValor, &mejorValor);
  escribirOptimo(mejorValor);
  return 0;
}

void entradaObjetos()
{
  int j;
  do {
    printf("\n\tNúmero de objetos: ");
    scanf("%d", &a.n);
  } while ((a.n < 1) || (a.n > M));
  for (j = 1; j <= a.n; j++)
  {
    printf("\tObjeto %d , <peso,valor> = ", j);
    scanf("%d %d", &a.obj[j-1].pso, &a.obj[j-1].val);
  }
}

void probarObjeto(int i, int pt, int va, int* mejorValor)
{
  int valExclusion;
  if (pt+a.obj[i-1].pso <= pesoMaximo)  /* objeto i se incluye */
```

```
    {
        actual.k++;
        actual.b[actual.k - 1] = i-1;          /* los índices de 0 a n-1 */
        if (i < a.n)
          probarObjeto(i+1, pt+a.obj[i-1].pso, va, mejorValor);
        else                                   /* los n objetos probados */
          if (va > (*mejorValor))
                                               /* es un valor mejor */
          {
            optimo = actual;
            (*mejorValor) = va;
          }
        actual.k-- ;   /* vuelta atrás, ensaya la exclusión de obj i */
    }
                        /* proceso de exclusión del objeto i para seguir
                           la búsqueda sistemática con el objeto i+1 */
    valExclusion = va - a.obj[i-1].val;  /* decrementa el valor del
                                            objeto excluido */
    if (valExclusion > (*mejorValor))    /* se puede alcanzar una mejor
                                         selección, sino poda la búsqueda */
      if (i < a.n)
        probarObjeto(i+1, pt, valExclusion, mejorValor);
      else
      {
        optimo = actual;
        (*mejorValor) = valExclusion;
      }
    }
    void escribirOptimo(int mejor)
    {
        int j;
        puts("\tSelección de objetos que forma la selección óptima");
        for (j = 1; j <= optimo.k; j++)
        {
          printf("\tObjeto %d = <%d,%d> \n", optimo.b[j-1] + 1,
                    a.obj[optimo.b[j-1]].pso, a.obj[optimo.b[j-1]].val);
        }
        printf("\t Valor óptimo = %d \t para un peso máximo = %d",
                                          mejor, pesoMaximo);
    }
```

RESUMEN

Una función se dice que es recursiva si tiene una o más sentencias que son llamadas a sí mismas. La recursividad puede ser directa o indirecta; esta última ocurre cuando la función `f()` llama a `p()`, que a su vez llama a `f()`. La recursividad es una alternativa a la iteración en la resolución de algunos problemas matemáticos. Los aspectos más importantes a tener en cuenta en el diseño y construcción de funciones recursivas son los siguientes:

- Un algoritmo recursivo con su función correspondiente normalmente contiene dos tipos de situaciones: uno o más casos que incluyen al menos una llamada recursiva y uno o más casos de terminación

o parada del problema sin ninguna llamada recursiva, sino con una sentencia simple. De otro modo, una función recursiva debe tener dos partes: una parte de terminación en la que se deja de hacer llamadas, caso base, y una llamada recursiva con sus propios parámetros.

- Muchos problemas tienen naturaleza recursiva y la solución más fácil se consigue mediante una función recursiva. De igual modo aquellos problemas que no entrañen una solución recursiva se deberán seguir resolviendo mediante algoritmos iterativos.
- Las funciones con llamadas recursivas utilizan memoria extra en dichas llamadas; existe un límite en las llamadas, que depende de la memoria de la computadora. En caso de superar este límite ocurre un error de desbordamiento (*overflow*).
- Cuando se codifica una función recursiva se debe comprobar siempre que incluye una condición de terminación; es decir: no se producirá una *recursión infinita*. Durante el aprendizaje de la recursividad es usual que se produzca este error.
- Para asegurarse de que el diseño de una función recursiva es correcto se deben cumplir las siguientes tres condiciones:

  1. No existir recursión infinita. Una llamada recursiva puede conducir a otra llamada recursiva y ésta conducir a otra, y así sucesivamente; pero cada llamada debe aproximarse más a la condición de terminación.
  2. En la condición de terminación, la función devuelve el valor correcto para ese caso.
  3. En los casos que implican llamadas recursivas, si cada una de las funciones devuelve un valor correcto, entonces el valor final devuelto por la función es el valor correcto.

- Una de las técnicas más utilizadas en la resolución de problemas es la denominada *divide y vencerás*. La implementación de estos algoritmos se puede realizar con funciones recursivas.
- Los algoritmos del tipo *vuelta atrás* o *backtracking* se caracterizan por realizar una búsqueda sistemática, exhaustiva de la solución: prueban con todas posibilidades disponibles para realizar una tarea que vaya encaminada hacia la solución. Cada tarea se expresa por una llamada recursiva, los elementos de la tarea se apuntan (se guardan). En la *vuelta atrás*, retorno de la llamada recursiva, se determina si se alcanzó la solución, y en caso contrario, se borra la anotación para probar de nuevo con otra posibilidad.
- La selección óptima se puede considerar como una variante de los algoritmos de vuelta atrás, en la que se busca no sólo los elementos que cumplen una condición, sino que éstos sean los mejores según el criterio que se haya establecido como *mejor*.

EJERCICIOS

5.1. Convertir la siguiente función iterativa en recursiva. La función calcula un valor aproximado de *e* (la base de los logaritmos naturales), sumando las series:

$$1 + 1/1! + 1/2! + \cdots + 1/n!$$

hasta que los términos adicionales no afecten a la aproximación

```
double loge()
{
  double enl, delta, fact;
  int n;
  enl = fact = delta = 1.0;
  n = 1;
  do
  {
    enl += delta;
    n++;
    fact * = n:
```

```
        delta = 1.0 / fact;
      } while (enl != enl + delta);
      return enl;
}
```

5.2. Explicar por qué la siguiente función puede producir un valor incorrecto cuando se ejecute:

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

5.3. ¿Cuál es la secuencia numérica generada por la función recursiva `f()` en el listado siguiente si la llamada es `f(5)`?

```
long f(int n)
{
  if (n == 0 || n == 1)
    return 1;
  else
    return 3 * f(n - 2) + 2 * f(n - 1);
}
```

5.4. Escribir una función recursiva `int vocales(char * cd)` para calcular el número de vocales de una cadena.

5.5. Proporcionar funciones recursivas que representen los siguientes conceptos:

a) El producto de dos números naturales.
b) El conjunto de permutaciones de una lista de números.

5.6. Suponer que la función *G* está definida recursivamente de la siguiente forma:

$$G(x, y) = \begin{cases} \dfrac{1}{G(x - y + 1)} & si \quad x \leqslant y \\ 2x - y & si \quad y < x \end{cases}$$

siendo *x*, *y* enteros positivos. Encontrar el valor de: *a*) $G(8, 6)$; *b*) $G(100, 10)$.

5.7. Escribir una función recursiva que calcule la función de Ackermann definida de la siguiente forma:

$A(m, n) = n + 1$ si $m = 0$
$A(m, n) = A(m - 1, 1)$ si $n = 0$
$A(m, n) = A(m - 1, A(m, n - 1))$ si $m > 0$ y $n > 0$

5.8. ¿Cuál es la secuencia numérica generada por la función recursiva siguiente, si la llamda es `f(8)`?

```
long f  (int n)
{
  if(n == 0||n ==1)
    return 1;
  else if (n % 2 == 0)
```

```
        return 2 + f(n - 1);
      else
        return 3 + f(n - 2);
    }
```

5.9. ¿Cuál es la secuencia numérica generada por la función recursiva siguiente?

```
int f(int n)
{
    if (n == 0)
      return 1;
    else if (n == 1)
      return 2;
    else
      return 2*f(n - 2) + f(n - 1);
}
```

5.10. El elemento mayor de un array entero de n elementos se puede calcular recursivamente. Suponiendo que la función:

```
int max(int x, int y);
```

devuelve el mayor de dos enteros x e y, definir la función

```
int maxarray(int a[], int n);
```

que utiliza recursión para devolver el elemento mayor de `a`

Condición de parada: `n == 1`

Incremento recursivo: `maxarray = max(max(a[0]...a[n-2]),a[n-1])`

5.11. Escribir una función recursiva,

```
int product(int v[], int b);
```

que calcule el producto de los elementos del array `v` mayores que `b`.

5.12. El Ejercicio 5.3 tiene una función recursiva; escribir de nuevo la función eliminando la recursividad.

5.13. La resolución recursiva de las Torres de Hanoi ha sido realizada con dos llamadas recursivas. Volver a escribir la solución con una sola llamada recursiva.

Nota: Sustituir la última llamada por un bucle `repetir-hasta`.

5.14. Aplicar el esquema de los algoritmos *divide y vencerás* para que dadas las coordenadas (x, y) de dos puntos en el plano, que representan los extremos de un segmento, se dibuje el segmento.

5.15. La suma de los primeros n números enteros responde a la fórmula:

$$1 + 2 + 3 + \cdots + n = n(n + 1)/2$$

Escribir una función para inicializar el array `v` que contenga los primeros 50 enteros. Sabiendo que la media de estos elementos del array es entonces igual a 25,5, comprobar la solución aplicando la función recursiva `media (int v[], int n)`.

PROBLEMAS

5.1. La expresión matemática $C(m, n)$ en el mundo de la teoría combinatoria de los números, representa el número de combinaciones de m elementos tomados de n en n elementos.

$$C(m, n) = \frac{m!}{n!(m - n)!}$$

Escribir un programa cuyas entradas sean los enteros m, n y calcular $C(m, n)$ donde $n!$ es el factorial de n.

5.2. Un palíndromo es una palabra que se lee exactamente igual 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 vencerás* para escribir una función recursiva que devuelva 1, si una palabra pasada como argumento es un palíndromo y devuelva 0 en caso contrario. Escribir un programa en el que se lea una cadena hasta que ésta sea un palíndromo.

5.3. Escribir un programa y una función recursiva que liste todos los subconjuntos de `n` letras para un conjunto dado de `m` letras; por ejemplo, si $m = 4$ y $n = 2$

```
[A, C, E, K] → [A, C], [A, E], [A, K], [C, E], [C, K], [E, K]
```

Nota: El número de subconjuntos es $C_{4,2}$.

5.4. El problema de las ocho reinas que se ha resuelto en el Apartado 5.7.2, obtiene una solución. Volver a escribir el programa para obtener todas las soluciones posibles al problema.

5.5. 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 cual es el entero cuya suma de dígitos es mayor. La suma de dígitos se ha de realizar con una función recursiva.

5.6. Desarrollar una función recursiva 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 o 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?)

5.7. Leer un número entero positivo $n < 10$. Calcular el desarrollo del polinomio $(x + 1)^n$. Imprimir cada potencia x^i en la forma `x**i`.

Sugerencia:

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

donde $C_{n,n}$ y $C_{n,0}$ toman el valor 1 para cualquier valor de n. La relación de recurrencia de los coeficientes binomiales es:

$C(n, 0) = 1$

$C(n, n) = 1$

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

Estos coeficientes constituyen el conocido triángulo de Pascal y será preciso definir el método que genera el triángulo:

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

5.8. Sea A una matriz cuadrada de $n \times n$ elementos. El determinante de A se puede definir de manera recursiva:

a) Si $n = 1$ entonces $\text{Deter}(A) = a_{1,1}$.
b) Para $n > 1$, el determinante es la suma alternada de productos de los elementos de una fila o columna elegida al azar por sus menores complementarios. A su vez, los menores complementarios son los determinantes de orden $n - 1$ obtenidos al suprimir la fila y columna en que se encuentra el elemento.

La expresión matemática es:

$$\text{Det}(A) = \sum_{i=1}^{n} (-1)^{i+j} * A[i,j] * \text{Det}(\text{Menor}(A[i,j]));$$ para cualquier columna j

o bien

$$\text{Det}(A) = \sum_{j=1}^{n} (-1)^{i+j} * A[i,j] * \text{Det}(\text{Menor}(A[i,j]));$$ para cualquier fila i

Se observa que la resolución del problema sigue la estrategia de los algoritmos *divide y vencerás*. Escribir un programa cuya entrada sean los elementos de la matriz A, y cuya salida sea la matriz A y el determinante de A. (*Nota:* Elegir la fila 1 para calcular el determinante.)

5.9. Escribir un programa que transforme números enteros en base 10 a otra en base *b*, siendo la base *b* de 8 a 16. La transformación se ha de realizar siguiendo una estrategia recursiva.

5.10. Escribir un programa para resolver el problema de la subsecuencia creciente más larga. La entrada es una secuencia de *n* números $a_1, a_2, a_3, \ldots, a_n$; hay que encontrar la subsecuencia más larga $a_{i1}, a_{i2}, \ldots, a_{ik}$, tal que $a_{i1} < a_{i2} < a_{i3} \cdots < a_{ik}$ y que $i1 < i2 < i3 < \cdots < ik$. El programa escribirá tal subsecuencia.

Por ejemplo, si la entrada es 3, 2, 7, 4, 5, 9, 6, 8, 1, la subsecuencia creciente más larga tiene longitud cinco: 2, 4, 5, 6, 8.

5.11. El sistema monetario consta de monedas de valor $p_1, p_2, p_3, \ldots, p_n$ (orden creciente) euros. Escribir un programa que tenga como entrada el valor de las n monedas en euros, en orden creciente, y una cantidad x euros de cambio. Calcule:

a) El número mínimo de monedas que se necesitan para dar el cambio x.
b) Calcule el número de formas diferentes de dar el cambio de x euros con la p_i monedas.

Aplicar técnicas recursivas para resolver el problema.

5.12. En un tablero de ajedrez, se coloca un alfil en la posición (x_0, y_0) y un peón en la posición (1, j), siendo $1 <= j <= 8$. Se pretende encontrar una ruta para el peón que llegue a la fila 8 sin ser comido por el alfil. *Nota:* el único movimiento permitido para el peón es el 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 problema planteado. (*Nota:* tener en cuenta que el alfil ataca por diagonales.)

5.13. Dados n números enteros positivos, encontrar combinaciones de ellos que mediante sumas o restas totalicen exactamente un valor objetivo Z. El programa debe de tener como entrada los n números y el objetivo Z; la salida ha de ser la combinación de números con el operador que le corresponde. Tener en cuenta que pueden formar parte de la combinación los n números o parte de ellos.

5.14. Dados `n` números, encontrar la combinación con sumas o restas que más se aproxime a un objetivo `Z`. La aproximación puede ser por defecto o por exceso. La entrada son los `n` números y el objetivo y la salida la combinación más proxima al objetivo.

5.15. Un laberinto se puede emular con una matriz `n × n` en la que los pasos libres están representados por un carácter (el blanco, por ejemplo) y los muros por otro carácter (el •, por ejemplo). Escribir un programa en el que se genere aleatoriamente un laberinto, se pida las coordenadas de entrada (la fila será la 1), las coordenadas de salida (la fila será la `n`) y encontrar todas las rutas que nos llevan de la entrada a la salida.

5.16. Realizar las modificaciones necesarias en el Problema 5.15 para encontrar la ruta más corta, considerando ruta más corta la que pasa por un menor número de casillas.

5.17. Una región castellana está formada por *n* pueblos dispersos. Hay conexiones directas entre algunos de estos pueblos y entre otros no existe conexión aunque puede haber un camino. Escribir un programa que tenga como entrada la matriz que representa las conexiones directas entre pueblos, de tal forma que el elemento `M(i,j)` de la matriz sea:

$$M(i, j) = \begin{cases} 0 & \text{si no hay conexión directa entre pueblo } i \text{ y pueblo } j \\ d & \text{hay conexión entre pueblo } i \text{ y pueblo } j \text{ de distancia } d \end{cases}$$

También se debe considerar como entrada un par de pueblos `(x,y)` y se ha de encontrar un camino entre ambos pueblos utilizando técnicas recursivas. La salida ha de ser la ruta que se ha de seguir para ir de `x` a `y` junto a la distancia de la ruta.

5.18. Hacer las modificaciones necesarias en el Problema 5.17 para encontrar todos los caminos posibles entre el par de pueblos `(x,y)`.

6

ALGORITMOS DE ORDENACIÓN Y BÚSQUEDA

OBJETIVOS

Después del estudio de este capítulo usted podrá:

- Conocer los algoritmos basados en el intercambio de elementos.
- Conocer el algoritmo de ordenación por inserción.
- Conocer el algoritmo de selección.
- Distinguir entre los algoritmos de ordenación basados en el intercambio y en la inserción.
- Deducir la eficiencia de los métodos básicos de ordenación.
- Conocer los métodos más eficientes de ordenación.
- Aplicar métodos mas eficientes de ordenación de arrays (arreglos).
- Diferenciar entre búsqueda secuencial y búsqueda binaria.

CONTENIDO

6.1. Ordenación.
6.2. Algoritmos de ordenación básicos.
6.3. Ordenación por intercambio.
6.4. Ordenación por selección.
6.5. Ordenación por inserción.
6.6. Ordenación por burbuja.
6.7. Ordenación Shell.
6.8. Ordenación rápida (*quicksort*).
6.9. Ordenación Binsort y Radixsort.
6.10. Búsqueda en listas: búsqueda secuencial y binaria.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Ordenación numérica.
- Ordenación alfabética.
- Complejidad cuadrática.
- Ordenación por burbuja.
- Ordenación rápida.
- Residuos.
- Ordenación por intercambio.
- Ordenación por inserción.
- Búsqueda en listas: búsqueda secuencial y búsqueda binaria.
- Complejidad logarítmica.
- Ordenación por selección.

INTRODUCCIÓN

Muchas actividades humanas requieren que en ellas las diferentes colecciones de elementos utilizados se coloquen en un orden específico. Las oficinas de correo y las empresas de mensajería ordenan el correo y los paquetes por códigos postales con el objeto de conseguir una entrega eficiente; los anuarios o listines telefónicos ordenan sus clientes por orden alfabético de apellidos con el fin último de encontrar fácilmente el número de teléfono deseado; los estudiantes de

una clase en la universidad se ordenan por sus apellidos o por los números de expediente, etc. Por esta circunstancia una de las tareas que realizan más frecuentemente las computadoras en el procesamiento de datos es la *ordenación*.

El estudio de diferentes métodos de ordenación es una tarea intrínsecamente interesante desde un punto de vista teórico y, naturalmente, práctico. El capítulo estudia los algoritmos y técnicas de ordenación más usuales y su implementación en C. De igual modo se estudiará el análisis de los algoritmos utilizados en diferentes métodos de ordenación con el objetivo de conseguir la máxima eficiencia en su uso real. En el capítulo se analizarán los métodos básicos y avanzados más empleados en programas profesionales.

6.1. ORDENACIÓN

La **ordenación** o **clasificación** de datos (*sort*, en inglés) es una operación consistente en disponer un conjunto —estructura— de datos en algún determinado orden con respecto a uno de los *campos* de elementos del conjunto. Por ejemplo, cada elemento del conjunto de datos de una guía telefónica tiene un campo nombre, un campo dirección y un campo número de teléfono; la guía telefónica está dispuesta en orden alfabético de nombres; los elementos numéricos se pueden ordenar en orden creciente o decreciente de acuerdo al valor numérico del elemento. En terminología de ordenación, el elemento por el cual está ordenado un conjunto de datos (o se está buscando) se denomina *clave*.

Una colección de datos (*estructura*) puede ser almacenada en un *archivo*, un *array* (*vector o tabla*), *un array de registros, una lista enlazada* o un *árbol*. Cuando los datos están almacenados en un array, una lista enlazada o un árbol, se denomina *ordenación interna*. Si los datos están almacenados en un archivo, el proceso de ordenación se llama *ordenación externa*.

Una *lista* se dice que *está ordenada por la clave k* si la lista está en orden ascendente o descendente con respecto a esta clave. La lista se dice que está en *orden ascendente* si:

`i < j` *implica que* `k[i] <= k[j]`

y se dice que está en *orden descendente* si:

`i > j` *implica que* `k[i] <= k[j]`

para todos los elementos de la lista. Por ejemplo, para una guía telefónica, la lista está clasificada en orden ascendente por el campo clave *k*, donde $k[i]$ es el nombre del abonado (apellidos, nombre).

```
4   5  14  21  32  45                    orden ascendente
75  70 35  16  14  12                    orden descendente
Zacarias Rodriguez Martinez Lopez Garcia  orden descendente
```

Los métodos (algoritmos) de ordenación son numerosos, por ello se debe prestar especial atención en su elección. ¿Cómo se sabe cuál es el mejor algoritmo? La *eficiencia* es el factor que mide la calidad y rendimiento de un algoritmo. En el caso de la operación de ordenación, dos criterios se suelen seguir a la hora de decidir qué algoritmo —de entre los que resuelven la ordenación— es el más eficiente: 1) *tiempo menor de ejecución en computadora*; 2) *menor número de instrucciones*. Sin embargo, no siempre es fácil efectuar estas medidas: puede no disponerse de instrucciones para medida de tiempo —aunque no sea éste el caso del lenguaje C—, y las instrucciones pueden variar, dependiendo del lenguaje y del propio estilo del programador. Por esta razón, el mejor criterio para medir la eficiencia de un algoritmo es aislar una operación específica clave en la ordenación y contar el número de veces que se realiza. Así, en el caso de los algoritmos de ordenación, se utilizará como medida de su eficiencia el número de comparaciones entre elementos efectuados. El algoritmo de *ordenación* A será más eficiente que el B, si requiere menor número de comparaciones.

Así, en el caso de ordenar los elementos de un vector, el número de comparaciones será *función* del número de elementos (`n`) del vector (array). Por consiguiente, se puede expresar el número de comparaciones en términos de `n` (por ejemplo, `n + 4`, o bien n^2 en lugar de números enteros (por ejemplo, `325`).

En *todos los métodos de este capítulo*, normalmente —para comodidad del lector— *se utiliza el orden ascendente* sobre vectores o listas (arrays unidimensionales).

Los métodos de ordenación se suelen dividir en dos grandes grupos:

- **directos** *burbuja, selección, inserción*
- **indirectos** *(avanzados)* *Shell, ordenación rápida, ordenación por mezcla, Radixsort*

En el caso de listas pequeñas, los métodos directos se muestran eficientes, sobre todo porque los algoritmos son sencillos; su uso es muy frecuente. Sin embargo, en listas grandes estos métodos se muestran ineficaces y es preciso recurrir a los métodos avanzados.

6.2. ALGORITMOS DE ORDENACIÓN BÁSICOS

Existen diferentes algoritmos de ordenación elementales o básicos cuyos detalles de implementación se pueden encontrar en diferentes libros de algoritmos. La enciclopedia de referencia es [KNUTH 1973][1] y sobre todo la 2.ª edición publicada en el año 1998 [KNUTH 1998][2]. Los algoritmos presentan diferencias entre ellos que los convierten en más o menos eficientes y prácticos según sea la rapidez y eficiencia demostrada por cada uno de ellos. Los algoritmos básicos de ordenación más simples y clásicos son:

- Ordenación por selección.
- Ordenación por inserción.
- Ordenación por burbuja.

Los métodos más recomendados son: *selección* e *inserción*, aunque se estudiará el método de *burbuja*, por aquello de ser el más sencillo aunque a la par también es el más *ineficiente*; por esta causa no recomendamos su uso, pero sí conocer su técnica.

Los datos se pueden almacenar en memoria central o en archivos de datos externos guardados en unidades de almacenamiento magnético (discos, cintas, disquetes, CD-ROM, DVD, discos *flash* USB, etc.) Cuando los datos se guardan en listas y en pequeñas cantidades, se suelen almacenar de modo temporal en arrays y registros; estos datos se almacenan exclusivamente para tratamientos internos que se utilizan en gestión masiva de datos y se guardan en arrays de una o varias dimensiones. Los datos, sin embargo, se almacenan de modo permanente en archivos y bases de datos que se guardan en discos y cintas magnéticas.

A tener en cuenta

Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*. Los métodos de ordenación se conocen como *internos* o *externos* según que los elementos a ordenar estén en la memoria principal o en la memoria externa.

[1] [KNUTH 1973] Donald E. Knuth. *The Art of Computer Programming*. Volume 3: *Sorting and Searching*. Addison-Wesley, 1973.

[2] [KNUTH 1998] Donald E. Knuth. *The Art of Computer Programming*. Volume 3: *Sorting and Searching*. Second Edition. Addison-Wesley, 1998.

Así pues, existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*. Los métodos de ordenación se conocen como *internos* o *externos* según que los elementos a ordenar estén en la memoria principal o en la memoria externa.

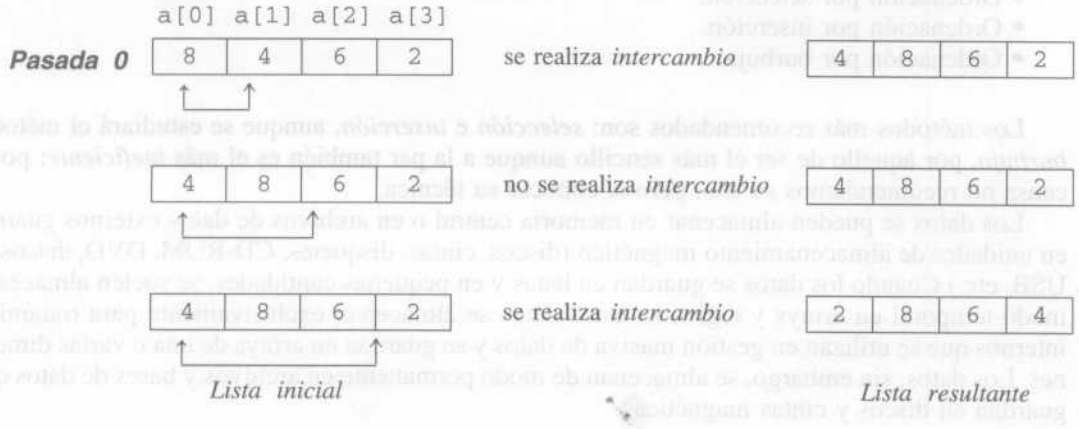
Las técnicas que se analizarán a continuación considerarán, esencialmente, la ordenación de elementos de una lista (*array*) en orden ascendente. En cada caso se desarrollará la eficiencia computacional del algoritmo.

Con el objeto de facilitar el aprendizaje del lector y aunque no sea un método utilizado por su poca eficiencia se describirá en primer lugar el método de ordenación por intercambio con un programa completo que manipula a la correspondiente función `ordIntercambio()`, por la sencillez de su técnica y con el objetivo de que el lector no introducido en algoritmos de ordenación pueda comprender su funcionamiento y luego asimilar más eficazmente los algoritmos básicos ya citados y los avanzados que se estudiarán más adelante.

6.3. ORDENACIÓN POR INTERCAMBIO

El **algoritmo de ordenación** tal vez más sencillo sea el denominado de *intercambio* que ordena los elementos de una lista en orden ascendente. Este algoritmo se basa en la lectura sucesiva de la lista a ordenar, comparando el elemento inferior de la lista con los restantes y efectuando intercambio de posiciones cuando el orden resultante de la comparación no sea el correcto.

El algoritmo se ilustra con la lista original `8, 4, 6, 2` que ha de convertirse en la lista ordenada `2, 4, 6, 8`. El algoritmo realiza $n - 1$ pasadas (3 en el ejemplo), siendo n el número de elementos, y ejecuta las siguientes operaciones.

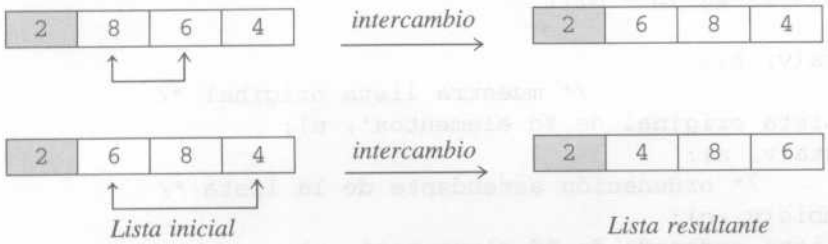


El elemento de índice 0 (`a[0]`) se compara con cada elemento posterior de la lista de índices 1, 2 y 3. En cada comparación se comprueba si el elemento siguiente es más pequeño que el elemento de índice 0, en ese caso se intercambian. Después de terminar todas las comparaciones, el elemento más pequeño se localiza en el índice 0.

Pasada 1

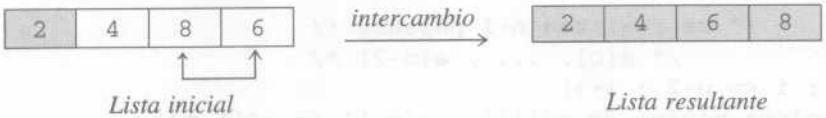
El elemento más pequeño ya está localizado en el índice 0, y se considera la sublista restante `8, 6, 4`. El algoritmo continúa comparando el elemento de índice 1 con los elementos posteriores de índices 2 y 3. Por cada comparación, si el elemento mayor está en el índice 1 se intercambian

los elementos. Después de hacer todas las comparaciones, el segundo elemento más pequeño de la lista se almacena en el índice 1.



Pasada 2

La sublista a considerar ahora es `8, 6` ya que `2, 4` está ordenada. Una comparación única se produce entre los dos elementos de la sublista



El método `ordIntercambio` utiliza dos bucles anidados. Suponiendo que la lista es de tamaño n, el rango del bucle externo irá desde el índice 0 hasta $n - 2$. Por cada índice i, se comparan los elementos posteriores de índices $j = i + 1, i + 2, ..., n - 1$. El intercambio (*swap*) de los elementos `a[i],a[j]` se realiza en un bloque que utiliza el algoritmo siguiente:

```
aux = a[i];
a[i] = a[j];
a[j]= aux ;
```

Ejercicio 6.1

El programa siguiente ordena una lista de *n* elementos de tipo entero introducida en un array y posteriormente la imprime (o visualiza) en pantalla.

```
#include <stdio.h>
#define N 100

void ordIntercambio (int a[], int n);
void entradaLista (int a[], int n);
void imprimirLista (int a[], int n);

int main()
{
  int n;
  int v[N];
```

```
    do {
      printf("\nIntroduzca el número de elementos: ");
      scanf("%d", &n);
    } while ((n < 1) && (n > N));

    entradaLista(v, n);
                                  /* muestra lista original */
    printf("\nLista original de %d elementos", n);
    imprimirLista(v, n);
                        /* ordenación ascendente de la lista */
    ordIntercambio(v, n);
    printf("\nLista ordenada de %d elementos", n);
    imprimirLista(v, n);
    return 0;
  }

  void ordIntercambio (int a[], int n)
  {
    int i, j;

                        /* se realizan n-1 pasadas */
                              /* a[o], ... , a[n-2] */
    for (i = 0 ; i <= n-2 ; i++)
          /* coloca mínimo de a[i+1]...a[n-1] en a[i] */
      for (j = i+1 ; j <= n-1 ; j++)
        if (a[i] > a[j])
        {
          int aux;
          aux = a[i];
          a[i] = a[j];
          a[j]= aux ;
        }
  }

  void imprimirLista (int a[], int n)
  {
    int i;

    for (i = 0 ; i < n ; i++)
    {
      char c;
      c = (i%10==0)?'\n':' ';
      printf("%c%d", c, a[i]);
    }
  }

  void entradaLista (int a[], int n)
  {
    int i;

    printf("\n Entrada de los elementos\n");
    for (i = 0 ; i < n ; i++)
```

```
    {
      printf("a[%d] = ",);
      scanf("%d", a+i);
    }
}
```

6.4. ORDENACIÓN POR SELECCIÓN

Considérese el algoritmo para ordenar un array `A` de enteros en orden ascendente, es decir, del número más pequeño al mayor. Es decir, si el array `A` tiene *n* elementos, se trata de ordenar los valores del array de modo que el dato contenido en `A[0]` sea el valor más pequeño, el valor almacenado en `A[1]` el siguiente más pequeño, y así hasta `A[n-1]`, que ha de contener el elemento de mayor valor. El algoritmo se apoya en sucesivas pasadas que intercambian el elemento más pequeño sucesivamente con el primer elemento de la lista, `A[0]` en la primera pasada. En síntesis, se busca el elemento más pequeño de la lista y se intercambia con `A[0]`, primer elemento de la lista.

```
A[0]  A[1]  A[2]....  A[n-1]
```

Después de terminar esta primera pasada, el frente de la lista está ordenado y el resto de la lista `A[1], A[2]...A[n-1]` permanece desordenado. La siguiente pasada busca en esta lista desordenada y *selecciona* el elemento más pequeño, que se almacena entonces en la posición `A[1]`. De este modo los elementos `A[0]` y `A[1]` están ordenados y la sublista `A[2], A[3]...A[n-1]` desordenada; entonces, se selecciona el elemento más pequeño y se intercambia con `A[2]`. El proceso continúa $n - 1$ pasadas y en ese momento la lista desordenada se reduce a un elemento (el mayor de la lista) y el array completo ha quedado ordenado.

Un ejemplo práctico ayudará a la comprensión del algoritmo. Consideremos un array `A` con 5 valores enteros `51, 21, 39, 80, 36`:

A[0]	A[1]	A[2]	A[3]	A[4]
51	21	39	80	36

↓ *pasada 0*

Pasada 0. Seleccionar `21`
Intercambiar `21` y `A[0]`

21	51	39	80	36

↓ *pasada 1*

Pasada 1. Seleccionar `36`
Intercambiar `36` y `A[1]`

21	36	39	80	51

↓ *pasada 2*

Pasada 2. Seleccionar `39`
Intercambiar `39` y `A[2]`

21	36	39	80	51

↓ *pasada 3*

Pasada 3. Seleccionar `51`
Intercambiar `51` y `A[3]`

21	36	39	51	80

Lista ordenada

6.4.1. Algoritmo de selección

Los pasos del algoritmo son:

1. Seleccionar el elemento más pequeño de la lista `A`; intercambiarlo con el primer elemento `A[0]`. Ahora la entrada más pequeña está en la primera posición del vector.
2. Considerar las posiciones de la lista `A[1], A[2], A[3]...`, seleccionar el elemento más pequeño e intercambiarlo con `A[1]`. Ahora las dos primeras entradas de `A` están en orden.
3. Continuar este proceso encontrando o seleccionando el elemento más pequeño de los restantes elementos de la lista, intercambiándolos adecuadamente.

6.4.2. Codificación en C del algoritmo de selección

La función `ordSeleccion()` ordena una lista o vector de números reales de `n` elementos. En la pasada `i`, el proceso de selección explora la sublista `A[i]` a `A[n-1]` y fija el índice del elemento más pequeño. Después de terminar la exploración, los elementos `A[i]` y `A[indiceMenor]` intercambian las posiciones.

```
/*
   ordenar un array de n elementos de tipo double
   utilizando el algoritmo de ordenación por selección
*/
void ordSeleccion (double a[], int n)
{
  int indiceMenor, i, j;
            /* ordenar a[0]..a[n-2] y a[n-1] en cada pasada */
  for (i = 0; i < n-1; i++)
  {
            /* comienzo de la exploración en índice i */
    indiceMenor = i;
            /* j explora la sublista a[i+1]..a[n-1] */
    for (j = i+1; j < n; j++)
      if (a[j] < a[indiceMenor])
        indiceMenor = j;
            /* sitúa el elemento más pequeño en a[i] */
    if (i != indiceMenor)
    {
      double aux = a[i];
      a[i] = a[indiceMenor];
      a[indiceMenor] = aux ;
    }
  }
}
```

El análisis del algoritmo de selección es sencillo y claro, ya que requiere un número fijo de comparaciones que sólo dependen del tamaño de la lista o vector (array) y no de la distribución inicial de los datos. En el Apartado 2.6.1 se realizó un estudio de la complejidad de este algoritmo.

6.5. ORDENACIÓN POR INSERCIÓN

El método de ordenación por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, que consiste en insertar un nombre en su posición correcta dentro de una lista o archivo que ya está ordenado. Así el proceso en el caso de la lista de enteros `A = 50, 20, 40, 80, 30`.

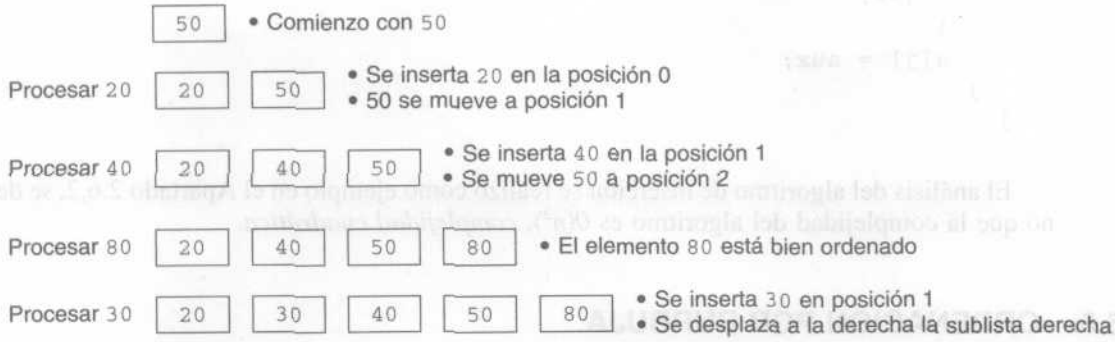


Figura 6.1. Método de ordenación por inserción.

6.5.1. Algoritmo de ordenación por inserción

El algoritmo correspondiente a la ordenación por inserción contempla los siguientes pasos:

1. El primer elemento `A[0]` se considera ordenado; es decir, la lista inicial consta de un elemento.
2. Se inserta `A[1]` en la posición correcta, delante o detrás de `A[0]`, dependiendo de que sea menor o mayor.
3. Por cada bucle o iteración `i (desde i=1 hasta n-1)` se explora la sublista `A[i-1] .. A[0]` buscando la posición correcta de inserción; a la vez se mueve hacia abajo (a la derecha en la sublista) una posición todos los elementos mayores que el elemento a insertar `A[i]`, para dejar vacía esa posición.
4. Insertar el elemento a la posición correcta.

6.5.2. Codificación en C del algoritmo de ordenación por inserción

La función `ordInsercion()` tiene dos argumentos, el array `a[]` que se va a ordenar crecientemente, y el número de elementos `n`. En la codificación se supone que los elementos son de tipo entero.

```
void ordInsercion (int [] a, int n)
{
  int i, j;
  int aux;

  for (i = 1; i < n; i++)
  {
    /* índice j explora la sublista a[i-1]..a[0] buscando la
       posición correcta del elemento destino, lo asigna a a[j] */
```

```
        j = i;
        aux = a[i];
        /* se localiza el punto de inserción explorando hacia abajo */
        while (j > 0 && aux < a[j-1])
        {
            /* desplazar elementos hacia arriba para hacer espacio */
          a[j] = a[j-1];
          j--;
        }
        a[j] = aux;
    }
}
```

El análisis del algoritmo de inserción se realizó como ejemplo en el Apartado 2.6.2, se determinó que la complejidad del algoritmo es $O(n^2)$, *complejidad cuadrática*.

6.6. ORDENACIÓN POR BURBUJA

El método de *ordenación por burbuja* es el más conocido y popular entre estudiantes y aprendices de programación, por su facilidad de comprensión y programación; por el contrario, es el menos eficiente y por ello, normalmente, se aprende su técnica pero no suele utilizarse.

La técnica utilizada se denomina ***ordenaciá por burbuja*** u ***ordenaciá por hundimiento*** debido a que los valores más pequeños «burbujean» gradualmente (suben) hacia la cima o parte superior del array de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del array. La técnica consiste en hacer varias pasadas a través del array. En cada pasada, se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente (o los valores son idénticos), se dejan los valores como están. Si una pareja está en orden decreciente, sus valores se intercambian en el array.

6.6.1. Algoritmo de la burbuja

En el caso de un array (lista) con n elementos, la ordenación por burbuja requiere hasta $n - 1$ pasadas. Por cada pasada se comparan elementos adyacentes y se intercambian sus valores cuando el primer elemento es mayor que el segundo elemento. Al final de cada pasada, el elemento mayor ha «burbujeado» hasta la cima de la sublista actual. Por ejemplo, después que la pasada 0 está completa, la cola de la lista $A[n - 1]$ está ordenada y el frente de la lista permanece desordenado. Las etapas del algoritmo son:

- En la pasada 0 se comparan elementos adyacentes:

  `(A[0],A[1]),(A[1],A[2]),(A[2],A[3]),...(A[n-2],A[n-1])`

  Se realizan $n - 1$ comparaciones, por cada pareja `(A[i],A[i+1])` se intercambian los valores si `A[i+1] < A[i]`. Al final de la pasada, el elemento mayor de la lista está situado en `A[n-1]`.
- En la pasada 1 se realizan las mismas comparaciones e intercambios, terminando con el elemento segundo mayor valor en `A[n-2]`.
- El proceso termina con la pasada $n - 1$, en la que el elemento más pequeño se almacena en `A[0]`.

El algoritmo tiene una mejora inmediata, el proceso de ordenación puede terminar en la pasada $n - 1$, o bien antes, si en una pasada no se produce intercambio alguno entre elementos del vector es porque ya está ordenado, entonces no es necesario más pasadas.

El ejemplo siguiente ilustra el funcionamiento del algoritmo de la burbuja con un array de 5 elementos (`A = 50, 20, 40, 80, 30`), donde se introduce una variable `interruptor` para detectar si se ha producido intercambio en la pasada.

Pasada 0

50	20	40	80	30

Intercambio `50` y `20`

20	50	40	80	30

Intercambio `50` y `40`

20	40	50	80	30

`50` y `80` ordenados

20	40	50	80	30

Intercambio `80` y `30`

20	40	50	30	80

- Elemento mayor es `80`
- `interruptor = TRUE`

En la pasada 1:

20	40	50	30	80

`20` y `40` ordenados

20	40	50	30	80

`40` y `50` ordenados

20	40	50	30	80

Se intercambian `50` y `30`

20	40	30	50	80

- `50` y `80` elementos mayores y ordenados
- `interruptor = TRUE`

En la pasada 2, sólo se hacen dos comparaciones:

20	40	30	50	80

`20` y `40` ordenados

20	30	40	50	80

- Se intercambian `40` y `30`
- `interruptor = TRUE`

En la pasada 3, se hace una única comparación de `20` y `30`, y no se produce intercambio:

20	30	40	50	80

20 y 30 ordenados

20	30	40	50	80

- Lista ordenada
- `interruptor = FALSE`

En consecuencia, el algoritmo de ordenación de burbuja mejorado contempla dos bucles anidados: el *bucle externo* controla la cantidad de pasadas (al principio de la primera pasada todavía no se ha producido ningún intercambio, por tanto la variable `interruptor` se pone a *valor falso* (0); el *bucle interno* controla cada pasada individualmente y cuando se produce un intercambio, cambia el valor de `interruptor` a *verdadero* (1).

El algoritmo terminará, bien cuando se termine la última pasada ($n - 1$) o bien cuando el valor del `interruptor` sea falso (0), es decir, no se haya hecho ningún intercambio. La condición para realizar una nueva pasada se define en la expresión lógica

```
(pasada < n-1) && interruptor
```

6.6.2. Codificación en C del algoritmo de la burbuja

La función `ordBurbuja()` implementa el algoritmo de ordenación de la burbuja; tiene dos argumentos, el array que se va a ordenar crecientemente, y el número de elementos `n`. En la codificación se supone que los elementos son de tipo entero largo.

```
void ordBurbuja (long a[], int n)
{
  int interruptor = 1;
  int pasada, j;

  for (pasada = 0; pasada < n-1 && interruptor; pasada++)
  {
      /* bucle externo controla la cantidad de pasadas */
    interruptor = 0;
    for (j = 0; j < n-pasada-1; j++)
      if (a[j] > a[j+1])
      {
          /* elementos desordenados, es necesario intercambio */
        long aux;
        interruptor = 1;
        aux = a[j];
        a[j] = a[j+1];
        a[j+1] = aux;
      }
  }
}
```

Una modificación al algoritmo anterior puede ser utilizar, en lugar de una variable bandera `interruptor`, una variable `indiceIntercambio` que se inicie a 0 (cero) al principio de cada

pasada y se establezca al índice del último intercambio, de modo que cuando al terminar la pasada el valor de `indiceIntercambio` siga siendo 0 implicará que no se ha producido ningún intercambio (o bien, que el intercambio ha sido con el primer elemento), y, por consiguiente, la lista estará ordenada. En caso de no ser 0, el valor de `indiceIntercambio` representa el índice del vector a partir del cual los elementos están ordenados. La codificación en C de esta alternativa es:

```
/*
  Ordenación por burbuja : array de n elementos
  Se realizan una serie de pasadas mientras indiceIntercambio > 0
*/

void ordBurbuja2 (long a[], int n)
{
  int i, j;
  int indiceIntercambio;
                    /* i es el índice del último elemento de la sublista */
  i = n-1;
                    /* el proceso continúa hasta que no haya intercambios */
  while (i > 0)
  {
    indiceIntercambio = 0;
                    /* explorar la sublista a[0] a a[i] */
    for (j = 0; j < i; j++)
                    /* intercambiar pareja y actualizar indiceIntercambio */
      if (a[j+1] < a[j])
      {
        long aux=a[j];
        a[j] = a[j+1];
        a[j+1] = aux;
        indiceIntercambio = j;
      }
                    /* i se pone al valor del índice del último intercambio */
    i = indiceIntercambio;
  }
}
```

6.6.3. Análisis del algoritmo de la burbuja

¿Cuál es la eficiencia del algoritmo de ordenación de la burbuja? Dependerá de la versión utilizada. En la versión más simple se hacen $n - 1$ pasadas y $n - 1$ comparaciones en cada pasada. Por consiguiente, el número de comparaciones es $(n - 1) * (n - 1) = n^2 - 2n + 1$, es decir, la complejidad es $O(n^2)$.

Si se tienen en cuenta las versiones mejoradas haciendo uso de las variables `interruptor` o `indiceIntercambio`, entonces se tendrá una eficiencia diferente a cada algoritmo. En el mejor de los casos, la ordenación de burbuja hace una sola pasada en el caso de una lista que ya está ordenada en orden ascendente y por tanto su complejidad es $O(n)$. En el caso peor se requieren $(n - i - 1)$ comparaciones y $(n - i - 1)$ intercambios. La ordenación completa requiere $\frac{n(n-1)}{2}$ comparaciones

y un número similar de intercambios. La complejidad para el caso peor es $O(n^2)$ *comparaciones* y $O(n^2)$ *intercambios*. De cualquier forma, el análisis del caso general es complicado dado que alguna de las pasadas pueden no realizarse. Se podría señalar, entonces, que el número medio de pasadas k sea $O(n)$ y el número total de comparaciones es $O(n^2)$. En el mejor de los casos, la ordenación por burbuja puede terminar en menos de $n - 1$ pasadas pero requiere, normalmente, muchos más intercambios que la ordenación por selección y su prestación media es mucho más lenta, sobre todo cuando los arrays a ordenar son grandes.

6.7. ORDENACIÓN SHELL

La **ordenación Shell** debe el nombre a su inventor, D. L. Shell. Se suele denominar también *ordenación por inserción con incrementos decrecientes*. Se considera que el método Shell es una mejora de los métodos de inserción directa.

En el álgoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el más pequeño hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo.

El algoritmo de Shell modifica los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño y con ello se consigue que la ordenación sea más rápida. Generalmente se toma como salto inicial $n/2$ (siendo n el número de elementos), luego se reduce el salto a la mitad en cada repetición hasta que el salto es de tamaño 1. El Ejemplo 6.1 ordena una lista de elementos siguiendo paso a paso el método de Shell.

Ejemplo 6.1

Obtener las secuencias parciales del vector al aplicar el método Shell para ordenar en orden creciente la lista:

6 1 5 2 3 4 0

El número de elementos que tiene la lista es 6, por lo que el salto inicial es 6/2 = 3. La siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondiente.

Recorrido	*Salto*	*Intercambios*	*Lista*
1	3	(6,2),(5,4),(6,0)	2 1 4 0 3 5 6
2	3	(2, 0)	0 1 4 2 3 5 6
3	3	*ninguno*	0 1 4 2 3 5 6
salto 3/2 = 1			
4	1	(4,2),(4,3)	0 1 2 3 4 5 6
5	1	*ninguno*	0 1 2 3 4 5 6

6.7.1. Algoritmo de ordenación Shell

Los pasos a seguir por el algoritmo para una lista de n elementos son:

1. Dividir la lista original en n/2 grupos de dos, considerando un incremento o salto entre los elementos de n/2.
2. Clarificar cada grupo por separado, comparando las parejas de elementos, y si no están ordenados, se intercambian.

3. Se divide ahora la lista en la mitad de grupos (n/4), con un incremento o salto entre los elementos también mitad (n/4), y nuevamente se clasifica cada grupo por separado.
4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un incremento o salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.
5. El algoritmo termina cuando se consigue que el tamaño del salto es 1.

Por consiguiente, los recorridos por la lista está condicionados por el bucle,

```
intervalo ← n / 2
mientras (intervalo > 0) hacer
```

Para dividir la lista en grupos y clasificar cada grupo se anida este código,

```
desde i ← (intervalo + 1) hasta n hacer
  j ← i - intervalo
    mientras (j > 0) hacer
      k ← j + intervalo
      si (a[j] <= a[k])   entonces
            j ← - 1
      sino
            Intercambio (a[j], a[k]);
            j ← j - intervalo
      fin_si
    fin_mientras
fin_desde
```

En el código anterior se observa que se comparan pares de elementos indexados por j y k, (a[j],a[k]), separados por un *salto,* intervalo. Así, si n = 8 el primer valor de intervalo = 4, y los índices i = 5, j = 1, k = 6. Los siguientes valores de los índices son i = 6, j = 2, k = 7, y así hasta recorrer la lista.

Para realizar un nuevo recorrido de la lista con la mitad de grupos, el intervalo se hace la mitad:

```
intervalo ← intervalo / 2
```

así se repiten los recorridos por la lista, *mientras intervalo > 0.*

6.7.2. Codificación en C del algoritmo del método Shell

Al codificar en C este método de ordenación se ha de tener en cuenta que el operador, /, realiza una división entera si los operandos son enteros, y esto es importante al calcular el ancho del salto entre pares de elementos: intervalo = n/2.

En cuanto a los índices, C toma como base el índice 0 y, por consiguiente, se ha de desplazar una posición a la izquierda las variables índice respecto a lo expuesto en el algoritmo.

```
void ordenacionShell(double a[], int n)
{
int intervalo, i, j, k;

intervalo = n / 2;
while (intervalo > 0)
{
```

```
    for (i = intervalo; i < n; i++)
    {
      j = i - intervalo;
      while (j >= 0)
      {
        k = j + intervalo;
        if (a[j] <= a[k])
          j = -1;          /* así termina el bucle, par ordenado */
        else
        {
          double temp;
          temp = a[j];
          a[j] = a[k];
          a[k] = temp;
          j -= intervalo;
        }
      }
    }
    intervalo = intervalo / 2;
  }
}
```

6.8. ORDENACIÓN RÁPIDA (*QUICKSORT*)

El algoritmo conocido como *quicksort* (ordenación rápida) recibe el nombre de su autor, Tony Hoare. La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar, por lo que se puede considerar que aplica la técnica *divide y vencerás*. El método es, posiblemente, el más pequeño de código, más rápido, más elegante, más interesante y eficiente de los algoritmos de ordenación conocidos.

El método se basa en dividir los *n* elementos de la lista a ordenar en dos partes o particiones separadas por un elemento: una *partición izquierda*, un elemento *central* denominado *pivote* o elemento de partición, y una *partición derecha*. La partición o división se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) son menores que todos los elementos de la segunda sublista (partición derecha). Las dos sublistas se ordenan entonces independientemente.

Para dividir la lista en particiones (*sublistas*) se elige uno de los elementos de la lista y se utiliza como *pivote* o *elemento de partición*. Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede seleccionar cualquier elemento de la lista como pivote, por ejemplo, el primer elemento de la lista. Si la lista tiene algún orden parcial conocido, se puede tomar otra decisión para el pivote. Idealmente, el pivote se debe elegir de modo que se divida la lista exactamente por la mitad, de acuerdo al tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían pivotes ideales, mientras que 1 o 10 serían elecciones «pobres» de pivotes.

Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el pivote y la otra, todos los elementos (claves) mayores que o iguales que el pivote (o al revés). Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *quicksort*. La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista. La primera etapa de *quicksort* es la división o «particionado» recursivo de la lista hasta que todas las sublistas constan de sólo un elemento.

Ejemplo 6.2

Se ordena una lista de números enteros aplicando el algoritmo quicksort, como pivote se elige el primer elemento de la lista.

1. *Lista original* | 5 | 2 | 1 | 7 | 9 | 3 | 8 | 7 |

   pivote elegido | 5 |

   ↑

   sublista izquierda1 (elementos menores que 5) | 2 | 1 | 3 |

   sublista derecha1 (elementos mayores o iguales a 5) | 9 | 8 | 7 |

2. El algoritmo se aplica a la sublista izquierda

   Sublista Izda1 2 1 3

   ↑ *sublista Izda* 1

   pivote *sublista Dcha* 3

   Sublista Izda1 *Izda* *pivote* *Dcha*

   | 1 | 2 | 3 |

3. El algoritmo se aplica a la sublista derecha

   Sublista Dcha1 9 8 7

   ↑ *sublista Izda* 7 8

   pivote *sublista Dcha*

   Sublista Dcha1 *Izda* *pivote* *Dcha*

   | 7 | 8 | 9 |

4. *Lista ordenada final*

   Sublista izquierda *pivote* *Sublista derecha*

   1 2 3 5 7 8 9

Para recordar

El algoritmo *quicksort* requiere una estrategia de partición y la selección idónea del pivote. Las etapas fundamentales del algoritmo dependen del pivote elegido aunque la estrategia de partición suele ser similar.

6.8.1. Algoritmo *quicksort*

La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se ha de buscar el sistema para situar en la sublista izquierda todos los elementos

menores que el pivote y en la sublista derecha todos los elementos mayores que el pivote. Supongamos que todos los elementos de la lista son distintos, aunque será preciso tener en cuenta los casos en que existan elementos idénticos. En el Ejemplo 6.3 se elige como pivote el elemento central de la lista actual.

Ejemplo 6.3

Ordenar una lista de números enteros aplicando el algoritmo quicksort, como pivote se elige el elemento central de la lista.

Lista original:	8	1	4	9	6	3	5	2	7	0
Pivote (elemento central)	6									

La etapa 2 requiere mover todos los elementos menores al pivote a la parte izquierda del array y los elementos mayores a la parte derecha. Para ello se recorre la lista de izquierda a derecha utilizando un índice `i` que se inicializa en la posición más baja (`inferior`) buscando un elemento mayor al pivote. También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un índice `j` inicializado en la posición más alta (`superior`).

El índice `i` se detiene en el elemento `8` (mayor que el pivote) y el índice `j` se detiene en el elemento `0` (menor que el pivote).

```
  8   1   4   9   [6]   3   5   2   7   0
→ i↑                                  j←↑
```

Ahora se intercambian `8` y `0` para que estos dos elementos se sitúen correctamente en cada sublista; y se incrementa el índice `i`, y se decrementa `j` para seguir los intercambios.

```
0   1   4   9   [6]   3   5   2   7   8
    i↑                            j↑
```

A medida que el algoritmo continúa, `i` se detiene en el elemento mayor, `9`, y `j` se detiene en el elemento menor, 2.

```
0   1   4   9   [6]   3   5   2   7   8
            i↑                j↑
```

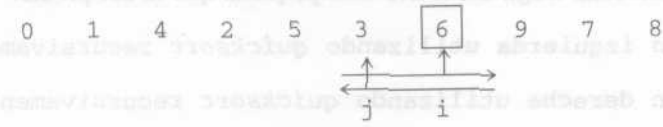
Se intercambian los elementos mientras que `i` y `j` no se crucen. En caso contrario se detiene este bucle. En el caso anterior se intercambian `9` y `2`.

```
0   1   4   2   [6]   3   5   9   7   8
            i↑            j↑
```

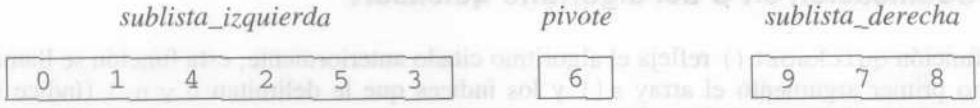
Continúa la exploración y ahora el contador `i` se detiene en el elemento `6` (que es el pivote) y el índice `j` se detiene en el elemento menor `5`.

```
0   1   4   2   5   3   [6]   9   7   8
            i↑↑j
```

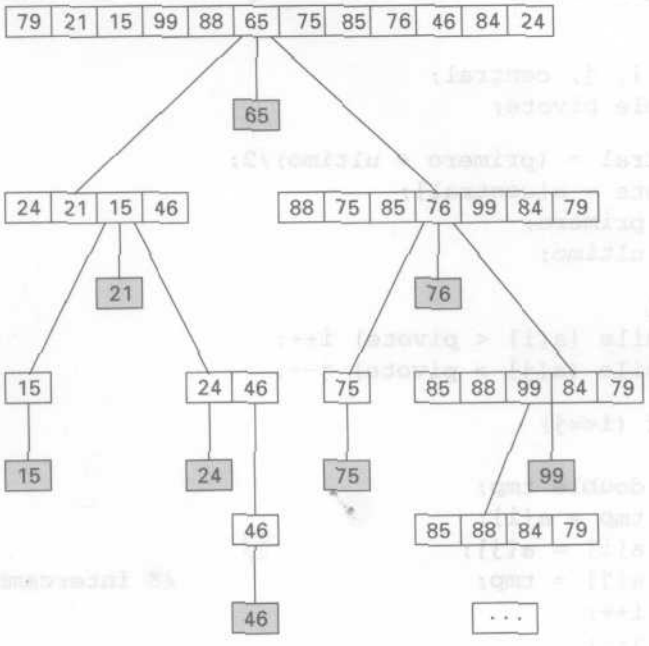
Los índices tienen actualmente los valores `i = 5`, `j = 5`. Continúa la exploración hasta que `i>j`, acaba con `i = 6`, `j = 5`.



En esta posición los índices `i` y `j` han cruzado posiciones en el array y en este caso se detiene la búsqueda y no se realiza ningún intercambio ya que el elemento al que accede `j` está ya correctamente situado. Las dos sublistas ya han sido creadas, la lista original se ha dividido en dos particiones:



El primer problema a resolver en el diseño del algoritmo de *quicksort* es seleccionar el pivote. Aunque la posición del pivote, en principio, puede ser cualquiera, una de las decisiones más ponderadas es aquella que considera el pivote como el elemento central o próximo al central de la lista. La Figura 6.2 muestra las operaciones del algoritmo para ordenar la lista `a[]` de `n` elementos enteros.



Izquierda: 24, 21, 15, 46
Pivote: 65
Derecha: 88, 75, 85, 76, 99, 84, 79

Figura 6.2. Ordenación rápida eligiendo como pivote el elemento central.

Los pasos que sigue el algoritmo quicksort:

```
Seleccionar el elemento central de a[0:n-1] como pivote
```

Dividir los elementos restantes en particiones *izquierda* y *derecha*, de modo que ningún elemento de la izquierda tenga una clave (valor) mayor que el pivote y que ningún elemento a la derecha tenga una clave más pequeña que la del pivote.

Ordenar la partición izquierda utilizando *quicksort* recursivamente.

Ordenar la partición derecha utilizando *quicksort* recursivamente.

La solución es partición *izquierda* seguida por el pivote y a continuación partición *derecha*.

6.8.2. Codificación en C del algoritmo *quicksort*

La función `quicksort()` refleja el algoritmo citado anteriormente; esta función se llama pasando como primer argumento el array `a[]` y los índices que le delimitan `0` y `n-1` (índice inferior y superior). La llamada a la función:

```
quicksort(a, 0, n-1);
```

y la codificación recursiva de la función:

```
void quicksort(double a[], int primero, int ultimo)
{

  int i, j, central;
  double pivote;

  central = (primero + ultimo)/2;
  pivote = a[central];
  i = primero;
  j = ultimo;

  do {
    while (a[i] < pivote) i++;
    while (a[j] > pivote) j--;

    if (i<=j)
    {
      double tmp;
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;                    /* intercambia a[i] con a[j] */
      i++;
      j--;
    }
  }while (i <= j);

  if (primero < j)
    quicksort(a, primero, j);/* mismo proceso con sublista izqda */

  if (i < ultimo)
    quicksort(a, i, ultimo); /* mismo proceso con sublista drcha */
}
```

6.8.3. Análisis del algoritmo *quicksort*

El análisis general de la eficiencia de *quicksort* es difícil. La mejor forma de ilustrar y calcular la complejidad del algoritmo es considerar el número de comparaciones realizadas teniendo en cuenta circunstancias ideales. Supongamos que n (número de elementos de la lista) es una potencia de 2, $n = 2^k$ ($k = \log_2 n$). Además, supongamos que el pivote es el elemento central de cada lista, de modo que *quicksort* divide la sublista en dos sublistas aproximadamente iguales.

En la primera exploración o recorrido hay $n - 1$ comparaciones. El resultado de la etapa crea dos sublistas aproximadamente de tamaño $n/2$. En la siguiente fase, el proceso de cada sublista requiere aproximadamente $n/2$ comparaciones. Las comparaciones totales de esta fase son $2(n/2) = n$. La siguiente fase procesa cuatro sublistas que requieren un total de $4(n/4)$ comparaciones, etc. Eventualmente, el proceso de división termina después de k pasadas cuando la sublista resultante tenga tamaño 1. El número total de comparaciones es aproximadamente:

$$n + 2(n/2) + 4(n/4) + \cdots + n(n/n) = n + n + \cdots + n$$
$$= n \cdot k = n \cdot \log_2 n$$

Para una lista normal la complejidad de *quicksort* es $O(n \log_2 n)$. El caso ideal que se ha examinado se realiza realmente cuando la lista (el array) está ordenado en orden ascendente. En este caso el pivote es precisamente el centro de cada sublista.

15	25	35	40	50	55	65	75

↑
pivote

Si el array está en orden ascendente, el primer recorrido encuentra el pivote en el centro de la lista e intercambia cada elemento en las sublistas inferiores y superiores. La lista resultante está casi ordenada y el algoritmo tiene la complejidad $O(n \log_2 n)$.

El escenario del caso peor de *quicksort* ocurre cuando el pivote cae consistentemente en una sublista de un elemento y deja el resto de los elementos en la segunda sublista. Esto sucede cuando el pivote es siempre el elemento más pequeño de su sublista. En el recorrido inicial, hay n comparaciones y la sublista grande contiene $n - 1$ elementos. En el siguiente recorrido, la sublista mayor requiere $n - 1$ comparaciones y produce una sublista de $n - 2$ elementos, etc. El número total de comparaciones es:

$$n + n - 1 + n - 2 + \cdots + 2 = (n - 1)(n + 2)/2$$

La complejidad es $O(n^2)$. En general el algoritmo de ordenación tiene como complejidad media $O(n \log_2 n)$ siendo posiblemente el algoritmo más rápido. La Tabla 6.1 muestra las complejidades de los algoritmos empleados en los métodos explicados en el libro.

Tabla 6.1. Comparación de complejidad en métodos de ordenación

Método	Complejidad
Burbuja	n^2
Inserción	n^2
Selección	n^2
Montículo	$n \log_2 n$
Fusión	$n \log_2 n$
Shell	$n \log_2 n$
Quicksort	$n \log_2 n$

En conclusión, se suele recomendar que para listas pequeñas los métodos más eficientes son inserción y selección; y para listas grandes, el *quicksort*. El algoritmo de Shell suele variar mucho su eficiencia en función de la variación del número de elementos por lo que es más difícil que en los otros métodos proporcionar un consejo eficiente. Los métodos de *fusión y por montículo* suelen ser muy eficientes para listas muy grandes.

El método de *ordenación por montículos*, también llamado *Heapsort*, se desarrolla más adelante, en el Capítulo 12, «Colas de prioridades y montículos», al ser una aplicación de la estructura montículo.

6.9. ORDENACIÓN BINSORT Y RADIXSORT

Estos métodos de ordenación utilizan *urnas* para depositar en ellas los registros en el proceso de ordenación. En cada recorrido de la lista se depositan en una $urna_i$ aquellos registros cuya clave tienen una cierta correspondencia con *i*.

6.9.1. Método de Binsort

Este método, también llamado *clasificación por urnas*, persigue conseguir funciones de tiempo de ejecución menores de `O(n log n)`, para ordenar una secuencia de `n` elementos siempre que se conozca *algo* acerca del tipo de las claves por las que se están ordenando.

Supóngase que se tiene un vector `v[]` de registros, se quiere ordenar respecto un campo clave de tipo entero, además se sabe que los valores de las claves se encuentran en el rango de `1` a `n`, sin claves duplicadas y siendo `n` el número de elementos. En estas circunstancias es posible colocar los registros ordenados en un array auxiliar `t[]` mediante este bucle:

```
for 1:= 1 to n do
   t[v[i].clave] = v[i];
```

Sencillamente determina la posición que le corresponde según el valor del campo clave. El bucle lleva un tiempo de ejecución de complejidad `O(n)`.

Esta ordenación tan sencilla que se ha expuesto es un caso particular del método de ordenación por urnas (binsort). Este método utiliza urnas, cada urna contiene todos los registros con una misma clave.

El proceso consiste en examinar cada registro `r` a clasificar y situarle en la urna `i`, coincidiendo `i` con el valor del campo clave de `r`. En la mayoría de los casos en que se utilice el algoritmo, será necesario guardar más de un registro en una misma urna por tener claves repetidas. Entonces estas urnas hay que concatenarlas en el orden de menor índice de urna a mayor, así quedará el array en orden creciente respecto al campo clave.

En la Figura 6.3 se muestra un vector de `m` urnas. Las urnas están representadas por listas enlazadas.

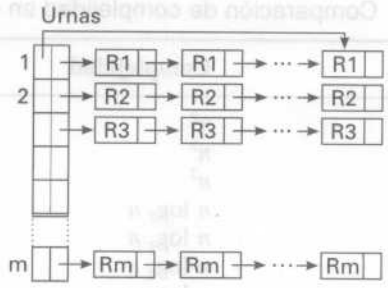


Figura 6.3. Estructura formada por `m` urnas.

Algoritmo de ordenación Binsort

Se considera que el campo clave de los registros que se van a ordenar son números enteros en el rango `1 . . m`. Son necesarias `m` urnas por lo que es necesario definir un vector de `m` urnas. Las urnas pueden ser representadas por listas enlazadas, cada elemento de la lista contiene un registro cuyo campo clave es el correspondiente al de la urna en la que se encuentra. Así en la urna 1 se sitúan los registros cuyo campo clave sea igual a 1, en la urna 2 los registros cuyo campo clave sea 2, y así sucesivamente en la urna *i* se sitúan los registros cuyo campo clave sea igual a *i*.

Una vez que se hayan distribuido los registros en las diversas urnas es necesario concatenar las listas. En la Figura 6.4 se muestra cómo realizar la concatenación.

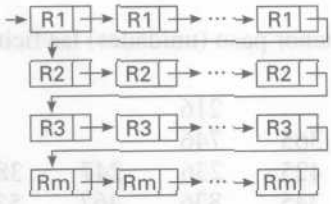


Figura 6.4. Concatenación de urnas representadas por listas enlazadas.

Los pasos que sigue el algoritmo expresado en pseudocódigo para un vector de `n` registros:

```
OrdenacionBinsort(vector, n)

inicio
  CrearUrnas(Urnas);
    {Distribución de registros en sus correspondientes urnas}
  desde j = 1 hasta n hacer
    AñadirEnUrna(Urnas[vector[j].clave], vector[j]);
  fin_desde

    {Concatena las listas que representan a las urnas
      desde Urna1 hasta Urnam}

  i = 1;                    {búsqueda de primera urna no vacía}
  mientras EstaVacia(Urnas[i]) hacer
    i = i+1
  fin_mientras

  desde j = i+1 a m hacer
    EnlazarUrna(Urnas[i], Urnas[j]);
  fin_desde

    {Se recorre la lista-urna resultante de la concatenación}
  j = 1;
  dir = <frente Urnas[i]>;
  mientras dir <> nulo hacer
    vector[j] = <registro apuntado por dir>;
    j = j+i;
    dir = Sgte(dir)
  fin_mientras

fin
```

6.9.2. Método de Radixsort

Este método puede considerarse como una generalización de la clasificación por urnas. Aprovecha la estrategia de la forma más antigua de clasificación manual, consistente en hacer diversos *montones* de fichas, cada uno caracterizado por tener sus componentes un mismo dígito (letra, si es alfabética) en la misma posición; estos montones se recogen en orden ascendente y se reparte de nuevo en montones según el siguiente dígito de la clave.

Como ejemplo, suponer que se han de ordenar estas fichas identificadas por tres dígitos:

345, 721, 425, 572, 836, 467, 672,194, 365, 236, 891, 746, 431, 834, 247, 529, 216, 389

Atendiendo al dígito de menor peso (unidades) las fichas se distribuyen en *montones* del 0 al 9;

				216		
431			365	746		
891	672	834	425	236	247	389
721	572	194	345	836	467	529
1	2	4	5	6	7	9

Recogiendo los *montones* en orden, la secuencia de fichas queda:

721, 891, 431, 572, 672, 194, 834, 345, 425, 365, 836, 236, 746, 216, 467, 247, 529, 389

De esta secuencia podemos decir que está ordenada respecto al dígito de menor peso, respecto a las unidades. Pues bien, ahora de nuevo se distribuye la secuencia de fichas en *montones* respecto al segundo dígito:

		236					
	529	836	247				
	425	834	746	467	672		194
216	721	431	345	365	572	389	891
1	2	3	4	6	7	8	9

Recogiendo de nuevo los *montones* en orden, la secuencia de fichas queda:

216, 721, 425, 529, 431, 834, 836, 236, 345, 746, 247, 365, 467, 572, 672, 389, 891, 194

En este momento esta secuencia de fichas ya están ordenadas respecto a los dos últimos dígitos, es decir, respecto a las decenas. Por último, se distribuye las fichas en *montones* respecto al tercer dígito:

	247	389	467				891
	236	365	431	572		746	836
194	216	345	425	529	672	721	834
1	2	3	4	5	6	7	8

Recogiendo de nuevo los *montones* en orden, la secuencia de fichas queda ya ordenada:

194, 216, 236, 247, 345, 365, 389, 425, 431, 467, 529, 572, 672, 121, 746, 834, 836, 891

Algoritmo de ordenación Radixsort

La idea clave de la ordenación Radixsort (también llamada *por residuos*) es clasificar por urnas primero respecto al dígito de menor peso (menos significativo) d_k, después concatenar las urnas, clasificar de nuevo respecto al siguiente dígito d_{k-1}, y así sucesivamente se sigue con el siguiente dígito hasta alcanzar el dígito más significativo d_l, en ese momento la secuencia estará ordenada. La concatenación de las urnas consiste en enlazar el final de una con el frente de la siguiente.

Al igual que en el método de Binsort, las urnas se representan mediante un vector de listas. En el caso de que la clave respecto a la que se ordena sea un entero, se tendrán 10 urnas, numeradas de 0 a 9. Si la clave respecto a la que se ordena es alfabética, habrá tantas urnas como letras distintas, desde la urna que represente a la letra *a* hasta la *z*.

Para el caso de que clave sea entera, en primer lugar se determina el máximo número de dígitos que puede tener la clave. En un bucle de tantas iteraciones como máximo de dígitos se realizan las acciones de distribuir por urnas los registros, concatenar...

La distribución por urnas exige obtener el dígito del campo clave que se encuentra en la posición definida por el bucle externo, dicho dígito será el índice de la urna.

```
OrdenacionRadixsort(vector, n)

inicio
  < cálculo el número máximo de dígitos: ndig >

    peso = 1 { permite obtener los dígitos de menor a mayor peso}
    desde i = 1 hasta ndig hacer
      CrearUrnas(Urnas);
      desde j = 1 hasta n hacer
        d = (vector[j] / peso) modulo 10;
        AñadirEnUma(Urnas[d], vector[j]);
      fin_desde

      < búsqueda de primera urna no vacía: j >

      desde r = j+1 hasta M hace      { M: número de urnas }
        EnlazarUma(Urnas[r], Urnas[j]);
      fin_desde

      {Se recorre la lísta-urna resultante de la concatenación}
      r = 1;
      dir = frente(Urna[j]);
      mientras dir <> nulo hacer
        vector[r] = dir.registro;
        r = r+1;
        dir = siguiente(dir)
      end

      peso = peso * 10;
    fin_desde

fin_ordenacion
```

6.10. BÚSQUEDA EN LISTAS: BÚSQUEDAS SECUENCIAL Y BINARIA

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arrays y registros, y por ello será necesario determinar si un array contiene un valor que coincida con un cierto *valor clave*. El proceso de encontrar un elemento específico de un array se denomina *búsqueda*. En esta sección se examinarán dos técnicas de búsqueda: *búsqueda lineal* o *secuencial*, la técnica más sencilla, y *búsqueda binaria* o *dicotómica*, la técnica más eficiente.

6.10.1. Búsqueda secuencial

La **búsqueda secuencial** busca un elemento de una lista utilizando un valor destino llamado ***clave***. En una búsqueda secuencial (a veces llamada ***búsqueda lineal***), los elementos de una lista o vector se exploran (se examinan) en secuencia, uno después de otro. La búsqueda secuencial es necesaria, por ejemplo, si se desea encontrar la persona cuyo número de teléfono es 958-220000 en un directorio o listado telefónico de su ciudad. Los directorios de teléfonos están organizados alfabéticamente por el nombre del abonado en lugar de por números de teléfono, de modo que deben explorarse todos los números, uno después de otro, esperando encontrar el número 958-220000.

El algoritmo de búsqueda secuencial compara cada elemento del array con la *clave* de búsqueda. Dado que el array no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primer elemento, el último elemento o cualquier otro. De promedio, al menos el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del array. El método de búsqueda lineal funcionará bien con arrays pequeños o no ordenados. La eficiencia de la búsqueda secuencial es pobre, tiene complejidad lineal, $O(n)$.

6.10.2. Búsqueda binaria

La búsqueda secuencial se aplica a cualquier lista. Si la lista está ordenada, la *búsqueda binaria* proporciona una técnica de búsqueda mejorada. Una búsqueda binaria típica es la búsqueda de una palabra en un diccionario. Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra del primer apellido o de la palabra que busca. Se puede tener suerte y acertar con la página correcta; pero, normalmente, no será así y se mueve el lector a la página anterior o posterior del libro. Por ejemplo, si la palabra comienza con «J» y se está en la «L» se mueve uno hacia atrás. El proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que la palabra no está en la lista.

Una idea similar se aplica en la búsqueda en una lista ordenada. Se sitúa la lectura en el centro de la lista y se comprueba si la clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sigue la búsqueda uno en la mitad inferior o superior del elemento central de la lista. En general, si los datos de la lista están ordenados se puede utilizar esa información para acortar el tiempo de búsqueda.

Ejemplo 6.4

Se desea buscar el elemento 225 y ver si se encuentra en el conjunto de datos siguiente:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
13	44	75	100	120	275	325	510

El punto central de la lista es el elemento `a[3] (100)`. El valor que se busca es `225`, mayor que `100`; *por consiguiente, la búsqueda continúa en la mitad superior del conjunto de datos de la lista, es* decir, en la sublista,

```
a[4]    a[5]    a[6]    a[7]
120     275     325     510
```

Ahora el elemento mitad de esta sublista `a[5] (275)`. El valor buscado, `225`, es menor que `275` y, por consiguiente, la búsqueda continúa en la mitad inferior del conjunto de datos de la lista actual; es decir, en la sublista de un único elemento:

```
a[4]
120
```

El elemento mitad de esta sublista es el propio elemento `a[4] (120)`. Al ser `225` mayor que `120`, la búsqueda debe continuar en una sublista vacía. Se concluye indicando que no se ha encontrado la clave en la lista.

6.10.3. Algoritmo y codificación de la búsqueda binaria

Suponiendo que la lista está almacenada como un array, los índices de la lista son: `bajo = 0` y `alto = n-1` y `n` es el número de elementos del array, los pasos a seguir:

1. Calcular el índice del punto central del array

   `central = (bajo + alto)/2` *(división entera)*

2. Comparar el valor de este elemento central con la clave:

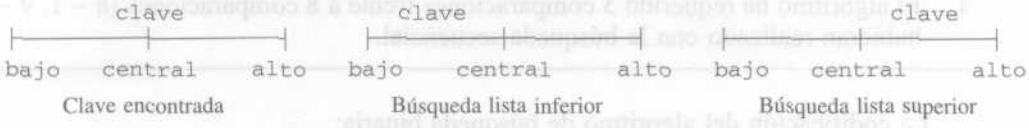


Figura 6.5. Búsqueda binaria de un elemento.

- Si `a[central] < clave`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo = central+1..alto`.
- Si `clave < a[central]`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo..central-1`.

El algoritmo se termina bien porque se ha encontrado la clave o porque el valor de `bajo` excede a `alto` y el algoritmo devuelve el indicador de fallo de –1 (*búsqueda no encontrada*).

Ejemplo 6.5

Sea el array de enteros `A (-8, 4, 5, 9, 12, 18, 25, 40, 60)`, buscar la clave, clave = 40.

1.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
-8	4	5	9	12	18	25	40	60

bajo = 0
alto = 8

↑ *central*

$$central = \frac{bajo + alto}{2} = \frac{0+8}{2} = 4$$

`clave (40) > a[4] (12)`

2. Buscar en sublista derecha

18	25	40	60

bajo = 5
alto = 8

↑

$$central = \frac{bajo + alto}{2} = \frac{5+8}{2} = 6 \quad (\textit{división entera})$$

`clave (40) > a[6] (25)`

3. Buscar en sublista derecha

40	60

bajo = 7
alto = 8

↑

$$central = \frac{bajo + alto}{2} = \frac{7+8}{2} = 7$$

`clave (40) = a[7] (40)` (*búsqueda con éxito*)

4. El algoritmo ha requerido 3 comparaciones frente a 8 comparaciones ($n - 1$, $9 - 1 = 8$) que se hubieran realizado con la búsqueda secuencial.

La codificación del algoritmo de búsqueda binaria:

```
/*
  búsqueda binaria.
  devuelve el índice del elemento buscado, o bien -1 caso de fallo
*/

int busquedaBin(int lista[], int n, int clave)
{
  int central, bajo, alto;
  int valorCentral;

  bajo = 0;
  alto = n-1;
  while (bajo <= alto)
```

```
{
  central = (bajo + alto)/2;          /* índice de elemento central */
  valorCentral = lista[central];      /* valor del índice central */
  if (clave == valorCentral)
    return central;                   /* encontrado, devuelve posición */
  else if (clave < valorCentral)
    alto = central -1;                /* ir a sublista inferior */
  else
    bajo = central + 1;               /* ir a sublista superior */
}
 return -1;                           /* elemento no encontrado */
}
```

6.10.4. Análisis de los algoritmos de búsqueda

Al igual que sucede con las operaciones de ordenación cuando se realizan operaciones de búsqueda es preciso considerar la eficiencia (complejidad) de los algoritmos empleados en la búsqueda. El grado de eficiencia en una búsqueda será vital cuando se trata de localizar una información en una *lista o tabla en memoria, o bien en un archivo de datos.*

Complejidad de la búsqueda secuencial

La complejidad de la búsqueda secuencial diferencia entre el comportamiento en el caso peor y mejor. El mejor caso se encuentra cuando aparece una coincidencia en el primer elemento de la lista y en ese caso el tiempo de ejecución es $O(1)$. El caso peor se produce cuando el elemento no está en la lista o se encuentra al final de la lista. Esto requiere buscar en todos los n términos, lo que implica una complejidad de $O(n)$.

El caso medio requiere un poco de razonamiento probabilista. Para el caso de una lista aleatoria es probable que una coincidencia ocurra en cualquier posición. Después de la ejecución de un número grande de búsquedas, la posición media para una coincidencia es el elemento central $n/2$. El elemento central ocurre después de $n/2$ comparaciones, que define el coste esperado de la búsqueda. Por esta razón, se dice que la prestación media de la búsqueda secuencial es $O(n)$.

Análisis de la búsqueda binaria

El caso mejor se presenta cuando una coincidencia se encuentra en el punto central de la lista. En este caso la complejidad es $O(1)$ dado que sólo se realiza una prueba de comparación de igualdad. *La complejidad del caso peor es $O(\log_2 n)$ que se produce cuando el elemento no está en la lista o el* elemento se encuentra en la última comparación. Se puede deducir intuitivamente esta complejidad. El caso peor se produce cuando se debe continuar la búsqueda y llegar a una sublista de longitud de 1. Cada iteración que falla debe continuar disminuyendo la longitud de la sublista por un factor de 2. El tamaño de las sublistas es:

$$n \quad n/2 \quad n/4 \quad n/8 \ \ldots \ 1$$

La división de sublistas requiere m iteraciones, en cada iteración el tamaño de la sublista se reduce a la mitad. La sucesión de tamaños de las sublistas hasta una sublista de longitud 1:

$$n \quad n/2 \quad n/2 \quad n/2^3 \quad n/2^4 \ldots \ n/2^m$$

siendo $n/2^m = 1$.

Tomando logaritmos en base 2 en la expresión anterior quedará:

$n = 2^m$
$m = \log_2 n$

Por esa razón la complejidad del caso peor es $O(\log_2 n)$. Cada iteración requiere una operación de comparación:

Total comparaciones $\approx 1 + \log_2 n$

Comparación de la búsqueda binaria y secuencial

La comparación en tiempo entre los algoritmos de búsqueda secuencial y binaria se va haciendo espectacular a medida que crece el tamaño de la lista de elementos. Tengamos presente que en el caso de la búsqueda secuencial, en el peor de los casos, coincidirá el número de elementos examinados con el número de elementos de la lista tal como representa su complejidad $O(n)$.

Sin embargo, en el caso de la búsqueda binaria, tengamos presente, por ejemplo, que $2^{10} = 1.024$, lo cual implica el examen de 11 posibles elementos; si se aumenta el número de elementos de una lista a 2.048 y teniendo presente que $2^{11} = 2.048$ implicará que el número máximo de elementos examinados en la búsqueda binaria es 12. Si se sigue este planteamiento, se puede encontrar el número m más pequeño para una lista de 1.000.000, tal que

$2^n \geq 1.000.000$

Es decir, $2^{19} = 524.288$, $2^{20} = 1.048.576$ y por tanto el número de elementos examinados (en el peor de los casos) es 21.

La Tabla 6.2 muestra la comparación de los métodos de búsqueda secuencial y búsqueda binaria. En la misma tabla se puede apreciar una comparación del número de elementos que se deben examinar utilizando búsquedas secuencial y binaria. Esta tabla muestra la eficiencia de la búsqueda binaria comparada con la búsqueda secuencial y cuyos resultados de tiempo vienen dados por las funciones de complejidad $O(\log_2 n)$ y $O(n)$ de las búsquedas binaria y secuencial respectivamente.

Tabla 6.2. Comparación de las búsquedas binaria y secuencial

	Números de elementos examinados	
Tamaño de la lista	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	1.000.000

A tener en cuenta

La búsqueda secuencial se aplica para localizar una clave en un vector no ordenado. Para aplicar el algoritmo de búsqueda binaria la lista, o vector, donde se busca debe de estar ordenado.

La *complejidad* de la *búsqueda binaria* es *logarítmica*, $O(\log n)$, más eficiente que la *búsqueda secuencial* que tiene *complejidad lineal*, $O(n)$.

RESUMEN

- Una de las aplicaciones más frecuentes en programación es la ordenación.
- Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos.*
- Los datos se pueden ordenar en orden ascendente o en orden descendente.
- Cada recorrido de los datos durante el proceso de ordenación se conoce como *pasada* o *iteración.*
- Los algoritmos de ordenación básicos son:

  — Selección.
  — Inserción.
  — Burbuja.

- Los algoritmos de ordenación más avanzados son:

  — Shell.
  — Mergesort.
  — Radixsort.
  — Binsort.
  — Quicksort.

- La eficiencia de los algoritmos de burbuja, inserción y selección es $O(n^2)$.
- La eficiencia de los algoritmos, *radixsort*, *mergesort* y *quicksort* es $O(n \log n)$.
- La búsqueda es el proceso de encontrar la posición de un elemento destino dentro de una lista
- Existen dos métodos básicos de búsqueda en arrays: **secuencial** y **binaria.**
- La **búsqueda secuencial** se utiliza normalmente cuando el array no está ordenado. Comienza en el principio del array y busca hasta que se encuentra el dato buscado y se llega al final de la lista.
- Si un array está ordenado, se puede utilizar un algoritmo más eficiente denominado **búsqueda binaria.**
- La eficiencia de una búsqueda secuencial es $\boldsymbol{O(n)}$.
- La eficiencia de una búsqueda binaria es $\boldsymbol{O(\log n)}$.

EJERCICIOS

6.1. ¿Cuál es la diferencia entre ordenación por intercambio y ordenación por el método de la burbuja?

6.2. Se desea eliminar todos los números duplicados de una lista o vector (array). Por ejemplo, si el array toma los valores

4 7 11 4 9 5 11 7 3 5

ha de cambiarse a

4 7 11 9 5 3

Escribir una función que elimine los elementos duplicados de un array.

6.3. Escribir una función que elimine los elementos duplicados de un vector ordenado. ¿Cuál es la eficiencia de esta función? Comparar la eficiencia con la correspondiente a la función del Ejercicio 6.2.

6.4. Un vector contiene los elementos mostrados a continuación. Los primeros dos elementos se han ordenado utilizando un algoritmo de inserción. ¿Cuál será el valor de los elementos del vector después de tres pasadas más del algoritmo?

3	13	8	25	45	23	98	58

6.5. Se tiene la siguiente lista:

47	3	21	32	56	92

Después de dos pasadas de un algoritmo de ordenación, el array se ha quedado dispuesto así:

3	21	47	32	56	92

¿Qué algoritmo de ordenación se está utilizando (selección, burbuja o inserción)? Justifique la respuesta.

6.6. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de ordenación Shell encuentre las pasadas y los intercambios que se realizan para su ordenación.

8	43	17	6	40	16	18	97	11	7

6.7. Partiendo del mismo array que en el Ejercicio 6.6, encuentre las particiones e intercambios que realiza el algoritmo de ordenación *quicksort* para su ordenación.

6.8. Un array de registros se quiere ordenar 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.

6.9. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número 88.

8	13	17	26	44	56	88	97

Igual búsqueda pero aplicada al caso del número 20.

6.10. Escribir la función de ordenación correspondiente al método Radixsort para poner en orden alfabético una lista de n nombres.

6.11. Escribir una función de búsqueda binaria aplicado a un array ordenado en modo descendente.

6.12. Supongamos que se tiene una secuencia de n números que deben ser clasificados:

1. Utilizar el método de Shell para determinar cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si:
   - Ya está clasificada.
   - Está en orden inverso.
2. Repetir el paso 1 para el método de *quicksort*.

PROBLEMAS

6.1. Un método de ordenación muy simple, pero no muy eficiente, de elementos $x_1, x_2, x_3, \ldots x_n$ en orden ascendente es el siguiente:

Paso 1: Localizar el elemento más pequeño de la lista x_1 a x_n; intercambiarlo con x_1.
Paso 2: Localizar el elemento más pequeño de la lista x_2 a x_n, intercambiarlo con x_2.
Paso 3: Localizar el elemento más pequeño de la lista x_3 a x_n, intercambiarlo con x_n.

En el último paso, los dos últimos elementos se comparan e intercambian, si es necesario, y la ordenación se termina. Escribir un programa para ordenar una lista de elementos, siguiendo este método.

6.2. Dado un vector *x* de *n* elementos reales, donde *n* es impar, diseñar una función que calcule y devuelva la mediana de ese vector. La mediana es el valor tal que la mitad de los números son mayores que el valor y la otra mitad son menores. Escribir un programa que compruebe la función.

6.3. Se trata de resolver el siguiente problema escolar. Dadas las notas de los alumnos de un colegio en el primer curso de bachillerato, en las diferentes asignaturas (5, por comodidad), se trata de calcular la media de cada alumno, la media de cada asignatura, la media total de la clase y ordenar los alumnos por orden decreciente de notas medias individuales.
Nota: Utilizar como algoritmo de ordenación el método Shell.

6.4. Escribir un programa de consulta de teléfonos. Leer un conjunto de datos de mil nombres y números de teléfono de un archivo que contiene los números en orden aleatorio. Las consultas han de poder realizarse por nombre y por número de teléfono.

6.5. Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria. Una lista A se rellena con 2.000 enteros aleatorios en el rango 0 ... 1.999 y a continuación se ordena. Una segunda lista B se rellena con 500 enteros aleatorios en el mismo rango. Los elementos de B se utilizan como claves de los algoritmos de búsqueda.

6.6. Se dispone de dos vectores, *Maestro* y *Esclavo*, del mismo tipo y número de elementos. Se deben imprimir en dos columnas adyacentes. Se ordena el vector *Maestro*, pero siempre que un elemento de *Maestro* se mueva, el elemento correspondiente de *Esclavo* debe moverse también; es decir, cualquier cosa que se haga en *Maestro*[*i*] debe hacerse en *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*.

6.7. Cada línea de un archivo de datos contiene información sobre una compañía de informática. La línea contiene el nombre del empleado, las ventas efectuadas por el mismo y el número de años de antigüedad del empleado en la compañía. Escribir un programa que lea la información del archivo de datos y a continuación se visualiza. La información debe ser ordenada por ventas de mayor a menor y visualizada de nuevo.

6.8. Se desea realizar un programa que realice las siguientes tareas:

a) Generar, aleatoriamente, una lista de 999 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.

Ampliar el programa anterior de modo que se puedan obtener y visualizar en el programa principal los siguientes tiempos:

t1. Tiempo empleado en ordenar la lista por cada uno de los métodos.
t2. Tiempo que se emplearía en *ordenar* la lista ya ordenada.
t3. Tiempo empleado en ordenar la lista ordenada en orden inverso.

6.9. Construir un método que permita ordenar por fechas y de mayor a menor un vector de *n* elementos que contiene datos de contratos ($n \leq 50$). Cada elemento del vector debe

ser un objeto con los campos día, mes, año y número de contrato. Pueden existir diversos contratos con la misma fecha, pero no números de contrato repetidos.
Nota: El método a utilizar para ordenar será el de radixsort.

6.10. Escribir un programa que genere un vector de 10.000 números aleatorios de 1 a 500. Realice la ordenación del vector por dos métodos:

- Binsort.
- Radixsort.

Escribir el tiempo empleado en la ordenación de cada método.

6.11. Se leen dos listas de números enteros, A y B de 100 y 60 elementos, respectivamente. Se desea resolver mediante procedimientos 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.

7

ALGORITMOS DE ORDENACIÓN DE ARCHIVOS

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Conocer el tratamiento de archivos en C.
- Procesar un archivo de organización secuencial.
- Procesar un archivo de acceso directo.
- Distinguir entre ordenación en memoria y ordenación externa.
- Conocer los algoritmos de ordenación de archivos basados en la mezcla.
- Realizar la ordenación de archivos secuenciales con un mezcla múltiple.
- Conocer la base de la ordenación polifásica.

CONTENIDO

7.1. Archivos en C.
7.2. Lectura y escritura de archivos de texto.
7.3. Archivos binarios en C.
7.4. Ordenación externa. Métodos de ordenación.
7.5. Método de mezcla directa (mezcla simple).
7.6. Fusión natural.
7.7. Mezcla equilibrada múltiple.
7.8. Método polifásico de ordenación externa.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Memoria interna.
- Memoria externa.
- Flujos.
- Archivos de texto.
- Acceso secuencial.
- Ordenación.
- Mezcla.
- Secuencia ordenada
- Secuencia de Fibonacci.

INTRODUCCIÓN

Los algoritmos de ordenación presentados ya, no se pueden aplicar si la cantidad de 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 los datos se almacenan en archivos cuya característica esencial es que, en cada momento, un componente o elemento es accesible directamente.

Es necesario aplicar nuevas técnicas de ordenación que se complementen con las ya estudiadas. Entre las técnicas más importantes destaca la *mezcla*. Mezclar, significa combinar dos (o más) secuencias en una sola secuencia ordenada por medio de una selección repetida entre los componentes accesibles en ese momento.

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.

7.1. ARCHIVOS EN C

Un *fichero* (archivo) de datos —o simplemente un **archivo**— es una colección de registros relacionados entre sí con aspectos en común y organizados para un propósito específico. Por ejemplo, un archivo de una clase escolar contiene un conjunto de registros de los estudiantes de esa clase. Otros ejemplos pueden ser el archivo de nóminas de una empresa, inventarios, stocks, etc.

Un archivo en una computadora es una estructura diseñada para contener datos. Los datos están organizados de tal modo que puedan ser recuperados fácilmente, actualizados o borrados y almacenados de nuevo en el archivo con todos los cambios realizados.

Según las características del soporte empleado y el modo en que se han organizado los registros, se consideran dos tipos de acceso a los registros de un archivo:

- Acceso secuencial.
- Acceso directo.

El *acceso secuencial* implica el acceso a un archivo según el orden de almacenamiento de sus registros, uno tras otro.

El *acceso directo* implica el acceso a un registro determinado, sin que ello implique la consulta de los registros precedentes. Este tipo de acceso sólo es posible con soportes direccionables.

La *organización* de un archivo define la forma en la que los registros se disponen sobre el soporte de almacenamiento, o también se define la organización como la forma en que se estructuran los datos en un archivo. En general, se consideran tres organizaciones fundamentales:

- Organización secuencial.
- Organización directa o aleatoria (*random*).
- Organización secuencial indexada (*indexed*).

7.1.1. Flujos

El manejo de archivos en C se hace mediante el concepto de **flujo** (*stream*) o canal, o también denominado secuencia. Un **flujo** (*stream*) es una abstracción que se refiere a un *flujo* o *corriente* de datos que fluyen entre un origen o fuente (*productor*) y un destino o sumidero (*consumidor*). Entre el origen y el destino debe existir una conexión o canal (*pipe*) por la cual circulen los datos. La apertura de un archivo supone establecer la conexión del programa con el dispositivo que contiene al archivo, por el canal que comunica el archivo con el programa van a fluir las secuencias de datos. Hay tres flujos o canales abiertos automáticamente:

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

Estas tres variables se inicializan al comenzar la ejecución del programa, y pueden admitir secuencias de caracteres, en modo texto. Tienen el siguiente cometido:

`stdin` *Asocia la entrada estándar (teclado) con el programa.*
`stdout` *Asocia la salida estándar (pantalla) con el programa.*
`stderr` *Asocia la salida de mensajes de error (pantalla) con el programa.*

Así, cuando se ejecuta `printf("Calle Mayor 2.")`: se escribe en `stdout`, en pantalla. Y cuando se quiere leer una variable entera con `scanf("%d",&x)`: se captan los dígitos de la secuencia de entrada `stdin`.

El acceso a los archivos se hace con un buffer intermedio. Se puede pensar en el buffer como un array donde se van almacenando los datos dirigidos al archivo, o desde el archivo; el buffer se vuelca cuando de una forma u otra se da la orden de vaciarlo. Por ejemplo, cuando se llama a una

función para leer del archivo una cadena, dicha función lee tantos caracteres como quepan en el buffer. Luego, la primera cadena del buffer es la que se obtiene; una siguiente llamada a la función obtendrá la siguiente cadena del buffer, así hasta que se quede vacío y sea llenado con una posterior llamada a la función de lectura.

El lenguaje C trabaja con archivos con buffer, y está diseñado para acceder a una amplia gama de dispositivos, de tal forma que trata cada dispositivo como una secuencia, pudiendo haber secuencias de caracteres y secuencias binarias. Con las secuencias se simplifica el manejo de archivos en C.

A tener en cuenta

El lenguaje C trabaja con archivos con buffer, y está diseñado para acceder a una amplia gama de dispositivos, de tal forma que trata cada dispositivo como una secuencia, pudiendo existir secuencias de caracteres y secuencias binarias. Con las secuencias se simplifica el manejo de archivos en C.

7.1.2. Puntero (apuntalador) FILE

Los archivos se ubican en dispositivos externos como cintas, cartuchos, discos, discos compactos (CD, DVD, etc.); tienen un nombre y unas características. En el programa el archivo tiene un nombre interno que es un puntero (apuntador) a una estructura predefinida (*puntero a archivo*). Esta estructura contiene información sobre el archivo, tal como la dirección del buffer que utiliza, el modo de apertura del archivo, el último carácter leído del buffer y otros detalles que generalmente el usuario no necesita saber. El identificador del tipo de la estructura es `FILE` (declarada en el archivo de cabecera `stdio.h`).

```
typedef struct
{
  short level;
  unsigned flags; /*estado del archivo: lectura, binario ... */
  char fd;
  unsigned char hold;
  short bsize;
  unsigned char *buffer, *curp;
  unsigned istemp;
  short token;
}FILE;
```

El detalle de los campos del tipo `FILE` puede cambiar de un compilador a otro. Al programador le interesa saber que existe el tipo `FILE` y que es necesario definir un puntero a `FILE` por cada archivo a procesar. Mucha de las funciones para procesar archivos son del tipo `FILE*`, y tienen argumento(s) de ese tipo.

Ejemplo 7.1

Se declara un puntero a `FILE`; se escribe el prototipo de una función de tipo puntero a `FILE` y con un argumento del mismo tipo.

```
FILE* pf;
FILE* mostrar(FILE*);          /* Prototipo de función definida por el
                                  Programador */
```

Recordar

La entrada estándar, al igual que la salida, están asociadas a variables puntero a `FILE: FILE *stdin, *stdout;`.

7.1.3. Apertura de un archivo

Para procesar un archivo en C (y en todos los lenguajes de programación) la primera operación a realizar es *abrir* el archivo. La apertura del archivo supone conectar el archivo externo con el programa, e indicar cómo va a ser tratado el archivo: binario, de caracteres, etc. El programa accede a los archivos a través de un puntero a la estructura `FILE`, y es la función de apertura, `fopen()`, la encargada de devolver dicho puntero o apuntador. El formato de llamada:

```
fopen(nombre_archivo, modo);
```

`nombre` ≡ `cadena` *Contiene el identificador externo del archivo.*
`modo` ≡ `cadena` *Contiene el modo en que se va a tratar el archivo.*

La llamada a `fopen()` se debe de hacer de tal forma que el valor devuelto se asigne a una variable puntero a `FILE`, para así después referirse a dicha variable.

Ejemplo 7.2

Declare una variable de tipo puntero (apuntador) a `FILE`; a continuación escriba una sentencia de apertura de un fichero.

```
FILE* pf;
pf = fopen(nombre_archivo, modo);
```

A tener en cuenta

La función `fopen()` puede detectar un error al abrir el archivo; por ejemplo, el archivo no existe y se desea leer, entonces devuelve `NULL`.

Ejemplo 7.3

Abrir un archivo de nombre LICENCIA.EST para obtener ciertos datos.

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

FILE *pf;
char nm[] = "C:\LICENCIA.EST";

pf = fopen(nm, "r");
if (pf == NULL)
{
  puts ("Error al abrir el archivo.");
  exit(1);
}
```

Ejemplo 7.4

Abrir el archivo de texto JARDINES.DAT para escribir en él los datos de un programa.

En la misma línea en que se ejecuta `fopen()` se comprueba que la operación ha sido correcta, en caso contrario termina la ejecución.

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

FILE *ff;
char* arch = "C:\AMBIENTE\JARDINES.DAT";

if ((ff = fopen(nm, "w"))==NULL)
{
  puts("Error al abrir el archivo para escribir.");
  exit(-1);
}
```

El prototipo de `fopen()` se encuentra en el archivo `stdio.h` y es el siguiente:

```
FILE* fopen (const char* nombre_archivo, const char* modo);
```

7.1.4. Modos de apertura de un archivo

Al abrir el archivo `fopen()` espera como segundo argumento el modo de tratar el archivo. Fundamentalmente se establece si el archivo es para leer, para escribir o para añadir; y si es de texto o binario. Los modos básicos se expresan en esta 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, pierden los datos).
"a+"	Abre el archivo para escribir/leer al final. Si no existe es como w+.

En estos modos no se ha establecido el tipo del archivo, texto o binario. Siempre hay una opción por defecto y, aunque depende del compilador utilizado, suele ser modo texto. Para no depender del entorno es mejor indicar si es de texto o binario. Se utiliza la letra `t` para modo texto, la letra `b` para modo binario como último carácter de la cadena `modo` (también se puede escribir como carácter intermedio).

Por consiguiente, los modos de *abrir un archivo de texto:*

```
"rt", "wt", "at", "r+t", "w+t", "a+t".
```

y los modos de *abrir un archivo binario:*

```
"rb", "wb", "ab", "r+b", "w+b", "a+b".
```

Ejemplo 7.5

El archivo de texto LICENCIA.EST existe y se desea leer para realizar un cierto proceso y escribir datos resultantes en al archivo binario RESUMEN.REC. Las operaciones de apertura son:

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

FILE *pf1, *pf2;
char org[] = "C:\LICENCIA.EST";
char dst[] = "C:\RESUMEN.REC";

pf1 = fopen(org, "rt");
pf2 = fopen(dst, "wb");
if (pf1 == NULL || pf2 == NULL)
{
  puts("Error al abrir los archivos.");
  exit(1);
}
```

7.1.5. Macros NULL y EOF

Las funciones de biblioteca que devuelven un puntero (`strcpy()`, `fopen()`...) especifican que si no pueden realizar la operación (generalmente si se produce un error) devuelven `NULL`. Ésta es una macro definida en varios archivos de cabecera, entre los que se encuentran `stdio.h` y `stdlib.h`.

Las funciones de biblioteca de E/S de archivos, generalmente empiezan por `f` de `file`, tienen especificado que son de tipo entero de tal forma que si la operación falla devuelven `EOF`, también devuelven `EOF` para indicar que se ha leído el fin de archivo. Esta macro está definida en `stdio.h`. El siguiente segmento de código lee del flujo estándar de entrada hasta el final del archivo:

```
int c;
while ((c=getchar()) != EOF)
{
  ...
}
```

7.1.6. Cierre de archivos

Los archivos en C trabajan con una memoria intermedia o buffer. La entrada y salida de datos se almacena en ese buffer, «volcándose» cuando está lleno. Al terminar la ejecución del programa podrá ocurrir que haya datos en el buffer; si no se «volcasen» en el archivo quedaría éste sin las últimas actualizaciones. Siempre que se termina de procesar un archivo, se termina la ejecución del programa, los archivos abiertos hay que cerrarlos para que entre otras acciones se «vuelque» el buffer.

La función `fclose(punteroFile)` cierra el archivo asociado al `punteroFile`, devuelve `EOF` si ha habido un error al cerrar. El prototipo de la función se encuentra en `stdio.h` y es:

```
int fclose(FILE* pf);
```

7.2. LECTURA Y ESCRITURA DE ARCHIVOS DE TEXTO

La biblioteca C proporciona diversas funciones para escribir datos en el archivo a través del puntero a `FILE` asociado. Las funciones de entrada y de salida de archivos tienen mucho parecido con las funciones utilizadas de entrada y salida para los flujos `stdin` (teclado) y `stdout` (pantalla): `printf()`, `scanf()`, `getchar()`, `putchar()`, `gets()` y `puts()`; todas ellas tienen una versión para archivos que empieza por la letra *f*; así, se dispone de `fprintf()`, `fscanf()`, `fputs()`, `fgets()`.

7.2.1. Escribir caracteres: `putc()`, `fputc()`

Para escribir carácter a carácter en un archivo de texto se invoca a alguna de estas dos funciones, `putc()` y `fputc()`. Ambas funciones son idénticas, escriben un carácter en el archivo asociado con el puntero a `FILE`; devuelven el carácter escrito, o bien `EOF` si no puede ser escrito. El formato de llamada:

```
putc(c, puntero_archivo);
fputc(c, puntero_archivo);
```

siendo `c` el carácter a escribir.

Ejemplo 7.6

Se desea crear un archivo SALIDA.TXT con los caracteres introducidos por teclado.

Una vez abierto el archivo, un bucle lee carácter a carácter y `mientras` no sea fin de archivo (macro `EOF`) se escribe en el archivo asociado al puntero `FILE`.

```
#include <stdio.h>

void main()
{
  int c;
  FILE* pf;
  char *salida = "\\SALIDA.TXT";

  if ((pf = fopen(salida,"wt") == NULL)
  {
    puts("ERROR EN LA OPERACIÓN DE APERTURA");
    exit(-1);
  }

  while ((c = getchar()) != EOF)
  {
    putc(c, pf);
  }
  fclose(pf);
}
```

En el programa se puede utilizar `fputc(c, pf)` en vez de `putc(c, pf)`. El prototipo de ambas funciones se encuentra en `stdio.h`, es el siguiente:

```
int putc(int c, FILE* pf);
int fputc(int c, FILE* pf);
```

7.2.2. Lectura de caracteres: funciones `getc()`, `fgetc()`

Las dos funciones `getc()` y `fgetc()` son iguales, tienen el mismo formato y la misma funcionalidad; pueden considerarse recíprocas de `putc()` y `fputc()`. Estas nuevas funciones, `getc()` y `fgetc()`, leen un carácter (el siguiente carácter) del archivo asociado al puntero a `FILE`. Devuelven el carácter leído o `EOF` si es fin de archivo (o si ha habido un error).

Formato de llamada:

```
getc(puntero_archivo);
fgetc(puntero_archivo);
```

Para procesar un archivo secuencial en su totalidad se escribe un bucle *mientras no sea fin de archivo* (macro `EOF`). El siguiente segmento de código lee todo el archivo carácter a carácter, si el carácter leído es el de de fin de línea, salta a la línea siguiente.

```
FILE* pf;
char *nombre = "\\SALIDA.TXT";

if ((pf = fopen(nombre,"rt")) == NULL)
{
  puts("ERROR EN LA OPERACIÓN DE APERTURA");
  exit(-1);
}

while ((c = getc(pf)) != EOF)
{
  if (c == '\n')
  {
    n++;
  }
  else
    putchar(c);
}
```

Los prototipos de ambas funciones se encuentran en `stdio.h` y son:

```
int getc(FILE* pf);
int fgetc(FILE* pf);
```

7.2.3. Escribir y leer cadenas de caracteres: funciones `fputs()` y `fgets()`

Las funciones `fputs()` y `fgets()` escriben o leen una cadena de caracteres en el archivo asociado. La función `fputs()` escribe una cadena de caracteres, devuelve `EOF` si no ha podido escribir la cadena, un valor no negativo si la escritura es correcta; el formato de llamada es:

```
fputs(cadena, punteroFile);
```

La función `fgets()` lee una cadena de caracteres del archivo; termina la captación de la cadena cuando lee el carácter de fin de línea, o bien cuando ha leído `n-1` caracteres, siendo `n` un argumento entero de la función. La función devuelve un puntero a la cadena devuelta, o `NULL` si ha habido un error. El formato de llamada es:

```
fgets(cadena, n, punteroFile);
```

Ejercicio 7.1

El archivo CARTAS.DAT contiene un texto al que se le quiere añadir nuevas líneas desde el archivo PRIMERO.DAT. Se considera que la longitud mínima de una línea ha de ser 30 caracteres.

El problema se resuelve abriendo el primer archivo en modo añadir (`"a"`), el segundo archivo en modo lectura (`"r"`). Las líneas se leen invocando a la función `fgets()`, si cumplen la condición de longitud se escriben en el archivo `CARTAS`. Al tener que realizar un proceso completo del archivo, se realizan iteraciones *mientras no sea fin de archivo*.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MX 121
#define MN 30

void main()
{
  FILE *in, *out;
  char nom1[]= "\\CARTAS.DAT";
  char nom2[]= "\\PRIMERO.DAT";
  char cad[MX];

  in = fopen(nom2,"rt");
  out= fopen(nom1,"at");
  if (in==NULL || out==NULL)
  {
    puts("Error al abrir archivos.");
    exit(-1);
  }

  while (fgets(cad,MX,in)) /*itera hasta que devuelve puntero NULL*/
  {
    if (strlen(cad) >= MN)
      fputs(cad,out);
    else
      puts(cad);
  }

  fclose(in);
  fclose(out);

}
```

Los prototipos de ambas funciones están en `stdio.h` y son:

```
int fputs(char* cad, FILE* pf);
char* fgets(char* cad, int n, FILE* pf);
```

7.2.4. Entrada y salida de datos de diferente tipo: `fprintf()` y `fscanf()`

Las funciones `printf()` y `scanf()` permiten escribir o leer variables cualquier tipo de dato estándar, los códigos de formato `(%d, %f ...)` indican a C la transformación que debe de realizar con la secuencia de caracteres (conversión a entero ...). La misma funcionalidad tiene `fprintf()` y `fscanf()` con los flujos (archivos asociados) a que se aplican. Estas dos funciones tienen como primer argumento el puntero (apuntador) a `FILE` asociado al archivo de texto.

Ejercicio 7.2

Se desea crear el archivo de texto PERSONAS.DAT de tal forma que cada línea contenga un registro con los datos de una persona con los campos nombre, fecha de nacimiento (dia(nn), mes(nn), año(nnnn) mes en ascii).

En la estructura `Persona` se declaran los campos correspondientes. Se define una función que devuelve una estructura persona leída del teclado. El mes en caracteres se obtiene de una función que tiene como entrada el número de mes y devuelve una cadena con el mes que le corresponde. Los campos de la estructura se escriben en el archivo llamando a `fprintf()`.

```
#include <malloc.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
   /* declaración de tipo global estructura */
typedef struct {
  char* nm;
  int dia;
  int ms;
  int aa;
  char mes[11];
}Persona;

void entrada(Persona* p);
char* mes_asci(short n);

void main()
{
  FILE *pff;
  char nf[]= "\\PERSONS.DAT";
  char r = 'S';

  if ((pff = fopen(nf,"wt")) == NULL)
  {
    puts("Error al abrir archivos. ");
    exit(-1);
  }

  while (toupper(r) == 'S')
  {
    Persona pt;
    entrada(&pt);
```

```
    printf("%s %d-%d-%d %s\n",pt.nm,pt.dia,pt.ms,pt.aa,pt.mes);
    fprintf(pff,"%s %d-%d-%d %s\n",pt.nm,pt.dia,pt.ms,pt.aa,pt.mes);
    printf("Otro registro?: "); scanf("%c%*c",&r);
  }

  fclose(pff);
}

void entrada(Persona* p)
{
  char bf[81];

  printf("Nombre: "); gets(bf);
  p->nm =(char*)malloc((strlen(bf)+1)*sizeof(char));
  strcpy(p->nm,bf);
  printf("Fecha de nacimiento(dd mm aaaa): ");
  scanf("%d %d %d%*c",&p->dia,&p->ms,&p->aa);
  printf("\n %s\n",mes_asci(p->ms));
  strcpy(p->mes,mes_asci(p->ms));
}

char* mes_asci(short n)
{
  static char *mes[12]= {
                "Enero", "Febrero", "Marzo", "Abril",
                "mayo", "Junio", "Julio", "Agosto",
                "Septiembre", "Octubre", "Noviembre", "Diciembre"};

  if (n >= 1 && n <= 12)
    return mes[n-1];
  else
    return "Error mes";
}
```

Los prototipos de ambas funciones están en `stdio.h`, y son:

```
int fprintf(FILE* pf,const char* formato,. . .);
int fscanf(FILE* pf,const char* formato,. . .);
```

7.2.5. Fin de archivo, `feof()` y `rewind()`

Diversas funciones de lectura de caracteres devuelven `EOF` cuando leen el carácter de fin de archivo. Con dicho valor, que es una macro definida en `stdio.h`, ha sido posible formar bucles para leer un archivo completo. La función `feof()` realiza el cometido anterior, devuelve un valor distinto de `0` `(true)` cuando se lee el carácter de fin de archivo, en caso contrario devuelve `0 (false)`.

Una vez que se alcanza el fin de un archivo, nuevas llamadas a `feof()` siguen devolviendo un valor distinto de cero (`true`). Con la función `rewind()` se sitúa el puntero (apuntador) del archivo al inicio de éste. El formato de llamada es:

```
rewind(punteroFile)
```

El prototipo de cada una de las funciones se encuentra en el archivo `stdio.h` y es el siguiente:

```
int feof(FILE* pf);
void rewind(FILE*pf);
```

7.3. ARCHIVOS BINARIOS EN C

Para abrir un archivo en modo binario hay que especificar la opción `b` en el `modo`. Los archivos binarios son secuencias de 0 y 1 en vez de las secuencias de caracteres de los archivos de texto. Una de las características de los archivos binarios es que optimizan la memoria ocupada por un archivo, sobre todo con campos numéricos. Así, almacenar en modo binario un entero supone una ocupación de 2 bytes o 4 bytes (depende del sistema), y un número real 4 bytes o 8 bytes; en modo texto; primero se convierte el valor numérico en una cadena de dígitos, `(%6d, %8.2f ...)`, y después se escribe en el archivo. La mayor eficiencia de los archivos binarios se contrapone con el hecho de que su lectura se tiene que hacer en modo binario y que sólo se pueden visualizar desde el entorno de un programa C.

Los modos para abrir un archivo binario son los mismos que para abrir un archivo de texto, sustituyendo la `t` por `b`:

```
"rb", "wb", "ab", "r+b", "w+b", "a+b"
```

La biblioteca de C proporciona dos funciones especialmente útiles para el proceso de entrada y salida de archivos binarios con buffer, son `fread()` y `fwrite()`.

7.3.1. Escribir y leer un buffer, funciones `fwrite()` y `fread()`

La función `fwrite()` escribe un buffer de cualquier tipo de dato en un archivo binario. El formato de llamada:

```
fwrite(direccionBuffer, tamaño, n, punteroFile);
```

Para leer de un archivo `n` bloques de bytes y almacenar lo leído en un buffer se invoca a la función `fread()`. El número de bytes de cada bloque (tamaño) se pasa como parámetro, al igual que el número `n` de bloques y la dirección del buffer o variable donde se almacena. El formato de llamada:

```
fread(direccionBuffer, tamaño, n, punteroFile);
```

La función devuelve el número de bloques que lee, y debe coincidir con `n`. El prototipo de ambas funciones está en `stdio.h`:

```
size_t fwrite(const void *ptr, size_t tam, size_t n, FILE *pf);
size_t fread(void *ptr, size_t tam, size_t n, FILE *pf);
```

El tipo `size_t` está definido en `stdio.h` y es un tipo `int`.

Ejemplo 7.7

Abrir un archivo en modo binario para escribir en el mismo. Iterativamente se escriben números reales de doble precisión. El buffer de datos que se escribe se corresponde con una variable de tipo `double`; `x`, *el tamaño se obtiene con el operador* `sizeof`.

```
FILE *fd;
double x;

fd = fopen("reales.num", "wb");

for (x = 0.5; x > 0.01;)
{
  fwrite(&x, sizeof(double), 1, fd);
  x = pow(x,2.);
}
```

Ejemplo 7.8

Abrir el archivo creado en el Ejemplo 7.7 en modo binario para lectura. El archivo se lee hasta el fin del archivo; cada lectura es de un número real que se acumula en la variable `s`.

```
FILE *fd;
double x,s=0.0;

if((fd = fopen("reales.num","rb")) == NULL)
  exit(-1);
while (!feof(fd))
{
  fread(&x, sizeof(double), 1, fd);
  s += x;
}
```

Los archivos binarios están especialmente indicados para guardar registros, estructuras en C. El método habitual es que se escriba una a una en el archivo asociado al puntero y de igual forma se *lean luego estos archivos.*

En el Ejemplo 7.9 se crea un archivo binario de puntos en el espacio.

Ejemplo 7.9

Un punto en el espacio tridimensional se caracteriza por tres componentes reales. Almacenar en un archivo un conjunto de `n` *puntos de coordenadas positivas.*

La primera operación a realizar es abrir el archivo para escritura. A continuación, requerir los *puntos que se desean almacenar y llamar a la función* `fwrite()` *para escribir en el archivo.* El punto en el espacio se representa mediante una estructura, el tamaño del buffer es justamente el que tiene la estructura, que se determina con el operador `sizeof`.

```
#include <stdio.h>
#define POSITIVO(x,y,z) (x>0.0 && y>0.0 && z>0.0) ? 1 : 0
```

```
typedef struct
{
  float x,y, z;
}Punto3D;

void main()
{
  Punto3D p;
  char *nom ="C:\PUNTOS.DAT";
  FILE *pp;

  if ((pp = fopen(nom,"wb")) == NULL)
  {
    puts("\nError en la operación de abrir archivo.");
    exit(-1);
  }

  puts("\n Puntos de coordenadas positivas, acaba con negativo");
  do {
    scanf("%f %f %f",&p.x, &p.y, &p.z);
    if (POSITIVO(p.x,p.y,p.z))
    {
      fwrite(&p, sizeof(Punto3D), 1, pp);
    }
  }while (POSITIVO(p.x,p.y,p.z));

  fclose(pp);
  return 0;
}
```

7.3.2. Funciones para acceso aleatorio

El *acceso directo —aleatorio—* a los datos de un archivo se hace mediante su posición; es decir, el lugar relativo que ocupan. Este tipo de acceso tiene la ventaja de que se pueden leer y escribir registros en cualquier orden y posición y son muy rápidos de acceso a la información que contienen. El principal inconveniente que tiene la organización directa es la necesidad de programar la relación existente entre el contenido de un registro y la posición que ocupan.

Las funciones `fseek()` y `ftell()` se usan principalmente para el acceso directo a archivos en C. Éstas consideran el archivo como una secuencia de bytes, el número de bytes es el índice del archivo. Según se van leyendo o escribiendo registros o datos en el archivo, el programa mantiene a través de un puntero la posición actual. Con la llamada a la función `ftell()` se obtiene el valor de dicha posición. La llamada a `fseek()` permite cambiar la posición del puntero al archivo a una dirección determinada.

Con la función `fseek()` se puede tratar un archivo en C como un array, que es una estructura de datos de acceso aleatorio; `fseek()` sitúa el puntero del archivo en una posición aleatoria, dependiendo del desplazamiento y el origen relativo que se pasan como argumentos. El formato para llamar a `fseek()`:

```
fseek(punteroFile, desplazamiento, origen);
```

`desplazamiento`: *número de bytes a mover; tienen que ser de tipo* `long`.
`origen`: *posición desde la cual se cuenta el número de bytes a mover.*

El argumento `origen` puede tener tres valores representados por sendas macros:

0 (*macro* `SEEK_SET`): *Cuenta desde el inicio del archivo.*
1 (*macro* `SEEK_CUR`): *Cuenta desde la posición actual del puntero al archivo.*
2 (*macro* `SEEK_END`): *Cuenta desde el final del archivo.*

La función `fseek()` devuelve un valor entero, distinto de cero si se comete un error en su ejecución; cero si no hay error.

La posición actual del archivo se puede obtener llamando a la función `ftell()` y pasando un puntero al archivo como argumento. La función devuelve la posición como un número de bytes (en entero largo: `long int`) desde el inicio del archivo (byte 0). Para llamar a la función se pasa como argumento el puntero a `FILE`.

El prototipo de ambas funciones se encuentra en `stdio.h`:

```
int fseek(FILE *pf,long dsplz,int origen);
long int ftell(FILE *pf);
```

7.4. ORDENACIÓN EXTERNA: MÉTODOS DE ORDENACIÓN

Los algoritmos de ordenación estudiados hasta ahora utilizan arrays para contener los elementos a ordenar, por lo que es necesario que la memoria interna tenga capacidad suficiente. Para ordenar secuencias grandes de elementos, que posiblemente no pueden almacenarse en memoria, se aplican los *algoritmos de ordenación externa*. La ordenación externa está ligada con los archivos y los dispositivos en que se encuentran; al leer el archivo para realizar la ordenación el tiempo de lectura de los registros es notablemente mayor que el tiempo que se tarda en realizar las operaciones de ordenación.

Un archivo está formado por una secuencia de `n` elementos, cada elemento es un registro. Cada registro $R(i)$ tiene asociado una clave $K(i)$, generalmente es un campo del registro. El archivo está ordenado respecto a la clave si:

```
∀ i < j ⇒ K(i) < K(j)
```

En la ordenación interna los registros están en todo momento en memoria principal, salvo cuando termina el proceso que se vuelven a almacenar en el dispositivo externo. La memoria principal es limitada y el número de registros de los que puede constar un archivo externo es muy elevado, y como consecuencia no caber en memoria.

La ordenación de los registros de un archivo mediante archivos auxiliares se denomina **ordenación externa.** Los distintos algoritmos de ordenación externa utilizan el esquema general de separación y fusión o mezcla. Por *separación* se entiende la distribución de secuencias de registros ordenados en varios archivos; por *fusión* se entiende la mezcla de dos o más secuencias ordenadas en una única secuencia ordenada. Variaciones de este esquema general dan lugar a diferentes algoritmos de ordenación externa.

A tener en cuenta

El tiempo de un algoritmo de ordenación de registros de un archivo, *ordenación externa*, depende notablemente del dispositivo de almacenamiento. Los algoritmos utilizan el esquema general de *separación* y *mezcla*, y consideran sólo el tratamiento secuencial.

7.5. MÉTODO DE MEZCLA DIRECTA (MEZCLA SIMPLE)

Es el método más simple de ordenación externa utiliza el esquema iterativo de separar secuencias y mezclarlas. Se trabaja con el archivo original y dos archivos auxiliares. El proceso consiste en:

1. Separar los registros individuales del archivo original *O* en dos archivos *F1* y *F2*.
2. Mezclar los archivos *F1* y *F2* combinando registros aislados (según sus claves) y formando pares ordenados que son escritos en el archivo *O*.
3. Separar pares de registros del archivo original *O* en dos archivos *F1* y *F2*.
4. Mezclar *F1* y *F2* combinando pares de registros y formando cuádruplos ordenados que son escritos en el archivo *O*.
5. Se repiten los pasos de separación y mezcla, combinando cuádruplos para formar óctuplos ordenados. En cada paso de separación y mezcla se duplica el tamaño de las subsecuencias mezcladas, así hasta que la longitud de la subsecuencia sea la que tiene el archivo y en ese momento el archivo original *O* está ordenado.

Ejemplo 7.10

Un archivo está formado por registros que tienen un campo clave de tipo entero; suponiendo estas claves enteras:

```
34 23 12 59 73 44 8 19 28 51
```

Se van a realizar los pasos que sigue el algoritmo de mezcla directa para ordenar la secuencia. Se considera el archivo *O* como el original, *F1* y *F2* archivos auxiliares.

Pasada 1

Separación:

```
F1: 34 12 73  8 28
F2: 23 59 44 19 51
```

Mezcla formando duplos ordenados:

```
O: 23 34 12 59 44 73 8 19 28 51
```

Pasada 2

Separación:

```
F1: 23 34 44 73 28 51
F2: 12 59  8 19
```

Mezcla formando cuádruplos ordenados:

```
O: 12 23 34 59 8 19 44 73 28 51
```

Pasada 3

Separación:

```
F1: 12 23 34 59 28 51
F2:  8 19 44 73
```

Mezcla formando óctuplos ordenados:

```
O: 8 12 19 23 34 44 59 73 28 51
```

Pasada 4

Separación:

```
F1:  8 12 19 23 34 44 59 73
F2: 28 51
```

Mezcla con la que ya se obtiene el archivo ordenado:

```
O: 8 12 19 23 28 34 44 51 59 73
```

En el ejemplo han sido necesarias cuatro pasadas, cada pasada constituye una fase de separación y otra de mezcla.

Después de `i` pasadas se tiene el archivo `O` con subsecuencias ordenadas de longitud 2^i, si $2^i \geq n$, siendo `n` el número de registros, el archivo estará ordenado. El número de pasadas se obtiene tomando logaritmos, $i \geq \log_2 n$, `[log n]` pasadas serán suficientes. Cada pasada escribe el total `n` de registros, por lo que el número total de movimientos es `O(n log n)`.

El tiempo de las comparaciones realizadas en la fase de fusión es insignificante respecto a las operaciones de movimiento de registros en los archivos externos; por esta razón no resulta interesante analizar el número de comparaciones.

7.5.1. Codificación del algoritmo de mezcla directa

El algoritmo se basa en dos rutinas: `distribuir()` y `mezclar()`. La primera *separa* secuencias de registros del archivo original en los dos archivos auxiliares. La segunda *mezcla* secuencias de los dos archivos auxiliares y la escribe en el archivo original. En cada pasada, llamada a `distribuir()` y `mezclar()`, se consigue una secuencia del doble de longitud de registros ordenados. El algoritmo termina cuando la longitud de la secuencia alcanza al número de registros.

A continuación se codifica un programa que implementa las dos funciones mencionadas; además, genera un archivo de registros con valores del campo clave que son números aleatorios.

```
#include <stdio.h>
#include <stdlib.h>
typedef int TipoClave;

typedef struct
{
  TipoClave clave;

} Registro;

void mezclaDirecta(FILE* f);
void distribuir(FILE* f, FILE* f1, FILE* f2, int lonSec, int numReg);
void subSecuencia(FILE* f, FILE* t, int longSec );
void mezclar(FILE* f1, FILE* f2, FILE* f, int* lonSec, int numReg);
int cuentaReg(FILE* f);

void escribir(FILE* f);
#define N 716
```

```
void main()
{
  int j;
  FILE* f;
  Registro r;

  f = fopen("fileorg","wt");
  randomize();

  for (j = 1; j <= N; j++)
  {
    r.clave = random(999)+1;
    fprintf(f,"%d ",r.clave);
  }
  fclose(f);

  mezclaDirecta(f);

  f = fopen("fileorg","rt");
  escribir(f);

}

int cuentaReg(FILE* f)
{
  int k = 0;
  Registro r;

  f = fopen("fileorg","rt");
  while (!feof(f) )
  {
    if (fscanf(f,"%d",&r.clave) != EOF)
    {
      k++;
    }
  }

  printf("\n Número de registros: %d\n", k);
  return k;
}

/* La función mezclaDirecta() recibe el archivo que ha sido
   abierto en modo lectura.
*/
void mezclaDirecta(FILE *f)
{
  int longSec;
  int numReg;
  FILE * f1, * f2;

  numReg = cuentaReg(f);
  longSec = 1;
```

```
  while (longSec < numReg)
  {
    distribuir(f, f1, f2, longSec, numReg);
    mezclar(f1, f2, f, &longSec, numReg);
  }
}

void distribuir(FILE* f, FILE* f1, FILE* f2, int lonSec, int numReg)
{
  int numSec, resto, i;

  numSec = numReg/(2*lonSec);
  resto = numReg%(2*lonSec);

  f = fopen("fileorg","rt");
  f1 = fopen("fileAux1","wt");
  f2 = fopen("fileAux2","wt");

  for (i = 1; i <= numSec; i++)
  {
    subSecuencia(f, f1, lonSec);
    subSecuencia(f, f2, lonSec);
  }
  /*
    Se procesa el resto de registros del archivo
  */
  if (resto > lonSec)
    resto -= lonSec;
  else
  {
    lonSec = resto;
    resto = 0;
  }
  subSecuencia(f, f1, lonSec);
  subSecuencia(f, f2, resto);

  fclose(f1); fclose(f2);fclose(f);

}

void subSecuencia(FILE* f, FILE* t, int longSec)
{
  Registro r;
  int j;

  for (j = 1; j <= longSec; j++)
  {
    fscanf(f, "%d", &r.clave);
    fprintf(t,"%d ",r.clave);
  }
}
```

```
void mezclar(FILE* f1, FILE* f2, FILE* f, int* lonSec, int numReg)
{
  int numSec, resto, s, i, j, k, n1, n2;
  Registro r1, r2;

  numSec = numReg/(2*(*lonSec));      /* número de subsecuencias */
  resto = numReg%(2*(*lonSec));

  f = fopen("fileorg","wt");
  f1 = fopen("fileAux1","rt");
  f2 = fopen("fileAux2","rt");

  fscanf(f1, "%d", &r1.clave);
  fscanf(f2, "%d", &r2.clave);

  for (s = 1; s <= numSec+1; s++)
  {
    n1 = n2 = (*lonSec);
    if (s == numSec+1)
      if (resto > (*lonSec))
        n2 = resto - (*lonSec);
      else
      {
        n1 = resto;
        n2 = 0;
      }
    }

    i = j = 1;
    while (i <= n1 && j <= n2)
    {
      Registro d;
      if (r1.clave < r2.clave)
      {
        d = r1;
        fscanf(f1, "%d", &r1.clave);
        i++;
      }
      else
      {
        d = r2;
        fscanf(f2, "%d", &r2.clave);
        j++;
      }
      fprintf(f, "%d ", d.clave);
    }
  /*
    Los registros no procesados se escriben directamente
  */
    for (k = i; k <= n1; k++)
```

```
        {
          fprintf(f, "%d ", r1.clave);
          fscanf(f1, "%d", &r1.clave);
        }

        for (k = j; k <= n2; k++)
        {
          fprintf(f, "%d ", r2.clave);
          fscanf(f2, "%d", &r2.clave);
        }
      }

      (*lonSec) *= 2;
      fclose (f);fclose(f1);fclose(f2);
    }

    void escribir(FILE* f)
    {
      int j;
      Registro r;

      j = 0;
      while (!feof(f) && j < N )
      {
        j++;

        if (fscanf(f,"%d",&r.clave) != EOF)
        {
          printf("%d ",r.clave);

        }
      }
      printf(" ---\n ");

    }
```

7.6. FUSIÓN NATURAL

Este método de ordenación introduce una pequeña variación en cuanto a la longitud de las secuencias de registros del método de mezcla directa. En el método de mezcla directa las secuencias de *registros tienen longitudes que son múltiplos de dos:* `1, 2, 4, 8, 16, ...` *de tal forma que el* número de *pasadas* a realizar es fijo, dependiente del número de registros. La mezcla directa no tiene en cuenta la existencia, *de manera natural*, de secuencias más largas ya ordenadas que también pueden mezclarse y dar lugar a otra secuencia ordenada.

Para recordar

El método de ordenación externa *fusión natural*, en todo momento, distribuye secuencias ordenadas lo más largas posibles y mezcla secuencias ordenadas lo más largas posibles.

7.6.1. Algoritmo de la fusión natural

La característica básica de este método es la mezcla de secuencias ordenadas máximas, o simplemente tramo máximo. Una secuencia ordenada a_i ... a_j es tal que:

$a_k \leq a_{k+1}$ *para* k = i ... j-1
$a_{i-1} > a_i$
$a_j > a_{j+1}$

Así por ejemplo, en esta lista de claves enteras, 4 9 11 5 8 12 9 17 18 21 26 18, los tramos máximos que se encuentran: 4 9 11; **5** 8 12; **9** 17 18 21 26; **18**. La ruptura de un tramo ocurre cuando la clave actual es menor que la clave anterior.

El método de fusión natural funde tramos máximos en lugar de secuencias de tamaño fijo y predeterminado. Esto hace que se optimice el número de pasadas a realizar, disminuyendo éstas en la fusión natural. Los tramos tienen la propiedad de que si se tiene dos listas de *n* tramos cada una y se mezclan, se produce una lista de *n* tramos exactamente. Por ello, el número total de tramos se divide por dos en cada pasada del algoritmo de ordenación de fusión natural.

Ejemplo 7.11

Un archivo está formado por registros que tienen un campo clave de tipo entero; suponiendo estas claves enteras:

```
17 31 5 59 33 41 43 67 11 23 29 47 3 7 71 2 19 57 37 61
```

se van a realizar los pasos que sigue el algoritmo de fusión natural para ordenar la secuencia. Se considera el archivo *O* como el original, *F1* y *F2* archivos auxiliares.

Los tramos máximos de la lista de claves se separan por el carácter ':

```
17 31' 5 59' 33 41 43 67' 11 23 29 47' 3 7 71' 2 19 57' 37 61
```

Pasada 1

Separación:

```
F1: 17 31 33 41 43 67'3 7 71' 37 61
F2: 5 59' 11 23 29 47' 2 19 57
```

Se puede observar que de manera natural, al distribuirse por tramos, la secuencia 17 31 se ha expandido junto a 33 41 43 67 y han formado una única secuencia ordenada.

Mezcla o fusión de tramos:

```
O: 5 17 31 33 41 43 59 67' 3 7 11 23 29 47 71' 2 19 37 57 61
```

Pasada 2

Separación:

```
F1: 5 17 31 33 41 43 59 67' 2 19 37 57 61
F2: 3 7 11 23 29 47 71
```

Mezcla o fusión de tramos:

```
O: 3 5 7 11 17 23 29 31 33 41 43 47 59 67 71' 2 19 37 57 61
```

Pasada 3

Separación:

```
F1: 3  5  7 11 17 23 29 31 33 41 43 47 59 67 71
F2: 2 19 37 57 61
```

Mezcla o fusión de tramos máximos:

```
O: 2 3 5 7 11 17 19 23 29 31 33 37 41 43 47 57 59 61 67 71
```

La lista ya está ordenada, la longitud del tramo máximo es igual al número de registros del archivo.

En el ejemplo han sido necesarias tres pasadas, cada pasada constituye una fase de separación y otra de mezcla o fusión.

El algoritmo, al igual que en la mezcla directa, se basa en dos rutinas: `separarNatural()` y `mezclaNatural()`. La primera *separa* tramos máximos del archivo original en los dos archivos auxiliares. La segunda *mezcla* tramos máximos de los dos archivos auxiliares y la escribe en el archivo original. El algoritmo termina cuando hay un único tramo, entonces el archivo está ordenado.

```
Ordenación MezclaNatural
inicio
  repetir
    separarNatural(f, f1, f2)
    mezclaNatural(f, f1, f2, numeroTramos)
  hasta numerosTramos = 1
fin
```

7.7. MEZCLA EQUILIBRADA MÚLTIPLE

La eficiencia de los métodos de ordenación externa es directamente proporcional al número de pasadas. Para aumentar la eficiencia hay que reducir el número de pasadas. Así, se observa que el método de *fusión natural* reduce el número de pasadas respecto a la mezcla directa, en ambos métodos se utilizan dos archivos auxiliares.

Otra forma de reducir el número de *pasadas* es incrementar el número de archivos auxiliares. Supóngase que se tienen `w` tramos distribuidos equitativamente en `m` archivos. La primera *pasada* mezcla los `w` tramos y da lugar a `w/m` tramos. En la siguiente *pasada*, la mezcla de los `w/m` tramos da lugar a w/m^2 tramos, en la siguiente se reducen a w/m^3 y después de `i` *pasadas* quedarán w/m^i tramos. Según esto, el número de pasadas necesarias para ordenar un archivo de `n` registros y `n` tramos iniciales, mediante mezcla de *m-uples* tramos es $[\log_m n]$, como cada pasada realiza `n` operaciones de movimiento con los registros la eficiencia es $O(n \log_m n)$. La mejora obtenida en cuanto a la disminución de las pasadas necesarias para la ordenación hace que los movimientos o transferencias de cada registro sea $\log_2 m$ veces menos.

Sucesión del número de tramos suponiendo tantos tramos iniciales como registros:

$$n,\ n/m,\ n/m^2,\ n/m^3 \quad \ldots \quad n/m^t = 1$$

Tomando logaritmos en base `m` se calcula el número de tramos `t`:

$$n = m^t\ ;\ \log_m n = \log_m m^t \Rightarrow t = \log_m n$$

7.7.1. Algoritmo 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. De esta forma el proceso de mezcla se realiza en una sola fase en vez de las dos fases (separación, fusión) de la mezcla directa o natural. Los pasos que sigue el algoritmo son:

1. Distribuir registros del archivo original por tramos en los `m/2` primeros archivos auxiliares. A continuación, éstos se consideran archivos de entrada.
2. Mezclar tramos de los `m/2` archivos de entrada y escribirlos consecutivamente en los `m/2` archivos de salida.
3. Cambiar la finalidad de los archivos, los de entrada pasan a ser de salida y viceversa; repetir a partir del segundo paso hasta que quede un único tramo; entonces la secuencia está ordenada.

7.7.2. Tipos de datos para la mezcla equilibrada múltiple

La principal variación, en cuanto a las variables que se utilizan, está en que los archivos auxiliares se agrupan en un array de punteros a `FILE`. Los registros del archivo a ordenar tienen un campo clave ordinal, respecto al cual se realiza la ordenación. La constante, macro `N`, contiene el número de archivos auxiliares, y `N2` el índice mitad (se ha tenido en cuenta que en C un array se indexa con base cero).

```
#define N 6
#define N2 (N/2 -1)

FILE* f[N];
```

La variable `f[]` representa a los archivos auxiliares; de modo alternativo, la primera y la segunda mitad irán cambiando su cometido, entrada o salida.

7.7.3. Cambio de finalidad de un archivo: *entrada* ↔ *salida*

La forma de cambiar la finalidad de los archivos (*entrada* ↔ *salida*) va a ser mediante una tabla de correspondencia entre índices de archivo. De tal forma que en vez de acceder a un archivo por el índice del array se accede por la tabla, la cual cambia alternativamente los índices de los archivos y así pasan alternativamente de ser de entrada a ser de salida.

`int c[N];` *Tabla de índices de archivo.*

Inicialmente `c[i] = i ∀i ε 0 .. N-1`.

Por consiguiente, los archivos de entrada serán:

```
f[c[0]], f[c[1]],..., f[c[N2]];
```

y los archivos de salida son la otra mitad:

```
f[c[N2+1]], f[c[N2+2]], ... f[c[N-1]]
```

Para realizar el cambio de archivo de entrada por archivo de salida hay que intercambiar los valores de las dos mitades de la tabla de correspondencia:

```
c[0] ↔ c[N2+1]
c[1] ↔ c[N2+2]
  . . .
c[N2] ↔ c[N-1]
```

En definitiva, con la tabla `c[]` siempre se accede de igual forma a los archivos, lo que cambia son los índices que contiene `c[]`.

Al mezclar tramos de los archivos de entrada no se alcanza el fin de tramo en todos los archivos al mismo tiempo. Un tramo termina cuando es fin de archivo, o bien la siguiente clave es menor que la actual, en cualquier caso el archivo que le corresponde ha de quedar inactivo. Para despreocuparse de si el archivo está activo o no, se utiliza un vector cuyas posiciones indican si el archivo *correspondiente está activo. En otra tabla de correspondencia,* `cd[]`, *sólo para archivos de entrada* tiene en todo momento los índices de los archivos de entrada activos (en los que no se ha alcanzado el fin de fichero).

7.7.4. Control del número de tramos

El primer paso es el de realizar la distribución inicial de los tramos del archivo original en los archivos de entrada, a la vez determinar el número de tramos. En todo momento es importante conocer el número de tramos, cuando quede un sólo tramo el archivo estará ordenado y éste será `f[c[0]]`.

En la ejecución del método de ordenación llega un momento en el que el número de tramos a mezclar va a ser menor que el número de archivos de entrada. La variable `k1` contiene el número de archivos de entrada, cuando el número de tramos `t` sea mayor o igual que la mitad de los archivos, el valor de `k1` será justamente la mitad; en caso de ser `t` menor que dicha mitad, `k1` será `t` y cuando `t` sea `1`, `k1` será `1` y el archivo `f[c[0]]` estará ordenado.

7.7.5. Codificación del algoritmo de mezcla equilibrada múltiple

El archivo origen, `fo`, y los archivos auxiliares `f[]` están definidos como variables globales. El proceso de mezcla de un tramo de cada uno de los `m/2` archivos de entrada, empieza leyendo de *cada uno de los archivos un registro y guardándolo en un array de registros* `rs[]`. *Esta actividad se* hace en la función `leerItems()`. La mezcla selecciona repetidamente el registro menor, la función `minimo()` se encarga de esta tarea y tiene en cuenta que los registros del array `rs[]` se correspondan con archivos activos. Cada vez que se selecciona un registro, se lee el siguiente registro en el archivo del que procede el registro seleccionado, a no ser que haya acabado el tramo correspondiente del archivo; así hasta que se terminan de mezclar todos los tramos. Precisamente, la función `fintramos()` determina justamente el hecho de que todos los tramos que se están mezclando hayan acabado.

La codificación que se presenta a continuación se realiza con 6 archivos auxiliares. Se escribe *un programa completo; en primer lugar, crea el archivo original, y a continuación llama a la función* que implementa el método de ordenación *mezcla equilibrada múltiple.*

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

```
#define N 6
#define N2 (N/2-1)
#define NR 360

#define MAXT 32767

char * nomf[N] = {"ar1", "ar2", "ar3", "ar4","ar5","ar6"};

typedef int TipoClave;
typedef struct
{
  TipoClave clave;
} Registro;

Registro rs[N2];
FILE* fo;
FILE* f[N];
int c[N], cd[N];
int actvs[N];

void mezclaEqMple();
int distribuir();
void leerItems(Registro r[], int nf, int cd[]);
int minimo(Registro r[], int activo[], int n);
int finDeTramos(int activo[], int n);
int cuentaReg(FILE* f);
void escribir(FILE* f);
void fichero();

void main()
{
  fichero();
  mezclaEqMple();
  fo = fopen(nomf[c[0]],"rt");
  printf("\n \t Archivo ordenado \n");
  escribir(fo);
}

void mezclaEqMple()
{
  int i, j, k, kl, t;
  TipoClave anterior;

    /* distribución inicial de los tramos del archivo origen */
  t = distribuir();
  for (i = 0; i < N; i++)
  {
    c[i] = i;
  }
```

```
/* bucle externo, termina cuando número de tramos, t, sea 1 */

do {
  k1 = (t <= N2) ? t-1 : N2;
  k1 += 1; /* número de archivos, tiene en cuenta índice 0 */
  k = k1;

  for (i = 0; i < k1; i++)
  {
    if ((f[c[i]] = fopen(nomf[c[i]],"rt"))==NULL)
    {
      printf("\n Error al abrir %d %d %s en modo rt \n ",
                  i, c[i], nomf[c[i]]);

      exit(-1);
    }
    cd[i] = c[i];
  }

  t = 0;                          /* Número de tramos mezclados */
  j = N2 + 1;                     /* índice de archivo de salida */
  for (i = j; i < N; i++)
  {
    if ((f[c[i]] = fopen(nomf[c[i]],"wt"))==NULL)
    {
      printf("\n Error al abrir %d %d %s en modo wt \n ",
                                  i, c[i], nomf[c[i]]);
      exit(-1);
    }

  }

  leerItems(rs, k1, cd);

  while (k1 > 0)
  {
    t++;                          /* mezcla de un nuevo tramo */
    for (i = 0; i < k1; i++) actvs[i] = 1;

    while (!finDeTramos(actvs, k))
    {
      int mn;

      mn = minimo(rs, actvs, k);
      fprintf(f[c[j]], "%d ", rs[mn].clave);
      anterior = rs[mn].clave;
      if (fscanf(f[cd[mn]],"%d", &rs[mn].clave) != EOF)
      {
        if (anterior > rs[mn].clave)              /* fin de tramo */
          actvs[mn] = 0;
      }
      else                        /* es fin de archivo y de tramo */
```

```
            {
              k1--;
              actvs[mn] = 0;
              fclose(f[cd[mn]]);        /* archivo cd[mn] inactivo */
              cd[mn] = cd[k1];          /*cambia contenido de k1 por mn */
              rs[mn] = rs[k1];
              actvs[mn] = actvs[k1];
              actvs[k1] =0;             /* no se accede a la posición k1 */
            }
          }

        j = (j < N-1) ? j+1 : N2+1;     /* siguiente archivo salida */
      }

      for (i = N2+1; i < N; i++)
      {
        fclose(f[c[i]]);

      }
    /* Cambio en la finalidad de los archivos */
    for (i = 0; i <= N2; i++)
    {
      int a;
      a = c[i];
      c[i] = c[i+N2+1];
      c[i+N2+1] = a;
    }

  } while (t > 1);

  }

  int distribuir()
  {

      int anterior, j, nt;
      Registro r;

      fo = fopen("fileorg","rt");
      if (fo == NULL)
  {
    puts("Error al abrir archivo origen. \n");
    exit(-1);
  }

  for (j = 0; j <= N2; j++)
  {
    f[j] = fopen(nomf[j], "wt");
    if (f[j] == NULL)
    {
```

```
    printf("Error al abrir modo salida archivo %s \n", nomf[j]);
    exit(-1);
  }
}
anterior = -MAXT;
r.clave = anterior + 1;
j = 0;      /* índice de archivo de salida */
nt = 0;
while (!feof(fo))
{
  /* copia un tramo de fo al archivo j */

    while (anterior <= r.clave &&!feof(fo))
    {
      if (fscanf(fo,"%d",&r.clave) != EOF)
        if (anterior <= r.clave)
        {
          fprintf(f[j],"%d ", r.clave);
          anterior = r.clave;
        }
    }

    nt++; /* se ha formado un nuevo tramo */
    j = (j < N2) ? j+1 : 0; /* siguiente archivo */

    if (!feof(fo)) /* empieza nuevo tramo */
    {
      fprintf(f[j],"%d ", r.clave);
      anterior = r.clave;
    }
  }
  fclose(fo);

  for (j = 0; j <= N2; j++)
    fclose(f[j]);

  printf("\n tramos %d \n",nt);
return nt;
}

void leerItems(Registro r[], int nf, int cd[])
{
  int n;
  for (n = 0; n < nf; n++)
    fscanf(f[cd[n]],"%d", &r[n].clave);
}

int minimo(Registro r[], int activo[], int n)
{
```

```
 int i, indice;
 TipoClave m;

 i = indice = 0;
 m = MAXT;

 for ( ;i < n; i++)
 {
   if (activo[i] && r[i].clave < m)
   {
     m = r[i].clave;
     indice = i;
   }
 }
 return indice;
}

int finDeTramos(int activo[], int n)
{
 int k, s = 1;
 for(k = 0; k < n; k++)
 {
   if (activo[k])s = 0;
 }
 return s;
}

void escribir(FILE* f)
{
 int j;
 Registro r;
 j = 0;
 while (!feof(f))
 {
   j++;

   if (fscanf(f,"%d",&r.clave) != EOF)
   {
     printf("%d ",r.clave);

   }
 }

}

void fichero()
{
 int j;
 Registro r;

 fo = fopen("fileorg","wt");
 randomize();
```

```
    for (j = 1; j <= NR; j++)
    {
      r.clave = random(999)+1;
      fprintf(fo,"%d ",r.clave);
    }
    fclose(fo);
  }
```

7.8. MÉTODO POLIFÁSICO DE ORDENACIÓN EXTERNA

La estrategia seguida en el método de mezcla equilibrada múltiple emplea `2m` archivos para ordenar `n` registros, de tal forma que si los `n` registros están distribuidos en `m` tramos en una *pasada* (distribución + mezcla) quedan ordenados. La utilización de `2m` archivos puede hacer que el número de *éstos sea demasiado elevado y no todas las aplicaciones puedan soportarlo. La mezcla equilibrada* múltiple puede mejorarse consiguiendo ordenar `m` tramos con sólo `m+1` archivos; para ello hay que abandonar la idea rígida de *pasada* cuya finalidad de los archivos de entrada no cambia hasta que se leen todos ellos.

El *método polifásico* utiliza `m` archivos auxiliares para ordenar `n` registros de un archivo. La característica que marca la diferencia de este método respecto a los otros es que en todo momento se mezclan registros desde los `m-1` archivos al archivo `m`. En el momento que en uno de los archivos de entrada se alcanza su final hay un cambio de cometido, pasa a ser considerado como archivo de salida, el archivo que en ese momento era de salida pasa a ser de entrada y la mezcla de tramos continúa. La sucesión de pasadas continúa hasta alcanzar el archivo ordenado.

Cabe recordar la propiedad base de todos los métodos de mezcla: *la mezcla de k tramos de los archivos de entrada se transforma en k tramos en el archivo de salida.*

A tener en cuenta

La mezcla polifásica se caracteriza por realizar una mezcla continua de tramos. De tal forma que si se utilizan `m` archivos auxiliares, en un momento dado uno de ellos es archivo de salida y los otros `m-1` archivos son de entrada. Durante el proceso, cuando se alcanza el *registro de fin de archivo* en un archivo de entrada, éste pasa a ser de salida, el anterior archivo de salida pasa a ser de entrada y la mezcla continúa. El método tiene la dificultad de que el número de tramos iniciales debe pertenecer a una sucesión de números que depende de `m`.

7.8.1. Mezcla polifásica con `m = 3` archivos

A continuación se muestra un ejemplo en el que se supone un archivo original de `55` tramos. El número de archivos auxiliares es de `m = 3`, por lo que en todo momento, `2` archivos son de entrada y un tercero de salida.

El método distribuye inicialmente los tramos, de manera no uniforme, en los archivos `F1`, `F2`. De los `55` tramos se sitúan `34` en `F1` y `21` en `F2`; el archivo `F3` es, inicialmente, de salida. Empieza la mezcla, y `21` tramos de `F1` se fusionan con los `21` tramos de `F2` dando lugar a `21` tramos en `F3`. La situación en este momento es que en `F1` quedan `13` tramos; en `F2` se ha alcanzado el fin de archivo y en `F3` hay `21` tramos; la mezcla continúa entre `F1` y `F3`, siendo `F2` el archivo de salida. Ahora se mezclan `13` tramos de `F1` con `13` tramos de `F3`, dando lugar a `13` tramos que se escriben en `F2`, y se alcanza el fin de archivo de `F1`. En el archivo `F2` hay `13` tramos y en el `F3` quedan `8` tramos,

continúa la mezcla con F1 como archivo de salida. En la nueva pasada se mezclan 8 tramos que se escriben en F1, quedan 5 tramos en F2 y en F3 ninguno ya que se ha alcanzado el fin de fichero. El proceso continúa hasta que queda un único tramo y el fichero ha quedado ordenado. La Figura 7.1 muestra las sucesivas pasadas y los tramos de cada archivo hasta que termina la ordenación.

Tramos		después de cada pasada						
iniciales	F1+F2	F1+F3	F2+F3	F1+F2	F1+F3	F2+F3	F1+F2	F1+F3
F1 34	13	0	8	3	0	2	1	0
F2 21	0	13	5	0	3	1	0	1
F3 0	21	8	0	5	2	0	1	0

Figura 7.1. Tramos en cada archivo después de cada pasada en la mezcla polifásica con 3 archivos.

Es evidente que se ha partido de una distribución óptima. Además, si se escribe la sucesión de tramos mezclados a partir de la distribución inicial: 21, 13, 8, 5, 3, 2, 1, 1, se comprueba que es justamente la sucesión de los números de Fibonacci:

$$f_{i+1} = f_i + f_{i-1} \quad \forall i \geq 1 \;, \quad f_1 = 1,\; f_0 = 0$$

Entonces, si el número de tramos iniciales f_k es tal que es un número de Fibonacci, la mejor forma de hacer la distribución inicial es según la sucesión de Fibonacci, f_{k-1} y f_{k-2}. Sin embargo, no será así en la mayoría de las ordenaciones y en esos casos se recurre a escribir tramos ficticios para conseguir un número de Fibonacci.

Distribución para m = 4 archivos

Se puede extender el proceso de mezcla polifásica a un número mayor de archivos. Así, si por ejemplo se parte de un archivo origen con 31 tramos y se utilizan m = 4 archivos para la mezcla polifásica, la distribución inicial y los tramos de cada archivo se muestran en la Figura 7.2.

Tramos		después de cada pasada			
iniciales	F1+F2+F3	F1+F2+F4	F1+F3+F4	F2+F3+F4	F1+F2+F3
F1 13	6	2	0	1	0
F2 11	4	0	2	1	0
F3 7	0	4	2	1	0
F4 0	7	3	1	0	1

Figura 7.2. Tramos en cada archivo en la mezcla polifásica con 4 archivos.

La Figura 7.3 muestra en forma tabular la distribución perfecta de los tramos en los archivos para cada pasada, pero numeradas en orden inverso.

L	$t_1^{L)}$	$t_2^{L)}$	$t_3^{L)}$	
5	13	11	7	*(total 31 tramos)*
4	7	6	4	
3	4	3	2	
2	2	2	1	
1	1	1	1	
0	1	0	0	

Figura 7.3. Tramos que intervienen en cada pasada para 3 archivos.

Observando la tabla de la Figura 7.3 se encuentran ciertas relaciones entre t_1, t_2, t_3 en un nivel:

$$t_3^{L+1)} = t_1^{L)}$$
$$t_2^{L+1)} = t_1^{L)} + t_3^{L)}$$
$$t_1^{L+1)} = t_1^{L)} + t_2^{L)}$$
$$\forall\ L > 0 \quad y \quad t_1^{0)} = 1\ ,\ t_2^{0)} = 0,\ +\ t_3^{0)} = 0$$

Estas relaciones van a ser muy útiles para encontrar la distribución inicial ideal cuando se quiera ordenar un archivo de un número arbitrario de tramos.

Además, haciendo el cambio de variable de f_i por $t_i^{L)}$, se tiene la sucesión de los números de Fibonacci de orden 2:

$$f_{i+1} = f_i + f_{i-1} + f_{i-2} \quad \forall i \geq 2\ ,\ , \quad f_2 = 1,\ f_1 = 0,\ f_0 = 0$$

Para recordar

En general, la sucesión de *números de Fibonacci de orden k* tiene la expresión:

$$f_{i+1} = f_i + f_{i-1} + \ldots + f_{i-k+1} \quad \forall i \geq k-1 \quad \text{tal que,}$$

$$f_{k-1} = 1,\ f_{k-2} = 0\ \ldots,\ f_0 = 0$$

7.8.2. Distribución inicial de tramos

En los dos ejemplos expuestos se ha aplicado el método polifásico de manera ideal, la distribución inicial de tramos era perfecta en función del número de archivos auxiliares y de la correspondiente sucesión de Fibonacci. Las fórmulas recurrentes que permiten obtener el número de tramos para cada pasada L, en el supuesto de utilizar m archivos son:

$$t_{m-1}^{L+1)} = t_1^{L)}$$
$$t_{m-2}^{L+1)} = t_1^{L)} + t_{m-1}^{L)}$$
$$\ldots$$
$$t_2^{L+1)} = t_1^{L)} + t_3^{L)}$$
$$t_1^{L+1)} = t_1^{L)} + t_2^{L)}$$
$$\forall\ L > 0 \quad y \quad t_0^{L)} = 1,\ t_0^{i)} = 0 \quad \forall i \leq m-1$$

Con estas relaciones puede conocerse de antemano el número de tramos necesarios para aplicar *el método polifásico con m archivos. Es evidente que no siempre el número de tramos iniciales del* archivo a ordenar va a coincidir con ese número ideal, ¿qué hacer en ese caso?

La respuesta es sencilla: los tramos necesarios para completar la distribución perfecta simulan su existencia, como tramos vacíos o tramos ficticios. ¿Cómo tratar los tramos ficticios? La selección de un tramo ficticio de un archivo `i` es, simplemente, ignorar el archivo, y por consiguiente desechar dicho archivo de la mezcla del tramo correspondiente. En el caso de que para los `m - 1` archivos el tramo sea ficticio, no habrá que hacer ninguna operación, simplemente considerar un tramo ficticio en el archivo de salida. La distribución inicial ha de repartir los tramos ficticios lo más uniformemente posible en los `m - 1` archivos.

Para tener en cuenta estas consideraciones relativas a los tramos, se utilizan dos arrays, `a[]` y `d[]`. El primero, `a[]`, contiene los números de tramos que ha de tener cada archivo de entrada en una pasada dada; el segundo, `d[]`, guarda el número de tramos ficticios que tiene cada archivo.

El proceso se inicia de «abajo arriba»; por ejemplo, si se observa la tabla de la Figura 7.3, se empieza asignando al vector `a[]` el numero de tramos que se corresponde con la última mezcla, siempre `(1,1,... ,1)`. A la vez, a `d[]` también `(1,1,... ,1)`, de tal forma que cada vez que se copie un tramo, del archivo orígen, en el archivo `i`, se decrementa `d[i]`, y así con cada uno de los `m-1` archivos.

Si no se ha terminado el archivo original, se determina de nuevo el número de tramos, pero del siguiente nivel y los tramos que hay que añadir a cada archivo para alcanzar ese segundo nivel, según las relaciones recurrentes. Los tramos ficticios de cada archivo, `d[i]`, coincidirán con el número de tramos que se deben añadir, para que posteriormente se vaya decrementando `d[i]` según se vayan añadiendo tramos al archivo.

Por ejemplo, sea un archivo original con 28 tramos y se van a utilizar `m = 4` archivos auxiliares, la distribución inicial de tramos en los `m - 1` archivos se realiza consecutivamente, de nivel a nivel de acuerdo a la tabla de la Figura 7.3: el primer nivel tiene tres tramos, `a(1,1,1)`, por lo que se distribuyen 3 tramos; el segundo nivel `a(2,2,1)`, los tramos a añadir `(2,2,1)-(1,1,1) = (1,1,0)` y el vector `d[]` será `(1,1,0)`, por lo que se distribuyen 2 nuevos tramos. Una vez completados los tramos para este nivel, `d[]`, se habrá quedado `(0,0,0)`, se pasa al siguiente nivel. Los tramos para el tercer nivel son `a(4,3,2)`, por lo que hay que añadir `(4,3,2)-(2,2,1) = (2,1,1)`, y los tramos ficticios `d(2,1,1)`, se distribuyen `4` nuevos tramos; ya van distribuidos `3 + 2 + 4 = 9` tramos.

Así sucesivamente, se calcula el número de tramos de cada nuevo nivel y los tramos que hay que añadir para obtener esa cantidad; esos tramos serán, inicialmente, los tramos ficticios; según se reparten tramos, se decrementa la posición correspondiente del vector `d[]`. La Figura 7.4 muestra los distintos valores que van tomando `a[]` y `d[]` en cada nivel, hasta completar los 28 tramos.

tramos, nivel, añadir

Tramos iniciales	número de nivel 2	3	4	5
a[] (1,1,1)	(2,2,1)	(4,3,2)	(7,6,4)	**(13,11,7)**
añadir	(1,1,0)	(2,1,1)	(3,3,2)	(6,5,3)
d[] (1,1,1)	(1,1,0)	(2,1,1)	(3,3,2)	(6,5,3)
(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	**(2,1,0)**

Figura 7.4. Distribución inicial con 28 tramos y `m = 4` archivos.

En el supuesto planteado, una vez completado el cuarto nivel se han repartido `17` tramos, se pasa al siguiente; ahora `a = (13,11,7)`, por lo que hay que añadir `(13,11,7)-(7,6,4) = (6,5,3)`, y los tramos ficticios se ponen siempre igual al número de tramos que se deben añadir: `d(6,5,3)`; quedan `28 - 17 = 11` tramos que se distribuyen uniformemente en los archivos, al final `d = (2,1,0)`, que son los tramos ficticios a considerar en los archivos correspondientes.

Cuantos más tramos haya más niveles se alcanzan; en cualquier caso, una vez terminado el proceso de distribución del archivo original, se tiene en `a[]` el número de tramos para ese nivel, y en `d[]` el número de tramos ficticios que tiene cada archivo; necesario después para el proceso de mezcla.

7.8.3. Algoritmo de la mezcla

Una vez realizada la distribución inicial en los `m-1` primeros archivos auxiliares, se repite el proceso de mezcla hasta que queda un único tramo. El número de pasadas que va a realizar es conocido de antemano, ha sido calculado en el proceso de distribución, y es el número de niveles alcanzado; de igual forma se conoce la distribución inicial de los tramos: se encuentra en `a[]`, y el número inicial de tramos ficticios, `d[]`.

En cada nivel, en cada pasada, el número de tramos que se mezclan es el mínimo de los elementos de `a[]` siempre se encuentra en `a[m-1]` siendo `m` el número de archivos. Al escribir tramos *mezclados en el archivo de salida, puede ocurrir que todos los tramos sean ficticios por lo que se* contabiliza en la posición que le corresponde en `d[]`.

En el algoritmo de mezcla que a continuación se escribe, `c[]` es la tabla de correspondencia de los `m - 1` archivos, de tal forma que en `c[]` se encuentran los índices de los archivos que se están mezclando. Como puede haber archivos no activos, debido a los tramos ficticios, `cd[]` es la tabla de correspondencia con los índices de los archivos activos, `k` es el número de archivos activos, y `actv[]`, un vector lógico, indexado por `c[i]` que está a `true` si el archivo que le corresponde está activo. Las rotaciones de los archivos que se realizan según acaba un archivo en la mezcla se hacen con la tabla `c[]`. La mezcla de tramos se realiza de igual forma que en la mezcla múltiple: con un *procedimiento de lectura de items de todos los archivos y la selección del mínimo.*

Algoritmo

```
desde i <-1 hasta m-1 hacer
  c[i] <- i
fin_desde

repetir
  { Mezcla de los archivos c[1]...c[m-1] }
  z <- a[m-1] {número de tramos a mezclar en esta pasada}
  d[m] <- 0 {inicializa tramos ficticios en archivo de salida }

  <Preparar para escribir F[c[m]]>

  repetir
      { Mezcla de un tramo de los m-1 archivos }
    k <- 0  { número de ficheros activos}
    desde i <- 1 hasta m-1 hacer
      si d[i] > 0 entonces {es un tramo ficticio}
        d[i] <- d[i]-1
      sino {es un tramo real}
        k <- k+1
        cd[k] <- c[i]
      fin_si
```

```
        fin_desde
        si k > 0 entonces
          <Mezclar los tramos de los k archivos>
        sino {Todos los tramos son ficticios}
          d[m] <- d[m]+1   {tramo ficticio en el archivo destino}
        fin_si

        z <- z-1 {tramo ya mezclado}
      hasta z = 0
        { Se han mezclado todos los tramos, el archivo de salida
          pasa a ser de entrada }

      <Preparar para lectura F[c[m]]>

      <Rotar los archivos en la tabla de correspondencia c[]>

      <Calcular los números de fibonacci, a[i], del siguiente nivel>

      Nivel <- Nivel-1
    hasta Nivel = 0
  fin
```

7.8.4. Mezcla polifásica *versus* mezcla múltiple

Las principales diferencias de la ordenación polifásica respecto a la mezcla equilibrada múltiple son:

1. En cada pasada hay un solo archivo destino (salida), en vez de los `m/2` que necesita la mezcla equilibrada múltiple.
2. La finalidad de los archivos (*entrada, salida*) cambia en cada pasada, rotando los índices de los archivos. Esto se controla mediante una tabla de correspondencia de índices de archivo. En la mezcla múltiple siempre se intercambian `m/2` archivos origen (entrada) por `m/2` archivos destino (salida).
3. El número de archivos origen (de entrada) varía dependiendo del número del tramo en proceso. Éste se determina en el momento de empezar el proceso de un tramo, a partir del contador d_i de tramos ficticios para cada archivo `i`. Puede ocurrir que $d_i > 0$ para todos los valores de `i, i=1..m - 1` e indica que hay que mezclar `m - 1` tramos ficticios dando lugar a otro tramo ficticio en el archivo destino. Esta situación se refleja incrementando d_m del archivo destino (de salida). No siempre será así, sino que a veces habrá que mezclar un tramo de cada archivo que verifique $d_i == 0$, y disminuir en uno `d` en los demás archivos; de esa forma queda indicada la existencia de un tramo ficticio menos.
4. Ahora el criterio de terminación de una fase radica en el número de tramos a ser mezclados en cada archivo. Puede ocurrir que se alcance el último registro del archivo `m - 1` y ser preciso más mezclas que utilicen tramos ficticios de ese archivo. En la fase de distribución inicial fueron calculados los números de fibonacci de cada nivel, de forma progresiva hasta alcanzar el nivel en el que se agotó el archivo de entrada; ahora, partiendo de los números del último nivel pueden recalcularse hacia atrás.

RESUMEN

La ordenación de archivos se denomina ordenación externa y requiere algoritmos apropiados. Una manera trivial de realizar la ordenación de un archivo secuencial consiste en copiar los registros a otro archivo de acceso directo, o bien secuencial indexado, usando como clave el campo por el que se desea ordenar.

Si se desea realizar la ordenación de archivos, utilizando solamente como estructura de almacenamiento auxiliar otros archivos secuenciales de formato similar al que se desea ordenar, hay que trabajar usando el esquema de *separación* y *mezcla*.

En el caso del algoritmo de *mezcla simple* se opera con tres archivos análogos: el original y dos archivos auxiliares. El proceso consiste en recorrer el archivo original y copiar secuencias de sucesivos registros en, alternativamente, cada uno de los archivos auxiliares. A continuación se mezclan las secuencias de los archivos y se copia la secuencia resultante en el archivo original. El proceso continúa, de tal forma que en cada pasada la longitud de la secuencia es el doble de la longitud de la pasada anterior. Todo empieza con secuencias de longitud 1, y termina cuando se alcanza una secuencia de longitud igual al número de registros.

El algoritmo de *mezcla natural* también opera con tres archivos; se diferencia en que distribuye secuencias ordenadas, en vez de secuencias de longitud fija.

Se estudian métodos avanzados de ordenación externa, como la *mezcla equilibrada múltiple* y la *mezcla polifásica*. En el primero se utiliza un número par de archivos auxiliares, la mitad de ellos son archivos de entrada y la otra mitad archivos de salida. El segundo se caracteriza por la existencia de una distribución inicial de tramos (según las secuencias de números de Fibonacci), para después realizar una *mezcla continuada* hasta obtener un único tramo ordenado.

La primera parte del capítulo revisa las operaciones fundamentales de la biblioteca estándar de entrada y salida para el tratamiento y manejo de archivos externos.

EJERCICIOS

7.1. Escribir las sentencias necesarias para abrir un archivo de caracteres, cuyo nombre y acceso se introduce por teclado, en modo lectura; en el caso de que el resultado de la operación sea erróneo, abrir el archivo en modo escritura.

7.2. Un archivo contiene enteros positivos y negativos. Utilice la función `fscanf()` para leer el archivo y determinar el número de enteros negativos.

7.3. Un archivo secuencial contiene registros con un campo clave de tipo entero en el rango de `0` a `2777`. Escribir la función `volcado()`, que genere un archivo directo en el número de registro coincida con el campo clave.

7.4. El archivo secuencial `F`, almacena registros con un campo clave de tipo entero. Supóngase que la secuencia de claves que se encuentra en el archivo es la siguiente:

```
14 27 33 5 8 11 23 44 22 31 46 7 8 11 1 99 23 40 6 11 14 17
```

Aplicando el algoritmo de mezcla directa realizar la ordenación del archivo y determinar el número de pasadas necesarias.

7.5. Considerando el mismo archivo del Ejercicio 7.4, aplicar el algoritmo de mezcla natural para ordenar el archivo. Comparar el número de pasadas con las obtenidas en el citado ejercicio.

7.6. Un archivo secuencial `F` contiene registros y se desea ordenar utilizando 4 archivos auxiliares. Suponiendo que la ordenación se hace respecto a un campo de tipo entero y con estos valores:

```
22 11 3 4 11 55 2 98 11 21 4 3 8 12 41 21 42 58 26 19 11 59 37 28 61 72 47
```

Aplicar el algoritmo de mezcla equilibrada múltiple y obtener el número de pasadas necesarias para su ordenación.

7.7. Con el mismo archivo que en el Ejercicio 7.6, y también con m = 4 archivos auxiliares aplicar el algoritmo de mezcla polifásica. Comparar el número de pasadas realizadas con ambos métodos.

PROBLEMAS

7.1. Escribir un programa que compare dos archivos de texto. El programa ha de mostrar las diferencias entre el primer archivo y el segundo, precedidas del número de línea y de columna.

7.2. Un atleta utiliza un pulsómetro para sus entrenamientos. El pulsómetro almacena las pulsaciones cada 15 segundos, durante un tiempo máximo de 2 horas. Escribir un programa para almacenar en un archivo los datos del pulsómetro del atleta, de tal forma que el primer registro contenga la fecha, hora y tiempo en minutos de entrenamiento, a continuación los datos del pulsómetro por parejas: tiempo, pulsaciones.

7.3. Las pruebas de acceso a la Universidad Popular de Mágina, UPMA, constan de 4 pruebas, cada una de las 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.

7.4. Dado el archivo de puntuaciones generado en el Problema 7.3, escribir un programa para ordenar dicho archivo utilizando el método de ordenación externa *de mezcla natural*.

7.5. Supóngase que se dispone del archivo ordenado de puntuaciones de la UPMA (Problema 7.4), y del archivo de la Universidad de Somosierra que tiene el mismo formato y también está ordenado. Escribir un programa que mezcle ordenadamente los dos archivos en un tercero.

7.6. Se dispone de un archivo con los datos de los pobladores de la comarca de Pinilla. Cada persona está representada por un registro con el campo clave primer apellido, y como campo secundario el segundo apellido. Escribir un programa para que tal archivo sea ordenado por el método de mezcla equilibrada múltiple. Utilizar 6 archivos auxiliares.

7.7. 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.

7.8. Escribir un programa para ordenar el archivo que se ha generado en el Problema 7.7. Utilizar el método de ordenación de mezcla polifásica, con m = 5 archivos auxiliares.

PARTE III

ESTRUCTURA DE DATOS FUNDAMENTALES (ABSTRACCIÓN DE DATOS, LISTAS, PILAS, COLAS Y TABLAS HASH)

8

TIPOS ABSTRACTOS DE DATOS Y OBJETOS

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Definir qué es un tipo de dato.
- Distinguir entre variables y tipos de datos.
- Conocer los tipos de datos básicos.
- Conocer tipos de datos estructurados.
- *Especificar los tipos abstractos de datos.*
- Aplicar la funcionalidad de un tipo abstracto de datos independientemente de la implementación.
- Conocer la importancia de los tipos abstractos de datos en la reutilización del software.

CONTENIDO

8.1. El papel (el rol) de la abstracción.
8.2. Modularidad.
8.3. Diseño de módulos.
8.4. Tipos de datos.
8.5. Abstracción en lenguajes de programación.
8.6. Tipos abstractos de datos.
8.7. Especificación de los TAD.
8.8. Tipos abstractos de datos en C.
8.9. Orientación a objetos.
8.10. Reutilización de software.
8.11. Desarrollo tradicional frente al orientado a objetos.

RESUMEN
EJERCICIOS
PROBLEMAS

CONCEPTOS CLAVE

- Tipos de datos y variables.
- Rol.
- Abstracción.
- Modularidad
- Software.
- Interfaz.
- Encapsulación (encapsulamiento).
- Componentes.
- Reutilización.

INTRODUCCIÓN

En este capítulo se examinan los conceptos de *modularidad* y *abstracción de datos*. La modularidad es la posibilidad de dividir una aplicación en piezas más pequeñas llamadas módulos. *Abstracción de datos* es la técnica de inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, facilitar la escritura del programa. La técnica de abstracción de datos es una técnica potente de propósito general que cuando se utiliza adecuadamente, puede *producir programas más cortos, más legibles y flexibles.*

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como `int`, `char` y `float` en C 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 modularización de un programa utiliza la noción de *tipo abstracto de dato* (**TAD**) siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado *TAD*.

8.1. EL PAPEL (EL ROL) DE LA ABSTRACCIÓN

Los programadores han tenido que luchar con el problema de la complejidad durante mucho tiempo desde el nacimiento de la informática; en este capítulo se revisan cuáles han sido los diferentes mecanismos utilizados por los programadores para controlar la complejidad. Entre todos ellos se destaca la *abstracción*. Como describe Wulft: «Los humanos hemos desarrollado una técnica excepcionalmente potente para tratar la complejidad: abstraernos de ella. Incapaces de dominar en su totalidad los objetos complejos, se ignora los detalles no esenciales, tratando en su lugar con el modelo ideal del objeto y centrándonos en el estudio de sus aspectos esenciales».

Para recordar

Abstracción es la capacidad para encapsular y aislar la información, del diseño y ejecución.

La progresión histórica de la abstracción en la programación comienza en los procedimientos y sigue en los módulos, tipos abstractos de datos y objetos.

8.1.1. La abstracción como un proceso natural mental

Las personas normalmente comprenden el mundo construyendo modelos mentales de partes del mismo, tratan de comprender cosas con las que pueden interactuar: un modelo mental es una vista simplificada de cómo funciona, de modo que se pueda interactuar contra ella. En esencia, este proceso de construcción de modelos es igual que el diseño de software, aunque el desarrollo de software es único, ya que produce el modelo que puede ser manipulado por una computadora.

Sin embargo, los modelos mentales deben ser más sencillos que el sistema al cual imitan, o en caso contrario serán inútiles. Por ejemplo, consideremos un mapa como un modelo de su territorio. A fin de ser útil, el mapa debe ser más sencillo que el territorio que modela. Un mapa nos ayuda, ya que abstrae sólo aquellas características del territorio que deseamos modelar. Un mapa de carreteras modela cómo conducir mejor de una posición a otra. Un mapa topográfico modela el contorno de un territorio, quizás para planear un sistema de largos paseos o caminatas.

De igual forma que un mapa debe ser más pequeño significativamente que su territorio e incluye sólo información seleccionada cuidadosamente, así los modelos mentales abstraen esas características de un sistema requerido para nuestra comprensión, mientras ignoran características irrelevantes. Este proceso de *abstracción* es psicológicamente necesario y natural; la abstracción es crucial para comprender este complejo mundo.

La abstracción es esencial para el funcionamiento de una mente humana normal, y es una herramienta muy potente para tratar la complejidad. Considerar por ejemplo el ejercicio mental de memorizar números. Un total de siete dígitos se puede memorizar con más o menos facilidad. Sin embargo, si se agrupan y se denominan números de teléfono, los dígitos individuales se relegan en sus detalles de más bajo nivel, creándose un nivel abstracto y más alto, en el que los siete números se organizan en una única entidad. Utilizando este mecanismo, se pueden memorizar algunos números de teléfonos de modo que la agrupación de diferentes entidades conceptuales es un mecanismo potente al servicio de la abstracción.

8.1.2. Historia de la abstracción del software

La abstracción es la clave para diseñar buen software. En los primeros días de la informática, los programadores enviaban instrucciones binarias a una computadora, manipulando directamente interrupciones en sus paneles frontales. Los nemotécnicos del lenguaje ensamblador eran abstraccio-

nes diseñadas para evitar que los programadores tuvieran que recordar las secuencias de bits que componen las instrucciones de un programa. El siguiente nivel de abstracción se consigue agrupando instrucciones primitivas para formar macroinstrucciones.

Por ejemplo, un conjunto se puede definir por abstracción como una colección no ordenada de elementos en el que no existen duplicados. Utilizando esta definición, se pueden especificar si sus elementos se almacenan en un *array*, una lista enlazada o cualquier otra estructura de datos. Un conjunto de instrucciones realizadas por un usuario se pueden invocar por una macroinstrucción. Una macroinstrucción instruye a la máquina para que realice muchas cosas. Tras los lenguajes de programación ensambladores aparecieron los lenguajes de programación de alto nivel, que supusieron un nuevo nivel de abstracción. Los lenguajes de programación de alto nivel permitieron a los programadores distanciarse de las interioridades arquitectónicas específicas de una máquina dada. Cada instrucción en un lenguaje de alto nivel puede invocar varias instrucciones máquina, dependiendo de la máquina específica donde se compila el programa. Esta abstracción permitía a los programadores escribir software para propósito genérico, sin preocuparse sobre qué máquina corre el programa.

Secuencias de sentencias de lenguajes de alto nivel se pueden agrupar en procedimientos y se invocan por una sentencia. La programación estructurada alienta el uso de abstracciones de control tales como bucles o sentencias **`if-then`** que se han incorporado en lenguajes de alto nivel. Estas sentencias de control permitieron a los programadores abstraer las condiciones comunes para cambiar la secuencia de ejecución.

El proceso de abstracción fue evolucionando desde la aparición de los primeros lenguajes de programación. El método más idóneo para controlar la complejidad de los problemas ha sido aumentar los niveles de abstracción. *Abstraer* un problema supone la capacidad de encapsular y aislar la información de diseño y ejecución. La progresión histórica de la abstracción va desde las estructuras de control, pasando por los procedimientos, los módulos, los tipos abstractos de datos y los objetos.

En las siguientes secciones se describen los mecanismos de abstracción que han conducido a las nuevas tecnologías basadas en los objetos: procedimientos, módulos, tipos abstractos de datos y objetos.

8.1.3. Procedimientos

Los **procedimientos** y las **funciones** fueron uno de los primeros mecanismos de abstracción que se utilizaron ampliamente en lenguajes de programación. Los procedimientos permitían tareas que se ejecutaban rápidamente, o eran ejecutados sólo con ligeras variaciones, que se reunían en una entidad y se reutilizaban, en lugar de duplicar el código varias veces. Por otra parte, el procedimiento proporcionó la primera posibilidad de *ocultación de información*. Un programador podía escribir un procedimiento o conjunto de procedimientos, que se utilizaban por otros programadores. Estos otros programadores no necesitaban conocer con exactitud los detalles de la implementación sólo necesitaban el interfaz necesario. Sin embargo, los procedimientos no resolvían todos los problemas. En particular, no era un mecanismo efectivo para ocultar la información y para resolver el problema que se producía al trabajar múltiples programadores con nombres idénticos.

Para ilustrar el problema, consideremos un programador que debe escribir un conjunto de rutinas para implementar una pila. Siguiendo los criterios clásicos de diseño de software, nuestro programador establece en primer lugar, una interfaz visible a su trabajo, es decir, cuatro rutinas: `insertar`, `quitar`, `pilavacia` y `pilallena`. A continuación implementa los datos rutinas mediante arrays, listas enlazadas, etc. Naturalmente los datos contenidos en la pila no se pueden hacer locales a cualquiera de las cuatro rutinas, ya que se deben compartir por todos. Sin embargo, si las únicas elecciones posibles son variables locales o globales, entonces la pila se debe mantener en variables globales; por el contrario, al ser las variables globales, no existe un método para limitar la accesibilidad o visibilidad de dichas variables. Por ejemplo, si la pila se representa en un array denominado `datospila`, este dato debe ser conocido por otros programadores, que puedan desear

crear variables utilizando el mismo nombre pero relativo a las referidas rutinas. De modo similar las rutinas citadas están reservadas y no se pueden utilizar en otras partes del programa para otros propósitos. En C existe el ámbito local y global; también cabe tener en cuenta que C simplifica los procedimientos en funciones en una sola construcción que es la función. Cualquier ámbito que permite acceso a los cuatro procedimientos debe permitir también el acceso a sus datos comunes. Para resolver este problema, se ha desarrollado un mecanismo de estructuración diferente.

8.1.4. Módulos

Un **módulo** es una técnica que proporciona la posibilidad de dividir sus datos y procedimientos en una *parte privada* —sólo accesible dentro del módulo— y *parte pública* —accesible fuera del módulo—. Los tipos, datos (variables) y procedimientos se pueden definir en cualquier parte.

El criterio a seguir en la construcción de un módulo es que si no se necesita algún tipo de información, no se debe tener acceso a ella. Este criterio es *la ocultación de información*.

Los módulos resuelven algunos problemas, pero no todos los problemas del desarrollo de software. Por ejemplo, los módulos permitirán a nuestros programadores ocultar los detalles de la implementación de su pila, pero ¿qué sucede si otros usuarios desean tener dos o más pilas? Supongamos que un programador ha desarrollado un tipo de dato `Complejo` (representación de un número complejo) y ha definido las operaciones aritméticas sobre números complejos (suma, resta, multiplicación y división); así mismo ha definido rutinas para convertir números convencionales a complejos. Se presenta un problema: sólo puede manipular un número complejo. El sistema de números complejos no será útil con esta restricción, pero es la situación en que se encuentra el programador con módulos simples.

Los módulos proporcionan un método efectivo de ocultación de la información, pero no permiten realizar *instanciación*, que es la capacidad de hacer múltiples copias de las zonas de datos.

8.1.5. Tipos abstractos de datos

Un *tipo abstracto de datos* (**TAD**) es un tipo de dato definido por el programador que se puede manipular de un modo similar a los tipos de datos definidos por el sistema. Al igual que los tipos definidos por el sistema, un tipo *de dato abstracto* corresponde a un conjunto (puede ser de tamaño indefinido) de valores legales de datos y un número de operaciones primitivas que se pueden realizar sobre esos valores. Los usuarios pueden crear variables con valores que están en el rango de valores legales y pueden operar sobre esos valores utilizando las operaciones definidas. Por ejemplo, en el caso de la pila ya citada, se puede definir dicha pila como un tipo abstracto de datos y las operaciones sobre la pila como las únicas operaciones legales que están permitidas para ser realizadas sobre instancias de la pila.

Los módulos se utilizan frecuentemente como una técnica de implementación para tipos abstractos de datos, y el tipo abstracto de datos es un concepto más teórico. Para construir un tipo abstracto de datos se debe poder:

1. Exponer una definición del tipo.
2. Hacer disponible un conjunto de operaciones que se pueden utilizar para manipular instancias de ese tipo.
3. Proteger los datos asociados con el tipo de modo que sólo se puede actuar sobre ellos con las rutinas proporcionadas.
4. Hacer instancia múltiples del tipo.

Los módulos son mecanismos de ocultación de información y no cumplen básicamente más que los tres primeros apartados.

8.1.6. Objetos

Un objeto es sencillamente un tipo abstracto de datos al que se añaden importantes innovaciones en compartición de código y reutilización. Los mecanismos básicos de orientación a objetos son: *objetos, mensajes y métodos, clases e instancias y herencia.*

Una idea fundamental es la comunicación de los objetos a través de *paso de mensajes*. Además de esta idea, se añaden los mecanismos de *herencia y polimorfismo*. La herencia permite diferentes tipos de datos para compartir el mismo código, permitiendo una reducción en el tamaño del código y un incremento en la funcionalidad. El polimorfismo permite que un mismo mensaje pueda actuar sobre objetos diferentes y comportarse de modo distinto.

La *persistencia* se refiere a la permanencia de un objeto, esto es, la cantidad de tiempo para el cual se asigna espacio y permanece accesible en la memoria del computador.

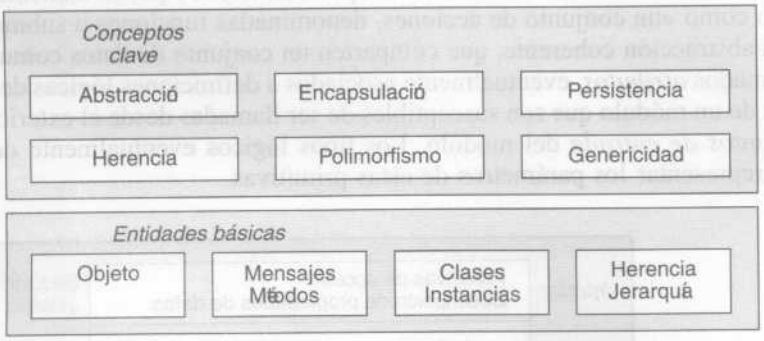


Figura 8.1. Principios básicos de la orientación a objetos.

8.2. MODULARIDAD

La programación modular trata de descomponer un programa en un pequeño número de abstracciones coherentes que pertenecen al dominio del problema y cuya complejidad interna es susceptible de ser enmascarada por la descripción de un interfaz.

Las abstracciones que se desean representar, pueden, en ciertos casos, corresponder a una única acción abstracta y se implementan en general con la noción de *objeto abstracto (o tipo abstracto)* caracterizado en todo instante por:

- Un *estado actual*, definido por un cierto número de atributos.
- Un *conjunto de acciones posibles*.

En consecuencia, la *modularidad* es la posibilidad de subdividir una aplicación en piezas más pequeñas (denominadas *módulos*) cada una de las cuales debe ser tan independiente como sea posible, considerando la aplicación como un todo, así como de las otras piezas de las cuales es una parte. Este principio básico desemboca en el principio básico de *construir programas modulares*. Esto significa que, aproximadamente, ha de subdividir un programa en piezas más pequeñas, o módulos, que son generalmente independientes de cada una de las restantes y se pueden *ensamblar* fácilmente para construir la aplicación completa. En esencia, las abstracciones se implementan en *módulos*, conocidos en la terminología de Booch, como objetos, que agrupan en una sola entidad:

- Un conjunto de datos.
- Un conjunto de operaciones que actúan sobre los datos.

Liskov define la modularización como «el proceso de dividir un programa en módulos que se pueden compilar separadamente, pero que tienen conexiones con otros módulos». Parnas va más

lejos y dice que «las conexiones entre módulos deben seguir el criterio de ocultación de la información: un sistema se debe descomponer de acuerdo al criterio general, de que cada módulo oculta alguna decisión de diseño del resto del sistema; en otras palabras, cada módulo *oculta* un secreto».

Si un programa se descompone (o subdivide en módulos) de modo consistente con el criterio de Parnas —es decir, aplicando el principio de ocultación de la información— se reduce la complejidad de cada módulo que compone la solución. Éstos se constituyen, en cierto modo independientes de los restantes y, por consiguiente, se reduce la necesidad de tomar decisiones globales, operaciones y datos.

8.2.1. La estructura de un módulo

Un módulo se caracteriza fundamentalmente por su *interfaz* y por *su implementación*. Parnas define el módulo como «un conjunto de acciones, denominadas funciones o submódulos, que corresponden a una abstracción coherente, que comparten un conjunto de datos comunes definidos estáticamente llamados *atributos*, eventualmente asociados a definiciones lógicas de tipos». Las acciones o funciones de un módulo que son susceptibles de ser llamadas desde el exterior se denominan *primitivas* o *puntos de entrada* del módulo. Los tipos lógicos eventualmente definidos en la interfaz permiten representar los parámetros de estas primitivas.

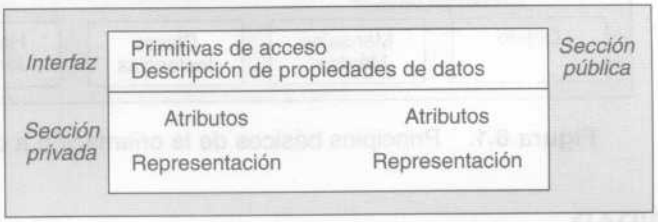


Figura 8.2. Estructura de un módulo.

8.2.2. Reglas de modularización

En primer lugar, un método de diseño debe ayudar al programador a resolver un problema dividiendo el problema en subproblemas más pequeños, que se puedan resolver independientemente unos de otros. Debe, también, ser fácil conectar los diferentes módulos a los restantes, dentro del programa que esté escribiendo.

Cada módulo tiene un significado específico propio y se debe asegurar que cualquier cambio a su implementación no afecte a su exterior (o al menos, lo mínimo). De igual modo los errores posibles, condiciones de límites o frontera, comportamientos erráticos no se propaguen más allá del módulo (o como máximo, a los módulos que estén directamente en contacto con el afectado).

Para obtener módulos con las características anteriores, se deben seguir las siguientes reglas.

Unidades modulares

El lenguaje debe proporcionar estructuras modulares con las cuales se puedan describir las diferentes unidades. De este modo, el lenguaje (y el compilador) puede reconocer un módulo y debe ser capaz de manipular y gobernar su uso, además de las ventajas evidentes relativas a la legibilidad del código resultante. Estas construcciones modulares pueden, como en el caso de los lenguajes orientados a objetos, mostrar características que facilitan la estructura del programa, así como la escritura de programas. En otras palabras, nos referimos a las unidades modulares lingüísticas que en el caso de C pueden se la *archivos de inclusión (.h)* y en C++ se conocen como *clases*.

En C el archivo *.h* es donde se declaran los tipos de datos y la interfaz representada por los prototipos de las funciones; la implementación de las funciones de la interfaz se realiza en el archivo correspondiente *.c*. C++ va más lejos, la construcción del lenguaje *clase* permite declarar los tipos de datos, normalmente parte oculta, y la interfaz, parte pública, de tal forma que el cliente de la *clase* conoce perfectamente cómo requerir una funcionalidad de la *clase* sin poder acceder directamente a los datos.

Interfaces adecuadas

En la estructuración de un programa en unidades es beneficioso que existan *pocas interfaces* y que éstas sean pequeñas. Es conveniente que existan pocos enlaces entre los diferentes módulos en que se descompone un programa. La *interfaz de un módulo* es la parte del módulo (datos, procedimientos, etc.) que es visible fuera del módulo.

Las interfaces deben ser también *pequeñas* (esto es, su tamaño debe ser pequeño con respecto al tamaño de los módulos implicados). De este modo, los módulos están acoplados débilmente; se enlazarán por un número pequeño de llamadas (Fig. 8.3.).

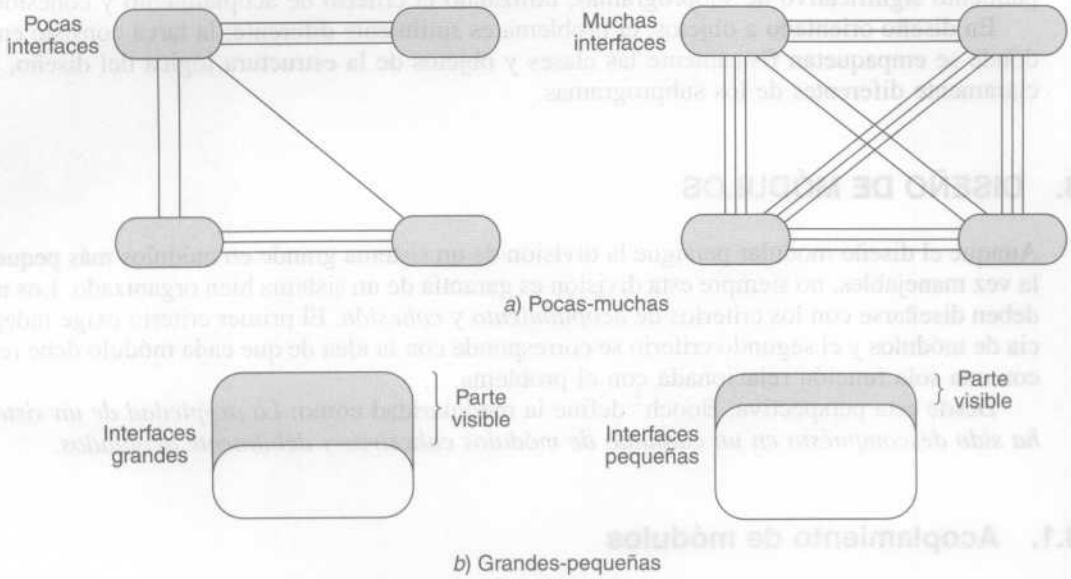


Figura 8.4. Interfaces adecuadas (grandes-pequeñas).

Interfaces explícitas

La interfaz o parte visible externamente de un módulo se debe declarar y describir explícitamente; el programa debe especificar cuáles son los datos y procedimientos que un módulo trata de exportar y cuáles deben permanecer ocultos del exterior. La interfaz debe ser fácilmente legible, tanto para el programador como para el compilador. Es decir, el programador debe comprender cómo funciona el programa y el compilador ha de poder comprobar si el código que accede al módulo se ha escrito correctamente.

Ocultación de la información

Todos los módulos deben seguir el principio de ocultación de la información; cada módulo debe representarse al menos un elemento de diseño (por ejemplo, la estructura de un registro, un algoritmo, una abstracción, etc.).

Otro criterio a tener en cuenta es la subdivisión de un sistema en módulos; es el principio denominado *abierto-cerrado*[1], formulado por Meyer. Este principio entiende que cada módulo se considerará *cerrado* (esto es, terminado y, por consiguiente, útil o activo desde dentro de otros módulos) y al mismo tiempo, debe ser *abierto* (esto es, sometido a cambios y modificaciones). El principio *abierto-cerrado* debe producirse sin tener que reescribir todos los módulos que ya utilizan el módulo que se está modificando.

A tener en cuenta

En lenguajes de programación clásicos, la modularización se centra en los subprogramas (procedimientos, funciones y subrutinas). En lenguajes orientados a objetos, la modularización o partición del problema se resuelve a través de los tipos abstractos de datos.

En diseño estructurado, la modularización —como ya se ha comentado— se centra en el agrupamiento significativo de subprogramas, utilizando el criterio de acoplamiento y cohesión.

En diseño orientado a objetos, el problema es sutilmente diferente: la tarea consiste en decidir dónde se empaquetan físicamente las clases y objetos de la estructura lógica del diseño, que son claramente diferentes de los subprogramas.

8.3. DISEÑO DE MÓDULOS

Aunque el diseño modular persigue la división de un sistema grande en módulos más pequeños y a la vez manejables, no siempre esta división es garantía de un sistema bien organizado. Los módulos deben diseñarse con los criterios de *acoplamiento* y *cohesión*. El primer criterio exige independencia de módulos y el segundo criterio se corresponde con la idea de que cada módulo debe realizarse con una sola función relacionada con el problema.

Desde esta perspectiva, Booch[2] define la modularidad como: *La propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.*

8.3.1. Acoplamiento de módulos

El *acoplamiento* es una medida del grado de interdependencia entre módulos, es decir, el modo en que un módulo está siendo afectado por la *estructura interna* de otro módulo. El grado de acoplamiento se puede utilizar para evaluar la calidad de un diseño de sistema. El objetivo es minimizar el acoplamiento entre módulos, es decir, minimizar su interdependencia, de modo que un módulo sea afectado lo menos posible por la *estructura de otro módulo*. El acoplamiento entre módulos varía en un amplio rango. Por un lado el diseño de un sistema puede tener una jerarquía de módulos totalmente desacoplados. Sin embargo, dado que un sistema debe realizar un conjunto de funciones o tareas de un modo organizado, no puede constar de un conjunto de módulos totalmente desacoplados. En el otro extremo, se tendrá una jerarquía de módulos estrechamente acoplados; es decir, hay un alto grado de dependencia entre cada pareja de módulos del diseño.

---

[1] Sobre el principio abierto-cerrado y su implementación en C++ y Eiffel, se puede consultar la bibliografía de Miguel Katrib. Algunos títulos destacados sobre orientación a objetos son: *Programación Orientada a Objetos a través de C++ y Eiffel.* V Escuela Internacional en Temas Selectos de Computación. Zacatecas, México, 1994 (esta escuela está organizada por la UNAM, México); *Programación Orientada a Objetos en C++.* Infosys, México, 1994; Collections and Iterators in Eiffel, Joop, vol. 6 n.º 7, nov/dic. 1993

[2] Booch, Grady: *Object-Oriented Design with applications.* Benjamin Cummings, 1991, pág. 52.

Tal como define Booch, un sistema modular débilmente acoplado facilita:

1. La sustitución de un módulo por otro, de modo que sólo unos pocos módulos serán afectados por el cambio.
2. El seguimiento de un error y el aislamiento del módulo defectuoso que produce ese error.

Existen varias clases de acoplamiento entre dos módulos (Tabla 8.1). Examinaremos los cinco tipos de acoplamiento, desde el menos deseable (esto es, acoplamiento estrecho o impermeable) al más deseable (esto es, acoplamiento más débil). La fuerza de acoplamiento entre dos módulos está *influenciada por el tipo de conexión, el tipo de comunicación entre ellos y la complejidad global de* su interfaz.

Tabla 8.1. Clasificación del acoplamiento de módulos

Tipo de acoplamiento	Grado de acoplamiento	Grado de mantenibilidad
Por contenido	Alto (fuerte)	Bajo
Común	↓	↓
De control		
Por sellado (estampado)		
Datos		
Sin acoplamiento	Bajo (débil)	Alto

8.3.2. Cohesión de módulos

La **cohesión** es una extensión del concepto de ocultamiento de la información. Dicho de otro modo, la cohesión describe la naturaleza de las interacciones dentro de un módulo software. Este criterio sugiere que un sistema bien modularizado es aquel en el cual los interfaces de los módulos son claros y simples. Un módulo cohesivo ejecuta una tarea sencilla de un procedimiento de software y requiere poca interacción con procedimientos que ejecutan otras partes de un programa. En otras palabras, un módulo cohesivo sólo hace (idealmente) una cosa.

La cohesión y el acoplamiento se miden como un «espectro» que muestra las escalas que siguen los módulos. La Tabla 8.1 muestra la clasificación de acoplamientos de módulos y su grado de acoplamiento. La Tabla 8.2 muestra los grados de cohesión: baja cohesión (no deseable) y alta cohesión (deseable), así como los diferentes tipos de cohesión.

Tabla 8.2. Clasificación de cohesión de módulos

Tipo de cohesión	Grado de cohesión	Grado de mantenimiento
Por coincidencia	Bajo	Bajo
Lógica	↓	↓
Temporal		
Por procedimientos		
Por comunicaciones		
Secuencial		
Funcional		
Informacional	Alto	Alto

Para recordar

Idealmente se buscan módulos altamente cohesivos y débilmente acoplados.

8.4. TIPOS DE DATOS

Todos los lenguajes de programación soportan algún tipo de datos. Por ejemplo, el lenguaje de programación convencional C soporta tipos base tales como enteros, reales y caracteres, así como tipos compuestos tales como *arrays* (vectores y matrices) y *estructuras* (registros).

A tener en cuenta

Un tipo de dato es un conjunto de valores y un conjunto de operaciones definidas sobre esos valores.

Un valor depende de su representación y de la interpretación de la representación, por lo que una definición informal de un tipo de dato es: *Representación + Operaciones*.

Un *tipo de dato* describe un conjunto de objetos con la misma representación. Existen un número de operaciones asociadas con cada tipo. Es posible realizar aritmética sobre tipos de datos enteros y reales, concatenar cadenas o recuperar o modificar el valor de un elemento.

La mayoría de los lenguajes tratan las variables y constantes de un programa como *instancias* de un *tipo de dato*. Un tipo de dato proporciona una descripción de sus instancias que indican al compilador cosas como cuánta memoria se debe asignar para una instancia, cómo interpretar los datos en memoria y qué operaciones son permisibles sobre esos datos. Por ejemplo, cuando se escribe una declaración tal como `float z` en C o C++, se está declarando una instancia denominada *z* del tipo de dato `float`. El tipo de datos `float` indica al compilador que reserve, por ejemplo, 32 bits de memoria, y qué operaciones tales como «sumar» y «multiplicar» están permitidas, mientras que operaciones tales como el «el resto» (módulo) y «desplazamiento de bits» no lo están. Sin embargo, no se necesita escribir la declaración del tipo `float` —el autor de compilador lo hizo por nosotros y se construyen en el compilador—. Los tipos de datos que se construyen en un compilador de este modo, se conocen como *tipos de datos fundamentales (predefinidos)*; por ejemplo, en C y C++ son entre otros: `int`, `char`, `float` y `double`.

Cada lenguaje de programación incorpora una colección de tipos de datos fundamentales, que incluyen normalmente enteros, reales, carácter, etc. Los lenguajes de programación soportan también un número de constructores de tipos incorporados que permiten generar tipos más complejos. Por ejemplo, C soporta registros y arrays.

En lenguajes convencionales tales como C, Pascal..., las operaciones sobre un tipo de dato son composiciones de constructores de tipo y operaciones de tipos base.

A tener en cuenta

El programador no tiene que preocuparse de saber cómo el compilador del lenguaje implementa los tipos de datos predefinidos, simplemente usa los tipos de datos en el programa.

8.5. ABSTRACCIÓN EN LENGUAJES DE PROGRAMACIÓN

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

Desde comienzos del decenio de los sesenta, en que se desarrollaron los primeros lenguajes de programación de alto nivel, ha sido posible utilizar las abstracciones más primitivas de ambas categorías (variables, tipos de datos, procedimientos, control de bucles, etc.). Ambas categorías de abstracciones han producido una gran cantidad de lenguajes de programación no siempre bien definidos.

8.5.1. Abstracciones de control

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

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

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

Abstracción procedimental (por procedimientos)

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

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

Otros mecanismos de abstracción de control

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

8.5.2. Abstracciones de datos

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

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

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

Los tipos de datos son abstracciones y el proceso de construir nuevos tipos se llaman abstracciones de datos. Los nuevos tipos de datos definidos por el usuario se llaman *tipos abstractos de datos*.

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

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

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

Las clases en SIMULA sólo cumplían la primera de las dos condiciones, mientras que otros lenguajes actuales cumplen las dos condiciones: Ada, Modula-2 y C++.

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

1. La representación: elección de las estructuras de datos.
2. Las operaciones: elección de los algoritmos.

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

8.6. TIPOS ABSTRACTOS DE DATOS

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

Para recordar

Un tipo abstracto de datos puede definirse mediante la ecuación:

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

La estructura de un tipo abstracto de dato (*clase*), desde un punto de vista global, se compone de la interfaz y de la implementación (Fig. 8.4).

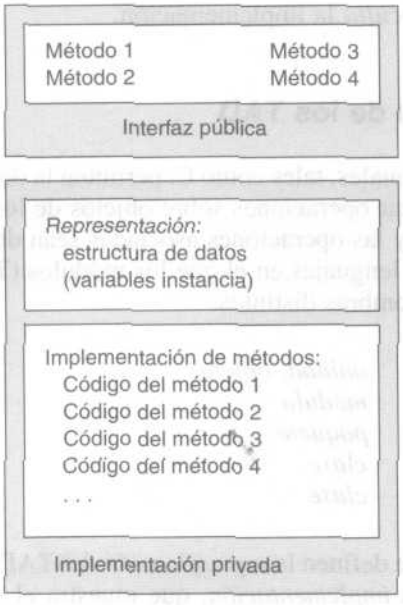


Figura 8.4. Estructura de un tipo abstracto de datos (TAD).

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

8.6.1. Ventajas de los tipos abstractos de datos

Un *tipo abstracto de datos* es un modelo (estructura) con un número de operaciones que afectan a ese modelo. Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

1. Permite una mejor *conceptualización* y *modelización* del mundo real. Mejora la representación y la comprensibilidad. Clarifica los objetos basados en estructuras y comportamientos comunes.
2. Mejora la *robustez* del sistema. Si hay características subyacentes en los lenguajes permiten la especificación del tipo de cada variable, los tipos abstractos de datos permiten la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.
3. Mejora el rendimiento (prestaciones). Para sistemas *tipeados* (tipificados), el conocimiento de los objetos permite la optimización de tiempo de compilación.
4. Separa la implementación de la especificación. Permite la modificación y mejora de la implementación sin afectar a la interfaz pública del tipo abstracto de dato.
5. Permite la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.
6. Recoge mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

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

8.6.2. Implementación de los TAD

Los lenguajes convencionales, tales como C, permiten la definición de nuevos tipos y la declaración de funciones para realizar operaciones sobre objetos de los tipos. Sin embargo, tales lenguajes no permiten que los datos y las operaciones asociadas sean declaradas juntos como una unidad y con un solo nombre. En los lenguajes en el que los módulos (TAD) se pueden implementar como una unidad, éstos reciben nombres distintos:

Turbo Pascal	*unidad, objeto*
Modula-2	*módulo*
Ada	*paquete*
C++	*clase*
Java	*clase*

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

En C no existe como tal una construcción del lenguaje para especificar un TAD. Sin embargo, se puede agrupar el interfaz y la representación de los datos en un archivo de inclusión, *archivo.h*. La implementación de la interfaz, de las funciones se realiza en el correspondiente *archivo.c*. Los detalles de la codificación de las funciones quedan *ocultos* en el *archivo.c*

Las ventajas de los TAD se pueden manifestar en toda su potencia, debido a que las dos partes de los módulos (*especificación* e *implementación*) se pueden compilar por separado mediante la técnica de compilación separada (*separate compilation*).

8.7. ESPECIFICACIÓN DE LOS TAD

Un tipo abstracto de datos es un tipo de datos definido por el usuario que tiene un conjunto de datos y unas operaciones. La especificación de un TAD consta de dos partes, la descripción matemática del conjunto de datos y las operaciones definidas en ciertos elementos de ese conjunto de datos. El objetivo de la especificación es describir el comportamiento del TAD.

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

8.7.1. Especificación informal de un TAD

La especificación informal consta de dos partes:

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

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

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

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

Operación (argumentos). *Descripción funcional*

Como ejemplo se va a especificar el tipo abstracto de datos `Conjunto`:

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

Operaciones, existen numerosas operaciones matemáticas sobre conjuntos, algunas de ellas:

Conjuntovacio.

Crea un conjunto sin elementos.

Añadir(Conjunto, elemento).

Comprueba si el elemento forma parte del conjunto, en caso negativo es añadido. La operación modifica al conjunto.

Retirar(Conjunto, elemento).

En el caso de que el elemento pertenezca al conjunto es eliminado de éste. La operación modifica al conjunto.

Pertenece(Conjunto, elemento).

Verifica si el elemento forma parte del conjunto, en cuyo caso devuelve *cierto*.

Esvacio(Conjunto).

Verifica si el conjunto no tiene elementos conjunto, en cuyo caso devuelve *cierto*.

Cardinal(Conjunto).

Devuelve el número de elementos del conjunto.

Union(Conjunto, Conjunto).

Realiza la operación matemática de la unión de dos conjuntos. La operación devuelve un conjunto con los elementos comunes y no comunes a los dos argumentos.

Así se puede seguir especificando más operaciones sobre conjuntos, todo dependerá de la aplicación que se quiera dar al TAD.

A tener en cuenta

La especificación informal de un TAD tiene como objetivo describir los datos del tipo y las operaciones según la funcionalidad que tienen. No sigue normas rígidas al hacer la especificación, simplemente se indica, de forma comprensible, lo que hace cada operación.

8.7.2. Especificación formal de un TAD

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

El esquema que se sigue para especificar formalmente un TAD consta de una cabecera con el nombre del TAD y los datos:

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

Sigue, a continuación, la *sintaxis* de las operaciones: se listan las operaciones indicando los tipos de los argumentos y el tipo del resultado:

Sintaxis

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

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

Semántica

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

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

A continuación se hace la especificación formal del TAD Conjunto, para formar la expresión resultado se hace uso, si es necesario, de la sentencia alternativa *si-entonces-sino*.

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

Sintaxis

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

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

  Añadir(Añadir(C, e1), e1) ⇒ Añadir(C, e1)
  Añadir(Añadir(C, e1), e2) ⇒ Añadir(Añadir(C, e2), e1)
  Retirar(Conjuntovacio, e1) ⇒ Conjuntovacio
   Retirar(Añadir(C, e1), e2) ⇒ si e1 = e2 entonces Retirar(C,e2)
                                sino Añadir(Retirar(C,e2),e1)
  Pertenece(Conjuntovacio, e1) ⇒ falso
  Pertenece(Añadir(C, e2), e1) ⇒ si e1 = e2 entonces cierto
                                 sino Pertenece(C, e1)
  Esvacio(Conjuntovacio) ⇒ cierto
  Esvacio(Añadir(C, e1)) ⇒ falso
  Cardinal(Conjuntovacio) ⇒ Cero
  Cardinal(Añadir(C, e1)) ⇒ si Pertenece(C,e1) entonces
                             Cardinal(C)
                            sino 1 + Cardinal(C)
  Union(Conjuntovacio,
                  Conjuntovacio) ⇒ Conjuntovacio
  Union(Conjuntovacio,
                  Añadir(C, e1)) ⇒ Añadir(C, e1)
  Union(Añadir(C, e1), D) ⇒ Añadir(Union(C, D), e1)
```

La implementación del tipo abstracto de datos debe incluir dos partes diferenciadas:

- Representación de los datos del tipo abstracto.
- Implementación de las operaciones descritas en la especificación.

8.8. TIPOS ABSTRACTOS DE DATOS EN C

Las características del lenguaje C que va a permitir implementar un **TAD** son, principalmente, las estructuras o registros para representación de los datos, y las funciones para representar las operaciones especificadas. Además, en un archivo de inclusión o cabecera se guarda la representación de los datos y el interfaz, prototipos de las funciones, del tipo abstracto.

Una estructura es una agrupación de campos o variables de cualquier tipo predefinido. Permite que un grupo de variables que tienen una cierta relación lógica sean tratadas como un *todo*. Por ejemplo, un punto en el espacio tridimensional se caracteriza por tres variables que representan a las respectivas coordenadas.

```
struct punto3D
{
   float x;
   float y;
   float z;
};
```

La sentencia `typedef` permite dar un nombre a un tipo de dato ya definido, de tal forma que a partir de ella el programador se puede referir al tipo de dato con el nuevo nombre.

```
typedef tipo_predefinido nuevo_nombre;
```

En los programas C, y en particular al declarar estructuras, la sentencia `typedef` se utiliza conjuntamente con la declaración del tipo estructura. Así, para la estructura que representa coordenadas tridimensionales.

```
typedef struct
{
   float x;
   float y;
   float z;
} punto3D;
```

Se ha declarado el nuevo tipo `punto3D`, que permitirá definir variables o declarar el tipo de una función.

```
punto3D p1, p2;
punto3D desplazamiento(punto3D, float x);
```

En programación

La sentencia `typedef` identifica a un tipo de dato con un nuevo nombre con el fin de que sea más legible el código fuente.

Los archivos de *inclusión* o de cabecera se utilizan para agrupar en ellos variables externas, declaraciones de datos comunes y prototipos de funciones. Estos archivos de cabecera se incluyen en los archivos que contienen la codificación de las funciones, archivos fuente, con la directiva del preprocesador `#include`. Al implementar un TAD en C, se agrupa, en cierto modo se *encierra*, en estos archivos la representación de los datos y el interfaz del TAD, a su vez, representado por los prototipos de las funciones. De esta forma en los archivos de código fuente que utilicen el TAD hay que escribir la directiva.

```
#include "tipodedato.h"
```

8.8.1. Implementación del TAD `Conjunto` en C

La implementación del tipo `Conjunto` se tiene que hacer en función de la especificación realizada en el Apartado 8.7. Para hacer la implementación lo más flexible posible, no se establece que el conjunto pueda tener un máximo de elementos. Esto exige el uso de asignación dinámica de memoria.

En el archivo de cabecera `conjunto.h` se realiza la declaración de la estructura que va a representar a los datos. Estos datos pueden ser de cualquier tipo, por lo que se hace referencia a `TipoDato` que es necesario especificar antes de incluir `conjunto.h`. La constante `M`, que arbitrariamente toma el valor de `10`, es el número de «huecos», posiciones de memoria, que se reservan cada vez que hay que ampliar el tamaño de la estructura.

***Archivo* `conjunto.h`**

```
#define M 10
typedef struct
{
   TipoDato* cto;
   int cardinal;
   int capacidad;
} Conjunto;

void conjuntoVacio(Conjunto* c);
int esVacio(Conjunto c);
void annadir(Conjunto* c, TipoDato elemento);
void retirar(Conjunto* c, TipoDato elemento);
int pertenece(Conjunto c, TipoDato elemento);
int cardinal(Conjunto c);
Conjunto unionC(Conjunto c1, Conjunto c2);
```

Este archivo de cabecera, `conjunto.h`, hay que incluirlo en todos los archivos con código C que vaya a utilizar el tipo conjunto. Es importante recordar que antes de escribir la sentencia `include` hay que asociar un tipo predefinido a `TipoDato`. Si, por ejemplo, los elementos del conjunto son las coordenadas de un punto en el plano.

```
typedef struct
{
  float x;
  float y;
} Punto;

typedef Punto TipoDato;
#include "conjunto.h"
```

Las funciones cuyos prototipos han sido ya escritos, se codifican y se guardan en el archivo `conjunto.c`. La compilación de `conjunto.c` da lugar al archivo con el código objeto que se ensamblará con el código objeto de otros archivos fuente que hacen uso del TAD `Conjunto`.

***Archivo* `conjunto.c`**

```
typedef struct {var(s)} Tipo;

typedef Tipo TipoDato;
#include "conjunto.h"
```

```
  /* iguales() devuelve 1(cierto) si todos los campos lo son.
     La implementación depende del tipo concreto de los datos
     del conjunto.
  */
int iguales(TipoDato e1, TipoDato e2)
{
  return (e1.v1 == e2.v1) && (e1.v2 == e2.v2) ... ;
}
void conjuntoVacio(Conjunto* c)
{
  c -> cardinal = 0;
  c -> capacidad = M;
  c -> cto = (TipoDato*)malloc(M*sizeof(TipoDato));
}

int esVacio(Conjunto c)
{
  return (c.cardinal == 0);
}

void annadir(Conjunto* c, TipoDato elemento)
{

  if (!pertenece(*c, elemento))
  {
    /* verifica si hay posiciones libres,
       en caso contrario amplia el conjunto */
    if (c -> cardinal == c -> capacidad )
    {
      Conjunto nuevo;
      int k;
      nuevo.cto = (TipoDato*)
                  malloc((c->capacidad + M)*sizeof(TipoDato));
      for (k = 0; k < c->capacidad; k++)
        nuevo.cto[k] = c -> cto[k];

      free(c -> cto);
      c -> cto = nuevo.cto;
    }

    c -> cto[c->cardinal++] = elemento;
  }
}

void retirar(Conjunto* c, TipoDato elemento)
{
  if (pertenece(*c, elemento))
  {
    int k;
    k = 0;
    while (!iguales(c->cto[k], elemento)) k++;

      /* desde el elemento k hasta la última posición
         mueve los elementos una posición a la izquierda */
```

```
    for (; k < c -> cardinal ; k++)
      c -> cto[k] = c -> cto[k+1];
    c->cardinal--;
  }
}
int pertenece(Conjunto c, TipoDato elemento)
{
  int k, encontrado;
  k = encontrado = 0;

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

int cardinal(Conjunto c)
{
  return c.cardinal;
}

Conjunto unionC(Conjunto c1, Conjunto c2)
{
  Conjunto u;
  int k;

  u.cardinal = 0;
  u.capacidad = c1.capacidad;
  u.cto = (TipoDato*)malloc(u.capacidad*sizeof(TipoDato));

  for (k = 0; k < c1.capacidad; k++)
    u.cto[k] = c1.cto[k];
  u.cardinal = c1.cardinal;

  for (k = 0; k < c2.capacidad; k++)
    annadir(&u, c2.cto[k]);

  return u;

}
```

8.8.2. Aplicación del tipo abstracto de dato `Conjunto`

Los conjuntos son utilizados con frecuencia en algoritmos que resuelven problemas diversos. Ahora se escribe un sencillo programa para demostrar el uso del tipo conjunto. El programa crea un conjunto de números racionales y realiza las operaciones del tipo abstracto. Se añade una operación complementaria, que visualiza los elementos del conjunto.

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

typedef struct
{
```

```
  int n,d;
} Racional;
typedef Racional TipoDato;

#include "conjunto.h"
void mostrar(Conjunto c);

void main()
{
  int j;
  Conjunto c,d,t;
  TipoDato r;

  conjuntoVacio(&c);
  randomize();

  for (j = 1; j <= 10; j++)
  {
    r.n = random(11)+1;
    r.d = random(11)+1;
    annadir(&c,r);
  }
  mostrar(c);
  printf("Elemento a eliminar: ");
  scanf("%d %d",&r.n,&r.d);
  retirar(&c,r);
  mostrar(c);

  conjuntoVacio(&d);
  for (j = 1; j <= 10; j++)
  {
    r.n = random(11)+1;
    r.d = random(11)+1;
    annadir(&d,r);
  }
  mostrar(d);

  t = unionC(c,d);
  mostrar(t);
}

void mostrar(Conjunto c)
{
  int k;
  puts("\n Mostrar conjunto ");
  for (k=0; k< c.cardinal;)
  {
    printf("%d/%d ",c.cto[k].n,c.cto[k].d);
    k++;
    if (k%10 == 0) puts(" ");
  }
}
```

La función `iguales()` adaptada para verificar si dos números racionales son iguales:

```
int iguales(TipoDato e1, TipoDato e2)
{
    return (e1.n == e2.n) && (e1.d == e2.d) ;
}
```

8.9. 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.*

El atractivo intuitivo de la orientación a objetos es que proporciona conceptos y herramientas con las cuales se modela y representa el mundo real tan fielmente como sea posible. Las ventajas de la orientación a objetos son muchas en programación y modelado de datos. Como apuntaban Ledbetter y Cox (1985):

> La programación orientada a objetos permite una representación más directa del modelo de mundo real en el código. El resultado es que la transformación radical normal de los requisitos del sistema (definido en términos de usuario) a la especificación del sistema (definido en términos de computador) se reduce considerablemente.

La Figura 8.5 ilustra el problema. Utilizando técnicas convencionales, el código generado para un problema de mundo real consta de una primera codificación del problema y a continuación la transformación del problema en términos de un lenguaje de computador Von Newmann. Las disciplinas y técnicas orientadas a objetos manipulan la transformación automáticamente, de modo que el volumen de código maximiza el problema y la transformación lo minimiza. De hecho, cuando se compara con estilos de programación convencionales *(procedimentales, por procedimientos),* las reducciones de código van desde un 40 por 100 hasta un orden de magnitud elevado cuando se adopta un estilo de programación orientado a objetos.

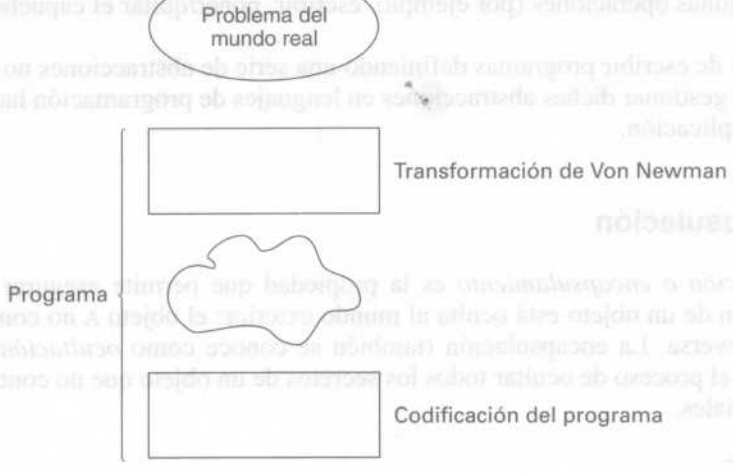


Figura 8.5. Construcción de software.

Los conceptos y herramientas orientadas a objetos son tecnologías que permiten que los problemas del mundo real sean expresados de modo fácil y natural. Las técnicas orientadas a objetos proporcionan mejoras metodológicas para construir sistemas de software complejos a partir de unidades de software modularizado y reutilizable.

Se necesita un nuevo enfoque para construir software en la actualidad. Este nuevo enfoque debe ser capaz de manipular, tanto sistemas grandes como pequeños, y debe crear sistemas fiables que sean flexibles, mantenibles y capaces de evolucionar, para cumplir las necesidades de cambio.

La tecnología orientada a objetos puede cubrir estos cambios y algunos otros más en el futuro. La orientación a objetos trata de cumplir las necesidades de los usuarios finales, así como las propias de los desarrolladores de productos software.

Estas tareas se realizan mediante la modelización del mundo real. El soporte fundamental es el *modelo objeto*. Los cuatro elementos (propiedades) más importantes de este modelo[3] son:

- Abstracción.
- Encapsulación.
- Modularidad.
- Jerarquía.

Como sugiere Booch, si alguno de estos elementos no existe, se dice que el modelo no es orientado a objetos.

8.9.1. Abstracción

La *abstracción* es uno de los medios más importantes mediante el cual nos enfrentamos con la complejidad inherente al software. La **abstracción** es la propiedad que permite representar las características esenciales de un objeto, sin preocuparse de las restantes características (no esenciales).

Una abstracción se centra en la vista externa de un objeto, de modo que sirva para separar el comportamiento esencial de un objeto, de su implementación. Definir una abstracción significa describir una entidad del mundo real, no importa lo compleja que pueda ser, y a continuación utilizar esta descripción en un programa.

El elemento clave de la programación orientada a objetos es la clase. Una **clase** se puede definir como una descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por su *estado* específico y por la posibilidad de realizar una serie de *operaciones*. Por ejemplo, una pluma estilográfica es un objeto que tiene un estado (llena de tinta o vacía) y sobre la cual se pueden realizar algunas operaciones (por ejemplo, escribir, poner/quitar el capuchón, llenar de tinta si está vacía).

La idea de escribir programas definiendo una serie de abstracciones no es nueva, pero el uso de clases para gestionar dichas abstracciones en lenguajes de programación ha facilitado considerablemente su aplicación.

8.9.2. Encapsulación

Encapsulación o encapsulamiento es la propiedad que permite asegurar que el contenido de la información de un objeto está oculta al mundo exterior: el objeto `A` no conoce lo que hace el objeto `B` y viceversa. La encapsulación (también se conoce como *ocultación de la información*), en esencia, es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales.

---

[3] Booch, Grady. *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, 1994.

La encapsulación permite la división de un programa en módulos. Estos módulos se implementan mediante clases, de forma que una clase representa la encapsulación de una abstracción. En la práctica, esto significa que cada clase debe tener dos partes: un interfaz y una implementación. El *interfaz* de una clase captura sólo su vista externa y la *implementación* contiene la representación de la abstracción, así como los mecanismos que realizan el comportamiento deseado.

8.9.3. Modularidad

La *modularidad* es la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas *módulos*), cada una las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.

La modularización, como indica Liskov, consiste en dividir un programa en módulos que se pueden compilar por separado, pero que tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la modularidad de diversas formas. Por ejemplo, en C++ los módulos son archivos compilados por separado. La práctica usual es situar los interfaces de los módulos en archivos con nombres con extensión *.h* (*archivos de cabecera*) y las implementaciones de los módulos se sitúan en archivos con nombres con extensión *.cpp*.

En Ada, el módulo se define como *paquete* (`package`). Un paquete tiene dos partes: la *especificación* del paquete y el *cuerpo* del paquete, también se pueden compilar por separado.

La *modularidad* es la propiedad de un sistema que permite su descomposición en un conjunto de módulos cohesivos y débilmente acoplados.

8.9.4. Jerarquía

La jerarquía es una propiedad que permite una ordenación de las abstracciones. Las dos jerarquías más importantes de un sistema complejo son: estructura de clases (jerarquía «**es-un**» [*is-a*]: generalización/especialización) y estructura de objetos (jerarquía «**parte-de**» [*part-of*]: agregación).

A tener en cuenta

No se deben confundir clases y objetos de la misma clase: un coche rojo y un coche azul no son objetos de clases diferentes, sino objetos de la misma clase con un atributo diferente.

Las jerarquías de *generalización/especialización* se conocen como *herencia*. Básicamente, la herencia define una relación entre clases, en donde una clase comparte la estructura o comportamiento definido en una o más clases (*herencia simple* y *herencia múltiple*, respectivamente).

La agregación es el concepto que permite el agrupamiento físico de estructuras relacionadas lógicamente. Así, un camión se compone de ruedas, motor, sistema de transmisión y chasis; en consecuencia camión es una *agregación* y ruedas, motor, transmisión y chasis son agregados de camión.

8.9.5. Polimorfismo

La quinta propiedad significativa de los lenguajes de programación orientados a objetos es el *polimorfismo*. Esta propiedad no suele ser considerada como fundamental en los diferentes modelos de objetos propuestos, pero dada su importancia, no tiene sentido considerar un *objeto modelo* que no soporte esta propiedad.

Polimorfismo es la propiedad, que indica, literalmente, la posibilidad de que una entidad tome *muchas formas*. En términos prácticos, el polimorfismo permite referirse a objetos de clases diferentes mediante el mismo elemento de programa y realizar la misma operación de diferentes formas, según sea el objeto que se referencia en ese momento.

Por ejemplo, cuando se describe la clase *mamíferos*, se puede observar que la operación *comer* es una operación fundamental en la vida de los mamíferos, de modo que cada tipo de mamífero debe poder realizar la operación o función *comer*. Por otra parte, una vaca o una cabra que pastan en un campo, un niño que se come un caramelo y un león que devora a otro animal, son diferentes formas que utilizan los distintos mamíferos, para realizar la misma función (*comer*).

El polimorfismo implica la posibilidad de tomar un objeto de un tipo (mamífero, por ejemplo) e indicarle que ejecute *comer*; esta acción se ejecutará de diferente forma según sea el objeto mamífero sobre el que se aplica.

Clases, herencia y polimorfismo son aspectos claves en la programación orientada a objetos y se reconocen a estos elementos como esenciales en la *orientación a objetos*. El polimorfismo adquiere su máxima expresión en la *derivación* o *extensión* de clases; es decir, cuando se obtiene una clase a partir de una clase ya existente, mediante la propiedad de derivación de clases o herencia. Así, por ejemplo, si se dispone de una figura que represente figuras genéricas, se puede enviar cualquier mensaje tanto a un tipo derivado (elipse, círculo, cuadrado, etc.) como al tipo base. Por ejemplo, una clase *figura*, pueden aceptar los mensajes *dibujar*, *borrar* y *mover*. Cualquier tipo derivado de una *figura* es un tipo de figura y puede recibir el *mismo* mensaje. Cuando se envía un mensaje, por ejemplo dibujar, esta tarea será distinta según que la clase sea un triángulo, un cuadrado o una elipse. Esta propiedad es el polimorfismo, que permite que una misma función se comporte de diferente forma según sea la clase sobre la que se aplica. La función *dibujar* se aplica igualmente a un círculo, un cuadrado o un triángulo y el objeto ejecutará el código apropiado dependiendo del tipo específico.

El polimorfismo requiere *ligadura tardía* o *postergada* (también llamada *dinámica*), y esto sólo se puede producir en lenguajes de programación orientados a objetos. Los lenguajes no orientados a objetos, soportan *ligadura temprana* o *anterior*; esto significa que el compilador genera una llamada a un nombre específico de función y el enlazador (*linker*) resuelve la llamada a la dirección absoluta del código que se ha de ejecutar. En POO, el programa no puede determinar la dirección del código hasta el momento de la ejecución; para resolver este concepto, los lenguajes orientados a objetos utilizan el concepto de *ligadura tardía*. Cuando se envía un mensaje a un objeto, el código que se llama no se determina hasta el momento de la ejecución. El compilador asegura que la función existe y realiza verificación de tipos de los argumentos y del valor de retorno, pero no conoce el código exacto que ejecutar.

Para realizar la ligadura tardía, el compilador inserta un segmento especial de código en lugar de la llamada absoluta. Este código calcula la dirección del cuerpo de la función para ejecutar en tiempo de ejecución utilizando información almacenada en el propio objeto. Por consiguiente, cada objeto se puede comportar de modo diferente de acuerdo al contenido de ese puntero. Cuando se envía un mensaje a un objeto, éste realmente sabe qué ha de hacer con ese mensaje.

8.9.6. Otras propiedades

El modelo objeto ideal no sólo tiene las propiedades anteriormente citadas al principio del apartado, sino que es conveniente que soporte, además, estas otras propiedades:

- Concurrencia (multitarea).
- Persistencia.
- Genericidad.
- Manejo de excepciones.

Muchos lenguajes soportan todas estas propiedades y otros sólo algunas de ellas. Así, por ejemplo, Ada soporta concurrencia y Ada y C++ soportan genericidad y manejo de excepciones. La

persistencia o propiedad de que las variables —y por extensión, a los objetos— existan entre las invocaciones de un programa, es posiblemente la propiedad menos implantada en los lenguajes orientados a objetos, aunque ya es posible considerar la persistencia en lenguaje tales como Smalltalk y C#, lo que facilitará el advenimiento de las bases de datos orientadas a objetos como así está sucediendo en la actualidad.

8.9.7. Lenguajes de programación orientados a objetos

El primer lenguaje de programación que introdujo el concepto de clase fue Simula-67, como entidad que contenía datos y las operaciones que manipulaban los datos. Así mismo, introdujo también el concepto de herencia.

El siguiente lenguaje orientado a objetos, y seguramente el más popular desde un enfoque conceptual exclusivamente de objetos, es Smalltalk, cuya primera versión comercial se desarrolló en 1976, y en 1980 se popularizó con la aparición de Smalltalk-80. Posteriormente, se ha popularizado gracias al desarrollo de Smalltalk V de la casa Digital y, que recientemente se ha implementado bajo entorno Windows. El lenguaje se caracteriza por soportar todas las propiedades fundamentales de la orientación a objetos, dentro de un entorno integrado de desarrollo, con interfaz interactiva de usuario basado en menús.

Entre los lenguajes orientados a objetos que se han desarrollado a partir de los años ochenta, destacan extensiones de lenguajes tradicionales tales como C++ y Objective-C (extensiones de C), Modula-2 y Object Pascal (extensión de Pascal) y recientemente Object Cobol, que a lo largo de 1994 aparecieron sus primeras versiones comerciales.

Otro lenguaje orientado a objetos puros es Eiffel, creado por Bertrand Meyer y que soporta todas las propiedades fundamentales de objetos. Hasta ahora no ha adquirido popularidad más que en ambientes universitarios y de investigación. Sin embargo, la aparición en el año 1995 de la versión 3, que corre bajo Windows ha hecho aumentar su difusión.

Ada ha sido, también, un lenguaje —en este caso *basado en objetos*— que soporta la mayoría de las propiedades orientadas a objetos. Sin embargo, la versión Ada-95 ya soporta herencia y polimorfismo.

De cualquier forma, existe un *rey* actual en los lenguajes orientados a objetos, **C++**. La normalización por ANSI y AT&T de la versión 3.0 y las numerosas versiones de diferentes fabricantes, tales como Borland, Microsoft, Symantec, etc., hacen que en la segunda mitad de los años noventa haya sido el lenguaje orientado a objetos más popular y utilizado en el mundo de la programación. A C++ se le ha unido en la segunda mitad de la década de los años noventa, **Java**, un lenguaje evolución de C++, creado específicamente para programar en Internet pero que puede ser utilizado también en aplicaciones de propósito general. **Java** se está convirtiendo en el lenguaje más utilizado en programación y en particular en orientación a objetos.

La evolución de los lenguajes orientados a objetos se muestra en la Figura 8.6, en la que se aprecia el tronco común a todos los lenguajes modernos Algol y las tres líneas fundamentales: *enfoque en Pascal* (Ada, Object Pascal), *enfoque puro* de orientación a objetos (Simula/Smalltalk) y *enfoque en C* (Objective-C, C++, Java y Ada-95).

Hace un par de años y con el objetivo de competir con Java, Microsoft construyó C#, un lenguaje orientado a objetos y para programación de Internet.

8.10. REUTILIZACIÓN DE SOFTWARE

Cuando se construye un automóvil, un edificio o un dispositivo electrónico, se ensamblan una serie de piezas independientes de modo que estos componentes se reutilizan en vez de fabricarlos cada vez que se necesita construir un automóvil o un edificio. En la construcción de software, esta pregunta es continua, ¿por qué no se utilizan programas ya construidos para formar programas más grandes? Es decir, si en electrónica, los computadores y sus periféricos se forman esencialmente

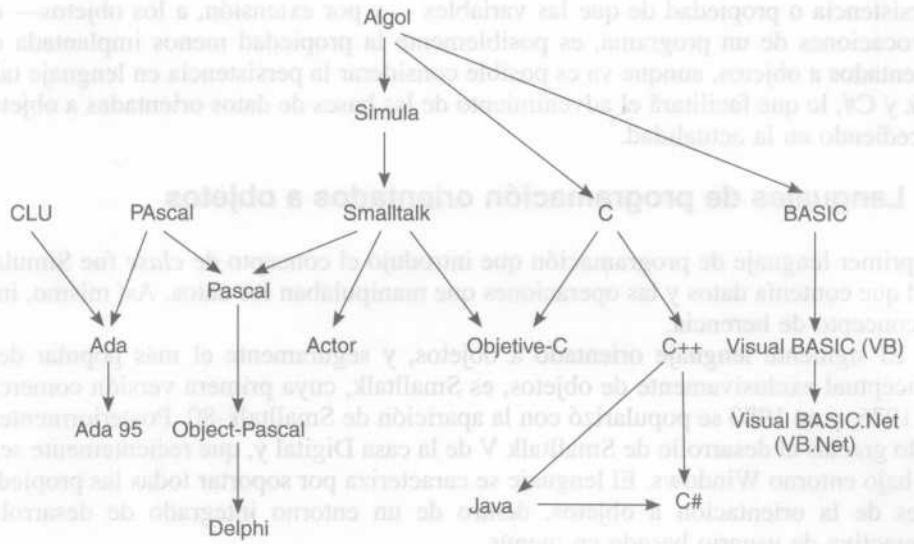


Figura 8.6. Evolución de los lenguajes orientados a objetos.

con el ensamblado de circuitos integrados, ¿existe algún método que permite realizar grandes programas a partir de la utilización de otros programas ya realizados? ¿Es posible reutilizar estos componentes de software?

Las técnicas orientadas o objetos proporcionan un mecanismo para construir *componentes de software* reutilizables que posteriormente puedan ser interconectados entre sí y formar grandes proyectos de software[4].

En los sistemas de programación tradicionales y en particular en los basados en lenguajes de programación estructurados (tales como FORTRAN, C, etc.) existen las bibliotecas de funciones, que contienen funciones (o procedimientos, según el lenguaje) que pueden ser incorporados en diferentes programas. En sistemas orientados a objetos se pueden construir componentes de software reutilizables, al estilo de las bibliotecas de funciones, normalmente denominados *bibliotecas de software o paquetes de software reutilizables*. Ejemplos de componentes reutilizables comercialmente disponibles son herramientas de C++, Java, C#, etc.

En el futuro inmediato, los ingenieros de software dispondrán de catálogos de paquetes de software reutilizable, al igual que sucede con los catálogos de circuitos integrados electrónicos como les ocurre a los ingenieros de hardware.

Las técnicas orientadas a objetos ofrecen una alternativa de escribir el mismo programa una y otra vez. El programador orientado a objetos modifica una funcionalidad del programa sustituyendo elementos antiguos u objetos por nuevos objetos, o bien conectando simplemente nuevos objetos en la aplicación.

La reutilización de código en programación tradicional se puede realizar copiando y editando mientras que en programación orientada a objetos se puede reutilizar el código, creando automáticamente una subclase y anulando alguno de sus métodos.

Muchos lenguajes orientados a objetos fomentan la reutilización mediante el uso de bibliotecas robustas de clases preconstruidas, así como otras herramientas como *hojeadores (browser)*, para localizar clases de interés y depuradores interactivos para ayudar al programador.

---

[4] Brad Cox, en su ya clásico libro *Object-Oriented Programming An Evolutionary Approach* (Cox, Novobilski, 91), acuñó el término «chip de software» (*Software-IC*), o componentes de software, para definir las clases de objetos como componentes de software reutilizables. Existe versión en español de Addison-Wesley/Díaz de Santos, 1993, con el título *Programación orientada a objetos. Un enfoque evolutivo.*

8.11. DESARROLLO TRADICIONAL FRENTE AL ORIENTADO A OBJETOS

El sistema tradicional del desarrollo del software para un determinado sistema, es la subdivisión del mismo en módulos, a la cual deben aplicarse criterios específicos de descomposición, los cuales se incluyen en metodologías de diseño. Estos módulos se refieren a la fase de construcción de un programa, que en el modelo clásico, sigue a la definición de los requisitos (fase de análisis).

El modelo clásico del ciclo de vida del software no es el único modelo posible, dado que es posible desarrollar código de un modo evolutivo, por refinamiento y prototipos sucesivos. Existen numerosos lenguajes de programación y metodologías que se han desarrollado en paralelo a los mismos, aunque, normalmente, con independencia de ellos.

En este apartado nos centraremos en la metodología más utilizada denominada *desarrollo estructurado*, que se apoya esencialmente en el *diseño descendente* y en la *programación estructurada*.

La *programación estructurada* es un estilo disciplinado de programación que se sigue en los lenguajes procedimentales (por procedimientos) tales como FORTRAN, BASIC, COBOL, C y C++.

Las metodologías *diseño descendente* (o descomposición funcional) se centran en operaciones y tienden a descuidar la importancia de las estructuras de datos. Se basan en la célebre ecuación de Wirth:

Datos + Algoritmos = Programas

La idea clave del diseño descendente es romper un programa grande en tareas más pequeñas, más manejables. Si una de estas tareas es demasiado grande, se divide en tareas más pequeñas. Se continúa con este proceso hasta que el programa se compartimentaliza en módulos más pequeños y que se programan fácilmente. Los subprogramas facilitan el enfoque estructurado, y en el caso de lenguajes como C, estas unidades de programas, llamadas *funciones*, representan las citadas tareas o módulos individuales. Las técnicas de programación estructuradas reflejan, en esencia, un modo de resolver un programa en términos de las acciones que realiza.

Para comprender mejor las relaciones entre los algoritmos (*funciones*) y los datos, consideremos una comparación con el lenguaje natural (por ejemplo, español o inglés), que se compone de muchos elementos, pero que reflejará poca expresividad si sólo se utilizan nombres y verbos. Una metodología que se basa *solo en datos* o *sólo en procedimientos* es similar a un lenguaje en el que sólo se utilizan nombres o verbos. Sólo enlazando nombres o verbos correctos (siguiendo las reglas semánticas), las expresiones tomarán formas inteligibles y su proceso será más fácil.

Las metodologías tradicionales se vuelven poco prácticas cuando han de aplicarse a proyectos de gran tamaño. El diseño orientado a objetos se apoya en lenguajes orientados a objetos, que se sustentan fundamentalmente en los tipos de datos y operaciones que se pueden realizar sobre los tipos de datos. Los datos no fluyen abiertamente en un sistema, como ocurre en las técnicas estructuradas, sino que están protegidos de modificaciones accidentales. En programación orientada a objetos, los *mensajes* (en vez de los datos) se mueven por el sistema. En lugar del enfoque funcional (invocar una función con unos datos), en un lenguaje orientado a objetos, «se envía un mensaje a un objeto».

De acuerdo con Meyer, el diseño orientado a objetos es el método que conduce a arquitecturas de software basadas en objetos que cada sistema o subsistema evalúa.

Recordemos *qué son los objetos*. Un **objeto** es una entidad cuyo comportamiento se caracteriza por las acciones que realiza. Con más precisión, un objeto se define como una entidad caracterizada por un estado; su comportamiento se define por las operaciones que puede realizar; es una *instancia* de una *clase*; se identifica por un nombre; tiene una visibilidad limitada para otros objetos; se define el objeto mediante su especificación y su implementación.

Una definición muy elaborada se debe a Meyer: «Diseño orientado a objetos es la construcción de sistemas de software como colecciones estructuradas de implementaciones de tipos de datos abstractos».

La construcción de un sistema se suele realizar mediante el ensamblado ascendente (abajo-arriba) de clases preexistentes. Las clases de un sistema pueden tener entre sí relaciones de uso (*cliente*), relaciones de derivación (*herencia*) o relaciones de agregación (*composición*) o, incluso sólo relaciones de asociación. Así, por ejemplo, con una relación de cliente, una clase puede utilizar los objetos de otra clase; con una relación de herencia, una *clase puede heredar o derivar* sus propiedades definidas en otra clase.

El *hardware* se ensambla a partir de componentes electrónicos, tales como circuitos integrados (chips), que se pueden utilizar repetidamente para diseñar y construir conjuntos mucho más grandes, que son totalmente reutilizables. La calidad de cada nivel de diseño se asegura mediante componentes del sistema que han sido probados previamente a su utilización. El ensamblado de componentes electrónicos se garantiza mediante interfaces adecuados.

Estos conceptos se aplican también con tecnologías de objetos. Las clases (tipos de objetos) son como los *chips de hardware*, Cox los llamó *chips de software*, que no sólo se pueden enlazar (ensamblar) entre sí, sino que también se pueden reutilizar (volver a utilizar). Las clases se agruparán, normalmente, en bibliotecas de clases, que son componentes reutilizables, fácilmente legibles.

En la actualidad, existen gran cantidad de software convencional, en su mayoría escrito normalmente para resolver problemas específicos; por esta razón, a veces es más fácil escribir nuevos sistemas que convertir los existentes.

Los objetos, al reflejar entidades del mundo real, permiten desarrollar aplicaciones creando nuevas clases y ensamblándolas con otras ya existentes. Normalmente, los desarrolladores experimentados gastan un porcentaje alto de su tiempo (20 al 40 por 100) en crear nuevas clases y el tiempo restante en ensamblar componentes probados de sistemas, construyendo sistemas potentes y fiables.

RESUMEN

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

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

En el lenguaje C un TAD se puede implementar utilizando las estructuras (`struct`) y los archivos de cabecera para agrupar el interfaz del tipo abstracto.

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

En este capítulo se considera también una introducción a los métodos de desarrollo orientados a objetos. El nuevo modelo de programación se apoya esencialmente en el concepto de objetos. Se comienza con una breve revisión de las propiedades de la orientación a objetos: abstracción, encapsulación, modularidad, jerarquía, polimorfismo.

La orientación a objetos modela el mundo real de un modo más fácil a la perspectiva del usuario que el modelo tradicional. La orientación a objetos proporciona mejores técnicas y paradigmas para construir componentes de software reutilizables y bibliotecas ampliables de módulos de software. Esta característica mejora la extensibilidad de los programas desarrollados a través de metodologías de orientación a objetos. Los usuarios finales, programadores de sistemas y desarrolladores de aplicaciones, se benefician de las tecnologías de modelado y programación orientadas a objetos.

Los conceptos fundamentales de orientación a objetos son: tipos abstractos de datos, herencia e identidad de los objetos. Un tipo abstracto de datos describe una colección con la misma estructura y comportamiento. Los tipos abstractos de datos extienden la noción de tipos de datos ocultando la implementación de operaciones definidas por el usuario (mensajes) asociados con los tipos de datos. Los tipos abstractos de datos se implementan a través de clases. Las clases pueden heredar unas de otras. Mediante la herencia, se pueden construir nuevos módulos de software (tales como clases) en la parte superior de una jerarquía existente de módulos. La herencia permite la compartición de código (y por consiguiente, reutilización) entre módulos de software. La identidad es la propiedad de un objeto que diferencia cada objeto de los restantes. Con la identidad de un objeto, los objetos pueden contener o referirse a otros objetos. La identidad del objeto organiza los objetos del espacio del objeto manipulado por un programa orientado a objetos.

Este capítulo examina el impacto de las tecnologías orientadas a objetos en lenguajes de programación, así como los beneficios que producen en el desarrollo de software.

EJERCICIOS

8.1. Realizar una especificación informal del TAD *Conjunto* con las operaciones: *ConjuntoVacio*, *Esvacio*, *Añadir* un elemento al conjunto, *Pertenece* un elemento al conjunto, *Retirar* un elemento del conjunto, *Union* de dos conjuntos, *Intersección* de dos conjuntos e *Inclusión* de conjuntos.

8.2. Realizar la especificación formal del TAD *Conjunto* con las operaciones indicadas en el Ejercicio 8.1. Considerar a las operaciones *ConjuntoVacio* y *Añadir* como constructores.

8.3. Especificar formalmente un tipo abstracto de datos pila de números enteros y que al menos soporte las siguientes operaciones:

CrearPila: Inicializa la pila con el estatus de vacía.
Borrar: Eliminar todos los números de la pila.
Copiar: Hace una copia de la pila actual.
Meter: Añadir un nuevo elemento en la cima de la pila.
Sacar: Quitar un elemento de la pila.
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.

Considerar a las operaciones *CrearPila* y *Meter* como constructores.

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

8.5. Crear un TAD que represente un dato tipo cadena (*string*) y sus diversas operaciones: *CadenaVacia*, *Asignar*, *Longitud*, *Buscar* posición de un carácter dado, *Concatenar* cadenas, *Extraer* una subcadena. Realizar la especificación informal y formal considerando como constructores las operaciones *CadenaVacia* y *Asignar*.

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

8.7. 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.

8.8. Diseñar el tipo abstracto de datos Vector con la finalidad de representar una secuencia de *n* elementos del mismo tipo. Las operaciones a definir: *CrearVector* (crea un vector *n* posiciones vacías), *Asignar* (asigna un elemento en la posición *j*), *ObtenerElemento* (devuelve el elemento que se encuentra en la posición *j*), *SubVector* (devuelve el vector comprendido entre las posiciones *i*, *j*). Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

8.9. 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.

PROBLEMAS

8.1. Ampliar la codificación del TAD Conjunto, realizada en el Apartado 8.8, con las nuevas operaciones especificadas en los Ejercicios 8.1 y 8.2.

8.2. Implementar el TAD Bolsa descrito en el Ejercicio 8.6. Probar la implementación con un programa que invoque a las operaciones del tipo abstracto Bolsa.

8.3. Implementar el TAD Cadena descrito en el Ejercicio 8.5. Probar la implementación con un programa que realice diversas operaciones con cadenas.

8.4. Implementar el TAD Complejo especificado en el Ejercicio 8.7. Escribir un programa en el que se realicen diversas operaciones con números complejos.

8.5. Implementar el TAD Vector especificado en el Ejercicio 8.8 con una estructura dinámica.

8.6. Implementar el TAD Matriz especificado en el Ejercicio 8.9 con una estructura dinámica. Escribir un programa que haciendo uso del TAD Matriz se realicen operaciones diversas y escriba las matrices generadas.

9

LISTAS, LISTAS ENLAZADAS

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Distinguir entre una estructura secuencial y una estructura enlazada.
- Definir el tipo abstracto de datos `ListaEnlazada`.
- Aplicar la asignación dinámica de memoria para crear estructuras de información.
- Conocer las operaciones básicas de la listas enlazadas.
- Implementar una lista enlazada para un tipo de elemento.
- Realizar aplicaciones donde los datos se representan con una estructura lista.
- Definir una lista doblemente enlazada.
- Definir una lista circular.
- Implementar listas enlazadas ordenadas.

CONTENIDO

9.1. Fundamentos teóricos de las listas enlazadas.
9.2. Clasificación de las listas enlazadas.
9.3. Tipo abstracto de datos (TAD) lista.
9.4. Operaciones en listas enlazadas.
9.5. Inserción de un elemento en una lista.
9.6. Búsqueda en listas enlazadas.
9.7. Eliminación de un nodo de una lista.
9.8. Lista doblemente enlazada.
9.9. Insertar un elemento en una lista doblemente enlazada.
9.10. Eliminación de un elemento en una lista doblemente enlazada.
9.11. Listas circulares.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Variables referencia y variables apuntadas.
- Punteros (apuntadores) y puntero nulo.
- Asignación dinámica de memoria.
- Pila de memoria libre.
- Estructura de una lista enlazada.
- Operaciones en listas enlazadas.
- Lista doblemente enlazada.
- Lista circular.
- Recorrido de una lista.
- Eliminación de nodos en una lista enlazada.
- Búsqueda de un nodo en una lista enlazada.

INTRODUCCIÓN

En este capítulo se comienza el estudio de las estructuras de datos dinámicas. Al contrario que las estructuras de datos estáticas (*arrays* —listas, vectores y tablas— y *estructuras*) en las que su tamaño en memoria se establece durante la compilación y permanece inalterable durante la ejecución del programa, las estructuras de datos dinámicas crecen y se contraen a medida que se ejecuta el programa.

La estructura de datos que se estudiará en este capítulo es la lista enlazada (ligada o encadenada, *linked list*) que es una colección de elementos (denominados nodos) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un «enlace» o «referencia». En el capítulo se desarrollan algoritmos para insertar, buscar y eliminar elementos en las listas enlazadas. De igual modo se muestra el tipo abstracto de datos (TAD) que representa a las listas enlazadas.

Las listas enlazadas son estructuras muy flexibles y con numerosas aplicaciones en el mundo de la programación.

9.1. FUNDAMENTOS TEÓRICOS DE LAS LISTAS ENLAZADAS

Los *arrays* han sido utilizados para implementar estructuras lineales de elementos homogéneos (listas, tablas, vectores). Esta técnica obliga a fijar por adelantado el espacio a ocupar en memoria, de modo que cuando se desea añadir un nuevo elemento que rebase el tamaño prefijado del array, no es posible realizar la operación sin que se produzca un error en tiempo de ejecución. Ello se debe a que los arrays hacen un uso ineficiente de la memoria. Gracias a la asignación dinámica de variables, se pueden implementar listas de modo que la memoria física utilizada se corresponda con el número de elementos de la tabla. Para ello se recurre a los **punteros** (*apuntadores*) que hacen un uso más eficiente de la memoria como ya se ha visto con anterioridad.

Una **lista enlazada** es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un «enlace» o «puntero». La idea básica consiste en construir una lista cuyos elementos llamados **nodos** se componen de dos partes o *campos*: la primera parte o campo contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado *Dato, TipoElemento, Info*, etc.) y la segunda parte o *campo* es un puntero (denominado *enlace* o *sgte*) que apunta al siguiente elemento de la lista.

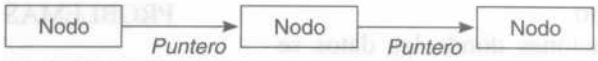
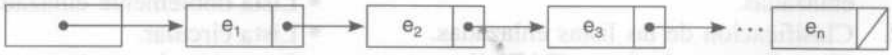


Figura 9.1. Lista enlazada (representación simple).

La representación gráfica más extendida es aquella que utiliza una caja (un rectángulo) con dos secciones en su interior. En la primera sección se escribe el elemento o valor del dato, y en la segunda sección, el enlace o puntero mediante una flecha que sale de la caja y apunta al nodo siguiente.



$e_1, e_2, \ldots e_n$ son valores del tipo *TipoElemento*

Figura 9.2. Lista enlazada (representación gráfica típica).

$e_1, e_2, \ldots e_n$ son valores del tipo `TipoElemento`.

Estructura de una lista

Una **lista enlazada** consta de un número de elementos y cada elemento tiene dos componentes (*campos*), un puntero al siguiente elemento de la lista y un valor, que puede ser de cualquier tipo.

Los enlaces se representan por flechas para facilitar la comprensión de la conexión entre dos nodos; ello indica que el enlace tiene la dirección en memoria del siguiente nodo. Los enlaces también sitúan los nodos en una secuencia. En la Figura 9.2 los nodos forman una secuencia desde el primer elemento (e_1) al último elemento (e_n). El primer nodo se enlaza al segundo nodo, el segundo nodo se enlaza al tercero y así sucesivamente hasta llegar al último nodo. El nodo último ha de ser representado de forma diferente para significar que este nodo no se enlaza a ningún otro. La Figura 9.3 muestran diferentes representaciones gráficas que se utilizan para dibujar el campo enlace del último nodo.

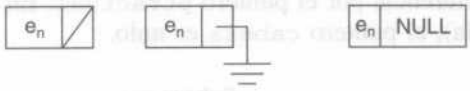


Figura 9.3. Diferentes representaciones gráficas del nodo último.

9.2. CLASIFICACIÓN DE LAS LISTAS ENLAZADAS

Las listas se pueden dividir en cuatro categorías:

- *Listas simplemente enlazadas*. Cada nodo (elemento) contiene un único enlace que conecta ese nodo al nodo siguiente o nodo sucesor. La lista es eficiente en recorridos directos («adelante»).
- *Listas doblemente enlazadas*. Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo («adelante») como en recorrido inverso («atrás»).
- *Lista circular simplemente enlazada*. Una lista enlazada simplemente en la que el último elemento (*cola*) se enlaza al primer elemento (*cabeza*) de tal modo que la lista puede ser recorrida de modo circular («en anillo»).
- *Lista circular doblemente enlazada*. Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular (*en anillo*) tanto en dirección directa («adelante») como inversa («atrás»).

Por cada uno de estos cuatro tipos de estructuras de listas, se puede elegir una implementación basada en arrays o una implementación basada en punteros. Como ya se ha comentado estas implementaciones difieren en el modo en que asigna la memoria para los datos de los elementos, cómo se enlazan juntos los elementos y como se accede a dichos elementos. De forma más específica, según la forma de reservar memoria las implementaciones pueden hacerse con:

- *Asignación fija*, o estática, de memoria mediante arrays.
- *Asignación dinámica* de memoria mediante punteros o apuntadores.

Dado que la asignación fija de memoria mediante arrays es más ineficiente, se utilizará en este capítulo, la asignación de memoria mediante punteros, dejando como ejercicio al lector la implementación mediante arrays.

Conceptos básicos sobre listas

Una lista enlazada consta de un conjunto de nodos. Un **nodo** consta de un campo dato y un puntero que apunta al «siguiente» elemento de la lista.

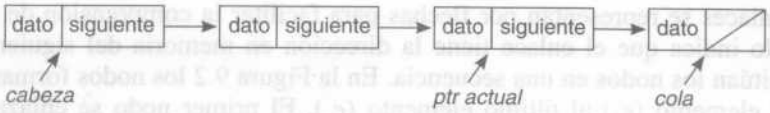


Figura 9.4. Representación gráfica de una lista enlazada.

El primer nodo, **frente**, es el nodo apuntado por **cabeza.** La lista encadena nodos juntos desde el frente al final (**cola**) de la lista. El final se identifica como el nodo cuyo campo puntero tiene el valor `NULL = 0`. La lista se recorre desde el primero al último nodo; en cualquier punto del recorrido la posición actual se referencia por el puntero `ptrActual`. En el caso en que la lista no contenga ningún nodo (está vacía), el puntero `cabeza` es nulo.

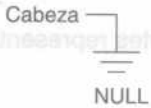


Figura 9.5. Representación de lista *vacía* con `cabeza == null`.

9.3. TIPO ABSTRACTO DE DATOS (TAD) LISTA

Una lista se utiliza para almacenar información del mismo tipo, con la característica de que puede contener un número indeterminado de elementos, y que estos elementos mantienen un orden explícito. Este ordenamiento explícito se manifiesta en que cada elemento contiene en sí mismo la dirección del siguiente elemento.

Cada elemento de una lista se denomina *nodo*. En un nodo podemos considerar que hay dos campos, campo de información y campo de enlace o dirección del elemento siguiente.

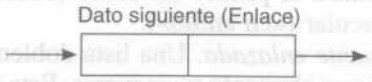


Figura 9.6. Representación de un nodo de una lista enlazada.

A una lista enlazada se accede desde un puntero externo que contiene la dirección (referencia) del primer nodo de la lista. El campo de dirección o enlace del último elemento de la lista no debe apuntar a ningún elemento, no debe tener ninguna dirección, por lo que contiene un valor especial denominado puntero nulo (*null*).

La *lista vacía*, aquella que no tiene nodos, tiene el puntero externo de acceso a la lista a nulo.

Una lista es una estructura de datos dinámica. El número de nodos puede variar rápidamente en un proceso. Aumentando los nodos por inserciones, o bien disminuyendo por borrado de nodos.

Las inserciones se pueden realizar por cualquier punto de la lista. Así, pueden realizarse por el comienzo de la lista, por el final de la lista, a partir o antes de un nodo determinado. Las eliminaciones también se pueden realizar en cualquier punto de la lista, aunque generalmente se hacen dando el campo de información o dato que se desea eliminar.

9.3.1. Especificación formal del TAD Lista

Matemáticamente, una lista es una secuencia de cero o más elementos de un determinado tipo.

`(a1, a2, a3, ... , an)` *siendo* `n >= 0`,
si `n = 0` *la lista es vacía.*

Los elementos de la lista tienen la propiedad de que sus elementos están ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que a_i precede a a_{i+1} para `i = 1 ...,` `n - 1;` y que a_i sucede a a_{i-1} para `i = 2 ... n`.

Para formalizar el tipo de datos abstracto `Lista` a partir de la noción matemática, se define un conjunto de operaciones básicas con objetos de tipo `Lista`. Las operaciones:

`∀ L ∈ Lista , ∀ x ∈ Lista , ∀ p ∈ puntero`

`Listavacia(L)`	Inicializa la lista `L` como lista vacía.
`Esvacia(L)`	Operación que determina si la lista `L` está vacía.
`Insertar(L,x,p)`	Inserta en la lista `L` un nodo con el campo dato `x`, delante del nodo de dirección `p`.
`Localizar(L,x)`	Operación que devuelve la posición/dirección donde está el campo de información `x`.
`Suprimir(L,x)`	Elimina de la lista el nodo que contiene el dato `x`.
`Anterior(L,p)`	Operación que devuelve la posición/dirección del nodo anterior a `p`.
`Primero(L)`	Operación que devuelve la posición/dirección del primer nodo de la lista `L`.
`Anula(L)`	Esta operación vacía la lista `L`.

Estas operaciones son las que pueden considerarse básicas para manejar listas. En realidad, la decisión de qué operaciones son las básicas depende de las características del problema que se va a resolver. También dependerá del tipo de representación elegido para las listas. Así, para añadir nuevos nodos a una lista se implementan, además de `insertar()`, versiones como:

`inserPrimero(L,x)`	Inserta un nodo con el dato `x` como primer nodo de la lista `L`.
`inserFinal(L,x)`	Inserta un nodo con el dato `x` como último nodo de la lista `L`.

Una operación típica de toda estructura enlazada de datos es *recorrer*, consistente en visitar cada uno de los datos o nodos de que consta, en las listas enlazadas, normalmente se realiza desde el nodo *cabeza* al último nodo o *cola* de la lista.

9.4. OPERACIONES EN LISTAS ENLAZADAS

Una lista enlazada requiere realizar la gestión de los elementos contenidos en ellas. Estos controles se manifiestan en forma de las operaciones básicas, especificadas al definir el tipo abstracto lista, que tendrán las siguientes funciones :

- Declaración de los tipos nodo y puntero a nodo.
- Inicialización o creación.
- Insertar elementos en una lista.
- Eliminar elementos de una lista.
- Buscar elementos de una lista (comprobar la existencia de elementos en una lista).
- Recorrer una lista enlazada (visitar cada nodo de la lista).
- Comprobar si la lista está vacía.

9.4.1. Declaración de un nodo

Una lista enlazada se compone de una serie de nodos enlazados mediante punteros. Cada nodo es una combinación de dos partes: un tipo de dato (entero, real, doble, carácter, o tipo predefinido) y un enlace (puntero) al siguiente nodo. En C, se puede declarar un nuevo tipo de dato, correspondiente a un nodo, mediante la especificación de `struct` que agrupa las dos partes citadas.

```
struct Nodo
{
  int dato;
  struct Nodo* enlace;
};
```

```
typedef struct Nodo
{
  int dato;
  struct Nodo *enlace;
}NODO;
```

La declaración utiliza el tipo `struct` que permite agrupar campos de diferentes tipos, el campo dato y el campo enlace. Con `typedef` se puede declarar a la vez un nuevo identificador de `struct Nodo`, en el caso anterior se ha elegido `NODO`.

Dado que los tipos de datos que se puede incluir en una lista pueden ser de cualquier tipo (enteros, dobles, caracteres o incluso cadenas), con el objeto de que el tipo de dato de cada nodo se pueda cambiar con facilidad, se suele utilizar la sentencia `typedef` para declarar el nombre de `elemento` como un sinónimo del tipo de dato de cada campo. El tipo `elemento` se utiliza entonces dentro de la estructura nodo, como se muestra a continuación:

```
typedef double elemento;

struct nodo
{
  elemento dato;
  struct nodo *enlace;
};
```

Entonces, si se necesita cambiar el tipo de elemento en los nodos, sólo tendrá que cambiar la sentencia de declaración de tipos que afecta a `elemento`. Siempre que una función necesite referirse al tipo del dato del nodo, puede utilizar el nombre `elemento`.

Ejemplo 9.1

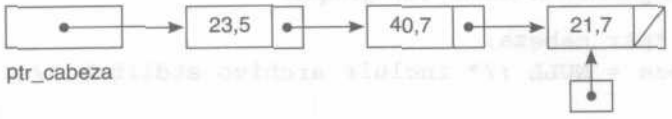
En este ejemplo se declara un tipo denominado `Punto`*, que representa un punto en el plano con su coordenada x e y. También se declara el tipo* `Nodo` *con el campo dato del tipo* `Punto`*. Por último, se define un puntero a* `Nodo`*.*

```
#include <stdlib.h>

typedef struct punto
{
  float x, y;
} PUNTO;

typedef struct nodo
{
  PUNTO dato;
  struct nodo* enlace;
}Nodo;

Nodo* cabecera;
cabecera = NULL;
```

9.4.2. Puntero de cabecera y cola

Normalmente, los programas no declaran variables de nodos. En su lugar, cuando se construye y manipula una lista enlazada, a la lista se accede a través de uno o más *punteros* a los nodos. El

acceso más frecuente a una lista enlazada es a través del primer nodo de la lista que se llama **cabeza** *o* **cabecera** *de la lista. Un puntero al primer nodo se llama* **puntero cabeza,** *o simplemente* **L.** *En* ocasiones, se mantiene también un puntero al último nodo de una lista enlazada. El último nodo es la **cola** de la lista, y un puntero al último nodo es el **puntero cola.** También se pueden mantener punteros otros nodos de una lista enlazada.



Declaración del nodo

```
typedef double elemento;
struct nodo
{
   elemento dato;
   struct nodo *enlace;
};
```

Definición de punteros

```
struct nodo *ptr_cabeza;

struct nodo *ptr_cola;
```

Figura 9.7. Declaraciones de tipos en lista enlazada.

Cada puntero a un nodo debe ser declarado como una variable puntero. Por ejemplo, si se mantiene una lista enlazada con un puntero de cabecera y otro de cola, se deben declarar dos variables puntero:

```
struct nodo *ptr _cabeza;
struct nodo *ptr_cola;
```

El tipo `struct nodo` a veces se simplifica utilizando la declaración `typedef`. Así se puede escribir :

```
typedef struct nodo Nodo;
typedef struct nodo* ptrNodo;
ptrNodo ptr_cabeza;
ptrNodo ptr_cola;
```

Acceso a una lista

La construcción y manipulación de una lista enlazada requiere el acceso a los nodos de la lista a través de uno o más punteros a nodos. Normalmente, un programa incluye un puntero al primer nodo (*cabeza*) y un puntero al último nodo (*cola*).

En cualquier forma, el último elemento de la lista contiene un campo de enlace con valor 0, esto es, un puntero nulo (NULL) que señala el final de la lista.

9.4.3. El puntero nulo

La Figura 9.8 muestra una lista con un puntero cabeza y un puntero nulo al final de la lista sobre el que se ha escrito la palabra NULL. La palabra NULL representa el **puntero nulo,** que es una constante especial de C. Se puede utilizar el puntero nulo para cualquier valor de puntero que no apunte a *ningún sitio. El puntero nulo se utiliza, normalmente, en dos situaciones:*

- Usar el puntero nulo en el campo enlace o siguiente del nodo final de una lista enlazada.
- Cuando una lista enlazada no tiene ningún nodo, se utiliza el puntero NULL como puntero de cabeza y de cola. Tal lista se denomina **lista vacá.**

En un programa, el puntero nulo se puede escribir como NULL, que es una constante de la biblioteca estándar `stdlib.h`[1]. El puntero nulo se puede asignar a una variable puntero con una sentencia de asignación ordinaria. Por ejemplo:

```
ptrNodo *ptr_cabeza;
ptr_cabeza = NULL ;/* incluir archivo stdlib.h */
```

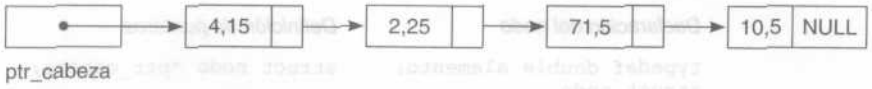


Figura 9.8. Puntero NULL.

El puntero de cabeza y de cola en una lista enlazada puede ser NULL, lo que indicará que la lista es vacía (no tiene nodos). Éste suele ser un método usual para construir una lista. Cualquier función que se escribe para manipular listas enlazadas debe poder manejar un puntero de cabeza y un puntero de cola nulos.

9.4.4. El operador -> de selección de un 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 considera como un operador simple (en vez de compuesto, al constar de dos símbolos independientes, – y >. Se denomina *operador de selección de miembro* o también *operador de selección de componente*. De modo visual el operador `p -> m` recuerda a una flecha que apunta del puntero `p` al objeto que contiene al miembro `m`.

Suponiendo que un programa ha de construir una lista enlazada y crear un puntero de cabecera `ptr_cabeza` a un nodo `Nodo`, el operador * de indirección aplicado a una variable puntero representa el contenido del nodo apuntado por `ptr_cabeza`. Es decir, `*ptr_cabeza` es un tipo de dato `Nodo`.

Al igual que con cualquier objeto, se puede acceder a los dos miembros de `*ptr_cabeza` en la Figura 9.8. Por ejemplo, la sentencia siguiente escribe los datos del nodo cabecera.

```
printf("%lf",(*ptr_cabeza).dato);
```

`(*ptr_cabeza)` miembro dato del nodo apuntado por `ptr_cabeza`

Precaución

Los paréntesis son necesarios alrededor de la primera parte de la expresión `(*ptr_cabeza)` ya que los operadores unitarios que aparecen a la derecha tienen prioridad más alta que los operadores unitarios que aparecen en el lado izquierdo (el asterisco de indirección).

[1] A veces algunos programadores escriben el puntero nulo como 0, pero pensamos es un estilo más claro escribirlo como NULL.

Sin los paréntesis, el significado de `ptr_cabeza` producirá un error de sintaxis, al intentar evaluar `ptr_cabeza.dato` antes de la indirección o desreferencia.

`p -> m` significa lo mismo que `(*p).m`

Utilizando el operador de selección `->` se pueden imprimir los datos del primer nodo de la lista

```
printf("%lf",ptr_cabeza->dato);
```

Error

Uno de los errores típicos en el tratamiento de punteros es escribir la expresión `*p` o bien `p->` cuando el valor del puntero `p` es el puntero nulo, ya que como se sabe el puntero nulo no apunta a nada.

9.4.5. Construcción de una lista

Una primera operación que se realiza con una lista es asignar un valor inicial; éste es el de *lista vacía*. Sencillamente consiste en asignar nulo al puntero de acceso a la lista:

```
Nodo* cabeza;
cabeza = NULL;
```

Para añadir elementos a la lista se utiliza la operación `insertar()`, con sus diferentes versiones. También, para la creación de una lista enlazada, se utiliza el siguiente algoritmo que entraña los siguientes pasos:

Paso 1. Declarar el tipo de dato y el puntero de cabeza o primero.
Paso 2. Asignar memoria para un elemento del tipo definido anteriormente utilizando alguna de las funciones de asignación de memoria (`malloc()`, `calloc()`, `realloc()`) y un *cast* para la conversión de `void*` al tipo puntero a nodo; la dirección del nuevo elemento es `ptr_nuevo`.
Paso 3. Crear iterativamente el primer elemento (cabeza) y los elementos sucesivos de una lista enlazada simplemente.
Paso 4. Repetir hasta que no haya más entradas para el elemento.

Ejemplo 9.2

Crear una lista enlazada de elementos que almacenen datos de tipo entero.

Un elemento de la lista se puede definir con la ayuda de la estructura siguiente:

```
struct Elemento
{
  int dato;
  struct Elemento * siguiente;
};
typedef struct Elemento Nodo;
```

En la estructura `Elemento` hay dos miembros, `dato` y `siguiente`, que es un puntero al siguiente nodo, y `dato`, que contiene el valor del elemento de la lista. También se declara un nuevo tipo: `Nodo` que es sinónimo de `struct Elemento`.

El siguiente paso para construir la lista es declarar la variable `primero` que apuntará al primer elemento de la lista:

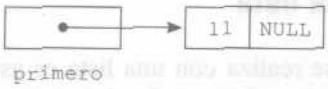
```
Nodo *primero = NULL        /* o bien primero = 0 */
```

El puntero `primero` (a veces se llama `cabeza`, o simplemente `L`) se ha inicializado a un valor nulo, lo que implica que la lista está vacía (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));
```

Con el operador `sizeof` se obtiene el tamaño de cada nodo de la lista, la función `malloc()` devuelve un puntero genérico (`void*`), por lo que se convierte a `Nodo*`. Ahora se puede asignar un valor al campo dato:

```
primero->dato = 11;
primero->siguiente = NULL;
```



El puntero `primero` apunta al nuevo elemento, que se inicializa a `11`. El campo `siguiente` del nuevo elemento toma el valor nulo, por no haber un nodo siguiente.

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:

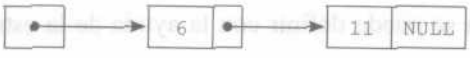
```
Nodo* crearNodo(int x, Nodo* enlace)
{
  Nodo *p;
  p = (Nodo*)malloc(sizeof(Nodo));
  p->dato = x;
  p->siguiente = enlace;
  return p;
}
```

La llamada a la función `crearNodo()` para crear el primer nodo de la lista:

```
primero = crearNodo(11, NULL);
```

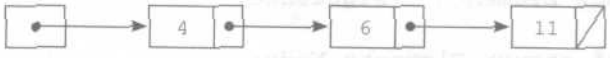
Si ahora se desea añadir un nuevo elemento con un valor `6`, y situarlo en el primer lugar de la lista se escribe simplemente:

```
primero = crearNodo(6,primero);
```



Por último, para obtener una lista compuesta de `4`, `6`, `11` se habría de ejecutar

```
primero = crearNodo(4,primero);
```



9.5. INSERCIÓN DE UN ELEMENTO EN UNA LISTA

El algoritmo empleado para añadir o insertar un elemento en una lista enlazada varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final de la lista (elemento último).
- Antes de un elemento especificado.
- Después de un elemento especificado.

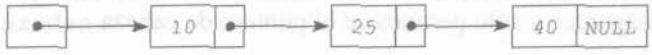
9.5.1. Insertar un nuevo elemento en la cabeza de una lista

Aunque normalmente se insertan nuevos datos al final de una estructura de datos, es más fácil y más eficiente insertar un elemento nuevo en la cabeza de una lista. El proceso de inserción se puede resumir en este algoritmo:

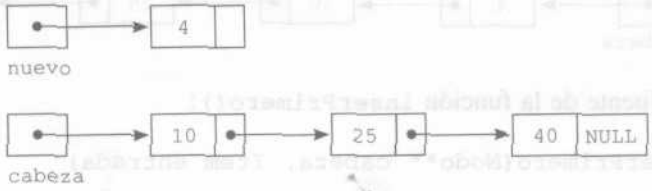
1. Asignar un nuevo nodo apuntado por `nuevo`, que es una variable puntero local que apunta al nuevo nodo que se va a insertar en la lista.
2. Situar el nuevo elemento en el campo `dato` del nuevo nodo.
3. Hacer que el campo enlace `siguiente` del nuevo nodo apunte a la cabeza (primer nodo) de la lista original.
4. Hacer que `cabeza` (puntero cabeza) apunte al nuevo nodo que se ha creado.

Ejemplo 9.3

Una lista enlazada contiene tres elementos, `10`, `25` y `40`. Insertar un nuevo elemento, 4, en cabeza de la lista.



Pasos 1 y 2



Código C

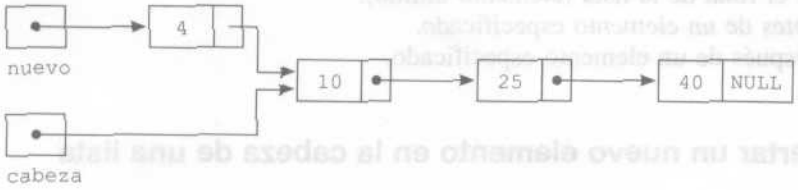
```
typedef int Item;
typedf struct nodo
{
  Item dato;
  struct nodo* siguiente;
} Nodo;                                  /* declaración del tipo Nodo */
Nodo* nuevo;
nuevo = (Nodo*)malloc(sizeof(Nodo));     /* se asigna un nuevo nodo */
nuevo -> dato = entrada;
```

Paso 3

El campo enlace (`siguiente`) del nuevo nodo apunta a la cabeza actual de la lista.

Código C

```
nuevo -> siguiente = cabeza;
```

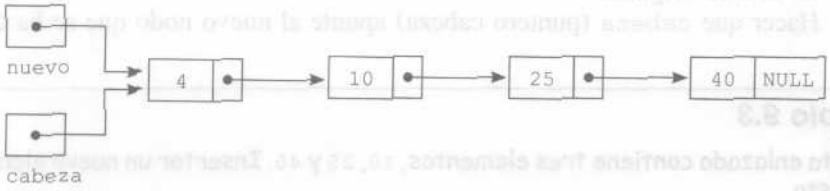


Paso 4

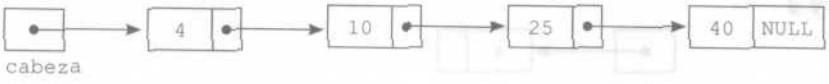
Se cambia el puntero de `cabeza` para apuntar al nuevo nodo creado; es decir, el puntero de `cabeza` apunta al mismo sitio que apunte `nuevo`.

Código C

```
cabeza = nuevo;
```



En este momento, la función de insertar un elemento en la lista termina su ejecución, la variable local `nuevo` desaparece y sólo permanece el puntero de cabeza `cabeza` que apunta a la nueva lista enlazada.



El código fuente de la función `inserPrimero()`:

```
void inserPrimero(Nodo** cabeza, Item entrada)
{
  Nodo *nuevo ;
  nuevo = (Nodo*)malloc(sizeof(Nodo));
  nuevo -> dato = entrada;              /* pone elemento en nuevo */
  nuevo -> siguiente = *cabeza;         /* enlaza con el primer nodo */
  *cabeza = nuevo;
}
```

Caso particular

La función `inserPrimero()` actúa también correctamente si se trata el caso de añadir un primer nodo o elemento a una lista vacía. En este caso, y como ya se ha comentado, `cabeza` apunta a `NULL` y termina apuntando al nuevo nodo de la lista enlazada.

Ejercicio 9.1

Se va a formar una lista enlazada de números aleatorios. El programa que realiza esta tarea inserta los nuevos nodos por la cabeza de la lista. Un vez creada la lista, se recorre los nodos para mostrar los número pares.

La función `inserPrimero()` añade un nodo a la lista, siempre como nodo cabeza. El primer argumento es un puntero a puntero porque tiene que modificar la variable `cabeza`, que es a su vez un puntero a `Nodo`. La función `crearNodo()` reserva memoria para un nodo, asigna el campo dato y devuelve la dirección del nodo creado.

La función `main()` inicializa primero el puntero `cabeza` a `NULL`, que es la forma de indicar *lista vacía*. En un bucle se generan números aleatorios, cada número generado se inserta como nuevo nodo en la lista, invocando a la función `inserPrimero()`; el bucle termina cuando se genera el 0. Una vez terminado el bucle de creación de la lista, otro bucle realiza tantas iteraciones como nodos tiene la lista, en cada pasada escribe el campo dato si es par.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 99
typedef int Item;
typedef struct Elemento
{
  Item dato;
  struct Elemento* siguiente;
} Nodo;

void inserPrimero(Nodo** cabeza, Item entrada);
Nodo* crearNodo(Item x);
void main()
{
  Item d;
  Nodo *cabeza, *ptr;
  int k;

  cabeza = NULL;                                  /* lista vacía */
  randomize();
    /* El bucle termina cuando se genera el número aleatorio 0 */
  for (d = random(MX); d; )
  {
    inserPrimero(&cabeza, d);
    d = random(MX);
  }
                /* se recorre la lista para escribir los pares */

  for (k = 0, ptr = cabeza; ptr; )
  {
    if (ptr -> dato%2 == 0)
    {
      printf("-%d ", ptr -> dato);
      k++;
```

```
        printf("%c",(k%12 ? ' ' : '\n'));          /* 12 datos por línea */
      }
      ptr = ptr -> siguiente;
    }
    printf("\n\n");
  }
  void inserPrimero(Nodo** cabeza, Item entrada)
  {
    Nodo *nuevo ;
    nuevo = crearNodo(entrada);
    nuevo -> siguiente = *cabeza;
    *cabeza = nuevo;
  }
  Nodo* crearNodo(Item x)
  {
    Nodo *a ;
    a = (Nodo*)malloc(sizeof(Nodo));                /* asigna nuevo nodo */
    a -> dato = x;
    a -> siguiente = NULL;
    return a;
  }
```

9.5.2. Inserción de un nodo al final de la lista

La inserción al final de la lista es menos eficiente debido a que, normalmente, no se tiene un puntero al último elemento de la lista y entonces se ha de seguir la traza desde la cabeza de la lista hasta el último nodo de la lista y a continuación realizar la inserción. Cuando `ultimo` es una variable puntero que apunta al último nodo de la lista, las sentencias siguientes insertan un nodo al final de la lista.

```
ultimo -> siguiente = crearNodo(x);
ultimo -> siguiente -> siguiente = NULL;
ultimo = ultimo -> siguiente;
```

La primera sentencia crea un nuevo nodo, llamada a `crearNodo()`, y se asigna al campo `siguiente` del último nodo de la lista (antes de la inserción) de modo que el nuevo nodo ahora es el último. La segunda sentencia establece el campo `siguiente` del nuevo último nodo a `NULL`. Y la última sentencia pone la variable `ultimo` al nuevo último nodo de la lista.

La función `inserFinal()` tiene como entrada el puntero cabeza, recorre la lista hasta situarse al final y realiza la inserción. Se tiene en cuenta la circunstancia de que la lista esté vacía, en cuyo caso inserta el nuevo nodo como nodo primero y único.

```
void inserFinal(Nodo** cabeza, Item entrada)
{
  Nodo *ultimo;
  ultimo = *cabeza;

  if (ultimo == NULL)                              /* lista vacía */
  {
```

```
    *cabeza = crearNodo(entrada);
  }
  else
  {
    for (; ultimo -> siguiente; )                    /* termina con ultimo
                                              referenciando al nodo final */
      ultimo = ultimo -> siguiente;

    ultimo -> siguiente = crearNodo(entrada);
  }
}
```

9.5.3. Inserción de un nodo entre dos nodos de la lista

La inserción de un nuevo nodo no siempre se realiza al principio (en cabeza) de la lista o al final se puede insertar entre dos nodos cualesquiera de la lista. Por ejemplo, se tiene una lista enlazada con los nodos `10`, `25`, `40` en la que se quiere insertar un nuevo elemento `75` entre el elemento `25` y el elemento `40`. La Figura 9.9 representa la lista y el nuevo nodo a insertar.

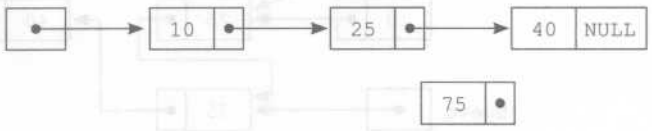


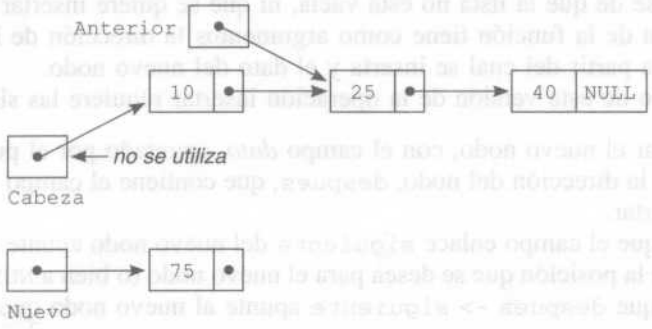
Figura 9.9. Inserción entre dos nodos

El algoritmo de la nueva operación insertar requiere las siguientes etapas:

1. Asignar el nuevo nodo, con el campo *dato,* apuntado por el puntero `nuevo`.
2. Hacer que el campo enlace `siguiente` del nuevo nodo apunte al nodo que va después de la posición que se desea para el nuevo nodo (o bien a `NULL` si no hay ningún nodo después de la nueva posición).
3. En la variable puntero `anterior` tener la dirección del nodo que está antes de la posición deseada para el nuevo nodo. Hacer que `anterior -> siguiente` apunte al nuevo nodo que se acaba de crear.

Etapa 1

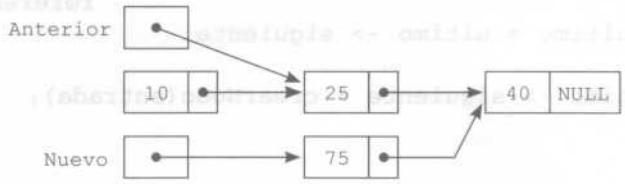
Se crea un nuevo nodo que contiene a `75`:



Código C

```
nuevo = crearNodo(entrada);
```

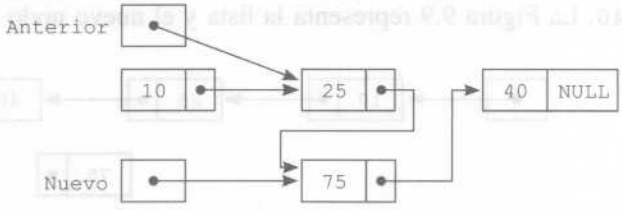
Etapa 2



Código C

```
nuevo -> siguiente = anterior -> siguiente
```

Etapa 3



Después de ejecutar todas las sentencias de las sucesivas etapas, la nueva lista comenzaría en el nodo `10`, seguiría `25`, `75` y por último `40`.

Código C

```
void insertar(Nodo* anterior, Item entrada)
{
  Nodo *nuevo;
    nuevo = crearNodo(entrada);
    nuevo -> siguiente = anterior -> siguiente;
    anterior -> siguiente = nuevo;
}
```

Para llamar a la función `insertar()`, previamente se ha de buscar la dirección del nodo anterior y asegurarse de que la lista no está vacía, ni que se quiere insertar como primer nodo.

Una versión de la función tiene como argumentos la dirección de inicio de la lista, `cabeza`, el campo dato a partir del cual se inserta y el dato del nuevo nodo.

El algoritmo de esta versión de la operación insertar requiere las siguientes etapas:

1. Asignar el nuevo nodo, con el campo *dato*, apuntado por el puntero `nuevo`.
2. Buscar la dirección del nodo, `despues`, que contiene el campo dato a partir del cual se ha de insertar.
3. Hacer que el campo enlace `siguiente` del nuevo nodo apunte al nodo que va a continuación de la posición que se desea para el nuevo nodo (o bien a `NULL` si no hay ningún nodo).
4. Hacer que `despues -> siguiente` apunte al nuevo nodo que se acaba de crear.

La primera comprobación, previa a las etapas a seguir, es que la lista esté vacía. Si es así se inserta como primer nodo. También se tiene que tener precaución en la búsqueda que se realiza en la etapa 2, ésta puede ser negativa, no se encuentra el dato; entonces no se inserta el nuevo nodo.

Código C

```
void insertar(Nodo** cabeza, Item testigo, Item entrada)
{
  Nodo *nuevo, *despues;

  nuevo = crearNodo(entrada);
  if (*cabeza == NULL)
    *cabeza = nuevo;
  else
  {
    int esta = 0;
    despues = *cabeza;                          /* etapa de búsqueda */
    while ((despues != NULL) && !esta)
    {
      if (despues -> dato != testigo)
        despues = despues -> siguiente;
      else
        esta = 1;
    }
    if (esta)
    {
      nuevo -> siguiente = despues -> siguiente;
      despues -> siguiente = nuevo;
    }
  }
}
```

9.6. BÚSQUEDA EN LISTAS ENLAZADAS

Dado que una función en C puede devolver un puntero, el algoritmo que sirva para localizar un elemento en una lista enlazada puede devolver un puntero a ese elemento.

La función `localizar()` utiliza una variable puntero denominada `indice` que va recorriendo la lista nodo a nodo. Mediante un bucle, `indice` apunta a los nodos de la lista de modo que si se encuentra el nodo buscado, se devuelve un puntero al nodo con la sentencia de retorno (`return`); en el caso de no encontrarse el nodo buscado la función debe devolver `NULL` (`return NULL`). El nodo que se busca es el que tiene un campo dato que coincide con una clave.

Código C

```
Nodo* localizar(Nodo* cabeza, Item destino)
/* cabeza: puntero de cabeza de una lista enlazada.
   destino: dato que se busca en la lista.
*/
{
  Nodo *indice;
```

```
    for (indice = cabeza; indice != NULL; indice = indice -> siguiente)
      if (destino == indice -> dato)
        return indice;
    return NULL;
  }
```

A tener en cuenta

La operación de búsqueda puede tener diferentes enfoques: búsqueda de la posición del primer nodo que contiene un dato, búsqueda de la posición de un nodo según el número de orden, determinación si existe o no un nodo con un campo dato determinado.

Ejemplo 9.4

Escribir una función que encuentre la dirección de un nodo dado el orden que ocupa en una lista enlazada.

El nodo o elemento se especifica por su número de orden en la lista; para ello se considera posición 1, la correspondiente al nodo de cabeza, posición 2, la correspondiente al siguiente nodo, y así sucesivamente.

El algoritmo de búsqueda del elemento comienza con el recorrido de la lista mediante un puntero `indice` que comienza apuntando al nodo cabeza de la lista. Un bucle mueve el `indice` hacia adelante el número correcto de sitios (lugares). A cada iteración del bucle se mueve el puntero `indice` un nodo hacia adelante. El bucle termina cuando se alcanza la posición deseada e `indice` apunta al nodo correcto. El bucle también se puede terminar si `indice` apunta a `NULL`, lo que indicará que la posición solicitada era más grande que el número de nodos de la lista.

```
Nodo* buscarPosicion(Nodo *cabeza, size_t posicion)
/* El programa que llame a esta función ha de incluir
   biblioteca stdlib.h (para implementar tipo size_t)
*/
{
  Nodo *indice;
  size_t i;

  if (posicion < 1)                /* posición ha de ser mayor que 1 */
    return NULL;

  indice = cabeza;
  for (i = 1 ;(i < posicion) && (indice != NULL) ; i++)
    indice = indice -> siguiente;

  return indice;
}
```

9.7. ELIMINACIÓN DE UN NODO EN UNA LISTA

La operación de eliminar (borrar) un nodo de una lista enlazada supone enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y liberar la memoria que ocupa.

El algoritmo para eliminar un nodo que contiene un dato se puede expresar en estos pasos o etapas:

1. Búsqueda del nodo que contiene el dato (se ha de tener la dirección del nodo a eliminar y la dirección del anterior).
2. El puntero `siguiente` del nodo anterior ha de apuntar al `siguiente` del nodo a eliminar.
3. En caso de que el nodo a eliminar sea el primero, `cabeza`, se modifica `cabeza` para que tenga la dirección del nodo `siguiente`.
4. Por último, se libera la memoria ocupada por el nodo.

A continuación se escribe una función que recibe la dirección de inicio, cabeza, de la lista y el dato del nodo que se quiere borrar.

Código C

```
void eliminar (Nodo** cabeza, Item entrada)
{
  Nodo* actual, *anterior;
  int encontrado = 0;

  actual = *cabeza; anterior = NULL;
    /* búsqueda del nodo y del anterior */
  while ((actual!=NULL) && (!encontrado))
  {
    encontrado = (actual -> dato == entrada);
    if (!encontrado)
    {
      anterior = actual;
      actual = actual -> siguiente;
    }
  }
  /* enlace de nodo anterior con siguiente */
  if (actual != NULL)
  {
                /* distingue entre que el nodo sea el cabecera o del
                                                    resto de la lista */
    if (actual == *cabeza)
      *cabeza = actual -> siguiente;
    else
      anterior -> siguiente = actual -> siguiente;

    free(actual);
  }
}
```

Ejercicio 9.2

Se desea crear una lista enlazada de números enteros ordenada. La lista se organiza de tal forma que el nodo cabecera tenga el menor elemento, y así en orden creciente los demás nodos. Una vez

creada la lista, se puede recorrer para escribir los datos por pantalla; también se ha de poder eliminar un nodo con un dato determinado.

Inicialmente la lista se crea con el primer valor. El segundo elemento se ha de insertar antes del primero o después, dependiendo de que sea menor o mayor. Así en general, para insertar un nuevo elemento, primero se busca la posición de inserción en la lista actual, que en todo momento está ordenada, del nodo a partir del cual se ha de enlazar el nuevo nodo para que la lista siga ordenada. Las diversas formas de realizar la inserción un nodo en una lista estudiadas en el Apartado 9.5 no se ajustan exactamente a la funcionalidad requerida, por ello se escribe la función `insertaOrden()` para añadir los nuevos elementos en el orden requerido. Los datos con los que se va a formar la lista se generan aleatoriamente.

La función `recorrer()` avanza por cada uno de los nodos de la lista con la finalidad de escribir el campo dato. Para eliminar nodos, se invoca a la función `eliminar()` escrita en el Apartado 9.7 con una pequeña modificación: «escribir un literal en el caso de que el nodo no esté en la lista».

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MX 101
typedef int Item;
typedef struct Elemento
{
  Item dato;
  struct Elemento* siguiente;
}Nodo;

void insertaOrden(Nodo** cabeza, Item entrada);
Nodo* crearNodo(Item x);
void recorrer(Nodo* cabeza);
void eliminar (Nodo** cabeza, Item entrada);

void main()
{
  Item d;
  Nodo* cabeza;

  cabeza = NULL;                    /* lista vacía */
  randomize();
            /* bucle termina cuando se genera el número aleatorio 0 */
  for (d = random(MX); d; )
  {
    insertaOrden(&cabeza,d);
    d = random(MX);
  }

  recorrer(cabeza);

  printf("\n Elemento a eliminar: ");
  scanf("%d", &d);
  eliminar(&cabeza,d);
  recorrer(cabeza);
```

```
}

void insertaOrden(Nodo** cabeza, Item entrada)
{
  Nodo *nuevo;

  nuevo = crearNodo(entrada);

  if (*cabeza == NULL)
    *cabeza = nuevo;
  else if (entrada < (*cabeza) -> dato)              /* primer nodo */
  {
    nuevo -> siguiente = *cabeza;
    *cabeza = nuevo;
  }
  else          /* búsqueda del nodo anterior a partir del que
                                                  se debe insertar */
  {
    Nodo* anterior, *p;
    anterior = p = *cabeza;

    while ((p -> siguiente != NULL) && (entrada > p -> dato))
    {
      anterior = p;
      p = p -> siguiente;
    }

    if (entrada > p -> dato)                /* inserta por el final */
      anterior = p;

                                /* Se procede al enlace del nuevo nodo */
      nuevo -> siguiente = anterior -> siguiente;
      anterior -> siguiente = nuevo;
    }
  }

  Nodo* crearNodo(Item x)
  {
    Nodo *a ;
    a = (Nodo*)malloc(sizeof(Nodo));
    a -> dato = x;
    a -> siguiente = NULL;
    return a;
  }

  void recorrer(Nodo* cabeza)
  {
    int k;
    printf("\n\t\t Lista Ordenada \n");
    for (k = 0; cabeza; cabeza = cabeza -> siguiente)
    {
      printf("%d ",cabeza -> dato);
```

```
      k++;
      printf("%c",(k%15 ?' ':'\n'));
    }
    printf("\n\n");
  }

  void eliminar (Nodo** cabeza, Item entrada)
  {
    Nodo* actual, *anterior;
    int encontrado = 0;

    actual = *cabeza; anterior = NULL;
    while ((actual!=NULL) && (!encontrado))
    {
      encontrado = (actual -> dato == entrada);
      if (!encontrado)
      {
        anterior = actual;
        actual = actual -> siguiente;
      }
    }
    if (actual != NULL)
    {
      if (actual == *cabeza)
        *cabeza = actual -> siguiente;
      else
        anterior -> siguiente = actual -> siguiente;

      free(actual);
    }
    else
      puts("Nodo no eliminado, elemento no esta en la lista ");
  }
```

9.8. LISTA DOBLEMENTE ENLAZADA

Hasta ahora el recorrido de una lista se realizaba en sentido directo (*adelante*) o, en algunos casos, en sentido inverso (*hacia atrás*). Sin embargo, existen numerosas aplicaciones en las que es conveniente poder acceder a los elementos o nodos de una lista en cualquier orden. En este caso se recomienda el uso de una **lista doblemente enlazada.** En tal lista, cada elemento contiene dos punteros, aparte del valor almacenado en el elemento. Un puntero apunta al siguiente elemento de la lista y el otro apunta al elemento anterior. La Figura 9.10 muestra una lista doblemente enlazada y un nodo de dicha lista.

Las operaciones básicas que se definen en el tipo abstracto *lista doble* son básicamente las mismas que las definidas en una lista simple y tendrán las siguientes funciones:

- Declaración de los tipos nodo y puntero a nodo.
- Inicialización o creación.
- Insertar elementos en una lista.
- Eliminar elementos de una lista.
- Buscar elementos de una lista (comprobar la existencia de elementos en una lista).

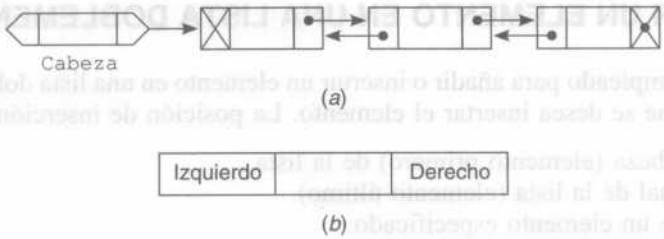


Figura 9.10. Lista doblemente enlazada. *a*) Lista con tres nodos; *b*) nodo.

- Recorrer una lista enlazada (visitar cada nodo de la lista).
- Comprobar si la lista está vacía.

Existe una operación de *insertar* y *eliminar* (borrar) en cada dirección. La Figura 9.11 muestra el problema de insertar un nodo *p* a la derecha del nodo actual, en esta operación deben asignarse cuatro nuevos enlaces.

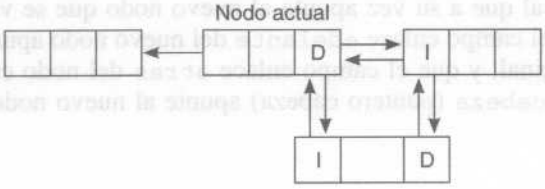


Figura 9.11. Inserción de un nodo en una lista doblemente enlazada.

En el caso de eliminar (borrar) un nodo de una lista doblemente enlazada es preciso cambiar dos punteros.

Figura 9.12. Eliminación de un nodo en una lista doblemente enlazada.

9.8.1. Declaración de una lista doblemente enlazada

Una lista doblemente enlazada con valores de tipo `int` necesita dos punteros y el valor del campo de datos. En una estructura se agrupan estos datos del modo siguiente:

```
typedef int Item;
struct unnodo
{
  Item dato;
  struct unnodo *adelante;
  struct unnodo *atras;
};
typedef struct unnodo Nodo;
```

9.9. INSERTAR UN ELEMENTO EN UNA LISTA DOBLEMENTE ENLAZADA

El algoritmo empleado para añadir o insertar un elemento en una lista doble varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final de la lista (elemento último).
- Antes de un elemento especificado.
- Después de un elemento especificado.

A continuación se desarrolla la inserción de un elemento en la cabeza y después de un elemento especificado.

9.9.1. Insertar un nuevo elemento en la cabeza de una lista doble

El proceso de inserción se puede resumir en este algoritmo:

1. Asignar un nuevo nodo el campo `dato` (`Info`) apuntado por `nuevo` que es una variable puntero local que a su vez apunta al nuevo nodo que se va a insertar en la lista doble.
2. Hacer que el campo enlace `adelante` del nuevo nodo apunte a la cabeza (primer nodo) de la lista original, y que el campo enlace `atras` del nodo cabeza apunte al nuevo nodo.
3. Hacer que `cabeza` (puntero cabeza) apunte al nuevo nodo que se ha creado.

Código C

```
typedef int Item;
typedf struct tipoNodo
{
  Item dato;
  struct tipoNodo* adelante;
  struct tipoNodo* atras;
}Nodo;

Nodo* nuevo;

nuevo = crearNodo(entrada);

nuevo -> adelante = cabeza;
nuevo -> atras = NULL;
cabeza -> atras = nuevo;
cabeza = nuevo;
```

En este momento, la función de insertar un elemento en la lista termina su ejecución, la variable local `nuevo` desaparece y sólo permanece el puntero de cabeza, `cabeza`, que apunta a la nueva lista doblemente enlazada.

9.9.2. Inserción de un nuevo nodo que no está en la cabeza de lista

La inserción de un nuevo nodo en una lista doblemente enlazada se puede realizar en un nodo intermedio de ella. El algoritmo de la nueva operación insertar requiere las siguientes etapas:

1. Asignar el nuevo nodo el campo *dato* (*Info*) apuntado por el puntero `nuevo`.

2. Hacer que el campo enlace `adelante` del nuevo nodo apunte al nodo que va después de la posición del nuevo nodo (o bien a `NULL` si no hay ningún nodo después de la nueva posición). El campo `atras` del nodo siguiente al nuevo tiene que apuntar a `nuevo`.
3. La dirección del nodo que está antes de la posición deseada para el nuevo nodo está en la variable puntero `anterior`. Hacer que `anterior -> adelante` apunte al nuevo nodo. El enlace `atras` del nuevo nodo debe de apuntar a `anterior`.

En la función que se escribe a continuación, se tiene en cuenta que la inserción sea en una lista vacía, en cuyo caso `cabeza` es `NULL`, o que sea como primer nodo. De igual forma se contempla que la inserción sea como último nodo.

Código C

```
void insertar(Nodo** cabeza, Item entrada, Nodo* anterior)
{
  Nodo *nuevo;

  if ((*cabeza) == NULL) || anterior == NULL)
  {
    inserPrim(&cabeza,entrada);
  }
  else {
    nuevo = crearNodo(entrada) ;
    nuevo -> adelante = anterior -> adelante;
    if (anterior -> adelante != NULL)        /* no es fin de lista */
      anterior -> adelante -> atras = nuevo;
                  /* atras del siguiente apunta al nodo nuevo creado */
    anterior -> adelante = nuevo;
    nuevo -> atras = anterior;
  }
}
```

9.10. ELIMINACIÓN DE UN ELEMENTO EN UNA LISTA DOBLEMENTE ENLAZADA

La operación de eliminar un nodo de una lista doble supone realizar el enlace de dos punteros: el nodo anterior con el nodo siguiente al que se desea eliminar, con el puntero `adelante`; y el nodo siguiente con el anterior, con el puntero `atras`, y liberar la memoria que ocupa.

El algoritmo para eliminar un nodo que contiene un dato es similar al algoritmo de eliminación en una lista simple. Ahora la dirección del nodo anterior se encuentra en el puntero `atras` del nodo a eliminar o borrar. Los pasos a seguir:

1. Búsqueda del nodo que contiene el dato. Se ha de tener la dirección del nodo a eliminar y la dirección del anterior.
2. El puntero `adelante` del nodo anterior tiene que apuntar al puntero `adelante` del nodo a eliminar (en el caso de no ser el nodo cabecera).
3. El puntero `atras` del nodo siguiente a eliminar debe apuntar al puntero `atras` del nodo a eliminar (en el caso de no ser el nodo último).
4. En caso de que el nodo a eliminar sea el primero, `cabeza`; se modifica `cabeza` para que tenga la dirección del nodo `siguiente`.
5. Por último, se libera la memoria ocupada por el nodo.

La codificación del algoritmo se presenta en la función `eliminar()`:

Código C

```
void eliminar (Nodo** cabeza, Item entrada)
{
  Nodo* actual;
  int encontrado = 0;

  actual = *cabeza;
    /* búsqueda del nodo */
  while ((actual!=NULL) && (!encontrado))
  {
    encontrado = (actual -> dato == entrada);
    if (!encontrado)
      actual = actual -> adelante;
  }
    /* Enlace de nodo anterior con siguiente */
  if (actual != NULL)
  {
            /* distingue entre el nodo cabecera y el
                                                resto de la lista */
    if (acual == *cabeza)
    {
      *cabeza = actual -> adelante;
      if (actual -> adelante != NULL)
        actual -> adelante -> atras = NULL;
    }
    else if (actual -> adelante != NULL) /* No es el último nodo */
    {
      actual -> atras -> adelante = actual -> adelante;
      actual -> adelante -> atras = actual -> atras;
    }
    else {                                  /* último nodo */
      actual -> atras -> adelante = NULL;
    }
    free(actual);
  }
}
```

Ejercicio 9.3

Escribir un programa que crea una lista doblemente enlazada con palabras introducidas por teclado. Una vez creada la lista, se eliminan los nodos que contienen una palabra cuya primera letra coincide con una letra dada.

Las palabras que se van a introducir se leen con la función `gets()`. El tipo del campo dato de los nodos es `char*`, de tal forma que al asignar la palabra leída se reserva memoria ajustada al número de caracteres que tiene la palabra:

```
dato = (char*)malloc((strlen(palabra)+1)*sizeof(char));
strcpy(dato,palabra);
```

La inserción de elementos en la lista se hace por el nodo cabecera, por lo que la función que se utiliza para añadir nuevos elementos es `inserPrim()`.

El número de elementos de la lista se requiere para ser introducido por teclado. También, una vez formada la lista, se pide la letra de la inicial de la palabras que van a ser eliminadas.

Para eliminar los elementos se recorre la lista, los nodos que contienen una palabra cuya inicial coincide con la letra requerida se borran de la lista. Para borrar los nodos se llama a la función `eliminarDir()`, que es una versión de `eliminar()` para aprovechar que ya se dispone de la dirección del nodo a suprimir.

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

typedef char* Item;
typedef struct tipoNodo
{
  Item dato;
  struct tipoNodo* adelante;
  struct tipoNodo* atras;
}Nodo;

void inserPrim(Nodo** cabeza, Item entrada);
Nodo* crearNodo(Item x);
void eliminarDir(Nodo** cabeza, Nodo* actual);
void recorrer(Nodo* ptr);

void main()
{
  Nodo *cabeza,*ptr;
  int n;
  char c, palabra[81];

  cabeza = NULL;              /* lista vacía */
  printf("\n Número de elementos a generar: ");
  scanf("%d%*c",&n);

  printf("\nIntroducir %d palabras \n",n);
  for ( ; n--; )
  {
    gets(palabra);
    inserPrim(&cabeza,palabra);
  }

  recorrer(cabeza);           /* muestra palabras de la lista */

  printf("\nInicial de las palabras a eliminar: ");
  scanf("%c",&c);

  for (ptr = cabeza; ptr; )
  {
    if ((ptr -> dato[0]) == c) /* nodo ptr se borra de la lista */
    {
      Nodo* t;
```

```
        t = ptr -> adelante;  /* Guarda el nodo por el que seguir */
        eliminarDir(&cabeza,ptr);
        ptr = t;
      }
      else
        ptr = ptr -> adelante;
    }

    recorrer(cabeza);  /* muestra palabras que quedan en la lista */
  }

  void eliminarDir (Nodo** cabeza, Nodo* actual)
  {

    /* Elimina el nodo de dirección actual */
    if (actual == *cabeza)
    {
      *cabeza = actual -> adelante;
      if (actual -> adelante != NULL)
        actual -> adelante -> atras = NULL;
    }
    else if (actual -> adelante != NULL)
    {
      actual -> atras -> adelante = actual -> adelante;
      actual -> adelante -> atras = actual -> atras;
    }
    else {
      actual -> atras -> adelante = NULL;
    }
    free(actual);
  }

  void recorrer(Nodo* ptr)
  {
    int k = 0;

    printf("\n\n\t Elementos de la lista\n");

    for ( ; ptr ; )
    {
      k++;
      printf("%s",ptr -> dato);
      printf("%c",(k%12 == 0 ? '\n' : ' '));
      ptr = ptr -> adelante;
    }
  }

  void inserPrim(Nodo** cabeza, Item entrada)
  {
    Nodo* nuevo;

    nuevo = crearNodo(entrada);
    nuevo -> adelante = *cabeza;
```

```
    nuevo -> atras = NULL;
    if (*cabeza != NULL)
        (*cabeza) -> atras = nuevo;
    *cabeza = nuevo;
}
Nodo* crearNodo(Item x)
{
    Nodo *a ;
    a = (Nodo*)malloc(sizeof(Nodo));
    a -> dato = (Item) malloc((strlen(x)+1)*sizeof(char));
    strcpy(a -> dato,x);
    a -> adelante = a -> atras = NULL;
    return a;
}
```

9.11. LISTAS CIRCULARES

En las listas lineales simples o en las dobles siempre hay un primer nodo y un último nodo que tiene el campo de enlace a nulo. Una lista circular, por propia naturaleza no tiene ni principio ni fin. Sin embargo, resulta útil establecer un nodo a partir del cual se acceda a la lista y así poder acceder a sus nodos.

La Figura 9.13 muestra una lista circular con enlace simple; podría considerarse como una lista lineal, de tal manera que el último nodo apunta al primero.

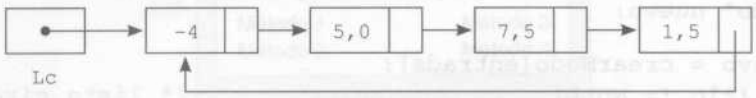


Figura 9.13. Lista circular.

Las operaciones que se realizan sobre una lista circular son similares a las operaciones sobre listas lineales, teniendo en cuenta que el último nodo no apunta a nulo sino al primero. Estas operaciones permiten construir el TAD *Lista Circular* y su funcionalidad:

- Declaración de los tipos nodo y puntero a nodo.
- Inicialización o creación.
- Insertar elementos en una lista circular.
- Eliminar elementos de una lista circular.
- Buscar elementos de una lista circular.
- Recorrer o visitar cada nodo de la lista circular.
- Comprobar si la lista está vacía.

La creación de una lista circular se puede hacer con un enlace simple o un enlace doble. Consideramos que la lista circular se enlaza con un solo enlace, la realización con enlace adelante y atrás es similar (se puede consultar el Apartado 9.9).

A tener en cuenta

El puntero de acceso a una lista circular, `lc`, normalmente apunta al último nodo añadido a la estructura. Esta convención puede cambiar ya que en una estructura circular no hay *primero* ni *último*.

9.11.1. Insertar un elemento en una lista circular

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. La posición de inserción puede variar, consideramos que se hace como nodo anterior al del nodo de acceso a la lista `lc`, y que `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;
}
```

9.11.2. Eliminar un elemento en una lista circular

La operación de eliminar un nodo de una lista circular sigue los mismos pasos que los dados para eliminar un nodo en una lista lineal. Hay que enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y liberar la memoria que ocupa.

El algoritmo para eliminar un nodo de una lista circular es:

1. Búsqueda del nodo que contiene el dato.
2. Se enlaza el nodo anterior con el siguiente.
3. En caso de que el nodo a eliminar sea el referenciado por el puntero de acceso a la lista, `lc`, se modifica `lc` para que tenga la dirección del nodo anterior.
4. Por último, se libera la memoria ocupada por el nodo.

En la función eliminar hay que tener en cuenta la característica de lista circular; así, para detectar si la lista es de un solo nodo se pregunta si se apunta al mismo, `lc == lc -> siguiente` si esta expresión es cierta la lista consta de un solo nodo.

A continuación se escribe el código de la función `eliminar` para una lista circular. Para ello se recorre la lista con un puntero al nodo anterior; por esa razón se accede al dato con la sentencia `actual -> siguiente -> dato`. Esto va a permitir, caso de encontrarse el nodo, tener en `actual` el nodo anterior y así poder enlazar con el nodo siguiente a borrar. Después del bucle es necesario volver a preguntar por el campo dato, ya que no se comparó el nodo `lc` y el bucle puede haber terminado sin encontrar el nodo:

Código C

```
void eliminar (Nodo** lc, Item entrada)
{
  Nodo* actual;
  int encontrado = 0;

  if ((*lc) == NULL) return;          /* lista vacía */

  actual = *lc;
    /* Bucle de búsqueda */
  while ((actual -> siguiente != *lc) && (!encontrado))
  {
    encontrado = (actual -> siguiente -> dato == entrada);
    if (!encontrado)
    {
      actual = actual -> siguiente;
    }
  }
  encontrado = (actual -> siguiente -> dato == entrada);

  /* Enlace de nodo anterior con siguiente */

  if (encontrado)
  {
    Nodo* p;
    p = actual -> siguiente;          /* Nodo a eliminar */
    if (*lc == (*lc) -> siguiente)    /* Lista con un solo nodo */
      *lc = NULL;
    else
    {
      if (p == *lc)
      {
        *lc = actual; /* Se borra el elemento referenciado por lc;
                         el nuevo acceso a la lista es el anterior */
      }
      actual -> siguiente = p -> siguiente;
    }
    free(p);
  }
}
```

Ejercicio 9.4

Se desea guardar una lista de números complejos en una estructura circular, de tal forma que cada nodo de la estructura contenga un número complejo. Una vez creada la lista, el programa debe ofrecer un conjunto de operaciones:

- Suma de los números complejos que se encuentran en la lista.
- Eliminación de los nodos de la lista que cumplan la condición de que el módulo es menor que un valor dado.
- Búsqueda del nodo que contenga un número complejo determinado.

El propio ejercicio indica el campo *dato* de los nodos de la lista circular, un número complejo que consta de dos números reales `(x, y)`. El acceso a la lista va a ser un puntero, `lc`, al último nodo insertado, para lo cual se invocará la función `insertaCircular()` para añadir nuevos números complejos a la lista.

Las operaciones que se pueden realizar con los números de la lista suponen recorrer la lista, en la primera operación para sumar los número complejos. En la segunda operación, para eliminar los nodos que cumplen la condición del módulo y para ello es necesario un puntero auxiliar, `actual`, con las dirección del nodo previo para que utilizando el campo `siguiente : actual->siguiente`, se compare el módulo del complejo del nodo que se pretende procesar, `actual->siguiente->dato`; así, poder realizar el enlace del nodo anterior con el nodo siguiente a eliminar. La condición para terminar es que se haya recorrido todos los nodos, `actual->siguiente == lc`, pero aún quedará por determinar si el propio nodo `lc` hay que eliminarlo.

La operación de búsqueda recorre la lista comparando el campo dato de los nodos con el número complejo buscado; devuelve un puntero al nodo encontrado, o bien NULL si es negativa la búsqueda.

Todo recorrido de la estructura circular se inicia en el nodo siguiente al nodo por el que se accede, `lc`, ya que se sigue el convenio de que el acceso a la lista es por el último nodo añadido. El recorrido termina cuando se procesa el nodo `lc` , de esta forma se ha visitado todos los nodos que forman la estructura circular.

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

typedef struct
{
   float x;
   float y;
}Complejo;
typedef Complejo Item;

typedef struct Elemento
{
 Item dato;
 struct Elemento* siguiente;
}Nodo;
  /* prototipo de funciones */
void insertaCircular(Nodo** lc, Item entrada);
Complejo sumarLista(Nodo* lc);
void borrarLista(Nodo** lc, float modulo);
Nodo* buscar(Nodo* lc, Complejo z);
void mostrarLista(Nodo* lc);
   /* funciones auxiliares */
Nodo* crearNodo(Item x);
Complejo entrada();

#define IGUAL(c1,c2) (c1.x == c2.x && c1.y == c2.y)
```

```
void main()
{
  Nodo* lc;
  int opc;
  float m;
  Complejo z, clave = {0.0,0.0};

  lc = NULL;              /* Lista vacía */

  printf("\n Entrada de números complejos. Termina con (0,0) \n");
  do {
    z = entrada();
    if (!IGUAL(z,clave))
    insertaCircular(&lc,z);
  } while (!IGUAL(z,clave));

  puts("\n\t Operaciones con la lista");

  do {
    puts("\n 1. Suma de los números complejos de lista.\n");
    puts("\n 2. Eliminar nodos de módulo menor.\n");
    puts("\n 3. Búsqueda de un número complejo.\n");
    puts("\n 4. Salir\n");

    do {
      scanf("%d",&opc);
    }while (opc < 1 || opc > 4);

    switch (opc) {
    case 1: z = sumarLista(lc);
            printf("Complejo suma: (%.1f, %.1f) \n", z.x,z.y);
            break;
    case 2: printf("\nMódulo por debajo del que se eliminan: ");
            scanf("%f",&m);
            borrarLista(&lc,m);
            break;
    case 3: printf("\nNúmero complejo a buscar: ");
            z = entrada();
            if (buscar(lc,z))
              puts("El número complejo está en la lista");
            else
              puts("El número complejo NO está en la lista");
            break;
    }
    mostrarLista(lc);
  } while (opc != 4);

}

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;
}

Complejo sumarLista(Nodo* lc)
{
  Complejo s;

  s.x = 0; s.y = 0;
  if (lc != NULL)
  {
    Nodo* p = lc -> siguiente;
    while (p != lc)
    {
      s.x += p -> dato . x;
      s.y += p -> dato . y;
      p = p -> siguiente;
    }
    s.x += lc -> dato . x;      /* falta por sumar último nodo */
    s.y += lc -> dato . y;
  }
  return s;
}

void borrarLista(Nodo** lc, float modulo)
{
  #define MOD(z) sqrt(z.x*z.x + z.y*z.y)
  Nodo* actual;

  if ((*lc) == NULL) return;          /* lista vacía */
  actual = *lc;
    /* se recorre la lista desde el nodo anterior */
  while (actual -> siguiente != *lc)
  {
    if (MOD(actual -> siguiente -> dato) < modulo)
    {
      Nodo* p;
      p = actual -> siguiente;          /* Nodo a eliminar */
      actual -> siguiente = p -> siguiente;
      free(p);
    }
    else
      actual = actual -> siguiente;
  }
  /* el nodo de acceso a la lista, lc, no se ha procesado */
```

```
    if (MOD(actual -> siguiente -> dato) < modulo)
    {
      Nodo* p;
      p = actual -> siguiente;          /* Nodo a eliminar */
      if (*lc == (*lc) -> siguiente)    /* Lista con un solo nodo */
        *lc = NULL;
      else
      {
        *lc = actual; /* nuevo acceso a la lista es el anterior */
        actual -> siguiente = p -> siguiente;
      }
      free(p);
    }
  }
  Nodo* buscar(Nodo* lc, Complejo z)
  {
    Nodo* actual; int encontrado = 0;

    if (lc == NULL) return NULL;          /* lista vacía */

    actual = lc -> siguiente;
    do
    {
      encontrado = IGUAL(actual -> dato, z);
      if (!encontrado)
      {
        actual = actual -> siguiente;
      }
    } while ((actual!= lc -> siguiente) && (!encontrado));
    if (encontrado)
      return actual;
    else
      return NULL;
  }

  void mostrarLista(Nodo* lc)
  {
    Nodo* p;
    if (lc != NULL)
    {
      puts("\nLista de números complejos de la lista");
      p = lc -> siguiente; /* se recorre a partir del primer nodo */
      do {
        int k = 0;
        k++;
        printf("(%.1f, %.1f) \t", p -> dato . x, p -> dato . y);
        if (k%7 == 0) puts("");
        p = p -> siguiente;
      }while(p != lc->siguiente);
    }
```

```
    else
      printf("\n\t Lista vacía.\n");
}

Nodo* crearNodo(Item x)
{
  Nodo *a ;
  a = (Nodo*)malloc(sizeof(Nodo));
  a -> dato = x;
  a -> siguiente = a;
  return a;
}

Complejo entrada()
{
  Complejo z;
  printf("Parte real: "); scanf("%f",&z.x);
  printf("Parte imaginaria: ");scanf("%f",&z.y);
  return z;
}
```

RESUMEN

La estructura de datos lista se puede implementar, bien como un array, bien como una lista enlazada.Una **lista enlazada** es una estructura de datos dinámica en la que sus componentes están ordenados lógicamente por sus campos punteros en vez de ordenados físicamente como están en un array. El final de la lista se señala mediante una constante o puntero especial llamado NULL.

La gran ventaja de una lista enlazada sobre un array es que la lista enlazada puede crecer y decrecer en tamaño, ajustándose al número de elementos.

Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único, a menos que sea el último, en cuyo caso no se enlaza con ningún otro nodo.

Cuando se inserta un elemento en una lista enlazada, se deben considerar cuatro casos: añadir a una lista vacía, añadir al principio de la lista, añadir en el interior y añadir al final de la lista.

Para borrar (eliminar) un elemento, primero hay que buscar el nodo que lo contiene y considerar dos casos: borrar el primer nodo y borrar cualquier otro de la lista.

El recorrido de una lista enlazada significa pasar por cada nodo (visitar) y procesarlo. El proceso puede ser: escribir su contenido o modificar el campo de datos.

Una **lista doblemente enlazada** es aquella en la que cada nodo tiene un puntero a su sucesor y otro a su predecesor. Las listas doblemente enlazadas se pueden recorrer en ambos sentidos. Las operaciones básicas son inserción, eliminación y recorrido de la lista; similares a las listas simples.

Una **lista enlazada circularmente** por propia naturaleza no tiene primero ni último nodo. Las listas circulares pueden ser de enlace simple o doble.

EJERCICIOS

9.1. Escribir una función que devuelva verdadero (≠0) si la lista está vacía.

9.2. Escribir una función entera que devuelva el número de nodos de una lista enlazada.

9.3. En una lista enlazada de números enteros se desea añadir un nodo entre dos nodos consecutivos con campos dato de distinto signo; el valor del campo dato del nuevo nodo que sea la diferencia en valor absoluto.

9.4. Escribir una función que elimine el nodo que ocupa la posición `i`, siendo el nodo cabecera el que ocupa la posición `0`.

9.5. Escribir una función que tenga como argumento el puntero `cabeza` al primer nodo de una lista enlazada. La función debe de devolver un puntero a una lista doble con los mismos campos dato pero en orden inverso.

9.6. Se dispone de una lista simplemente enlazada de números reales. Escribir una función para obtener una lista doble ordenada respecto al campo dato, con los valores reales de la lista simple.

9.7. Escribir una función para crear una lista doblemente enlazada de palabras introducidas por teclado. La función debe tener un argumento puntero `ld` en el que se devuelva la dirección del nodo que está en la posición intermedia.

9.8. Se dispone de `lc`, una lista circular de palabras. Escribir una función que cuente el número de veces que una palabra dada se encuentra en la lista.

9.9. Escribir una función entera que tenga como argumento una lista circular de números enteros. La función debe de devolver el dato del nodo con mayor valor.

9.10. Se tiene una lista de simple enlace, el campo dato es un registro (estructura) con los campos de un alumno: nombre, edad, sexo. Escribir una función para transformar la lista de tal forma que si el primer nodo es un alumno de sexo masculino, el siguiente sea de sexo femenino.

9.11. Una lista circular de cadenas está ordenada alfabéticamente. El puntero `lc` tiene la dirección del nodo alfabéticamente mayor, apunta al nodo alfabéticamente menor. Escribir una función para añadir una nueva palabra, en el orden que le corresponda, a la lista.

9.12. Dada la lista del Ejercicio 9.11 escribir una función que elimine una palabra dada.

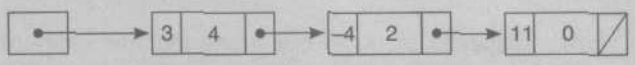
PROBLEMAS

9.1. Escribir un programa o funciones individuales que realicen las siguientes tareas:

- Crear una lista enlazada de números enteros positivos al azar, la inserción se realiza por el último nodo.
- Recorrer la lista para mostrar los elementos por pantalla.
- Eliminar todos los nodos que superen un valor dado.

9.2. Se tiene un archivo de texto de palabras separadas por un blanco o el carácter de fin de línea. Escribir un programa para formar una lista enlazada con las palabras del archivo. Una vez formada la lista se pueden añadir nuevas palabras o borrar alguna de ellas. Al finalizar el programa escribir las palabras de la lista en el archivo.

9.3. Un polinomio se puede representar como una lista enlazada. El primer nodo de la lista representa el primer término del polinomio, el segundo nodo al segundo término del polinomio y así sucesivamente. Cada nodo tiene como campo dato el coeficiente del término y el exponente. Por ejemplo, el polinomio $3x^4 - 4x^2 + 11$ se representa:



Escribir un programa que permita dar entrada a polinomios en x, representándolos con una lista enlazada simple. A continuación obtener una tabla de valores del polinomio para valores de x = 0.0, 0.5, 1.0, 1.5, ..., 5.0.

9.4. Teniendo en cuenta la representación de un polinomio propuesta en el Problema 9.3, hacer los cambios necesarios para que la lista enlazada sea circular. El puntero de acceso debe de tener la dirección del último término del polinomio, el cuál apuntará al primer término.

9.5. Según la representación de un polinomio propuesta en el Problema 9.4, escibir un programa para realizar las siguientes operaciones:

- Obtener la lista circular suma de dos polinomios.
- Obtener el polinomio derivado.
- Obtener una lista circular que sea el producto de dos polinomios.

9.6. Escribir un programa para obtener una lista doblemente enlazada con los caracteres de una cadena leída desde el teclado. Cada nodo de la lista tendrá un carácter. Una vez que se tiene la lista, ordenarla alfabéticamente y escribirla por pantalla.

9.7. Un conjunto es una secuencia de elementos todos del mismo tipo sin duplicidades. Escribir un programa para representar un conjunto de enteros mediante una lista enlazada. El programa debe contemplar las operaciones:

- Cardinal del conjunto.
- Pertenencia de un elemento al conjunto.
- Añadir un elemento al conjunto.
- Escribir en pantalla los elementos del conjunto.

9.8. Con la representación propuesta en el Problema 9.7, añadir las operaciones básicas de conjuntos:

- Unión de dos conjuntos.
- Intersección de dos conjuntos.
- Diferencia de dos conjuntos.
- Inclusión de un conjunto en otro.

9.9. Escribir un programa en el que dados dos archivos F1, F2 formados por palabras separadas por un blanco o fin de línea, se creen dos conjuntos con las palabras de F1 y F2 respectivamente. Posteriormente encontrar las palabras comunes y mostarlas por pantalla. Para resolver el problema, utilizar la representación y operaciones sobre conjuntos definidas en los Problemas 9.7 y 9.8.

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. 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.

9.12. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa que permita representar mediante listas enlazadas un vector disperso. Los

nodos de la lista son los elementos de la lista distintos de cero; en cada nodo se representa el valor del elemento y el índice (posición del vector). El programa ha de realizar las operaciones: sumar dos vectores de igual dimensión y hallar el producto escalar.

9.13. Escribir un programa que forme lista ordenada de registros de empleados. La ordenación ha de ser respecto al campo entero `Sueldo`. 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 por 100 el sueldo de los empleados que ganan menos de P euros.
- Aumentar en un 3 por 100 el sueldo de los empleados que ganan más de P euros.
- Dar de baja a los empleados con más de 35 años de antigüedad.

9.14. 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 número de línea en que aparece la palabra en el archivo. Escribir un programa que resuelva el problema propuesto estructurado de tal forma que en un *archivo de inclusión (.h)* se encapsule el tipo lista de números enteros con sus operaciones, y en otro *archivo de inclusión (.h)* el tipo lista de palabras con las operaciones necesarias que manejan la lista de palabras respectivamente.

10
PILAS Y SUS APLICACIONES

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Especificar el tipo abstracto de datos `Pila`.
- Conocer aplicaciones de las pilas en la programación.
- Definir e implementar el tipo `Pila`.
- Conocer las diferentes formas de escribir una expresión.
- Evaluar una expresión utilizando la estructura `Pila`.

CONTENIDO

10.1. Concepto de pila.
10.2. El tipo `Pila` implementado con arrays.
10.3. El tipo `Pila` implementado como una lista enlazada.
10.4. Evaluación de expresiones aritméticas mediante pilas.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Concepto de tipo abstracto de datos.
- Concepto de pila.
- Listas enlazadas.
- Expresión aritmética.
- Prioridad.
- Notación de una expresión.
- Operadores y operandos.

INTRODUCCIÓN

En este capítulo se estudia en detalle la estructura de datos `Pila` utilizada frecuentemente en los programas más usuales y en la vida diaria. La organización de una pila es muy conocida: 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 retirada de elementos se realiza por un mismo extremo denominado *cima* de la pila.

El desarrollo de las pilas como tipos abstractos de datos es el motivo central de este capítulo. En el mismo se verá cómo utilizar el *TAD* pila para la aplicación de evaluación de expresiones algebraicas; otra aplicación típica de las pilas es la organización de la memoria.

10.1. CONCEPTO DE PILA

Una **pila** (*stack*) es una colección ordenada de elementos a los que sólo se puede acceder por un único lugar o extremo de la pila. Los elementos de la pila se añaden o quitan (eliminan) de la misma sólo por su parte superior (**cima**) de la pila. Éste es el caso de una pila de platos, una pila de libros, la organización de la memoria libre, etc.

A recordar

Una pila es una estructura de datos de entradas ordenadas tales que sólo se pueden introducir y eliminar por un extremo, llamado **cima**.

Cuando se dice que *la pila está ordenada*, lo que se desea decir es que hay un elemento al que se puede acceder primero (el que está encima de la pila), otro elemento al que se puede acceder en segundo lugar (justo el elemento que está debajo de la cima), un tercero, etc. No se requiere que las entradas se puedan comparar utilizando el operador «menor que» (<) y pueden ser de cualquier tipo.

Las entradas de la pila deben ser eliminadas en el orden inverso al que se situaron en la misma. Por ejemplo, se puede crear una pila de libros, situando primero un diccionario, encima de él una enciclopedia y encima de ambos una novela de modo que la pila tendrá la novela en la parte superior.

Figura 10.1. Pila de libros

Cuando se quitan los libros de la pila, primero debe quitarse la novela, luego la enciclopedia y por último el diccionario.

Debido a su propiedad específica «último en entrar, primero en salir» se conoce a las pilas como estructura de datos **LIFO** (*last-in, first-out*).

Las operaciones usuales en la pila son *Insertar* y *Quitar*. La operación **Insertar** o **Poner** (`push`) añade un elemento en la cima de la pila y la operación **Quitar** o **Retirar** (`pop`) elimina o saca un elemento de la pila. La Figura 10.2 muestra una secuencia de operaciones *Insertar* y *Quitar*. El último elemento añadido a la pila es el primero que se quita de la pila.

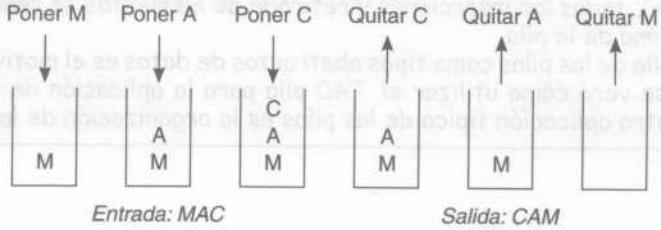


Figura 10.2. Poner y quitar elementos de la pila.

La operación `insertar` (*push*) sitúa un elemento dato en la cima de la pila y `Quitar` (*pop*) elimina o quita el elemento de la pila.

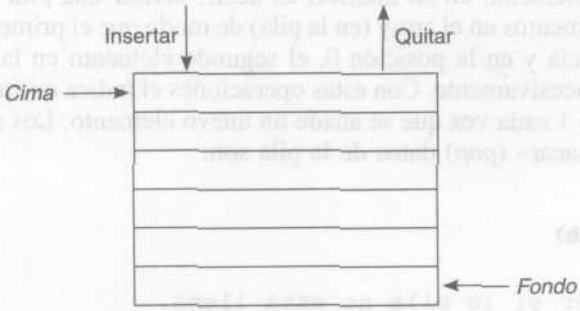


Figura 10.3. Operaciones básicas de una pila.

La pila se puede implementar mediante arrays, en cuyo caso su dimensión o longitud es fija, y mediante listas enlazadas, en cuyo caso se utiliza memoria dinámica y no existe limitación en su tamaño excepto la memoria libre del ordenador.

Una pila puede estar *vacía* (no tiene elementos) o *llena* (en el caso de tener tamaño fijo, si no caben más elementos en la pila). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error, debido a que esa operación es imposible; esta situación se denomina **desbordamiento negativo** (*underflow*). Por el contrario, si un programa intenta poner un elemento en una pila se produce un error llamado **desbordamiento** (*overflow*) o **rebosamiento.** Pata evitar estas situaciones se diseñan funciones, que comprueban si la pila está llena o vacía.

10.1.1. Especificaciones del tipo abstracto de datos Pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes (no todas ellas se implementan al definir una pila):

Tipo de dato:	Dato que se almacena en la pila
Operaciones:	
CrearPila	Inicia la pila como vacía
Insertar (push)	Pone un dato en la pila
Quitar (pop)	Retira (saca) un dato de la pila
Pila vacía	Comprobar si la pila no tiene elementos
Pila llena	Comprobar si la pila está llena de elementos
Limpiar pila	Quitar todos sus elementos y dejar la pila vacía
Cima	Obtiene el elemento cima de la pila
Tamaño de la pila	Número de elementos máximo que puede contener la pila

10.2. EL TIPO `PILA` IMPLEMENTADO CON ARRAYS

Una pila se puede implementar mediante arrays o mediante listas enlazadas. Una implementación estática se realiza utilizando un array de tamaño fijo y una implementación dinámica mediante una lista enlazada.

El tipo `Pila` implementado con arrays incluye una lista (array) y un índice a la cima de la pila; además, una constante con el máximo número de elementos. Al utilizar un array para contener los

elementos de la pila hay que tener en cuenta que el tamaño de la pila no puede exceder el número de elementos del array, y la condición *pila llena* será significativa para el diseño.

El método usual de introducir elementos en una pila es definir el *fondo* de la pila en la posición –1 y sin ningún elemento en su interior; es decir, definir una *pila vacía*; a continuación, se van introduciendo elementos en el array (en la pila) de modo que el primer elemento añadido se introduce en una pila vacía y en la posición 0, el segundo elemento en la posición 1, el siguiente en la posición 2 y así sucesivamente. Con estas operaciones el índice que apunta a la cima de la pila se va incrementando en 1 cada vez que se añade un nuevo elemento. Los algoritmos de poner «insertar» (*push*) y quitar «sacar» (*pop*) datos de la pila son:

Insertar (*push*)

```
1. Verificar si la pila no está llena.
2. Incrementar en 1 el puntero índice de la pila.
3. Almacenar elemento en la posición del puntero de la pila.
```

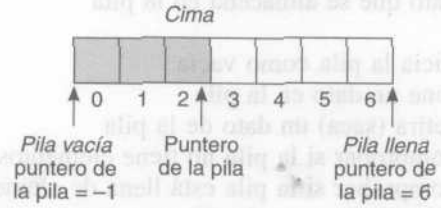
Quitar (*pop*)

```
1. Si la pila no está vacía.
2. Leer el elemento de la posición del puntero de la pila.
3. Decrementar en 1 el puntero de la pila.
```

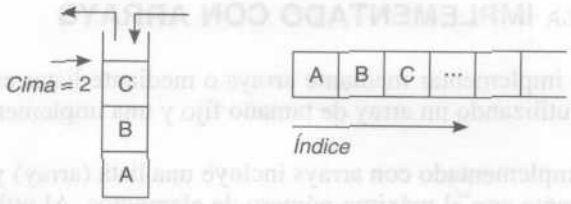
En el caso de que el array que define la pila tenga `TAMPILA` elementos, las posiciones del array, es decir, el índice o puntero de la pila, estarán comprendidas en el rango `0` a `TAMPILA-1` elementos, de modo que *en una pila llena* el puntero de la pila apunta a `TAMPILA-1` y *en una pila vacía* el puntero de la pila apunta a –1, ya que 0 será el índice del primer elemento.

Ejemplo 10.1

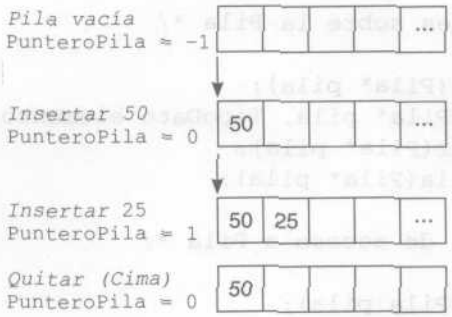
Una pila de 7 elementos se puede representar gráficamente así:



Si se almacenan los datos `A, B, C, . . .` en la pila se puede representar gráficamente de alguna de estas formas:



Veamos ahora cómo queda la pila en función de diferentes situaciones de un posible programa.



10.2.1. Especificación del tipo abstracto `Pila`

La declaración de un tipo abstracto incluye la representación de los datos y la definición de las operaciones. En el *TAD Pila* los datos pueden ser de cualquier tipo, y las operaciones, las ya citadas anteriormente en el Apartado 10.1.1.

1. Datos de la pila (tipo `TipoDato`, que es conveniente definirlo mediante `typedef`).
2. `crearPila` inicializa una pila. Es como crear una pila sin elementos, por tanto, vacía.
3. Verificar que la pila no está llena antes de `insertar` o poner (*push*) un elemento en la pila; verificar que una pila no está vacía antes de `quitar` o sacar (*pop*) un elemento de la pila. Si estas precondiciones no se cumplen se debe visualizar un mensaje de error y el programa debe terminar.
4. `pilaVacia` devuelve 1 (verdadero) si la pila está vacía y 0 (falso) en caso contrario.
5. `pilaLlena` devuelve 1 (verdadero) si la pila está llena y 0 (falso) en caso contrario. Estas funciones se utilizan para verificar las precondiciones de *insertar* y *quitar*.
6. `limpiarPila`. Se limpia o vacía la pila, dejándola sin elementos y disponible para otras tareas.
7. `cima`, devuelve el valor situado en la cima de la pila, pero no se decrementa el puntero de la pila, ya que la pila queda intacta.

Declaración

Para que la declaración sea lo más abstracta posible, en un archivo *.h* se declara el tipo de los datos que contiene la pila y los prototipos de las funciones que representan las operaciones del *tipo abstracto* `Pila`:

```
/* archivo pilaarray.h */

#define TAMPILA 100

typedef struct
{
 TipoDato listaPila[TAMPILA];
```

```
  int cima;
}Pila;

  /* Operaciones sobre la Pila */

void crearPila(Pila* pila);
void insertar(Pila* pila, TipoDato elemento);
TipoDato quitar(Pila* pila);
void limpiarPila(Pila* pila);

  /* Operación de acceso a Pila */

TipoDato cima(Pila pila);

  /* verificación estado de la Pila */

int pilaVacia(Pila pila);
int pilaLlena(Pila pila);
```

Antes de incluir el archivo `pilaarray.h` debe de declararse el `TipoDato`. Así, si se quiere una pila de números enteros:

```
typedef int TipoDato;
#include "pilaarray.h"
```

En el caso de que la pila fuera de números complejos:

```
typedef struct
{
  float x,y;
}TipoDato;
#include "pilaarray.h"
```

Ejemplo 10.2

Escribir un programa que cree una pila de enteros y realice operaciones de añadir datos a la pila, quitar ... ..

En el archivo `pilaarray.h` se encuentra lo esencial del tipo abstracto de datos `Pila`, que es la representación de los datos y las operaciones en forma de los prototipos de la funciones. El ejemplo pretende hacer uso de este tipo de datos sin tener en cuenta la implementación de las funciones. Entonces, se supone que están ya codificadas las funciones y lo único que hay que hacer es invocar a estas funciones u operaciones del *TAD Pila*. El programa crea una pila de números enteros, inserta en la pila elementos introducidos desde el teclado y a continuación retira y visualiza los elementos hasta que se cumpla la condición de *pila vacía*.

```
typedef int TipoDato;
#include "pilaarray.h"
#include <stdio.h>

void main()
{
```

```
  Pila pila;
  int x;
  crearPila(&pila);          /* Crea una pila vacía */
  puts("\nLa pila crece con números enteros, terminar con 0");
  do {
   scanf("%d",&x);
   if (x != 0)
    if (!pilaLlena(pila))
      insertar(&pila, x);
    else
    {
      puts("\n La pila se ha llenado. \n");
      x = 0;
    }
  } while (x != 0);

  puts("Visualiza los elementos de la pila hasta que quede vacía");

  while (!pilaVacia(pila))
  {
    printf("\n \t %d ", cima(pila));
    quitar(&pila);
  }
}
```

10.2.2. Implementación de las operaciones sobre pilas

Las operaciones del *TAD Pila* definidas en la especificación se implementan en el archivo `pilaarray.c` para después formar un proyecto con otros módulos y la función principal. La codificación de las funciones exige incluir el archivo donde se encuentra la representación de los elementos y los prototipos.

Archivo `pilaarray.c`

```
typedef int TipoDato;            /* tipo de los elementos de la pila */
#include "pilaarray.h"
#include <stdlib.h>
void crearPila(Pila* pila)
{
  pila -> cima = -1;
}
```

La operación `insertar` un elemento en la pila incrementa el puntero de la pila (cima) en 1 y asigna el nuevo elemento a la lista de la pila. Cualquier intento de añadir un elemento en una pila llena produce un mensaje de error `Desbordamiento pila` y debe terminar el programa.

```
/* poner un elemento en la pila */

void insertar(Pila* pila, TipoDato elemento)
{
```

```
    /* si la pila está llena, termina el programa */
    if (pilaLlena(*pila))
    {
      puts("Desbordamiento pila");
      exit (1);
    }

    pila -> cima++;
    pila -> listaPila[pila->cima] = elemento;
}
```

La operación `quitar` elimina un elemento de la pila copiando primero el valor de la cima de la pila en una variable local auxiliar, `tem`, y a continuación decrementa el puntero de la pila en 1. El objetivo de la operación es retirar el elemento `cima`; no obstante, la función devuelve la variable `tem` para que pueda ser utilizada. Si se intenta eliminar o borrar un elemento en una pila vacía se debe producir un mensaje de error y terminar.

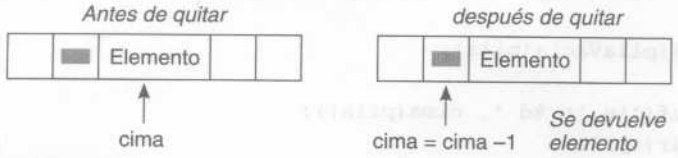


Figura 10.4. Operación de extraer un elemento de la pila.

```
/* Quitar un elemento de la pila */

TipoDato quitar(Pila* pila)
{
  TipoDato tem;
    /* si la pila está vacía, termina el programa */
  if (pilaVacia(*pila))
  {
    puts(" Se intenta sacar un elemento en pila vacía");
    exit (1);
  }

  tem = pila -> listaPila[pila->cima];

  pila -> cima--;
  return tem;
}
```

La operación `cima` devuelve el elemento que se encuentra en la cima de la pila: no se modifica la pila.

```
TipoDato cima(Pila pila)
{
  if (pilaVacia(pila))
  {
    puts(" Error de ejecución, pila vacía");
    exit (1);
  }
```

```
    return pila.listaPila[pila.cima];
}
```

10.2.3. Operaciones de verificación del estado de la pila

Se debe proteger la integridad de la pila, para lo cual el tipo `Pila` ha de proporcionar operaciones que comprueben el estado de la pila: *pila vacía* o *pila llena*. Asimismo, se ha de definir una operación que restaure la condición inicial de la pila, que se establece por el inicializador `crearPila` (cima de la pila `a-1`); de esta tarea se encarga `limpiarPila()`.

La función `pilaVacia` comprueba (verifica) si la cima de la pila es –1. En ese caso, la pila está vacía y se devuelve un 1 (verdadero); en caso contrario, se devuelve 0 (falso).

(Continúa el archivo `pilaarray.c`)

```
int pilaVacia(Pila pila)
{
  return pila.cima == -1;
}
```

La función `pilaLlena` comprueba (verifica) si la cima es `TAMPILA-1`. En ese caso, la pila está *llena y se devuelve un 1 (verdadero); en caso contrario, se devuelve 0 (falso).*

```
int pilaLlena(Pila pila)
{
  return pila.cima == TAMPILA - 1;
}
```

Por último, la operación `limpiarPila` vacía íntegramente la pila, para lo que pone cima al valor que se corresponde con pila vacía (–1).

```
void limpiarPila(Pila* pila)
{
  pila -> cima = -1;
}
```

Ejercicio 10.1

Escribir un programa que utilice el *TAD Pila* para comprobar si una determinada frase/palabra (cadena de caracteres) es un palíndromo. *Nota:* una palabra o frase es un palíndromo cuando la lectura directa e indirecta de la misma tiene igual valor: **alila** es un palíndromo; **cara (arac)** no es un palíndromo.

La palabra o frase a investigar se lee carácter a carácter, de tal forma que a la vez que se añade a una cadena (*string*) se inserta en una pila de caracteres. Una vez leída la palabra, se compara el primer carácter del *string* con el carácter que se extrae de la pila: si son iguales sigue la comparación con siguiente carácter del *string* y de la pila; así hasta que la pila se queda vacía o hay un carácter no coincidente.

Al guardar los caracteres de la palabra en la pila se garantiza que las comparaciones se realizan entre caracteres que están en orden inverso: primero con último...

La codificación consta de tres archivos, el archivo `pilarray.h` con las declaraciones de la pila; el archivo `pilarray.c` con la implementación de las operaciones de la pila. El tercer archivo, `paldromo.c`, tiene la función principal en la que se lee la palabra y se comprueba, con ayuda de la pila, si es palíndromo. Los dos primeros archivos ya están escritos, ahora sólo se escribe el archivo `paldromo.c`. Para formar un ejecutable se crea un proyecto con `pilarray.c` y `paldromo.c`; ambos deben de incluir el archivo con las declaraciones de tipos y prototipos: `pilarray.h`.

```
/*          Archivo paldromo.c          */
typedef char TipoDato;
#include <stdlib.h>
#include "pilarray.h"

#include <ctype.h>

void main()
{
  char palabra[100], ch;
  Pila pila;
  int j, esPaldro;

  crearPila(&pila);

     /* se lee la palabra o frase */

  do {
    puts("\n Palabra a comprobar si es palíndromo");
    for (j = 0; (ch = getchar()) != '\n'; )
    {
      palabra[j++] = ch;
      insertar(&pila, ch);          /* pone ch en la pila */
    }
    palabra[j] = '\0';

   /* comprueba si es palíndromo */

   esPaldro = 1;
   for (j = 0; esPaldro && !pilaVacia(pila); )
   {
     esPaldro = palabra[j++] == quitar(&pila);
   }
   limpiarPila(&pila);
   if (esPaldro)
     printf("\n La palabra %s es un palíndromo \n",palabra);
   else
     printf("\n La palabra %s no es un palíndromo \n",palabra);

   printf("\n ¿ Otra palabra ?: "); scanf("%c%*c", &ch);
 }while (tolower(ch) == 's');

}
```

10.3. EL TIPO PILA IMPLEMENTADO COMO UNA LISTA ENLAZADA

La realización dinámica de una pila se hace almacenando los elementos como nodos de una lista enlazada. Con la particularidad de que siempre que se desee insertar o poner (empujar) un elemento se hace por el mismo extremo que se extrae.

Esta realización tiene la ventaja de que el tamaño se ajusta exactamente a los elementos de la pila, no hay un máximo de elementos y, por tanto, no existe la condición de *pila llena*. Sin embargo, para cada elemento es necesaria mas memoria ya que hay que guardar el campo de enlace. En la realización de una pila con arrays se requiere establecer un máximo de posibles elementos, como contrapartida el acceso es más rápido ya que se hace con una variable subindicada.

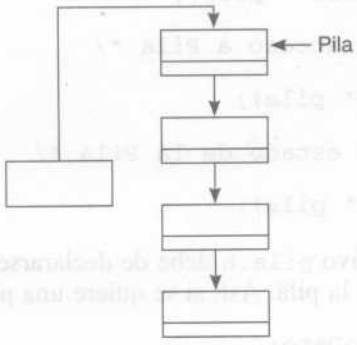


Figura 10.5. Representación de una pila con una lista enlazada.

A tener en cuenta

Una pila realizada con una lista enlazada crece y decrece dinámicamente. En tiempo de ejecución se reserva memoria según se ponen elementos en la pila y se libera memoria según se extraen elementos de la pila.

10.3.1. Tipos de datos para una pila con listas enlazadas

Los tipos de datos en la realización de una pila con listas son muy similares a los ya expuestos en listas enlazadas. Los elementos de la pila van a ser los nodos de la lista, con un campo para guardar el elemento y otro de enlace; la base para todo ello es la utilización de variables puntero (apuntador). Las operaciones del tipo pila implementadas con listas son las mismas que si la pila se implementa con arrays, salvo la operación que controla si la pila está llena, `pilaLlena`, que con listas enlazadas no se produce al poder crecer indefinidamente, con el único límite de la memoria. También se añade la operación que elimina el elemento cabeza, *suprimir*, que se diferencia de *quitar* en que no devuelve el elemento extraído.

En el archivo `pila.h` se declaran los tipos de datos y los prototipos de las funciones que representan las operaciones. Este archivo es necesario incluirlo en los archivos que hacen uso del *TAD pila*.

```
/* archivo pila.h */

#include <stdlib.h>

typedef struct nodo
{
```

```
    TipoDato elemento;
    struct nodo* siguiente;
  } Nodo;
    /* la declaración de una pila: Nodo* pila */

    /* prototipo de las operaciones */
  void crearPila(Nodo** pila);
  void insertar(Nodo** pila, TipoDato elemento);
  void suprimir(Nodo** pila);
  TipoDato quitar(Nodo** pila);
  void limpiarPila(Nodo** pila);

    /* Operación de acceso a Pila */

  TipoDato cima(Nodo* pila);

    /* verificación estado de la Pila */

  int pilaVacia(Nodo* pila);
```

Antes de incluir el archivo `pila.h` debe de declararse el `TipoDato` según el tipo de los elementos que va a almacenar la pila. Así, si se quiere una pila de números reales:

```
typedef double TipoDato;
#include "pila.h"
```

En el caso de que la pila fuera de elementos mas complejos, por ejemplo, el par *<palabra, número de línea>*:

```
typedef struct
{
  char palabra[81];
  int numLinea;
}TipoDato;
#include "pila.h"
```

10.3.2. Implementación de las operaciones de pilas con listas enlazadas

Las operaciones del *TAD pila* cuya declaración se encuentra en el archivo `pila.h`, se implementan en el archivo `pila.c` para formar después un proyecto con otros módulos y la función principal. La codificación de las funciones exige incluir el archivo `pila.h` y previamente declarar el tipo concreto de los elementos de la pila.

La codificación que a continuación se escribe es para una pila de números reales; al estar la pila representada por una lista enlazada, esta codificación es muy similar a la de funciones de listas enlazadas.

Archivo `pila.c`

```
typedef double TipoDato;          /* tipo de los elementos de la pila */
#include "pila.h"
```

Creación de una pila sin elementos, vacía

```
void crearPila(Nodo** pila)
{
```

```
  *pila = NULL;
}
```

Verificación del estado de la pila

```
int pilaVacia(Nodo* pila)
{
  return pila == NULL;
}
```

Meter un elemento en la pila

Se crea un nodo con el elemento que se pone en la pila y se enlaza por la cima de la pila.

```
void insertar(Nodo** pila, TipoDato elemento)
{
  Nodo* nuevo;

  nuevo = (Nodo*)malloc(sizeof(Nodo));          /* crea un nuevo nodo */
  nuevo -> elemento = elemento;
  nuevo -> siguiente = *pila;
  (*pila) = nuevo;
}
```

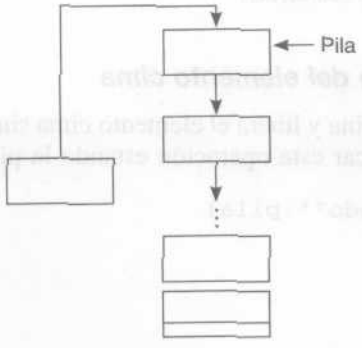


Figura 10.6. Operación de añadir un elemento a la pila.

Eliminación del elemento cima

Con esta operación se elimina y libera el elemento cabeza, modificando la pila.

```
void suprimir(Nodo** pila)
{
  if (!pilaVacia(*pila))
  {
    Nodo* f;
    f = *pila;
    (*pila) = (*pila) -> siguiente;
      free(f);
  }
}
```

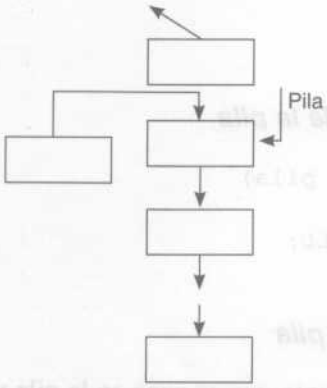


Figura 10.7. Operación de suprimir la cima de la pila.

Obtención del elemento cabeza o cima de la pila, sin modificar la pila

```
TipoDato cima(Nodo* pila)
 {
   if (pilaVacia(pila))
   {
      puts(" Error de ejecución, pila vacía");
      exit (1);
   }

   return pila -> elemento;
 }
```

Extracción y obtención del elemento cima

Esta operación no sólo elimina y libera el elemento cima sino que además devuelve dicho elemento. Se considera un error invocar esta operación estando la pila vacía.

```
TipoDato quitar(Nodo** pila)
{
  TipoDato tem;
  Nodo* q;

  if (pilaVacia(*pila))
  {
    puts(" Se intenta sacar un elemento en pila vacía");
    exit (1);
  }

  tem = (*pila) -> elemento;
  q = *pila;
  (*pila) = (*pila) -> siguiente;
  free(q);

  return tem;
}
```

La operación de *quitar* es, en realidad, la unión de dos operaciones: *cima* y *suprimir*, y en consecuencia se puede codificar con sendas llamadas a ellas:

```
TipoDato quitar(Nodo** pila)
```

```
{
   TipoDato tem = cima(*pila);
   suprimir(pila);
   return tem;
}
```

Vaciado de la pila

Esta operación libera todos los nodos de que consta la pila. Se basa en la operación suprimir.

```
void limpiarPila(Nodo** pila)
{
  while(!pilaVacia(*pila))
  {
    suprimir(pila);
  }
}
```

10.4. EVALUACIÓN DE EXPRESIONES ARITMÉTICAS MEDIANTE PILAS

Una *expresión aritmética* está formada por operandos y operadores. Así, la expresión `x*y - (a+b)` consta de los operadores `*, -, +` y de los operandos `x, y, a, b`. Los operandos pueden ser valores constantes, variables o, incluso, otra expresión. Los operadores son los símbolos conocidos de las operaciones matemáticas. La evaluación de una expresión aritmética da lugar a un valor numérico, se realiza sustituyendo los operandos que son variables por valores concretos y ejecutando las operaciones aritméticas representadas por los operadores. Así, si los operandos de la expresión anterior toman los valores: `x = 5, y = 2, a = 3, b = 4`, el resultado de la evaluación es:

```
5*2 - (3+4) = 5*2 - 7 = 10 - 7 = 3
```

La forma habitual de escribir expresiones matemáticas es aquella en la que el operador está entre sus dos operandos. La expresión anterior escrita de esa forma, recibe el nombre de *notación infija*. Esta forma de escribir las expresiones exige, en algunas ocasiones, el uso de paréntesis para *encerrar* subexpresiones con mayor prioridad.

Conviene recordar que, en las operaciones, los operadores tienen distintos niveles de precedencia o prioridad a la hora de su evaluación. A continuación se recuerda estos niveles de prioridad, en C, en orden de mayor a menor:

```
Paréntesis                : ( )
Potencia                  : ^
Multiplicación/división:   *, /
Suma/Resta                : +,-
```

También hay que recordar que, a igualdad de precedencia (*asociatividad*), se evalúan de izquierda a derecha, excepto la potencia, que es de derecha a izquierda.

10.4.1. Notación prefija y notación postfija de una expresión aritmética

La forma habitual de escribir operaciones aritméticas sitúa el operador entre sus dos operandos, conocida como *notación infija*. Esta forma de notación obliga en muchas ocasiones a utilizar paréntesis para indicar el orden de evaluación. Las expresiones

```
r = a*b/(a+c)        g = a*b/a+c
```

son distintas al no poner paréntesis en la expresión `g`. Igual ocurre con estas otras:

```
r = (a-b)^c+d        g = a-b^c+d
```

Existen otras formas de escribir expresiones aritméticas, que se diferencian por la ubicación del operador respecto de los operandos. La notación en la que el operador se coloca delante de los dos operandos, *notación prefija*, se conoce como notación polaca (en honor del matemático polaco que la propuso). En el Ejemplo 10.3 se muestra expresiones en notación *prefija* o notación *polaca*.

Ejemplo 10.3

Dadas las expresiones: `a*b/(a+c); a*b/a+c; (a-b)^c+d`. Escribir las expresiones equivalentes en notación prefija.

A continuación se escribe, paso a paso, la transformación de cada expresión algebraica en la expresión equivalente en notación polaca.

```
a*b/(a+c)  (infija) → a*b/+ac → *ab/+ac → /*ab+ac (polaca)
a*b/a+c    (infija) → *ab/a+c → /*aba+c → +/*abac (polaca)
(a-b)^c+d  (infija) → -ab^c+d → ^-abc+d → +^-abcd (polaca)
```

Se puede observar en el Ejemplo 10.3 que no es necesaria la utilización de paréntesis al escribir la expresión en notación polaca. La propiedad fundamental de la notación polaca, el orden en que se van a realizar las operaciones y está determinado por las posiciones de los operadores y los operandos en la expresión.

Notación postfija

Existen otras formas de escribir las operaciones. Así, la notación *postfija* o *polaca inversa* coloca el operador a continuación de sus dos operandos. En el Ejemplo 10.4, las expresiones del Ejemplo 10.3 se escriben en notación polaca inversa.

Ejemplo 10.4

Dadas las expresiones: `a*b/(a+c); a*b/a+c; (a-b)^c+d`. Escribir las expresiones equivalentes en notación postfija.

Paso a paso se transforma cada subexpresión en notación polaca inversa.

```
a*b/(a+c)  (infija) → a*b/ac+ → ab*/ac+ → ab*ac+/ (polaca inversa)
a*b/a+c    (infija) → ab*/a+c → ab*a/+c → ab*a/c+ (polaca inversa)
(a-b)^c+d  (infija) → ab-^c+d → ab-c^+d → ab-c^d+ (polaca inversa)
```

A tener en cuenta

Las diferentes formas de escribir una misma expresión algebraica depende de la ubicación de los operadores respecto a los operandos. Es importante tener en cuenta que tanto en la notación prefija como en la postfija no son necesarios los paréntesis para cambiar el orden de evaluación.

10.4.2. Evaluación de una expresión aritmética

A la hora de evaluar una expresión aritmética escrita a la *manera habitual*, en notación infija, se siguen dos pasos principales:

1. Transformar la expresión de notación infija a postfija.
2. Evaluar la expresión en notación postfija.

En los algoritmos que se aplican en cada uno de los pasos es básico el uso del *TAD pila*. El tipo pila asegura que el *último en entrar es el primero en salir*, y de esa forma el algoritmo de transformación a expresión *postfija* sitúa los operadores después de los operandos según la prioridad (precedencia) correspondiente a cada operador. Una vez se tiene la expresión en notación *postfija*, se utiliza una segunda pila, de números reales, para obtener el valor numérico de la expresión algebraica.

10.4.2.1. *Transformación de expresión infija a postfija*

Se parte de una expresión en notación infija que tiene operandos, operadores y puede tener paréntesis. Los operandos vienen representados por letras y los operadores son:

`^ (potenciación), *, /, +, -`

La transformación se realiza utilizando una pila en la cual se almacenan los operadores y los paréntesis izquierdos. La expresión aritmética se va leyendo desde el teclado carácter a carácter, los operandos pasan directamente a formar parte de la expresión en *postfija* la cual se guarda en un array.

Los operadores se meten en la pila siempre que ésta esté vacía, o bien siempre que tengan mayor prioridad que el operador cima de la pila (o bien igual si es la máxima prioridad). Si la prioridad es menor o igual se saca el elemento cima de la pila y se vuelve a hacer la comparación con el nuevo elemento cima.

Los paréntesis izquierdos siempre se meten en la pila; dentro de la pila se les considera de mínima prioridad para que todo operador que se encuentra dentro del paréntesis entre en la pila. Cuando se lee un paréntesis derecho, hay que sacar todos los operadores de la pila pasando a formar parte de la expresión *postfija*, hasta llegar a un paréntesis izquierdo, el cual se elimina ya que los paréntesis no forman parte de la expresión *postfija*.

El algoritmo termina cuando no hay más items de la expresión y la pila esté vacía. Sea, por ejemplo, la expresión `a*(b+c-(d/e^f)-g)-h`, la expresión en postfija se va a ir formando así:

Expresión en postfija

a	Operando `a` pasa a la expresión *postfija*; operador `*` a la pila.
ab	Operador `(` pasa a la pila; operando `b` a la expresión.
abc	Operador `+` pasa a la pila; operando `c` a la expresión.
	En este momento el estado de la pila:

el nuevo carácter de la expresión, `-`, tiene igual prioridad que el operador de la cima (`+`), da lugar:

`abc+`

`abc+d` operador `(` pasa a la pila; operando `d` a la expresión.

`abc+de` operador `/` pasa a la pila; operando `e` a la expresión.

`abc+def` operador `^` pasa a la pila; operando `f` a la expresión.

El siguiente ítem, `)` (paréntesis derecho), provoca vaciar la pila hasta un `(`.

La pila, en este momento, se encuentra con estos operadores:

`abc+def^/` da lugar a la pila

`abc+def^/-` operador `-` pasa a la pila y, a su vez, se extrae `-`; el siguiente carácter, el operando `g` pasa a la expresión.

`abc+def^/-g` El siguiente carácter, `)`, por lo que son extraídos de la pila los operadores hasta un `(`, la pila queda

`abc+def^/-g-` el siguiente es el operador `-`, hace que se saque de la pila el operador `*` y se meta en la pila el `-`

`abc+def^/-g-*` el operando `h` se pasa directamente a la expresión

`abc+def^/-g-*h` Fin de entrada, se vacía la pila pasando los operadores a la expresión:

`abc+def^/-g-*h-`

En la descripción realizada se observa que el paréntesis izquierdo tiene la máxima prioridad fuera de la pila, es decir, en la *notación infija*; sin embargo, cuando está dentro de la pila la prioridad es mínima. De igual forma, para tratar que el hecho de que varios operadores de potenciación son evaluados de derecha a izquierda, este operador tendrá mayor prioridad cuando todavía no esté metido en la pila que cuando esté puesto en la pila. Las prioridades son determinadas según la Tabla 10.1.

Tabla 10.1. Tabla de prioridades de los operadores considerados

Operador	Prioridad dentro pila	Prioridad fuera pila
^	3	4
*, /	2	2
+, -	1	1
(	0	5

Obsérvese que no se trata el paréntesis derecho ya que este provoca sacar operadores de la pila hasta el paréntesis izquierdo.

Algoritmo de paso de notación infija a postfija

Los pasos a seguir para transformar una expresión algebraica de notación infija a postfija son:

1. Obtener caracteres de la expresión y repetir los pasos 2 al 4 para cada carácter.
2. Si es un operando, pasarlo a la expresión postfija.
3. Si es operador:
   3.1. Si pila está vacía, meterlo en la pila. Repetir a partir de 1.
   3.2. Si la pila no está vacía:
   - Si la prioridad del operador leído es mayor que la prioridad del operador cima de la pila, meterlo en la pila y repetir a partir de 1.
   - Si la prioridad del operador es menor o igual que la prioridad del operador de la cima pila, sacar operador cima de la pila y pasarlo a la expresión postfija, volver a 3.
4. Si es paréntesis derecho:
   4.1. Sacar operador cima de pila y pasarlo a la expresión postfija.
   4.2. Si nueva cima es paréntesis izquierdo, suprimir elemento cima.
   4.3. Si cima no es paréntesis izquierdo, volver a 4.1.
   4.4. Volver a partir de 1.
5. Si quedan elementos en la pila pasarlos a la expresión postfija.
6. Fin del algoritmo.

Para codificar este algoritmo es necesario el *TAD pila* de caracteres. Se utiliza el *archivo de inclusión* `pila.h` y la implementación de las operaciones en `pila.c` (Apartado 10.3.2).

```
typedef char TipoDato;
#include "pila.h"
```

Codificación de las funciones para transformar a postfija

Los tipos de datos para hacer esta transformación, además del tipo pila, son principalmente un registro o estructura para representar un elemento de la expresión, con un campo carácter en el que se guarda el operando, o bien el operador; otro campo para indicar si es operador u operando.

```
struct Elemento
{
   char c;
   int oprdor;
}
```

y un array de registros para contener la expresión en *postfija*.

En cuanto a las funciones, la prioridad de un operador se determina en dos funciones: una para representar la prioridad del operador dentro de la pila, `prdadDentro()`, y la otra para cuando está fuera de la pila, `prdadFuera()`. La función `postFija()` es en la que se implementa los pasos del algoritmo de transformación. Además, se escriben dos funciones de control, una valida el carácter leído del teclado, `valido()`, y otra determina si un elemento es un operando.

El archivo `postfija.h` encapsula los nuevos tipos de datos junto a los prototipos de las funciones, y en `postfija.c` se codifican las funciones.

Archivo `postfija.h`

```
typedef char TipoDato;
#include "pila.h"
```

```
typedef struct
{
  char c;
  int oprdor;
}Elemento ;

#define N 120

void postFija(Elemento expsion[N], int* n);
int prdadDentro(TipoDato op);
int prdadFuera(TipoDato op);
int valido(TipoDato c);
int operando(TipoDato c);
```

Archivo postfija.c

```
#include "postfija.h"
#include <ctype.h>

/*
  En esta función se lee la expresión en notación infija. La
  expresión se lee hasta el fin de línea, carácter \n. La
  expresión en postfija es almacenada en el vector expsion.
  Además, n es el número de elementos de la expresión.
*/

void postFija(Elemento expsion[N], int* n)
{
  Nodo* pila;
  char ch, it;
  int desapila, j;

  ch = ' '; j = -1;
  crearPila(&pila);

  while (ch != '\n')
  {
      do {
        scanf("%c", &ch);
        ch = toupper(ch);
      }while(!valido(ch));

      if (operando(ch))      /* pasa directamente a la expresión */
      {
        expsion[++j].oprdor = 0;
        expsion[j].c = ch;
      }
      else if (ch != '\n')                    /* es un operador */
      {
        if (ch != ')')
        {
          desapila = 1;
```

```
        while (desapila)
        {
          if (pilaVacia(pila) ||
               (prdadFuera(ch) > prdadDentro(cima(pila))))
          {
            insertar(&pila, ch);
            desapila = 0;
          }
          else if (prdadFuera(ch) < prdadDentro(cima(pila)))
          {
            it = quitar(&pila);
            expsion[++j].c = it;
            expsion[j].oprdor = 1;
          }
        }
      }
      else                                  /* es un ')' */
      {
          it = quitar(&pila);
          do{
            expsion[++j].c = it;
            expsion[j].oprdor = 1;
            it = quitar(&pila);
          }while (it != '(');
      }
    }
  }
    /*
      una vez que se termina la entrada se vuelcan los operadores
      que quedan en la pila y se pasan a la expresión.
    */
  while (!pilaVacia(pila))
  {
    it = quitar(&pila);
    expsion[++j].c = it;
    expsion[j].oprdor = 1;
  }
  *n = j+1;              /* número de elementos de la expresión */
}

int prdadDentro(TipoDato op)
{
  int pdp;
  switch (op)
  {
    case '^' : pdp = 3;
               break;
    case '*' : case '/':
               pdp = 2;
               break;
    case '+' : case '-':
               pdp = 1;
```

```
                break;
    case '(' : pdp = 0;
  }
  return pdp;
}

                    /* prioridad del operador en la expresión infija */

int prdadFuera(TipoDato op)
{
  int pfp;
  switch (op)
  {
    case '^' : pfp = 4;
               break;
    case '*' : case '/':
               pfp = 2;
               break;
    case '+' : case '-':
               pfp = 1;
               break;
    case '(' : pfp = 5;
  }
  return pfp;
}
int valido(TipoDato c)
{
  return (c >= 'A' && c <= 'Z') ||
         (c >= 'a' && c <= 'z') ||
         (c=='^' || c=='/' || c=='*' ||
          c=='+' || c=='-' || c=='\n');
}

int operando(TipoDato c)
{
  return (c >= 'A' && c <= 'Z');
}
```

10.4.2.2. *Evaluación de la expresión en notación postfija*

En un vector ha sido almacenada la expresión aritmética transformada a notación postfija, en la que los operandos están representados por variables de una sola letra. Antes de evaluar la expresión se requiere dar valores numéricos a los operandos.

Una vez se tienen los valores de los operandos, la expresión es evaluada. El algoritmo de evaluación utiliza una pila de operandos, en definitiva de números reales. Al describir el algoritmo, `expsion` es el vector que contiene la expresión. El número de elementos de que consta la expresión es `n`.

1. Examinar `expsion` desde el elemento `1` hasta el `n`. Repetir los pasos 2 y 3 para cada elemento de `expsion`.
2. Si el elemento es un operando, meterlo en la pila.

3. Si el elemento es un operador, lo designamos con &, entonces:
   - Sacar los dos elementos superiores de la pila, los denominamos con los identificadores b y a respectivamente.
   - Evaluar a & b; el resultado es z = a & b.
   - El resultado z, meterlo en la pila.
   - Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento cima de la pila.
5. Fin del algoritmo.

Codificacion de evaluacion de expresión en postfija

Las operaciones, funciones para manejar la pila de operandos (pila de números reales), son encapsuladas en el archivo pilaopdos.h; la codificación se encuentra en el archivo pilaopdos.c. Los nombres de las operaciones son cambiados para distinguirlos de los utilizados en la pila de caracteres, pilas.h. La parte de codificación es la misma, por ello, únicamente se escribe la interfaz pilaopdos.h.

```
/* archivo pilaopdos.h */

#include <stdlib.h>

typedf struct nodop
{
  TipoDatop elemento;
  struct nodop* siguiente;
} Nodop;

void crearPilap(Nodop** pila);
void insertarp(Nodop** pila, TipoDatop elemento);
void suprimirp(Nodop** pila);
TipoDatop quitarp(Nodop** pila);
void limpiarPilap(Nodop** pila);

TipoDatop cimap(Nodop* pila);

int pilaVaciap(Nodop* pila);
```

En el archivo con la función principal, main(), se escribe la función para asignar los valores numéricos de los operandos y la función evalua() en la que se codifica el algoritmo de evaluación.

```
#include "postfija.h"
typedef double TipoDatop;
#include "pilaoprdos.h"
#include <stdio.h>
#include <math.h>

double evalua(Elemento expsion[N], int n, double v[]);
void valorOprdos(Elemento expsion[N], int n, double v[]);

void main()
```

```
{
  double v[26];  /* vector para los valores de los operandos */
  Elemento expsion[N];
  int i, n;
  double resultado;

  puts("\nExpresión aritmética(en una línea) ");
  postFija(expsion, &n);
  printf("\nExpresión aritmética en postfija: ");
  for (i = 0; i < n; i++)
    printf("%c ", expsion[i].c);

  puts("\n Asignación de valores numéricos a los operandos ");
  valorOprdos(expsion, n, v);

  resultado = evalua(expsion, n, v);
  printf("\nResultado de la evaluación: %.2lf", resultado);
}

double evalua(Elemento expsion[N], int n, double v[])
{
  double valor, a, b;
  Nodop* pila;
  char op;
  int i;

  crearPilap(&pila);

  for (i = 0; i < n; i++)
  {
    if (!expsion[i].oprdor)
    {
      int indice;
      indice = expsion[i].c - 'A';
      insertarp(&pila, v[indice]);
    }

    else if (expsion[i].oprdor)
    {
      b = quitarp(&pila);
      a = quitarp(&pila);

      switch (expsion[i].c)
      {
        case '^': valor = pow(a,b);
                  break;
        case '*': valor = a * b;
                  break;
        case '/': if (b != 0.0)
                    valor = a / b;
                  else
                  {
                    puts ("\n División por cero.");
```

```
                    exit(1);
                }
                break;
            case '+': valor = a + b;
                break;
            case '-': valor = a - b;
        }
        insertarp(&pila, valor);
    }
  }
  return quitarp(&pila);
}

void valorOprdos(Elemento expsion[N], int n, double v[])
{
  /* se asignan valores numéricos a los operandos */
  int i;
  char ch;

  for (i = 0; i < n; i++)
  {
    if (!expsion[i].oprdor)
    {
      int indice;
      indice = expsion[i].c - 'A';
      printf(" %c = ", expsion[i].c);
      scanf("%lf", &v[indice]);
    }
  }
}
```

RESUMEN

- Una *pila* es una estructura de datos tipo **LIFO** (*last in first out*, «último en entrar, primero en salir») en la que los datos (todos del mismo tipo) se añaden y eliminan por el mismo extremo, denominado *cima* de la pila.
- Se definen las siguientes operaciones básicas sobre pilas: `crearPila`, `insertar`, `cima`, `suprimir`, `quitar`, `pilaVacia`, `pilaLlena` y `liberarPila`.
- `crearPila` inicializa la pila como pila vacía. Esta es la primera operación a ejecutar.
- `insertar` añade un elemento en la cima de la pila. Debe de haber espacio en la pila.
- `cima` devuelve el elemento que está en la cima, sin extraerlo.
- `suprimir` extrae de la pila el elemento cima.
- `quitar` extrae de la pila el elemento cima de la pila que es el resultado de la operación.
- `pilaVacia` determina si el estado de la pila es vacía, en su caso devuelve el valor que representa a *true*.
- `pilaLlena` determina si existe espacio en la pila para añadir un nuevo elemento. Si existe espacio devuelve `false(0)`. Esta operación se aplica en la representación de la pila mediante array.
- `liberarPila`: el espacio asignado a la pila se libera, queda disponible.
- Las aplicaciones de las pilas en la programación son numerosas; entre ellas está la evaluación de expresiones aritméticas. Primero se transforma la expresión a notación postfija, a continuación se evalúa.
- Las expresiones en notación *polaca*, *postfija* o *prefija*, tienen la característica de que no necesitan paréntesis.

EJERCICIOS

10.1. ¿Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es `int`?

```
Nodo* pila:
int x=4, y;

crearPila(&pila);
insertar(&pila, x);
printf("\n %d ", cima(pila));
y = quitar(&pila);
insertar(&pila, 32);
insertar(&pila, quitar(&pila));
do {
  printf("\n %d ", quitar(&pila));
}while (!pilaVacia(pila));
```

10.2. Utilizando una pila de caracteres, transformar la siguiente expresión a su equivalente expresión en postfija.

```
(x-y)/(z+w) - (z+y)^x
```

10.3. Transformar la expresión algebraica del Ejercicio 10.2 en su equivalente expresión en prefija.

10.4. Escribir la función `mostrarPila()` para escribir los elementos de una pila de cadenas de caracteres, utilizando sólo las operaciones básicas y una pila auxiliar.

10.5. Dada la expresión algebraica

```
r = x*y - (z+w)/(z+y)^x
```

transformar la expresión a notación postfija y a continuación evaluar la expresión para los valores: `x = 2`, `y = -1`, `z = 3`, `w = 1`. Para obtener el resultado de la evaluación seguir los pasos del algoritmo del Apartado 10.4.2.

10.6. Se tiene una lista enlazada a la que se accede por el puntero lista que referencia al primer nodo. Escribir una función que imprima los nodos de la lista en orden inverso, desde el último nodo al primero; como estructura auxiliar utilizar una pila y sus operaciones.

PROBLEMAS

10.1. Escribir una función, `copiarPila()`, que copie el contenido de una pila en otra. La función tendrá dos argumentos de tipo pila, uno para la pila fuente y otro para la pila destino. Utilizar las operaciones definidas sobre el tipo de datos pila.

10.2. Escribir una función para determinar si una secuencia de caracteres de entrada es de la forma:

```
X & Y
```

donde `X` es una cadena de caracteres e `Y` es la cadena inversa. El carácter `&` es el separador.

10.3. 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 expresión siguiente tiene un número de paréntesis equilibrado:

```
((a+b)*5) - 7
```

A esta otra expresión le falta un corchete:

```
2*[(a+b)/2.5 + x - 7*y
```

10.4. Se dispone de un archivo de texto del cual se quiere determinar las frases que son palíndromos. La estrategia sugerida es:

- 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*. Su comparación determina si es palíndromo o no.

Escribir un programa que lea cada línea del archivo y determine si es palíndromo.

10.5. Escribir un programa en el que se manejen un total de $n = 5$ pilas: P_1, P_2, P_3, P_4 y P_5. La entrada de datos será pares de enteros `(i,j)` tal que `1 ≤ abs(i) ≤ n`. El criterio de selección de pila:

- Si `i` es positivo, debe de insertarse el elemento `j` en la pila P_i.
- Si `i` es negativo, debe de eliminarse el elemento `j` de la pila P_i.
- Si `i` es cero, fin del proceso de entrada.

Los datos de entrada se introducen por teclado. Cuando termina el proceso el programa debe de escribir el contenido de la *n* pilas en pantalla.

10.6. Modificar el Problema 10.5 para que las entradas sean tripletas de números enteros `(i,j,k)`, donde `i,j` tienen el mismo significado que en 10.5, y `k` es un número entero que puede tomar los valores `-1`, `0` con este significado:

- `-1`, simplifica borrar todos los elementos de la pila.
- `0`, el proceso es el indicado en 10.5 con `i` y `j`.

10.7. Se desea sumar dos números enteros positivos arbitrariamente grandes. Estos números son tan grandes que no pueden almacenarse en una variable numérica. Utilizando el *TAD Pila*, escribir un programa para realizar la suma de dos enteros grandes.

11 COLAS

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Comprender la organización de datos en una cola.
- Conocer las operaciones básicas del *TAD Cola*.
- Definir el *TAD Cola* con memoria estática (arrays).
- Definir el *TAD Cola* con memoria dinámica (listas enlazadas).
- Encontrar las diferencias fundamentales entre pilas y colas.
- Aplicar el *TAD Cola* para resolver ciertos problemas.

CONTENIDO

11.1. Concepto de cola.
11.2. Tipo Cola implementada con arrays.
11.3. Definición del *TAD Cola* con arrays *circulares*.
11.4. Realización del *TAD Cola* con una lista enlazada.
11.5. Bicolas: colas de doble entrada.
RESUMEN
EJERCICIOS
PROBLEMAS

CONCEPTOS CLAVE

- Tipo abstracto de datos.
- Concepto de pila.
- Listas enlazadas.
- Listas circulares.
- Estructura FIFO.
- Tiempo de proceso.

INTRODUCCIÓN

En este capítulo se estudia pormenorizadamente la estructura de datos Cola, estructura muy utilizada en la vida cotidiana y también para resolver problemas en programación. Esta estructura, al igual que las pilas, almacena y recupera sus elementos atendiendo a un estricto orden. Las colas se conocen como estructuras **FIFO** (*First-in, First-out*, «primero en entrar, primero en salir»), debido a la forma y orden de inserción y de extracción de elementos de la cola. Las colas tienen numerosas aplicaciones en el mundo de la computación: colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades.

11.1. CONCEPTO DE COLA

Una **cola** es una estructura de datos que almacena elementos en una lista y permite acceder a los datos por uno de los dos extremos de la lista (Fig. 11.1). Un elemento se inserta en la cola (parte final) de la lista y se suprime o elimina por la frente (parte inicial, frente) de la lista. Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia.

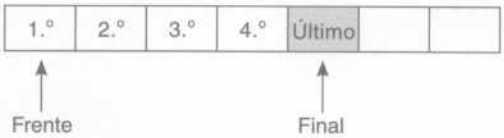


Figura 11.1. Una cola.

Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacenan y, por consiguiente, una cola es una estructura de tipo **FIFO** (*first-in/firs-out*, «primero en entrar/primero en salir» o bien «primero en llegar/primero en ser servido»). El servicio de atención a clientes en un almacén es un ejemplo típico de cola. La acción de gestión de memoria intermedia (*buffering*) de trabajos o tareas de impresora en un distribuidor de impresoras (*spooler*) es otro ejemplo típico de cola[1]. Dado que la impresión es una tarea (un trabajo) que requiere más tiempo que el proceso de la transmisión real de los datos desde la computadora a la impresora, se organiza una cola de trabajos de modo que los trabajos se imprimen en el mismo orden en que se recibieron por la impresora. Este sistema tiene el gran inconveniente de que si su trabajo personal consta de una única página para imprimir y delante de su petición de impresión existe otra petición para imprimir un informe de 300 páginas, deberá esperar a la impresión de esas 300 páginas antes de que se imprima su página.

A recordar

Una cola es una estructura de datos cuyos elementos mantienen un cierto orden, tal que sólo se pueden añadir elementos por un extremo, **final** de la cola, y eliminar o extraer por el otro extremo, llamado **frente**.

Las operaciones usuales en las colas son *Insertar* y *Quitar*. La operación **Insertar** añade un elemento por el extremo **final** de la cola, y la operación **Quitar** elimina o extrae un elemento por el extremo opuesto, el **frente** o **primero** de la cola. La organización de elementos en forma de cola asegura que *el primero en entrar es el primero en salir*. En la Figura 11.2 se realizan las operaciones básicas sobre colas, insertar y retirar elementos.

11.1.1. Especificaciones del tipo abstracto de datos Cola

Las operaciones que sirven para definir una cola y poder manipular su contenido son las siguientes:

Tipo de dato: Dato que se almacena en la cola.

Operaciones:

CrearCola Inicia la cola como vacía.

[1] Recordemos que este caso sucede en sistemas multiusuario donde hay varios terminales y sólo una impresora de servicio. Los trabajos se «encolan» en la cola de impresión.

Insertar	Añade un dato por el final de la cola.
Quitar	Retira (extrae) el elemento frente de la cola.
Cola vacía	Comprobar si la cola no tiene elementos.
Cola llena	Comprobar si la cola está llena de elementos.
Frente	Obtiene el elemento frente o primero de la cola.
Tamaño de la cola	Número de elementos máximo que puede contener la cola.

Desde el punto de vista de estructura de datos, una cola es similar a una pila. En una cola, los datos se almacenan de un modo lineal y el acceso a los datos sólo está permitido en los extremos de la cola.

La forma que los lenguajes tienen para representar el *TAD Cola* depende de dónde se almacenen los elementos, en un array o en una lista dinámica. La utilización de arrays tiene el problema de que la cola no puede crecer indefinidamente, está limitada por el tamaño del array; como contrapartida, el acceso a los extremos es muy eficiente. Utilizar una lista dinámica permite que el número de nodos se ajuste al de elementos de la cola; por el contrario, cada nodo necesita memoria extra para el enlace y también está el limite de memoria de la pila del computador.

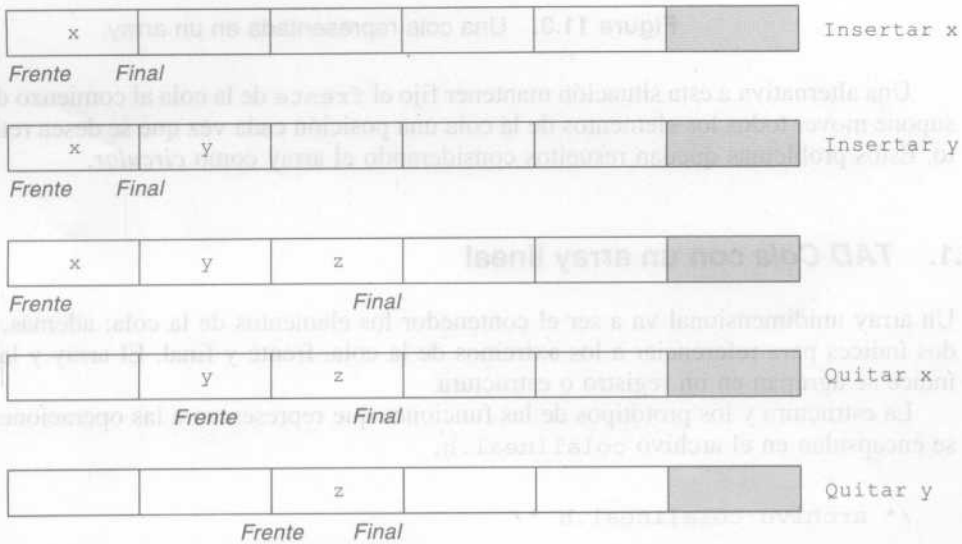


Figura 11.2. Operaciones Insertar y Quitar en una cola.

11.2. TIPO COLA IMPLEMENTADA CON ARRAYS

Al igual que las pilas, las colas se pueden implementar utilizando arrays o listas enlazadas. En esta sección se considera la implementación utilizando arrays. La definición de una `Cola` ha de contener un array para almacenar los elementos de la cola, y dos marcadores o apuntadores para mantener las posiciones `frente` y `final` de la cola; es decir, un marcador apuntando a la posición de la cabeza de la cola y el otro al primer espacio vacío que sigue al final de la cola. Cuando un elemento se añade a la cola, se verifica si el marcador `final` apunta a una posición válida, entonces se añade el elemento a la cola y se incrementa el marcador `final` en 1. Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador (puntero) de cabeza y éste se incrementa en 1.

La operación de añadir un nuevo elemento en la cola comienza a partir de la posición `final` 0, cada vez que se añade un nuevo elemento se incrementa `final` en 1. La extracción de un

elemento se hace por el `frente`, cada vez que se extrae un elemento avanza `frente` una posición. En la Figura 11.3 se puede observar cómo avanza el puntero `frente` al extraer un elemento.

El avance lineal de `frente` y `final` tiene un grave problema, deja *huecos* por la *izquierda del array*. Puede ocurrir que `final` alcance el índice más alto del array, no pudiéndose insertar nuevos elementos y, sin embargo, existir posiciones libres a la izquierda de `frente`.

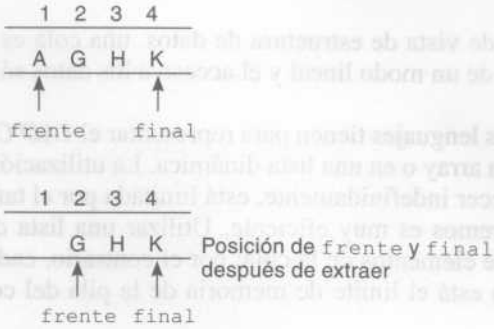


Figura 11.3. Una cola representada en un array.

Una alternativa a esta situación mantener fijo el `frente` de la cola al comienzo del array, lo que supone mover todos los elementos de la cola una posición cada vez que se desea retirar un elemento. Estos problemas quedan resueltos considerando el array como *circular*.

11.2.1. *TAD Cola* con un array lineal

Un array unidimensional va a ser el contenedor los elementos de la cola; además, son necesarios dos índices para referenciar a los extremos de la cola, frente y final. El array y las dos variables índice se agrupan en un registro o estructura.

La estructura y los prototipos de las funciones que representan a las operaciones del *TAD cola* se encapsulan en el archivo `colalineal.h`.

```
/* archivo colalineal.h */

#include <stdlib.h>

#define MAXTAMQ 100

typedef struct
{
   TipoDato listaCola[MAXTAMQ];
   int frente, final;
}Cola;

   /* Operaciones sobre la CCola */

void crearCola(Cola* cola);
void insertar(Cola* cola, TipoDato elemento);
TipoDato quitar(Cola* cola);

   /* Operación de acceso a Cola */
```

```
TipoDato frente(Cola cola);

  /* verificación estado de la Cola */

int colaVacia(Cola cola);
int colaLlena(Cola cola);
```

Antes de incluir el archivo `colalineal.h` debe de declararse el tipo de los elementos, `TipoDato`. Así, en el caso de una cola de caracteres:

```
typedef char TipoDato;
#include "colalineal.h"
```

y en el cado de desear mantener en la cola son punteros a nodos de una estructura dinámica:

```
typedef Nodo* TipoDato;
#include "colalineal.h"
```

Codificación de funciones del TAD Cola con array lineal

La codificación se realiza en el archivo `colalineal.c` que a continuación se desarrolla; se supone que el tipo de los elementos es entero.

```
  /* archivo colalineal.c */

typedef int TipoDato;
#include "colalineal.h"
```

CrearCola

La operación `crearCola()` inicializa la cola como cola vacía; al estar los arrays indexados con base 0, el índice `final`, apunta al último elemento, se pone a `-1`.

```
void crearCola(Cola* cola)
{
  cola -> frente = 0;
  cola -> final= -1;
}
```

Insertar

Añade el elemento `entrada` a la cola. Se modifica la cola por el extremo `final`. Es necesario verificar el estado de la cola, si está llena se produce un error.

```
void insertar(Cola* cola, TipoDato entrada)
{
  if (colaLlena(*cola))
  {
    puts("Cola completa");
    exit (1);
  }
```

```
  cola -> final++;
  cola -> listaCola[cola->final] = entrada;
}
```

Quitar

Elimina y devuelve el frente de la cola. En esta operación, la modificación de la cola es por el extremo `frente`. La operación extrae el elemento y lo devuelve.

```
TipoDato quitar(Cola* cola)
{
 TipoDato tem;

  if (colaVacia(*cola))
  {
    puts(" Se intenta sacar un elemento en cola vacía");
    exit (1);
  }

  tem = cola -> listaCola[cola->frente];

  cola -> frente++;
  return tem;
}
```

Cola vacía

Esta operación verifica el estado de la cola. Devuelve *cierto* (verdadero, 1) si la cola no tiene elementos.

```
int colaVacia(Cola cola)
{
  return cola.final < cola.frente;
}
```

Cola llena

La función `colaLlena()` comprueba si no se pueden añadir nuevos elementos a la cola. Este hecho no implica el que no haya posiciones libres en el array ya que al extraer el elemento primero, el frente avanza una posición quedando libre la posición anterior. En la Figura 11.3 se puede observar *el hueco* que queda al aplicar la operación de *quitar*. Una solución a este problema consiste en desplazar una posición a la izquierda todos los elementos que quedan en la cola cada vez que se extrae el frente. Esta alternativa no se considera por el coste adicional que supone todos estos desplazamientos. La alternativa más eficiente es considerar el array como *circular*; el siguiente apartado desarrolla esta alternativa.

```
int colaLlena(Cola cola)
{
```

```
  return cola.final == MAXTAMQ - 1;
}
```

Frente

Esta operación devuelve el elemento primero de la cola. Requerir el `frente` de una cola vacía se considera un error.

```
TipoDato frente(Cola cola)
{
  if (colaVacia(cola))
  {
    puts(" Error de ejecución, cola vacía");
    exit (1);
  }
  return cola.listaCola[cola.frente];
}
```

A tener en cuenta

La realización de una cola con un array lineal es notablemente ineficiente: se puede alcanzar la condición de cola llena existiendo elementos del array sin ocupar. Esto es debido a que al extraer un elemento, avanza el `frente`, y por consiguiente, las posiciones anteriores quedan desocupadas, no accesibles. No se aconseja utilizar esta implementación.

11.3. DEFINICIÓN DEL *TAD COLA* CON ARRAYS CIRCULARES

El medio más eficiente para almacenar una cola en un array es utilizar un tipo especial de array que una el extremo final de la cola con su extremo cabeza. Tal array se denomina *array circular* y permite que el array completo se utilice para almacenar elementos de la cola sin dejar posiciones libres a las que no se puede acceder; el array sigue siendo una estructura lineal, es a nivel lógico como se considera circular el array. Un array circular con *n* elementos se visualiza en la Figura 11.4.

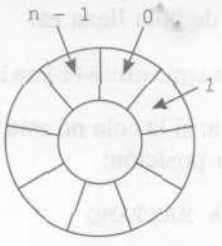


Figura 11.4. Un array circular.

El array se almacena de modo natural en la memoria tal como un bloque lineal de *n* elementos. Se necesitan dos marcadores (apuntadores) `frente` y `final` para indicar la posición del elemento cabeza y la posición del final, donde se almacenó el último elemento añadido.

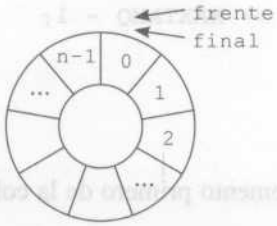


Figura 11.5. Una cola vacía.

La variable `frente` es siempre la posición del elemento primero de la cola y se avanza en el sentido de las agujas del reloj. La variable `final` es la posición de la última inserción; una nueva inserción supone mover `final` circularmente a la derecha y asignar el nuevo elemento.

La simulación del movimiento circular de los índices se realiza utilizando la *teoría de los restos* de tal forma que se generen índices de `0` a `MAXTAMQ-1`:

Mover `final` adelante = `(final + 1) % MAXTAMQ`
Mover `frente` adelante = `(frente + 1) % MAXTAMQ`

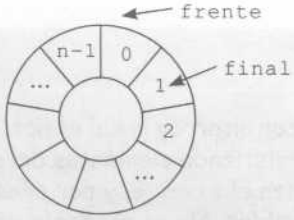


Figura 11.6. Una cola que contiene un elemento.

Los algoritmos que formalizan la gestión de colas en un array circular han de incluir las operaciones básicas del *TAD Cola*, en concreto, al menos las siguientes tareas:

- Creación de una cola vacía, de tal forma que `final` apunte a una posición inmediatamente anterior a `frente`: `frente = 0; final = MAXTAMQ-1`.
- Comprobar si una cola está vacía:

  es `frente == siguiente(final) ?`

- Comprobar si una cola está llena. Para distinguir de la condición de *cola llena* de *cola vacía* se sacrifica una posición del array, de tal forma que la capacidad de la cola va a ser `MAXTAMQ-1`. La condición de cola llena es:

  ```
  frente == siguiente(siguiente(final)) ?
  ```

- Añadir un elemento a la cola: si la cola no está llena, establecer `final` a la siguiente posición y añadir el elemento en esa posición:

  ```
  final = (final + 1) % MAXTAMQ
  ```

- Eliminación de un elemento de una cola: si la cola no está vacía, suprimirlo de la posición frente y establecer `frente` a la siguiente posición:

  ```
  (frente + 1) % MAXTAMQ
  ```

- Obtener el elemento primero de la cola —si la cola no está vacía— sin suprimirlo de la cola.

11.3.1. Codificación del *TAD Cola* con un array circular

La representación de los elementos de la cola se hace de igual manera que con un array lineal. Las funciones que representan a las operaciones tienen también la misma interfaz considerada en el archivo `colalineal.h`. Por esta razón el archivo `colacircular.h` es similar.

A continuación se escribe el código de cada una de las funciones del *TAD Cola*. Además, para obtener la posición siguiente de una dada, aplicando la *teoría de los módulos*, se escribe la función `siguiente()` se obtiene el archivo `colacircular.c`.

Siguiente circular

```
int siguiente(int n)
{
  return (n + 1) % MAXTAMQ;
}
```

Crear Cola

La primera operación que se realiza sobre una cola es su creación como *cola vacía*, con el objetivo de permitir que se puedan añadir elementos a la cola.

```
void crearCola(Cola* cola)
{
  cola -> frente = 0;
  cola -> final = MAXTAMQ-1;
}
```

Insertar

Antes de que comience el proceso de inserción, el índice `final` apunta al último elemento insertado; éste avanza circularmente y se sitúa el nuevo elemento.

```
void insertar(Cola* cola, TipoDato entrada)
{
  if (colaLlena(*cola))
  {
    puts(" desbordamiento cola");
    exit (1);
  }
    /* avance circular al siguiente del final */

cola -> final = siguiente(cola -> final);
cola -> listaCola[cola -> final] = entrada;
}
```

Quitar

Esta operación retira el elemento del frente de la cola que, además, es el resultado de la operación. El índice `frente` avanza *circularmente*.

```
TipoDato quitar(Cola* cola)
{
  TipoDato tmp;
  if (colaVacia(*cola))
  {
    puts(" Extracción en cola vacía ");
    exit (1);
  }

  tmp = cola -> listaCola[cola->frente];

    /* avanza circularmente frente */

  cola -> frente = siguiente(cola -> frente);
  return tmp;
}
```

Frente

Obtiene el elemento del frente de la cola, sin modificar la cola.

```
TipoDato frente(Cola cola)
{

  if (colaVacia(cola))
  {
    puts(" Se requiere frente de una cola vacía ");
    exit (1);
  }

  return cola.listaCola[cola.frente];
}
```

Cola vacía

Esta operación verifica el estado de la cola y prueba si la cola no tiene elementos.

```
int colaVacia(Cola cola)
{
  return cola.frente == siguiente(cola.final);
}
```

Cola llena

Esta operación prueba si la cola no puede contener más elementos. En el array queda una posición no ocupada, y así se distingue de la condición de cola vacía.

```
int colaLlena(Cola cola)
{
  return cola.frente == siguiente(siguiente(cola.final));
}
```

Ejemplo 11.1

Encontrar un número capicúa leído del dispositivo estándar de entrada.

Para encontrar el capicúa buscado se utiliza el *TAD Cola*. Los dígitos se leen carácter a carácter (un dígito es un carácter del '0' al '9'), almacenándose en una cola y a la vez en una pila. Una vez leído el número se extraen consecutivamente elementos de la cola y de la pila y se comparan por igualdad. *De producirse alguna «no coincidencia» es que el número no es capicúa y en ese caso se* vacían las estructuras para pedir, a continuación, otra entrada. El número es capicúa si el proceso de verificación termina coincidiendo todos los dígitos en orden inverso, por tanto, con la pila y la cola vacía.

¿Por qué utilizar una pila y una cola? Sencillamente, por el orden inverso en procesar los elementos; en la pila el *último en entrar es el primero en salir*, en la cola el *primero en entrar es el primero en salir*.

Sólo se incluye el archivo con la función principal, se crean tanto la pila como la cola y después se invoca a las operaciones de *Insertar* y *Quitar*. Con el fin de que no se produzca colisión con los nombres de las operaciones, en el *TAD Pila* se añade la letra *'p'* a *Insertar* y *Quitar*: `insertarp()`, `quitarp()`.

```
typedef char TipoDato;
#include "colacircular.h"
#include "pila.h"
#include <stdio.h>

void main()
{
   char d;
   Cola cola;
   Pila pila
   int capicua;

   capicua = 0;
   crearCola(&q);
   crearPila(&pila);

   while (!capicua)
   {
      printf(" Número a investigar: ");
      while ((d = getchar())!= '\n')
      {
         if ( d < '0' || d > '9')
         {
            puts(" \n Error en el número introducido ");
         }
         else
         {
            insertar(&cola, d);
            insertarp(&pila, d);
         }
      }
```

```
        capicua = 0;

        do {
           capicua = quitar(&cola) == quitarp(&pila);
        } while (capicua && !colaVacia(cola));

        if (capicua)
          puts(" \n El numero introducido es capicúa ");
        else
        {
          puts(" \n El numero no es capicúa, intente con otro " );
          crearCola(&cola);  /* se inicializan las estructuras */
          crearPila(&pila);
        }
    }
}
```

11.4. REALIZACIÓN DEL *TAD COLA* CON UNA LISTA ENLAZADA

La realización de una cola con arrays exige reservar memoria contigua para el máximo de elementos previstos. En muchas ocasiones esta característica produce un desaprovechamiento de memoria; también puede ocurrir lo contrario, que se llene la cola y no se pueda seguir con la ejecución del programa. La alternativa reside en utilizar memoria que se ajusta en todo momento al número de elementos de la cola, empleando memoria dinámica mediante una lista enlazada, aun a pesar de tener el inconveniente de la memoria extra utilizada para realizar los encadenamientos entre nodos. Esta implementación utiliza dos punteros para acceder a la lista, `frente` y `final` (los extremos por donde salen los elementos y por donde se insertan respectivamente).

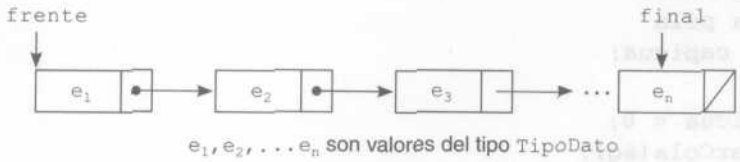


Figura 11.7. Cola con lista enlazada.

La variable puntero `frente` referencia al primer elemento de la cola, el primero en ser retirado de la cola. La otra variable puntero, `final`, referencia al último elemento en ser añadido, que será el último en ser retirado.

Al ser una estructura dinámica puede crecer y decrecer según las necesidades (el límite está en la memoria libre del computador), y por tanto, no tiene sentido la operación que prueba si la cola está llena.

Nota de programación

Una estructura dinámica puede crecer y decrecer según las necesidades, según el número de nodos (el límite está en la memoria libre del computador). En una estructura dinámica no tiene sentido la operación que verifica un posible *desbordamiento*; esta prueba sí es necesaria en las estructuras de tamaño fijo.

11.4.1. Declaración de tipos de datos y operaciones

La representación del tipo abstracto de datos cola con listas enlazadas maneja dos tipos de datos; el tipo nodo de la lista, y el tipo que *agrupa* a las variables puntero `frente` y `final` que es la estructura `Cola`. En el nodo se encapsula los elementos de la cola y el campo de enlace que es una variable puntero a nodo.

En el archivo `coladinamica.h` se declaran estos tipos y los prototipos de las funciones que representan a las operaciones básicas del *TAD Cola*.

```
/* archivo coladinamica.h */

#include <stdlib.h>

struct nodo
{
  TipoDato elemento;
  struct nodo* siguiente;
};

typedef struct nodo Nodo;
typedef struct
{
  Nodo* frente;
  Nodo* final;
}Cola;

/* prototipos de las operaciones */

void crearCola(Cola* cola);
void insertar(Cola* cola,TipoDato entrada);
TipoDato quitar(Cola* cola);
void borrarCola(Cola* cola);    /* libera todos los nodos de la cola */

/* acceso a la cola */
TipoDato frente(Cola cola);

/* métodos de verificación del estado de la cola */
int colaVacia(Cola cola);
```

11.4.2. Codificación de las funciones del *TAD Cola* con listas

La codificación se almacena en el archivo fuente `coladinamica.c`. En primer lugar, hay que declarar el tipo concreto de los elementos de la cola e incluir el archivo `coladinamica.h`.

La inicialización de la cola, al ser una realización con punteros, consiste en asignar el puntero nulo a `frente` y `final`.

La operación insertar se realiza creando un nuevo nodo, para lo cual se escribe la función `crearNodo()` y se enlaza a partir del nodo `final`. La operación de retirar un elementos de la cola se realiza sobre el otro extremo, el extremo `frente`.

El tipo de los datos de los elementos de la cola se supone cadenas de caracteres, representado por `char*`. La codificación completa es:

```
/* archivo coladinamica.c */
typedef char* TipoDato;
#include "coladinamica.h"

void crearCola(Cola* cola)
{
  cola -> frente = cola -> final = NULL;
}

Nodo* crearNodo(TipoDato elemento)
{
  Nodo* t;
  t = (Nodo*)malloc(sizeof(Nodo));
  t -> elemento = elemento;
  t -> siguiente = NULL;
  return t;
}

int colaVacia(Cola cola)
{
  return (cola.frente == NULL);
}

void insertar(Cola* cola, TipoDato entrada)
{
  Nodo* a;
  a = crearNodo(entrada);
  if (colaVacia(*cola))
  {
    cola -> frente = a;
  }
  else
  {
    cola -> final -> siguiente = a;
  }
  cola -> final = a;
}

TipoDato quitar(Cola* cola)
{
  TipoDato tmp;
  if (!colaVacia(*cola))
  {
    Nodo* a;
    a = cola -> frente;
    tmp = cola -> frente -> elemento;
    cola -> frente = cola -> frente -> siguiente;
```

```
    free(a);
  }
  else
  {
    puts("Error cometido al eliminar de una cola vacía");
    exit(-1);
  }
  return tmp;
}
TipoDato frente(Cola cola)
{
  if (colaVacia(cola))
  {
    puts(" Error: cola vacía ");
    exit(-1);
  }
  return (cola.frente -> elemento);
}

void borrarCola(Cola* cola)
{
  /* Elimina y libera todos los nodos de la cola */
  for (; cola->frente != NULL;)
  {
    Nodo* n;
    n = cola -> frente;
    cola -> frente = cola -> frente -> siguiente;
    free(n);
  }
}
```

11.4.3. Realizacion del *TAD Cola* con una lista circular

Esta otra forma de implementar una cola es una variante de la realización con listas enlazadas. Al utilizar una lista circular, por conveniencia, se establece que el puntero de acceso a la lista, `lc`, referencia al último nodo o elemento insertado, y que el nodo siguiente es considerado el primero o el que más tarde se insertó. Según este convenio, y teniendo en cuenta la definición de cola, las inserciones de nuevos elementos de la cola serán realizados por el nodo referenciado por `lc` (final de la cola) y la retirada de elementos o extracción, por el nodo siguiente a `lc` (primero de la cola). De esta manera, con un único puntero se puede acceder al final y al frente de la cola.

Las declaraciones de tipos para esta representación:

```
/* archivo colaciar.h */

#include <stdlib.h>

struct nodo
{
  TipoDato elemento;
  struct nodo* siguiente;
};
```

```
typedef struct nodo Nodo;
typedef Nodo* Cola;

/* prototipos de las operaciones */

void crearCola(Cola cola);
void insertar(Cola cola,TipoDato entrada);
TipoDato quitar(Cola cola);
void borrarCola(Cola cola);
TipoDato frente(Cola cola);
int colaVacia(Cola cola);
```

Las operaciones son similares a las realizadas con listas circulares. El puntero para realizar inserciones y eliminaciones es el mismo: `cola`. Teniendo en cuenta que `cola` referencia al último (`final`), y `cola ->siguiente` al primero (`frente`). La función auxiliar `crearNodo()` cambia para reflejar que el nodo sea circular.

```
Nodo* crearNodo(TipoDato elemento)
{
  Nodo* t;
  t = (Nodo*)malloc(sizeof(Nodo));
  t -> elemento = elemento;
  t -> siguiente = t;
  return t;
}

void insertar(Cola* cola, TipoDato entrada)
{
  Nodo* a;
  a = crearNodo(entrada);

  if (!colaVacia(*cola))
  {
    /*inserta a partir de cola */
    a -> siguiente = (*cola) -> siguiente;
    (*cola) -> siguiente = a;
  }
  *cola = a
}
TipoDato quitar(Cola* cola)
{
  TipoDato tmp;
  Nodo *a
  if (!colaVacia(*cola))
  {
    a = (*cola) -> siguiente;
    tmp = a -> elemento;
    if (a == (*cola)) (*cola) = NULL;
      else (*cola) -> siguiente = a -> siguiente;
    free(a);
```

```
    }
    else
    {
      puts("Error cometido al eliminar de una cola vacía");
      exit(-1);
    }
    return tmp;
  }
```

Ejercicio 11.1

Una variación del problema matemático llamado «problema de José» permite generar números de la suerte. Se parte de una lista inicial de `n` números; esta lista se va reduciendo con el siguiente algoritmo:

1. Se genera un número aleatorio `n1`.
2. Si `n1 <= n` se quitan de la lista los números que ocupan las posiciones `1, 1+n1, 1+2*n1, ...` ; `n` toma el valor del número de elementos que quedan en la lista.
3. Se vuelve al paso 1.
4. Si `n1 > n`, fin del algoritmo; los números de la suerte son los que quedan en la lista.

El problema se va a resolver utilizando el *TAD Cola*. En primer lugar, se genera una lista de `n` números aleatorios que se almacena en una cola. A continuación se siguen los pasos del algoritmo; en cada pasada se eliminan los elementos de la cola que están en las posiciones `(múltiplos de n1) +1`. Estas posiciones `i` se pueden expresar matemáticamente:

```
i modulo n1 = 1
```

Una vez que termina el algoritmo se extraen los elementos que han quedado en la cola, que son los números de la suerte.

El *TAD Cola* que se utiliza está realizado con listas enlazadas. No se vuelve a escribir el archivo `coladinamica.c`, únicamente se escribe el archivo con la función principal y, además, la función `mostrarCola()`, que muestra por pantalla los elementos que han quedado en la cola una vez seguida la estrategia propuesta en el algoritmo.

```
typedef int TipoDato;
#include "coladinamica.h"
#include <time.h>
#include <stdio.h>

void mostrarCola(Cola* cola);

void main()
{
  Cola cola;
  int n, n1, n2, n3, i;

  randomize();
    /* Número de elementos de la lista */
  n = 1 + random(50);
  crearCola(&cola);
    /* Se generan n números aleatorios */
  for (i = 1; i<= n; i++)
```

```
        insertar(&cola, 1+random(1001));

    n1 = 1 + random(11);
    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 = quitar(&cola);
        if (i % n1 == 1)
        {
          printf("\t %d se quita.", n3);
        }
        else
        {
          insertar(&cola, n3);        /* se vuelve a meter en la cola */
          n2++;
        }
      }
      n = n2;
      n1 = 1 + random(11);
    }

    printf("\n Los números de la suerte: ");
    mostrarCola(&cola);

  }

void mostrarCola(Cola* cola)
{
  while (!colaVacia(*cola))
  {
    printf("%d ", quitar(cola));
  }
}
```

11.5. BICOLAS: COLAS DE DOBLE ENTRADA

Como una variación del *TAD Cola* se puede considerar la estructura *bicola* o *cola de doble entrada*. Una **bicola** es un conjunto ordinal de elementos del cual se pueden añadir o quitar elementos desde cualquier extremo de la misma; el acceso a la bicola está permitido desde cualquier extremo. Se puede decir que una bicola es una cola bidireccional.

Los dos extremos de una bicola se pueden identificar como `izquierdo` y `derecho` respectivamente. Las operaciones básicas que definen una bicola son una ampliación de la operaciones que define una cola:

CrearBicola	Inicializa una bicola sin elementos.
BicolaVacia	Devuelve true si la bicola no tiene elementos.
InserIzqda	Añade un elemento por extremo izquierdo.
InserDrcha	Añade un elemento por extremo derecho.

QuitarIzqda	Devuelve el elemento Izquierdo y lo retira de la bicola.
QuitarDrcha	Devuelve el elemento Derecho y lo retira de la bicola.
Izquierdo	Devuelve el elemento que se encuentra en el extremo izquierdo.
Derecho	Devuelve el elemento que se encuentra en el extremo derecho.

Al tipo de datos bicola se puede poner restricciones respecto al tipo de entrada o al tipo de salida. Una *bicola con restricción de entrada* es aquella que sólo permite inserciones por uno de los dos extremos, pero que permite retirar elementos por los dos extremos. Una *bicola con restricción de salida* es aquella que permite inserciones por los dos extremos, pero sólo permite retirar elementos por un extremo.

Para representar una bicola se puede elegir una representación estática, con arrays, manteniendo los elementos de la bicola con un array circular y dos variables índice que se correspondan con ambos extremos, el extremo izquierdo y derecho respectivamente.

11.5.1. Representación de una bicola con listas dobles

Otra posible representación de una bicola es la representación dinámica, ajustando el tamaño al número de elementos. Para esta representación de la bicola puede elegirse una lista doblemente enlazada para guardar los elementos, los dos extremos de la lista representan a los extremos de la bicola.

Los tipos de datos a declarar son el del nodo de una lista doble, con dos campos puntero para enlazar con el nodo anterior y el siguiente. Además, una estructura para encerrar las dos variables que apuntan a los extremos de la bicola. En el archivo `bicola.h` se declaran estos tipos y los prototipos de las funciones que representan a las operaciones del *TAD Bicola*.

```
/* archivo bicola.h */

#include <stdlib.h>

struct nodo
{
  TipoDato elemento;
  struct nodo* siguiente;
  struct nodo* anterior;
};
typedef struct nodo Nodo;
typedef struct
{
  Nodo* izquierdo;
  Nodo* derecho;
}Bicola;

/* prototipos de las operaciones */

void crearBicola(Bicola* bicola);
void inserIzqda(Bicola* bicola,TipoDato entrada);
void inserDrcha(Bicola* bicola,TipoDato entrada);
TipoDato quitarIzqda(Bicola* bicola);
TipoDato quitarDrcha(Bicola* bicola);
int bicolaVacia(Bicola bicola);
```

```
/* acceso a la bicola */
TipoDato izquierdo(Bicola bicola);
TipoDato derecho(Bicola bicola);
```

La implementación de estas operaciones no supone ninguna dificultad, se pueden consultar los Apartados 9.9 y 9.10 en el capítulo de listas enlazadas.

Ejercicio 11.2

La salida a pista de las avionetas de un aeródromo está organizada en forma de fila (línea), con una capacidad máxima de aparatos en espera de 13 avionetas. Las avionetas llegan por el extremo izquierdo y salen por el extremo derecho. Un piloto puede decidir retirarse de la fila por razones técnicas, en ese caso todas las avionetas a su derecha han de ser retiradas, retirar el aparato y las avionetas desplazadas colocarlas de nuevo en el mismo orden relativo en que estaban. La salida de una avioneta de la fila supone que las demás se mueven hacia adelante, de tal forma que los espacios libres del estacionamiento estén en la parte izquierda.

El programa para emular este estacionamiento tiene como entrada un carácter que indica una acción sobre la avioneta, y la matrícula de la avioneta. La acción puede ser llegada (E), salida (S) de la avioneta que ocupa la primera posición y retirada (T) de una avioneta de la fila. En la llegada puede ocurrir que el estacionamiento esté lleno; si ocurre esto la avioneta espera hasta que se quede una plaza libre.

El estacionamiento va a estar representado por una bicola de salida restringida. ¿Por qué esta elección? La salida siempre se hace por el mismo extremo, sin embargo, la entrada se puede hacer por los dos extremos, y así contemplar dos acciones: que llegue una avioneta nueva y que tenga que entrar una avioneta que ha sido movida para que salga una intermedia.

Las avionetas que se mueven para poder retirar del estacionamiento una intermedia, se disponen en una lista *LIFO*, así la ultima en entrar será la primera en añadirse en el extremo salida del estacionamiento (bicola) y seguir en el mismo orden relativo.

Las avionetas se pueden representar mediante una cadena para almacenar el número de matrícula (tipo *avioneta*). Los elementos de la pila y de la bicola son del tipo *avioneta*. La lista *LIFO* es, en realidad, una pila. La codificación de las operaciones sobre pilas se encuentran el el Capítulo 10, ahora simplemente se incluye el archivo `pila.h`.

Las operaciones del *TAD Bicola* se realizan considerando el almacenamiento de los elementos en un array circular. La restricción de salida se refleja en que no es válida la operación `quitarIzqda`. Los extremos de la estructura se denominan `entrada` y `salida`, de tal forma que son índices de las avionetas que menos tiempo lleva en la fila y que más tiempo lleva respectivamente. En el archivo `bicolacircular.h` se encuentran los tipos de datos y los prototipos de las operaciones. Los tipos y constantes:

```
#define MAXTAMQ 14

typedef struct
{
  TipoDato listaBicola[MAXTAMQ];
  int salida, entrada;
}Bicola;
```

A continuación se presenta el archivo con la implementación de las operaciones más importantes. Para mover circularmente los índices hacia delante y hacia atrás se codifican las funciones `siguiente()` y `anterior()` respectivamente.

Codificación del TAD Bicola

```
typedef struct
{
  char matricula[51];
} Avioneta;
typedef Avioneta TipoDato;
#include "bicolacircular.h"

int siguiente(int n)
{
  return (n + 1) % MAXTAMQ;
}

int anterior(int n)
{
    /* anterior de 0 es el indice mas alto */
  return ((n-1) < 0) ? MAXTAMQ-1 : n-1;

}

void crearBicola(Bicola* bicola)
{
  bicola -> salida = 0;
  bicola -> entrada = MAXTAMQ-1;
}

void inserIzqda(Bicola* bicola, TipoDato avta)
{
  if (!bicolaLlena(*bicola))
  {
    bicola -> entrada = siguiente(bicola->entrada);
    bicola -> listaBicola[bicola->entrada] = avta;
  }
}

void inserDrcha(Bicola* bicola, TipoDato avta)
{
  if (!bicolaLlena(*bicola))
  {
    bicola -> salida = anterior(bicola->salida);
    bicola -> listaBicola[bicola->salida] = avta;
  }
}

TipoDato quitarIzqda(Bicola* bicola)
{
  TipoDato tmp;

  if (bicolaVacia(*bicola))
  {
    puts(" Extracción por entrada en bicola vacía ");
```

```
    exit (1);
  }

  tmp = bicola -> listaBicola[bicola->entrada];
    /* retrocede el índice entrada */
  bicola -> entrada = anterior(bicola->entrada);
  return tmp;
}

TipoDato quitarDrcha(Bicola* bicola)
{
  TipoDato tmp;

  if (bicolaVacia(*bicola))
  {
    puts(" Extracción en bicola vacía ");
    exit (1);
  }

  tmp = bicola -> listaBicola[bicola->salida];
    /* avanza el índice salida */
  bicola -> salida = siguiente(bicola->salida);
  return tmp;
}

int bicolaVacia(Bicola bicola)
{
  return bicola.salida == siguiente(bicola.entrada);
}

int bicolaLlena(Bicola bicola)
{
  return bicola.salida == siguiente(siguiente(bicola.entrada));
}
```

Codificación de la función principal

En el archivo fuente con la función principal se codifica la simulación a realizar. Se incorpora la función `posicion()` que determina la posición que ocupa una avioneta en la fila. También, la operación `retirar()` que simula el hecho de que una avioneta que se encuentra en cualquier posición de la bicola decide salir de la fila; esta función hace uso del *TAD Pila* para guardar las avionetas que temporalmente tienen que *apartarse* de la fila para que pueda salir la avioneta en cuestión.

```
typedef struct
{
  char matricula[51];
} Avioneta;
typedef Avioneta TipoDato;
#include "bicolacircular.h"
#include "pila.h"
#include <ctype.h>
```

```
#include <stdio.h>

int posicion(Bicola bicola, Avioneta av);
int retirar(Bicola* bicola, Avioneta av);

void main()
{
  Avioneta u;
  char ch;
  Bicola fila;
  int esta, mas = 1;

  crearBicola(&fila);

  while (mas)
  {
    puts("\n Entrada de datos: [acción: E/S/T] matrícula.");
    puts(" Para terminar la simulación: X.");

    do {
      scanf("%c%*c",&ch); ch = tolower(ch);
    } while(ch != 'e' && ch != 's' && ch != 't' && ch != 'x');

    if (ch == 's')
    {
      if (!bicolaVacia(fila))
      {
        u = quitarDrcha(&fila);
        printf("Salida de la avioneta: %s", u.matricula);
      }
    }
    else if (ch == 'e')
    {
      if (!bicolaLlena(fila))
      {
        gets(u.matricula);
        inserIzqda(&fila, u);
      }
    }
    else if (ch == 't')
    {
      if (!bicolaLlena(fila))
      {
        gets(u.matricula);
        esta = retirar(&fila, u);
        if (!esta) puts("Error de datos, avioneta no encontrada");
      }
    }
    mas = !(ch == 'x');
  }
}
```

```
int posicion(Bicola bicola, Avioneta a)
{
  int p, encontrada;

  p = anterior(bicola.salida);
  encontrada = 0;
  while (!encontrada && p != bicola.entrada)
  {
    p = siguiente(p);
    encontrada = strcmp(bicola.listaBicola[p].matricula,
                        a.matricula) == 0;
  }
  if (encontrada)
    return p;
  else
    return -1;
}

int retirar(Bicola* bicola, Avioneta av)
{
  int p;

  p = posicion(*bicola, av);
  if (p == -1)                    /* error no está la avioneta */
    return 0;
  else
  {
    Pila pila

    if (p == bicola->salida) /* está en el extremo de salida */
      quitarDrcha(bicola);
    else
    {
        /* se meten en la pila los elementos siguientes a p */
      crearPila(&pila);
      do {
        insertar(&pila, quitarDrcha(bicola));
      } while (bicola->salida != p);
      printf("Avioneta %s es retirada\n",
                        quitarDrcha(bicola).matricula);

        /* elementos de la pila se meten de nuevo en la fila */

      while (!pilaVacia(pila))
      {
        inserDrcha(bicola,quitar(&pila));
      }
    }
    return 1;        /* operación realizada */
  }
}
```

La resolución del problema no toma acción cuando una avioneta no puede incorporarse a la fila por estar llena. El lector puede añadir el código necesario para que, utilizando una cola, las avionetas que no pueden entrar en la fila se guarden en la cola, y cada vez que salga una avioneta se añada una desde la cola.

RESUMEN

- Una cola es una lista lineal en la cual los datos se insertan por un extremo (*final*) y se extraen por el otro extremo (*frente*). Es una estructura FIFO (*first in first out*, «primero en entrar, primero en salir»).
- *Las operaciones básicas que se aplican sobre colas son:* `crearCola`, `colaVacia`, `colaLlena`, `insertar`, `frente`, `retirar`.
- `crearCola` inicializa a una cola sin elementos. Es la primera operación a realizar con una cola.
- `colaVacia` determina si una cola tiene o no elementos. Devuelve 1 (`true`) si no tiene elementos.
- `colaLlena` determina si no se pueden almacenar más elementos en una cola. Se aplica esta operación cuando se utiliza un array para guardar los elementos de la cola.
- `insertar` añade un nuevo elemento a la cola, siempre por el extremo final.
- `frente` devuelve el elemento que está en el extremo frente de la cola, sin extraerlo.
- `retirar` extrae el elemento frente de la cola.
- *El TAD Cola se puede implementar con arrays y con listas enlazadas. La implementación con un array* lineal es muy ineficiente; se ha de considerar el array como una estructura circular y aplicar la teoría de los restos para avanzar el frente y el final de la cola.
- La realización de una cola con listas enlazadas permite que el tamaño de la estructura se ajuste al número de elementos. La cola puede crecer indefinidamente, con el único tope de la memoria libre.
- Numerosos modelos de sistemas del mundo real son de tipo cola: cola de impresión en un servidor de impresoras, programas de simulación, colas de prioridades en organización de viajes. Una cola es la estructura típica que se suele utilizar como almacenamiento de datos, cuando se envían datos desde un componente rápido de una computadora a un componente lento (por ejemplo, una impresora).
- *Las bicolas son colas dobles, las operaciones básicas de insertar y retirar elementos se pueden realizar por* los dos extremos. A veces se ponen restricciones de entrada o de salida por algún extremo.

EJERCICIOS

11.1. Considerar una cola de nombres representada por una array circular con 6 posiciones, el campo *frente con el valor:* `frente = 2`, *y los elementos de la cola:* `Mar`, `Sella`, `Centurión`. *Escribir los* elementos de la cola y los campos `frente` y `final` según se realizan estas operaciones:

- Añadir `Gloria` y `Generosa` a la cola.
- Eliminar de la cola.
- Añadir `Positivo`.
- Añadir `Horche` a la cola.
- Eliminar todos los elementos de la cola.

11.2. *Una* `bicola` *es una estructura de datos lineal cuyas operaciones de inserción y eliminación se* pueden hacer tanto por el extremo `frente` como por el extremo `final`. Suponer que se ha elegido una representación de los elementos con una lista enlazada, y que los extremos de la lista se denominan `frente` y `final`. Escribir la clase `Bicola` con la representación de los datos y la implementación de las operaciones:

`insertarFrente()`, `insertarFinal()`, `eliminarFrente()` y `eliminarFinal()`

11.3. Considerar una bicola de caracteres, representada en un array circular. El array consta de 9 posiciones. Los extremos actuales y los elementos de la bicola:

```
frente = 5 final = 7     Bicola: A,C,E
```

Escribir los extremos y los elementos de la bicola según se realizan estas operaciones:

- Añadir los elementos `F` y `K` por el `final` de la bicola.
- Añadir los elementos `R`, `W` y `V` por el `frente` de la bicola.
- Añadir el elemento `M` por el `final` de la bicola.
- Eliminar dos caracteres por el `frente`.
- Añadir los elementos `K` y `L` por el `final` de la bicola.
- Añadir el elemento `S` por el `frente` de la bicola.

11.4. A la estructura que representa una cola implementada con un array circular y dos variables `frente` y `final`, se le añade una variable más que guarda el número de elementos de la cola. Escribir de nuevo las funciones de manejo de colas considerando este campo contador.

11.5. Supóngase que se tiene ya codificadas las declaraciones para definir el tipo cola. Escribir una función para crear una copia de una cola determinada. La función tendrá dos argumentos, el primero es la cola origen y el segundo la cola que va a ser la copia. Las operaciones que se han de utilizar serán únicamente las del *TAD Cola*.

11.6. Se ha estudiado en el capítulo la realización de una cola mediante un array circular, una variación a esta representación es aquella en la que además de tener una variable con la posición del elemento frente, otra variable con la longitud de la cola (número de elementos), `longCola`, y el array considerado circular. Dibujar una cola vacía; añadir a la cola 6 elementos; extraer de la cola tres elementos; añadir elementos hasta que haya desbordamiento. En todas las representaciones escribir los valores de `frente` y `longCola`.

11.7. Escribir las funciones correspondientes a las operaciones del *TAD Cola* para la realización del Ejercicio 11.6.

PROBLEMAS

11.1. Con un archivo de texto se desea realizar las siguientes acciones: formar una lista de colas, de tal forma que en cada nodo de la lista contenga la dirección de una cola con todas las palabras del archivo que empiezan por una misma letra. Visualizar las palabras del archivo, empezando por la cola que contiene las palabras que comienzan por la letra *a*, a continuación las palabras que empiezan por la letra *b*, y así sucesivamente.

11.2. Se tiene un archivo de texto del cual se quiere determinar las frases que son palíndromos. *Sugerencia:* seguir la siguiente estrategia:

- Considerar cada línea del texto una frase.
- Añadir cada carácter de la frase a una pila y a la vez a una cola.
- Extraer carácter a carácter, y simultáneamente de la pila y de la cola. Su comparación determina si es un palíndromo o no.

Escribir un programa que lea cada línea del archivo y determine si es un palíndromo.

11.3. Escribir un programa en el que se generen 100 números aleatorios en el rango -25 ... +25 y se guarden en una cola implementada mediante un array considerado circular. Una vez creada la cola, el usuario puede pedir que se forme otra cola con los números negativos que tiene la cola original.

11.4. Escribir una función que tenga como argumentos dos colas del mismo tipo y devolver *cierto* (verdadero) si las dos colas son idénticas utilizando las operaciones del *TAD Cola*.

11.5. Un pequeño supermercado dispone en la salida de tres cajas de pago. En el local hay 25 carritos de compra. Escribir un programa que simule el funcionamiento, siguiendo las siguientes reglas:

- Si cuando llega un cliente no hay ningún carrito disponible, espera a que lo haya.
- Ningún cliente se impacienta y abandona el supermercado sin pasar por alguna de las colas de las cajas.
- Cuando un cliente finaliza su compra, se coloca en la cola de la caja que hay menos gente, y no se cambia de cola.
- En el momento en que un cliente paga en la caja, el carro de la compra que tiene queda disponible.

Representar la lista de carritos de la compra y las cajas de salida mediante colas.

11.6. En un archivo F están almacenados números enteros arbitrariamente grandes. La disposición es tal que hay un número entero por cada línea de F. Escribir un programa que muestre por pantalla la suma de todos los números enteros. Al resolver el problema habrá que tener en cuenta que al ser enteros grandes no pueden almacenarse en variables numéricas.

Utilizar dos pilas para guardar los dos primeros números enteros, almacenándose dígito a dígito. Al extraer los elementos de la pila salen en orden inverso y, por tanto, de menos peso a mayor peso, se suman dígito a dígito y el resultado se guarda en una cola, también dígito a dígito. A partir de este primer paso, se obtiene el siguiente número del archivo, se guarda en una pila y a continuación se suma dígito a dígito con el número que se encuentra en la cola; el resultado se guarda en otra cola. El proceso se repite: nuevo número del archivo se mete en la pila, que se suma con el número actual de la cola.

11.7. Con un archivo de texto se desea realizar esta 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 aquellas que empiezan por la letra *a*, luego las de la letra *b*, y así sucesivamente.

11.8. 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:

- 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.
- 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.

11.9. El supermercado «Esperanza» quiere simular los tiempos de atención al cliente a la hora de pasar por la caja; los supuestos de que se parte para la simulación son los siguientes:

- Los clientes forman una única fila. Si alguna caja está libre, el primer cliente de la fila es atendido. En el caso de que haya mas de un caja libre, la elección del número de caja por parte del cliente es aleatoria.
- El número de cajas de que se dispone para atención a los clientes es tres, salvo que haya mas de 20 personas esperando en la fila, entonces se habilita una cuarta caja, que cierra cuando no quedan clientes esperando. El tiempo de atención de cada una de las cajas está distribuido uniformemente, la caja 1 entre 1,5 y 2,5 minutos; la caja 2 entre 2 y 5 minutos, la caja 3 entre 2 y 4 minutos. La caja 4, cuando está abierta, tiene un tiempo de atención entre 2 y 4,5 minutos.
- Los clientes llegan a la salida en intervalos de tiempo distribuidos uniformemente, con un tiempo medio de 1 minuto.

Hacer un programa de simulación para siete horas de trabajo. Se desea obtener una estadística con los siguientes datos:

- Clientes atendidos durante la simulación.
- Tamaño medio de la fila de clientes.
- Tamaño máximo de la fila de clientes.
- Tiempo máximo de espera de los clientes.
- Tiempo en que está abierto la cuarta caja.

11.10. La universidad de La Alcarria dispone de 15 computadores conectados a Internet. Se quiere hacer una simulación de la utilización de los computadores por los alumnos. Para ello se supone que la frecuencia de llegada de un alumno es de 18 minutos las dos primeras horas y de 15 minutos el resto del día. El tiempo de utilización del computador es un valor aleatorio, entre 30 y 55 minutos.

El programa debe tener como salida líneas en las que se refleje 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 computadores libres el alumno no espera, se mostrará el correspondiente aviso. En una cola de prioridad han de 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.

11.11. La entrada a una sala de arte que ha inaugurado una gran exposición sobre la evolución del arte rural, se realiza por tres torniquetes. Las personas que quieren ver la exposición forman un única fila y llegan de acuerdo a una distribución exponencial, con un tiempo medio entre llegadas de 2 minutos. Una persona que llega a la fila y ve más de 10 personas esperando se va con una probabilidad del 20 por 100, aumentando en 10 puntos por cada 15 personas mas que haya esperando, hasta un tope del 50 por 100. El tiempo medio que tarda una persona en pasar es de 1 minuto (compra de la entrada y revisión de los bolsos). Además, cada visitante emplea en recorrer la exposición entre 15 y 25 minutos distribuidos uniformemente.

La sala sólo 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.
- Tiempo medio que debe esperar una persona para entrar en la sala.

12

COLAS DE PRIORIDADES Y MONTÍCULOS

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Conocer el tipo de dato *Cola de prioridad.*
- Describir las operaciones fundamentales de las colas de prioridades.
- Implementar colas de prioridades con listas enlazadas.
- Conocer la estructura de datos montículo binario.
- Implementar las operaciones básicas de los montículos con *complejidad logarítmica.*
- Ordenar una lista de elementos con el *método del montículo.*

CONTENIDO

12.1. Cola de prioridades.

12.2. Construcción de una cola de prioridades.

12.3. Montículos.

12.4. Ordenación por montículos (*HeapSort*).

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Prioridad.
- Árbol binario.
- Listas enlazadas.
- Cola.
- Montículo.
- Complejidad.
- Ordenación.
- Nodo *padre*, nodo *hijo*.

INTRODUCCIÓN

En este capítulo se estudia pormenorizadamente la estructura de datos *Colas de prioridades*, una estructura utilizada para planificar tareas en aplicaciones informáticas donde la prioridad de la tarea se corresponde con la clave de ordenación. También se aplican en los sistemas de simulación donde los sucesos se deben procesar en orden cronológico. El común denominador de estas aplicaciones es que siempre se selecciona el *elemento mínimo* una y otra vez hasta que no quedan más elementos. Tradicionalmente se ha designado la prioridad más alta con una clave menor, así los elementos de prioridad 1 tienen más prioridad que los elementos de prioridad 3.

Se pueden usar listas enlazadas para implementar las colas de prioridades, o bien una estructura de multicolas. El montículo binario, simplemente montículo, es la estructura más apropiada para implementar eficientemente las colas de prioridades.

La *idea intuitiva* de un montículo es la que mejor explica esta estructura de datos. En la *parte más alta* del montículo se escuentra el *elemento más pequeño;* los elementos descendientes de uno dado son mayores en la estructura montículo. La estructura montículo garantiza que el tiempo de las operaciones básicas insertar y elimar mínimo, sea de complejidad logarítmica.

12.1. COLA DE PRIORIDADES

El término *cola* sugiere la forma en que ciertos objetos esperan la utilización de un determinado servicio. Por otro lado, el término *prioridad* sugiere que el servicio no se proporciona únicamente aplicando el concepto «primero en llegar, primero en ser atendido» sino que cada objeto tiene asociado una prioridad basada en criterio objetivo. Un aplicación típica de organización formando colas de prioridades es el sistema de tiempo compartido, necesario para mantener una serie de procesos que esperan para que sean ejecutados por el procesador. Cada proceso está diseñado de tal forma que lleva asociado una prioridad.

El orden en que los elementos de una cola de prioridades son procesados sigue estas reglas:

- Se elige la cola no vacía que se corresponde con la mayor prioridad.
- En la cola de mayor prioridad los elementos se procesan según el orden de llegada: *primero en entrar, primero en salir.*

Las simulaciones de sucesos que ocurren en un tiempo discreto, como la atención de una fila de clientes en un sistema de *n* ventanillas de despacho de billetes de transporte, se realiza con una estructura de colas de prioridades.

12.1.1. Declaración del *TAD Cola de prioridad*

La estructura Cola de prioridad se caracteriza porque repetidamente se puede seleccionar del conjunto de elementos (procesos) el de clave máxima, o prioridad máxima. Normalmente se insertan nuevos elementos en la estructura, a la vez que se procesan otros (los de máxima prioridad). Esto lleva a especificar las operaciones que se detallan a continuación.

Tipo de dato: *Proceso* que tiene asociado una *prioridad.*

Operaciones:

CrearColaPrioridad	Inicia la estructura con las prioridades sin elementos.
InserEnPrioridad	Añade un elemento a la cola según prioridad.
ElementoMin	Devuelve el elemento de la cola con la prioridad más alta.
QuitarMin	Devuelve y retira el elemento de la cola con prioridad más alta.
Cola vacía	Comprobar si una determinada cola no tiene elementos
Cola Prioridad vacía	Comprueba si todas las colas de la estructura están vacías.

A tener en cuenta

En programación siempre se ha seguido la convención de considerar un intervalo de máxima a mínima prioridad con valores clave en orden inverso; por ejemplo, asignar a la máxima prioridad la clave 0, a la siguiente la clave 1, y así sucesivamente de tal forma que la clave mínima sea n.

12.2. CONSTRUCCIÓN DE UNA COLA DE PRIORIDADES

Una forma simple de definir una cola de prioridades es mediante una tabla o un array de tantos elementos como prioridades existan; cada elemento de la tabla se organiza a la manera de una cola (*primero en entrar, primero en salir*). La Figura 12.1 muestra una una tabla en la que cada elemento se corresponde con una prioridad y de la que emerge una cola.

La operación `inserEnPrioridad` añade un nuevo elemento `T` de prioridad `m` a la estructura, siguiendo estos pasos:

1. Buscar la prioridad `m` en la tabla.
2. Si existe, poner el elemento `T` al final de la cola `m`.
3. Si la prioridad `m` está fuera del rango de prioridades, genera un error.

La operación `elementoMin` devuelve el elemento `frente` de la cola no vacía que tiene la máxima prioridad. La operación `quitarMin` también devuelve el elemento de máxima prioridad y además lo extrae de la cola.

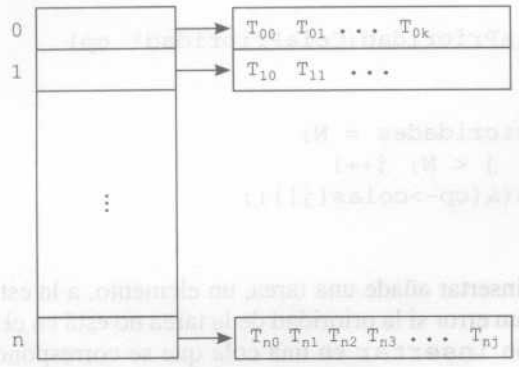


Figura 12.1. Cola de prioridades, de *0* a *n* prioridades.

12.2.1. Definición del *TAD Cola de prioridades*

En el archivo `colaprioridad.h` se declaran los tipos de datos necesarios para representar una tabla o vector con tantos elementos como prioridades, se supone de *0* a *n*. Cada elemento de la tabla es una cola, a su vez representada con variables puntero a los extremos `frente` y `final`. Por ello se incluye el archivo `coladinamica.h` para poder aplicar las operaciones y hacer uso de los tipos de datos ya declarados.

***Archivo* `colaprioridad.h`**

```
#define N 12

typedef struct
{
  int prioridad;
  char nombre[51];
}Tarea;

typedef Tarea TipoDato;

#include "coladinamica.h"

typedef struct
{
  int numprioridades;
  Cola colas[N];
} ColaPrioridad;
```

```
void crearColaPrioridad(ColaPrioridad* cp);
void inserEnPrioridad(ColaPrioridad* cp, Tarea t);
Tarea elementoMin(ColaPrioridad cp);
Tarea quitarMin(ColaPrioridad* cp);
int colaPrioridadVacia(ColaPrioridad cp);
```

En la creación de la cola de prioridades se supone que `N` es la prioridad más baja y que la prioridad más alta es la de índice cero. También se asume que hay una correspondencia biunívoca entre el índice de la tabla de colas y el ordinal de la prioridad. Esta operación crea una cola vacía, inicializa, para cada una de las prioridades.

```
void crearColaPrioridad(ColaPrioridad* cp)
{
  int j;
  cp -> numprioridades = N;
  for (j = 0; j < N; j++)
    crearCola(&(cp->colas[j]));
}
```

La operación de insertar añade una tarea, un elemento, a la estructura en la prioridad que tiene asignada. Se comete un error si la prioridad de la tarea no está en el rango permitido. Todo el trabajo lo realiza la operación `insertar` en una cola que se corresponde con la prioridad.

```
void inserEnPrioridad(ColaPrioridad* cp, Tarea t)
{
  if (t.prioridad < cp->numprioridades)
  {
    insertar(&cp->colas[t.prioridad],t);
  }
  else
    puts("Tarea con prioridad fuera de rango");
}
```

Las operaciones `elementoMin` y `quitarMin` buscan el elemento de máxima prioridad; es una búsqueda de un elemento mínimo por el convenio establecido de que la máxima prioridad se corresponde con la clave mínima. Se considera un error llamar a alguna de estas operaciones si la estructura está vacía. Una vez encontrado el índice de la cola con mayor prioridad, se llama a `quitar`, del tipo `Cola`, para terminar de realizar la operación.

```
Tarea quitarMin(ColaPrioridad* cp)
{
  int i = 0;
  int indiceCola ;
  indiceCola = -1;
    /* búsqueda de la primera cola no vacía */
  do {
    if (!colaVacia(cp->colas[i]))
    {
      indiceCola = i;
      i = cp->numprioridades +1;          /* termina el bucle */
    }
    else
```

```
      i++;
    }while (i < cp->numprioridades);

    if (indiceCola != -1)
      return quitar(&cp->colas[indiceCola]);
    else
    {
      puts("Cola de prioridades vacía");
      exit(-1);
    }
  }
```

La operación `elementoMin` sigue los mismos pasos que `quitarMin`, con la diferencia de que no extrae el elemento de la cóla, simplemente devuelve el elemento frente de la cola de mayor prioridad:

```
return frente(cp->colas[indiceCola]);
```

Por último, la operación `colaPrioridadVacia` comprueba que cada una de las colas está vacía.

```
int colaPrioridadVacia(ColaPrioridad cp)
{
  int i = 0;

  while (colaVacia(cp.colas[i]) && i<cp.numprioridades-1)
    i++;
  return colaVacia(cp.colas[i]);
}
```

12.3. MONTÍCULOS

El *montículo binario*, o simplemente **montículo,** es una estructura abstracta que almacena los datos de tal forma que pueden recuperarse en un orden. Por ello se utiliza para implementar las colas de prioridades en las cuales el orden de proceso es esencial, también para uno de los algoritmos de ordenación más eficientes: *ordenación por montículos* o *HeapSort*. Además de la propiedad del orden, los montículos se caracterizan por la organización de los datos en forma de un vector.

La idea intuitiva de montículo es muy conocida: *una agrupación en forma piramidal de elementos en la que para cualquier nivel el peso de éstos es menor que la de los elementos adjuntos del nivel inferior, y por consiguiente, en la parte más alta se encuentra el elemento más pequeño.*

12.3.1. Definición de montículo

Un montículo binario *de tamaño* `n` *se define como un árbol binario*[1] *casi completo de* `n` *nodos, tal que el contenido de cada nodo es menor o igual que el contenido de su hijos.*

[1] Aunque el estudio de los árboles binarios se realiza en el Capítulo 14, se adelanta su concepto, por coherencia con la temática de este capítulo. Ante cualquier duda, por favor, consulte el citado Capítulo 14.

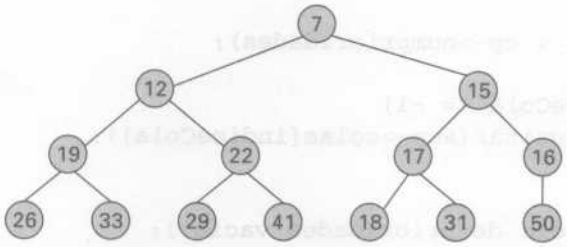


Figura 12.2. Montículo de elementos enteros.

La definición formal de la estructura montículo está relacionada con los árboles binarios. Así, en la Figura 12.2 se puede observar un árbol binario con todos los niveles completos, excepto el último, que puede no estar completo y tener los nodos situados lo más a la izquierda posible, que forma un montículo binario.

Sin embargo, el árbol binario de la Figura 12.3 no es completo, ya que en el último nivel no se han colocado los nodos de izquierda a derecha. Tampoco cumple la propiedad de ordenación de los montículos, el nodo con clave 21 es mayor que uno de sus hijos, con clave 19.

Definición

Un *árbol binario completo*[2] es un árbol con todos los niveles llenos, con la excepción del último nivel, que se llena de izquierda a derecha. Esta estructura tiene la propiedad importante de que la altura de un árbol binario completo de `n` nodos es `[log n]`.

Los montículos tienen dos propiedades fundamentales: la propiedad de organizarse como un árbol binario y la propiedad de la ordenación. La primera asegura complejidades en las operaciones con cotas logarítmicas, y además una fácil representación en una estructura tipo vector o array.

La propiedad de ordenación: la clave de cualquier nodo es inferior o igual a la de sus hijos (si tiene hijos). Por consiguiente, el nodo con menor clave debe ser el nodo raíz del árbol, siempre el nodo menor es el raíz por lo que la operación de *buscar mínimo* se puede implementar con *complejidad constante, O(1).*

Las operaciones definidas con la estructura montículo puede violar alguna de las propiedades de éste, por lo que será necesario restablecer la condición de montículo.

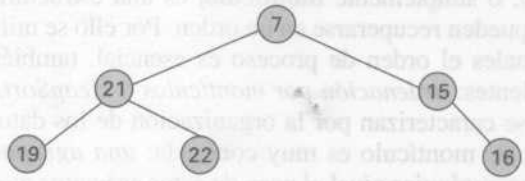


Figura 12.3. Árbol binario no completo.

A tener en cuenta

En un montículo la clave que se encuentra en la raíz es la menor de todos los elementos; sin embargo, no existe un orden entre las claves de los elementos que se encuentran en el mismo nivel.

[2] Vease la teoría completa de árboles vinarios en Capítulo 14.

12.3.2. Representación de un montículo

La característica de que no faltan nodos en un árbol binario completo permite que las claves de los nodos se representen secuencialmente con un vector o array. La raíz del árbol se sitúa en la posición 1, sus hijos en las posiciones 2 y 3; los hijos de éstos en las posiciones 4, 5, 6 y 7 respectivamente, y así sucesivamente. El montículo de la Figura 12.4 consta de 8 nodos que han sido numerados según las posiciones que ocuparán en un almacenamiento secuencial.

Esta disposición natural de las claves de un árbol completo (las posiciones se corresponde con los niveles del árbol, para un nivel de izquierda a derecha) en un array es muy útil ya que permite acceder desde un nodo al nodo padre, al hijo izquierdo y derecho (si los tiene). Un nodo que esté en la posición `i` su nodo padre ocupa la posición `[i/2]`, el nodo hijo izquierdo se ubica en la posición `2*i` y el nodo hijo derecho en `2*i+1`. La representación secuencial del montículo binario de la Figura 12.4:

15	39	18	41	72	37	19	56

A tener en cuenta

La forma secuencial de un montículo de `n` elementos implica que si `2*i` es mayor que `n` entonces el nodo `i` no tiene hijo izquierdo (tampoco hijo derecho), y si `2*i+1` es mayor entonces el nodo `i` no tiene hijo derecho.

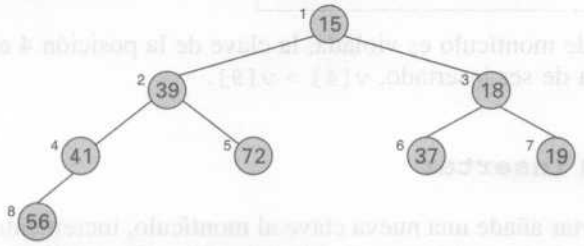


Figura 12.4. Montículo de 8 elementos numerando sus posiciones secuenciales.

12.3.3. Propiedad de ordenación: condición de montículo

La ordenación parcial de los nodos de un montículo se conoce como *condición de montículo*, que permite encontrar con una sola operación el elemento mínimo. Según la representación secuencial, el elemento mínimo siempre se almacena en la primera posición, `v[1]`. Esto hace que el montículo sea una estructura idónea para representar al tipo *cola de prioridades*, ya que el elemento a extraer de la cola siempre estará en la posición 1 del array.

Definición

La *condición de montículo* establece que la clave de cada nodo debe ser menor o igual que las claves de sus hijos. Esta condición deben satisfacerla todos los nodos del montículo, por lo que cada nodo debe ser menor que todos sus descendientes.

Utilizando la representación secuencial de los montículos la propiedad de ordenación parcial de las claves se expresa:

```
v[i] ≤ v[2*i]
                    ∀i = 1 .. n/2
v[i] ≤ v[2*i+1]
```

El montículo de la Figura 12.2 se representa secuencialmente:

7	12	15	19	22	17	16	26	33	29	41	18	31	50

Cualquier clave cumple la condición; así `v[5] == 22`, tiene como nodos hijos a `v[10]` y `v[11]` cuyas claves respectivas, 29 y 41 son menores que 22.

12.3.4. Operaciones definidas en un montículo

Se va a suponer que se trabaja con montículos de `n` claves (de tipo entero) almacenadas secuencialmente, en un array. Las operaciones básicas `insertar`, `buscarMinimo`, `eliminarMinimo` (además de `crearMontículo` y `esVacio`) hacen una simple modificación en el montículo, pero esta origina, posiblemente, una violación de la *condición de montículo*. Entonces se necesita de operaciones auxiliares para recomponer el montículo, que recorren la estructura de arriba abajo (de la raíz al último nivel), o en orden inverso.

Así el montículo de la Figura 12.4 consta de 8 claves, la inserción de la clave `32` se debe hacer en la posición 9, `v[9] == 32`:

15	39	18	41	72	37	19	56	**32**

La condición de montículo es violada, la clave de la posición 4 es mayor que la clave del hijo derecho que acaba de ser insertado, `v[4] > v[9]`.

12.3.5. Operación `insertar`

La operación insertar añade una nueva clave al montículo, incrementa su tamaño en uno por lo que se debe incrementar `n` y en el *hueco* se coloca la clave. De esta forma no se viola la estructura del montículo. En el caso de que al situar la nueva clave no se viole la condición de ordenación el proceso ha terminado. Entonces, se ha de comprobar que la clave del nodo *padre,* del nuevo nodo, sea menor o igual que la nueva clave insertada; en caso negativo el padre baja al hueco. En cada paso la clave situada en el padre se compara con la nueva clave, y en caso de que esta sea menor que la del padre, éste baja al hueco; en definitiva, se *sube por el árbol* para encontrar la posición que le corresponde a la clave que se está insertando. El proceso de búsqueda de la posición de inserción termina cuando la clave del nodo padre es menor o igual (no se viola la condición de montículo), o bien se alcanza la raíz; a continuación se asigna la clave a la posición.

El montículo binario de la Figura 12.5 consta de 8 claves. Ahora se quiere añadir la clave 23, para lo cual se hace en primer lugar un hueco (posición 9 del array) en el árbol.

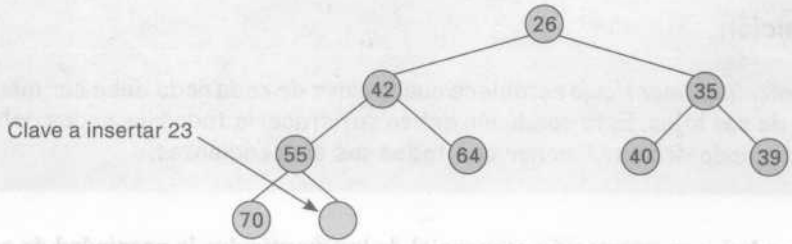


Figura 12.5. Montículo al que se añade la clave 23.

El nodo padre, 55, es mayor que 23, por lo que se viola la condición de ordenación. Baja la clave 55 al hueco y se sube al siguiente nivel del árbol en busca de la posición de inserción. A continuación se compara la clave 42 (nuevo nodo padre) con 23, como es mayor baja 42 al hueco dejado anteriormente por 55, se sube por el árbol al siguiente nivel, se compara 26 con 23 y al ser mayor 26 baja al hueco y el proceso termina pues se ha alcanzado la raíz del árbol. La nueva raíz es 23.

La Figura 12.6 muestra las comparaciones que se realizan hasta encontrar la posición donde insertar la clave 23; en el ejemplo la nueva clave se posiona en la raíz del árbol.

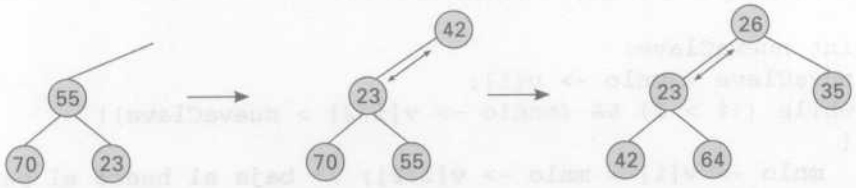


Figura 12.6. Reconstrucción del montículo moviendo hacia arriba la nueva clave.

El número máximo de pasos *que se realizan al insertar una clave en el montículo ocurre cuando la clave es el nuevo valor mínimo. Éste coincide con el número de niveles del árbol (para un montículo de* `n` nodos, `[log n]`*); entoces el tiempo necesario, en el peor de los casos, para realizar la inserción es O(*`[log n]`*).*

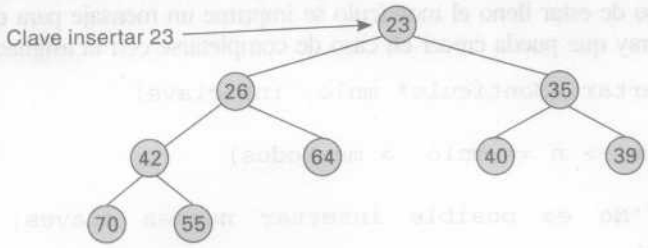


Figura 12.7. Montículo después de insertar 23.

A recordar

La inserción de una clave en el montículo se realiza en la siguiente posición libre del array, para hacer a continuación que *flote hacia arriba*, hasta encontrar la posición adecuada en el árbol. La complejidad de la operación es logarítmica, O(`[log n]`).

Implementación

Las claves de un montículo pueden ser de cualquier tipo de dato al que se pueda aplicar las operaciones de *comparación* (menor, igual, ...). En la codificación que se escribe se asume claves de tipo entero; así resulta más fácil y simple la comparación entre ellas.

Primero se declara *un tipo montículo*, en el se encierra el array que almacena secuencialmente al montículo, el máximo de nodos o claves previsto y el número actual.

```
#define MAXNODOS 1001
typedef struct
{
  int v[MAXNODOS+1];
  int maxNodos;
  int n;
} Monticulo;
```

En la función `flotar()` se realiza el *movimiento hacia arriba* en busca de la posición de inserción. A la función se le pasa el índice, `i`, correspondiente al hueco donde inicialmente se encuentra la nueva clave. Las comparaciones entre la nueva clave y la del nodo padre se realizan eficientemente ya que el índice del nodo *padre*, según la propiedad estructural, es `i/2` siendo `i` el índice del *nodo hijo* (hueco donde se insertará). El bucle en el que se realiza el proceso termina cuando la clave del nodo padre es menor o igual (`v[i/2] <= nuevaClave`), o bien se *ha subido* hasta la raíz (`i == 1`).

```
void flotar(Montículo* mnlo, int i)
{
  int nuevaClave;
  nuvaClave = mnlo -> v[i];
  while ((i > 1) && (mnnlo -> v[i/2] > nuevaClave))
  {
    mnlo -> v[i] = mnlo -> v[i/2]; /* baja al hueco el padre */
    i = i/2;                       /* sube un nivel en el árbol */
  }
  mnlo -> v[i] = nuvaClave; /* sitúa la clave en su posición */
}
```

La función `insertar` comprueba que no esté lleno el montículo, incrementa el número de claves, hace un hueco para la clave, y llama a *flotar* para que restablezca la condición de ordenación del montículo. En el caso de estar lleno el montículo se imprime un mensaje para que éste se amplíe; se puede considerar un array que pueda crecer en caso de completarse con la asignación dinámica de memoria.

```
void insertar (Montículo* mnlo, int clave)
{
  if (mnlo -> n == mnlo -> maxNodos)
  {
    puts("No es posible insertar nuevas claves: montículo lleno");
    return;
  }
  (mnlo -> n)++;
  flotar(mnlo, mnlo->n);
}
```

Aunque no es estrictamente necesario, se codifica también la operación `crearMonticulo` que inicializa el montículo vacío; simplemente se pone el numero de claves, `n`, a cero.

```
void crearMonticulo(Montículo* mnlo)
{
  mnlo -> n = 0;
  mnlo -> maxNodos = MAXNODOS;
}
```

12.3.6. Operación `buscar mínimo`

La propiedad de ordenación parcial de los montículos asegura que el el elemento mínimo se encuentra en la raíz del árbol, que en el almacenamiento secuencial se corresponde con la primera posición. Por ello la función que implementa esta operación tiene complejidad constante *O(1)*.

```
int buscarMinimo(Monticulo mnlo)
{
  if (!esVacio(mnlo))
  return mnlo -> v[1];
}
```

12.3.7. Operación `eliminar mínimo`

El elemento o clave mínima, como ya se ha comentado anteriormente, se encuentra en la raíz del árbol que representa al montículo. La operación implica eliminar la clave que se encuentra en la raíz y en consecuencia reducir el número de claves, `n-1`. Ahora bien, la estructura resultante debe seguir siendo un montículo, manteniendo la ordenación parcial de las claves y la forma estructural.

La operación extrae el elemento raíz, elemento `v[1]` del array, se crea un *hueco* en esa posición. Al tener el montículo una clave menos se reemplaza el último elemento (hoja del árbol más a la derecha) por la raíz y así se restablece la estructura del montículo. La Figura 12.8 muestra este primer paso en la eliminación del elemento mínimo.

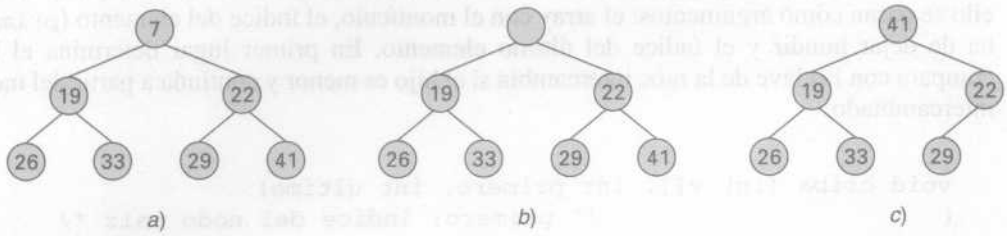


Figura 12.8. *a*) Montículo al que se elimina el mínimo (7); *b*) *hueco* dejado, *c*) sube el último elemento (41).

Una vez asignado el último nodo al hueco dejado en la raíz, queda por analizar si la ordenación parcial, condición de montículo, se mantiene. Esto es difícil que ocurra ya que los últimos nodos son mayores que los del anterior nivel, por tanto, hay que restablecer la condición de ordenación. Para lo cual se procede a la inversa de la inserción: se deja *hundir la clave por el camino de claves mínimas*.

El algoritmo a seguir compara la raíz con el menor de los hijos, si la raíz es mayor se intercambia con el menor; así en el montículo 12.8*c* la clave 41 se intercambia por 19. Esto puede hacer que de nuevo se viole la condición de ordenación del montículo pero en el siguiente nivel. Entoces se procede de igual manera, se elige el menor de los hijos, se compara con la clave y si ésta es mayor se intercambia con el menor; en el ejemplo el menor de los hijos es 26, por lo que se intercambia por 41. Se observa que la clave que inicialmente subió a la raíz baja por el camino de las claves mínimas; el proceso continúa hasta que no se viole la condición de montículo, o bien se llege al último nivel.

La Figura 12.9 muestra los intercambios que se realizan en el montículo 12.8*c* hasta situar la clave 41 en una posición tal que que se cumpla la condición de montículo, en este caso como nodo hoja.

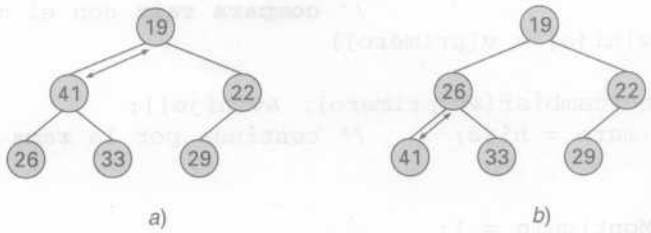


Figura 12.9. Proceso que restablece la condición de ordenación parcial del montículo. *a*) Intercambio de 41 con 19; *b*) intercambio de 41 con 26.

A recordar

El procedimiento de hundir la clave situada en la raíz hasta encontrar la posición en la que no se viole la condición de montículo, realiza un máximo de intercambios igual al número de niveles menos uno, por lo que la complejidad que tiene es *logarítmica*, `O([log n])`.

Implementación

La función `criba()` implementa el algoritmo que restablece la condición de montículo; en definitiva, las acciones para que la clave raíz se hunda en el árbol por el camino de claves mínimas. Para ello se pasan como argumentos: el array con el montículo, el índice del elemento (`primero`) que se ha de dejar hundir y el índice del último elemento. En primer lugar determina el hijo menor, compara con la clave de la raíz, intercambia si el hijo es menor y continúa a partir del índice del hijo intercambiado.

```
void criba (int v[], int primero, int ultimo)
{                          /* primero: indice del nodo raiz */
  int esMonticulo;
  int hijo;

  esMonticulo = 0;
  while ((primero <= ultimo / 2) && !esMonticulo)
  {              /* primera condición expresa que no sea un nodo hoja */
                                /* determina el índice del hijo menor */
    if (2*primero == ultimo)          /* tiene un único descendiente */
      hijo = 2*primero;
    else
    {
      if (v[2*primero] < v[2*primero+1])
        hijo = 2*primero;
      else
        hijo = 2*primero+1;
    }
                                /* compara raiz con el menor de los hijos */
    if (v[hijo] < v[primero])
    {
      intercambiar(&v[primero], &v[hijo]);
      primero = hijo;       /* continua por la rama de claves mínimas */
    }
    else
      esMonticulo = 1;
  }
}
```

La función `eliminarMinimo()` comprueba la condición de montículo vacío, extrae la clave raíz y llama a `criba()`, que realiza todo el trabajo de restablecer la condición de ordenación.

```
int eliminarMinimo(Monticulo* mnlo)
{
  if (!esVacio(mnlo))
  {
    int menor = buscarMinimo(mnlo);
    mnlo -> v[1] = mnlo -> v[mnlo->n];
    (mnlo -> n)--;

    criba(mnlo -> v, 1, mnlo -> n);
    return menor;
  }
}
```

12.4. ORDENACIÓN POR MONTÍCULOS (*HeapSort*)

Una de las aplicaciones que se ha dado a la estructura del montículo es la de ordenar los `n` elementos de un vector. Haciendo uso de las operaciones definidas anteriormente se puede diseñar un algoritmo de ordenación. Los pasos a seguir:

1. Crear el montículo vacío `v[]`.
2. Insertar cada uno de los n elementos en el montículo. Se utiliza la operación `insertar`.
3. Extraer cada uno de los elementos llamando a la operación `eliminarMinimo` y asignado el elemento extraído en el vector `w[]`.

Un problema de este algoritmo es que es necesario un segundo vector en el que almacenar los elementos en el orden de extracción. Con el fin de realizar la ordenación en el mismo vector, si necesidad de memoria adicional se sigue una estrategia que se conoce como algoritmo de *ordenación HeapSort*, o simplemente *ordenación por montículo*.

En la operación `eliminarMinimo` el montículo se queda con un elemento menos (el mínimo), entonces el algoritmo *heapsort* evita el uso de un segundo vector utilizando, en cada pasada, la actual última posición para guardar el elemento eliminado. La Figura 12.10*b* muestra el montículo después de eliminar la clave mínima y ser guardada en la última posición del array; la operación ha reconstruido el montículo por lo que el nuevo mínimo se corresponde con la clave 22. En 12.10*c* se ha aplicado de nuevo la operación, el mínimo se guarda en la actual última posición, en el ejemplo posición 5; la operación reconstruye el montículo. Con cuatro pasadas más el array queda ordenado, aunque en orden decreciente.

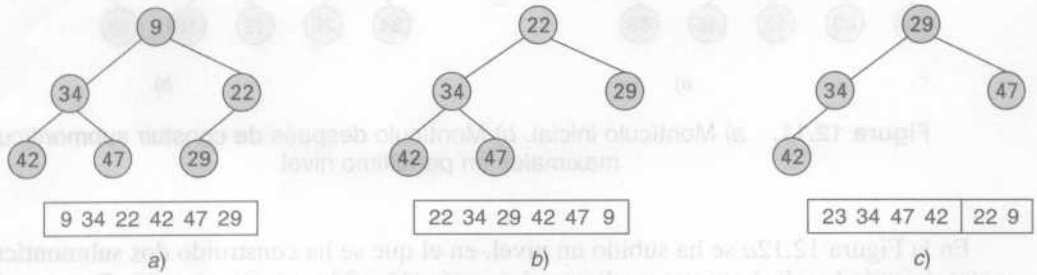


Figura 12.10. *a*) Montículo de 6 elementos. *b*) Montículo después de eliminar mínimo. *c*) Después de volver aplicar eliminar mínimo.

Los algoritmos de ordenación interna siempre ordenan a los elementos en orden creciente. Para que el algoritmo del montículo lo haga así simplemente hay que usar el montículo maximal, cambiando la condición de ordenación de tal forma que la clave del nodo padre sea mayor o igual que la de sus hijos.

A tener en cuenta

El método de ordenación por montículos ordena *descendentemente* ya que siempre intercambia el elemento menor, `v[1]`, con el último elemento del montículo actual. Para una ordenación ascendente se considera *un montículo maximal*, simplemente se invierte el sentido de las desigualdades (cambiar < por > y viceversa).

12.4.1. Algoritmo

El algoritmo arranca con un vector de `n` elementos que no cumple la condición de montículo. Entoces lo primero que tiene que hacer el algoritmo es construir el montículo inicial. La solución más sencilla consiste en considerar el montículo como una estructura recursiva. Los nodos del montículo del último nivel del árbol se pueden considerar que cada uno es un submontículo de 1 nodo. Subiendo un nivel en el árbol, 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 arriba*), entonces al aplicar la función `criba()` (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.

La Figura 12.11*a* se corresponde con el array inicial en la forma de árbol completo. En la Figura 12.11*b* todos los subárboles del penúltimo nivel son *montículos maximales*; se pueden observar los intercambios realizados para que cumplan la propiedad.

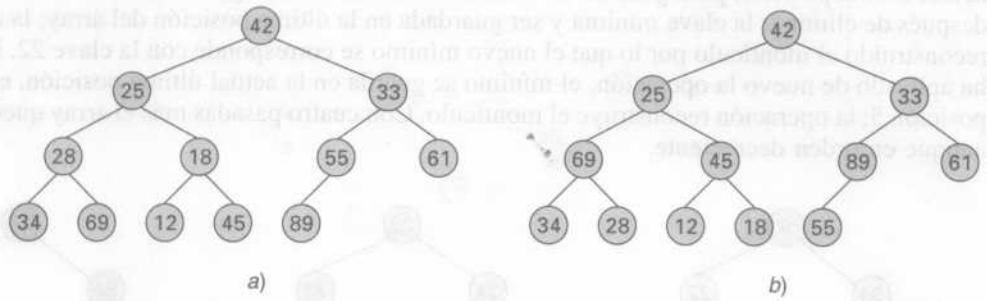


Figura 12.11. *a*) Montículo inicial. *b*) Montículo después de constuir submontículos maximales en penúltimo nivel.

En la Figura 12.12*a* se ha subido un nivel, en el que se ha construido dos submontículos maximales dejando hundir la correspondiente clave raíz (25 y 33 respectivamente). Para terminar con el montículo de todos los nodos en la Figura 12.12*b*, en esta última reconstrución se ha hundido la clave 42 y ha subido al raíz la clave 89, que es el valor mayor.

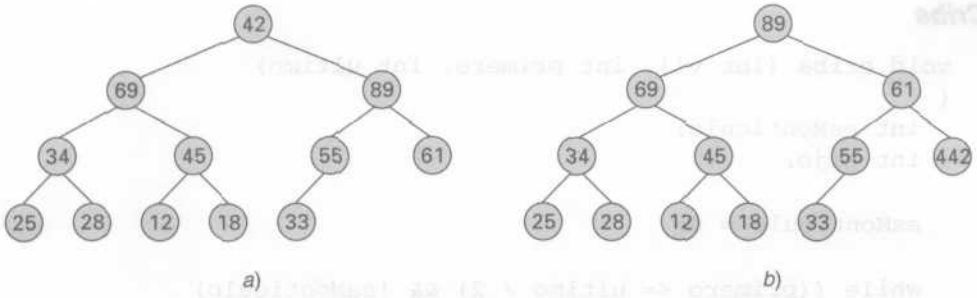


Figura 12.12. *a)* Reconstrucción de montículo en segundo nivel.
b) Montículo maximal reconstruido de abajo arriba.

Según esto podemos descomponer el método de ordenación por montículos o *HeapSort* en los siguientes pasos:

1. *Construir un montículo inicial con todos los elementos del vector:* `v[1], v[2], ...v[n]`.
2. Intercambiar los valores de `v[1]` y `v[n]`.
3. Reconstruir el montículo con los elementos `v[1], v[2], ...v[n-1]`.
4. Intercambiar los valores de `v[1]` y `v[n-1]`.
5. *Reconstruir el montículo con los elementos* `v[1], v[2], ...v[n-2]`.

Es un un proceso iterativo que, partiendo de un montículo inicial, repite *intercambiar* los extremos, decrementar en 1 la posición del extremo superior y reconstruir el montículo del nuevo vector. En forma algorítmica:

```
OrdenacionMonticulo(vector, n)

inicio

  <Construir monticulo inicial (vector, 1, n)>

  desde k ← n hasta 2 hacer
    intercambiar (vector[1], vector[k])
    reconstruir monticulo (vector, 1, k-1)
  fin_desde

fin OrdenacionMonticulo
```

El problema de construir el montículo inicial y el de reconstruir se resuelve con el procedimiento `criba()` que constituye la base del algoritmo.

12.4.2. Codificación

La implementación se realiza considerando el montículo *maximal* para que de esa forma los elementos queden en orden ascendente. En esta codificación no se ha tenido en cuenta el índice 0 de los arrays para no introducir una complejidad adicional. Si se quiere considerar el índice 0, hay que tener en cuenta que las posiciones de los hijos se obtienen: índice `2*i+1` para el nodo hijo izquierdo, `2*i+2` para el hijo derecho.

Aunque ya se escribió anteriormente la función `criba()`, se codifica de nuevo teniendo en cuenta que se trabaja con un montículo maximal, por tanto se deja *hundir* la clave por el camino de claves máximas y asciende la clave mayor.

Criba

```
void criba (int v[], int primero, int ultimo)
{
  int esMonticulo;
  int hijo;

  esMonticulo = 0;

  while ((primero <= ultimo / 2) && !esMonticulo)
  {
    if (2*primero == ultimo)          /* tiene un único descendiente */
      hijo = 2*primero;
    else
    {                                 /* selecciona el mayor de los dos hijos */
      if (v[2*primero] > v[2*primero+1])
        hijo = 2*primero;
      else
        hijo = 2*primero+1;
    }

    if (v[primero] < v[hijo])
    {
      intercambiar(&v[primero], &v[hijo]);
      primero = hijo;             /* continua por rama de claves máximas */
    }
    else
      esMonticulo = 1;
  }
}
```

Ordenación

Para construir el montículo inicial se llama a `criba()` pasando como segundo argumento la raíz de los subárboles desde el penúltimo nivel del árbol hasta la raíz; en definitiva: `criba(v, j, n)` para todo `j = n/2, n/2 - 1, ..., 1`. Por tanto, el bucle que construye el montículo:

```
for (j = n / 2; j >= 1; j--)
  criba (v, j, n);
```

De esta forma en `criba()` reside todo el trabajo de la realización del método de ordenación *HeapSort*. Por último, la codificación de la función de ordenación:

```
void ordenacionMonticulo(int v[], int n)
{
  int j;

  for (j = n / 2; j >= 1; j--)
    criba (v, j, n);

  for (j = n; j >= 2; j--)
  {
```

```
        intercambiar(&v[1], &v[j]);
        criba (v, 1, j-1);
    }
}
```

12.4.3. Análisis del algoritmo de ordenación por montículos

El análisis de la eficiencia del algoritmo de ordenación *HeapSort* depende de la complejidad que tiene la función `criba()`. Supóngase que el árbol binario que representa al montículo está completo, y que el número de niveles de que consta es k. Además, el nivel que se corresponde con el elemento raíz es el cero, por lo que los niveles de que consta: $0, 1, 2, 3, \ldots k-1$.

El número de elementos de cada nivel es potencia de 2: nivel 0, $2^0 = 1$; nivel 1, $2^1 = 2$, $2^2 = 4$, $2^3 = 4, \ldots 2^{k-1}$. *Por lo que el total de elementos* $= 1 + 2 + 2^2 + 2^3 + \cdots + 2^{k-1}$.

Si n es el número de elementos a ordenar, aplicando la fórmula de la suma de los términos de una progresión geométrica de razón 2 se ha de cumplir:

$$n = \frac{1 \cdot 2^{k-1}}{2} \cdot 2$$

tomando logaritmos se obtiene $k = \log_2 n + 1$.

La función `criba()` realiza, en el peor de los casos, tantas iteraciones como niveles tiene el árbol menos 1, hasta que llega al penúltimo nivel ($\log_2 n$) Por ello se puede concluir que la complejidad es $O(\log n)$, complejidad logarítmica.

La función de ordenación consta de dos bucles, el primero:

```
for (j = n / 2; j >= 1; j--)
```

construye el montículo inicial y se ejecuta $n/2$ veces, por lo que su complejidad:

$$O(n/2 \cdot \log n)$$

El segundo bucle: `for (j = n; j >= 2; j--)` se ejecuta $n - 1$ veces, y por consiguiente su complejidad:

$$O((n - 1) \cdot \log n)$$

Se puede concluir que la eficiencia del método de ordenación:

$$O(n/2 \cdot \log n) + O((n - 1) \cdot \log n)$$

que por las propiedades de la notación O se puede afirmar que la complejidad del método de ordenación *HeapSort* es:

$$O\,(n \cdot \log n)$$

Por tanto, este método de ordenación es de los más eficientes, tiene la misma complejidad que el método *quicksort*.

RESUMEN

La cola de prioridades es una estructura de datos que tiene la propiedad de que siempre se accede al *mínimo elemento*. Una cola de prioridad puede asemejarse a una estructura de *n* colas, tantas como prioridades se quieran establecer en la estructura. La prioridad asignada a cada elemento es la clave de ordenación parcial de los elementos que forman parte de una cola de prioridades. Las operaciones básicas de esta estructura son: *insertar*, *elemento mínimo*, *quitar mínimo*. Insertar añade un elemento a la estructura en el orden que le corresponde según la prioridad y a igual prioridad el último. Las otras dos operaciones acceden al elemento mínimo, en cuanto a la clave, que en realidad es el que tiene mayor prioridad.

En el capítulo se han representado las colas de prioridades con una estructura multienlazada formada por un array de tantas colas como prioridades. A su vez, cada cola implementada con una lista enlazada. Tradicionalmente las colas de prioridades se han implementado con una estructura de árbol binario, llamada *montículo binario*. La definición que se ha dado a esta estructura se corresponde con los montículos minimales los cuales tienen en la raíz el elemento mínimo; también existe la estructura de montículo maximal que en vez de tener el mínimo en la parte más alta tienen el máximo. Las operaciones definidas sobre los montículos: *insertar*, *buscar mínimo* y *eliminar mínimo*, se han implementado representando el montículo en un array, de tal forma que siendo `i` el índice de un nodo, `2*i` y `2*i+1` son los índices del nodo hijo izquierdo y derecho respectivamente.

Como aplicación del montículo *maximal*, el capítulo desarrolla el algoritmo de ordenación *HeapSort*, también llamado *ordenación por montículos*. Se analiza la complejidad del algoritmo, siendo esta `O(n log)`, que está entre los algoritmos de ordenación más eficientes y además fácil de implementar.

EJERCICIOS

12.1. El *TAD Cola de prioridad* se ha implementado utilizando una tabla de colas, tantas como prioridades. Escribir las funciones correspondientes a las operaciones `inserEnPrioridad` y `quitarMin` considerando que se utiliza una lista enlazada en la que se guardan, en orden de prioridad, los elementos de la cola de prioridad.

12.2. Describa un algoritmo para implementar la operación `cambiar()` que sustituya la clave del elemento *k* en un montículo.

12.3. Demuestre la certeza de la siguiente afirmación: en un árbol binario completo de *n* claves, hay exactamente *n*/2 hojas.

12.4. Dibujar un montículo binario *maximal* cuando, a partir de un montículo vacío, se realicen las siguientes operaciones: `insertar(mnlo,10)`, `insertar(mnlo,50)`, `insertar(mnlo,20)`, `insertar(mnlo,60)`, `eliminarMinimo(mnlo)`, `insertar(mnlo,70)`, `insertar(mnlo,30)`.

12.5. Considere el árbol completo de la Figura 12.13; suponiendo que se guarda secuencialmente en el array `v[]` encontrar el montículo que se forma llamando a la función `insertar()` de la estructura montículo, transmitiendo en cada llamada la clave `v[k+1]` para valores `k = 1, 2, ... n-1`; *n* es el número de nodos, en este ejercicio n = 12.

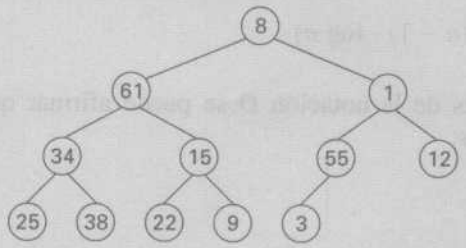


Figura 12.13. Árbol binario completo de 12 nodos.

12.6. Mostrar el resultado del algoritmo de ordenación por montículos sobre la entrada: 18, 11, 22, 33, 11, 34, 44, 2, 8, 11.

12.7. Diseñe un algoritmo para que dados dos montículos binarios se mezclen formando un único montículo. ¿Qué complejidad tiene el algoritmo diseñado?

12.8. 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.

12.9. En un montículo minimal diseñe un algoritmo que encuentre el elemento con mayor clave. ¿Qué complejidad tiene el algoritmo diseñado?

PROBLEMAS

12.1. Escribir un programa en el que se codifique los métodos de ordenación *HeapSort* y *quicksort* y compare la eficiencia de los métodos para los casos de:

a) 1000 elementos ordenados ascendentemente.
b) 1000 elementos ordenados descendentemente.
c) 1000 elementos aleatorios.

12.2. Implementar el algoritmo escrito en el Ejercicio 12.2 para la operación *cambiar* un elemento que ocupa la posición *k*.

12.3. Implementar las operaciones del tipo abstracto *cola de prioridades:* `inserEnPrioridad` y `quitarMin` con llamadas a las operaciones definidas en un montículo.

12.4. La Universidad de La Alcarria dispone de 15 computadores conectados a internet. Se quiere hacer una simulación de la utilización de los computadores por los alumnos. Para ello se supone que la frecuencia de llegada de un alumno es de 18 minutos las 2 dos primeras horas y de 15 minutos el resto del día. El tiempo de utilización del computador es un valor aleatorio, entre 30 y 55 minutos.

El programa debe tener como salida líneas en las que se refleja la llega 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 computadores libres el alumno no espera, se mostrará el correspondiente aviso. En una cola de prioridad se tiene que guardar los distintos «eventos» que se producen, de tal forma que el programa avance de evento a evento. Suponer que la duración de la simulación es de las 10 de la mañana a las 8 de la tarde.

Nota: Utilizar una cola de prioridad/montículo.

12.5. 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 más de 10 personas esperando se va con una probabilidad del 20 por 100, aumentando en 10 puntos por cada 15 personas más que haya esperando, hasta un tope del 50 por 100. El tiempo medio que tarda una persona en pasar es de 1 minuto (compra de la entrada y revisión de los bolsos). Además, cada visitante emplea en recorrer la exposición entre 15 y 25 minutos distribuido uniformemente.

La sala sólo admite, como máximo, 50 personas. Simular el sistema durante un periodo de 6 h para determina:

- Número de personas que llegan a la sala y número de personas que entran.
- Tiempo medio que debe esperar una persona para entrar en la sala.

Utilizar para simulación el *TAD Cola de prioridad.*

13

TABLAS DE DISPERSIÓN, FUNCIONES HASH

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Distinguir entre una tabla lineal y una tabla hash.
- Conocer las aplicaciones que tienen las tablas hash.
- Manejar funciones matemáticas sencillas con las cuales obtener direcciones según una clave.
- Conocer diversos métodos de resolución de colisiones.
- Realizar en C una tabla hash con resolución de colisiones.

CONTENIDO

13.1. Tablas de dispersión.
13.2. Funciones de dispersión.
13.3. Colisiones y resolución de colisiones.
13.4. Exploración de direcciones.
13.5. Realización de una tabla dispersa.
13.6. Direccionamiento enlazado.
13.7. Realización de una tabla dispersa encadenada.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Acceso aleatorio.
- Colisión.
- Dispersión.
- Diccionario.
- Exploración.
- Factor de carga.
- Hueco.
- Tabla.

INTRODUCCIÓN

Las tablas de datos permiten el acceso directo a un elemento de una secuencia, indicando la posición que ocupan. Un *diccionario* es también una secuencia de elementos, pero éstos son realmente pares formados, por ejemplo, por un identificador y un número entero. Las tablas de dispersión se suelen utilizar para implementar cualquier tipo de *diccionario*. La potencia de las tablas *hash* o dispersas radica en la búsqueda de elementos; conociendo el campo clave se puede obtener directamente la posición que ocupa, y por consiguiente, la información asociada a dicha clave. Sin embargo, no permiten algoritmos eficientes para acceder a todos los elementos de la tabla, en su recorrido. El estudio de tablas hash acarrea el estudio de funciones hash o dispersión, que mediante expresiones matemáticas permiten obtener direcciones según una clave que es el argumento de la función.

En el capítulo se estudian diversas funciones hash y cómo resolver el problema de *colisión* (dos o más claves tienen la misma dirección).

13.1. TABLAS DE DISPERSIÓN

Las tablas de dispersión o, simplemente, *tablas hash* son estructuras de datos que se usan en aplicaciones que manejan una secuencia de elementos, de tal forma que cada elemento tiene asociado un valor *clave*, que es un número entero positivo perteneciente a un rango de valores, relativamente pequeño. En estas organizaciones, cada uno de los elementos ha de tener una clave que identifica de manera unívoca al elemento. Por ejemplo, el campo *número de matrícula* del conjunto de alumnos puede considerarse un campo clave para organizar la información relativa al alumnado de la universidad ya que el número de matrícula es único; hay una relación biunívoca, uno a uno, entre el campo y el registro alumno. Puede suponerse que no existen, simultáneamente, dos registros archivo con el mismo número de matrícula.

A tener en cuenta

Un *diccionario* es un tipo abstracto de datos en el que los elementos tienen asociado una clave única en el conjunto de los números enteros positivos tal que para cualquier par de elementos distintos, sus claves son también distintas. Con las tablas de dispersión se implementa eficientemente el tipo *diccionario*.

13.1.1. Definición de una tabla de dispersión

Las tablas de dispersión son estructuras de datos que tienen como finalidad realizar la búsqueda o eliminación de un registro con una complejidad constante (tiempo constante). La organización ideal de una tabla es aquella en la cual el campo clave de los elementos se corresponde directamente con el índice de la tabla. Si por ejemplo una compañía tiene 300 empleados, cada uno identificado con un número de nómina de 0 a 999, la forma de organizar la tabla es con un array de 1.000 registros:

```
struct Empleado Tabla[1000];
```

El elemento `Tabla[i]` almacena al empleado cuya nómina es `i`. Con esta organización la búsqueda de un empleado se ha realizado directamente, con un único acceso, debido a que el número de nómina es la posición en la tabla. La eficiencia se puede expresar como tiempo constante, $O(1)$.

Sin embargo, muchas posiciones de la tabla están vacías ya que se corresponden con números de nómina que no existen. En este ejemplo el rango del campo clave es relativamente pequeño; imagínese que los números de nómina fueran de 5 dígitos, las posiciones no existentes estarían en clara desproporción y la memoria ocupada por la tabla estaría desaprovechada. Pero enseguida se puede plantear una solución: tomar los tres primeros dígitos del número de nómina, campo clave, como índice del array o tabla de registros, entonces se ha hecho una transformación del campo clave en un entero de 3 dígitos:

```
h(número de nómina) → indice
```

Se puede concluir que el primer problema que plantea esta organización es: ¿cómo evitar que el array utilizado esté en una proporción adecuada al número de registros? Las funciones de transformación de claves, funciones hash, permiten que el rango posible de índices estén en proporción al número real de registros.

Una función que transforma números grandes en otros más pequeños se conocen como *funciones de dispersión* o *funciones hash*.

Para recordar

Una tabla de dispersión consta de una array, donde se almacenan los registros o elementos, y de una función hash que transforma el campo clave elegido en el rango entero del array.

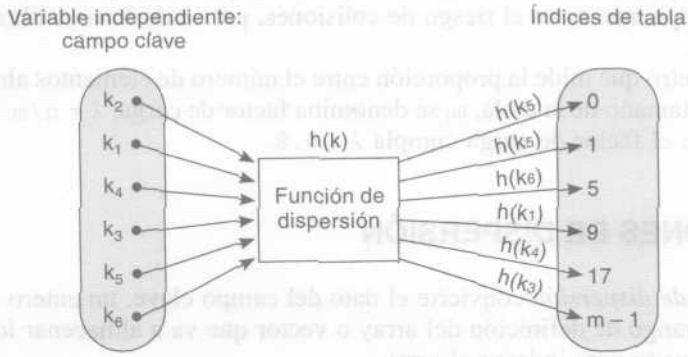


Figura 13.1. Función de dispersión.

13.1.2. Operaciones de una tablas de dispersión

Las tablas dispersas implementan eficientemente los tipos de datos denominados *diccionarios*. Un diccionario asocia una *clave* con un *valor*. Así, por ejemplo el diccionario de inglés asocia una palabra con la traducción en inglés. La operación que se realiza con más frecuencia es *buscar*; dada una palabra, se localiza su posición, y se obtiene la traducción correspondiente (el *valor asociado*).

Una tabla dispersa hay que crearla y dar de alta en ella a elementos. La operación de *insertar* añade un elemento, en la posición que le corresponde según la clave del elemento. De igual manera, también pueden darse de baja elementos. La operación *eliminar* extrae un elemento de la tabla que se encuentra en la posición que le corresponde según su clave.

En las tablas dispersas no se utiliza directamente la clave para indexar, el índice se calcula con una función matemática, *función hash h(x)*, que tiene como argumento la clave del elemento y devuelve una dirección o índice en el rango de la tabla. Según esto, considerando *h(x)* la función hash, se pueden especificar las operaciones del tipo *tabla dispersa*:

Buscar (tabla T, clave x)

```
devuelve el elemento de la tabla T[h(x)]
```

Insertar (tabla T, elemento k)

```
añade el elemento k, T[h(clave(k)] ← k
```

Eliminar (tabla T, clave x)

```
retira de la tabla el elemento con clave x, T[h(x)] ← LIBRE
```

La ventaja de utilizar tablas de dispersión radica en la eficiencia de estas operaciones. Si la función hash es de complejidad constante, la complejidad de cada una de estas tres operaciones

también es de complejidad constante, $O(1)$. Un problema que surge con la función de dispersión o función hash es que pueden darse colisiones: para dos claves distintas x_i, x_j se obtiene la misma dirección o índice, $h(x_i) = h(x_j)$. La operación *insertar* ha de incorporar el proceso de *resolución de colisiones*: no pueden estar dos elementos en la misma posición. De igual forma, el proceso de las operaciones *buscar* y *eliminar* también quedan afectadas por la resolución de colisiones.

Las tablas dispersas se diseñan teniendo en cuenta el problema de las colisiones. Siempre se reservan más posiciones de memoria, m, que elementos están previstos almacenar, n. Cuanta más posiciones haya, menor es el riesgo de colisiones, pero más *huecos* libres quedan (memoria desaprovechada).

El parámetro que mide la proporción entre el número de elementos almacenados, n, en una tabla dispersa y el tamaño de la tabla, m, se denomina factor de carga, $\lambda = n/m$. Se recomienda elegir m de tal forma que el factor de carga cumpla $\lambda \leq 0.8$.

13.2. FUNCIONES DE DISPERSIÓN

Una *función de dispersión* convierte el dato del campo clave, un entero o una cadena, en un valor entero en el rango de definición del array o vector que va a almacenar los elementos, de tal forma que sea adecuado para indexar el array.

La idea básica es utilizar la clave de un registro para determinar su dirección, pero sin desperdiciar mucho espacio, para lo que hay que realizar una transformación, mediante una función hash, del conjunto K de claves sobre el conjunto L de direcciones de memoria.

$$h(x) : K \rightarrow L$$

Ésta es la función de direccionamiento *hash* o función de dispersión. Si *x* es una clave, entonces *h(x)* se denomina direccionamiento *hash* de la clave *x*, además es el índice de la tabla donde se debe guardar el registro con esa clave. Así, si la tabla tiene un tamaño de `tamTabla=199`, la función hash elegida tiene que generar índices en el rango `0 .. tamTabla-1`. Si la clave es un entero, por ejemplo el número de serie de un artículo (hasta 6 dígitos) y se dispone de una tabla de `tamTabla` elementos, la función de direccionamiento tiene que ser capaz de transformar valores pertenecientes a un conjunto de `999999` elementos en un conjunto de `tamTabla`. La clave también puede ser una cadena de caracteres, en ese caso se hace una transformación previa a valor entero. La función más simple utiliza el operador *módulo* (*resto entero*).

```
x % tamTabla = genera un número entero de 0 a tamTabla-1
```

Hay que contemplar el hecho de que la *función hash*, `h(x)`, no dé valores distintos: es posible (según la función elegida) que dos claves diferentes `c1` y `c2` den la misma dirección, `h(c1) = = h(c2)`. Entonces se produce el fenómeno de la *colisión*, y se debe usar algún método para resolverla. Por tanto, en el estudio del *direccionamiento hash* hay que dividirlo en dos partes: *búsqueda de funciones hash* y *resolución de colisiones*.

Existe un número considerable de funciones hash. A la hora de seleccionar una función, dos criterios deben tenerse en cuenta. En primer lugar, `h(x)` ha de evaluarse fácilmente, dependerá de la clave y se ha de procurar que tenga complejidad constante. En segundo lugar, `h(x)` debe distribuir uniformemente las direcciones sobre el conjunto L, de forma que se minimice el número de *colisiones*. Nunca existirá una garantía plena de que no haya colisiones, y máxime sin conocer de antemano las claves y las direcciones. La experiencia enseña que siempre habrá que preparar la *resolución de colisiones* para cuando se produzca alguna.

Algunas de las funciones hash de cálculo más fácil y rápido se exponen en las secciones siguientes. Para todas ellas se sigue el criterio de considerar a x una clave cualquiera, m el tamaño de la tabla, y por tanto, los índices de la tabla varían de `0` a `m-1` y el número de elementos es n.

13.2.1. Aritmética modular

Una *función de dispersión* que utiliza la *aritmética modular* genera valores dispersos calculando el resto de la división entera entre la clave `x` y el tamaño de la tabla `m`.

```
h(x)= x mod m
```

Normalmente la clave asociada con un elemento es de tipo entero. A veces los valores de la clave no son enteros, entonces, previamente, hay que transformar la clave a un valor entero. Así ocurre cuando la clave es una cadena de caracteres: se puede transformar considerando el valor `ASCII` de cada carácter como un dígito de una cantidad entera de base `128`.

La operación resto siempre genera un numero entre `0` y `m-1`. Por tanto, esta *función de dispersión* calcula valores enteros de `0 .. m-1`.

Hay que tener ciertas precauciones con la elección del tamaño de la tabla, `m`, para que esta función disperse lo más uniformemente posible. No es recomendable escoger `m` como múltiplo de `2`, ni tampoco múltiplo de `10`. Si $m = 2^j$ entonces la distribución de `h(x)` se basa únicamente en los `j` dígitos menos significativos; y si $m = 10^j$ ocurre lo mismo. En estos casos, la distribución de las claves se basa sólo en una parte, los `j` dígitos menos significativos, de la información que suministra la clave y esto sesgará la dispersión hacia ciertos valores o índices de la tabla. Si por ejemplo el tamaño elegido para la tabla es $m = 100$ (10^2), entonces `h(128) = h(228) = h(628) = 28`.

Las elecciones del tamaño de la tabla, `m`, recomendadas, son números primos que estén cercanos al número de elementos, `n`, que se tiene previsto almacene la tabla.

Ejemplo 13.1

Considerar una aplicación en la que se debe almacenar `n = 900` registros. El campo clave elegido es el número de identificación. Elegir el tamaño de la tabla de dispersión y calcular la posición que ocupa los elementos cuyo número de identificación es:

```
245643    245981    57135
```

Una buena elección de `m`, en este supuesto, es `997` al ser un número primo próximo y tener como factor de carga (`n/m`) aproximadamente `0.8` cuando se hayan guardado todos los elementos.

Teniendo en cuenta el valor de `m`, se aplica la función hash de aritmética modular y se obtienen estas direcciones:

```
h(245643) = 245643 mod 997 = 381
h(245981) = 245981 mod 997 = 719
h(257135) = 257135 mod 997 = 906
```

13.2.2. Plegamiento

La técnica del *plegamiento* se utiliza cuando el valor entero del campo clave elegido sea demasiado grande, pudiendo ocurrir que no pueda ser almacenado en memoria. Consiste en partir la clave `x` en varias partes, x_1, x_2, x_3 . . . x_n, y la combinación de las partes de un modo conveniente (a menudo sumando las partes) da como resultado la dirección del registro.

Cada parte x_i, con la excepción a lo sumo de la última, tiene el mismo número de dígitos que la dirección especificada.

La función hash se define:

$$h(x) = x_1 + x_2 + x_3 + \ldots + x_r$$

En la operación que se realiza para el cálculo de la función hash, se desprecian los dígitos más significativos que se obtienen del acarreo.

A tener en cuenta

La técnica de plegar la clave de dispersión, a menudo se utiliza para transformar una clave muy *grande* en otra más *pequeña*, y a continuación aplicar la función hash de aritmética modular.

Ejemplo 13.2

Los registros de pasajeros de un tren de largo recorrido se identifican por un campo de 6 dígitos, que se va a utilizar como clave para crear una tabla dispersa de m = 1.000 posiciones (rango de 0 a 999). La función de dispersión que se utiliza pliega la clave, 6 dígitos, en dos grupos de tres dígitos, y a continuación, se suman los valores de cada grupo. Aplicar esta función a los registros:

```
245643    245981    257135
```

La primera clave, `245643`, al dividirla en dos grupos de 3 dígitos: `245` y `643` se obtiene la dirección:

```
h(245643) = 245 + 643 = 888
```

para las otras claves:

```
h(245981) = 245 + 981 = 1226 = 226 (se ignora el acarreo 1)
h(257135) = 257 + 135 = 392
```

A veces se hace la inversa a las partes pares, x_2, x_4 . . . antes de sumarlas, con el fin de conseguir mayor dispersión. Así, con las mismas claves se obtienen las direcciones:

```
h(245643) = 245 + 346 = 591
h(245981) = 245 + 189 = 434
h(257135) = 257 + 531 = 788
```

13.2.3. Mitad del cuadrado

Esta técnica de obtener direcciones dispersas consiste, en primer lugar, en calcular el cuadrado de la clave `x`, y a continuación extraer de x^2, los dígitos que se encuentran en ciertas posiciones. El número de dígitos a extraer depende del rango de dispersión que se desea obtener. Así, si el rango es de `0 .. 999`, se extraen tres dígitos, siempre, aquellos que están en las mismas posiciones.

Un problema potencial, al calcular x^2, es que sea demasiado grande y *exceda* el máximo entero. Es importante, al aplicar este método de dispersión, utilizar siempre las mismas posiciones para todas las claves.

Ejemplo 13.3

Aplicar el método de mitad del cuadrado a los mismos registros del Ejemplo 13.2.

Una vez elevado al cuadrado el valor de la clave, se eligen los dígitos que se encuentran en las posiciones 4, 5 y 6 por la derecha. El valor de esa secuencia es la dirección obtenida al aplicar este método de dispersión.

Para `245643, h(245643) = 483;` *paso a paso:*

`245643` → 245643^2 → `60340483449` → (dígitos 4, 5 y 6 por la derecha) `483`

Para `245981, h(245981) = 652;` *paso a paso:*

`245981` → 245981^2 → `60506652361` → (dígitos 4, 5 y 6 por la derecha) `652`

Para `257135, h(257135) = 408;` *paso a paso:*

`257135` → 257135^2 → `66118408225` → (dígitos 4, 5 y 6 por la derecha) `408`

13.2.4. Método de la multiplicación

La dispersión de una clave utilizando el *método de la multiplicación* genera direcciones en tres pasos. Primero, multiplica la clave `x` por una constante real, `R`, comprendida entre `0` y `1` `(0. < R < 1.0)`. En segundo lugar, determina la parte decimal, `d`, del número obtenido, `Rx`, y por último, multiplica el tamaño de la tabla, `m`, por `d` y al truncarse el resultado se obtiene un número entero en el rango `0 .. m-1`.

1. `R * x`
2. `d = R * x - ParteEntera (R * x)`
3. `h(x) = ParteEntera (m * d)`

Una elección de la constante `R` es la inversa de la razón áurea, `R = 0.6180334`. Así, si por ejemplo la clave `x = 245981, m = 1.000,` la dirección que se obtiene:

1. `R * x:  0.6180334 * 245981 → 152024.4738`
2. `d:  152024.4738 - ParteEntera (152024.4738) → 0.4738`
3. `h (245981): 1.000 * 0.4738 → ParteEntera (473.8) → 473`

Este método de obtener direcciones dispersas tiene dos características importantes. La primera, dos claves con los dígitos permutados no tienen mayor probabilidad de que se produzca una colisión (igual dirección) que dos claves cualesquiera. La segunda característica, dos valores de claves muy próximas numéricamente, produce direcciones dispersas que pueden estar muy separadas.

Sea por ejemplo, la clave `x = 245982` para el mismo valor de `R = 0.6180334` y `m = 1.000,` la dispersión:

1. `R * x:` `0.6180334 * 245982 → 152025.0918`

2. `d:` `152025.0918 - ParteEntera (152025.0918) → 0.0918`

3. `h (245982): 1.000 * 0.0918 → ParteEntera (91.8) → 91`

La dispersión obtenida para `245891`, `h(245981) = 473` está muy alejada de `91`.

Ejercicio 13.1

Los registros que representan a los objetos de una perfumería se van a guardar en una tabla dispersa de `m = 1.024` *posiciones. El campo clave es una cadena de caracteres cuya máxima longitud es 10. Se decide aplicar el método de la multiplicación como función de dispersión. Con este supuesto, codificar la función de dispersión y mostrar 10 direcciones dispersas.*

Para generar la dispersión de las claves, primero se transforma la cadena, que es el campo clave, en un valor entero. Una vez hecha la transformación, se aplica el método de la multiplicación.

La transformación de la cadena se realiza considerando que es una secuencia de valores numéricos en base `27`. Así por ejemplo, la cadena `'RIO'` se transforma en:

$$'R' * 27^2 + 'I' * 27^1 + 'O' * 27^0$$

El valor entero de cada carácter es el ordinal `ASCII`. En C, un carácter se representa como un valor entero que es, precisamente, el ordinal. La transformación da lugar a valores que sobrepasan el máximo entero (incluso con enteros largos), generando números negativos. No es problema, simplemente se cambia de signo.

Para generar las 10 direcciones dispersas y probar la función, el programa tiene como entrada cada una de las 10 cadenas, invoca a la función de dispersión y se escribe la dirección.

```
/*
                función hash: método de la multiplicación

  Las claves son cadenas de caracteres, primero se transforma
  a valor entero. A continuación se aplica el método de
  multiplicación
*/

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

#define m 1024
#define R 0.618034

long transformaClave(const char* clave);
int dispersion(long x);

void main()
{
  char clave[11];
```

```
  long valor;
  int k;

  for (k = 1; k <= 10; k++)
  {

    printf("\nClave a dispersar: ");
    gets(clave);
    valor = transformaClave(clave);
    valor = dispersion(valor);
    printf ("Dispersion de la clave %s → %ld\n", clave, valor);
  }
}
long transformaClave(const char* clave)
{
  int j;
  long d;

  d = 0;
  for (j = 0; j < strlen(clave); j++)
  {
    d = d * 27 + clave[j];
  }

    /*
      En caso de valor grande no se puede almacenar en memoria
      y genera un numero negativo
    */

  if (d < 0) d = -d;

  return d;
}

int dispersion (long x)
{
  double t;
  int v;
  t = R * x - floor (R * x); /* parte decimal */
  v = (int) m * t;

  return v;
}
```

13.3. COLISIONES Y RESOLUCIÓN DE COLISIONES

La dirección de dos registros, según la clave elegida, puede que sea la misma posición de la tabla, entonces se produce una **colisión** que hay que resolver. Una función hash ideal, `h(x)`, debe de generar direcciones distintas para dos claves distintas. No siempre es así, no siempre proporciona

direcciones distintas: puede ocurrir que para dos claves diferentes existan direcciones diferentes $x_1, x_2 \rightarrow h(x_1) = h(x_2)$. Este hecho es conocido como colisión; es evidente que al diseñar una tabla dispersa se debe proporcionar métodos de *resolución de colisiones*.

A tener en cuenta

Hay dos cuestiones que se deben considerar a la hora de diseñar una tabla hash. Primero, seleccionar una buena función hash, que disperse lo más uniformemente. Segundo, seleccionar un método para resolver colisiones.

Considérese, por ejemplo, una comunidad de 88 vecinos que tienen muchos campos de información relevante: nombre, edad ... Se elige como campo clave el Documento Nacional de Identidad (DNI) y una tabla de 101 direcciones. Con la función hash del *módulo*, la claves `123445678`, `123445880` proporcionan las direcciones:

```
h(123445678) = 123445678 mod 101 = 44
h(123445880) = 123445880 mod 101 = 44
```

Para dos claves distintas, a las que se aplica la función hash del módulo, se obtienen dos direcciones iguales, se dice que las claves han colisionado.

Es importante tener en cuenta que la manera de resolver las colisiones, en una tabla de dispersión, afecta directamente a la eficiencia de las operaciones básicas sobre la tabla: *insertar*, *buscar* y *eliminar*.

Se consideran dos modelos para resolver colisiones: *exploración de direcciones* y *direccionamiento enlazado*. En las siguientes secciones se muestran los diversos métodos de resolver las colisiones.

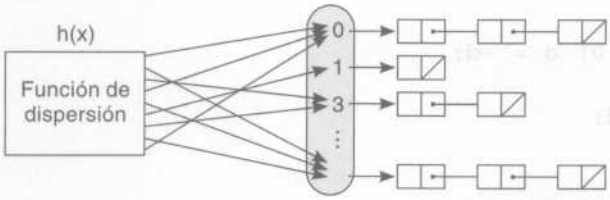


Figura 13.2. Resolución de colisiones por dispersión abierta (enlazada).

13.4. EXPLORACIÓN DE DIRECCIONES

Los diversos métodos de exploración se utilizan cuando todos los elementos, colisionados o no, se almacenan en la misma tabla. Las colisiones se resuelven explorando sucesivamente en una secuencia de direcciones, hasta que se encuentra una posición *libre*, un *hueco*, en el caso del proceso de *insertar*, o se encuentra el elemento buscado en las operaciones *buscar* y *eliminar*.

Es importante, al diseñar una tabla dispersa en la cual se hace la resolución de colisiones con una secuencia de exploración, inicializar todas las posiciones de la tabla a un valor que indique vacío, `NULL` por ejemplo, o cualquier parámetro que indique posición vacía. Al insertar un elemento, si se produce una colisión, la secuencia de exploración termina cuando se encuentra una dirección de la secuencia vacía.

Al buscar un elemento por su clave, es posible tener que explorar la secuencia de posiciones hasta encontrar la clave buscada. En la operación de eliminar, una vez encontrada la clave, se indica

con un parámetro el estatus de borrado (eliminación). Dependiendo de la aplicación una posición eliminada puede utilizarse, posteriormente, para una inserción.

A tener en cuenta

La secuencia de índices a la que da lugar un método de exploración ha de ser la misma para cada una de las operaciones que se aplican a una tabla dispersa. Es importante señalar con un parámetro que una posición de la tabla esté vacía.

13.4.1. Exploración lineal

Es la forma más primaria y simple de resolver una colisión entre claves al aplicar una función de dispersión. Supóngase que se tiene un elemento de clave `x`, la dirección que devuelve la función es `h(x) = p`, si esta posición ya está ocupada por otro elemento se ha producido una colisión. La forma de resolver está colisión con exploración lineal, consiste en buscar la primera posición disponible que siga a `p`. La secuencia de exploración que se genera es lineal: `p, p+1, p+2, ... m-1, 0, 1, ...` y así consecutivamente hasta encontrar una posición vacía. La tabla se ha de considerar *circular*, si `m-1` es la última posición, la que sigue es la posición `0`.

Ejemplo 13.4

Existen 9 elementos cuyas claves simbólicas son: `x1, x2, x3, x4, x5, x6, x7, x8` y `x9`. Para cada una de ellas la función de dispersión genera las direcciones:

Elemento:	x1	x2	x3	x4	x5	x6	x7	x8	x9
h(x):	5	8	11	9	5	7	8	6	14

Entonces las posiciones de almacenamiento en la tabla, aplicando la exploración lineal:

Elemento:	x1	x2	x3	x4	x5	x6	x7	x8	x9
Posición:	5	8	11	9	5	7	8	6	14

La primera colisión se produce al almacenar el elemento `x5`, según la función hash le corresponde la posición `5` que está ocupada. La siguiente posición está libre y a ella se asigna el elemento. Al elemento `x8` le corresponde la dirección dispersa `6`, sin embargo, esa posición ya está ocupada por `x5` debido a una colisión previa. La siguiente posición libre es la `12`, que es la dirección que le corresponde en la exploración lineal.

Las operaciones de *buscar* y *eliminar* un elemento en una tabla dispersa con exploración lineal sigue los mismos pasos que insertar (Ejemplo 13.4). La búsqueda se inicia en la posición que devuelve la función hash, `h(clave)`. Si la clave en esa posición es la clave de búsqueda la operación ha tenido éxito, en caso contrario la exploración continúa en las siguientes posiciones hasta encontrar la clave, o bien una posición vacía y en este caso no existe en la tabla. Por ejemplo, para buscar el elemento `x8` se examinan las posiciones `6, 7, 8, 9, 10, 11` y `12`.

La exploración lineal es muy sencilla de implementar; tiene un inconveniente importante, el agrupamiento de elementos en la tabla. Sobre todo, cuando el factor de ocupación se acerca a 0.5, el agrupamiento de elementos en posiciones adyacentes es notable.

Análisis de la exploración lineal

La eficiencia de una función hash, junto al método de resolución de colisiones, supone analizar la operación de *insertar* y la operación de *buscar* (*eliminar* supone una búsqueda). El factor de carga de la tabla, λ, es determinante a la hora de la eficiencia de las operaciones.

La eficiencia de la inserción se suele medir como *el número medio de posiciones examinadas*. En la exploración lineal esta magnitud se aproxima:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

Al analizar la operación de búsqueda de un elemento se considera que tenga éxito y que no se encuentre. Para ello se mide mediante dos parámetros:

`S(λ)` = número medio de comparaciones para una búsqueda con éxito.

`U(λ)` = número medio de comparaciones para una búsqueda sin éxito.

Se puede demostrar que para la exploración lineal estos parámetros:

$$S(\lambda)=\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right) \quad \text{y} \quad U(\lambda)=\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

A tener en cuenta

La exploración lineal es fácil de implementar en cualquier lenguaje de programación. Tiene como principal inconveniente que agrupa los elementos en posiciones contiguas, cuando el factor de carga supera el 50 por 100. El situar los elementos en posiciones contiguas aumenta el tiempo medio de búsqueda.

13.4.2. Exploración cuadrática

La resolución de colisiones con la exploración lineal provoca que se agrupen los elementos de la tabla según se va acercando el factor de carga a `0.5`. Una alternativa para evitar la agrupación es la *exploración cuadrática*. Suponiendo que un elemento con clave `x` le corresponde la dirección `p`, y la posición de la tabla indexada por `p` está *ocupada*, el método de prueba cuadrática busca en las direcciones `p, p+1, p+4, p+9, ... p+i²`, considerando el array como circular. El nombre de *cuadrática* a esta forma de explorar se debe al desplazamiento relativo, `i²`, para valores de `i= 1, 2, 3, ...`.

Ejemplo 13.5

Considere 9 elementos cuyas claves simbólicas son: `x1, x2, x3, x4, x5, x6, x7, x8` y `x9`. El tamaño de la tabla donde se guarda es `17` (número primo), rango de `0` a `16`. Para cada uno de los elementos la función de dispersión genera las direcciones:

Elemento:	x1	x2	x3	x4	x5	x6	x7	x8	x9
h(x) :	5	8	10	8	5	11	6	7	7

Las posiciones de almacenamiento en la tabla, aplicando la exploración cuadrática:

Elemento:	x1	x2	x3	x4	x5	x6	x7	x8	x9
Posición:	5	8	10	9	6	11	7	16	15

La primera colisión se produce al almacenar el elemento `x4`; según la función hash le corresponde la posición `8`, que está ocupada. La secuencia de exploración cuadrática es: `8, 8+1, 8+4, ...`, como la posición `9`, está libre, a ella se asigna el elemento.

Al elemento `x5` le corresponde la dirección dispersa `5`, sin embargo, hay una colisión con el elemento `x1`; la secuencia de exploración para esta clave es: `5, 5+1, 5+4, ...`, como la posición `6` está libre, se asigna el elemento.

La siguiente colisión se da con `x7`, le corresponde la dirección dispersa `6`, sin embargo, esa posición está ya ocupada por una colisión; la secuencia de exploración para esta clave es: `6, 6+1, 6+4, ...`, la posición `7` está libre y se asigna el elemento.

Al elemento `x8` le corresponde la dirección `7` ya asignada; la secuencia de exploración determina la dirección 16. El elemento `x9` le correponde la dirección 7, se produce una colisión. La secuencia de exploración, `7, 7+1, 7+4, 7+9, 7+16(6), 7+25 ...`, resulta que ahora hay que tratar la secuencia en forma circular al estar ocupadas las cuatro primeras posiciones. El siguiente valor de la secuencia es `23`, se corresponde con `6` (*teoría de restos*), también está ocupado. A continuación se analiza el valor `32`, que se corresponde con `15` por la *teoría de restos*, al estar libre se asigna.

Análisis de la exploración cuadrática

Este método de resolver colisiones reduce el agrupamiento que produce la exploración lineal. Sin embargo, si no se elige convenientemente el tamaño de la tabla no se puede asegurar que se prueben todas las posiciones de la tabla.

Se puede demostrar que si el tamaño de la tabla es un número primo y el factor de carga no alcanza el 50 por 100, todas las *pruebas* que se realicen con la secuencia $p+i^2$ se hacen sobre posiciones de la tabla distintas y siempre se podrá insertar.

13.4.3. Doble dirección dispersa

Este método de resolución de colisiones utiliza una segunda función hash. Considere una función hash principal, `h(x)`, y otra función secundaria, `h'(x)`. El primer intento de insertar, o buscar, un nuevo elemento inspecciona la posición `h(x) = p`; si hay una colisión se resuelve con ayuda de la segunda función, `h'(x) = p'`. Se inspecciona la posición `p+p'`; en el caso de nueva colisión, la inspección continúa en `p+2p'`; de ser necesario, la siguiente posición es `p+3 p'` y así sucesivamente hasta encontrar una posición libre, o bien, el elemento buscado.

13.5. REALIZACIÓN DE UNA TABLA DISPERSA

Se va a realizar una implementación completa, en C, de una tabla dispersa aplicando la exploración cuadrática para la resolución de colisiones. Se supone que la tabla va a dar soporte a un conjunto de elementos, que a título de ejemplo son las *Casas rurales de la comarca de La Alcarria*.

Los elementos son *estructuras*, registros, que constan de una serie de datos: *población*, *dirección*, *habitaciones*, *precio* por día y *código* de identificación. El código consta de 5 caracteres y

tiene una relación biunívoca con la Casa Rural, por lo que se elige como campo clave. Los campos *población* y *dirección*, son de tipo cadena, cada uno de ellos tiene un máximo de caracteres.

La tabla dispersa consiste en un vector de *estructuras* `CasaRural`; en realidad, va a consistir en un vector de punteros, referencias, a esa estructura. El tamaño de la tabla está en función del número de *Casas* conocido, 50, por lo cual el tamaño es 101. Cada posición de la tabla contiene `NULL`, o apunta a una *estructura* `CasaRural`. Los elementos que se dan de baja van a permanecer en la tabla, por ello se añade un nuevo campo, `esAlta`, que si está activo la estructura está dada de *alta*, en caso contrario se dio de *baja*. El número de elementos que hay en la tabla, incluyendo las bajas, se almacena en la variable `numElementos`; además, se añade la variable `factorCarga`, de tal forma que cuando se alcance el 0.5 se pueda generar un aviso. Las declaraciones:

```
#define TAMTABLA 101
struct TipoCasa
{
    char codigo[5];
    char poblacion[31];
    char direccion[51];
    int habitaciones;
    double precio;
    int esAlta;
};
typedef struct TipoCasa CasaRural;

struct TipoTabla
{
    CasaRural* tabla[TAMTABLA];
    int numElementos;
    double factorCarga;
};
typedef struct TipoTabla TablaDispersa;
```

13.5.1. Creación de la tabla dispersa

La creación de la tabla es, simplemente, una inicialización de cada posición a `NULL` y las variables a cero. Estas acciones se escriben en la función `crearTabla()`; es la primera función a invocar a la hora de manejar una tabla dispersa.

```
void crearTabla(TablaDispersa* t)
{
    int j;

    for (j = 0; j < TAMTABLA ; j++)
        t -> tabla[j] = NULL;
    t -> numElementos = 0;
    t -> factorCarga = 0.0;
}
```

13.5.2. Posición de un elemento

Al añadir un nuevo elemento a la tabla, o bien al buscar un elemento, hay que determinar la posición en ella. Se emplea la función hash *aritmética modular*, con exploración cuadrática. El campo

clave es de tipo cadena, por lo que primero hay que convertir la cadena en un valor entero. Se considera a la cadena como si fuera un secuencia numérica; los caracteres son los dígitos cuyo valor es el ordinal ASCII, y la base de numeración, debería ser 127, se toma 27 para no generar valores demasiado altos. La función `transforma()` realiza la conversión.

Se prueba con cada posición que determina la exploración cuadrática hasta encontrar una posición vacía (NULL), o bien una posición con la misma clave. La función `direccion()` gestiona estas acciones.

```
long transforma(const char* clave)
{
   int j;
   long d = 0;

   for (j = 0; j < strlen(clave); j++)
   {
      d = d * 27 + clave[j];
   }
   return ((d >= 0) ? d : -d);
}

int direccion(TablaDispersa t, const char* clave)
{
   int i = 0;
   long p, d;

   d = transforma(clave);
      /* se aplica aritmética modular para obtener dirección base */
   p = d % TAMTABLA;
                                          /* bucle de exploración */
   while (t.tabla[p]!= NULL && strcmp(t.tabla[p]->codigo,clave)
                                                        != 0)
   {
      i++;
      p = p + i*i;
      p = p % TAMTABLA; /* considera array como circular */
   }

   return (int)p;
}
```

13.5.3. Insertar un elemento en la tabla

Para añadir un nuevo elemento a la tabla, se llama primero a la función `direccion()`, que devuelve la posición donde asignar el nuevo elemento. La operación se despreocupa del hecho que en esa posición haya un elemento; si es así, se *sobrescribe*. El número de entradas de la tabla se incrementa y se hace un nuevo cálculo del factor de carga. La única acción que se hace con el factor de carga es escribir un mensaje, si éste supera el 50 por 100.

```
void insertar(TablaDispersa* t, CasaRural r)
{
```

```
  CasaRural* pr ;
  int posicion;

  pr = (CasaRural*) malloc(sizeof(CasaRural));

  strcpy(pr->poblacion, r.poblacion);
  strcpy(pr->direccion, r.direccion);
  strcpy(pr->codigo, r.codigo);
  pr->habitaciones = r. habitaciones;
  pr->precio = r.precio;
  pr->esAlta = 1;

  posicion = direccion(*t, r.codigo);
  t -> tabla[posicion] = pr;
  t -> numElementos++;
  t -> factorCarga = (t -> numElementos)/TAMTABLA;
  if (t -> factorCarga > 0.5)
  puts("\n!!! Factor de carga supera el 50% de la tabla !!!");
}
```

13.5.4. Búsqueda de un elemento

La búsqueda de un elemento en la tabla se implementa de tal forma que la función `buscar()` devuelve un puntero al elemento, si se encuentra en la tabla. De no encontrarse, o bien esté dado de baja, devuelve `NULL`.

```
CasaRural* buscar(TablaDispersa t, const char* clave)
{
 CasaRural* pr;
 int posicion;

 posicion = direccion(t, clave);
 pr = t.tabla[posicion];
 if (pr != NULL)
   if (!(pr -> esAlta)) pr = NULL;

 return pr;
}
```

13.5.5. Dar de baja un elemento

Para dar de baja a un elemento se siguen los mismos pasos que en la operación de búsqueda. Primero se determina la posición del elemento en la tabla: se llama a la función `direccion()`. A continuación, si en la posición hay un elemento, simplemente se desactiva.

```
void eliminar(TablaDispersa* t, const char* clave)
{
  int posicion;
  posicion = direccion(*t, clave);
```

```
    if (t -> tabla[posicion] != NULL)
        t -> tabla[posicion] -> esAlta = 0;
}
```

13.6. DIRECCIONAMIENTO ENLAZADO

En el apartado anterior se han resuelto las colisiones analizando consecutivamente las posiciones de una secuencia de exploración (exploración lineal, exploración cuadrática y doble direccionamiento). Una alternativa a la secuencia de exploración es el direccionamiento o *hashing* enlazado. Se basa en utilizar listas enlazadas (cadenas), de tal forma que en cada lista se colocan los elementos que tienen la misma dirección *hash*. Todos los elementos que *colisionan*: `h(x1) = h(x2) = h(x3) ...`, van a estar ubicados en la misma lista enlazada.

En la Figura 13.3 se puede ver la estructura de datos básica para este método de dispersión. La idea fundamental es que si se ha elegido una función hash que genera un rango de `m` direcciones, `0` a `m-1`, se ha de crear una tabla de `m` elementos, indexada de `0` a `m-1`, que contiene los punteros de acceso a las listas.

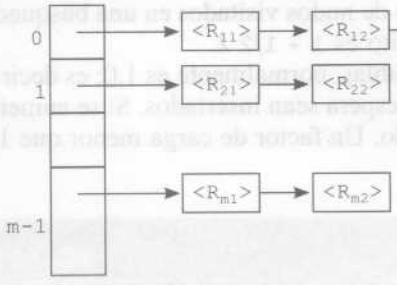


Figura 13.3. Tabla de dispersión enlazada con rango `0..m-1` direcciones.

La estructura de datos que se emplea para el direccionamiento enlazado es un vector de listas. Cada posición `i` del vector es un puntero a la lista enlazada cuyos nodos son los elementos de la aplicación para los cuales la función hash devuelve la dirección `i`. Ahora, los elementos de la tabla es necesario que tengan un campo adicional para poder enlazar con el *siguiente*, además de los propios. El esquema de declaraciones en C:

```
struct TipoElemento
{
    < variables, según los datos a representar >
    struct TipoElemento* sgte;
};
typedef struct TipoElemento Elemento;
struct TipoTabla
{
    Elemento* tabla[M];
};
typedef struct TipoTabla TablaDispersa;
```

13.6.1. Operaciones de la tabla dispersa enlazada

Las operaciones básicas con tablas dispersas (*insertar*, *buscar* y *eliminar*) se convierten con este método en operaciones sobre listas una vez que se obtiene el índice de la tabla mediante la función hash.

Para añadir un elemento con clave `x`, se inserta en la lista de índice `h(x)`. Puede ocurrir que la lista esté vacía, por no haber habido colisión, o bien que ya tenga elementos. La inserción en la lista se hace como primer elemento, y de esa forma la complejidad de la inserción es `O(1)`.

Un elemento de la tabla puede ahora eliminarse *físicamente*; no es necesario marcar el elemento como *no activo*, como ocurre con las secuencias de exploración. Los pasos a seguir son: obtener el índice de la tabla `h(x)` y a continuación aplicar la operación de *eliminar* un nodo en una lista enlazada.

La búsqueda de un elemento sigue los mismos pasos: se obtiene el índice de la tabla mediante la función hash y a continuación, se aplica la operación de *búsqueda* en listas.

13.6.2. Análisis del direccionamiento enlazado

El planteamiento de *hashing* enlazado (encadenado) permite acceder directamente a la lista enlazada donde se va a buscar un elemento. La búsqueda es una operación lineal, en el peor de los casos recorre todos los nodos de la lista; a pesar de ello, las listas han de tener un número suficientemente pequeño y así el tiempo de búsqueda será reducido.

Teniendo en cuenta el factor de carga, λ, la longitud media de cada lista es, precisamente, λ. Por esta razón el número medio de nodos visitados en una búsqueda sin éxito es λ, y el promedio de nodos en una búsqueda con éxito es $1 + 1/2\ \lambda$.

El factor de carga en estas tablas, normalmente es 1.0; es decir, el rango del vector coincide con el número de elementos que se espera sean insertados. Si se aumenta el factor de carga la eficiencia de la búsqueda va disminuyendo. Un factor de carga menor que 1.0 hace que aumente la memoria no utilizada.

A tener en cuenta

El principal inconveniente del direccionamiento con encadenamiento es el espacio adicional, de cada elemento, necesario para enlazar un nodo de la lista con el siguiente.

13.7. REALIZACIÓN DE UNA TABLA DISPERSA ENCADENADA

El desarrollo de las operaciones de una tabla dispersa encadenada se basa en las operaciones del *TAD Lista*. Los pasos del algoritmo son:

1. Declarar los tipos de datos, ajustándose a los elementos de la aplicación.
2. Definir la función hash con la cual se van a obtener los índices de la tabla.
3. Inicializar la estructura, como un vector de listas vacías.
4. Definir la operación insertar un elemento. La inserción llamará a la función que añade un nodo como primer elemento de la lista.
5. Definir la operación de eliminar un elemento. En esta operación se busca el elemento dando su clave, antes de llamar a la función que elimina un nodo de la lista se pide conformidad ya que se retira el nodo de la lista.
6. Definir la función de búsqueda de un elemento, para ello que se pide la clave del elemento a buscar, si la búsqueda tiene éxito, devuelve la dirección del nodo, en caso contrario `NULL`.

Para hacer referencia a una aplicación, la tabla dispersa va a almacenar los socios del club de montaña *La Parra*. Cada socio se identifica con estos atributos: *número de socio, nombre completo, edad, sexo, fecha de alta*. Se elige como clave el número de socio, que es un entero en el rango de

101 a 1999. Actualmente, el club tiene 90 socios y por ello la tabla dispersa se declara con 97 posiciones, dado que no hay perspectivas de mucho crecimiento. El factor de carga, con este supuesto, estará próximo a uno.

La función hash utiliza el método de la multiplicación, y genera valores dispersos entre 0 y 96. Los tipos de datos:

```
struct fecha
{
   int dia;
   int mes;
   int anno;
};

struct TipoSocio
{
   int codigo;
   char* nom;
   int edad;
   struct fecha f;
};
typedef struct TipoSocio Socio;

struct TipoElemento
{
   Socio socio;
   struct TipoElemento* sgte;
};
typedef struct TipoElemento Elemento;

#define M 97

struct TipoTabla
{
   Elemento* tabla[M];
   int numElementos;
};
typedef struct TipoTabla TablaDispersa;
```

Función hash, método de multiplicación

En el Apartado 13.2.4 se explica este método de generar índices dispersos. El rango de la dispersión depende del tamaño de la tabla, M.

```
#define R 0.618034

int dispersion(int x)
{
   double t;
   int v;
   t = R * x - floor(R * x); /* parte decimal */
   v = (int) M * t;
   return v;
}
```

Inicializar la tabla dispersa

La operación inicial a realizar en la tabla, antes de dar de alta elementos, consiste en establecer cada posición del vector a la condición de lista vacía. Se realiza esta acción, con un bucle con tantas iteraciones como tamaño de la tabla.

```
void crearTabla(TablaDispersa* t)
{
  int k;
  for (k = 0; k < M; k++)
    t -> tabla[k] = NULL;
  t -> numElementos = 0;
}
```

13.7.1. Dar de alta un elemento en una tabla dispersa encadenada

Para dar de alta un elemento en la tabla, se determina primero el número de lista que le corresponde según la función de dispersión. A continuación se inserta como primer nodo de la lista. Se elige ese modo de inserción para que la eficiencia sea máxima, la complejidad de la operación es `O(1)`.

La operación «dar de alta» ha de crear un nuevo nodo, para ello se escribe una función auxiliar que reserva memoria para un nodo y asigna el elemento.

```
Elemento* crearNodo(Socio a)
{
  Elemento* n;
  n = (Elemento*)malloc(sizeof(Elemento));
  n -> socio = a;
  n -> sgte = NULL;
  return n;
}

void darAlta(TablaDispersa* t, Socio s)
{
  int posicion;
  Elemento* nuevo;

  posicion = dispersion(s.codigo);
  nuevo = crearNodo(s);
  nuevo -> sgte = t -> tabla[posicion];
  t -> tabla[posicion] = nuevo;
  t -> numElementos++;
}
```

13.7.2. Eliminar un elemento de la tabla dispersa encadenada

La supresión de un elemento se hace dando como entrada la clave del elemento, en el ejemplo el número de socio. Con la función de dispersión se obtiene el número de la lista para hacer una búsqueda secuencial dentro de la lista. Una vez encontrado el elemento, se muestran los campos y se pide

conformidad. La retirada del nodo se hace enlazando el nodo anterior con el nodo siguiente, para ello en la búsqueda se mantiene en un puntero la dirección del nodo anterior al actual.

La función que establece que se está de acuerdo con la supresión del nodo no se escribe, simplemente muestra el elemento y se pulsa una letra para indicar la acción a tomar.

```
int conforme(Socio a);

void darBaja(TablaDispersa*t, int codigo)
{
  Elemento* p;
  int posicion;

  posicion = dispersion(codigo);

  if (t -> tabla[posicion])
  /* no está vacía */
  {
    Elemento* anter;

    anter = NULL;
    p = t -> tabla[posicion];
    while ((p -> sgte != NULL) && p -> socio.codigo != codigo)
    {
      anter = p;
      p = p-> sgte;
    }
    if (p -> socio.codigo != codigo)
      printf("No se encuentra en la tabla el socio %d \n",
             codigo);
    else if (conforme (p -> socio))          /* se retira el nodo */
    {
      if (anter == NULL)                     /* primer nodo */
        t -> tabla[posicion] = p -> sgte;
      else
        anter -> sgte = p -> sgte;
      free(p);
      t -> numElementos--;
    }
  }
}
```

13.7.3. Buscar un elemento de la tabla dispersa encadenada

En la búsqueda de un elemento se siguen los mismos pasos que en la operación de eliminación. La operación devuelve la dirección del nodo (puntero) que contiene el código de búsqueda, o bien NULL si no se encuentra en la lista enlazada.

```
Elemento* localizar(TablaDispersa t, int codigo)
{
  Elemento* p = NULL;
  int posicion;
```

```
    posicion = dispersion(codigo);

    if (t.tabla[posicion])
    {
      p = t.tabla[posicion];
      while ((p -> sgte != NULL) &&p -> socio.codigo != codigo)
        p = p-> sgte;
      if (p -> socio.codigo != codigo)
        p = NULL;
    }
    return p;
  }
```

RESUMEN

Las tablas de dispersión son estructuras de datos en las cuales la complejidad de las operaciones *insertar*, *eliminar* y *buscar* es constante. Cuando se utilizan tablas de dispersión, los elementos que se guardan han de estar identificados por un campo clave, de tal forma que dos elementos distintos tengan claves distintas. La correspondencia entre un elemento y el índice de la tabla se hace con las funciones de transformación de claves, funciones hash, que convierten la clave de un elemento en un índice o posición de la tabla.

Dos criterios se deben seguir al elegir una *función hash*, `h(x)`. En primer lugar, la complejidad de `h(x)` que sea constante y fácil de evaluar. En segundo lugar, `h(x)` debe distribuir uniformemente las direcciones sobre el conjunto de índices posibles, de forma que se minimice el número de *colisiones*. El fenómeno de la *colisión* se produce, cuando para dos claves diferentes `c1` y `c2`, la función hash devuelve la misma dirección, `h(c1) = h(c2)`. Por muy buena distribución que realice `h(x)`, siempre existe la posibilidad de que colisionen dos claves, por ello el estudio del *direccionamiento disperso* se divide en dos partes: búsqueda de *funciones hash* y *resolución de colisiones*. Hay muchas funciones hash, la más popular es la *aritmética modular*, que aplica la *teoría de restos* para obtener valores dispersos dentro del rango determinado por el tamaño de la tabla, que debe elegirse como un número primo.

Se consideran dos modelos para resolver colisiones: *exploración* de direcciones y *hashing* enlazado. Las tablas hash con exploración resuelven las colisiones examinando una secuencia de posiciones de la tabla a partir de la posición inicial, determinada por `h(x)`. La exploración lineal lo hace secuencialmente hasta encontrar una posición vacía (caso de insertar); tiene el problema de la agrupación de elementos en posiciones contiguas, que afecta a la eficiencia. La exploración cuadrática examina las posiciones de la tabla $p, p+1, p+2^2, \ldots, p+i^2$ considerando la tabla como un array circular. Para asegurar la eficiencia de la exploración cuadrática, el tamaño de la tabla debe elegirse como un número primo y el *factor de carga* no superar el 50 por 100.

Las tablas dispersas enlazadas se basan en utilizar listas (cadenas), de tal forma que en cada lista se colocan los elementos que tienen la misma dirección hash. Las operaciones *insertar*, *buscar*, *eliminar*, determinan primero el índice de la lista que le corresponde con la función hash, a continuación aplican la operación correspondiente del *TAD Lista*. El factor de carga en las tablas enlazadas se aconseja sea próximo a 1, en el caso de crecer mucho puede empeorar la eficiencia de la búsqueda al crecer el número de nodos de las listas.

Una de las aplicaciones de las tablas hash está en los compiladores de lenguajes de programación. La tabla donde se guardan los identificadores del programa, *tabla de símbolos*, se implementa como una tabla hash. La clave es el *símbolo* que se transforma en un valor entero para aplicar alguna de las funciones hash.

EJERCICIOS

13.1. Una aplicación se espera maneje `50` elementos. ¿Cuál es el tamaño apropiado de una tabla hash que los almacene?

13.2. Una tabla hash se ha implementado con exploración lineal, considerando que actualmente el factor de carga es `0.30`. ¿Cuál es el número esperado de posiciones de la tabla que se prueban en una búsqueda de una clave, con éxito y sin éxito?

13.3. Para la secuencia de claves: `29, 41, 22, 31, 50, 19, 42, 38`, una tabla hash de tamaño `12`, y la función hash *aritmética modular*, mostrar las posiciones de almacenamiento suponiendo la exploración lineal.

13.4. Para la misma secuencia de claves del Ejercicio 13.3, igual tamaño de la tabla y la misma función hash, mostrar la secuencia de almacenamiento con la exploración cuadrática.

13.5. Se pretende utilizar como función hash el método *mitad del cuadrado*. Si el tamaño de la tabla es de `100`, encontrar los índices que le corresponden a las claves: `2134, 5231, 2212, 1011`.

13.6. Para resolución de colisiones se utiliza el método de doble función hash. Siendo la primera función el método *aritmética modular* y la segunda el método de la *multiplicación*. Encontrar las posiciones que ocupan los elementos con claves: `14, 31, 62, 26, 39, 44, 45, 22, 15, 1` en una tabla de tamaño `17`.

13.7. Escribir la función `estaVacia()` para determinar que una tabla dispersa con exploración no tiene ningún elemento activo.

13.8. Escribir la función `estaVacia()` para determinar que una tabla dispersa enlazada no tiene elementos asignados.

13.9. Una tabla de dispersión con exploración tiene asignada `n` elementos. Demostrar que si el factor de carga es λ, y la función hash distribuye uniformemente las claves, entonces se presentan `(n - 1)*` λ`/2` pruebas de posiciones de la tabla cuando se quiere insertar un elemento con una clave previamente insertada.

13.10. En el *hashing* enlazado la operación de insertar un elemento se ha realizado como primer elemento de la lista. Si la inserción en la lista enlazada se hace de tal forma que esté ordenada respecto la clave. ¿Qué ventajas y desventajas se tiene respecto a la eficiencia de las operaciones *insertar*, *buscar* y *eliminar*?

13.11. *Considérese esta otra estrategia para resolver colisiones en una tabla: en un vector auxiliar, vector de desbordamiento*, se sitúan los elementos que colisionan con una posición ya ocupada, en orden relativo al campo clave. ¿Qué ventajas y desventajas se pueden encontrar en cuanto a la eficiencia de las operaciones de la tabla?

13.12. Empleando una tabla de dispersión con exploración cuadrática para almacenar 1.000 cadenas de caracteres. El máximo valor que se quiere alcanzar del factor de carga es 0.6. Suponiendo que las cadenas tienen un máximo de 12 caracteres calcular:

a) El tamaño de la tabla.
b) La memoria total que necesita la tabla.

PROBLEMAS

13.1. Escribir la función `posicion()`, que tenga como entrada la clave de un elemento que representa los coches de alquiler de una compañía, devuelva una dirección dispersa

en el rango `0 .. 99`. Utilizar el método *mitad del cuadrado*, siendo la clave el número de matrícula pero sólo considerando los caracteres de orden par.

13.2. Escribir la función `plegado()`, para dispersar en el rango de 0 a 997 considerando como clave el número de la Seguridad Social. Utilizar el método de plegamiento.

13.3. Se desea almacenar en un archivo los atletas participantes en una carrera 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.

13.4. Las palabras de un archivo de texto se quieren mantener en una tabla hash con exploración cuadrática para poder hacer consultas rápidas. Los elementos de la tabla son la cadena con la palabra y el número de línea en el que aparece; sólo se consideran las 10 primeras ocurrencias de una palabra. Escribir un programa que lea el archivo, según se vaya capturando una palabra y su número de línea, se insertará en la tabla dispersa. Téngase en cuenta que una palabra puede estar ya en la tabla, si es así se añade el número de línea.

13.5. Para comparar los resultados teóricos de la eficiencia de las tablas de dispersión, escribir un programa que inserte 1.000 enteros generados aleatoriamente en una tabla dispersa con exploración lineal. Cada vez que se inserte un nuevo elemento contabilizar el número de posiciones de la tabla que se exploran. Calcular el número medio de posiciones exploradas para los factores de carga: 0.1, 0.2, ... 0.9.

13.6. 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 13.5.

13.7. Determinar la lista enlazada más larga (mayor número de nodos) cuando se utiliza una tabla hash *enlazada* para guardar 1.000 números enteros generados aleatoriamente. Escribir un programa que realice la tarea propuesta, considerar que el factor de carga es 1.

13.8. La realización de una tabla dispersa con exploración puede hacerse de tal manera que el tamaño de la tabla se establezca dinámicamente. Se utilizan las funciones de asignación dinámica de memoria para crear el vector de tamaño `m`. Con estos vectores dinámicos se puede hacer una *reasignación* de tal forma que si el factor de carga alcanza un determinado valor, se amplíe el tamaño en una cantidad determinada y se vuelvan a asignar los elementos, pero calculando un nuevo índice disperso ya que el tamaño de la tabla ha aumentado. Se pide escribir una función de *reasignación*.

13.9. Para una tabla dispersa con exploración, escribir una función que determine el numero de elementos dados de baja.

13.10. Partiendo de la función escrita en el Problema 13.9, añadir el código necesario para cuando el número de elementos eliminados supere el 10 por 100 se produzca una *reasignación* de tal forma que el factor de carga supere el 40 por 100.

13.11. En una tabla de dispersión enlazada, la eficiencia de la búsqueda disminuye según aumenta la longitud media de las listas enlazadas. Se quiere realizar una *reasignación* cuando la longitud media supere un factor determinado. La tabla dispersa, inicialmente, es de tamaño `m`, la función de dispersión es la aritmética modular: `h(x) = x modulo m`. Cada ampliación incrementa el tamaño de la tabla en 1, así en la primera el número de posiciones será `m+1`, en el rango de `0` a `m`; por 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 modulo 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 modulo 2*(m+1)`. Así sucesivamente se va ampliando la estructura que almacena los elementos de una tabla dispersa enlazada.

Escribir las funciones que implemente la operación de insertar siguiendo la estrategia indicada.

13.12. Escribir un programa que utilice la tabla hash enlazada descrita en el Problema 13.11 para guardar 1.000 números enteros generados aleatoriamente. Inicialmente, el tamaño de la tabla es 50, cada vez que la longitud media de las cadenas (listas enlazadas) sea 4.5 se debe ampliar la tabla dispersa según el método descrito en 13.11.

PARTE IV

ESTRUCTURAS DE DATOS AVANZADAS (ÁRBOLES Y GRAFOS)

14

ÁRBOLES. ÁRBOLES BINARIOS Y ÁRBOLES ORDENADOS

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Estructurar datos en orden jerárquico.
- Conocer la terminología básica relativa a árboles.
- Distinguir los diferentes tipos de árboles binarios.
- Recorrer un árbol binario de tres formas diferentes.
- Reconocer la naturaleza recursiva de las operaciones con árboles.
- Representar un árbol binario con una estructura enlazada.
- Evaluar una expresión algebraica utilizando un árbol binario.
- Definir un árbol binario ordenado (de búsqueda).
- Construir un árbol binario de búsqueda según su naturaleza.

CONTENIDO

14.1. Árboles generales.
14.2. Árboles binarios. TAD árbol binario.
14.3. Estructura de un árbol binario.
14.4. Operaciones en árboles binarios.
14.5. Árboles de expresiones.
14.6. Recorrido de un árbol.
14.7. Árbol binario de búsqueda.
14.8. Operaciones en árboles binarios de búsqueda.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Jerarquía.
- Raíz.
- Hoja.
- Árbol.
- Árbol binario.
- Árbol binario de búsqueda.
- Subárbol.
- Recorrido de un árbol.
- Nodo.
- *Preorden.*
- *Postorden.*
- Enorden.
- Conceptos teóricos (nivel, profundidad, raíz, hoja, rama...).

INTRODUCCIÓN

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras *no lineales*, al contrario que los arrays y las listas enlazadas, que constituyen *estructuras lineales*.

Los árboles son 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, proceso de textos y algoritmos de búsqueda.

En el capítulo se estudiará el concepto de árbol general y los tipos de árboles más usuales, binario y binario de búsqueda o árbol ordenado. Asimismo se estudiarán algunas aplicaciones típicas del diseño y la construcción de árboles.

14.1. ÁRBOLES GENERALES

Intuitivamente el concepto de árbol implica una estructura en la cual los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas. El árbol genealógico es el ejemplo típico más representativo del concepto de árbol general. La Figura 14.1 representa un ejemplo de árbol general; gráficamente puede verse como un árbol invertido, la raíz en la parte más alta de la que salen ramas que llegan a las hojas, que están en la parte baja.

Figura 14.1. Estructura jerárquica tipo árbol.

Un **árbol** consta de un conjunto finito de elementos, denominados **nodos,** y un conjunto finito de líneas dirigidas, denominadas **ramas,** que conectan los nodos. El número de ramas asociado con un nodo es el **grado** del nodo.

Definición 1: Un **árbol** consta de un conjunto finito de elementos, llamados nodos, y un conjunto finito de líneas dirigidas, llamadas ramas, que conectan los nodos.

Definición 2: Un **árbol** es un conjunto de uno o más nodos tales que:

1. Hay un nodo diseñado especialmente llamado **raíz.**
2. Los nodos restantes se dividen en $n \geqslant 0$ conjuntos disjuntos, $T_1 \ldots T_n$, en donde cada uno de estos conjuntos es un árbol. A $T_1, T_2, \ldots, T_n$ se les denomina *subárboles* del raíz.

Si un árbol no está vacío, entonces el primer nodo se llama **raíz.** Obsérvese en la definición 2 que el árbol ha sido definido de modo recursivo ya que los subárboles se definen como árboles. La Figura 14.2 muestra un árbol general.

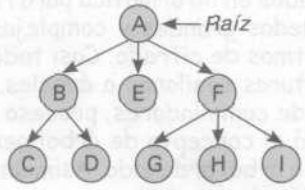


Figura 14.2. Árbol.

Terminología

Además del raíz existen muchos términos utilizados en la descripción de los atributos de un árbol. En la Figura 14.3, el nodo A es el raíz. Utilizando el concepto de árboles genealógicos, un nodo puede ser considerado como **padre** si tiene nodos sucesores.

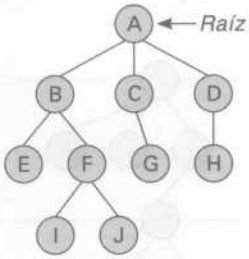


Figura 14.3. Árbol general.

Estos nodos sucesores se llaman **hijos.** Por ejemplo, el nodo B es el padre de los hijos E y F. El padre de H es el nodo D. Un árbol puede representar diversas generaciones en la familia. Los hijos de un nodo y los hijos de estos hijos se llaman **descendientes** y el padre y abuelos de un nodo son sus **ascendientes.** Por ejemplo, los nodos E, F, I y J son descendientes de B. Cada nodo no raíz tiene un único padre y cada padre tiene cero o más nodos hijos. Dos o más nodos con el mismo padre se llaman **hermanos.** Un nodo sin hijos, tales como E, I, J, G y H se llaman nodos **hoja.**

El **nivel** de un nodo es su distancia al nodo raíz. El raíz tiene una distancia cero de sí misma, por lo que se dice que el raíz está en el nivel 0. Los hijos del raíz están en el nivel 1, sus hijos están en el nivel 2 y así sucesivamente. Un hecho importante que se aprecia entre los niveles de nodos es la relación entre niveles y *hermanos*. Los hermanos están siempre al mismo nivel, pero no todos los nodos de un mismo nivel son necesariamente hermanos. Por ejemplo, en el nivel 2 (Fig. 14.4), C y D son hermanos, al igual que lo son G, H e I, pero D y G no son hermanos ya que ellos tienen diferentes padres.

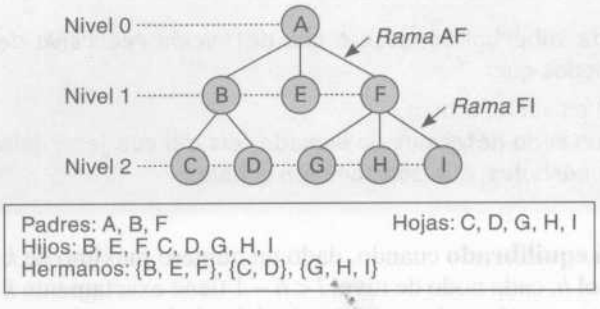


Figura 14.4. Terminología de árboles.

Existen varias formas de dibujar los atributos de los árboles y sus nodos. Un **camino** es una secuencia de nodos en los que cada nodo es adyacente al siguiente. Cada nodo del árbol puede ser alcanzado (se llega a él) siguiendo un único camino que comienza en el raíz. En la Figura 14.4, el camino desde el raíz a la hoja I, se representa por AFI. Incluye dos ramas distintas AF y FI.

La **altura** o **profundidad** de un árbol es el nivel de la hoja del camino más largo desde la raíz más uno. Por definición[1] la altura de un árbol vacío es 0. La Figura 14.4 contiene nodos en tres niveles: 0, 1 y 2; su altura es 3.

[1] También se suele definir la profundidad de un árbol como el nivel máximo de cada nodo. En consecuencia, la profundidad del nodo raíz es 0, la de su hijo 1, etc. Las dos terminologías son aceptadas.

Definición

El nivel de un nodo es su distancia desde el nodo raíz. La altura de un árbol es el nivel de la hoja del camino más largo desde el raíz más uno.

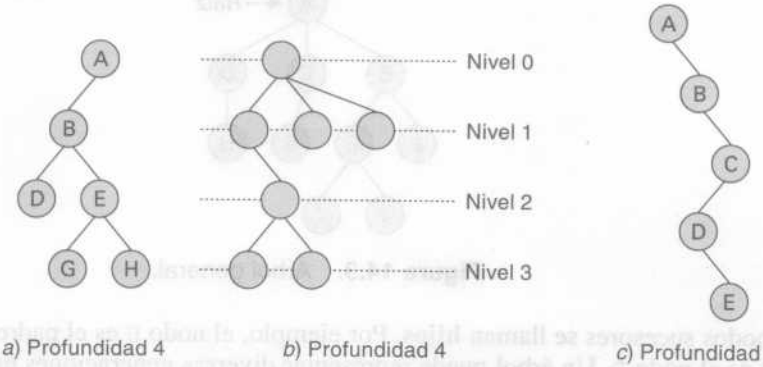


Figura 14.5. Árboles de profundidades diferentes.

Un árbol se divide en subárboles. Un **subárbol** es cualquier estructura conectada por debajo del raíz. Cada nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos los descendientes del nodo. El primer nodo de un subárbol se conoce como el raíz del subárbol y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subárboles. En la Figura 14.4, `BCD` es un subárbol al igual que `E` y `FGHI`. Obsérvese que por esta definición, un nodo simple es un subárbol. Por consiguiente, el subárbol `B` se puede dividir en subárboles `C` y `D` mientras que el subárbol `F` contiene los subárboles `G`, `H` e `I`. Se dice que `G`, `H`, `I`, `C` y `D` son subárboles sin descendientes.

Definición recursiva

El concepto de subárbol conduce a una definición recursiva de un árbol. Un árbol es un conjunto de nodos que:

1. O bien es vacío, o bien,
2. tiene un nodo determinado llamado raíz del que jerárquicamente descienden cero o más subárboles, que son también árboles.

Un árbol **está equilibrado** cuando, dado un número máximo de k hijos para cada nodo y siendo la **altura del árbol** h, cada nodo de nivel $l < h - 1$ tiene exactamente k hijos. El árbol está **equilibrado perfectamente** cuando cada nodo de nivel $l < h$ tiene exactamente k hijos.

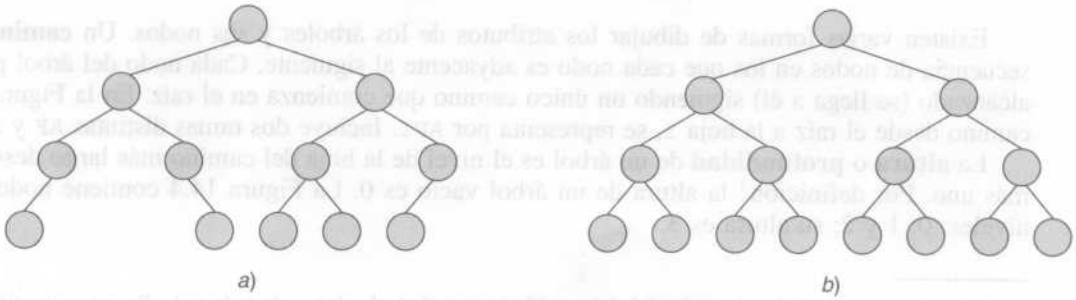


Figura 14.6. *a)* Un árbol equilibrado. *b)* Un árbol perfectamente equilibrado.

14.1.1. Representación gráfica de un árbol

Cuando se ha de representar en papel un árbol, existen tres formas diferentes de representación.

Representación como árbol invertido

Es el diagrama o carta de organización utilizada hasta ahora en las diferentes figuras. El nódo raíz se encuentra en la parte más alta de una jerarquía de la que descienden ramas que van a parar a los nodos hijos, y así sucesivamente. La Figura 14.7 muestra esta representación para una descomposición de una computadora.

Figura 14.7. Árbol general (computadora).

Representación en niveles de profundidad

Este tipo de representación es el utilizado para representar sistemas jerárquicos en modo texto o número en situaciones tales como facturación, gestión de *stocks* en almacenes, etc.

Por ejemplo, en la Figura 14.8 se aprecia la descomposición de una computadora en sus diversos componentes representada por *niveles de profundidad*.

Número código	Descripción
501	Computadora
501-11	Monitor
...	...
501-21	CPU
501-211	Controlador
501-212	ALU
...	...
501-219	ROM
501-31	Teclado
...	...
501-41	Periféricos
501-411	Escáner
501-412	Impresora
501-51	Discos
501-511	3½
501-512	CD-ROM
501-513	DVD

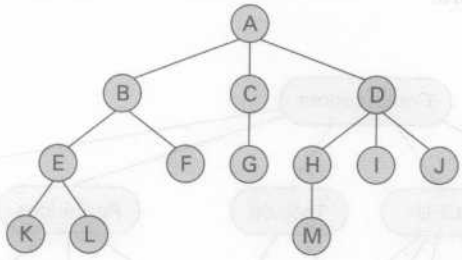
Figura 14.8. Árbol en nivel de profundidad (computadora).

Representación de lista

Otro formato utilizado para representar un árbol es la lista entre paréntesis. Ésta es la notación utilizada con expresiones algebraicas. En esta representación, cada paréntesis abierto indica el comienzo de un nuevo nivel; cada paréntesis cerrado completa un nivel y se mueve hacia arriba un nivel en el árbol. La notación en paréntesis correspondiente al árbol de la Figura 14.2: `A(B (C, D), E, F, (G, H, I)).`

Ejemplo 14.1

Convertir el árbol general siguiente en representación en lista.



La solución: `A(B(E(K, L), F), C(G), D(H(M), I, J))).`

14.1.2. Resumen de definiciones

1. Dado un conjunto `E` de elementos:
   - Un árbol puede estar *vacó* ; es decir, no contiene ningún elemento.
   - Un árbol *no vacó* puede constar de un único elemento $e \in E$ denominado un **nodo raíz.**
   - Un árbol consta de un nodo raíz $e \in E$, conectado por arcos directos a un número finito de otros árboles.
2. Definiciones:
   - El primer nodo de un árbol, normalmente dibujado en la posición superior, se denomina **raíz** del árbol.
   - Las flechas que conectan un nodo a otro se llaman **arcos** o **ramas.**
   - Los **nodos terminales,** esto es, nodos de los cuales no se deduce ningún nodo, se denominan **hojas.**
   - Los nodos que no son hojas se denominan **nodos internos** o **nodos no terminales.**
   - En un árbol una rama va de un nodo n_1 a un nodo n_2, se dice que n_1 es el padre de n_2 y que n_2 es un **hijo** de n_1.
   - n_1 se llama **ascendiente** de n_2 si n_1 es el padre de n_2 o si n_1 es el padre de un ascendiente de n_2.
   - n_2 se llama **descendiente** de n_1 si n_1 es un ascendiente de n_2.
   - Un **camino** de n_1 a n_2 es una secuencia de arcos contiguos que van de n_1 a n_2.
   - La **longitud** de un camino es el número de arcos que contiene, o de forma equivalente, el número de nodos del camino menos uno.
   - El **nivel** de un nodo es la longitud del camino que lo conecta al nodo raíz.
   - La **profundidad** o **altura** de un árbol es la longitud del camino más largo que conecta el raíz a una hoja más uno.
   - Un **subárbol** de un árbol es un subconjunto de nodos del árbol, conectados por ramas del propio árbol, esto es, a su vez un árbol.

- Sea `S` un subárbol de un árbol `A`: si para cada nodo n de `SA`, `SA` contiene también todos los descendientes de n en `A`. `SA` se llama un **subárbol completo** de `A`.
- Un árbol está **equilibrado** cuando, dado un número máximo k de hijos de cada nodo y la **altura del árbol** h, cada nodo de nivel $k < h - 1$ tiene exactamente k hijos. El árbol está equilibrado perfectamente entre cada nodo de nivel $l < h$ tiene exactamente k hijos.

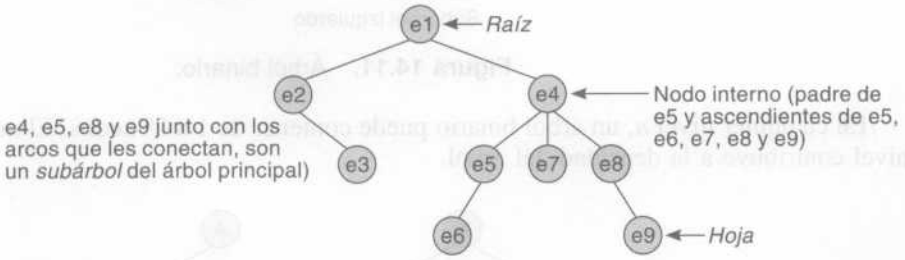


Figura 14.9. Un árbol y sus nodos.

14.2. ÁRBOLES BINARIOS. TAD ÁRBOL BINARIO

Un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos subárboles. En un árbol binario, cada nodo puede tener cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como *hijo izquierdo* y el nodo de la derecha como *hijo derecho*.

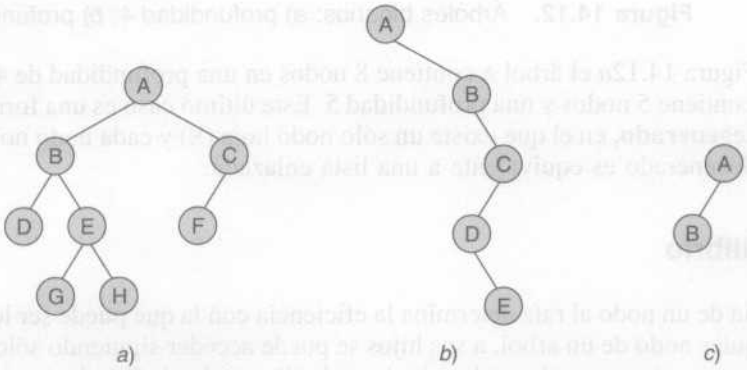


Figura 14.10. Árboles binarios.

Nota

Un árbol binario no puede tener más de dos subárboles.

Un árbol binario es una estructura recursiva. Cada nodo es el raíz de su propio subárbol y tiene hijos, que son raíces de árboles llamados los subárboles derecho e izquierdo del nodo, respectivamente. Un árbol binario se divide en tres subconjuntos disjuntos:

`{R}`	Nodo raíz
$\{I_1, I_2, \ldots, I_n\}$	Subárbol izquierdo de `R`
$\{D_1, D_2, \ldots, D_n\}$	Subárbol derecho de `R`

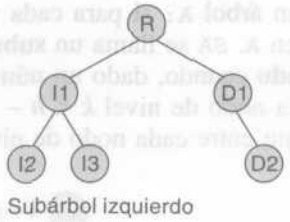


Figura 14.11. Árbol binario.

En cualquier nivel n, un árbol binario puede contener de 1 a 2^n nodos. El número de nodos por nivel contribuye a la densidad del árbol.

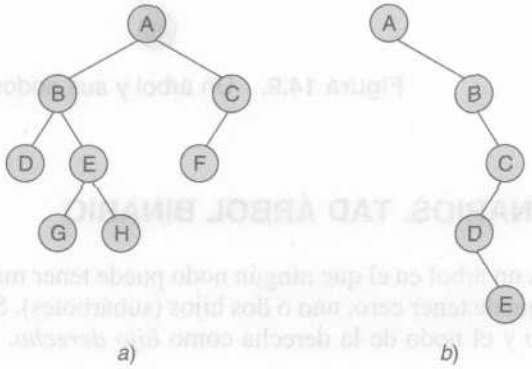


Figura 14.12. Árboles binarios: *a*) profundidad 4; *b*) profundidad 5.

En la Figura 14.12*a* el árbol A contiene 8 nodos en una profundidad de 4, mientras que la Figura 14.12*b* contiene 5 nodos y una profundidad 5. Este último caso es una forma especial, denominado **árbol degenerado,** en el que existe un sólo nodo hoja (E) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

14.2.1. Equilibrio

La distancia de un nodo al raíz determina la eficiencia con la que puede ser localizado. Por ejemplo, dado cualquier nodo de un árbol, a sus hijos se puede acceder siguiendo sólo un camino de bifurcación o de ramas, el que conduce al nodo deseado. De modo similar, los nodos a nivel 2 de un árbol sólo pueden ser accedidos siguiendo sólo dos ramas del árbol.

La característica anterior nos conduce a una característica muy importante de un árbol binario, su **balance** o **equilibrio.** Para determinar si un árbol está equilibrado, se calcula su factor de equilibrio. El **factor de equilibrio** de un árbol binario es la diferencia en altura entre los subárboles derecho e izquierdo. Si la altura del subárbol izquierdo es h_I y la altura del subárbol derecho es h_D, entonces el factor de equilibrio del árbol B se determina por la siguiente fórmula:

$$B = h_D - h_I$$

Utilizando esta fórmula, el equilibrio del nodo raíz los árboles de la Figura 14.10 son: *a*) −1, *b*) 4 y *c*) −1.

Un árbol está **perfectamente equilibrado** si su equilibrio o balance es *cero* y sus subárboles son también perfectamente equilibrados. Dado que esta definición ocurre raramente se aplica una definición alternativa. Un árbol binario está equilibrado si la altura de sus subárboles difiere en no más de uno (su factor de equilibrio es −1, 0, +1) y sus subárboles son también equilibrados.

14.2.2. Árboles binarios completos

Un árbol binario **completo** de profundidad n es un árbol en el que 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 lleno es un árbol binario que tiene el máximo número de entradas para su altura. Esto sucede cuando el último nivel está lleno. La Figura 14.13 muestra un árbol binario completo; el árbol de la Figura 14.14*b* se corresponde con uno lleno.

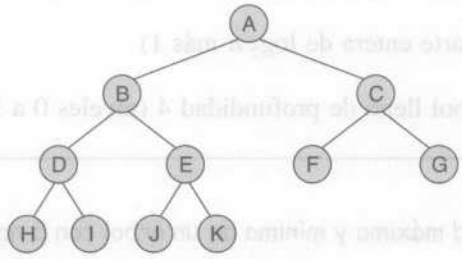
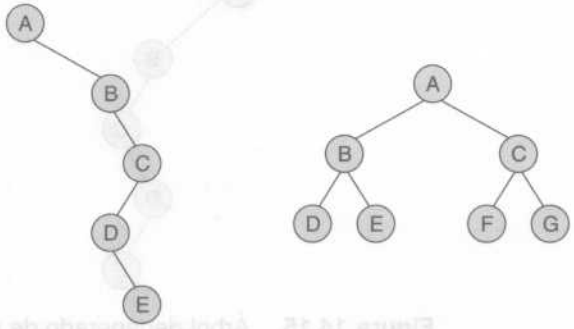


Figura 14.13. Árbol completo (profundidad 4).



a) Degenerado (profundidad 5) *b*) Lleno (profundidad 3)

Figura 14.14. Clasificación de árboles binarios: *a*) degenerado; *b*) lleno.

El último caso de árbol es un tipo especial denominado **árbol degenerado** en el que hay un solo nodo hoja (E) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

Los árboles binarios y llenos de profundidad k+1 proporcionan algunos datos matemáticos que es necesario comentar. En cada caso, existe un nodo (2^0) al nivel 0 (raíz), dos nodos (2^1) a nivel 1, cuatro nodos (2^2) a nivel 2, etc. A través de los primeros $k - 1$ niveles se puede demostrar, considerando la suma de los términos de una progresión geométrica de razón 2, que hay $2^k - 1$ nodos.

$$1 + 2 + 4 + \cdots + 2^{k-1} = 2^k - 1$$

A nivel k, el número de nodos adicionados para un árbol completo está en el rango de un mínimo de 1 a un máximo de 2^k (lleno). Con un árbol lleno, el número de nodos es

$$1 + 2 + 4 + \cdots + 2^{k-1} + 2^k = 2^{k+1} - 1$$

El número de nodos n en un árbol binario completo de profundidad k+1 (0 a k niveles) cumple la inigualdad:

$$2^k \leqslant n \leqslant 2^{k+1} - 1 < 2^{k+1}$$

Aplicando logaritmos a la ecuación de la desigualdad anterior:

$$k \leqslant \log_2 (n) < k + 1$$

Se deduce que la altura o profundidad de un árbol binario completo de n nodos es:

$h = \lfloor \log_2 n \rfloor + 1$ (parte entera de $\log_2 n$ más 1)

Por ejemplo, un árbol lleno de profundidad 4 (niveles 0 a 3) tiene $2^4 - 1 = 15$ nodos.

Ejemplo 14.2

Calcular la profundidad máxima y mínima de un árbol con 5 nodos.

La profundidad máxima de un árbol con 5 nodos es 5, se corresponde con un árbol degenerado.

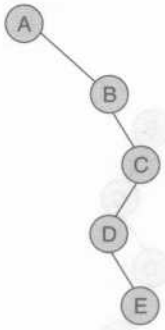


Figura 14.15. Árbol degenerado de raíz A.

La profundidad mínima h (número de niveles más uno) de un árbol con 5 nodos, teniendo en cuenta la inecuación del número de nodos de un árbol binario completo:

$$k \leqslant \log_2 (5) < k + 1$$

$\log_2 (5) = 2,32$ y la profundidad $h = 3$.

Ejemplo 14.3

Suponiendo que se tienen $n = 10.000$ elementos que van a ser los nodos de un árbol binario completo, determinar la profundidad del árbol.

En el árbol binario completo con n nodos, la profundidad del árbol es el valor entero de $log_2 n + 1$, que es a su vez la distancia del camino más largo desde el raíz a un nodo más uno.

$$\text{Profundidad} = \text{int} (\log_2 10000) + 1 = \text{int} (13.28) + 1 = 14$$

14.2.3. Especificación del TAD árbol binario

Un árbol binario es aquel en el que cada nodo tiene como máximo dos descendientes o dos ramas, por lo que el grado es dos. Las operaciones básicas que sirven para definir un árbol binario y poder manipular su contenido son las siguientes (no todas ellas se implementan al definir el árbol):

Tipo de dato:	Dato que se almacena en los nodos del árbol.
Operaciones:	
CrearArbol	Inicia el árbol como vacío.
Construir	Crea un árbol con un elemento raíz y dos ramas, izquierda y derecha que son, a su vez, árboles.
EsVacio	Comprueba si el árbol no tiene nodos.
Raiz	Devuelve el nodo raíz.
Izquierdo	Obtiene la rama o subarbol izquierdo de un árbol dado.
Derecho	Obtiene la rama o subarbol derecho de un árbol dado.
Borrar	Elimina del árbol el nodo con un elemento determinado.
Pertenece	Determina si un elemento se encuentra en el árbol.

14.3. ESTRUCTURA DE UN ÁRBOL BINARIO

La estructura de un árbol binario se construye con nodos. Cada nodo debe contener el campo *dato* (datos a almacenar) y dos campos de tipo *puntero*, uno al subárbol izquierdo y otro al subárbol derecho, que se conocen como **puntero izquierdo** (**izquierdo**, **izdo**) y **puntero derecho** (**derecho**, **dcho**) respectivamente. Un valor `NULL` indica un árbol o un subárbol vacío.

Figura 14.16. Representación gráfica de los campos de un nodo.

El tipo de datos correspondiente a la estructura de un nodo de un árbol binario es el siguiente:

```
Nodo
  subarbolIzquierdo    < puntero a Nodo>
  datos                < Tipodato>
  subarbolDerecho      < puntero a Nodo>
Fin Nodo
```

La Figura 14.17 muestra la representación enlazada de dos árboles binarios de raíz `A`. El primero es un árbol degenerado a la izquierda; el segundo es un árbol binario completo de profundidad 4.

La Figura 14.18 se corresponde con un árbol binario y su estructura en nodos. Se puede observar que los nodos de un árbol binario que son hojas se caracterizan por tener sus dos campos de enlace a `NULL`.

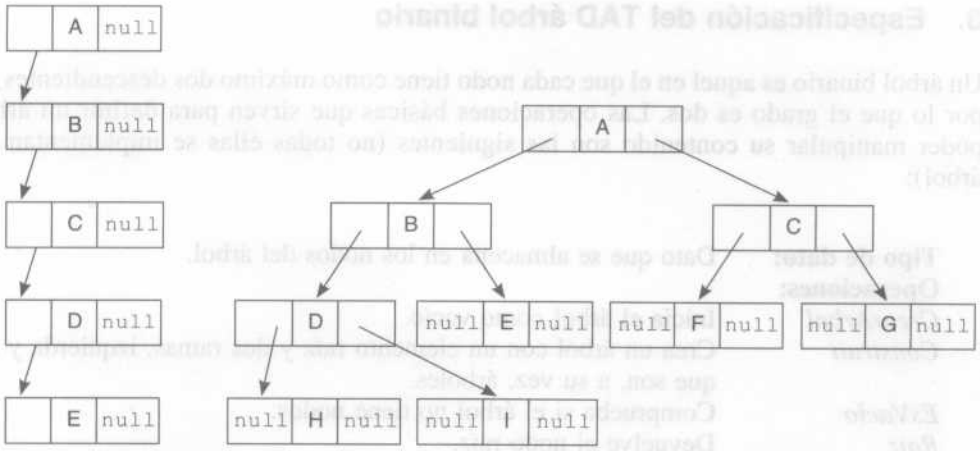


Figura 14.17. Representación enlazada de dos árboles binarios.

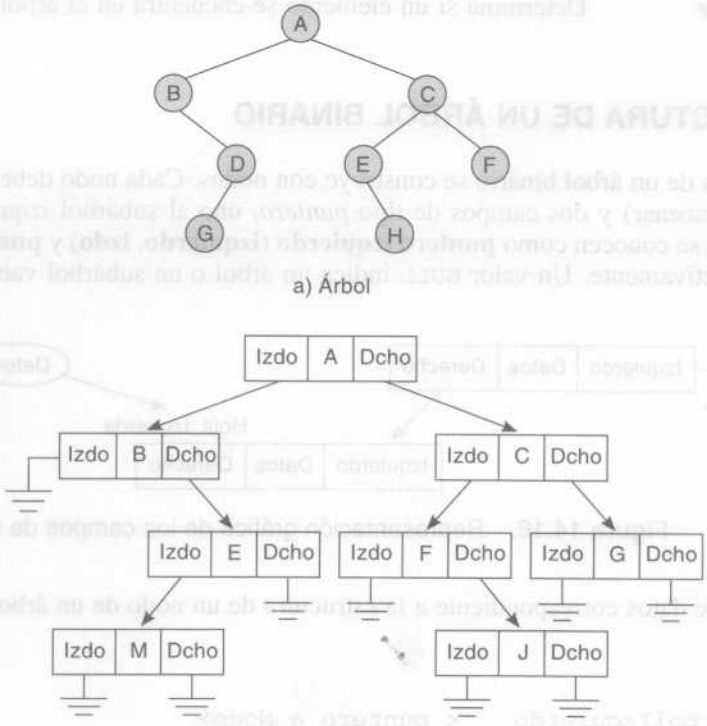
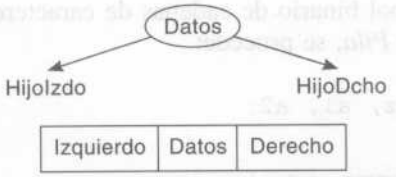


Figura 14.18. Árbol binario y su estructura en nodos.

14.3.1. Representación de un nodo

El nodo se representa mediante una estructura `(struct)` para así agrupar a todos los campos de que consta. Se supone que el nodo tiene los campos `dato`, `izdo` (rama izquierda) y `dcho` (rama derecha). El tipo de dato de los elementos se generaliza como `tipoElemento`.



Representación

```
typedef int TipoElemento;          Cualquier tipo de dato

typedef struct nodo {
  TipoElemento dato;
  struct nodo *izdo, *dcho;
}Nodo;

typedef Nodo* ArbolBinario;
```

14.3.2. Creación de un árbol binario

A partir del nodo raíz de un árbol se puede acceder a los demás nodos del árbol, por ello el puntero que permite acceder al árbol es el que referencia al raíz. Las ramas izquierda y derecha son a su vez árboles binarios que tienen su raíz, y así recursivamente hasta llegar a las hojas del árbol.

La formación del árbol pasa por la creación de cada uno de los nodos y el enlace con el correspondiente nodo *padre*. Para crear un nodo de un árbol binario, se reserva memoria para el nodo, se asigna el dato al campo correspondiente y se inicializa los punteros `izdo`, `dcho` a `NULL`.

```
ArbolBinario crearNodo(TipoElemento x)
{
  ArbolBinario a;
  a = (ArbolBinario) malloc(sizeof(Nodo));
  a -> dato = x;
  a -> dcho = a ->izdo = NULL;
  return a;
}
```

La función `nuevoArbol()` crea un árbol cuya raíz es un nodo con el campo dato el que se pasa como tercer argumento, la rama izquierda y derecha del árbol se pasan como segundo y cuarto argumentos.

```
void nuevoArbol(ArbolBinario* raiz, ArbolBinario ramaIzqda,
                TipoElemento x, ArbolBinario ramaDrcha)
{
  *raiz = crearNodo(x);
  (*raiz) -> izdo = ramaIzqda;
  (*raiz) -> dcho = ramaDrcha;
}
```

Así, para crear un árbol binario de cadenas de caracteres utilizando un esquema secuencial y con la estructura auxiliar *Pila*, se procede:

```
ArbolBinario raiz, a1, a2;
Pila pila;

nuevoArbol(&a1, NULL,"Maria",NULL);
nuevoArbol(&a2, NULL,"Rodrigo",NULL);
nuevoArbol(&raiz, a1,"Esperanza",a2);
insertar(&pila, raiz);

nuevoArbol(&a1, NULL,"Anyora",NULL);
nuevoArbol(&a2, NULL,"Abel",NULL);
nuevoArbol(&raiz, a1,"M. Jesus",a2);
insertar(&pila, raiz)

a2 = quitar(&pila);
a1 = quitar(&pila);
nuevoArbol(&raiz, a1,"Esperanza",a2);
```

La Figura 14.19 muestra el árbol formado.

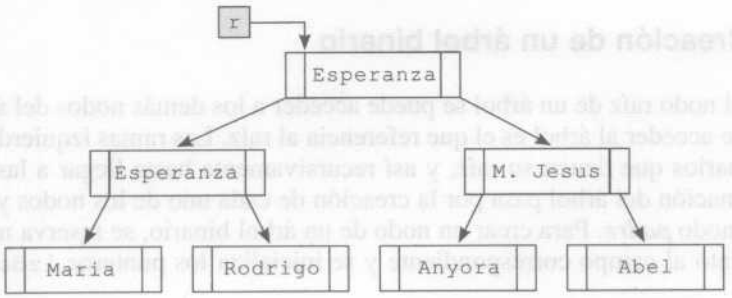


Figura 14.19. Árbol binario de cadenas de caracteres.

14.4. OPERACIONES EN ÁRBOLES BINARIOS

Una vez que se tiene creado un árbol binario, se pueden realizar diversas operaciones sobre él. El hacer uso de una operación u otra dependerá de la aplicación que se le quiera dar al árbol. Algunas de las operaciones típicas que se realizan en árboles binarios son:

- Determinar su altura.
- Determinar su número de elementos.
- Hacer una copia.
- Visualizar el árbol binario en pantalla o en impresora.
- Determinar si dos árboles binarios son idénticos.
- Borrar (eliminar el árbol).
- Si es un árbol de expresión, evaluar la expresión.

Todas estas operaciones se pueden realizar recorriendo el árbol binario de un modo sistemático. El recorrido de un árbol es la operación de visita al árbol, o lo que es lo mismo, la visita a cada nodo del árbol una vez y sólo una. La visita de un árbol es necesaria en muchas ocasiones, por ejemplo, si se desea imprimir la información contenida en cada nodo. Existen diferentes formas de visitar o recorrer un árbol que se estudiarán más adelante.

14.5. ÁRBOLES DE EXPRESIÓN

Una aplicación muy importante de los árboles binarios son los *árboles de expresión*. Una **expresión** es una secuencia de *tokens* (componentes de léxicos que siguen unas reglas establecidas). Un *token* puede ser o bien un operando o bien un operador.

La Figura 14.20 representa la expresión infija `a * (b + c) + d` junto a su árbol de expresión. El nombre de *infija* es debido a que los operadores se sitúan *entre* los operandos. En una primera observación vemos que los paréntesis de la expresión no aparecen en el árbol y esto resulta muy interesante para la evaluación de la expresión.

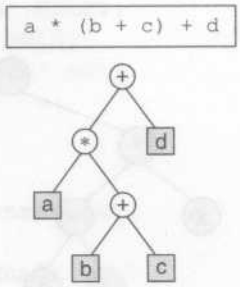


Figura 14.20. Una expresión infija y su árbol de expresión.

Un **árbol de expresión** es un árbol binario con las siguientes propiedades:

1. Cada hoja es un operando.
2. Los nodos raíz y nodos internos son operadores.
3. Los subárboles son subexpresiones en las que el nodo raíz es un operador.

Los árboles binarios se utilizan para representar expresiones en memoria; esencialmente, en compiladores de lenguajes de programación. La Figura 14.21 muestra un árbol binario de expresiones para la expresión aritmética `(a + b) * c`.

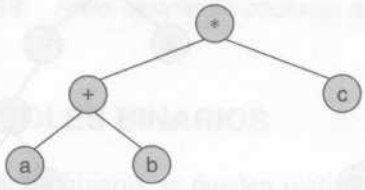


Figura 14.21. Árbol binario de expresiones que representa `(a + b) * c`.

Obsérvese que los paréntesis no se almacenan en el árbol pero están implicados en la forma del árbol. 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. Cada operando puede ser una letra (`x`, `y`, `a`, `b`, etc.) o una subexpresión representada como un subárbol. En la Figura 14.22 se

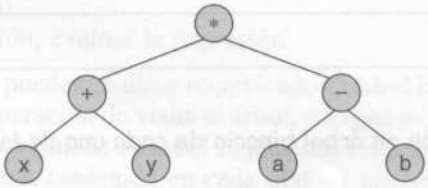


Figura 14.22. Árbol de expresión `(x + y) * (a - b)`.

puede ver cómo el operador que está en la raíz es *, su subárbol izquierdo representa la subexpresión `(x + y)` y su subárbol derecho representa la subexpresión `(a - b)`. El nodo raíz del subárbol izquierdo contiene el operador (+) de la subexpresión izquierda y el nodo raíz del subárbol derecho contiene el operador (–) de la subexpresión derecha. Todos los operandos letras se almacenan en nodos hojas. Utilizando el razonamiento anterior, la expresión almacenada como:

```
(x * (y - z)) + (a - b)
```

en donde se han insertado paréntesis alrededor de las subexpresiones del árbol, forma el árbol binario de la Figura 14.23.

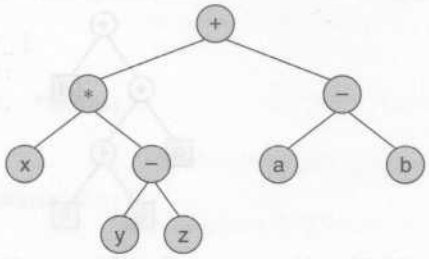
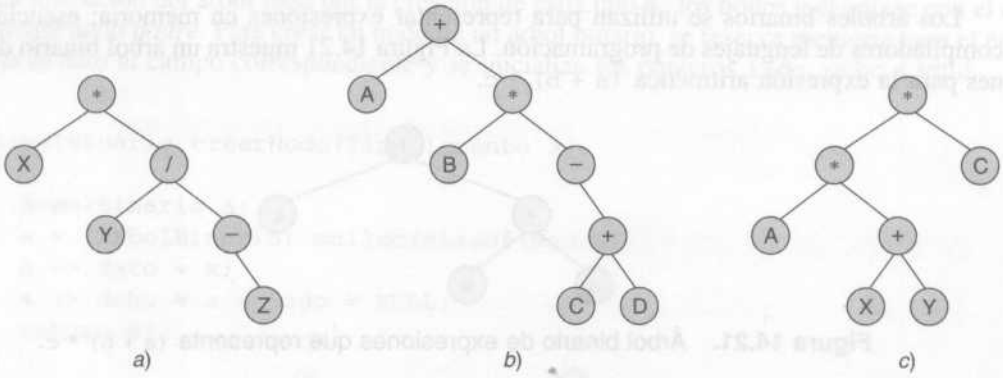


Figura 14.23. Árbol de expresión `(x * (y - z)) + (a - b)`.

Ejemplo 14.4

Deducir las expresiones que representan los siguientes árboles binarios.



Soluciones

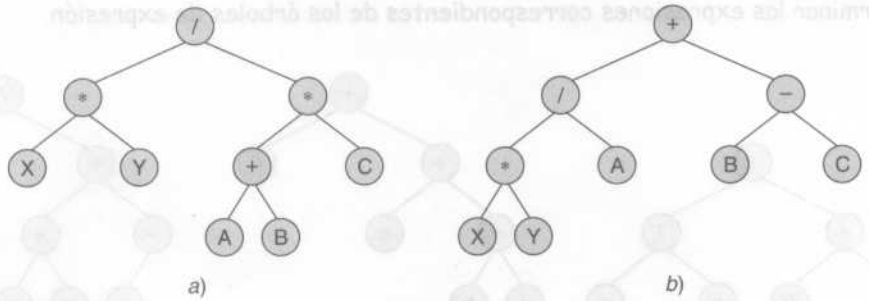
a) `X * Y/(-Z)`
b) `A + B * (-(C+D))`
c) `A * (X + Y)) * C`

Ejemplo 14.5

Dibujar la representación en árbol binario de cada una de las siguientes expresiones:

a) `X * Y/(A + B) * C`
b) `X * Y/A + B - C`

Soluciones



14.5.1. Reglas para la construcción de árboles de expresión

Los árboles de expresiones se utilizan en las computadoras para evaluar expresiones usadas en programas. El algoritmo más sencillo para construir un árbol de expresión es aquel que lee una expresión completa que contiene paréntesis en la misma. Una expresión con paréntesis es aquella en que:

1. La prioridad se determina sólo por paréntesis.
2. La expresión completa se sitúa entre paréntesis.

A fin de ver la prioridad en las expresiones, considérese la expresión

```
a * c + e / g - (b + d)
```

Los operadores con prioridad más alta son * y /; es decir,

```
(a * c) + (e / g) - (b + d)
```

Los operadores que siguen en orden de prioridad son + y −, que se evalúan de izquierda a derecha. Por consiguiente, se puede escribir

```
((a * c) + (e / g)) - (b + d)
```

Por último, la expresión completa entre paréntesis será

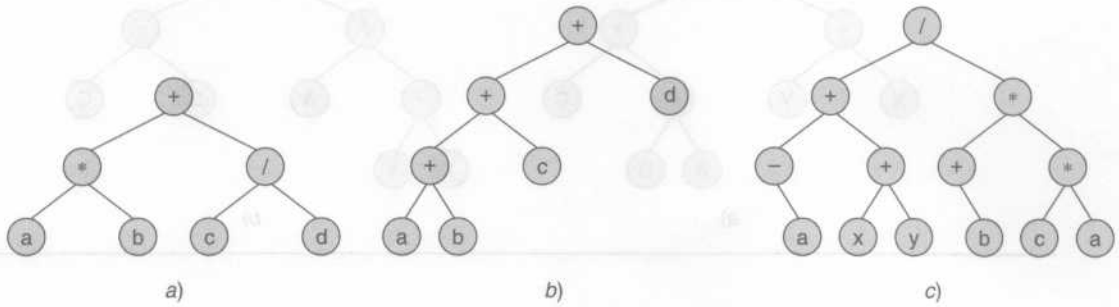
```
(((a * c) + (e / g)) - (b + d))
```

El algoritmo para la construcción de un árbol de expresión puede expresarse en los siguientes pasos:

1. La primera vez que se encuentra un paréntesis a izquierda, crea un nodo y lo hace en el nodo raíz. Se le llama *nodo actual* y se sitúa su puntero en una pila.
2. Cada vez que se encuentre un nuevo paréntesis a izquierda, crear un nuevo nodo. Si el nodo actual no tiene un hijo izquierdo, hacer el nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho. Hacer el nuevo nodo el nodo actual y situar su puntero en una pila.
3. Cuando se encuentra un operando, crear un nuevo nodo y asignar el operando a su campo de datos. Si el nodo actual no tiene un hijo izquierdo, hacer el nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho.
4. Cuando se encuentra un operador, sacar un puntero de la pila y situar el operador en el campo datos del nodo del puntero.
5. Ignorar paréntesis derecho y blancos.

Ejemplo 14.6

Determinar las expresiones correspondientes de los árboles de expresión.



Las soluciones correspondientes son:

a) `((a * b) + ( c / d))`
b) `(((a + b) + c) + d)`
c) `(((-a) + (x + y)) / ((+b) * (c * a)))`

14.6. RECORRIDO DE UN ÁRBOL

Para visualizar o consultar los datos almacenados en un árbol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. Al contrario que las listas enlazadas, los árboles binarios no tienen realmente un primer valor, un segundo valor, tercer valor, etc. Se puede afirmar que el nodo raíz viene el primero, pero ¿quién viene a continuación? Existen diferentes métodos de recorrido de árbol ya que la mayoría de las aplicaciones binarias son bastante sensibles al orden en el que se visitan los nodos, de forma que será preciso elegir cuidadosamente el tipo de recorrido.

Un **recorrido de un árbol binario** requiere que cada nodo del árbol sea procesado (visitado) una vez y sólo una en una secuencia predeterminada. Existen dos enfoques generales para la secuencia de recorrido, profundidad y anchura.

En el **recorrido en profundidad,** el proceso exige un camino desde el raíz a través de un hijo, al descendiente más lejano del primer hijo antes de proseguir a un segundo hijo. En otras palabras, en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo.

En el **recorrido en anchura,** el proceso se realiza horizontalmente desde el raíz a todos sus hijos, a continuación a los hijos de sus hijos y así sucesivamente hasta que todos los nodos han sido procesados. En otras palabras, en el recorrido en anchura, cada nivel se procesa totalmente antes de que comience el siguiente nivel.

Definición

El **recorrido** de un árbol supone visitar cada nodo sólo una vez.

Dado un árbol binario que consta de raíz, un subárbol izquierdo y un subárbol derecho se pueden definir tres tipos de secuencia de recorrido en profundidad. Estos recorridos estándar se muestran en la Figura 14.24.

Figura 14.24. Recorridos de árboles binarios.

La designación tradicional de los recorridos utiliza un nombre para el nodo raíz (**N**), para el subárbol izquierdo (**I**) y para el subárbol derecho (**D**).

Según sea la estrategia a seguir, los recorridos se conocen como **enorden** (*inorder*), **preorden** (*preorder*) y **postorden** (*postorder*):

Preorden	(nodo-izquierdo-derecho) (**NID**)
Enorden	(izquierdo-nodo-derecho) (**IND**)
Postorden	(izquierdo-derecho-nodo) (**IDN**)

14.6.1. Recorrido *preorden*

El recorrido *preorden*[2] (**NID**) conlleva los siguientes pasos, en los que el nodo raíz va antes que los subárboles:

1. Visitar el nodo raíz (**N**).
2. Recorrer el subárbol izquierdo (**I**) en preorden.
3. Recorrer el subárbol derecho (**D**) en preorden.

Dadas las características recursivas de los árboles, el algoritmo de recorrido tiene naturaleza recursiva. Primero se procesa la raíz, a continuación el subárbol izquierdo y después el subárbol derecho. Para procesar el subárbol izquierdo, se hace una llamada recursiva al procedimiento *preorden* y luego se hace lo mismo con el subárbol derecho.

Regla

En el recorrido preorden, el raíz se procesa antes que los subárboles izquierdo y derecho.

Si utilizamos el recorrido *preorden* del árbol de la Figura 14.25 se visita primero el raíz (nodo `A`). A continuación se visita el subárbol izquierdo de `A`, que consta de los nodos `B`, `D` y `E`. Dado que el subárbol es, a su vez, un árbol, se visitan los nodos utilizando el orden `NID`. Por consiguiente, se visita primero el nodo `B`, después `D` (izquierdo) y por último `E` (derecho).

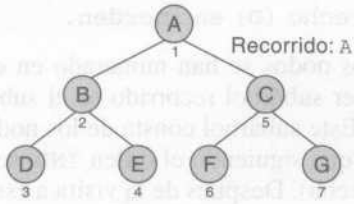


Figura 14.25. Recorrido *preorden* de un árbol binario.

[2] El nombre *preorden* viene del prefijo latino *pre*, que significa «ir antes».

A continuación se visita el subárbol derecho de A, que es un árbol que contiene los nodos C, F y G. De nuevo siguiendo el orden NID, se visita primero el nodo C, a continuación F (izquierdo) y por último G (derecho). En consecuencia el orden del recorrido *preorden* para el árbol de la Figura 14.25 es A-B-D-E-C-F-G.

En la función preorden() se implementa el algoritmo de recorrido en C, la condición que indica que continúe el recorrido es: raiz !== NULL, o simplemente, raiz.

```
void preorden (ArbolBinario raiz)
{
  if (raiz)
  {
    visitar(raiz -> dato);
    preorden(raiz -> izdo);
    preorden(raiz -> dcho);
  }
}
```

Gráficas de las llamadas recursivas de preorden

El medio gráfico de mostrar el recorrido algorítmico recursivo es similar a las diferentes etapas del algoritmo. Así, en el árbol de la Figura 14.25, la primera llamada procesa la raíz del árbol A. A continuación se llama recursivamente a procesar subárbol izquierdo, procesa el nodo B. La tercera llamada procesa el nodo D, que es un subárbol formado por un único nodo. En ese punto, se llama en *preorden*, con un puntero nulo, que produce un retorno inmediato al subárbol D para procesar a su subárbol derecho. Debido a que el subárbol derecho de D es también nulo, se vuelve al nodo B de modo que va a procesar (visitar) su subárbol derecho, E. Después de procesar el nodo E, se hacen dos llamadas más, una con el puntero izquierdo null de E y otra con su puntero derecho null. Como el subárbol B ha sido totalmente procesado, se vuelve a la raíz del árbol y se procesa su subárbol derecho, C. Después de procesar C, llama para procesar su subárbol izquierdo F. Se hacen dos llamadas con null, vuelve al nivel donde está el nodo C para procesar su rama derecha G. Aún se realizan dos llamadas más, una al subárbol izquierdo null y otra al subárbol derecho. Entonces se retorna en el árbol y se concluye el recorrido.

4.6.2. Recorrido *enorden*

El recorrido **enorden** (*inorder*) procesa primero el subárbol izquierdo, después el raíz y a continuación el subárbol derecho. El significado de *in-* es que la raíz se procesa entre los subárboles.

Si el árbol no está vacío, el método implica los siguientes pasos:

1. Recorrer el subárbol izquierdo (**I**) en inorden.
2. Visitar el nodo raíz (**N**).
3. Recorrer el subárbol derecho (**D**) en inorden.

En el árbol de la Figura 14.26, los nodos se han numerado en el orden en que son visitados durante el recorrido *enorden*. El primer subárbol recorrido es el subárbol izquierdo del nodo raíz (árbol cuyo nodo contiene la letra B). Este subárbol consta de los nodos B, D y E, y es a su vez otro árbol con el nodo B como raíz, por lo que siguiendo el orden IND, se visita primero D, a continuación B (nodo raíz) y por último E (derecha). Después de la visita a este subárbol izquierdo se visita el nodo raíz A y por último se visita el subárbol derecho que consta de los nodos C, F y G. A continuación, siguiendo el orden IND para el subárbol derecho, se visita primero F, después C (nodo raíz) y por último G. Por consiguiente, el orden del recorrido *enorden* de la Figura 14.26 es D-B-E-A-F-C-G.

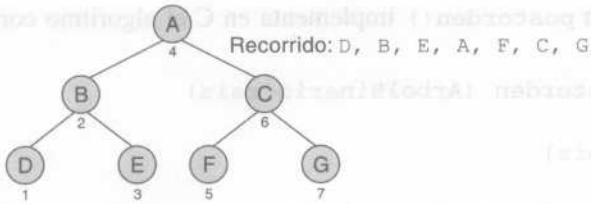


Figura 14.26. Recorrido *enorden* de un árbol binario.

La siguiente función visita y escribe el contenido de los nodos de un árbol binario de acuerdo al recorrido *EnOrden*. La función tiene como parámetro un puntero al nodo raíz del árbol.

```
void enorden (ArbolBinario raiz)
{
  if (raiz)
  {
    enorden(raiz -> izdo);     /* recorrer subárbol izquierdo */
    visitar(raiz -> dato);     /* procesa raíz */
    enorden (raiz -> dcho);    /* recorrer subárbol derecho */
  }
}
```

14.6.3. Recorrido *postorden*

El recorrido *postorden* (**IDN**) procesa el nodo raíz (*post*) después de que los subárboles izquierdo y derecho se han procesado. Se comienza situándose en la hoja más a la izquierda y se procesa. A continuación se procesa su subárbol derecho. Por último se procesa el nodo raíz. Las etapas del algoritmo, si el árbol no está vacío, son:

1. Recorrer el subárbol izquierdo (**I**) en postorden.
2. Recorrer el subárbol derecho (**D**) en postorden.
3. Visitar el nodo raíz (**N**).

Si se utiliza el recorrido postorden del árbol de la Figura 14.27 se visita primero el subárbol izquierdo de `A`. Este subárbol consta de los nodos `B`, `D` y `E` y siguiendo el orden `IDN`, se visitará primero `D` (izquierdo), luego `E` (derecho) y por último `B` (nodo). A continuación, se visita el subárbol derecho de `A` que consta de los nodos `C`, `F` y `G`. Siguiendo el orden `IDN` para este árbol, se visita primero `F` (izquierdo), después `G` (derecho) y por último `C` (nodo). Finalmente se visita el nodo raíz `A`. Así el orden del recorrido postorden del árbol de la Figura 14.27 es `D-E-B-F-G-C-A`.

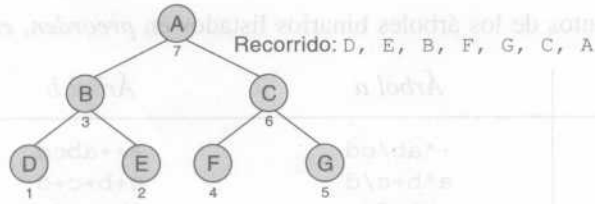


Figura 14.27. Recorrido *postorden* de un árbol binario.

La función `postorden()` implementa en C el algoritmo correspondiente:

```
void postorden (ArbolBinario raiz)
{
  if (raiz)
  {
    postorden (raiz -> izdo);
    postorden (raiz -> dcho);
    visitar(raiz -> dato);
  }
}
```

Nota de programación modular

La visita al nodo raíz del árbol se representa mediante una llamada a la función `visitar()`. ¿Qué hacer en la función? Depende de la aplicación que se esté realizando. Si simplemente se quiere listar los nodos, la sentencia `printf()` puede emplearse para ello. Así, en el supuesto de datos de tipo entero:

```
void visitar (TipoElemento dato)
{
  printf("%d ",dato);
}
```

Ejercicio 14.1

Deducir los elementos de los árboles binarios siguientes en cada uno de los tres recorridos fundamentales.

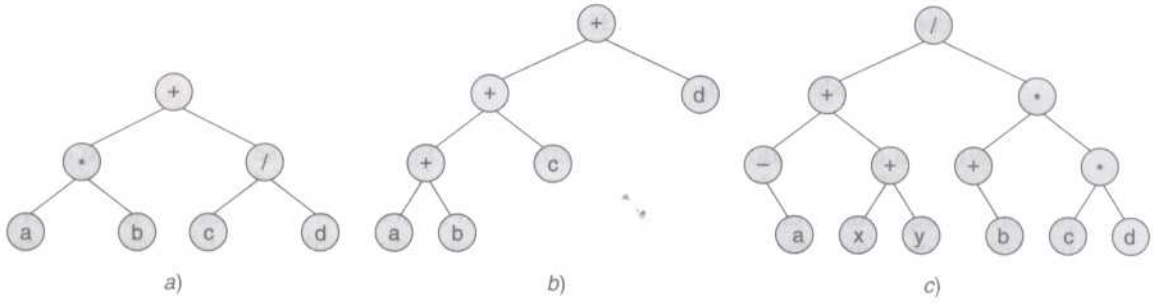


Figura 14.28. Árboles de expresión.

Los elementos de los árboles binarios listados en *preorden*, *enorden* y *postorden*.

	Árbol a	*Árbol b*	*Árbol c*
preorden	`+*ab/cd`	`+++abcd`	`/+-a+xy*+b*cd`
enorden	`a*b+c/d`	`a+b+c+d`	`-a+x+y/+b*c*d`
postorden	`ab*cd/+`	`ab+c+d+`	`a-xy++b+cd**/`

Ejemplo 14.7

Dado un árbol binario, eliminar cada uno de los nodos de que consta.

La función `eliminarbol()` utiliza un recorrido *postorden*, de tal forma que el hecho de visitar el nodo se convierte en llamar a la función `free()` para liberar la memoria ocupada. Este recorrido asegura la liberación de la memoria ocupada por un nodo después de haber liberado su rama izquierda y derecha.

```
void eliminarbol(ArbolBinario r)
{
  if (r != NULL)
  {
    eliminarbol(r -> izdo);
    eliminarbol(r -> dcho);
    printf("\tNodo borrado ");
    free(r);
  }
}
```

14.6.4. Profundidad de un árbol binario

La profundidad de un árbol binario es una característica que se necesita conocer con frecuencia durante el desarrollo de una aplicación con árboles. La función `profundidad()` determina la *profundidad* de un árbol binario. Para ello tiene un parámetro que es un puntero a la raíz del árbol.

El caso más sencillo de cálculo de la profundidad es cuando el árbol está vacío en cuyo caso la profundidad es 0. Si el árbol no está vacío, cada subárbol debe tener su propia profundidad, por lo que se necesita evaluar cada una por separado. Las variables `profundidadI`, `profundidadD` almacenan las profundidades de los subárboles izquierdo y derecho respectivamente.

El método de cálculo de la profundidad de los subárboles utiliza llamadas recursivas a la función `profundidad()` con punteros a los respectivos subárboles como parámetros de la misma. La función devuelve como resultado la profundidad del subárbol más profundo más 1 (la misma del raíz).

```
int profundidad (ArbolBinario raiz)
{
  if (!raiz)
    return 0 ;
  else
  {
    int profundidadI = profundidad (raiz -> izdo);
    int profundidadD = profundidad (raiz -> dcho);
    if (profundidadI > profundidadD)
      return profundidadI + 1;
    else
      return profundidadD + 1;
  }
}
```

14.6.5. Evaluación de un árbol de expresión

En el Apartado 14.5 se han construido diversos árboles binarios para representar expresiones algebraicas. La evaluación consiste en, una vez dados valores numéricos a los operandos, obtener el valor resultante de la expresión.

El algoritmo de evaluación es aplicable para los operadores algebraicos binarios: +, –, *, / y *potenciación* (^). Se basa en el recorrido del árbol de la expresión en *postorden*. De esta forma se evalúa primer operando (*rama izquierda*), segundo operando (*rama derecha*) y según el operador (nodo raíz) se obtiene el resultado.

A tener en cuenta

Una cuestión importante a considerar, para entender el algoritmo de evaluación, es que los operandos siempre son nodos hoja en el árbol binario de expresiones.

Los operandos se han representado mediante letras, suponemos en mayúsculas. Por ello, el vector `operandos[]` tiene tantos elementos como letras mayúsculas tiene el código `ASCII`, de tal forma que la posición `0` se corresponde con `'A'`, y así sucesivamente hasta la `'Z'`. El valor numérico de un operando se encuentra en la correspondiente posición del vector.

```
double operandos[26];
```

Si, por ejemplo, los operandos son `A`, `D`, `G`; los valores se guardan en las posiciones: `operandos[0]`, `operandos[3]`, `operandos[6]`, respectivamente.

La función real `evaluar()` tiene como entrada la raíz del árbol y el vector con los valores de los operandos.

```
double evaluar(ArbolBinario raiz, double operandos[])
{
  double x, y;
  char ch ;
  if (raiz)     /* el árbol no está vacío */
  {
    ch = raiz -> datos;
    if (ch >= 'A' && ch <= 'Z')
      return operandos[ch - 'A'];
    else
    {
      x = evaluar(raiz -> izdo, operandos);
      y = evaluar(raiz -> dcho, operandos);

      switch (ch) {
        case '+': return x + y;
                  break;
        case '-': return x - y
                  break;
        case '*': return x * y;
                  break;
        case '/': if (y != 0)
                    return x/y;
```

```
                    else
                    {
                      puts("Error: división por 0");
                      exit(-1);
                    }
                    break;
            case '^': return pow(x,y);
        }
      }
    }
  }
```

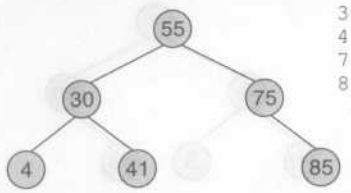
14.7. ÁRBOL BINARIO DE BÚSQUEDA

Los árboles vistos hasta ahora no tienen un orden definido; sin embargo, los árboles binarios ordenados tienen sentido. Estos árboles se denominan árboles binarios de búsqueda, debido a que se pueden buscar en ellos un término utilizando un algoritmo de búsqueda binaria similar al empleado en arrays.

Un **árbol binario de búsqueda** es aquel que dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que sus propios datos. El árbol binario del Ejemplo 14.8 es de búsqueda.

Ejemplo 14.8

Árbol binario de búsqueda para nodos con el campo de datos de tipo `int`.



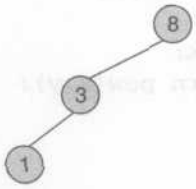
30 menor que 55
41 mayor que 30
75 mayor que 55
85 mayor que 75
4 menor que 30

14.7.1. Creación de un árbol binario de búsqueda

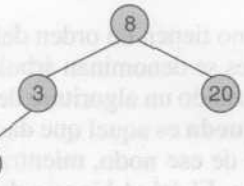
Supongamos que se desea almacenar los números 8 3 1 20 10 5 4 en un árbol binario de búsqueda. Siguiendo la regla, dado un nodo en el árbol todos los datos a su izquierda deben ser menores que todos los datos del nodo actual, mientras que todos los datos a la derecha deben ser mayores que los datos. Inicialmente el árbol está vacío y se desea insertar el 8. La única elección es almacenar el 8 en el raíz:

A continuación viene el 3, ya que 3 es menor que 8, el 3 debe ir en el subárbol izquierdo:

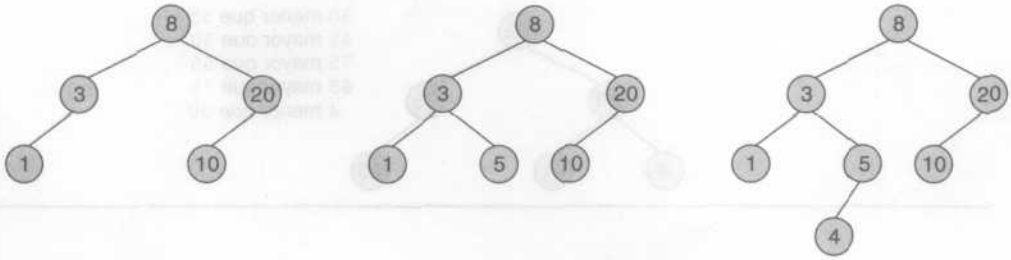
A continuación se ha de insertar 1 que es menor que 8 y que 3, por consiguiente irá a la izquierda y debajo de 3:



El siguiente número es 20, mayor que 8, lo que implica debe ir a la derecha de 8:



Cada nuevo elemento se inserta como una *hoja* del árbol. Los restantes elementos se pueden situar fácilmente:

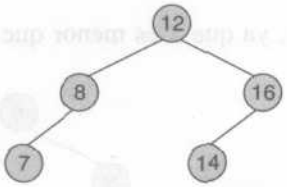


Una propiedad de los árboles binarios de búsqueda es que no son únicos para los mismos datos.

Ejemplo 14.9

Construir un árbol binario para almacenar los datos 12, 8, 7, 16 y 14.

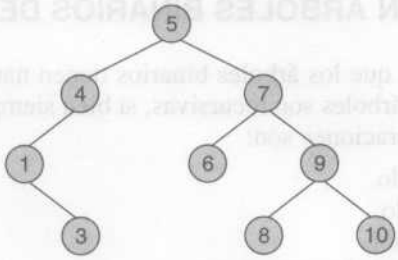
Solución



Ejemplo 14.10

Construir un árbol binario de búsqueda que corresponda a un recorrido *enorden* cuyos elementos son: 1, 3, 4, 5, 6, 7, 8, 9 y 10.

Solución



14.7.2. Nodo de un árbol binario de búsqueda

Un nodo de un árbol binario de búsqueda no difiere en nada de los nodos de un árbol binario, tiene un campo de datos y dos punteros a los subárboles izquierdo y derecho respectivamente.

Un árbol de búsqueda se puede utilizar cuando se necesita que la información se encuentre rápidamente. Un ejemplo de árbol binario de búsqueda es el que cada nodo contiene información relativa a una persona. Cada nodo almacena un nombre de un alumno (dato de tipo cadena) y el número de matrícula en su universidad (dato entero).

Declaración de tipos

```
struct nodo {
  int nummat;          /* número de matrícula */
  char nombre[30];     /* nombre del alumno */
  struct nodo *izdo, *dcho;
};
typedef struct nodo Nodo;
```

Creación de un nodo

La función tiene como entrada un dato entero, que representa un número de matrícula y el nombre. Devuelve un puntero al nodo creado.

```
Nodo* crearNodo(int id, const char* n)
{
   Nodo* t;
   t = (Nodo*) malloc(sizeof(Nodo));
```

```
        t -> nummat = id;
        strcpy(t -> nombre, n);
        t -> izdo = t -> dcho = NULL;
        return t;
    }
```

14.8. OPERACIONES EN ÁRBOLES BINARIOS DE BÚSQUEDA

De lo expuesto se deduce que los árboles binarios tienen naturaleza recursiva y, en consecuencia, las operaciones sobre los árboles son recursivas, si bien siempre se tiene la opción de realizarlas de forma iterativa. Estas operaciones son:

- *Búsqueda* de un nodo.
- *Inserción* de un nodo.
- *Recorrido* de un árbol.
- *Borrado* (eliminación) de un nodo.

14.8.1. Búsqueda

La búsqueda de un nodo comienza en el nodo raíz y sigue estos pasos:

1. La clave buscada se compara con la clave del nodo raíz.
2. Si las claves son iguales, la búsqueda se detiene.
3. Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el subárbol derecha. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda con el subárbol izquierdo.

A recordar

En los árboles binarios ordenados la búsqueda de una clave da lugar a un *camino de búsqueda*, de tal forma que *baja* por la rama izquierda si la clave buscada es menor que la clave del raíz, *baja* por la rama derecha si la clave es mayor.

Buscar una información específica

Si se desea encontrar un nodo en el árbol que contenga una información determinada, la función `buscar()` tiene dos parámetros, un puntero al árbol y el dato que se busca. Como resultado, la función devuelve un puntero al nodo en el que se almacena la información; en el caso de que la información no se encuentre se devuelve el valor 0. El algoritmo de búsqueda es el siguiente:

1. Comprobar si el árbol está vacío. En caso afirmativo se devuelve 0. Si la raíz contiene el dato buscado, la tarea es fácil: el resultado es, simplemente, un puntero a la raíz.
2. Si el árbol no está vacío, el subárbol específico depende de que dato requerido sea más pequeño o mayor que el dato del nodo raíz.
3. La función de búsqueda se consigue llamando recursivamente a la función `buscar()` con un puntero al subárbol izquierdo o derecho como parámetro.

El código C de la función `buscar()`:

```
Nodo* buscar (Nodo* raiz, TipoElemento buscado)
{
  if (!raiz)
    return 0;
  else if (buscado == raiz -> dato)
    return raiz;
```

```
    else if (buscado < raiz -> dato)
        return buscar (raiz -> izdo, buscado);
    else
        return buscar (raiz -> dcho, buscado);
}
```

Ejemplo 14.11

Se desea aplicar la función de búsqueda para encontrar la dirección del nodo que contenga el número de matrícula de un alumno.

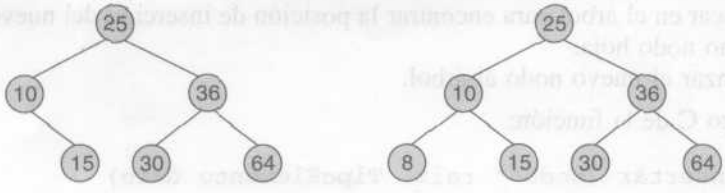
La función se escribe con un bucle, en vez de hacerlo recursivamente. El bucle termina cuando se encuentra el nodo, o bien cuando el camino de búsqueda ha finalizado.

La función se mueve por el árbol, de tal forma que baja por la rama izquierda o derecha según la clave sea menor o mayor que la clave del nodo actual.

```
Nodo* buscar (Nodo* raiz, int matricula)
{
  int encontrado = 0;

  while (!encontrado && raiz != NULL)
  {
    if (matricula == raiz -> nummat)
      encontrado = 1;
    else if (matricula < raiz -> nummat)
      raiz = raiz -> izdo;
    else if (matricula > raiz -> nummat)
      raiz = raiz -> dcho;
  }
  return raiz;
}
```

14.8.2. Insertar un nodo

Una característica fundamental que debe poseer el algoritmo de inserción es que el árbol resultante de una inserción en un árbol de búsqueda ha de ser también de búsqueda. En esencia, el algoritmo de inserción se apoya en la localización de un elemento, de modo que si se encuentra el elemento (*clave*) buscado, no es necesario hacer nada; en caso contrario, se inserta el nuevo elemento justo en el lugar donde ha acabado la búsqueda (es decir, en el lugar donde habría estado en el caso de existir).



Antes de insertar 8 *Después de insertar* 8

Figura 14.29. Inserción en un árbol binario de búsqueda.

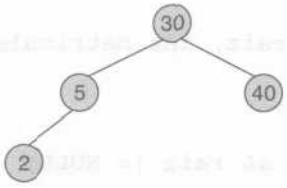
Por ejemplo, considérese el caso de añadir el nodo `8` al árbol de la Figura 14.29. Se comienza el recorrido en el nodo raíz `25`; la posición `8` debe estar en el subárbol izquierdo de `25` (`8 < 25`). En el nodo `10`, la posición de `8` debe estar en el subárbol izquierdo de `10`, que está actualmente vacío. El nodo `8` se introduce como un hijo izquierdo del nodo `10`.

A recordar

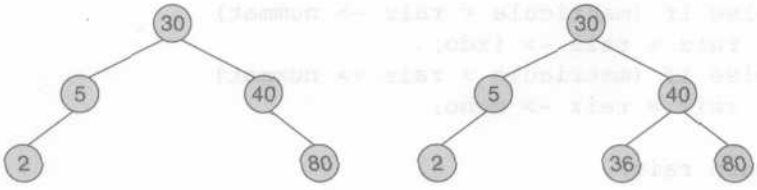
La inserción de un nuevo nodo en un árbol de búsqueda siempre se hace como nodo hoja. Para ello se *baja* por el árbol según el *camino de búsqueda*.

Ejemplo 14.12

Insertar un elemento con clave 80 en el árbol binario de búsqueda siguiente:



A continuación, insertar un elemento con clave 36 en el árbol binario de búsqueda resultante:



a) Inserción de 80 *b*) Inserción de 36

14.8.3. Función `insertar()`

La función `insertar()` que inserta nuevos nodos es sencilla. Se deben declarar dos argumentos, un puntero al raíz del árbol y el dato que tendrá el nodo. La función creará un nuevo nodo y lo inserta en el lugar correcto en el árbol de modo que el árbol permanezca como binario de búsqueda.

La operación de *inserción* de un nodo es una extensión de la operación de búsqueda. Los pasos a seguir son:

1. Asignar memoria para una nueva estructura nodo.
2. Buscar en el árbol para encontrar la posición de inserción del nuevo nodo, que se colocará como nodo hoja.
3. Enlazar el nuevo nodo al árbol.

El código C de la función:

```
void insertar (Nodo** raiz, TipoElemento dato)
{
  if (!(*raiz))
    *raiz = crearNodo(dato);
```

```
    else if (dato < (*raiz) -> dato)
      insertar (&((*raiz) -> izdo), dato);
    else
      insertar (&((*raiz) -> dcho), dato);
  }
```

Si el árbol está vacío, es fácil insertar la entrada en el lugar correcto. El nuevo nodo es la raíz del árbol y el puntero `raiz` se pone apuntando a ese nodo. El parámetro `raiz` debe ser un parámetro referencia ya que debe ser leído y actualizado, por esa razón se declara puntero a puntero (`Nodo**`) . Si el árbol no está vacío, se debe elegir entre insertar el nuevo nodo en el subárbol izquierdo o derecho, dependiendo de que el *dato* sea más pequeño o mayor que el *dato* en la raíz del árbol.

La función `crearNodo()` reserva memoria para el nuevo nodo y asigna el nuevo dato. Esta función se debe adaptar a cada aplicación, así en el Apartado 14.7.2, se crea un nodo para guardar el número de matrícula y el nombre de una persona.

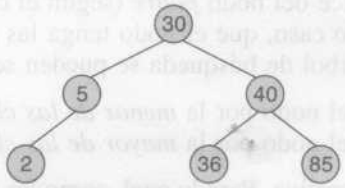
14.8.4. Eliminación

La operación de *eliminación* de un nodo es también una extensión de la operación de búsqueda, si bien más compleja que la inserción debido a que el nodo a suprimir puede ser cualquiera y la operación de supresión debe mantener la estructura de árbol binario de búsqueda después de la eliminación. Los pasos a seguir son:

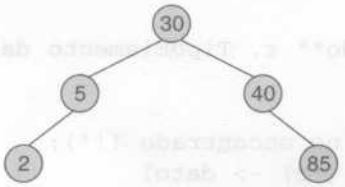
1. Buscar en el árbol para encontrar la posición de nodo a eliminar.
2. Reajustar los punteros de sus antecesores si el nodo a suprimir tiene menos de 2 hijos, o subir a la posición que éste ocupa el nodo más próximo en valor del campo clave (inmediatamente superior o inmediatamente inferior) con objeto de mantener la estructura de árbol binario.

Ejemplo 14.13

Suprimir el elemento de clave 36 del siguiente árbol binario de búsqueda:

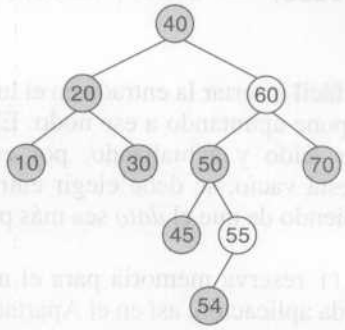


El nodo del árbol donde se encuentra la clave `36` es una hoja, por ello simplemente se reajustan los enlaces del nodo precedente en el camino de búsqueda. El árbol resultante:

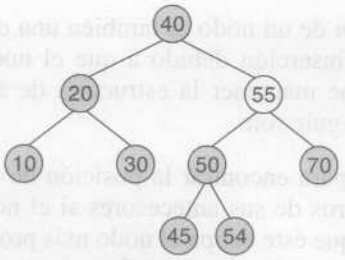


Ejemplo 14.14

Borrar el elemento de clave `60` del siguiente árbol:



Se reemplaza `60` o bien con el elemento mayor `(55)` en su subárbol izquierdo o el elemento más pequeño `(70)` en su subárbol derecho. Si se opta por reemplazar con el elemento mayor del subárbol izquierdo, se mueve el `55` al raíz del subárbol y se reajusta el árbol.



14.8.5. Función `eliminar()`

La función `eliminar()` presenta dos casos claramente diferenciados. El primero, si el nodo a eliminar es una hoja o tiene un único descendiente, resulta una tarea fácil, ya que lo único que hay que hacer es asignar al enlace del nodo *padre* (según el camino de búsqueda) el descendiente del nodo a eliminar. El segundo caso, que el nodo tenga las dos ramas no vacías; en cuyo caso para mantener la estructura de árbol de búsqueda se pueden seguir dos alternativas:

- Reemplazar el dato del nodo por la *menor de las claves mayores* en su subárbol derecho.
- Reemplazar el dato del nodo por la *mayor de las claves menores* en su subárbol izquierdo.

Se elige la segunda alternativa. Para lo cual, como las claves menores están en la rama izquierda, se *baja* al primer nodo de la rama izquierda, y como la clave mayor está en la rama derecha, se continúa *bajando* por la rama derecha hasta alcanzar el nodo hoja. Éste es el mayor de los menores que reemplaza a la clave del nodo a eliminar. La función `reemplazar()` realiza la tarea descrita. El código en C:

```
void eliminar (Nodo** r, TipoElemento dato)
{
  if (!(*r))
    puts("!! Nodo no encontrado !!");
  else if (dato < (*r) -> dato)
    eliminar(&(*r) -> izdo, dato);
```

```
  else if (dato > (*r) -> dato)
    eliminar(&(*r) -> dcho,dato);
  else                                          /* Nodo encontrado */
  {
    Nodo* q;                                /* puntero al nodo a suprimir */
    q = (*r);
    if (q -> izdo == NULL)
      (*r) = q -> dcho;
    else if (q -> dcho == NULL)
      (*r) = q -> izdo;
    else
    {                                   /* tiene rama izquierda y derecha */
      reemplazar(&q);
    }
    free(q);
  }
}

void reemplazar(Nodo** act)
{
  Nodo* a, *p;

  p = *act;
  a = (*act) -> izdo;                           /* rama de menores */
  while (a -> dcho)
  {
    p = a;
    a = a -> dcho;
  }
  (*act) -> dato = a -> dato;
  if (p == (*act))
    p -> izdo = a -> izdo;
  else
    p -> dcho = a -> izdo;
  (*act) = a;
}
```

Ejercicio 14.2

Con los registros de estudiantes, formar un árbol binario de búsqueda, ordenado respecto al campo clave número de matrícula. El programa debe tener las opciones de mostrar los registros ordenados y eliminar un registro dando el número de matrícula.

Cada registro tiene sólo dos campos de información: `nombre` y `nummat`. Además, los campos de enlace con el subárbol izquierdo y derecho.

Las operaciones utilizadas son `insertar`, `eliminar`, `buscar` y `visualizar` el árbol. Los algoritmos de las tres primeras operaciones ya están descritos anteriormente. La operación de `visualizar` consiste en recorrer en `inorden` el árbol, cada vez que se visita el nodo raíz se escriben los datos del estudiante.

No se vuelve a escribir el código de las funciones; dicho código puede consultarse en los apartados anteriores. Si se escribe `visualizar()`, que es una aplicación de recorrer en *inorden* el árbol de búsqueda, este tipo de recorrido hace que se muestren por pantalla los registros.

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

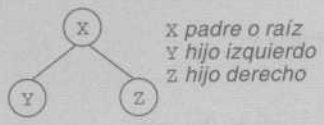
```
#include <stdlib.h>
#include <conio.h>
struct nodo {
  int nummat;
  char nombre[30];
  struct nodo *izdo, *dcho;
};
typedef struct nodo Nodo;

Nodo* crearNodo(int id, char* n);
Nodo* buscar (Nodo* p, int buscado);
void insertar (Nodo** raiz, int matricula, char* nombre);
void reemplazar(Nodo** act);
void eliminar (Nodo** raiz, int matricula);
void visualizar (Nodo* r);

void main()
{
  int nm;
  char nom[30];
  Nodo* raiz = NULL;
            /* Crea el árbol */
  do{
    printf("Numero de matricula(0 -> Fin): "); scanf("%d%*c",&nm);
    if (nm)
    {
      printf("Nombre: "); gets(nom);
      insertar(&raiz,nm,nom);
    }
  }while (nm);
    /* Opciones de escribir el árbol o borrar un registro */
  clrscr();
  do{
    puts("     1. Mostrar el árbol\n");
    puts("     2. Eliminar un registro\n");
    puts("     3. Salir\n ");
    do scanf("%d%*c", &nm); while(nm<1 || nm>3);
    if (nm == 1) {
      printf("\n\t Registros ordenados por número de matrícula:\n");
      visualizar(raiz);
    }
    else if (nm == 2){
      int cl;
      printf("Clave: "); scanf("%d",&cl);
      eliminar(&raiz,cl);
    }
  }while (nm != 3);
}
void visualizar (Nodo* r)
{
  if (r)
  {
```

```
        visualizar(r -> izdo);
        printf("Matricula %d \t %s \n",r->nummat,r->nombre);
        visualizar(r -> dcho);
    }
}
```

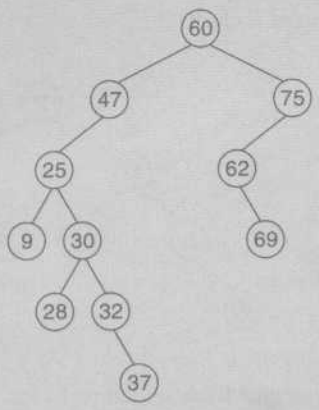
RESUMEN

En este capítulo se ha introducido y desarrollado la estructura de datos dinámica **árbol.** Esta estructura, muy potente, se puede utilizar en una gran variedad de aplicaciones de programación.

La estructura árbol más utilizada normalmente es el *árbol binario*. Un árbol binario es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho.

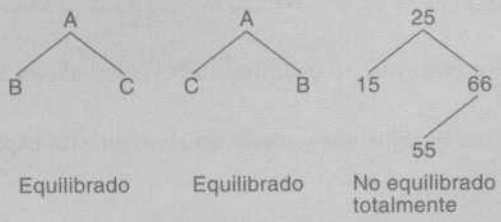


En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre, pero sí cada elemento restante tiene un padre. X es un *antecesor* o *ascendente* del elemento Y.



La altura de un árbol binario es el número de ramas entre el raíz y la hoja más lejana, más 1. Si el árbol A es vacío, la altura es 0. La altura del árbol anterior es 6. *El nivel* o *profundidad* de un elemento es un concepto similar al de altura. En el árbol anterior el nivel de 30 es 3 y el nivel de 37 es 5. Un nivel de un elemento se conoce también como *profundidad*.

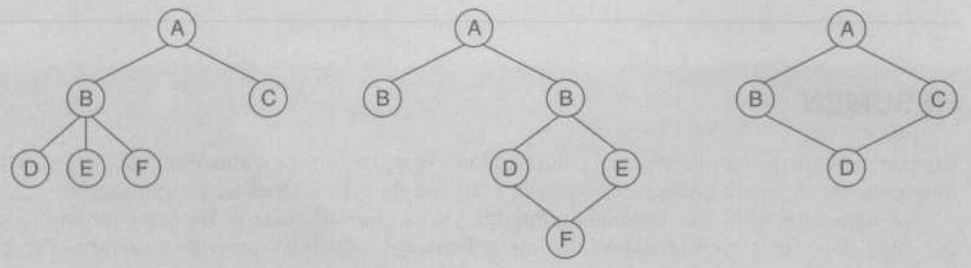
Un árbol binario no vacío está *equilibrado totalmente* si sus subárboles izquierdo y derecho tienen la misma altura y ambos son o bien vacíos o totalmente equilibrados.



Los árboles binarios presentan dos tipos característicos: *árboles binarios de búsqueda* y *árboles binarios de expresiones*. Los árboles binarios de búsqueda se utilizan fundamentalmente para mantener una colección ordenada de datos y los árboles binarios de expresiones para almacenar expresiones.

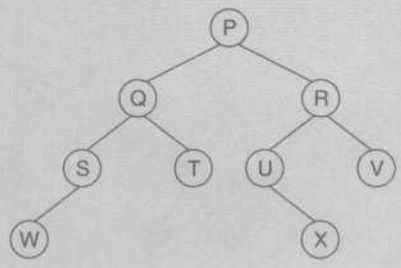
EJERCICIOS

14.1. Explicar por qué cada una de las siguientes estructuras no es un árbol binario.



14.2. Considérese el árbol siguiente:

a) ¿Cuál es su altura?
b) ¿Está el árbol equilibrado? ¿Por qué?
c) Listar todos los nodos hoja.
d) ¿Cuál es el predecesor inmediato (padre) del nodo U.
e) Listar los hijos del nodo R.
f) Listar los sucesores del nodo R.



14.3. Para cada una de las siguientes listas de letras:

a) Dibujar el árbol binario de búsqueda que se construye cuando las letras se insertan en el orden dado.
b) Realizar recorridos *enorden*, *preorden* y *postorden* del árbol y mostrar la secuencia de letras que resultan en cada caso.

i) `M, Y, T, E, R`
ii) `T, Y, M, E, R`
iii) `R, E, M, Y, T`
iv) `C, O, R, N, F, L, A, K, E, S`

14.4. En el árbol del Ejercicio 14.2, recorrer dicho árbol utilizando los órdenes siguientes: NDI, DNI, DIN.

14.5. Dibujar los árboles binarios que representan las siguientes expresiones:

a) `(A+B)/(C-D)`
b) `A+B+C/D`
c) `A-(B-(C-D)/(E+F))`
d) `(A+B)*((C+D)/(E+F))`
e) `(A-B)/((C*D)-(E/F))`

14.6. El recorrido *preorden* de un cierto árbol binario produce:

ADFGHKLPQRWZ

y en recorrido *enorden* produce:

GFHKDLAWRQPZ

Dibujar el árbol binario.

14.7. Escribir una función recursiva que cuente las hojas de un árbol binario.

14.8. Escribir una función que determine el número de nodos que se encuentran en un nivel *n* de un árbol binario.

14.9. Escribir una función que tome un árbol como entrada y devuelva el número de hijos del árbol.

14.10. Escribir una función booleana a la cual se le pase un puntero a un árbol binario y devuelva verdadero (*true*) si el árbol es completo y falso (*false*) en caso contrario.

14.11. Se dispone de un árbol binario de elementos de tipo entero. Escribir funciones que calculen:

a) La suma de sus elementos.
b) La suma de sus elementos que son múltiplos de 3.

14.12. Diseñar una función iterativa que encuentre el número de nodos hoja en un árbol binario.

14.13. En un árbol de búsqueda cuyo campo clave es de tipo entero, escribir un método que devuelva el número de nodos cuya clave se encuentra en el rango [x1, x2].

14.14. Diseñar una función que visite los nodos del árbol por niveles; primero el nivel 0, después los nodos del nivel 1, nivel 2 y así hasta el último nivel.

PROBLEMAS

14.1. Crear un archivo de datos en el que cada línea contenga la siguiente información:

Columnas	1-20	Nombre
	21-31	Número de la Seguridad Social
	32-78	Dirección

Escribir un programa que lea cada registro de datos y lo inserte en un árbol, de modo que cuando el árbol se recorra utilizando recorrido en orden, los números de la seguridad social se ordenen en orden ascendente. Imprimir una cabecera «DATOS DE EMPLEADOS ORDENADOS POR NÚMERO SEGURIDAD SOCIAL». A continuación se han de imprimir los tres datos utilizando el siguiente formato de salida:

Columnas	1-11	Número de la Seguridad Social
	25-44	Nombre
	58-104	Dirección

14.2. Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferentes contenidas en el texto, así como su frecuencia de aparición. Hacer uso de la estructura árbol binario de búsqueda, cada nodo del árbol que tenga una palabra y su frecuencia.

14.3. Escribir un programa que procese un árbol binario cuyos nodos contengan caracteres y a partir del siguiente menú de opciones:

`I` (seguido de un carácter): Insertar un carácter
`B` (seguido de un carácter): Buscar un carácter
`RE`: Recorrido en orden
`RP`: Recorrido en *preorden*
`RT`: Recorrido *postorden*
`SA`: Salir

14.4. Escribir una función booleana `identicos()` que permita decir si dos árboles binarios son iguales.

14.5. 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 DE 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

14.6. Diseñar un programa interactivo que permita dar altas, bajas, listar, etc., en un árbol binario de búsqueda.

14.7. Construir una función recursiva para escribir todos los nodos de un árbol binario de búsqueda cuyo campo clave sea mayor que un valor dado (el campo clave es de tipo entero).

14.8. Escribir una función que determine la altura de un nodo y un programa que cree un árbol binario con números generados aleatoriamente y muestre por pantalla:

- La altura de cada nodo del árbol.
- La diferencia de altura entre rama izquierda y derecha de cada nodo.

14.9. Diseñar funciones no recursivas que listen los nodos de un árbol en *enorden*, *preorden* y *postorden*.

14.10. Dados dos árboles binarios A y B se dice que son *parecidos* si el árbol A puede ser transformado en el árbol B intercambiando los hijos izquierdo y derecho (de alguno de sus nodos). Escribir una función que determine si dos árboles son *parecidos*.

14.11. Dado un árbol binario de búsqueda construir su árbol espejo. (Árbol espejo es el que se construye a partir de uno dado, convirtiendo el subárbol izquierdo en subárbol derecho y viceversa.)

14.12. 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 `2I + 1`. Diseñar a partir de esta representación las funciones con las operaciones correspondientes para gestionar interactivamente un árbol de números enteros.

14.13. Una matriz de *N* elementos almacena cadenas de caracteres. Utilizando un árbol binario de búsqueda como estructura auxiliar, ordene ascendentemente la cadena de caracteres.

14.14. Dado un árbol binario de búsqueda diseñe una función que liste los nodos del árbol ordenados de modo descendentemente.

15 ÁRBOLES EQUILIBRADOS DE BÚSQUEDA

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Conocer la eficiencia de un árbol de búsqueda.
- Construir un árbol binario equilibrado conociendo el número de claves.
- Construir un árbol binario de búsqueda equilibrado.
- Describir los diversos tipos de movimientos que se hacen cuando se desequilibra un árbol.
- Realizar en C las operaciones de insertar y eliminar en un árbol binario equilibrado.

CONTENIDO

15.1. Eficiencia de la búsqueda en un árbol *binario ordenado.*

15.2. Árbol binario equilibrado, árboles AVL.

15.3. Inserción en árboles de búsqueda equilibrados. Rotaciones.

15.4. Realización en C de la inserción con balanceo y rotaciones.

15.5. Eliminación de un nodo en un árbol equilibrado.

RESUMEN

EJERCICIOS

PROBLEMA

CONCEPTOS CLAVE

- Equilibrio.
- Factor de equilibrio.
- Complejidad logarítmica.
- Árbol de búsqueda.
- Camino de búsqueda.
- *Altura de un árbol.*
- Hoja.
- Rotaciones.

INTRODUCCIÓN

En el Capítulo 14 se introdujo el concepto de árbol binario. Se utiliza un árbol binario de búsqueda para almacenar datos organizados jerárquicamente. Sin embargo, en muchas ocasiones, las inserciones y eliminaciones de elementos en el árbol no ocurren en un orden predecible; es decir, los datos no están organizados jerárquicamente.

En este capítulo se estudian tipos de árboles adicionales: los árboles equilibrados o árboles AVL, como también se les conoce, que ayudan eficientemente a resolver estas situaciones.

El concepto de árbol equilibrado así como los algoritmos de manipulación son el motivo central de este capítulo. Los métodos que describen este tipo de árboles fueron descritos en 1962 por los matemáticos rusos G. M. Adelson-Velskii y E. M. Landis.

15.1. EFICIENCIA DE LA BÚSQUEDA EN UN ÁRBOL BINARIO ORDENADO

La eficiencia para una búsqueda de una clave en un árbol binario ordenado varía entre `O(n)` y `O(log(n))`, dependiendo de la estructura que presente el árbol.

Si los elementos se añaden en el árbol mediante el algoritmo de inserción expuesto en el capítulo anterior, la estructura resultante del árbol dependerá del orden en que sean añadidos. Así, si todos los elementos se insertan en orden creciente o decreciente, el árbol va a tener todas la ramas izquierda o derecha, respectivamente, vacías. Entonces, la búsqueda en dicho árbol será totalmente secuencial.

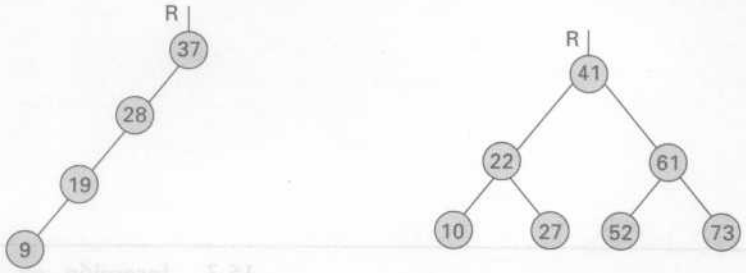


Figura 15.1. Árbol degenerado y equilibrado de búsqueda.

Sin embargo, si la mitad de los elementos insertados después de otro con clave `K` tienen claves menores de `K` y la otra mitad claves mayores de `K`, se obtiene un árbol equilibrado (también llamado *balanceado*), en el cual las comparaciones para obtener un elemento son un máximo de $\text{Log}_2(n)$ para un árbol de *n* nodos.

En los árboles de búsqueda el número promedio de comparaciones que deben realizarse para las operaciones de inserción, eliminación y búsqueda varía entre $\text{Log}_2(n)$, para el mejor de los casos, y `n` para el *peor de los casos*. Para optimizar los tiempos de búsqueda en los árboles ordenados surgen los árboles *casi equilibrados*, en los que la complejidad de la búsqueda es logarítmica, `O(Log n))`.

15.2. ÁRBOL BINARIO EQUILIBRADO, ÁRBOLES AVL

Un árbol totalmente equilibrado se caracteriza por que la altura de la rama izquierda es igual que la altura de la rama derecha para cada uno de los nodos del árbol. Es un árbol ideal, no siempre se puede conseguir que el árbol esté totalmente balanceado.

La estructura de datos de árbol equilibrado que se utiliza es el árbol **AVL.** El nombre es en honor de Adelson-Velskii-Landis, que fueron los primeros científicos en estudiar las propiedades de esta estructura de datos. Son árboles ordenados o de búsqueda que además cumplen la condición de balanceo para cada uno de los nodos.

Definición

Un árbol equilibrado o *árbol AVL* es un árbol binario de búsqueda en el que las alturas de los subárboles izquierdo y derecho de cualquier nodo difieren como máximo en 1.

La Figura 15.2 muestra dos árboles de búsqueda, el de la izquierda está equilibrado. El de la derecha es el resultado de insertar la clave 2 en el anterior, según el algoritmo de inserción en árboles de búsqueda. La inserción provoca que se *viole* la condición de equilibrio en el nodo raíz del árbol.

Figura 15.2. *Dos árboles de búsqueda: el de la izquierda, equilibrado; el otro, no.*

La condición de equilibrio de cada nodo implica una restricción en las alturas de los subárboles de un árbol AVL. Si v_r es la raíz de cualquier subárbol de un árbol equilibridado, y `h` la altura de la rama izquierda entonces la altura de la rama derecha puede tomar los valores: `h-1`, `h`, `h+1`. Esto aconseja asociar a cada nodo el parámetro denominado *factor de equilibrio o balance de un nodo*. Se define como la altura del subárbol derecho menos la altura del subárbol izquierdo correspondiente. El factor de equilibrio de cada nodo en un árbol equilibrado puede tomar los valores: `1`, `-1` o `0`. La Figura 15.3 muestra un árbol balanceado con el factor de equilibrio de cada nodo.

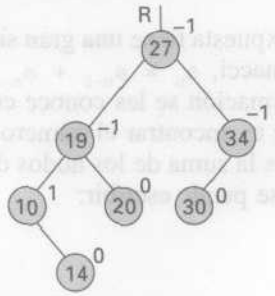


Figura 15.3. Árbol equilibrado con el factor de equilibrio de cada nodo.

A tener en cuenta

La altura o profundidad de un árbol binario es el nivel máximo de sus hojas más uno. La altura de un árbol nulo se considera cero.

15.2.1. Altura de un árbol equilibrado, árbol AVL

No resulta fácil determinar la altura promedio de un árbol AVL, por lo que se determina la altura en el *peor de los casos*, es decir, la altura máxima que puede tener un árbol equilibrado con un número de nodos n. La altura es un parámetro importante ya que coincide con el número de iteraciones que se realizan para *bajar* desde el nodo raíz al nivel más profundo de las hojas. La eficiencia de los algoritmos de búsqueda, inserción y borrado depende de la altura del árbol AVL.

Para calcular la altura máxima de un árbol AVL de n nodos se parte del siguiente razonamiento: *¿cuál es el número mínimo de nodos que puede tener un árbol binario para que se considere equilibrado con una altura h?* Si ese árbol es A_h, tendrá dos subárboles izquierdo y derecho respectivamente, A_i y A_d cuya altura difiera en 1, supongamos que tienen de altura $h - 1$ y $h - 2$ respectivamente. Al considera que A_h es el árbol de menor número de nodos de altura h, entonces A_i y A_d también son árboles AVL de menor número de nodos, pero de altura $h - 1$ y $h - 2$ y son designados como A_{h-1} y A_{h-2} . Este razonamiento se sigue extendiendo a cada subárbol y se obtienes árboles equilibrados como los de la Figura 15.4.

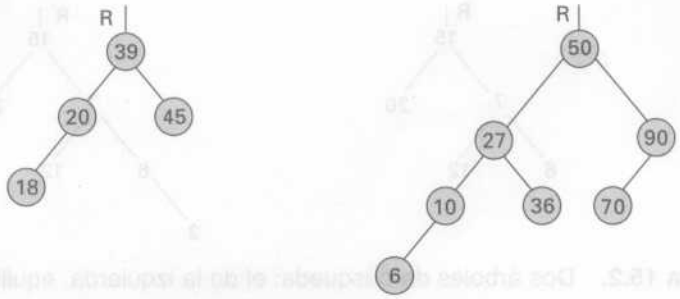


Figura 15.4. Árboles de Fibonacci.

La construcción de árboles binarios equilibrados siguiendo esta estrategia puede representarse matemáticamente:

$$A_h = A_{h-1} + A_{h-2}$$

La expresión matemática expuesta tiene una gran similitud con la ley de recurrencia que permite encontrar números de Fibonacci, $a_n = a_{n-1} + a_{n-2}$. Por esa razón a los árboles equilibrados construidos con esta ley de formación se les conoce como *árboles de Fibonacci*.

El objetivo que se persigue es encontrar el número de nodos, *n*, que hace la altura máxima. El número de nodos de un árbol es la suma de los nodos de su rama izquierda, rama derecha más uno (la raíz). Si ese número es N_h se puede escribir:

$$N_h = N_{h-1} + N_{h-2} + 1$$

donde $N_0 = 1$, $N_1 = 2$, $N_2 = 4$, $N_3 = 7$, y así sucesivamente. Se observa que los números $N_h + 1$ cumplen la definición de los números de Fibonacci. El estudio matemático de la función generadora de los números de Fibonacci permite encontrar esta relación:

$$N_h + 1 \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3}$$

Tomando logaritmos se encuentra la altura `h` en función del número de nodos, N_h:

$$h \approx 1.44 \log (N_h)$$

Como conclusión, el árbol equilibrado de `n` nodos menos denso tiene como altura `1.44log n`. Como `n` es el número de nodos en el *peor de los casos* de árbol AVL de altura `h`, se puede afirmar que la complejidad de una búsqueda es `O(Log n)`.

A tener en cuenta

La altura de un árbol binario perfectamente equilibrado de `n` nodos es `log n`. Las operaciones que se aplican a los árboles AVL no requieren más del 44 por 100 de tiempo (en el caso más desfavorable) que si se aplican a un árbol perfectamente equilibrado.

Ejercicio 15.1

Se dispone de *n* claves que se van a organizar jerárquicamente formando un árbol equilibrado. Escribir un programa para formar el árbol AVL siguiendo la estrategia de los árboles de Fibonacci descrita en la sección anterior.

La formación de un árbol balanceado de *n* nodos se parece mucho a la secuencia de los números de Fibonacci:

```
a(n) = a(n-2) + a(n-1)
```

Un *árbol de Fibonacci* (árbol equilibrado) puede definirse:

1. Un árbol vacío es el árbol de Fibonacci de altura 0.
2. Un nodo único es un árbol de Fibonacci de altura 1.
3. Si A_{h-1} y A_{h-2} son árboles de Fibonacci de alturas h-1 y h-2 entonces

   $A_h = < A_{h-1}, x, A_{h-2} >$ es árbol de Fibonacci de altura h.

El número de nodos, A_h, viene dado por la sencilla relación recurrente:

$N_0 = 0$
$N_1 = 1$
$N_h = N_{h-1} + 1 + N_{h-2}$

Para conseguir un árbol AVL con un número dado, n, de nodos hay que distribuir equitativamente los nodos a la izquierda y a la derecha de un nodo dado. En definitiva, es seguir la relación de recurrencia anterior, que expresada recursivamente:

1. Crear nodo raíz.
2. Generar el subárbol izquierdo con $n_i = n/2$ nodos del nodo raíz utilizando la misma estrategia.
3. Generar el subárbol derecho con $n_d = n - n_i - 1$ nodos del nodo raíz utilizando la misma estrategia.

En este caso el árbol no va a ser de búsqueda. Simplemente, un árbol binario de claves enteras generadas aleatoriamente.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TOPE 99

typedef int TipoElemento;
typedef struct nodo
{
  TipoElemento dato;
  struct nodo *izdo, *dcho;
}Nodo;
typedef Nodo* ArbolBinario;
```

```
ArbolBinario crearNodo(TipoElemento x)
{
  ArbolBinario a;
  a = (ArbolBinario) malloc(sizeof(Nodo));
  a -> dato = x;
  a -> dcho = a ->izdo = NULL;
  return a;
}

ArbolBinario arbolFibonacci(int n);
void dibujarArbol(ArbolBinario raiz, int h);

void main()
{
  ArbolBinario raiz;
  int n;
  randomize();
  printf("\n Número de nodos del árbol de Fibonacci: ");
  scanf("%d", &n);

  raiz = arbolFibonacci(n);
  printf("\n\t Árbol equilibrado de n nodos, con máxima altura \n");
  dibujarArbol(raiz, 0);
}

ArbolBinario arbolFibonacci(int n)
{
  int nodosIz, nodosDr;
  TipoElemento clave;
  ArbolBinario nuevo;

  if (n == 0)
    return NULL;
  else
  {
    nodosIz = n / 2;
    nodosDr = n - nodosIz -1;
      /* Se crea el nuevo nodo */
    clave = random(TOPE) + 1;
    nuevo = crearNodo(clave);
    nuevo -> izdo = arbolFibonacci(nodosIz);
    nuevo -> dcho = arbolFibonacci(nodosDr);

    return nuevo;
  }
}

void dibujarArbol(ArbolBinario raiz, int h)
{
  /* pinta las claves del arbol de Fibonacci; h establece
     una separacion entre nodos   */
  int i;
```

```
    if (raiz != NULL)
    {
      dibujarArbol(raiz -> izdo, h+1);

      for (i = 1; i <= h; i++) printf(" ");
      printf("%d\n",raiz -> dato);

      dibujarArbol(raiz -> dcho, h+1);
    }
  }
```

15.3. INSERCIÓN EN ÁRBOLES DE BÚSQUEDA EQUILIBRADOS. *ROTACIONES*

Los árboles equilibrados, árboles AVL, son árboles de búsqueda y por consiguiente la inserción se ha de hacer siguiendo el *camino de búsqueda*. El proceso de inserción se hace aplicando el algoritmo de inserción de una nueva clave en un árbol de búsqueda, comparando la nueva clave con la clave del nodo raíz, y siguiendo por la rama izquierda o derecha según sea menor o mayor; termina insertándose como nodo hoja. Esta operación, como ha quedado demostrado al determinar la altura en el peor de los casos, tiene una complejidad logarítmica. Sin embargo, la nueva inserción puede hacer que aumente la altura de una rama, de manera que cambie el factor de equilibrio del nodo raíz de dicha rama. Este hecho hace necesario que el algoritmo de inserción, *regrese* por el *camino de búsqueda* actualizando el factor de equilibrio de los nodos. La Figura 15.5 muestra un árbol equilibrado y el mismo árbol justo después de la inserción de una nueva clave que provoca se *rompa* la condición de balanceo.

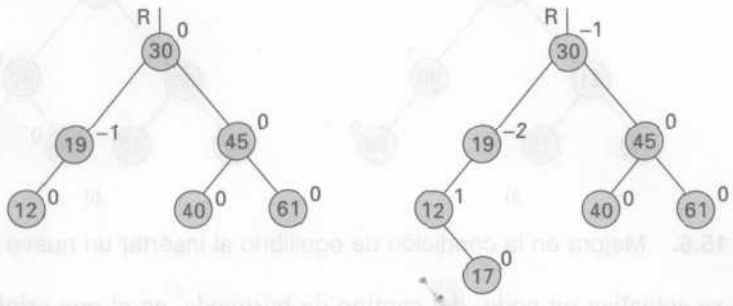


Figura 15.5. Árbol equilibrado; el mismo después de insertar la clave 17.

A recordar

Una inserción de una nueva clave, o una eliminación, puede destruir el criterio de equilibrio de varios nodos del árbol. Se debe recuperar la condición de equilibrio del árbol antes de dar por finalizada la operación para que el árbol siga siendo equilibrado.

La estructura del nodo en un árbol equilibrado es, en lo fundamental, la misma que en un árbol de búsqueda. Para determinar si el árbol está equilibrado debe manejarse información relativa al

balanceo o factor de equilibrio de cada nodo del árbol. Por esta razón, se añade, al tipo de datos que representa cada nodo, un campo más: el *factor de equilibrio* (`fe`). *Este campo puede tomar los valores:* `-1, 0, +1`.

```
struct nodo
{
  TipoElemento dato ;
  int fe ;
  struct nodo *izdo, *dcho;
};
typedef struct nodo Nodo;
```

15.3.1. Proceso de inserción de un nuevo nodo

Se considera un árbol binario de búsqueda equilibrado, el árbol puede estar vacío y va a ser formado. Se inserta un nuevo nodo, para lo que se aplica el algoritmo de inserción en un árbol de búsqueda, este sigue el *camino de búsqueda* hasta llegar al fondo del árbol y se inserta como nodo *hoja* y con *factor de equilibrio* `0`. *Pero el proceso no puede terminar, hay que recorrer el camino de búsqueda en sentido contrario, hacia la raíz, para actualizar el criterio de equilibrio, el campo adicional factor de equilibrio.* Después de una inserción sólo los nodos que se encuentran en el camino de búsqueda pueden haber cambiado el factor de equilibrio.

La actualización del factor de equilibrio (`fe`) puede hacer que este *mejore*. Esto ocurre cuando un nodo está descompensado a la izquierda y se inserta el nuevo nodo en la rama izquierda, al crecer en altura dicha rama el `fe` se hace `0`, se ha *mejorado* el equilibrio. La Figura 15.6 muestra el árbol *a)* en el que el nodo `90` tiene `fe = 1`; en el árbol *b)*, después de insertar el nodo con clave `60`, el nodo `90` tiene `fe = 0`.

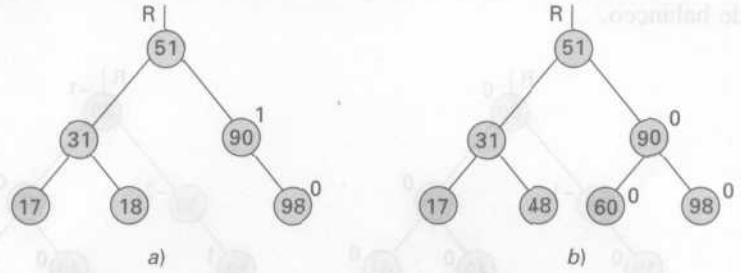


Figura 15.6. Mejora en la condición de equilibrio al insertar un nuevo nodo con clave 60.

Cuando se actualiza un nodo, del camino de búsqueda, en el que originalmente las ramas izquierda y derecha del árbol tienen la misma altura ($h_{Ri} = h_{Rd}$), se haya insertado en la rama izquierda o en la rama derecha no va a ser causa de romper el equilibrio.

Al actualizar el nodo cuyas ramas izquierda y derecha del árbol tienen altura diferente, $|h_{Ri}-h_{Rd}|=1$, si se inserta el nodo en la rama más alta, entonces se rompe el criterio de equilibrio del árbol, la diferencia de altura pasa a ser 2 y es necesario reestructurarlo.

Hay cuatro casos que se deben tener en cuenta al reestructurar un nodo `A`, según el lugar donde se haya hecho la inserción:

1. Inserción en el subárbol izquierdo de la rama izquierda de `A`.
2. Inserción en el subárbol derecho de la rama izquierda de `A`.
3. Inserción en el subárbol derecho de la rama derecha de `A`.
4. Inserción en el subárbol izquierdo de la rama derecha de `A`.

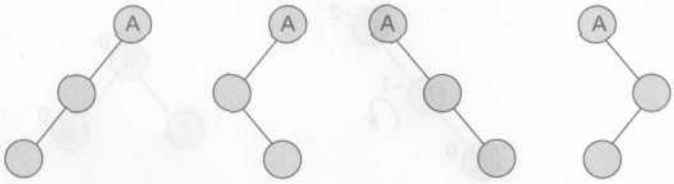


Figura 15.7. *Cuatro tipos de reestructuraciones del equilibrio de un nodo.*

El primer y tercer caso (*izquierda-izquierda, derecha-derecha*) se resuelven con una *rotación simple*. El segundo y cuarto caso (*izquierda-derecha, derecha-izquierda*) se resuelven con una *rotación doble*.

La rotación simple implica a dos nodos, el nodo `A` (nodo con `|fe| = 2`) y el descendiente izquierdo o derecho según el caso. En la rotación doble están implicados los tres nodos, el nodo `A`, nodo descendiente izquierdo y el descendiente derecho de este; o bien el caso simétrico, nodo `A`, descendiente derecho y el descendiente izquierdo de este.

A recordar

Una reestructuración de los nodos implicados en la violación de criterio de equilibrio, ya sea una rotación simple o doble, hace que se recupere el equilibrio en todo el árbol, no siendo necesario seguir analizando los nodos del camino de búsqueda.

15.3.2. Rotación simple

Una vez que se ha insertado un nodo en un árbol de búsqueda equilibrado, el proceso de regreso termina cuando se llega a la raíz del árbol, o cuando se realiza la reestructuración en un nodo del mismo; en cuyo caso no es necesario determinar el *factor de equilibrio* de los restantes nodos, debido a que dicho factor queda como el que tenía antes de la inserción, ya que el efecto de la reestructuración hace que no aumente la altura.

La violación del factor de equilibrio se resuelve con una reestructuración en la que se realiza un desplazamiento particular de los nodos implicados, también denominado *rotación* de los nodos. Se distingue entre *rotación simple* y *rotación doble*.

La *rotación simple* resuelve la violación del equilibrio de un nodo *izquierda-izquierda*, simétrica a la *derecha-derecha*. El árbol de la Figura 15.8*a* tiene el nodo `C` con factor de equilibrio -1; el árbol de la Figura 15.8*b* es el resultado de insertar el nodo `A`. Resulta que ha crecido la altura de la rama izquierda, es un desequilibrio izquierda-izquierda que se resuelve con una rotación simple, que en este caso se puede denominar rotación `II`. La Figura 15.9 es el árbol resultante de la rotación, en el que el nodo `B` se ha convertido en la raíz, el nodo `C` su rama derecha y el nodo `A` continúa como rama izquierda. Con estos movimientos el árbol sigue siendo de búsqueda y se equilibra.

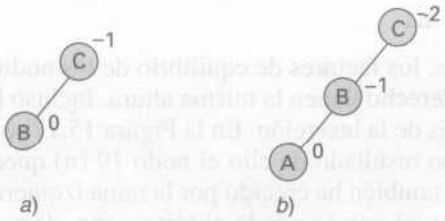


Figura 15.8. Árbol binario AVL y árbol después de insertar nueva clave por la izquierda.

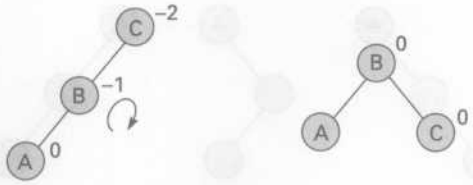


Figura 15.9. Árbol binario después de rotación simple `II`.

En la Figura 15.10 se muestra el otro caso de violación de la condición de equilibrio que se resuelve con una rotación simple. Inicialmente, el nodo `A` tiene como factor de equilibrio +1, al insertar el nodo `C` y después de actualizar, el factor de equilibrio de `A` pasa a ser +2, se ha insertado por la derecha y por consiguiente ha crecido la altura de la rama derecha. La Figura 15.11 muestra la resolución de este desequilibrio, una rotación simple que se puede denominar rotación `DD`. En el árbol resultante de la rotación el nodo `B` se ha convertido en la raíz, el nodo `A` es su rama izquierda y el nodo `C` continúa como rama derecha. Con estos movimientos el árbol sigue siendo de búsqueda y se equilibra.

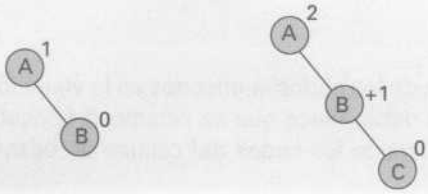


Figura 15.10. Árbol binario AVL y árbol después de insertar nueva clave por la derecha.

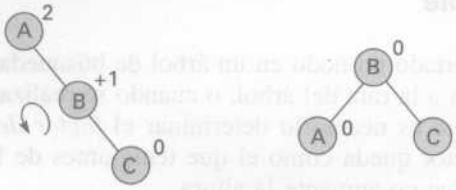


Figura 15.11. Árbol binario después de rotación simple `DD`.

15.3.3. Movimiento de punteros en la rotación simple

Los cambios necesarios descritos en la rotación simple afectan a dos nodos, el tercero no se modifica, es necesario sólo una rotación. En el caso de ser la rotación simple a la *izquierda*, rotación `II`, los ajustes necesarios de los punteros suponiendo `n` el puntero al nodo problema y `n1` el puntero al nodo de su rama izquierda:

```
n->izdo        = n1->dcho
n1->dcho       = n
n              = n1
```

Realizada la rotación, los factores de equilibrio de los nodos que intervienen siempre son 0, los subárboles izquierdo y derecho tienen la misma altura. Incluso la altura del subárbol implicado es la misma que antes, después de la inserción. En la Figura 15.12 se muestra un árbol binario después de insertar la clave 5. Como resultado de ello el nodo 19 (`n`) queda desequilibrado por la izquierda y como su nodo izquierdo también ha crecido por la rama izquierda, la rotación `II` resuelve el problema y obtiene un árbol en el que se puede observar que, después de la rotación, la altura neta del subárbol no ha cambiado.

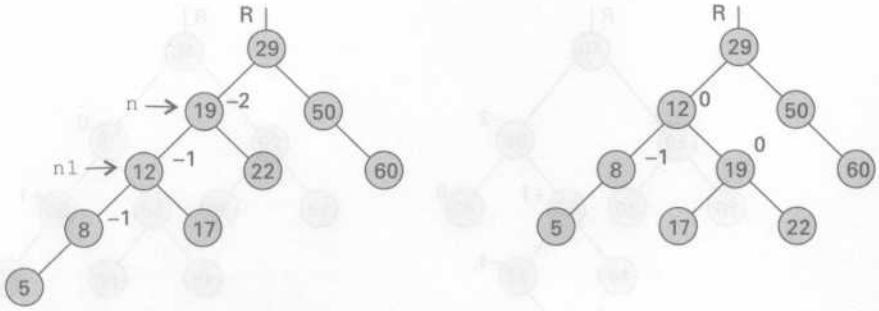


Figura 15.12. Rotación simple a izquierda en un árbol después de insertar la clave 5.

En el caso de que la rotación simple sea a *derecha*, rotación `DD`, los cambios en los punteros del nodo `n` (con factor de equilibrio +2) y `n1`, puntero al nodo de su rama derecha:

```
n->dcho        = n1->izdo
n1->izdo       = n
n              = n1
```

Realizada la rotación, los factores de equilibrio de los nodos que intervienen es 0. Se puede observar que estos ajustes son los simétricos a los realizados en la rotación `II`.

15.3.4. Rotación doble

Con los movimientos de punteros que se realizan en la rotación simple no es posible resolver todos los casos de violación del criterio de equilibrio. En la Figura 15.13*a* se observa un árbol de búsqueda desequilibrado, con factores de equilibrio +2, -1 y 0. La Figura 15.13*b* aplica la rotación simple, rotación `DD`. La única solución consiste en *subir* el nodo 40 como raíz del subárbol, como rama izquierda situar al nodo 30 y como rama derecha el nodo 60, la altura del subárbol resultante es la misma que antes de insertar. Se ha realizado una rotación doble, derecha-izquierda, en la que intervienen los tres nodos.

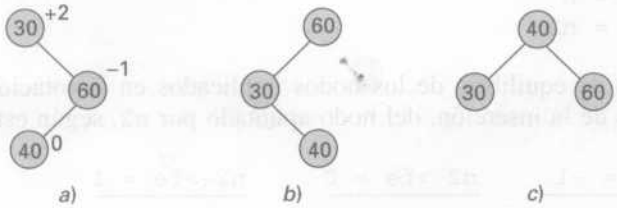


Figura 15.13. *a*) Árbol después de insertar la clave 40. *b*) Rotación `DD`. *c*) Rotación doble que resuelve el problema.

La Figura 15.14*a* muestra un árbol binario de búsqueda después de insertar la clave 60. Al *volver* por el camino de búsqueda para actualizar los factores de equilibrio, el nodo 75 pasa a tener `fe = -1` (se ha insertado por su izquierda), el nodo 50 pasa a tener `fe = +1` y el nodo 80 tendrá `fe = +2`. Es el caso simétrico al descrito en la Figura 15.13, se reestablece el equilibrio con una rotación doble, simétrica con respecto a la anterior como se muestra en la Figura 15.14*b*.

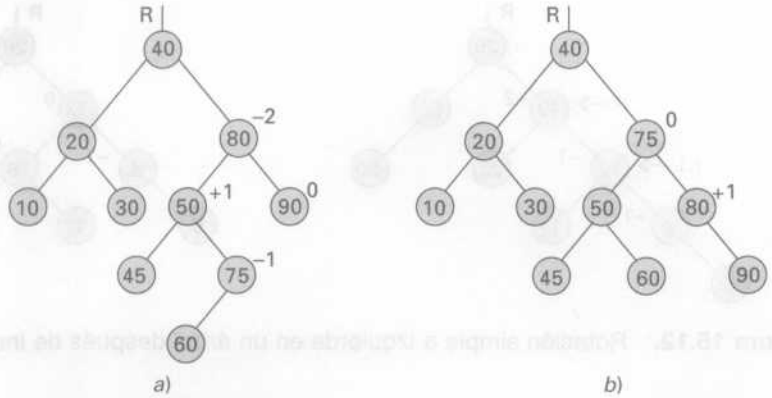


Figura 15.14. *a*) Árbol de equilibrado búsqueda después de insertar clave 60. *b*) Rotación doble, izquierda derecha.

> **A recordar**
>
> La rotación doble resuelve dos casos simétricos, se pueden denominar rotación `ID` y rotación `DI`. En la rotación doble hay que mover los punteros de tres nodos, el nodo padre, el descendiente y el *descendiente del descendiente* por la rama contraria.

15.3.5. Movimiento de punteros en la rotación doble

Los cambios necesarios descritos en la rotación simple afectan a tres nodos, el nodo problema apuntado por `n`, el descendiente por la rama desequilibrada apuntado por `n1`, y el descendiente de `n1` (por la izquierda o la derecha, según el tipo de rotación doble) apuntado por `n2`. Los dos casos simétricos de rotación doble son la rotación *izquierda-derecha* (rotación `ID`) y la rotación *derecha-izquierda* (rotación `DI`), en ambos el nodo `n2` pasa a ser la raíz del nuevo subárbol.

Los movimientos de los punteros para realizar la rotación `ID`:

```
n1->dcho  = n2->izdo
n2->izdo  = n1
n->izdo   = n2->dcho
n2->dcho  = n
n         = n2
```

Los factores de equilibrio de los nodos implicados en la rotación `ID` depende del factor de equilibrio, antes de la inserción, del nodo apuntado por `n2`, según esta tabla:

Si	`n2->fe = -1`	`n2->fe = 0`	`n2->fe = 1`
`n->fe` =	1	0	0
`n1->fe` =	0	0	-1
`n2->fe` =	0	0	0

Los movimientos de los punteros para realizar la rotación `DI` (observar la simetría en los movimientos de los punteros):

```
n1->izdo = n2->dcho
n2->dcho = n1
```

```
n->dcho    = n2->izdo
n2->izdo   = n
n          = n2
```

Los factores de equilibrio de los nodos implicados en la rotación `DI` depende del factor de equilibrio, antes de la inserción, del nodo apuntado por `n2`, según la tabla:

Si	`n2->fe = -1`	`n2->fe = 0`	`n2->fe = 1`
`n->fe =`	0	0	-1
`n1->fe =`	1	0	0
`n2->fe =`	0	0	0

A recordar

La complejidad del algoritmo de inserción de una clave en un árbol de búsqueda AVL es la suma de la complejidad para *bajar* al nivel de las hojas (`O(log n)`) más la complejidad en el peor de los casos de la vuelta por el camino de búsqueda para actualizar el factor de equilibrio de los nodos, que es `O(log n)`, más la complejidad de los movimientos de los punteros en la rotación, que tiene complejidad constante. En definitiva, la complejidad de la inserción es `O(log n)`, complejidad logarítmica.

15.4. REALIZACIÓN EN C DE LA INSERCIÓN CON BALANCEO Y ROTACIONES

La realización en C del proceso de inserción de un nuevo nodo primero utiliza el algoritmo recursivo desarrollado en los árboles de búsqueda. Una vez insertado, se activa un *flag* (indicador, bandera) si ha crecido en altura para *regresar* por el camino de búsqueda y determinar los nuevos factores de equilibrio de los nodos que forman el camino que va hacia la raíz. Cuando la inserción se ha realizado por la rama izquierda del nodo, la altura crece por la izquierda y, por tanto, disminuye en 1 el factor de equilibrio. En el caso de que la inserción se haga por la rama derecha, el factor de equilibrio aumenta en 1. Si la altura del subárbol no aumenta, el proceso termina. Si se produce un desequilibrio, se realiza la correspondiente rotación y el proceso termina ya que cualquier rotación tiene la propiedad de que la altura del subárbol resultante es la misma que antes de la inserción.

A continuación se escribe el código en C de los cuatro tipos de rotaciones y de la operación de inserción. En cuanto a las funciones que implementa las rotaciones se incluye la actualización del *factor de equilibrio en casos que se dan únicamente en la eliminación de una clave que se estudia en* el siguiente apartado.

Rotaciones

```
void rotacionII(Nodo** n, Nodo* n1)
{
  (*n) -> izdo = n1 -> dcho;
  n1 -> dcho = (*n);
                        /* actualización de los factores de equilibrio */
  if (n1 -> fe == -1)          /* en la inserción siempre se cumple */
  {
    (*n) -> fe = 0;
    n1 -> fe = 0;
```

```
    }
    else
    {
      (*n) -> fe = -1;
      n1 -> fe = 1;
    }

    (*n) = n1;
  }

  void rotacionDD(Nodo** n, Nodo * n1)
  {
    (*n) -> dcho = n1 -> izdo;
    n1 -> izdo = (*n);
                          /* actualización de los factores de equilibrio */
    if (n1 -> fe == +1)          /* en la inserción siempre se cumple */
    {
      (*n) -> fe = 0;
      n1 -> fe = 0;
    }
    else
    {
      (*n) -> fe = +1;
      n1 -> fe = -1;
    }

    (*n) = n1;
  }

  void rotacionID(Nodo** n, Nodo * n1)
  {
    Nodo* n2;

    n2 = n1 -> dcho;
    (*n) -> izdo = n2 -> dcho;
    n2 -> dcho = *n;
    n1 -> dcho = n2 -> izdo;
    n2 -> izdo = n1;
                          /* actualización de los factores de equilibrio */
    if (n2 -> fe == +1)
      n1 -> fe = -1;
    else
      n1 -> fe = 0;

    if (n2 -> fe == -1)
      (*n) -> fe = 1;
    else
      (*n) -> fe = 0;

    n2 -> fe = 0;
    (*n) = n2;
```

```
}
void rotacionDI(Nodo** n, Nodo* n1)
{
  Nodo* n2;

  n2 = n1 -> izdo;
  (*n) -> dcho = n2 -> izdo;
  n2 -> izdo = *n;
  n1 -> izdo = n2 -> dcho;
  n2 -> dcho = n1;
                        /* actualización de los factores de equilibrio */
  if (n2 -> fe == +1)
    (*n) -> fe = -1;
  else
    (*n) -> fe = 0;

  if (n2 -> fe == -1)
    n1 -> fe = 1;
  else
    n1 -> fe = 0;

  n2 -> fe = 0;
  (*n) = n2;
}
```

Inserción con balanceo

```
void insertarEquilibrado (Nodo** raiz, TipoElemento clave, int* h)
{
  Nodo* n1;

  if (!(*raiz))
  {
    *raiz = crearNodo(clave);
    *h = 1;
  }
  else if (clave < (*raiz) -> dato)
  {
    insertarEquilibrado(&((*raiz) -> izdo), clave, h);
                                  /* regreso por los nodos del camino */
    if (*h)
    {
          /* reduce en 1 el fe al aumentar altura de rama izquierda */
      switch ((*raiz) -> fe)
      {
         case  1:  (*raiz) -> fe = 0;
                   (*h) = 0;
                   break;

         case  0:  (*raiz) -> fe = -1;
                   break;
```

```
            case -1:          /* viola el equilibrio, rotación izquierda */
              n1 = (*raiz) -> izdo;
              if (n1 -> fe == -1)
                rotacionII(raiz,n1);
              else
                rotacionID(raiz,n1);

              (*h) = 0;
          }
        }
      }
      else if (clave > (*raiz) -> dato)
      {
        insertarEquilibrado(&((*raiz) -> dcho), clave, h);
                                        /* regreso por los nodos del camino */
        if (*h)
        {
                    /* incrementa en 1 el fe al aumentar altura de rama derecha */
          switch ((*raiz) -> fe)
          {
            case 1:  /* viola el equilibrio, rotación derecha */
              n1 = (*raiz) -> dcho;
              if (n1 -> fe == +1)
                rotacionDD(raiz,n1);
              else
                rotacionDI(raiz,n1);

              (*h) = 0;
              break;

            case 0: (*raiz) -> fe = +1;
              break;

            case -1: (*raiz) -> fe = 0;
              (*h) = 0;
              break;
          }
        }
      }

      else
      {
        puts("En arboles de búsqueda no puede haber claves repetidas." );
        (*h) = 0;
      }
    }
```

Ejercicio 15.2

Se pretende formar un árbol binario de búsqueda equilibrado de altura 5. Las claves de búsqueda son enteros positivos de valor máximo 555. Una vez formado el árbol, mostrar las claves en orden creciente y el número de nodos de que consta el árbol.

La formación del árbol equilibrado se puede hacer con repetidas llamadas a la función `insertarEquilibrado()`, que ya realiza las llamadas, cuando es necesario, a las *rotaciones*. La condición para terminar la formación del árbol está expuesta en el enunciado: la altura del árbol sea igual a 5. Por ello se escribe la función `altura()` para determinar dicho parámetro. También se escribe la función `visualizar()`, que es un recorrido en *inorden*, para mostrar las claves en orden creciente y a la vez contar los nodos visitados.

Las funciones que implementan las rotaciones y la inserción no se vuelven a escribir, sólo sus correspondientes prototipos.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <time.h>

#define TOPE 555
#define MAX(x,y) ((x > y) ? (x) : (y))
typedef int TipoElemento;

struct nodo {
  TipoElemento dato;
  int fe;
  struct nodo *izdo, *dcho;
};
typedef struct nodo Nodo;

Nodo* crearNodo(TipoElemento clave);
void insertarEquilibrado (Nodo** raiz, TipoElemento clave, int *h);
int visualizar (Nodo* r);
int altura(Nodo* r);

  /* Rotaciones */
void rotacionII(Nodo** n, Nodo* n1);
void rotacionDD(Nodo** n, Nodo* n1);
void rotacionID(Nodo** n, Nodo* n1);
void rotacionDI(Nodo** n, Nodo* n1);

void main()
{
  TipoElemento clave;
  int numNodos;
  int creceAltura;
  Nodo* raiz;
  clrscr();randomize();
  raiz = NULL;
  while (altura(raiz) < 5)
  {
    insertarEquilibrado(&raiz, random(TOPE)+1,&creceAltura);
  }
  numNodos = visualizar(raiz);
```

```
    printf("\n Número de nodos: %d \n", numNodos);
  }
  Nodo* crearNodo(TipoElemento clave)
  {
    Nodo* t;
    t = (Nodo*) malloc(sizeof(Nodo));
    t -> dato = clave;
    t -> izdo = t -> dcho = NULL;
    t -> fe = 0;
    return t;
  }

  int visualizar (Nodo* r)
  {
    if (r)
    {
      int cuantosIzquierda, cuantosDerecha;

      cuantosIzquierda = visualizar(r -> izdo);
      printf(" %d ",r->dato);
      cuantosDerecha = visualizar(r -> dcho);
      return cuantosIzquierda + cuantosDerecha + 1;
    }
    else
      return 0;
  }

  int altura(Nodo* r)
  {
    if (r)
      return MAX(altura(r -> izdo), altura(r -> dcho)) + 1;
    else
      return 0;
  }
```

15.5. ELIMINACIÓN DE UN NODO EN UN ÁRBOL EQUILIBRADO

La operación de eliminación consiste en eliminar un nodo con cierta clave de un árbol de búsqueda equilibrado. Evidentemente, el árbol resultante debe seguir siendo un árbol de búsqueda balanceado ($|h_{Ri} - h_{Rd}| <= 1$).

El algoritmo de eliminación puede descomponerse en dos partes diferenciadas. La primera sigue la estrategia de la eliminación en árboles de búsqueda. La segunda es la actualización del *factor de equilibrio*, para lo que recorre el *camino de búsqueda* hacia la raíz, actualizando el factor de equilibrio de los nodos. Ahora, al eliminarse un nodo, la altura de la rama en que se encuentra disminuye en 1. Si después de la actualización de un nodo ocurre que se *viola* la condición de equilibrio, el `fe = ± 2`, hay que restaurar del equilibrio con una rotación simple o doble. En la Figura 15.15*a* se muestra un árbol equilibrado con el factor de equilibrio de cada nodo, en la Figura 15.15*b* el árbol después de eliminar la clave 15, al actualizar el factor de equilibrio del nodo 28 pasa a valer +2 ya que se ha eliminado por su rama izquierda, hay que restablecer el equilibrio con una rotación, en este caso con una rotación doble.

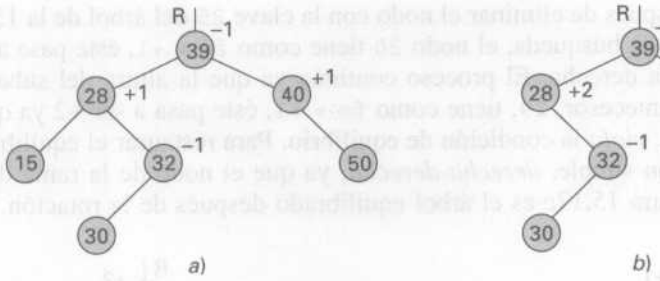


Figura 15.15. *a*) Árbol de búsqueda equilibrado. *b*) Después de eliminar la clave 15 y actualizar los factores de equilibrio.

15.5.1. Algoritmo para eliminar un nodo de un árbol equilibrado

En el algoritmo para eliminar un nodo que contiene una cierta a clave de un árbol de búsqueda lo primero que se hace es buscar el nodo, para lo que se sigue el camino de búsqueda. A continuación se procede a eliminar el nodo, se distinguen los siguientes casos:

1. El nodo a borrar es un nodo hoja, o con un único descendiente. Entonces, simplemente se suprime, o bien se sustituye por su descendiente.
2. El nodo a eliminar tiene dos subárboles. En este caso se busca el nodo más a la derecha del subárbol izquierdo, es decir, el de *mayor clave en el subárbol de los menores*, se copia este en el nodo a eliminar y por último se elimina el nodo copiado (que tiene las características del caso primero).

Una vez eliminado el nodo, el algoritmo tiene que prever actualizar los factores de equilibrio de los nodos que han formado el camino de búsqueda ya que la altura de alguna de las dos ramas ha decrecido. Por consiguiente, *regresa* por el camino de búsqueda, hacia la raíz, calculando los nuevos factores de equilibrio de los nodos visitados. No siempre es necesario recorrer todo el camino de regreso, si se actualiza un nodo con factor de equilibrio 0, el nuevo será ± 1, pero la altura neta del subárbol con raíz el nodo actualizado no ha cambiado, entonces el algoritmo termina ya que los factores de equilibrio de los nodos restantes no cambia. La Figura 15.16*a* muestra un árbol equilibrado, en la Figura 15.16*b* el árbol resultante después de eliminar el nodo con la clave `90`, al volver por el camino de búsqueda el nodo `70` su factor de equilibrio pasa a ser 0, el nodo `60` tiene como factor 0 se actualiza y pasa a ser −1 (ha disminuido su rama derecha), su altura neta no ha cambiado, sigue siendo 3, por lo que el algoritmo termina.

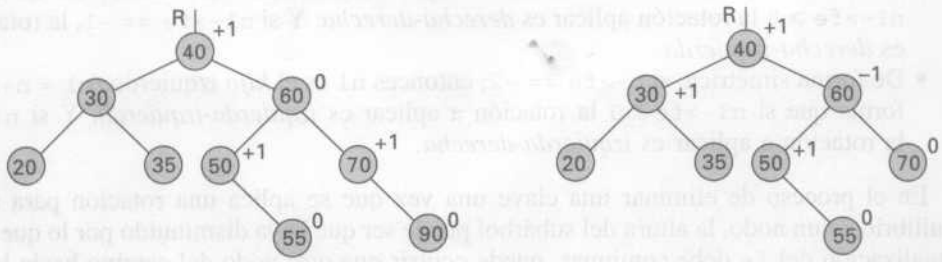


Figura 15.16. *a*) Árbol de búsqueda equilibrado. *b*) El árbol después de eliminar el nodo 90.

Otro caso a considerar en la parte de actualización es que un nodo tenga como `fe` = ± `1`, y la eliminación se haya hecho por la rama más alta, entonces el equilibrio mejora, pasa a ser cero. Ahora bien, en este caso la altura del subárbol ha disminuido, por lo que el algoritmo debe seguir retrocediendo por la nodos por si es necesario una reestructuración. En la Figura 15.17*b* se tiene el

árbol de búsqueda después de eliminar el nodo con la clave `25` del árbol de la 15.17*a*. Al actualizar los nodos del camino de búsqueda, el nodo `20` tiene como `fe = +1`, éste pasa a ser 0 ya que se ha eliminado por la rama derecha. El proceso continúa ya que la altura del subárbol de raíz `20` ha disminuido, el nodo antecesor, `29`, tiene como `fe = +1`, éste pasa a ser +2 ya que se ha eliminado por su rama izquierda, *viola* la condición de equilibrio. Para restaurar el equilibrio es necesario, en este caso, una rotación simple, *derecha-derecha*, ya que el nodo de la rama derecha de `29` tiene como `fe = 0`. La Figura 15.17*c* es el árbol equilibrado después de la rotación.

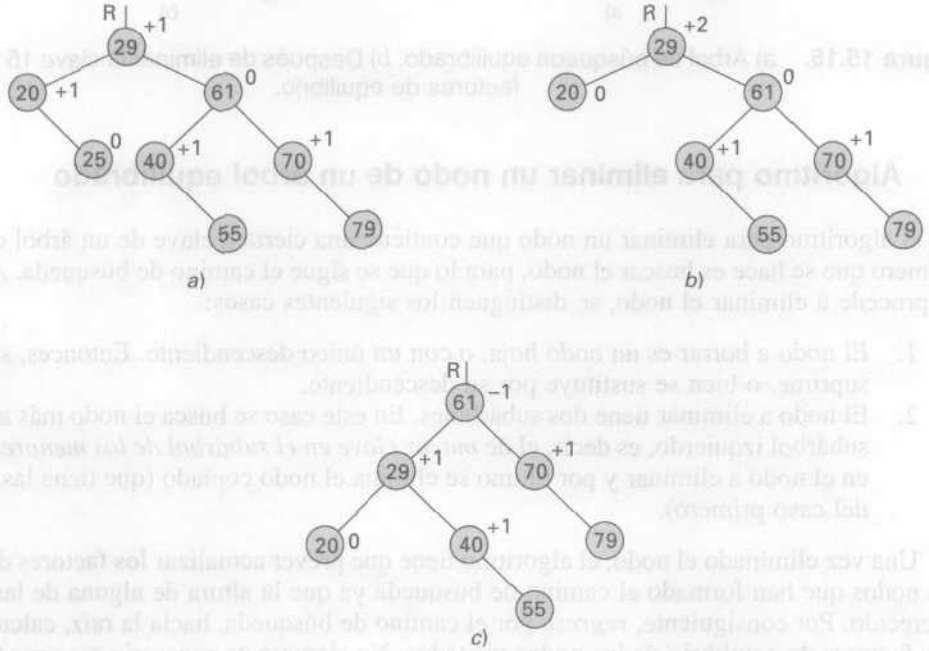


Figura 15.17. *a*) Árbol de búsqueda equilibrado. *b*) El árbol después de eliminar el nodo `25`. *c*) Árbol una vez realizada la rotación *derecha-derecha*.

Los nodos que tienen como `fe = ± 1`, al actualizar el factor de equilibrio éste puede pasar a `± 2`, hay que reestructurarlos con una rotación simple o doble. El tipo concreto de rotación depende del `fe` del nodo problema, apuntado por `n`, y del nodo descendiente `n1`:

- Si `n->fe == +2`, entonces `n1` es el *hijo* derecho (`n1 = n->dcho`) de tal forma que si `n1->fe ≥ 0` la rotación aplicar es *derecha-derecha*. Y si `n1->fe == -1`, la rotación a aplicar es *derecha-izquierda*.
- De forma simétrica, si `n->fe == -2`, entonces `n1` es el *hijo* izquierdo (`n1 = n->izdo`) de tal forma que si `n1->fe ≤ 0` la rotación a aplicar es *izquierda-izquierda*. Y si `n1->fe == +1`, la rotación a aplicar es *izquierda-derecha*.

En el proceso de eliminar una clave una vez que se aplica una rotación para restablecer el equilibrio de un nodo, la altura del subárbol puede ser que haya disminuido por lo que el proceso de actualización del `fe` debe continuar; puede ocurrir que otro nodo del camino hacia la raíz viole la condición de equilibrio y sea necesario aplicar otra rotación. La Figura 15.18*b* es un árbol de búsqueda después de haber eliminado la clave `21`, y con la actualización de los factores de equilibrio, el nodo `43` está desequilibrado, hay que aplicar una rotación *derecha-derecha*. La Figura 15.18*c* muestra el árbol después de la rotación y la posterior actualización del factor de equilibrio del nodo `70`, que también exige aplicar otra rotación, en este caso *derecha-izquierda*. La Figura 15.18*d* es el árbol equilibrado después de la última rotación y fin del algoritmo.

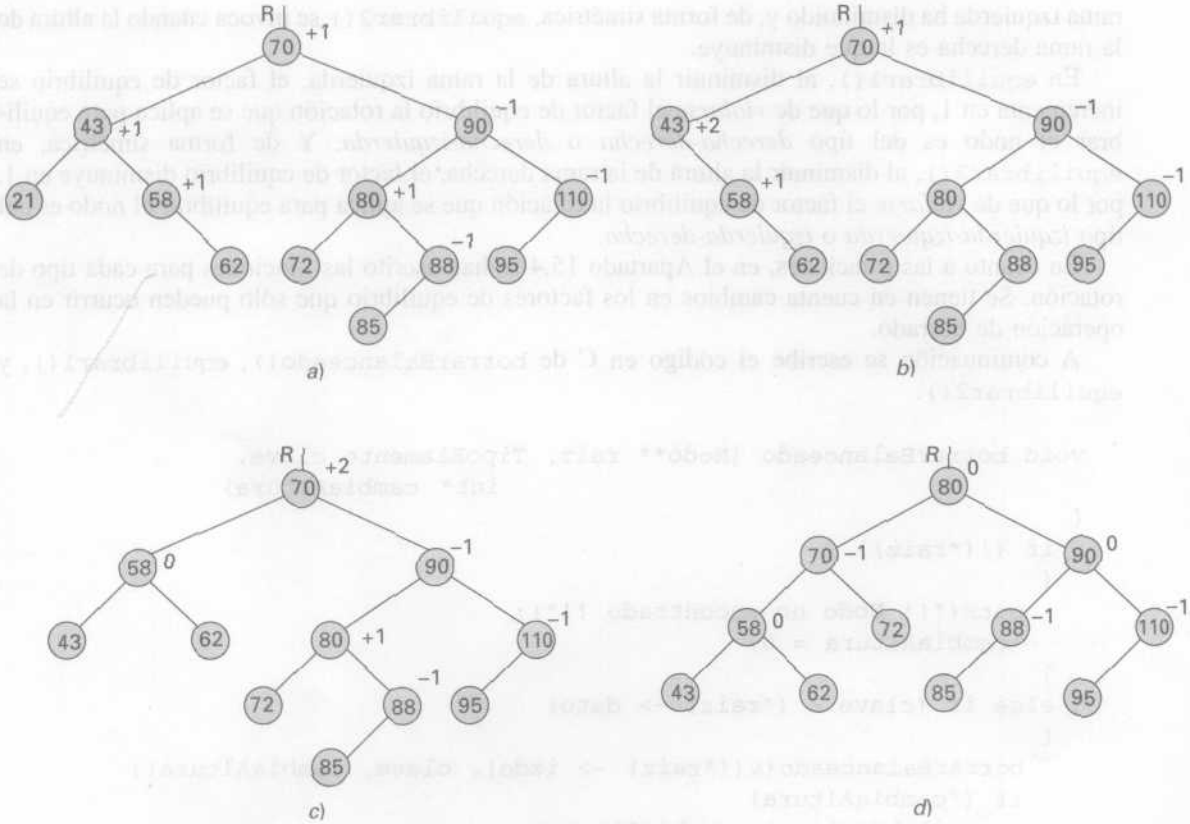


Figura 15.18. *a*) Árbol de búsqueda equilibrado. *b*) El árbol después de eliminar el nodo 21. *c*) Árbol una vez realizada la rotación *derecha-derecha*. *d*) Árbol después de la nueva rotación, *derecha-izquierda*.

A recordar

El algoritmo de eliminación de una clave en un árbol de búsqueda AVL puede necesitar aplicar más de una rotación para que el árbol resultante siga siendo equilibrado. En cuanto a la complejidad, es la suma de la complejidad para encontrar el nodo, `O(log n)`, más la complejidad de la *vuelta por el camino de búsqueda*, `O(log n)`, más la complejidad de los movimientos de los punteros en la rotación o rotaciones, que es una complejidad constante. En definitiva, la complejidad de la eliminación puede considerarse logarítmica.

15.5.2. Realización en C de la operación de eliminación en un árbol equilibrado

La función que implementa la operación de eliminación requiere utilizar un argumento lógico para indicar que se ha producido un cambio en la altura de una rama. Entonces el parámetro `cambiaAltura` se va a activar cuando la altura del subárbol disminuya, debido a la eliminación de un nodo, o bien porque al aplicar las rotaciones se haya reducido la altura del subárbol. Además para aislar el proceso de los diferentes casos que ocurren cuando disminuye la altura se introducen dos funciones, `equilibrar1()` y `equilibrar2()`. La primera se invoca cuando la altura de la

rama izquierda ha disminuido y, de forma simétrica, `equilibrar2()` se invoca cuando la altura de la rama derecha es la que disminuye.

En `equilibrar1()`, al disminuir la altura de la rama izquierda, el factor de equilibrio se incrementa en 1, por lo que de *violarse* el factor de equilibrio la rotación que se aplica para equilibrar el nodo es del tipo *derecha-derecha* o *derecha-izquierda*. Y de forma simétrica, en `equilibrar2()`, al disminuir la altura de la rama derecha, el factor de equilibrio disminuye en 1, por lo que de *violarse* el factor de equilibrio la rotación que se aplica para equilibrar el nodo es del tipo *izquierda-izquierda* o *izquierda-derecha*.

En cuanto a las rotaciones, en el Apartado 15.4 se han escrito las funciones para cada tipo de rotación. Se tienen en cuenta cambios en los factores de equilibrio que sólo pueden ocurrir en la operación de borrado.

A continuación se escribe el código en C de `borrarBalanceado()`, `equilibrar1()`, y `equilibrar2()`.

```
void borrarBalanceado (Nodo** raiz, TipoElemento clave,
                                    int* cambiaAltura)
{
  if (!(*raiz))
  {
    puts("!! Nodo no encontrado !!");
    *cambiaAltura = 0;
  }
  else if (clave < (*raiz) -> dato)
  {
    borrarBalanceado(&((*raiz) -> izdo), clave, cambiaAltura);
    if (*cambiaAltura)
      equilibrar1(raiz,cambiaAltura);
  }
  else if (clave > (*raiz) -> dato)
  {
    borrarBalanceado(&((*raiz) -> dcho), clave, cambiaAltura);
    if (*cambiaAltura)
      equilibrar2(raiz,cambiaAltura);
  }
  else                                          /* Nodo encontrado */
  {
    Nodo* q;                          /* puntero al nodo a suprimir */
    q = (*raiz);

    if (q -> izdo == NULL)
    {
      (*raiz) = q -> dcho;
      *cambiaAltura = 1;
    }
    else if (q -> dcho == NULL)
    {
      (*raiz) = q -> izdo;
      *cambiaAltura = 1;
    }
    else
    {                       /* tiene rama izquierda y derecha */
      reemplazar(&q,&(q -> izdo),cambiaAltura);
```

```
      if (*cambiaAltura)
        equilibrar1(raiz,cambiaAltura);
    }
    free(q);
  }

}

void reemplazar(Nodo** n,Nodo** act,int * cambiaAltura)
{
  if ((*act) -> dcho)
  {
    reemplazar(n,&((*act) -> dcho),cambiaAltura);
    if (*cambiaAltura)
      equilibrar2(act,cambiaAltura);
  }
  else
  {
    (*n) -> dato = (*act) -> dato;
    (*n) = *act;
    (*act) = (*act) -> izdo;
    *cambiaAltura = 1;
  }
}

void equilibrar1(Nodo** n, int* cambiaAltura)
{
  Nodo* n1;

  switch ((*n) -> fe)
  {
    case -1: (*n) -> fe = 0;
             break;

    case 0 : (*n) -> fe = 1;
             *cambiaAltura = 0;
             break;

    case +1: /* Es necesario aplicar un tipo de rotación derecha */
             n1 = (*n) -> dcho;
             if (n1 -> fe >= 0)
             {
               if (n1 -> fe == 0)     /*altura no vuelve a disminuir */
                 *cambiaAltura = 0;
               rotacionDD(n,n1);
             }
             else
               rotacionDI(n,n1);
             break;

  }
}
```

```
void equilibrar2(Nodo** n, int* cambiaAltura)
{
  Nodo* n1;

  switch ((*n) -> fe)
  {
    case -1:   /* Se aplica un tipo de rotación izquierda */
             n1 = (*n) -> izdo;
             if (n1 -> fe <= 0)
             {
               if (n1 -> fe == 0)
                 *cambiaAltura = 0;
               rotacionII(n,n1);
             }
             else
               rotacionID(n,n1);
             break;

    case 0 : (*n) -> fe = -1;
             *cambiaAltura = 0;
             break;

    case +1: (*n) -> fe = 0;
              break;

  }
}
```

RESUMEN

Los árboles binarios de búsqueda son estructuras de datos importantes que permiten localizar una *clave de búsqueda* con un coste logarítmico. Sin embargo, el tiempo de búsqueda puede crecer hasta hacerse lineal si el árbol está degenerado; en ese sentido, la eficiencia de las operaciones dependen de lo aleatorias que sean las claves de entrada. Para evitar este problema surgen los árboles de búsqueda equilibrados, árboles AVL. La eficiencia de la *búsqueda* en los árboles equilibrados es mayor que en los árboles de búsqueda no equilibrados.

Un árbol binario equilibrado, *árbol AVL*, es un árbol de búsqueda en el cual las alturas de las ramas derecha e izquierda del nodo raíz difieren a lo sumo en uno, y los subárboles izquierdo y derecho son, a su vez, árboles equilibrados.

Las operaciones que se aplican sobre los árboles de búsqueda se aplican sobre los árboles equilibrados. Las operaciones de inserción y de eliminación tienen una primera parte en la que se siguen los mismos pasos que las mismas operaciones en los árboles de búsqueda no equilibrados. Además hay que añadir una segunda parte de reestructuración del árbol para asegurar el equilibrio de cada nodo. La solución que se aplica ante un desequilibrio de un nodo viene dada por las rotaciones de nodos. Se pueden aplicar dos tipos de rotaciones, rotación simple y rotación doble; la aplicación de una u otra depende del nodo desequilibrado y del *equilibrio* del hijo que se encuentra en la rama más alta.

Las operaciones de inserción y de eliminación en árboles binarios equilibrados son más costosas que en los no equilibrados, debido a que el algoritmo debe *retroceder* por el camino de búsqueda para actualizar el factor de equilibrio de los nodos que lo forman y al coste de la rotación cuando se viola la condición de equilibrio. Por el contrario, la operación de búsqueda es, de promedio, menos costosa que en los árboles binarios no equilibrados.

EJERCICIOS

15.1. Dibujar el árbol binario de búsqueda equilibrado que resulta de aplicar la operación de insertar con las claves: `14, 6, 24, 35, 59, 17, 21, 32, 4, 7, 15` y `22`.

15.2. Dada la secuencia de claves enteras: `100, 29, 71, 82, 48 39, 101, 22, 46, 17, 3, 20, 25, 10`. Dibujar el árbol AVL correspondiente. Eliminar claves consecutivamente hasta encontrar un nodo en el que se *viole* la condición de equilibrio cuya restauración sea con una rotación doble.

15.3. En el árbol construido en el Ejercicio 15.1, eliminar el nodo raíz. Hacerlo tantas veces como sea necesario hasta que se desequilibre un nodo y haya que aplicar una rotación simple.

15.4. Encontrar una secuencia de n claves que al ser insertadas en un árbol binario de búsqueda vacío se apliquen las cuatro rutinas de rotación: `II, ID, DD, DI`.

15.5. Dibujar el árbol equilibrado después de insertar en orden creciente 31 ($2^5 - 1$) elementos del 11 al 46.

15.6. ¿Cuál es el número mínimo de nodos de un árbol binario de búsqueda equilibrado de altura 10?

15.7. Escribir la función recursiva `buscarMin()` que devuelva la clave mínima de un árbol de búsqueda equilibrado.

16.2. *TAD ÁRBOL B* Y SU REPRESENTACIÓN

A la forma de organizar claves descrita como estructura árbol B, se le asocian una serie de operaciones básicas para poder buscar una clave, añadir claves, y recíprocamente, eliminar claves. En definitiva, un árbol B es un tipo abstracto de datos con las operaciones básicas:

Crear	Inicializa el árbol B como árbol vacío.
Buscar	Dada una clave realiza la búsqueda de la clave. Devuelve la dirección del nodo y la posición en este.
Insertar	Añade una nueva clave al árbol B. El árbol resultante sigue teniendo las características de árbol B.
Eliminar	Borra una clave del árbol B. El árbol resultante mantiene las características de árbol B.

16.2.1. Representación de un nodo de árbol B

Los datos de que consta un árbol B son los nodos o *páginas* del árbol. Una *página* guarda las claves y las direcciones de las ramas de los nodos o *páginas descendientes*. De manera natural surge la utilización de dos vectores, y un campo adicional que en todo momento contenga el número de claves de la página. La *estructura* `Pagina` agrupa la representación de una *pagina* o nodo del árbol; además, se incorpora la operación `nodoLLeno()`, devuelve *verdadero* si el número de claves es `m-1` que es el máximo de claves que puede haber en una página. La operación `nodoSemiVacio()`, devuelve *verdadero* si el número de claves es menor que `m/2`, que es el mínimo de claves que puede haber en una página.

En esta representación, el tipo de las *claves de búsqueda* es entero, para facilitar la comprensión de la estructura y de las operaciones. El orden del árbol B es `5`; éstos son los más utilizados. Los tipos de datos y los prototipos de las funciones se realizan en el archivo de cabecera `pagina.h`; y la implementación en el archivo `pagina.c`.

archivo pagina.h

```
#define m 5

typedef int tipoClave;

typedef struct pagina
{
  tipoClave claves[m];
  struct pagina* ramas[m];
  int cuenta;
} Pagina;

int nodoLLeno(Pagina* actual);
int nodoSemiVacio(Pagina* actual);
void escribeNodo(Pagina* actual);
```

archivo pagina.c

```
int nodoLLeno(Pagina* actual)
{
```

```
    return (actual->cuenta == m -1);
  }

  int nodoSemiVacio(Pagina* actual) {
    return (actual->cuenta < m/2);
  }

  void escribeNodo(Pagina* actual)
  {
    int k;
    printf("\n Nodo: ");
    for (k = 1; k <= actual->cuenta; k++)
      printf(" %d ",actual->claves[k]);
    printf("\n");
  }
```

Nota de programación

En un *árbol B de orden m*, el número de ramas de una página *no hoja* es uno más que el de sus claves. Por ello las ramas se indexan de `0` a `m-1`, y las claves de `1` a `m-1`. El subárbol de `ramas[o]` se corresponde con las claves descendientes menores que `claves[1]`.

16.2.2. Tipo abstracto árbol B

Las operaciones que constituyen la interfaz del tipo abstracto *árbol B*, `buscar()`, `insertar()` y `eliminar()` se encapsulan en el archivo de cabecera `ArbolB.h`. Además hay otras funciones auxiliares que son únicamente necesarias para el desarrollo de las operaciones del *tipo abstracto*. La operación `crearArbolB()` simplemente inicializa el árbol B a la clave que indica que está vacío:

```
void crearArbolB(Pagina** raiz) {*raiz = NULL;}
```

Archivo ArbolB.h

```
void crearArbolB(Pagina** raiz);

Pagina* buscar(Pagina* actual, tipoClave cl, int* k);
void insertar(Pagina**raiz, tipoClave cl);
void eliminar(Pagina**raiz, tipoClave cl);
    /*
      funciones auxiliares
    */
```

Nota de programación

Es necesario que en todo programa que incluya el archivo *Arbolb.h* también incluya, previamente, el archivo de cabecera `Pagina.h`; debido a que se utiliza el tipo de dato correspondiente al nodo del árbol.

16.3. PROCESO DE FORMACIÓN DE UN ÁRBOL B

Las claves que se añaden a un árbol B siempre se insertan a partir de un nodo hoja, como ocurre en los árboles binarios. Además, al estar un árbol B perfectamente equilibrado, todas la hojas de un árbol B se encuentran en el mismo nivel, lo que impone el comportamiento característico de los árboles B: *crecen «hacia arriba», crecen en la raíz*. Los pasos a seguir para añadir un nueva clave en un árbol B son los siguientes:

- Primero se realiza la búsqueda de la clave a insertar en el árbol. Se sigue el *camino de búsqueda* que determina las claves de los nodos.
- En el caso de que la clave no esté en el árbol, la búsqueda termina en un nodo *hoja*. Entonces, empieza el proceso de inserción de la nueva clave.
- De no estar lleno el nodo hoja, la inserción es posible en dicho nodo y termina la inserción.
- En caso de estar llena la *hoja*, la inserción no es posible en dicho nodo; el comportamiento característico de los árboles B se pone de manifiesto, entonces se divide el nodo en dos en el mismo nivel del árbol, excepto la clave *mediana*, que no se incluye en ninguno de los dos nodos, sino que *sube* en el árbol por el *camino de búsqueda*, para a su vez repetir el proceso de inserción en el nodo *antecedente*. Por esto un ***árbol B crece hacia arriba***; esta *ascensión* de la clave *mediana* puede ocurrir que llegue al nodo raíz, entonces éste se divide en dos nodos y la clave enviada hacia arriba se convierte en una nueva raíz. Ésta es la forma en que el árbol B crece en altura.

16.3.1. Creación de un árbol B de orden 5

A continuación se van a seguir los pasos de formación de un árbol B de orden `5` con claves de tipo entero. Al ser de orden `5`, el número máximo de claves en cada nodo es `4` y el máximo de ramas `5`.

Supóngase que las claves que se insertan son:

```
6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27
```

- Con las cuatro primeras se completa el primer nodo o página; eso sí, en orden creciente a la manera de árbol de búsqueda. La Figura 16.2*a* muestra este nodo.
- La clave siguiente, `8`, encuentra el nodo ya lleno. La clave *mediana* de las cinco claves es `6`. El nodo lleno se divide en dos, la clave mediana *«sube»* y como no hay nodo antecedente, se crea otro nodo con la clave mediana, la nueva raíz del árbol. El árbol ha crecido en altura como puede observarse en la Figura 16.2*b*.

Figura 16.2. Formación de un árbol B de orden 5.

- Las siguientes claves —`9, 12`— se insertan en el nodo rama derecha de la raíz; según el criterio de búsqueda, ambas claves son mayores que `6`. La Figura 16.3*a* muestra el árbol después de esta inserción.
- La clave `21` *«baja»* por el camino de búsqueda, rama derecha del nodo raíz. El nodo hoja que le corresponde está lleno; por consiguiente, el nodo se parte en dos y la clave mediana, `11`, asciende por el camino de búsqueda para ser insertada en el nodo antecedente, en este caso la

raíz. El proceso se repite, ahora sí hay *hueco* para que la clave `11` sea insertada. La Figura 16.3*b* muestra el árbol resultante después de la creación del nuevo nodo.

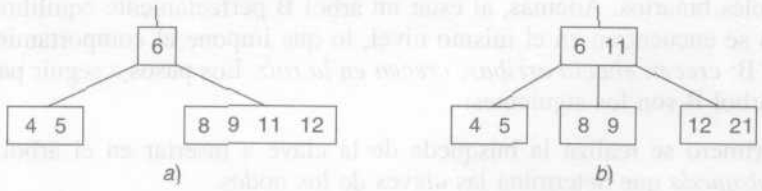


Figura 16.3. Inserción de nuevas claves en un árbol B de orden 5.

- Las siguientes claves —`14, 10, 19`— se insertan en los nodos *hoja* que les corresponde, según el *camino de búsqueda*, sin tener que dividir el nodo ya que no se encuentra lleno.
- Al insertar la siguiente clave, `28`, el *camino de búsqueda* determina que se inserte en un nodo que está lleno (nodo derecho del último nivel). De nuevo se produce el proceso de división del nodo y ascensión, por el *camino de búsqueda*, de la clave *mediana* que ahora es `19`. El antecedente, nodo raíz, tiene dos claves y por consiguiente se inserta `19` en el nodo.
- Las claves que vienen a continuación, `3, 17, 32, 15,` son insertadas en los nodos *hoja* que determina el *camino de búsqueda*. Cada uno de estos nodos tiene un número de claves menor que el máximo, `m-1`, con lo que cada clave se insertan directamente en la hoja, en la posición que le corresponde según su valor. La Figura 16.4 representa al árbol después de estas inserciones.

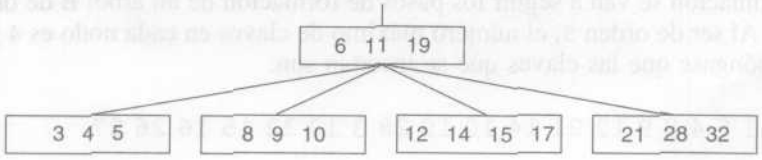


Figura 16.4. Representación de árbol B de orden 5.

- Ahora hay que añadir la clave `16`; *«baja»* por el *camino de búsqueda* (rama derecha de clave `11`). El nodo donde va a ser insertado está lleno, se produce la división del nodo y la clave mediana, `15`, *«sube»* por el *camino de búsqueda* para que sea añadida en el nodo antecedente. Como éste no está lleno, se inserta la clave `15`; como se puede observar en la Figura 16.5, el nodo raíz ha quedado completo.

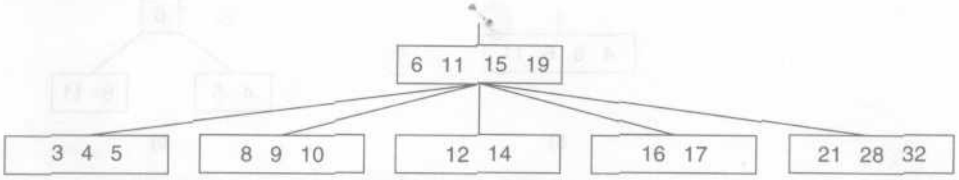


Figura 16.5. Árbol B de orden 5 después de una división de nodo.

- Al insertar la clave `26` se completa el nodo más a la derecha del último nivel. Por último, la clave `27` *«baja»* por el mismo *camino de búsqueda*, el nodo *hoja* que le corresponde está completo, por lo que se divide en dos y *«sube»*, al nodo *padre*, la clave *mediana*, `27`. Ocurre que el nodo *padre* también está completo, se repite el proceso de división en dos nodos, y *«sube»* la nueva clave *mediana*, `15`. Como el nodo dividido es el raíz, con esta clave *mediana*

se forma un nuevo nodo raíz: el árbol ha crecido en altura. El *árbol crece hacia arriba*, hacia la raíz.

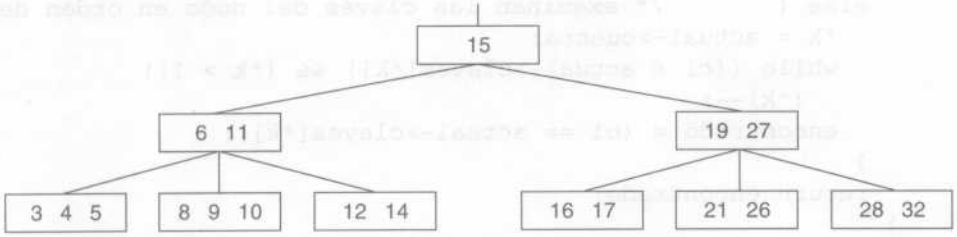


Figura 16.6. Árbol B de orden 5 con todas las claves.

A tener en cuenta

En el proceso de formación de un *árbol B* hay que destacar dos hechos relevantes. El primero, la división de un nodo prepara a la estructura para nuevas inserciones ya que deja *huecos* en los nodos implicados. En segundo lugar, la clave que *«sube»* al nodo antecedente es la clave *mediana* del nodo lleno, no tiene por qué coincidir con la clave que se está insertando; se puede afirmar que no importa el orden en que lleguen las claves en el balanceo del árbol.

16.4. BÚSQUEDA DE UNA CLAVE EN UN ÁRBOL B

Las claves de un nodo en un árbol B son *claves de búsqueda*, dividen a las ramas inmediatas, rama izquierda y derecha, a la manera de un árbol de búsqueda. Por consiguiente, el algoritmo sigue la misma estrategia que la búsqueda en un árbol binario de búsqueda, salvo que en los árboles B, cuando se está analizando una página o nodo, hay que inspeccionar las claves de que consta. La inspección da como resultado la posición de la clave, o bien la rama del árbol por donde seguir la búsqueda.

La operación es realizada por la función `buscar()` que tiene como entrada la raíz del árbol B y la clave a localizar. Devuelve la dirección (puntero) del nodo que contiene a la clave y la posición que ocupa, si encuentra la clave; en el caso de no encontrar la clave, devuelve `NULL`. La función realmente lo que hace es descender por el árbol por la ruta de búsqueda determinada por la clave y los nodos del árbol, la inspección de cada nodo se realiza en la función auxiliar `buscarNodo()`.

16.4.1. Función `buscarNodo()`

Esta función realiza la tarea específica de búsqueda de la clave en un nodo. Analiza el nodo, teniendo en cuenta que las claves están ordenadas, de tal forma que la función devuelve el resultado de la búsqueda, y en el argumento `k` la posición que ocupa, o bien la rama por la que continuar el proceso. Para ello, inspecciona las claves de la página en orden descendente, terminando cuando `claves[k] <= cl`, así se asegura que `k` termina con la posición que ocupa la clave o el índice de la rama en la que puede ser encontrada.

```
int buscarNodo(Pagina* actual, tipoClave cl, int* k)
{
  int encontrado;

  if (cl < actual->claves[1])
  {
```

```
        encontrado = 0;
        *k = 0;
      }
      else {        /* examinan las claves del nodo en orden descendente */
        *k = actual->cuenta;
        while ((cl < actual->claves[*k]) && (*k > 1))
          (*k)--;
        encontrado = (cl == actual->claves[*k]);
      }
      return encontrado;
    }
```

16.4.2. Función `buscar()`

La función `buscar()` controla el proceso de búsqueda, *baja* por las ramas del árbol hasta encontrar la clave buscada. La función `buscarNodo()` es la que determina, para cada nodo del *camino de búsqueda*, la rama por donde continuar la búsqueda o el éxito de ésta. La función se implementa recursivamente la condición para terminar de hacer llamadas recursivas es que se haya localizado la clave, o bien el nodo es una hoja y, por tanto, no hay más nodos en los que buscar. Devuelve la dirección, puntero, del nodo donde se encuentra la clave, o bien `NULL`.

```
Pagina* buscar(Pagina* actual, tipoClave cl, int* indice)
{
  if (actual == NULL)
  {
    return NULL;
  }
  else {
    int esta;
    esta = buscarNodo(actual,cl,indice);
    if (esta)
      return actual;
    else
      return buscar(actual -> ramas[*indice],cl,indice);
  }
}
```

16.5. OPERACIÓN DE INSERCIÓN EN UN ÁRBOL B

En cuanto a la organización de las claves en un árbol B, se puede considerar que es un árbol *m-ario de búsqueda*. Es por ello por lo que la operación de inserción de una nueva clave en un árbol B sigue la misma estrategia que la inserción en un árbol binario de búsqueda.

La nueva clave se inserta siempre en un nodo hoja, para lo cual *baja* por el *camino de búsqueda*, hasta alcanzar el nodo hoja. En esta primera parte del algoritmo, la llamada a la función `buscarNodo()` determina la rama por donde *bajar*.

Una vez en el nodo hoja, si no está lleno se inserta directamente en la posición que devuelve `buscarNodo()`, de tal forma que la clave queda ordenada. Este caso de inserción involucra únicamente a un nodo; en la función `meterHoja()` se realiza esta inserción.

Sólo si el nodo está lleno la inserción afecta a la estructura del árbol ya que se ha de crear un nodo nuevo y, además, para que el árbol siga cumpliendo las características de árbol B, *asciende*

la clave mediana al nodo antecedente. La función `dividirNodo()` se encarga de realizar este módulo.

Una formulación recursiva va a ser la adecuada para reflejar el proceso de propagación hacia arriba en la *división de los nodos*, debido a que al retornar la llamada recursiva *regresa por el camino de búsqueda*, se pasa por los nodos en sentido inverso a como se *bajó*.

La función `insertar()` es la interfaz de la operación, tiene como argumento la nueva clave que se va a añadir al árbol. Pasa control a `empujar()`, que es la función recursiva que se encarga de realizar el proceso de inserción. Las funciones `meterHoja()` y `dividirNodo()` se llaman desde `empujar()` según que la inserción se realice en un nodo con posiciones vacías o llenas respectivamente.

En el archivo `ArbolB.h` se añaden los prototipos de las funciones auxiliares necesarias para la operación de inserción:

```
...
void insertar(Pagina**raiz, tipoClave cl);
void empujar(Pagina* actual, tipoClave cl, int* subeArriba,
                                tipoClave* mediana, Pagina** nuevo);
void meterHoja(Pagina* actual, tipoClave cl, Pagina* rd, int k);
void dividirNodo(  Pagina* actual, tipoClave cl, Pagina* rd,
                        int k, tipoClave* mediana, Pagina** nuevo);
```

16.5.1. Función `insertar()`

Constituye la interfaz de la operación para añadir una nueva clave al árbol. Se encarga de crear un nodo raíz en el caso de la *propagación hacia arriba* del proceso de división llegue a la raíz; si ocurre así, el árbol aumenta en altura. La variable `subeArriba` detecta este hecho.

```
void insertar(Pagina**raiz, tipoClave cl)
{
  int subeArriba;
  tipoClave mediana;
  Pagina *p,*nd;

  empujar(*raiz, cl, &subeArriba, &mediana, &nd);

  if (subeArriba)
  {
    p = (Pagina*) malloc(sizeof(Pagina));
    p -> cuenta = 1;
    p -> claves[1] = mediana;
    p -> ramas[0] = *raiz;
    p -> ramas[1] = nd;
    *raiz = p;
  }
}
```

16.5.2. Función `empujar()`

Esta rutina es la más importante dado que controla la realización de las tareas más importantes en el proceso de inserción. Lo primero que hace es *«bajar»* por el camino de búsqueda hasta llegar a una

rama vacía. Cuando esto ocurre se prepara para *«subir»* por el camino y realizar la inserción; simplemente se activa el indicador, `subeArriba`, para que al devolver control las llamadas recursivas se inserte la clave. Primero se encuentra con el nodo hoja, si hay *hueco*, inserta la clave y el proceso termina. De estar lleno, llama a la función `dividirNodo()` que crea un nuevo nodo para repartir las claves; la clave mediana sigue *«subiendo»* por el camino de búsqueda.

La simulación de *«bajar»* y luego *«subir»* por el camino de búsqueda se implementa fácilmente mediante llamadas recursivas a `empujar()`, de tal forma que cuando las llamadas retornan, se está volviendo (*«subiendo»*) a los nodos por los que se pasó anteriormente, *regresa* por el *camino de búsqueda*.

La inserción de la clave mediana en el nodo antecesor, debido a la división del nodo, puede a su vez causar el desbordamiento del mismo, dando como resultado la propagación del proceso de partición, pudiendo llegar esta propagación al nodo raíz. Ésta es la única forma de que aumente la altura del árbol B.

```
void empujar(Pagina* actual, tipoClave cl, int* subeArriba,
                             tipoClave* mediana, Pagina** nuevo)
{
  int k;

  if (actual == NULL)
  {
    *subeArriba = 1;
    *mediana = cl;
    *nuevo = NULL;
  }
  else {
    int esta;
    esta = buscarNodo(actual,cl,&k);

    if (esta)
    {
      puts ("\nClave duplicada");
      *subeArriba = 0;
      return;
    }

    empujar(actual->ramas[k], cl, subeArriba, mediana, nuevo);

              /* devuelve control; vuelve por el camino de búsqueda */

    if (*subeArriba)
    {
      if (nodoLLeno(actual))
        dividirNodo(actual, *mediana, *nuevo, k, mediana, nuevo);
      else
      {
        *subeArriba = 0;
        meterHoja(actual, *mediana, *nuevo, k);
      }
    }
  }
}
```

Nota de programación

En un árbol B los elementos son claves de búsqueda, por lo que no se contempla el que haya elementos repetidos. Por ello la función *empujar* considera que es un error el que la función *buscarNodo()* encuentre la clave que se quiere insertar. En ese caso devuelve control y para que no realice ninguna acción posterior pone el indicador *subeArriba* a *0*.

16.5.3. Función `meterHoja()`

La finalidad de esta función es insertar una clave en un nodo que tiene un número de claves menor que el máximo. A esta función se le llama cuando en `empujar()` se ha determinado que hay *hueco* para añadir al nodo una nueva clave. Se le pasa como argumentos la clave, la dirección del nodo (página), la dirección de la rama con el nodo sucesor, y la posición a partir de la cual se inserta.

```
void meterHoja(Pagina* actual, tipoClave cl, Pagina* rd, int k)
{
  int i;
          /* desplaza a la derecha los elementos para hacer un hueco */
  for (i = actual->cuenta; i >= k+1; i -- )
  {
    actual -> claves[i+1] = actual -> claves[i];
    actual -> ramas[i+1] = actual -> ramas[i];
  }
  actual -> claves[k+1] = cl;
  actual -> ramas[i+1] = rd;
  actual -> cuenta++ ;
}
```

16.5.4. Función `dividirNodo()`

Esta función resuelve el problema que supone que el nodo en el que hay que insertar la clave está lleno. Virtualmente, el nodo es dividido en dos y la clave mediana enviada *hacia arriba* para una *reinserción* posterior. Para ello se crea un nuevo nodo al que se desplazan las claves mayores de la clave mediana y sus correspondientes ramas, dejando en el nodo original las claves menores. En primer lugar se encuentra la posición que ocupa la clave mediana, teniendo en cuenta la posición central del nodo y la posición donde debería insertarse la nueva clave. A continuación, se crea un nuevo nodo al que se asignan las claves y ramas del nodo origen mayores que la mediana. Por último, se inserta la nueva clave en el nodo que le corresponde y se *«extrae»* la clave mediana que siempre se encuentra en el nodo original. Para que ascienda la clave mediana se utiliza el *argumento* mediana pasando su dirección (puntero), que con la técnica de llamadas recursiva de la función `empujar()` se inserta en el nodo antecedente.

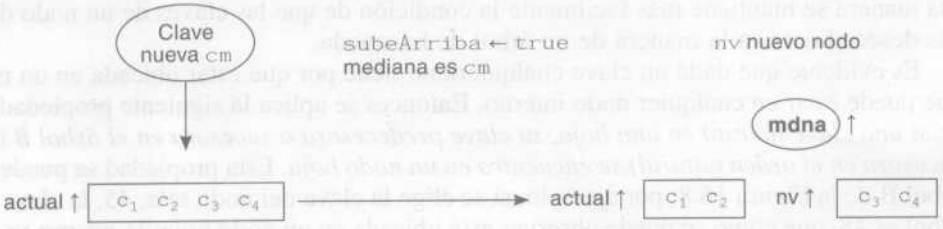


Figura 16.7. Proceso de partición de un nodo.

```
void dividirNodo(  Pagina* actual, tipoClave cl, Pagina* rd,
                        int k, tipoClave* mediana, Pagina** nuevo)
{
  int i, posMdna;

  posMdna = (k <= m/2)? m/2: m/2+1;      /* posición de clave mediana */

  (*nuevo) = (Pagina*) malloc(sizeof(Pagina));          /* nuevo nodo */
  for (i = posMdna+1; i < m; i++)
  {
    /* Es desplazada la mitad derecha al nuevo nodo, la clave
                                   mediana se queda en el nodo origen */

    (*nuevo)-> claves[i-posMdna] = actual -> claves[i];
    (*nuevo)-> ramas[i-posMdna] = actual -> ramas[i];
  }
  (*nuevo)-> cuenta = (m-1) - posMdna;    /* claves en el nuevo nodo */
  actual -> cuenta = posMdna;            /* claves en el nodo origen */

  /* Es insertada la clave y rama en el nodo que le corresponde */
  if (k <= m/2)
    meterHoja(actual,cl,rd,k);           /* inserta en nodo origen */
  else
    meterHoja(*nuevo,cl,rd,k-posMdna);

    /* se extrae clave mediana del nodo origen */

  *mediana = actual->claves[actual->cuenta];
    /* Rama0 del nuevo nodo es la rama de la mediana */
  (*nuevo) -> ramas[0] = actual->ramas[actual->cuenta];
  actual->cuenta--   ;          /* disminuye ya que se quita la mediana */
}
```

16.6. ELIMINACIÓN DE UNA CLAVE EN UN ÁRBOL B

Ésta es otra de las operaciones básicas del tipo abstracto árbol B. El objetivo de la operación es que dada una clave de un árbol, extraerla del nodo en que se encuentra con la condición de que la estructura resultante siga siendo árbol B. De igual forma que la inserción de una clave siempre se hace en una página *hoja*, la eliminación también se hace en una página (nodo) *hoja*, debido a que de esa manera se mantiene más fácilmente la condición de que las claves de un nodo dividan a las de sus descendientes a la manera de un árbol de búsqueda.

Es evidente que dada un clave cualquiera no tiene por qué estar ubicada en un nodo *hoja*, sino que puede estar en cualquier nodo interno. Entonces se aplica la siguiente propiedad de los árboles B: *si una clave no está en una hoja, su clave predecesora o sucesora en el árbol B (predecesora o sucesora en el orden natural) se encuentra en un nodo hoja*. Esta propiedad se puede observar en el árbol B de la Figura 16.8; por ejemplo, si se elige la clave del nodo raíz, 45, la clave sucesora en el árbol es 48, que como se puede observar está ubicada en un nodo *hoja*; lo mismo se puede decir de la clave predecesora, 32, ubicada en otro nodo hoja.

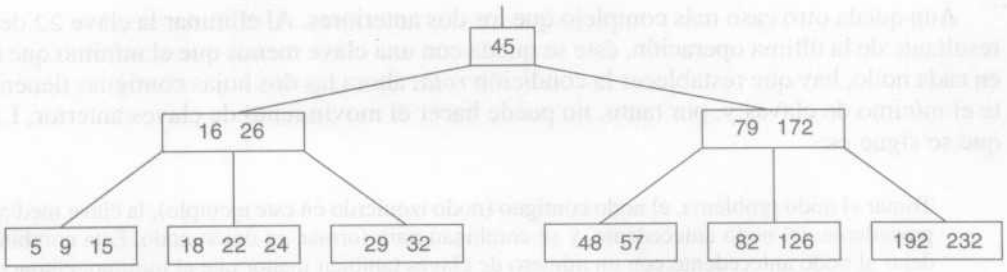


Figura 16.8. Árbol B de orden 5 para suprimir claves.

Como consecuencia de todo ello, si la clave a eliminar no está ubicada en una *hoja*, se procede a sustituirla por la clave predecesora o sucesora que sí está en una *hoja*, a continuación se suprime la clave elegida en la *hoja* donde se encuentra. De nuevo en la Figura 16.8, supóngase que se desea eliminar la clave 16, que no está en una *hoja*; se elige la clave sucesora inmediata de 16, que es 18, se pone en la posición de 16 y a continuación se suprime 18 en el nodo *hoja* donde se encuentra. Como consecuencia de esta operación el nodo *hoja* se queda con dos claves, no hay que realizar una acción posterior.

La operación que acaba de realizarse es el supuesto más fácil, es el caso en el que la *hoja* donde se va a eliminar contiene un número de claves mayor que el mínimo que puede haber en un nodo, una vez eliminada no hay que hacer una acción posterior.

En el supuesto de que la *hoja* donde se vaya a eliminar la clave tenga el número mínimo de claves, al eliminar una de ellas se queda con menos claves que el mínimo exigido por cada nodo o página. Ahora el proceso continúa para restablecer la característica de un árbol B relativo al número de claves por nodo, hay que realizar un movimiento de claves entre los nodos adyacentes y el nodo padre. Las operaciones del algoritmo son:

> Examinar los nodos inmediatamente adyacentes con el mismo nodo antecedente. Si uno de estos nodos tiene un número de claves mayor que el mínimo, puede moverse una de ellas para así restablecer el mínimo de claves en el nodo problema. Para que las claves del árbol sigan dividiendo a las claves de sus nodos descendientes a la manera de árbol de búsqueda, el movimiento de claves consiste en subir una clave al nodo antecedente, a su vez bajar de éste otra clave al nodo que se quiere restaurar. Así queda restaurado el nodo y sigue siendo un árbol B.

En el árbol resultante de la eliminación anterior ahora va a ser eliminada la clave 24. El nodo donde está ubicada contiene 2 claves, entonces se elimina la clave y como consecuencia se queda con 1 clave, Figura 16.9*a*, viola una de las características de un árbol B relativas al número de claves. Se procede a analizar los dos nodos contiguos con el mismo antecedente. Resulta que el *hermano* izquierdo tiene 3 claves, por tanto, más que el mínimo, entonces se realiza este movimiento de claves: sube la clave 15 al nodo antecedente y a su vez de éste baja la clave 18 y se inserta en el nodo a restaurar. En la Figura 16.9*b* se muestra el resultado del movimiento.

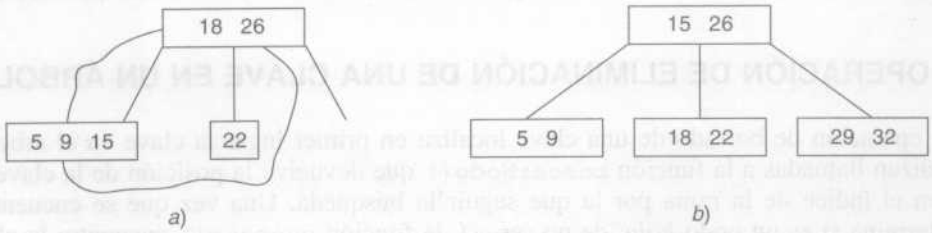


Figura 16.9. *a*) Nodo hoja después de eliminar clave 24. *b*) Nodos después del movimiento de claves.

Aún queda otro caso más complejo que los dos anteriores. Al eliminar la clave 22 del nodo hoja resultante de la última operación, éste se queda con una clave menos que el mínimo que ha de haber en cada nodo, hay que restablecer la condición *rota*; ahora las dos hojas contiguas tienen únicamente el mínimo de claves y, por tanto, no puede hacer el movimiento de claves anterior. La estrategia que se sigue es:

> Tomar el nodo problema, el nodo contiguo (nodo izquierdo en este ejemplo), la clave mediana de ambas procedente del nodo antecedente, y se combinan para formar un único nodo. Esta combinación puede dejar al nodo antecedente con un número de claves también menor que el mínimo, entonces el proceso se propaga hacia arriba, hacia la raíz. En el caso límite, la última clave se extrae del nodo raíz y por consiguiente disminuye la altura del árbol B.

Volviendo al árbol B, en el que se ha eliminado la clave 22, el nodo donde se encuentra se queda con una sola clave, y los nodos adyacentes tienen justamente el mínimo de claves. La siguiente combinación restaura el nodo como se muestra en la Figura 16.10.

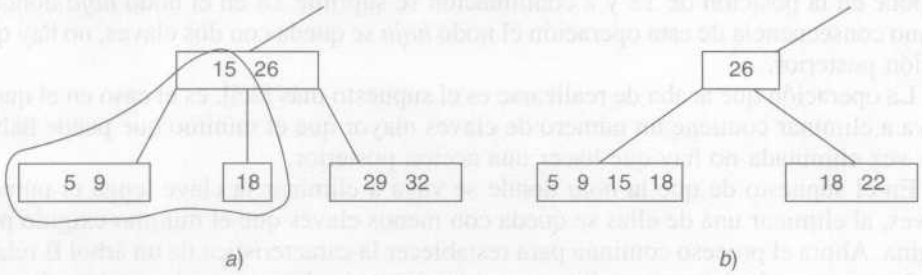


Figura 16.10. *a*) Combinación para restaurar árbol B. *b*) Árbol después de la combinación.

Este proceso ha dejado al nodo antecedente con una clave, menos que el mínimo necesario. El proceso se propaga al nivel anterior del árbol, hay que realizar una nueva combinación, como muestra la Figura 16.11*a*. El resultado es que la altura del árbol B disminuye tal como se observa en la Figura 16.11*b*.

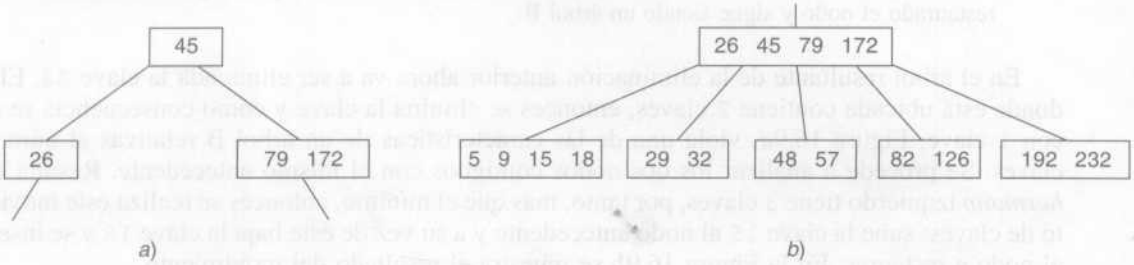


Figura 16.11. *a*) Nueva combinación para restaurar árbol B. *b*) Árbol B después de la combinación.

16.7. OPERACIÓN DE ELIMINACIÓN DE UNA CLAVE EN UN ÁRBOL B

La operación de borrado de una clave localiza en primer lugar la clave en el árbol, para ello se realizan llamadas a la función `buscarNodo()` que devuelve la posición de la clave en el nodo, o bien el índice de la rama por la que seguir la búsqueda. Una vez que se encuentra el nodo se determina si es un nodo *hoja*; de no ser así, la función `sucesor()` encuentra la clave inmediata sucesora, que se encuentra en una *hoja*, la cual reemplaza a la clave a eliminar. En el caso de que el nodo donde se encuentra la clave sea una *hoja* se llama a la función `quitar()` que extrae la clave.

Una vez que se ha eliminado la clave, se ha de inspeccionar si el nodo contiene un número de claves menor que el mínimo que exige las características de árbol B; en caso negativo, la función `restablecer()` tiene como objetivo restaurar esta característica, determina la manera de hacerlo según el número de claves de los nodos *hermano izquierdo y derecho*. Como en la restauración del mínimo de claves siempre interviene la clave del nodo *antecedente* que divide a las ramas a la manera de árbol de búsqueda, la llamada a `restablecer()` se hace desde el nodo *antecedente* (en el *regreso* por el camino de búsqueda, por tanto, al devolver control la recursividad). Las funciones `moverIzqda()` y `moverDrcha()` se encargan de esta restauración en el caso de que el hermano derecho o izquierdo, respectivamente, tengan un número de claves mayor que el mínimo; la función `combina()` realiza la restauración en caso contrario.

El proceso empieza con la llamada a la función `eliminar()`, la interfaz de la operación, que pasa control a `eliminarRegistro()`, que se encarga de controlar todo el proceso. El proceso de eliminación se puede propagar y alcanzar el nodo raíz, en el caso de que éste se quede sin claves, `eliminar()` obtiene la nueva raíz y *libera* el nodo antiguo. En este caso el árbol disminuye en altura.

De nuevo se añade al archivo `ArbolB.h` los prototipos de las funciones auxiliares necesarias para la operación de borrado:

```
...
void eliminar(Pagina**raiz, tipoClave cl);

 void eliminarRegistro(Pagina* actual, tipoClave cl, int* encontrado);
void quitar(Pagina* actual, int k);
void sucesor(Pagina* actual, int k);
void restablecer(Pagina* actual, int k);
void moverDrcha(Pagina* actual, int k);
void moverIzqda(Pagina* actual, int k);
void combina(Pagina* actual, int k);
```

16.7.1. Función `eliminar()`

Constituye la interfaz de todo el proceso de eliminación de una clave. Tiene como misión principal modificar la raíz si, debido al proceso de eliminación, ésta se queda sin claves.

```
void eliminar(Pagina**raiz, tipoClave cl)
{
  int encontrado;

  eliminarRegistro(*raiz, cl, &encontrado);
  if (encontrado)
  {
    printf("Clave %d eliminada\n",cl);
    if ((*raiz) -> cuenta == 0)
    {
        /* La raiz está vacía, se libera el nodo y se establece la
           nueva raíz */
      Pagina* p = *raiz;
      *raiz = (*raiz) -> ramas[0];
      free(p);
    }
  }
```

```
    else
      puts("La clave no se encuentra en el árbol\n");
  }
```

16.7.2. Función `eliminarRegistro()`

El control del proceso de la operación de borrado se realiza en esta función. En primer lugar, busca el nodo donde se encuentra la clave que se quiere eliminar. Una vez encontrada la clave, si el nodo es una *hoja*, llama a la función que extrae del nodo la clave. De no ser *hoja*, la clave inmediatamente mayor, que se encuentra en un nodo *hoja*, se sitúa en la misma posición que ocupa la clave a eliminar; después se suprime la clave sucesora en el nodo *hoja*. El procedimiento comprueba, un vez eliminada la clave, que el número de éstas es mayor o igual que el mínimo que impone la condición de árbol B.

La forma más fácil de implementar este proceso es la recursiva, de tal forma que al retornar la llamada recursiva se comprueba si el número de claves que han quedado es mayor o igual que el mínimo exigido en un árbol B, 2 claves en los de orden 5, de no ser así hay que mover claves al nodo para alcanzar el número necesario de claves.

```
void eliminarRegistro(Pagina* actual, tipoClave cl, int* encontrado)
{
  int k;

  if (actual != NULL)
  {
    *encontrado = buscarNodo(actual, cl, &k);
    if (*encontrado)
    {
      if (actual->ramas[k-1] == NULL)            /* es un nodo hoja */
        quitar(actual, k);
      else {
        sucesor(actual, k);
                                  /* se elimina la clave sucesora en su nodo */
        eliminarRegistro(actual->ramas[k], actual->claves[k],
                                                   encontrado);
      }
    }
    else {
      eliminarRegistro(actual->ramas[k], cl, encontrado);
    }
        /* Las llamadas recursivas devuelven control a este punto.
           Se comprueba el número de claves del nodo descendiente,
           desde el nodo actual en la ruta de búsqueda seguida. */

    if (actual->ramas[k] != NULL)
      if (actual->ramas[k]->cuenta < m/2)
        restablecer(actual,k);
  }
  else
    *encontrado = 0;
}
```

16.7.3. Función `quitar()`

La función recibe la dirección (puntero) del nodo donde se encuentra la clave a eliminar y la posición que ocupa. Elimina la clave y la rama que le corresponde moviendo una posición a la izquierda las claves y ramas posteriores a ella.

```
void quitar(Pagina* actual, int k)
{
  int j;

  for (j = k+1; j <= actual->cuenta; j++)
  {
    actual -> claves[j-1] = actual -> claves[j];
    actual -> ramas[j-1] = actual -> ramas[j];
  }
  actual->cuenta--;
}
```

16.7.4. Función `sucesor()`

La función busca la clave menor dentro del subconjunto de claves mayores a la que se desea borrar. La rama `k` contiene la dirección de los nodos con claves inmediatamente mayores, a partir de esa dirección, la rama `0` es donde se encuentran las claves menores; por lo que «*baja*» por la rama `0` hasta llegar al nodo *hoja*. Entonces, la clave menor de esa *hoja* es la inmediata sucesora, y ésta reemplaza a la clave `k` (clave que se está eliminando).

```
void sucesor(Pagina* actual, int k)
{
  Pagina* q;

  q = actual -> ramas[k];
  while (q -> ramas[0] != NULL)
    q = q -> ramas[0];
                                        /* q referencia al nodo hoja */
  actual -> claves[k] = q -> claves[1];
}
```

16.7.5. Función `restablecer()`

En esta función se realizan las acciones más complejas del proceso, restaura el nodo `actual -> ramas[K]` que se ha quedado con un número de claves menor que el mínimo que impone las características de árbol B. Por cuestión de claridad de código, la función realiza el proceso de restauración siempre con el nodo *hermano izquierdo* (claves menores), sólo utiliza al *hermano derecho* cuando no tiene *hermano izquierdo*.

```
void restablecer(Pagina* actual, int k)
{

    /* actual tiene la dirección del nodo antecedente de
    actual -> ramas[k] que tiene menos claves que el mínimo */
```

```
    if (k > 0)    /* Tiene "hermano" izquierdo */

      if (actual->ramas[k-1]->cuenta > m/2)
        moverDrcha(actual, k);

      else
        combina(actual, k);
    else        /* Sólo tiene "hermano" derecho */

      if (actual->ramas[1]->cuenta > m/2)
        moverIzqda(actual, 1);
      else
        combina(actual, 1);
  }
```

16.7.6. Funciones `moverDrcha()` y `moverIzqda()`

En estas dos funciones se mueve una clave del nodo *antecedente* al nodo que se está restaurando. A su vez, en `moverDrcha()` *asciende* la clave mayor del *hermano izquierdo* al nodo antecedente; o bien, en `moverIzqda()` *asciende* la clave menor del *hermano derecho*.

Es importante tener en cuenta que el primer argumento de ambas funciones es un puntero al nodo antecedente del que se ha quedado con menos claves que el mínimo.

```
void moverDrcha(Pagina* actual, int k)
{
  int j;
  Pagina* nodoProblema = actual->ramas[k];
  Pagina* nodoIzqdo = actual->ramas[k-1];

  for (j = nodoProblema->cuenta; j >= 1; j--)        /* deja "hueco" */
  {
    nodoProblema -> claves[j+1] = nodoProblema -> claves[j];
    nodoProblema -> ramas[j+1] = nodoProblema -> ramas[j];
  }
  nodoProblema -> cuenta++;
  nodoProblema -> ramas[1] = nodoProblema -> ramas[0];
    /* "baja" la clave del nodo padre */
  nodoProblema -> claves[1]= actual -> claves[k];
    /* "sube" la clave desde el hermano izquierdo al nodo
       padre, para reemplazar la que antes bajó */
  actual->claves[k] = nodoIzqdo->claves[nodoIzqdo->cuenta];
  nodoProblema->ramas[0] = nodoIzqdo->ramas[nodoIzqdo->cuenta];
  nodoIzqdo -> cuenta-- ;
}
```

MoverIzquierda

```
void moverIzqda(Pagina* actual, int k)
{
  int j;
```

```
    Pagina* nodoProblema = actual->ramas[k-1];
    Pagina* nodoDrcho = actual->ramas[k];

    nodoProblema -> cuenta++;
    nodoProblema->claves[nodoProblema->cuenta] = actual->claves[k];
    nodoProblema->ramas[nodoProblema->cuenta] = nodoDrcho->ramas[0];

      /* "sube" la clave desde el hermano derecho al nodo
         padre, para reemplazar la que antes bajó */

    actual->claves[k] = nodoDrcho -> claves[1];
    nodoDrcho -> ramas[1] = nodoDrcho -> ramas[0];
    nodoDrcho -> cuenta-- ;

    for (j = 1; j <= nodoDrcho->cuenta; j++)
    {
      nodoDrcho -> claves[j] = nodoDrcho -> claves[j+1];
      nodoDrcho -> ramas[j] = nodoDrcho -> ramas[j+1];
    }
  }
```

16.7.7. Función `combina()`

Esta función es, además de `eliminar()`, la única en que se reduce el número de nodos del árbol; *combina* el nodo que está en la rama `k` con el que está en la rama `k-1` y la clave *mediana* de ambos, que se encuentra en el nodo *antecedente*, para formar un único nodo. Una vez realizada la unión ordenada de claves en un solo nodo, libera el nodo obsoleto.

```
void combina(Pagina* actual, int k)
{
  int j;
  Pagina *nodoIzqdo, *q;

  q = actual -> ramas[k];
  nodoIzqdo = actual -> ramas[k-1];
                        /* "baja" clave mediana desde el nodo padre */
  nodoIzqdo->cuenta++;
  nodoIzqdo->claves[nodoIzqdo->cuenta] = actual->claves[k];
  nodoIzqdo->ramas[nodoIzqdo->cuenta]= q->ramas[0];
                        /* "mueve" claves del nodo derecho */
  for (j = 1; j <= q->cuenta; j++)
  {
    nodoIzqdo->cuenta++;
    nodoIzqdo->claves[nodoIzqdo->cuenta] = q->claves[j];
    nodoIzqdo->ramas[nodoIzqdo->cuenta] = q->ramas[j];
  }
      /* Se ajustan claves y ramas del nodo padre debido a que
      antes descendió la clave k */
  for (j = k; j <= actual->cuenta-1; j++)
  {
    actual -> claves[j] = actual -> claves[j+1];
```

```
      actual -> ramas[j] = actual -> ramas[j+1];
    }
    actual->cuenta-- ;

    free(q);
  }
```

16.8. LISTADO DE LAS CLAVES DE UN ÁRBOL B

Recorrer una estructura de datos implica *visitar* cada uno de los elementos de que consta la estructura, así ocurre con los vectores, con las listas lineales y circulares. En el recorrido de árboles se distinguen distintos tipos, recorrido en profundidad y recorrido en anchura. El recorrido en profundidad también se puede hacer en *preorden*, *inorden* y *postorden*; lo mismo se puede hacer en los árboles B, con la particularidad de que en el nodo de un árbol B hay, normalmente, más de una clave y más de dos subárboles.

Un recorrido que resulta interesante en un árbol B es el que *visita* las claves del árbol en orden creciente. Como cada clave del nodo de un árbol B divide a las de sus ramas adyacentes a la manera de un árbol de búsqueda, se aplica el recorrido en *inorden* de un árbol binario, con una pequeña variación, para *visitar* en orden creciente las claves de un árbol B.

Al archivo de cabecera `ArbolB.h` se añade el prototipo correspondiente de la función `void listarCreciente(Pagina* actual)`, que muestra las claves del árbol en orden creciente.

```
void listarCreciente(Pagina* actual)
{
  if (actual != NULL)
  {
    int j;
    listarCreciente(actual -> ramas[0]);

    for (j = 1; j <= actual -> cuenta; j++)
    {
      printf("%d \t",actual -> claves[j]);
      listarCreciente(actual -> ramas[j]);
    }
  }
}
```

Nota de programación

El recorrido en *inorden* del árbol B se puede implementar iterativamente. Para ello se utiliza el *TAD Pila* en la que se guardan punteros al nodo del árbol (`Pagina*`).

Ejercicio 16.1

Escribir una aplicación para gestionar un árbol B de orden 5, utilizando las operaciones del *TAD* ya implementadas, en la cual se realicen estas acciones:

- Dar de alta claves.
- Buscar una clave dada.
- Eliminar una clave dada.
- Listar las claves en orden creciente.

El programa crea un `ArbolB` con claves de tipo entero. Las operaciones se presentan en forma de menú; la opción *«dar de alta claves»* genera *n* claves aleatoriamente, en el rango de `1` a `9999`, y se insertan en el árbol llamando a la función `insertar()`. Las acciones de *buscar, eliminar y listar* se corresponden con las operaciones implementadas con el mismo nombre.

La aplicación consta de los archivos de cabecera `Pagina.h` y `ArbolB.h`. En el archivo `Arbol.c` se encuentra las codificaciones de todas las funciones relativas a las operaciones del árbol. Estos archivos no se escriben ya que el lector los puede encontrar en los diversos apartados del capítulo. Ahora se escribe la codificación de las funciones que muestran las opciones disponibles (*menu*), elegir una opción, obtener una clave aleatoria y la función `main()`.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "Pagina.h"
#include "ArbolB.h"

#define CLAVEMAX 9999

void escribeMenu();
int opcion();
void annadeNclaves(Pagina** arbol, int n);
void inicAleat();
int nuevaClave();

void main ()
{
  Pagina* arbol;
  int accion;

  crearArbolB(&arbol);
  inicAleat();

  do {
    tipoClave x;
    int posicion, n;
    Pagina *p;
    escribeMenu();
    accion = opcion();

    if (accion == 1)
    {
      printf("Numero de claves: ");
      scanf("%d",&n);
      annadeNclaves(&arbol,n);
    }
    else if(accion == 2)
    {
      printf("Clave a eliminar: ");
      scanf("%d",&x);
      eliminar(&arbol,x);
    }
    else if(accion == 3)
```

```
      {
        printf("Clave a buscar: ");
        scanf("%d",&x);
        p = buscar(arbol,x,&posicion);
        if (p != NULL)
        {
          printf("*Clave encontrada dentro del nodo: ");
          escribeNodo(p);
          printf("\t en posición %d \n", posicion);
        }
        else
          puts("Clave no se encuentra en el árbol");
      }
      else if(accion == 4)
      {
        puts("Listado de las claves en orden creciente");
        listarCreciente(arbol);
      }

    } while (accion != 5);

}

void escribeMenu()
{
  puts("  \n 1. Insertar n claves");
  puts("\n\n 2. Eliminar una clave");
  puts("\n\n 3. Buscar una clave");
  puts("\n\n 4. Listar las claves");
  puts("\n\n 5. Salir");
}

int opcion()
{
  int k;

  do {
    printf("Opción 1..5: ");
    scanf("%d", &k);
  } while (k<1 || k>5);
  return k;
}

void annadeNclaves(Pagina** arbol, int n)
{
  int k;

  printf("Claves insertadas: ");

  for (k = 1; k <= n; k++)
  {
    tipoClave cl;
```

```
        cl = nuevaClave();
        insertar(arbol, cl);
        printf("%d ", cl);
    }
}

void inicAleat()
{

    srand( (unsigned)time( NULL ) );
}

int nuevaClave()
{
    return rand() % CLAVEMAX +1 ;
}
```

RESUMEN

Un árbol B es una estructura de datos que se puede utilizar para implementar una tabla que reside en memoria externa, tal como un disco. Los *árboles B* tienen una serie de características que permiten que la búsqueda de una clave sea muy eficiente, por ello, si las claves se encuentran en un disco, el número de accesos al disco disminuye notablemente. Hay que tener en cuenta que el mayor tiempo de proceso, en aplicaciones que acceden a archivos que están es disco, es el de las operaciones de *entrada/salida*. Reducir el número de pasos para buscar una clave, o para insertar, supone reducir el número de operaciones de entrada y salida y por consiguiente el tiempo de proceso.

El número de claves de un nodo en un árbol B *de orden m*, que no sea la raíz, es como mínimo `m/2` y como máximo `m-1`. Además, las claves de un nodo divide a las claves de sus ramas a la manera de árbol de búsqueda. El número de ramas descendientes de un nodo es una más que el número de claves, excepto los nodos que son *hoja*.

Un árbol B siempre está perfectamente equilibrado. La inserción de una nueva clave se realiza en un nodo *hoja*; si la *hoja* está llena entonces *asciende* la clave *mediana* al nodo *padre*; si éste también está lleno sigue *ascendiendo*, pudiendo llegar al nodo raíz, si también se encuentra lleno se crea un nuevo nodo con la clave *mediana*. Se dice que los árboles B crecen hacia arriba, hacia la raíz. Los árboles B más utilizados son los de orden 5.

EJERCICIOS

16.1. 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.

16.2. En el árbol R del problema 1, elimine la clave 91 y dibuje el árbol resultante. Vuelva a eliminar ahora la clave 48. Dibujar el árbol resultante. ¿Ha habido reducción en el número de nodos?

16.3. La operación de inserción de una clave en un nodo lleno se ha realizado partiendo el nodo y elevando la clave mediana. Analizar si es posible esta otra estrategia: buscar el número de claves del hermano izquierdo y derecho, si alguno no está lleno mover claves para realizar la inserción.

16.4. En un árbol B de orden 5 se insertan las claves de manera secuencial: 1, 2, 3, ... *n*. ¿Qué claves dan origen a la división de un nodo? ¿Qué claves hacen que la altura del árbol crezca?

16.5. En el árbol B del Ejercicio 16.4 las claves son eliminadas en el mismo orden en que fueron insertadas. ¿Qué claves hacen que los nodos se queden con un número de claves menor que 2 y den lugar a la unión de dos nodos? ¿Qué claves hacen que la altura del árbol disminuya?

16.6. La búsqueda de una clave en un árbol B se ha implementado siguiendo una estrategia recursiva, implementar de nuevo la operación con una estrategia iterativa.

16.7. Se supone que la unidad de disco contiene un número de pistas, y a su vez cada pista está dividida en un número de bloques de tamaño *m*. En cuanto a tiempo de proceso, las operaciones que realiza un programa de acceso a los archivos ubicados en disco prevalecen frente a las operaciones en memoria; el tiempo de acceso a disco es del orden de decenas de milisegundo (cada vez se mejora este tiempo). Con el fin de minimizar el tiempo de acceso al disco, el orden de árbol B debe ser tal que permita que una hoja de tamaño máximo quepa en un bloque del disco.

Determinar el número máximo de accesos al disco necesario para encontrar un elemento en un árbol B de *n* elementos.

16.8. Se estima que un archivo va a tener 1 millón de registros, cada registro ocupa 50 bytes de memoria. En el supuesto de que cada bloque tenga una capacidad de 1000 bytes y que el puntero al bloque ocupa 4 bytes, diseñar una organización con árboles B para este archivo.

16.9. Un árbol B* se considera un tipo de árbol B con la característica adicional de que cada nodo está, al menos, lleno en las dos terceras partes, en lugar de la mitad como ocurre a un árbol B, excepto quizás el nodo raíz . La inserción de una nueva clave en el árbol B* sigue los mismos pasos que la inserción en un árbol B, salvo que si el nodo donde se ha de insertar está lleno, se mueven claves entre el nodo padre, el hermano (con un número de claves menor que el máximo), y el nodo donde se inserta para dejar hueco y realizar la operación. De esta forma se pospone la división del nodo hasta que los hermanos estén completos, entonces éstos pueden dividirse en tres nodos, cada uno estará con una ocupación de las dos terceras partes.

Codificar los cambios que necesitan las funciones que implementan la operación de inserción para aplicarla a un árbol B*.

16.10. 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 que construya con esas claves.

16.11. Especificar los cambios que es preciso introducir en las funciones que implementan la operación de eliminar una clave para aplicar la operación a un árbol B*.

16.12. 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 nodo *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.

PROBLEMAS

16.1. Escribir la codificación de una nueva función de prototipo: `void listadoEnRanno(Pagina* arbol, tipoClave c1, tipoClave c2)` que muestre las claves de un árbol B mayores que c_1 y menores que c_2.

16.2. Escribir la codificación de la función `listadoDecreciente()`, que recorra y muestre las claves de un árbol B en orden decreciente.

16.3. Cada uno de los centros de enseñanza del Estado consta de una biblioteca escolar. Cada centro de enseñanza está asociado con número de orden (valor entero), los centros de cada provincia tienen números consecutivos, en el rango de las unidades de 1000. Así por ejemplo, a Madrid le corresponde del 1 al 1000, a Toledo del 1001 al 2000...

Escribir un programa que permita gestionar la información indicada, formando una estructura en memoria de árbol B con un máximo de 4 claves por página. La clave de búsqueda del árbol B es el número de orden del centro, además tiene asociado el nombre del centro.

El programa debe permitir añadir centros, eliminar, buscar la existencia de un centro por la clave y listar los centros existentes.

16.4. En el Problema 16.3, cuando se termina la ejecución se pierde toda la información. Modificar el programa de modo 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 ello, al terminar la ejecución, debe escribirse de nuevo el árbol en el archivo.

16.5. Se quiere dar más contenido a la información tratada en el Problema 16.3. 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.

16.6. A la estructura ya creada del Problema 16.3, añadir las operaciones que responden a estos requerimientos:

- Dada una provincia cuyo rango de centros es conocido, por ejemplo de 3001 a 3780, eliminar en todos sus centros escolares los libros que estén repetidos en cada centro, y que informe del total de libros liberados.
- Dado un centro *n*, en su biblioteca se desea que de ciertos libros haya *m* ejemplares (por ejemplo 5).

16.7. Implemente el TAD árbol B en memoria externa, en disco. Al hacerlo en memoria externa no se utilizan variables dinámicas, los enlaces no pueden ser punteros sino números de registro en los que se almacena el nodo o página descendiente. Para crear un nuevo nodo hay que buscar un «hueco» en el archivo, representado por el número de registro, y escribir el nodo. Cada vez que se modifica un nodo, por una inserción de una clave o un borrado hay que escribir el registro.

16.8. En un archivo de texto se tienen los artículos de las tiendas «Deportes Bravo» junto con sus existencias y precio por unidad. El formato de cada línea es:

código artículo	artículo	existencias	precio (euros)
11	*zapatillas-EF1*	*45*	*110*

...

Los clientes realizan pedidos por internet y se almacenan en otro archivo de texto con el formato:

código cliente	artículo	código artículo	cantidad
RT1109	*zapatillas-EF1*	*11*	*1*

...

Escribir una aplicación para:

a) Crear un árbol B de orden 5 con el archivo de los artículos de la tienda. Se debe tomar como clave de búsqueda el código-artículo (tipo entero).

b) Procesar el archivo de pedidos, obteniendo por la pantalla si se puede atender el pedido del cliente y en dicho caso el coste total. Al atender los pedidos las existencias de los artículos disminuyen, si el número de éstas llega a ser cero se tendrá que dar de baja el artículo del árbol, y se emitirá un mensaje para que se reponga el artículo. Si no hay existencias suficientes para atender el pedido del cliente se indicará con un mensaje.

c) Una vez finalizado el proceso anterior, se obtendrá en pantalla todos los artículos cuyas existencias estén comprendidas entre 1 y 5.

17

GRAFOS I: REPRESENTACIÓN Y OPERACIONES

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Distinguir entre relaciones jerárquicas y otras relaciones.
- Definir un grafo e identificar sus componentes.
- Conocer estructuras de datos para representar un grafo.
- Conocer las operaciones básicas que se aplican sobre grafos.
- Encontrar los caminos que puede haber entre dos nodos de un grafo.
- Realizar en C la representación de los grafos y las operaciones básicas.

CONTENIDO

17.1. Conceptos y definiciones.
17.2. Representación de los grafos.
17.3. *TAD grafo.*
17.4. Recorrido de un grafo.
17.5. Conexiones en un grafo.
17.6. Matriz de caminos: cierre transitorio.
17.7. Puntos de articulación de un grafo.

RESUMEN

EJERCICIOS

PROBLEMAS

CONCEPTOS CLAVE

- Relación jerárquica.
- Vértice y arco.
- Factor de peso.
- Camino.
- Matriz de adyacencia.
- Lista de adyacencia.
- Conexión y componente conexa.

INTRODUCCIÓN

Este capítulo introduce al lector en 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 de la computadora. También se estudian operaciones importantes y algoritmos de grafos que son significativos en informática.

17.1. CONCEPTOS Y DEFINICIONES

Un grafo `G` agrupa *entes* físicos o conceptuales y las relaciones entre ellos. Por tanto, un grafo está formado por un conjunto de vértices o nodos `V`, que representan a los *entes*, y un conjunto de arcos `A`, que representan las *relaciones* entre vértices. Se representa con el par `G = (V, A)`. La Figura 17.1 muestra un grafo formado por los vértices `V = {1,4,5,7,9}` y el conjunto de arcos `A = {(1,4), (4,1), (5,1), (1, 5), (7,9), (9,7), (7,5), (5,7), (4,9), (9,4)}`.

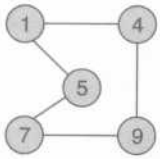


Figura 17.1. Grafo no dirigido.

Un arco o arista representa una *relación* entre dos nodos. Esta relación, al estar formada por un par de nodos, se representa por `(u, v)`, siendo `u`, `v` el par de nodos. El grafo es *no dirigido* si los arcos están formados por pares de nodos no ordenados, no apuntados; se representa con un segmento uniendo los nodos, `u -> v`. El grafo de la Figura 17.1 es no dirigido.

Un *grafo* es *dirigido*, también denominado *digrafo*, si los pares de nodos que forman los arcos son ordenados; se representan con una flecha que indica la dirección de la relación, `u -> v`. El grafo de la Figura 17.2 consta de los vértices `V = {C,D,E,F,H}` y de los arcos `A = {(C,D,), (D,F), (E,H), (H,E), (E,C)}`, que forman el grafo dirigido `G = {V,A}`.

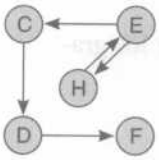


Figura 17.2. Grafo dirigido.

Dado el arco `(u,v)` de un grafo, se dice que los vértices `u` y `v` son adyacentes. Si el grafo es dirigido, el vértice `u` es adyacente a `v`, y `v` es adyacente a `u`.

En los modelos realizados con grafos a veces una relación entre dos nodos tiene asociada una *magnitud* denominada factor de peso, en cuyo caso se dice que es un *grafo valorado*. Por ejemplo, los pueblos que forman una comarca junto a la relación entre un par de pueblos de *estar unidos por un camino*: esta relación tiene asociada el factor de peso, que es la distancia en kilómetros. La Figura 17.3 muestra un grafo valorado en el que cada arco tiene asociado un peso que es la longitud entre dos nodos.

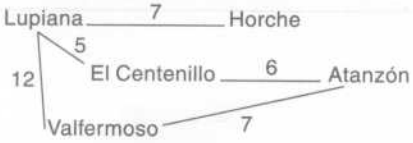


Figura 17.3. Grafo no dirigido valorado.

Definición

Un grafo permite modelar relaciones arbitrarias entre objetos. Un grafo `G = (V,A)` es un par formado por un conjunto de vértices o nodos, `V`, y un conjunto de arcos o aristas, `A`. Cada arco es el par `(u,w)`, siendo `u`, `w` dos vértices relacionados.

17.1.1. Grado de entrada, grado de salida de un nodo

El **grado** es una cualidad que se refiere a los nodos de un grafo. En un grafo no dirigido el grado de un nodo `v`, `grado(v)`, es el número de arcos que contiene a `v`. En un grafo dirigido se distingue entre grado de entrada y grado de salida; *grado de entrada* de un nodo `v`, `gradent(v)`, es el número de arcos que llegan a `v`; *grado de salida* de `v`, `gradsal(v)`, es el número de arcos que salen de `v`.

Así, en el grafo no dirigido de la Figura 17.3, `grado(Lupiana) = 3`. En el grafo dirigido de la Figura 17.2, `gradent(D) = 1` y el `gradsal(D)=1`.

17.1.2. Camino

Un camino `P` de longitud `n` desde el vértice v_0 a v_n en un grafo `G` es la secuencia de `n+1` vértices $v_0, v_1, v_2, \ldots, v_n$ *tal que* $(v_i, v_{i+1}) \in A(arcos)$ *para* $0 \le i \le n$. *Matemáticamente, el camino* se representa por $P = (v_0, v_1, v_2, \ldots, v_n)$.

En la Figura 17.4 se pueden encontrar más de un camino; por ejemplo, $P_1 = (4,6,9,7)$ es un camino de longitud 3. Otro de los caminos es $P_2 = (10, 11)$, que tiene de longitud 1.

Por tanto, se puede afirmar que la longitud del camino es el número de arcos que lo forman.

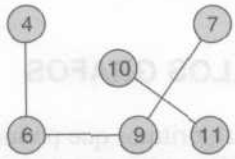


Figura 17.4. Grafo no dirigido de 5 vértices.

Definición

La longitud de un camino es el número de arcos en el camino. En un grafo valorado, la longitud del camino con pesos es la suma de los pesos de los arcos en el camino.

En el grafo valorado de la Figura 17.3, el camino `(Lupiana, Valfermoso, Atanzón)` tiene de longitud `12 + 7 = 19`.

En algunos grafos existen arcos desde un vértice a sí mismo, `(v,v)`; entonces el camino `v -> v` es un bucle. Normalmente en los grafos no hay nodos relacionados con sí mismos, no es frecuente encontrarse grafos que contengan bucles.

Un camino $P = (v_0, v_1, v_2, \ldots, v_n)$ es simple si todos los nodos que forman el camino son distintos, pudiendo ser iguales v_0, v_n, es decir, los extremos del camino. En el grafo de la Figura 17.4 los caminos P_1 y P_2 son caminos simples.

En un grafo dirigido, un ciclo es un camino simple cerrado. Por tanto, un ciclo empieza y termina en el mismo nodo, $v_0 = v_n$; debe tener más de un arco. A un grafo dirigido sin ciclos (acíclico) se

hace referencia a él con frecuencia mediante la abreviatura GDA (Grafo Dirigido Acíclico o sin ciclos). La Figura 17.5 muestra un grafo dirigido en el que los vértices (A, E, B, F, A) forman un ciclo de longitud 4. En general, un ciclo de longitud *k* se denomina *k-ciclo*.

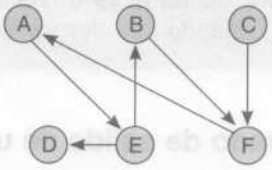


Figura 17.5. Grafo dirigido con ciclos.

Un grafo no dirigido es *conexo* si existe un camino entre cualquier par de nodos que forman el grafo. En el caso de un grafo dirigido con esta propiedad se dice que es *fuertemente* conexo. Además, un *grafo completo* es aquel que tiene un arco para cualquier par de vértices.

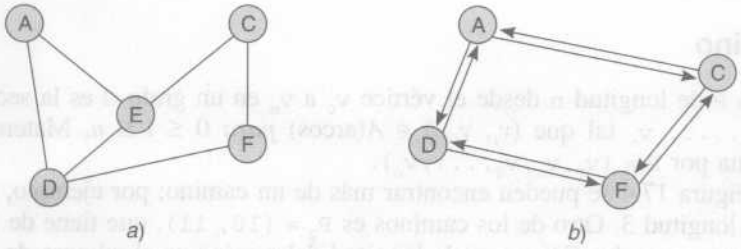


Figura 17.6. *a*) Grafo conexo. *b*) Grafo fuertemente conexo.

17.2. REPRESENTACIÓN DE LOS GRAFOS

Para trabajar con grafos y aplicar algoritmos que permiten encontrar propiedades entre los nodos hay que pensar en su representación. ¿Cómo representar un grafo en memoria interna, qué tipos o estructuras de datos se deben utilizar para considerar los nodos y los arcos?

Una primera simplificación que se hace es considerar a los vértices o nodos como números consecutivos, empezando por el vértice 0. Hay que tener en cuenta que se ha de representar un número (finito) de vértices y de arcos que unen dos vértices. Se puede elegir una representación secuencial, mediante un array bidimensional, conocida como *matriz de adyacencia*; o bien, una representación dinámica, mediante una estructura multienlazada, denominada *lista de adyacencia*. La elección de una representación u otra depende del tipo de grafo y de las operaciones que se vayan a realizar sobre los vértices y arcos. Para un grafo denso lo mejor es utilizar una matiz de adyacencia. Para un grafo disperso se suelen utilizar listas de adyacencia que se ajustan al número de arcos.

17.2.1. Matriz de adyacencia

La característica más importante de un grafo, que distingue a un grafo de otro, es el conjunto de pares de vértices que están *relacionados*, o que son adyacentes. Por ello, la forma más sencilla de representar un grafo es mediante una matriz, de tantas filas/columnas como nodos, que permite modelar fácilmente esa cualidad.

Sea G = (V, A) un grafo de n nodos, siendo $V = \{v_0, v_1, \ldots, v_{n-1}\}$ el conjunto de nodos, y $A = \{(v_i, v_j)\}$ el conjunto de arcos. Los nodos se pueden representar por números consecu-

tivos de 0 a n-1. La representación de los arcos se hace con una matriz A de n × n elementos, denominada matriz de adyacencia, tal que todo elemento a_{ij} puede tomar los valores:

$$a_{ij} \begin{cases} 1 & \textit{si hay un arco } (v_i,\ v_j) \\ 0 & \textit{si no hay arco } (v_i,\ v_j) \end{cases}$$

La Figura 17.7 representa un grafo dirigido, suponiendo que el orden de los vértices es el de la secuencia {D,F,K,L,R}, entonces la matriz de adyacencia:

$$A = \begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix}$$

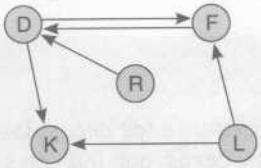


Figura 17.7. Grafo dirigido con los vértices {D,F,K,L,R}.

El grafo de la Figura 17.8 es no dirigido, está formado por 5 vértices. La matriz de adyacencia es:

$$A = \begin{vmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{vmatrix}$$

Como se puede observar, es una matriz simétrica. En los grafos no dirigidos, la matriz de adyacencia siempre es simétrica ya que las relaciones entre vértices no son ordenadas: si v_i *está relacionado* con v_j, entonces v_j *está relacionado* con v_i.

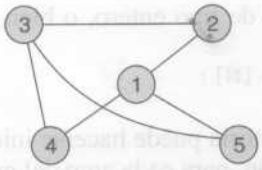


Figura 17.8. Grafo no dirigido con 5 vértices.

Los grafos que modelan problemas en los que un arco tiene asociado una magnitud, un *factor de peso*, también se representan mediante una matriz de tantas filas/columnas como nodos. Ahora un elemento cualquiera, a_{ij} representa el coste o *factor de peso* del arco $(v_i,\ v_j)$. Un arco que no existe se puede representar con un valor imposible, por ejemplo *infinito*; también se puede representar con el valor 0, dependiendo de que para el grafo el 0 no pueda ser un factor de peso significativo de un arco. A esta matriz también se la denomina *matriz valorada*.

El grafo valorado de la Figura 17.9 se corresponde con un *grafo dirigido con factor de peso*. Si suponemos los vértices se numeran en el orden de `V = {Alicante, Barcelona, Cartagena, Murcia, Reus}`, la matriz de pesos `P` en la que se representa con 0 la no existencia de arco:

$$P = \begin{vmatrix} 0 & 7 & 3 & 5 & 0 \\ 0 & 0 & 3 & 2 & 3 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{vmatrix}$$

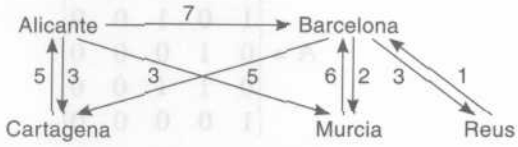


Figura 17.9. Grafo dirigido con factor de peso.

A considerar

La matriz de adyacencia representa los arcos, las relaciones entre un par de nodos de un grafo. Es una matriz de unos y ceros, que indican si dos vértices son adyacentes o no. En un grafo valorado, cada elemento representa el peso de la arista, y por ello se la denomina *matriz de pesos*.

Matriz de adyacencia en C

Al ser una representación estática debe estar previsto el máximo de nodos al que puede llegar el grafo. Los nodos o vértices en los problemas reales tienen nombres en vez de números consecutivos. Es necesario tener una tabla de correspondencia entre el nombre del vértice y el número; lo más sencillo es un vector de cadenas, de tal forma que el índice del vector es el número de vértice y el contenido es el nombre del vértice. Si por ejemplo, el máximo número de vértices o nodos es 99 y el nombre un máximo de 30 caracteres:

```
#define N 99
char vertices[N][31];
```

La matriz de adyacencia es de tipo entero, o bien de tipo real si el grafo es valorado:

```
int matAdyacencia [N][N];
```

La entrada del grafo en memoria puede hacerse inicializando cada elemento de la matriz a cero (no adyacencia) y, a continuación, para cada arco del grafo asignar 1 a la posición que le corresponde en la matriz. En cuanto al tiempo de esta operación, es inmediato que es cuadrático, $O(n^2)$, siendo `n` el número de nodos.

17.2.2. Listas de adyacencia

La representación de un grafo con matriz de adyacencia puede resultar poco eficiente cuando el grafo es poco denso, cuando es disperso, es decir, tiene pocos arcos, y por tanto la matriz de adyacencia tiene muchos ceros. Para grafos dispersos la matriz de adyacencia ocupa el mismo

espacio que si el grafo tuviera muchos arcos. Cuando esto ocurre se elige la representación del grafo mediante listas enlazadas, denominadas *listas de adyacencia*.

Las listas de adyacencia son una estructura multienlazada formada por una lista directorio; cada nodo representa un vértice del grafo, del que además emerge una lista enlazada con todos sus vértices adyacentes. Esto es, cada lista representa a los arcos con vértice origen el del nodo de la lista directorio, por eso se llama lista de adyacencia.

La Figura 17.10 representa a un grafo dirigido, cuya representación mediante listas de adyacencia se encuentra en la Figura 17.11. Si por ejemplo se analiza el vértice 5, se observa que es adyacente a los vértices 1, 2 y 4; por lo que su lista de adyacencia consta de tres nodos, cada uno con el vértice destino que forma el arco. El caso contrario es el vértice 4, no hay arco que lo tenga como vértice origen, como consecuencia la lista de adyacencia está vacía.

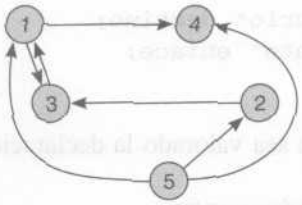


Figura 17.10. Grafo dirigido.

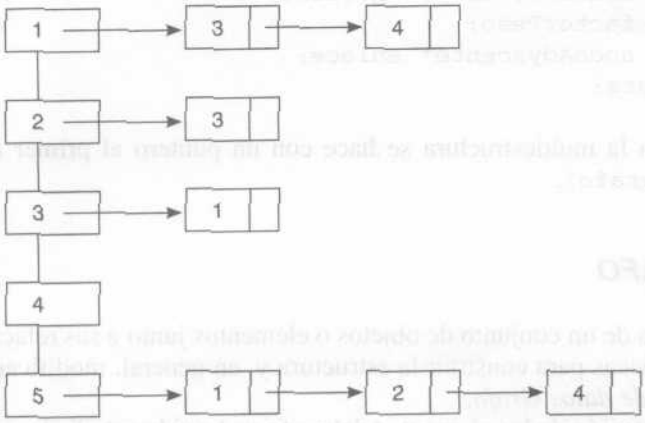


Figura 17.11. Listas de adyacencia del grafo de la Figura 17.10.

Listas de adyacencia en C

Cada nodo de la lista directorio debe guardar el vértice que representa, el puntero a lista de adyacencia y, además, la dirección del nodo siguiente de la lista directorio. Por esta razón cada nodo puede ser visto:

Vértice	*lista adyacencia*	enlace

A su vez, la lista de adyacencia de un vértice u consta de tantos nodos como arcos le tienen por origen. Cada uno de estos nodos contiene una *referencia* al nodo del grafo adyacente desde u. Entonces se va a guardar la dirección del vértice, en la lista directorio, con el que forma un arco. Además, si el grafo es valorado se hace una pequeña modificación consistente en añadir el campo, de tipo numérico, para el factor de peso. Según esto, el nodo puede ser visto:

dirVértice	*<peso>*	enlace

La estructura multienlazada se maneja con las operaciones típicas de listas enlazadas. Las declaraciones de los nodos de cada lista:

```
typedef struct nodoDirectorio
{
  char vertice[31];
  struct nodoAdyacente* lista;
  struct nodoDirectorio* enlace;
} NodoGrafo;

typedef struct nodoAdyacente
{
  struct nodoDirectorio* destino;
  struct nodoAdyacente* enlace;
} NodoLista;
```

En el caso de que el grafo sea valorado la declaración del nodo de la lista de adyacencia:

```
typedef struct nodoAdyacente
{
  struct nodoDirectorio* destino;
  double factorPeso;
  struct nodoAdyacente* enlace;
} NodoLista;
```

El acceso a la multiestructura se hace con un puntero al primer nodo de la lista directorio: `NodoGrafo* grafo;`.

17.3. *TAD GRAFO*

Un grafo consta de un conjunto de objetos o elementos junto a sus relaciones. Es preciso definir las operaciones básicas para construir la estructura y, en general, modificar sus elementos; es decir, el *tipo abstracto de datos Grafo*.

La representación de los elementos del grafo ya ha sido estudiada, con una matiz de adyacencia o con listas de adyacencia. Ahora se definen operaciones básicas y su realización depende de la representación.

arista (u, v)	Añade el arco o arista `(u,v)` al grafo.
aristaPeso(u,v, w)	Para un grafo valorado, añade el arco `(u,v)` al grafo y el coste del arco, `w`.
borraArco(u,v)	Elimina del grafo el arco `(u,v)`.
adyacente(u,v)	Operación que devuelve *cierto* si los vértices `u, v` forman un arco.
nuevoVertice(u)	Añade el vértice `u` al grafo `G`.
borraVértice(u)	Elimina el vértice `u` del grafo `G`.

Estas operaciones son las básicas, a partir de las cuales se construye el grafo. Realmente estas operaciones tienen plena aplicación en la representación multienlazada, en la que además se hace uso de las operaciones propias de una lista enlazada.

La representación con matriz de adyacencia exige inicializar la matriz con un número fijo de nodos, y acceder a cada una de las n^2 posiciones. Por el contrario, ofrece la ventaja de que permite conocer si dos vértices, (v_i, v_j), son adyacentes con una complejidad `O(1)` ya que es un acceso a la matriz, `A(i,j)`.

17.3.1. Realización de las operaciones con listas de adyacencia

Las operaciones para construir un grafo con listas utilizan punteros y, evidentemente, variables apuntadas creadas dinámicamente. Las operaciones auxiliares, tales como crear un vértice y asignar el dato, que ahora es el nombre del vértice, no se vuelven a escribir.

NuevoVertice

Esta operación añade un nuevo vértice a la lista directorio, lista con todos los vértices del grafo. El puntero del nodo creado a la lista de adyacencia, se inicializa a `NULL`, posteriormente se crea la lista de adyacencia. Para que el tiempo de la operación sea `O(1)` se añade siempre como primer nodo de la lista. El código de la función:

```
void nuevoVertice(NodoGrafo** grafo, const char* vertice)
{
  NodoGrafo* nuevo;
  nuevo = crearNodoG(vertice);
  nuevo -> enlace = *grafo;
  *grafo = nuevo;
}
```

Arista

Con esta operación se da entrada a un nuevo arco del grafo. Se implementa de tal forma que se tiene como entrada el vértice origen y el vértice destino. Se busca la dirección (el puntero) del nodo origen y del nodo destino; esta búsqueda se realiza por la función auxiliar `direccion()`. En la lista de adyacencia del nodo origen se inserta un nuevo nodo con el vértice destino, realmente con el puntero al vértice destino. La inserción se hace como primer nodo para que el tiempo sea constante, `O(1)`.

```
void arista(NodoGrafo* grafo, const char* org, const char* dst)
{
  NodoGrafo* v1, *v2;
  NodoLista* ady;

  v1 = direccion(grafo,org);
  v2 = direccion(grafo,dst);
            /* comprueba la existencia de los vértices */
  if (v1 && v2)
  {
    ady = crearNodoA(v2);
    ady -> enlace = v1 -> lista;
    v1 -> lista = ady;
  }
}
```

La operación que añade una arista valorada, sólo presenta la diferencia de asignar el *factor de peso* al crear el nodo de la lista de adyacencia.

Borrar arco

Para eliminar un arco del grafo se parte del vértice origen y destino del arco. Una vez encontrada la dirección de ambos vértices en la lista de nodos se accede a la correspondiente lista de adyacencia para proceder a borrar el nodo que contiene al vértice destino.

```
void borrarArco(NodoGrafo* grafo, const char* org, const char* dst)
{
  NodoGrafo* v1, *v2;

  v1 = direccion(grafo,org);
  v2 = direccion(grafo,dst);
          /* comprueba la existencia de los vértices */
  if (v1 && v2)
  {
    NodoLista* ady;
    NodoLista* a = NULL;
    ady = v1 -> lista;       /* recorre lista de adyacencia */
    while (ady)
    {
      if (ady->destino == v2)
      {          /* encontrado el arco se elimina de la lista*/
        if (a == NULL)
          v1 -> lista = ady -> enlace;
        else
          a -> enlace = ady -> enlace;
        free(ady);
        ady = NULL;        /* para terminar bucle de búsqueda */
      }
      else
      {
        a = ady;
        ady = ady -> enlace;
      }
    }
  }
}
```

Adyacente

La finalidad de esta operación es determinar si dos vértices, dados sus nombres, forman un arco. En definitiva, si el vértice `v2` está en la lista de adyacencia de `v1`. La función correspondiente devuelve 1 en el caso afirmativo y 0 si no son adyacentes.

En la implementación de la función `arista()` no se ha comprobado si el vértice destino está ya en la correspondiente lista de adyacencia. Con una llamada a `adyacente()` se puede realizar la comprobación antes de insertar el nuevo nodo.

```
int adyacente(NodoGrafo* grafo, const char* org, const char* dst)
{
  NodoGrafo* v1, *v2;
  int esta = 0;

  v1 = direccion(grafo,org);
  v2 = direccion(grafo,dst);
  if (v1 &&v2)
  {
    NodoLista* ady;
```

```
        ady = v1 -> lista;        /* recorre lista de adyacencia */
        while (ady)
        {
          if (ady->destino == v2)
          {
            esta = 1;
            ady = NULL;        /* para terminar bucle de búsqueda */
          }
          else
            ady = ady -> enlace;
        }
      }
      return esta;
    }
```

Eliminar vértice

Eliminar un vértice de un grafo es una operación que puede ser considerada costosa en tiempo ya que supone acceder a cada uno de los elementos de la multiestructura. En primer lugar hay que buscar la dirección (puntero) del vértice en la lista directorio. En segundo lugar, eliminar cada uno de los nodos de la correspondiente lista de adyacencia. Y por último, recorrer cada una de las listas de adyacencia para que en el caso de encontrar algún nodo, se elimine el vértice que se quiere suprimir. *(Se deja al lector como ejercicio la implentación de la operación.)*

17.4. RECORRIDO DE UN GRAFO

En general, la operación *recorrer una estructura* consiste en visitar (procesar) cada uno de los nodos a partir de uno dado; así se ha realizado el recorrido de un árbol en, por ejemplo, *preorden* partiendo del nodo raíz. De igual forma, *recorrer un grafo* consiste en visitar todos los vértices alcanzables a partir de uno dado. Muchos de los problemas que se plantean con los grafos exigen examinar las aristas de que consta y procesar los vértices del grafo.

Sea `V` el conjunto de vértices del grafo, los algoritmos de *recorrido* de grafos parten de un nodo, `v`, y mantienen un conjunto de nodos *marcados* (*procesables*), `W`, que inicialmente es el nodo o vértice de partida, `v`. Cada *pasada* del algoritmo va retirando un nodo, `w`, del conjunto `W`, se procesa y para cada una de las aristas que tenga como origen el vértice `w`, `(w,u)`, su vértice adyacente, `u`, se añade a `W` si no está marcado. El algoritmo continúa hasta que el conjunto `W` se queda vacío, en ese momento todo vértice no marcado no es accesible desde el nodo de partida `v`. Si se necesita un recorrido completo, se puede continuar desde cualquier nodo o vértice no marcado.

A tener en cuenta

Hay dos formas de recorrer un grafo: *recorrido en profundidad y recorrido en anchura*. Si el conjunto de nodos *marcados* se trata como una cola, entonces el recorrido es en anchura; si se trata como una pila, el recorrido es en profundidad.

17.4.1. Recorrido en anchura

Este método hace uso de una cola como estructura en la que se mantienen los vértices marcados que se van a procesar posteriormente. El orden en que una cola trata los elementos hace que, a partir del

vértice de partida, v, se procesen primero todos los nodos adyacentes a v, después se procesen los nodos adyacentes de éstos que no estuvieran ya marcados o visitados, y así sucesivamente con los adyacentes de los adyacentes.

El orden de procesar o visitar los nodos, en el recorrido en anchura, se puede expresar de manera más concisa en estos pasos:

1. *Marcar* el vértice de partida v.
2. *Meter* en la cola el vértice de partida v.
3. Repetir los pasos 4 y 5 hasta que se cumpla la condición *cola vacía*.
4. *Quitar* nodo frente de la cola, w, visitar w.
5. *Meter* en la cola todos los vértices adyacentes a w que no estén marcados, a continuación marcar esos vértices.
6. Fin del recorrido.

Como ejemplo, se va a recorrer en anchura el grafo dirigido de la Figura 17.12. El recorrido se inicia a partir del nodo D, se marca y se mete en la cola. Ahora iterativamente, se quita el nodo frente, se procesa, se mete en la cola sus adyacentes no marcados y se marcan. Los sucesivos elementos de la cola se muestran en la Figura 17.13. En la columna de vértices procesados el nodo en *negrita* es el que se procesa en esa pasada, y en la cola los nodos en cursiva son los que se meten en la cola y son marcados.

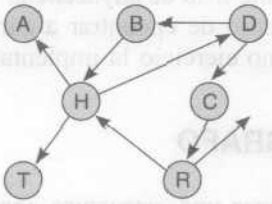


Figura 17.12. Grafo dirigido.

Cola	Vértices procesados
D	
B C	**D**
C *H*	**B**
H *R*	**C**
R A *T*	**H**
A T	**R**
T	**A**
cola vacía	**T**

Figura 17.13. Recorrido en anchura del grafo de la Figura 17.12.

El recorrido del grafo en anchura, partir del nodo D, del grafo de la Figura 17.12 ha accedido a todos los nodos del grafo y se puede afirmar que todos los vértices del grafo son alcanzables desde el vértice D.

Realización del recorrido en anchura

El recorrido en anchura utiliza una cola en la que se guardan los nodos adyacentes al que se ha procesado, por lo que al codificar el algoritmo se ha de hacer uso de las operaciones del *TAD Cola*. Para determinar si un vértice está o no marcado se pueden seguir varias alternativas, una de ellas utiliza el array m[], de tantas posiciones como nodos y con una correspondencia directa entre

índice y número de nodo, para indicar si un nodo está marcado. El estado de nodo `i` *no marcado* se establece con una *clave* (por ejemplo, valor entero suficientemente elevado), y si está marcado tendrá un número finito.

Al recorrer el grafo se puede obtener cierta información relativa a los vértices. En la codificación que se presenta en el Ejercicio 17.1 se guarda en `m[i]` el número mínimo de aristas para cualquier camino, desde el vértice de partida hasta el vértice `i`. Se supone que el vértice de partida es el `0`, por lo que se inicializa `m[0] = 0` ya que los caminos parten desde ese vértice. Las otras posiciones de `m[i]` se inicializan al valor clave que expresa *vértice no marcado*. La representación del grafo, para el recorrido en anchura, puede ser con una matriz de adyacencia, o bien con listas de adyacencia, en el ejercicio se emplea la matriz de adyacencia. La realización con listas de adyacencia sigue los mismos pasos, naturalmente con la adaptación pertinente a las listas.

Ejercicio 17.1

Se tiene un grafo dirigido de n nodos, representado por su matriz de adyacencia, `A`. Se quiere realizar el recorrido en anchura del grafo a partir del vértice `v` para determinar los vértices accesibles desde `v` y el número mínimo de arcos para alcanzarlos. También, mostrar aquellos vértices no accesibles desde `v`.

Las operaciones y la representación del tipo cola se encuentran en el archivo de cabecera `cola.h`. Se escribe la función `recorrerAnchura()` que tiene como entrada la matriz de adyacencia, el número de nodos y el vértice origen `v`. El último argumento es el array de nodos marcados `m[]`.

```
#define N 75
#define CLAVE 9999
#include "cola.h"

void recorrerAnchura(int a[][N], int n, int v, int m[])
{
  int i, u ,w;
  Cola cola;

  crearCola(&cola);

  /* inicializa los vértices como no marcados */

  for (i = 1; i < n; i++)
    m[i] = CLAVE;

  m[v] = 0;              /* vértice origen queda marcado */
  insertar(&cola,v);

  while (!colaVacia(cola))
  {
    w = quitar(&cola);
    printf("Nodo %d visitado\n", w);

    /* inserta en la cola los adyacentes de w no marcados */

    for (u = 0; u < n; u++)
      if (a[w][u] && m[u] == CLAVE)
```

```
            {
              /* vértice u es marcado con el número de arcos hasta el */
              m[u] = m[w] +1;
              insertar(&cola,u);
            }
        }
    }
```

La llamada a la función con la escritura por pantalla los nodos alcanzables y los no alcanzables:

```
recorrerAnchura(a,n,v,m);

puts("\nVértices alcanzables desde nodo origen (mínimo
        de arcos)");

for (i = 1; i < n; i++)
{
  if (m[i]!= CLAVE)
  {
    printf("\nNodo %d (%d) ",i,m[i]);
  }
}

printf("\nVértices no alcanzables desde nodo origen ");
for (i = 1; i < n; i++)
  if (m[i] == CLAVE)
    printf("\tNodo %d",i);
```

En el caso de querer visitar todos los vértices del grafo, una vez que ha terminado el recorrido a partir de uno dado, v, hay que buscar si queda algún vértice sin marcar, en cuyo caso se vuelve a recorrer a partir de él; así sucesivamente hasta que todos los vértices estén marcados.

17.4.2. Recorrido en profundidad

La búsqueda de los nodos y aristas de un grafo en profundidad persigue el mismo objetivo que el recorrido en anchura: *visitar* todos los vértices del grafo alcanzables desde un vértice dado. La diferencia de este recorrido respecto al recorrido en anchura sólo está en el orden en que se procesan los vértices adyacentes, en este nuevo recorrido el orden es el que determina la estructura pila. De tal forma que siempre se quita el último nodo añadido.

El recorrido empieza por un vértice v del grafo; éste se marca como visitado y se *mete* en la pila. Después se recorre en profundidad cada vértice adyacente a v no visitado; así hasta que no haya más vértices adyacentes no visitados. Esta estrategia de examinar los nodos se denomina en profundidad porque la dirección de *«visitar»* es hacia *adelante* mientras que sea posible; al contrario que la búsqueda en anchura que primero visita todos los vértices posibles en *amplitud*.

Como ejemplo, se recorre en profundidad el grafo dirigido de la Figura 17.12. El recorrido se inicia a partir del nodo D, se marca y se mete en la pila. Ahora, iterativamente, se quita el nodo cabeza, se procesa, se meten en la pila sus adyacentes no marcados y se marcan. Los sucesivos elementos de la pila se muestran en la Figura 17.14. En la columna de vértices procesados el nodo en *negrita* es el que se *visita* en esa pasada, en la pila los nodos en cursiva son los que se meten en la pila y a la vez son marcados.

Pila	Vértices procesados
D	
B C	**D**
B *R*	**C**
B *H*	**R**
B *A T*	**H**
B A	**T**
B	**A**
pila vacía	**B**

Figura 17.14. Recorrido en profundidad del grafo de la Figura 17.12.

Realización del recorrido en profundidad

Además de implementar este recorrido con una pila de vértices, la definición recursiva permite una implementación de la misma naturaleza. De igual forma que en el recorrido en anchura, los vértices ya visitados se marcan en el array `m[]`.

A continuación se escribe la codificación de este tipo de recorrido en un grafo, representado con listas de adyacencia, utilizando las operaciones del *TAD Pila* que se encuentran en el archivo de cabecera `pila.h`. Al utilizar listas de adyacencia resulta más cómodo referirse a los nodos con el *nombre* (cadena) que se les asigna, así las operaciones con listas de adyacencia no se modifican, aunque obliga a transformar dicho nombre al índice correspondiente, `0 .. n-1`, para acceder al array de vértices *marcados*; la función `indice()` cuyo prototipo `int indice(const char*)`; hace una búsqueda en la tabla de vértices y devuelve el índice que le corresponde.

En el Ejercicio 17.2 se codifica de nuevo este recorrido para un grafo representado con su matriz de adyacencia y utilizando la técnica recursiva.

```
#define CLAVE 9999
typedef NodoGrafo* TipoDato;
#include "pila.h"

void recorreProfundidad(NodoGrafo* grafo, int n,
                                    const char* v, int m[])
{
  NodoGrafo* v1, *v2, *w;
  NodoLista* ady;
  NodoPila* pila;

  crearPila(&pila);
        /* inicializa los vértices como no marcados */
  for (i = 0; i < n; i++)
    m[i] = CLAVE;

  m[indice(v)] = 0;          /* vértice origen queda marcado */
  v1 = direccion(grafo,v);
  insertar(&pila,v1);

  while (!pilaVacia(pila))
  {
    w = quitar(&pila);
    printf("Nodo %s visitado\n", w -> vertice);
```

```
                /* inserta en la pila los adyacentes de w no marcados */
        ady = w -> lista;
        while (ady)
        {
          int k;
            /* posición del vértice adyacente */
          k = indice(ady -> destino ->vertice);
          if (m[k] != CLAVE)
          {
            insertar(&pila,ady->destino);
            m[k] = 1;                          /* vértice es marcado */
          }
          ady = ady -> enlace;
        }
      }
    }
```

17.5. CONEXIONES EN UN GRAFO

Al modelar un conjunto de objetos y sus relaciones mediante un grafo, una de las cuestiones que generalmente interesa saber es si desde cualquier vértice se puede acceder al resto de los vértices del grafo; es decir, si todos los vértices están conectados, o simplemente, si el grafo es conexo. En el caso de un grafo dirigido, la conectividad entre todos los vértices se denomina *grafo fuertemente conexo.*

17.5.1. Componentes conexas de un grafo

Un grafo no dirigido G es conexo si existe un camino entre cualquier par de nodos que forman el grafo. En el caso de que el grafo no sea conexo resulta interesante determinar aquellos subconjuntos de vértices que mutuamente están conectados, esto es, las componentes conexas del mismo. El algoritmo para determinar las componentes conexas de un grafo G está basado en el recorrido de un grafo, tanto en *anchura* como en *profundidad*. Los pasos a seguir son:

1. Realizar un recorrido del grafo a partir de cualquier vértice w.
2. Si en el recorrido se han *marcado* todos los vértices, entonces el grafo es conexo.
3. Si el grafo no es conexo, los vértices marcados forman una componente conexa.
4. Se toma un vértice no marcado, z, y se realiza de nuevo el recorrido del grafo a partir de z. Los nuevo vértices marcados forman otra componente conexa.
5. El algoritmo termina cuando todos los vértices han sido marcados (visitados).

A continuación se escribe un segmento de código en el que se obtienen las componentes conexas de un grafo no dirigido representado con la matriz de adyacencia. Como se puede observar, todo consiste en llamar a `recorrerAnchura()` y analizar el array de nodos marcados. Las componentes se escriben por pantalla.

```
recorrerAnchura(a,n,v,m);

printf("\nComponente conexa obtenida desde el nodo %d\n",v);
```

```
for (i = 1; i < n; i++)
{
  if (m[i]!= CLAVE)
  {
    printf("\nNodo %d ",i);
  }
}

printf("\nVértices que no se encuentran en la componente conexa: ");
for (i = 1; i < n; i++)
  if (m[i] == CLAVE)
    printf("\tNodo %d",i);
```

17.5.2. Componentes fuertemente conexas de un grafo

Un grafo dirigido fuertemente conexo es aquel en el cual existe un camino entre cualquier par de vértices del grafo. De no ser fuertemente conexo, se pueden determinar componentes fuertemente conexas del grafo. La Figura 17.15 muestra un grafo dirigido y sus dos componentes fuertemente conexas.

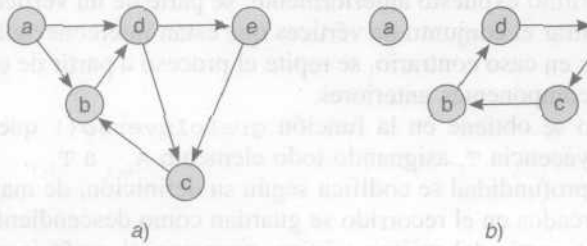


Figura 17.15. *a)* Grafo dirigido. *b)* Componentes fuertes del grafo.

El recorrido en profundidad de un grafo a partir de un vértice dado permite diseñar un algoritmo para encontrar si un grafo es fuertemente conexo, o en su caso determinar las componentes fuertemente conexas. A continuación se indican los pasos que sigue:

1. *Obtener el conjunto de vértices descendientes del vértice de partida* `v`, `D(v)`, *incluido el* propio vértice `v`.
2. Obtener el conjunto de ascendientes de `v`, `A(v)`, incluido el propio vértice `v`.
3. Los vértices comunes entre el conjunto de descendientes y ascendientes de `v`, `D(v)` $\cap$ $\cap$ `A(v)`, es el conjunto de vértices que forman la componente fuertemente conexa a la que pertenece `v`. Si este conjunto es igual al conjunto de vértices del grafo, entonces es fuertemente conexo.
4. Si no es un grafo fuertemente conexo se selecciona un vértice cualquiera, `w`, que no esté en *ninguna componente fuerte de las encontradas* (`w` $\neq$ `D(v)` $\cap$ `A(v)`) *y se procede de la* misma manera, es decir, se repite a partir de 1 hasta obtener todas las componentes fuertes del grafo.

Para buscar los vértices descendientes del vértice `v` se realiza un recorrido en profundidad del grafo a partir de `v`. Los vértices que se alcanzan se guardan en el conjunto `D`.

Los vértices ascendientes de `v` *son aquellos desde los que se puede alcanzar a* `v`. *Por lo que se* procede a cambiar la dirección de las aristas del grafo, a continuación se recorre en profundidad el grafo inverso a partir de `v`. Los vértices alcanzados en el recorrido son los ascendientes de `v`.

Al recorrer el grafo de la Figura 17.15 en profundidad a partir del vértice `d`, el conjunto de vértices que alcanza: `{d, c, b, e}`. Repitiendo el recorrido en el grafo inverso (Fig. 17.16) se obtiene los vértices ascendientes: `{d, b, c, e}`. Los vértices comunes (en este ejemplo `d, b, c, e`) forman una componente fuertemente conexa.

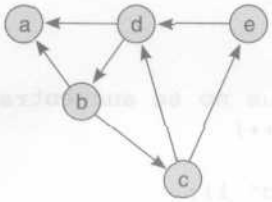


Figura 17.16. Grafo inverso del 17.15.

Ejercicio 17.2

En un grafo dirigido de `n` nodos, representado por su matriz de adyacencia, `A`, se desea determinar si el grafo es fuertemente conexo, o bien en el caso de que no lo sea, encontrar las componentes fuertemente conexas. Estas componentes se mostrarán por pantalla.

Se sigue el algoritmo expuesto anteriormente: se parte de un vértice cualquiera, por ejemplo el primero, para encontrar el conjunto de vértices que están interconectados. Si son todos, el grafo es fuertemente conexo; en caso contrario, se repite el proceso a partir de cualquier vértice que no esté formando parte de componentes anteriores.

El grafo inverso se obtiene en la función `grafoInverso()` que, sencillamente, forma una nueva matriz de adyacencia `T`, asignando todo elemento $A_{i,j}$ a $T_{j,i}$.

El recorrido en profundidad se codifica según su definición, de manera recursiva; de tal forma que los vértices marcados en el recorrido se guardan como descendientes del vértice de partida. El recorrido se repite a partir del mismo vértice, pero con el grafo inverso, los vértices marcados forman los vértices ascendientes del vértice de partida. Los vértices marcados en ambos recorridos forman una componente fuerte del grafo.

La existencia de una correspondencia biunívoca entre el número de nodo y el índice del array de nodos, `descentientes[]` y `ascendientes[]` implica que un nodo `i` está presente se activa la posición `i` del array (se pone a 1); de esa forma se guarda el nodo en el correspondiente conjunto. Esta acción facilita la operación de intersección ya que, simplemente, si se cumple `ascendientes[i] && descendientes[i]` el nodo `i` pertenece a ambos conjuntos. Además, en el array `bosque[]` se marcan los nodos que ya están en una componente conexa.

En la función *principal* se da entrada al grafo, se llama a la función que invierte el orden de las aristas e iterativamente se recorre en profundidad el grafo original, el grafo inverso para encontrar nodos descendientes y ascendientes. La intersección de ambos conjuntos da lugar a la componente conexa. El array `bosque[]` guarda los nodos, los marca, que ya forman parte de alguna componente. La función `todosArboles()` devuelve un nodo que todavía no forma parte de componente conexa, el proceso termina cuando devuelve el valor clave `-1`.

```
#define N 10
#define CLAVE 9999

int a[N][N], t[N][N];

int m[N];
int v = 0;
```

```
int n,i;
int descendientes[N], ascendientes[N], componente[N], bosque[N];

entradGrafo(a,&n);
grafoInverso(a,n,t);

          /* inicializa array que almacena vértices en componentes */

for (i = 0; i < n; i++)
  bosque[i] = 0;

                    /* Se toma el vértice 0 como vértice de partida */

v = 0;
do {
  for (i = 0; i < n; i++)
    m[i] = CLAVE;

  recorrerProfundidad(a,n,v,m);

                    /* se obtiene conjunto de vértices descendientes */

  for (i = 0; i < n; i++)
  {
    descendientes[i] = m[i] != CLAVE ? 1 : 0;
  }

                    /* recorre el grafo inverso y obtiene ascendientes */

  for (i = 0; i < n; i++)
    m[i] = CLAVE;

  recorrerProfundidad(t,n,v,m);

  for (i = 0; i < n; i++)
  {
    ascendientes[i] = m[i] != CLAVE ? 1 : 0;
  }

  printf ("\nComponente conexa { ");

  for (i = 0; i < n; i++)
  {
    componente[i] = descendientes[i] && ascendientes[i];
    if (componente[i])
    {
      printf("%d, ",i);
      bosque[i] = 1;
    }
  }
  puts("}");
```

```
      /* Vértice a partir del cual se obtiene otra componente */
    v = todosArboles(bosque,n);

  } while (v != -1);

      /* funciones invocadas */

  void recorrerProfundidad(int a[][C], int n, int v, int m[])
  {
    int u;

    m[v] = 0;

    for (u = 0; u < n; u++)
      if (a[v][u] && m[u] == CLAVE)
      {
        recorrerProfundidad(a, n, u, m);
      }
  }

  void grafoInverso(int a[][C], int n, int t[][C])
  {
    int i,j;

    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        t[j][i] = a[i][j];
  }

  int todosArboles(int bosque[C], int n)

  {
    int i;
    int w;

    w = i = -1;
    do
    {
      if (!bosque[++i])
        w = i;
    } while ((i < n-1) && (w == -1));
    return w;
  }
```

17.6. MATRIZ DE CAMINOS: CIERRE TRANSITIVO

En un grafo G un camino de longitud n desde el nodo v_0 hasta el nodo v_n es una *secuencia* de n+1 nodos $v_0, v_1, v_2, \ldots, v_n$, tal que el nodo inicial es v_0, el nodo final v_n y son adyacentes (v_i, v_{i+1}) para $0 \le i \le n$. Encontrar caminos entre un par de vértices de una determinada

longitud es una tarea relativamente sencilla, aunque poco eficiente, si se tiene la matriz de adyacencia del grafo.

Sea A la matriz de adyacencia de un grafo, la expresión $A_{i,k}$ && $A_{k,j}$ es *verdadera* si y sólo si los valores de ambos operandos son también verdaderos. Esta hipótesis implica que hay un arco desde el vértice i al vértice k y otro desde el vértice k al j. También se puede afirmar que: «la expresión sea verdadera implica la existencia de un camino de longitud 2 desde el vértice i al j».

Este sencillo razonamiento se puede ampliar, en la siguiente expresión:

$$(A_{i1} \;\&\&\; A_{1j}) \;||\; (A_{i2} \;\&\&\; A_{2j}) \;||\; \ldots \;||\; (A_{in} \;\&\&\; A_{nj})$$

En el caso se que esta expresión sea cierta implicaría que hay al menos un camino de longitud 2 desde el vértice i al vértice j que pase por el vértice 1, o por el 2 . . . o por el vértice n.

La expresión anterior se puede relacionar con el producto matricial, A * A; así, si se cambia && por la operación *producto* y || por *suma*, el resultado produce el elemento A_{ij} de la matriz producto A^2. Entonces, la conclusión inmediata es que los elementos (A_{ij}) de la matriz A^2 son distintos de cero si existe un camino de longitud 2 desde el vértice i al vértice j, $\forall i, j = 1..n$.

El razonamiento se puede extender para encontrar caminos de longitud 3; realizando el producto matricial $A^2 \times A = A^3$ se encuentran caminos de longitud 3 entre cualquier par de vértices del grafo.

A tener en cuenta

Una manera de encontrar los caminos de longitud m entre cualquier par de vértices de un grafo es mediante el producto matricial de A^{m-1} por la matriz de adyacencia A.

En la Figura 17.17 se muestra un grafo dirigido de 5 vértices, su matriz de adyacencia es:

$$A = \begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

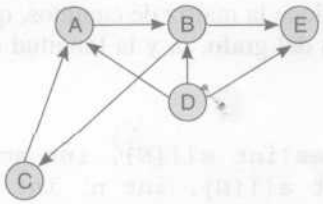


Figura 17.17. Grafo dirigido de 5 vértices.

El producto matricial A × A, considerando todo valor distinto de cero como 1 (*verdadero*):

$$A^2 = \begin{vmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Así por ejemplo, A_{15} tiene el valor 1 (verdadero) ya que hay un camino de longitud 2 desde A - E, formado por los arcos (A -> B -> E).

De igual manera, el producto matricial $A^2 \times A$:

$$A^3 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Así, A_{41} es 1 ya que hay un camino longitud 3 desde D - A, formado por los arcos: (D -> -> B -> C -> A).

Si realmente se hubiera hecho el producto matricial, el valor almacenado en cualquier posición A_{ij} no sólo indica si hay camino de longitud m, sino además el número de caminos de dicha longitud entre los vértices i y j.

Para recordar

La forma de obtener el número de caminos de longitud k entre cualquier par de vértices de un grafo es obtener el producto matricial $A^2, A^3 \ldots A^k$. Entonces el elemento $A_k(i,j)$ contiene el número de caminos de longitud k desde el vértice i hasta el vértice j.

17.6.1. Implementación en C

La codificación del algoritmo que obtiene el número de caminos de longitud k entre dos vértices cualesquiera de un grafo se puede hacer en dos funciones. La función productoMatrices() tiene como entrada dos matrices, la primera es la matriz de adyacencia A; la segunda, A^{m-1}, realiza el producto de ellas y devuelve la matriz producto A^m. Esta función es llamada repetidamente desde la función caminoLongK(); con la primera llamada se obtiene A^2, que a su vez se pasa como argumento para que en la siguiente llamada se obtenga A^3 y así hasta obtener A^k.

A continuación se escriben ambas funciones y la función auxiliar de *copia* de una matriz. En caminoLongitugK() se obtiene la matriz de caminos, que tiene como entrada la matriz de adyacencia, el número de vértices del grafo, n, y la longitud de los caminos, k.

```
#define N 30
void productoMatrices(int a[][N], int am[][N], int n, int p[][N]);
void caminoLongK(int a[][N], int n, int k, int c[][N]);
void copia(int a[][N], int d[][N],int n);

void productoMatrices(int a[][N], int am[][N], int n, int p[][N])
{
  int f, c, k, s;
  for (f = 0; f < n; f++)
  {
    for (c = 0; c < n; c++)
    {
      s = 0;
```

```
            for (k = 0; k < n; k++)
                s += a[f][k] * am[k][c];
            p[f][c] = s;
        }
    }
}

void copia(int a[][N], int d[][N], int n)
{
    int f,c;

    for (f = 0; f < n; f++)
        for (c = 0; c < n; c++)
            d[f][c] = a[f][c];
}

void caminoLongK(int a[][N], int n, int k, int c[][N])
{
    int j;
    int am[N][N];

    copia(a,am,n);
    productoMatrices(a,am,n,c);

    for (j = 3; j <= k; j++)
    {
        copia(c,am,n);
        productoMatrices(a, am, n, c);
    }
}
```

A tener en cuenta

La eficiencia del algoritmo que encuentra el número de caminos de longitud `k` es muy *pobre*. El producto matricial se realiza con tres bucles anidados, complejidad cúbica $O(n^3)$; además, este producto se realiza $k - 1$ veces.

17.6.2. Matriz de caminos y cierre transitivo

Sea `G` un grafo con `n` vértices, la matriz de caminos de `G` es la matriz `P` de `n * n` elementos, definida:

$$P_{ij} \begin{cases} 1 & \textit{si hay un camino desde } (v_i, v_j) \\ 0 & \textit{si no hay camino desde } (v_i, v_j) \end{cases}$$

Se puede encontrar la siguiente relación entre la matriz de caminos `P`, la matriz de adyacencia `A` y las sucesivas potencias de `A` para encontrar los caminos de longitud `m`: «dada la matriz $B_n = A + A_2 + A_3 + \ldots + A_n$, la matriz de caminos `P` es tal que un elemento $P_{ij} = 1$ *si y sólo si* $B_n(i,j) = 1$, y en otro caso, $P_{ij} = 0$».

El cierre transitivo o contorno transitivo de un grafo `G` es otro grafo `G'` que consta de los mismos vértices que `G` y que tiene como matriz de adyacencia la matriz de caminos `P` del grafo `G`.

17.7. PUNTOS DE ARTICULACIÓN DE UN GRAFO

Un punto de articulación de un grafo no dirigido es un vértice `v` que tiene la propiedad de que si se elimina junto a sus arcos, el componente conexo en que está el vértice se divide en dos o más componentes. Por ejemplo, en la Figura 17.18 se tiene un grafo con dos puntos de articulación, el vértice `A` y el vértice `C`. En la Figura 17.18*b* se observa que al retirar el vértice `A` y sus arcos, el grafo, que es conexo, se divide en estas dos componentes conexas: `{C,B,E,F}` y `{D}`. De igual forma, en 17.18*c* se puede observar que al eliminar el vértice `C` el grafo se divide en dos componentes conexas: `{B,E,F}` y `{A,D}`. Sin embargo, al suprimir cualquier otro vértice del grafo, éste no se divide en componentes.

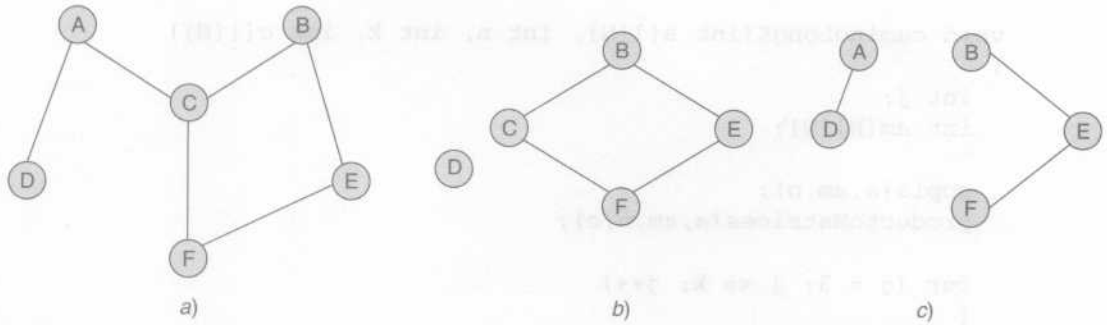


Figura 17.18. *a*) Grafo origen. *b*) Después de quitar el nodo A. *c*) Después de quitar nodo C.

El estudio de los *puntos de articulación* es debido a que los grafos tienen propiedades relativas a ellos. Así, un grafo sin puntos de articulación se dice que es un *grafo biconexo*. De no ser el grafo biconexo, es interesante encontrar componentes biconexos. Un grafo tiene *conectividad* `k` si la eliminación de `k-1` vértices cualesquiera del grafo no lo divide en componentes conexas (no lo desconecta).

A considerar

El estudio de los puntos de articulación de ciertos grafos, como las redes, es importante ya que cuanto mayor es la *conectividad* de un grafo tanto mayor probabilidad tiene de mantener la estructura ante el *fallo* (eliminación) de alguno de sus vértices.

17.7.1. Búsqueda de los puntos de articulación

El algoritmo de búsqueda utiliza el recorrido en profundidad de un grafo para encontrar todos los puntos de articulación. El recorrido recursivo en profundidad de un grafo a partir de un vértice `A` puede ser representado mediante un árbol de expansión. La raíz del árbol es el vértice de partida `A`, cada arco del grafo será una arista en el árbol.

Se aprovecha el recorrido para encontrar aristas del grafo *hacia delante* y aristas *hacia atrás*. Así, si en el recorrido por los vértices adyacentes de `v`, arcos `(v,u)`, el vértice `u` no está visitado, entonces `(v,u)` es un arco *hacia delante* y pasa a ser una arista del árbol. Si el vértice `u` está ya

visitado pero no se ha terminado de procesar todos sus adyacentes, entonces `(v,u)` es un arco *hacia atrás*, no será una arista verdadera del árbol y por ello se dibuja con una línea discontinua.

La Figura 17.19 muestra un grafo no dirigido y el árbol a que da lugar el recorrido en profundidad a partir del vértice `A`. El recorrido empieza visitando los vértices `B`, `C`, `D` que forman las correspondientes aristas del árbol, Figura 17.19*b*. Los vértices adyacentes de `D` sin procesar son `E`, `F` y `A`. Los arcos `(D,E)`, `(D,F)` forman las correspondientes aristas del árbol; el arco `(D,A)` es considerado *hacia atrás* ya que `A` fue marcado anteriormente. En el árbol de la Figura 17.19*b* se señala la arista `D - A` con línea discontinua.

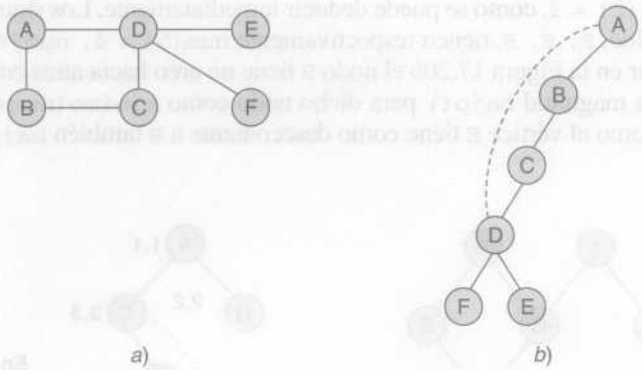


Figura 17.19. *a*) Grafo no dirigido. *b*) Árbol de expansión.

En el recorrido para formar el árbol se numeran los nodos del árbol y así se obtiene el orden en que han sido visitados. El algoritmo para encontrar los puntos de articulación de un grafo conexo sigue estos pasos:

1. Recorrer el grafo en profundidad a partir de cualquier vértice. Se numeran en el orden en que son visitados los vértices; esta numeración queda asociada a cada vértice con `num(v)`.
2. Para cada vértice `v` del árbol del recorrido en profundidad se determina el vértice de numeración *más baja*, denominado `bajo(v)`, que es alcanzable desde `v` a través de cero o más aristas del árbol y como mucho una arista *hacia atrás*.
3. Una vez que se tienen los valores `num(v)`, `bajo(v)` se determinan los puntos de articulación aplicando estas reglas:
   - La raíz del árbol (vértice de partida) es punto de articulación si y sólo si tiene dos o más hijos.
   - Cualquier otro vértice `w` es punto de articulación si y sólo si `w` tiene al menos un hijo `u` tal que `bajo(u)` ≥ `num(w)`.
4. Fin del algoritmo.

En el paso 2 se introduce el concepto de numeración *más baja*, `bajo(v)`, magnitud asociada a cada vértice. Su cálculo se expresa matemáticamente como el mínimo de los siguientes tres valores:

a) `num(v)`.
b) El menor valor de `bajo()` para los vértices `w` de las aristas *hacia atrás* `(v,w)` del árbol.
c) El menor valor de `bajo()` para los vértices `w` de las aristas `(v,w)` del árbol.

En el Ejemplo 17.1 se obtienen las *numeraciones*, `num()` y `bajo()`, de los vértices del grafo no dirigido de la Figura 17.20. Aplicando las reglas descritas en el paso 3 del algoritmo, tiene dos puntos de articulación, el vértice `A` y el `C`.

Ejemplo 17.1

Dado el gafo de la Figura 17.20*a* encontrar las numeraciones de cada uno de sus vértices y comprobar que coinciden con las señaladas en la Figura 17.20*b*.

El número de vértices del grafo es 6, y para facilitar la comprensión, se numeran como `0, 1, ... 5` correspondiéndose con `A .. F`.

El recorrido empieza en `A` (vértice 0), su numeración es `num(0) = bajo(0) = 1`. Los vértices que siguen en el recorrido, vértices `D` y `C`, tienen como numeración `num(3) = bajo(3) = 2`, `num(2) = bajo(2) = 3`, como se puede deducir inmediatamente. Los siguientes vértices por los que se pasa en el recorrido, `F, E, B`, tienen respectivamente `num(5) = 4, num(4) = 5, num(1) = 6`. Como se puede observar en la Figura 17.20*b* el nodo `B` tiene un arco hacia atrás cuyo destino es `C`, por tanto, se puede calcular la magnitud `bajo()` para dicho nodo como `mínimo(num(B), num(C))=3`. Entonces `bajo(1) = 3`, como el vértice `E` tiene como descendiente a `B` también `bajo(4) = 3` y lo mismo con el vértice `C`.

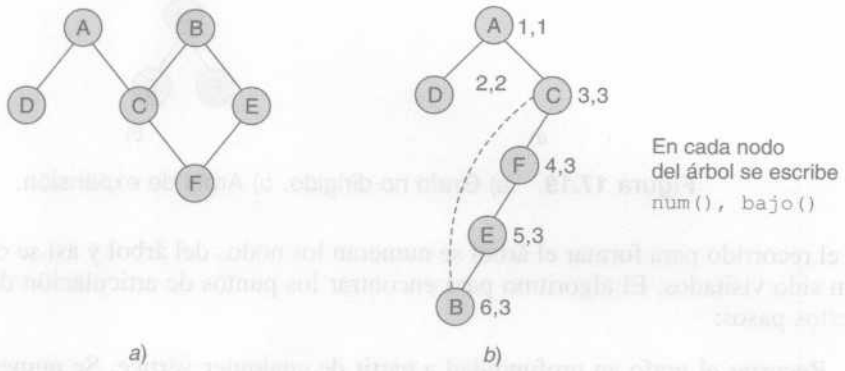


Figura 17.20. *a*) Grafo no dirigido. *b*) Árbol de expansión con *numeraciones*.

17.7.2. Implementación en C

La forma de asignar los correspondientes valores de `num(v)` a cada vértice del grafo consiste en incrementar, en cada llamada recursiva a la función de recorrido el contador de llamadas, que es la magnitud `num(v)` para el vértice *actual*. A la vez, en el array `arista[]` se guarda `v` en la posición que se corresponde con el siguiente vértice adyacente, `w`, por el que seguirá el recorrido: `arista[w] = v`; los vértices `(v,w)` forman una arista del árbol de expansión.

Con el array `arista[]` se determinan las aristas hacia atrás con este razonamiento: *si al analizar los vértices adyacentes de* `v` *resulta que si* `w` *está ya visitado y arista*`[v] != w` *es que* `(v,w)` *es una arista hacia atrás.*

A continuación se presenta la función recursiva `puntosArticulacion()` para un grafo no dirigido conexo, representado con su matriz de adyacencia. En esta implementación se utiliza el array `visitado[]`, indexado por el nodo, para marcar un nodo cuando se *pase* por él. El número de nodos se supone que es `n`.

```
#define N 40

int a[N][N];
int num[N], bajo[N], arista[N], visitado[N];
int n, i, paso;
```

```
    /* Llamada a la función tomando el vértice 0 como partida */

  for (i = 0; i < n; i++)
    visitado[i] = 0;
  v = 0;
  puntosArticulacion(a,n,v,num,0,visitado,arista,bajo);

    /* código de la función */

  void puntosArticulacion(int a[][N], int n, int v, int num[],
                          int paso, int visitado[], int arista[], int bajo[])
  {
    int w;

    visitado[v] = 1;
    num[v] = ++paso;
    bajo[v] = num[v];          /* valor inicial para cálculo de bajo() */
    for (w = 0; w < n; w++)
    {
      if (a[v][w])                                        /* adyacente w */
      {
        if (!visitado[w])
        {
          arista[w] = v;      /* arista del árbol de expansión */
          puntosArticulacion(a,n,w,num,paso,visitado,arista,bajo);
          if (bajo[w] >= num[v])     /* v cumple la regla 3.2 */
            printf("\nVértice %d es punto de articulación.\n",v);

          bajo[v] = minimo(bajo[v],bajo[w]);
        }
        else if (arista[v] != w)              /* arco hacia atrás */
          bajo[v] = minimo(bajo[v],num[w]); /* regla b */
      }
    }
  }
```

RESUMEN

Un grafo G consta de un par de conjuntos (G = (V,E)): conjunto V de vértices o nodos y el conjunto E de *aristas* (que conectan dos vértices). Si las parejas de vértices que forman una arista no están ordenadas, G se denomina grafo *no dirigido*; si los pares están ordenados, entonces G se denomina grafo *dirigido*. El término grafo *dirigido* se suele también designar como *digrafo* y el término grafo sin calificación significa grafo *no dirigido*.

El método natural para dibujar un grafo es representar los vértices como puntos o círculos y las aristas como segmentos de líneas o arcos que conectan los vértices. Si el grafo está *dirigido*, entonces los segmentos de línea o arcos tienen puntas de flecha que indican la dirección.

Los grafos se pueden implementar de dos formas típicas: *matriz de adyacencia* y *listas de adyacencia*. Cada una tiene sus ventajas y desventajas relativas. La elección depende, entre otros factores, de la densidad del grafo; para grafos *densos*, con muchos arcos, se recomienda representarlos con la *matriz de adya-*

cencia. Grafos poco densos y que experimenten modificaciones en sus componentes se representan con *listas de adyacencia*.

Dos vértices de un grafo *no dirigido* se llaman adyacentes si existe una *arista* desde el primero al segundo. Un *camino* es una secuencia de vértices distintos, cada uno adyacente al siguiente. Un *ciclo* es un *camino* que contiene al menos tres vértices, tal que el último vértice en el camino es *adyacente* al primero. Un grafo se denomina *conectado* si existe un camino desde cualquier vértice a cualquier otro vértice.

Un grafo *dirigido* se denomina *conectado fuertemente* si hay un *camino* dirigido desde un vértice a cualquier otro. Si se suprime la dirección de los arcos y el grafo *no dirigido* resultante se conecta, se denomina grafo dirigido *débilmente conectado*.

El *recorrido* de un grafo, *visita* a cada uno de los vértices, puede ser, en analogía con los árboles, *recorrido en profundidad* y *recorrido en anchura*. El recorrido en profundidad es una generalización del recorrido en *preorden* de un árbol. El recorrido en anchura visita los vértices por *niveles*, al igual que el recorrido por anchura de un árbol, permite encontrar el número mínimo de aristas para alcanzar un vértice cualquiera desde un vértice de partida.

Los *puntos de articulación* de un grafo *conexo* son aquellos vértices que si se retiran del grafo, junto a sus aristas, dividen a éste en dos *componentes conexas*. El algoritmo que permite encontrar los *puntos de articulación*, construye el *árbol de expansión* del grafo a partir de un nodo origen; recorre en profundidad el grafo buscando *aristas de árbol* y *aristas hacia atrás*.

EJERCICIOS

17.1. Dado el grafo no dirigido, G, de la Figura 17.21.

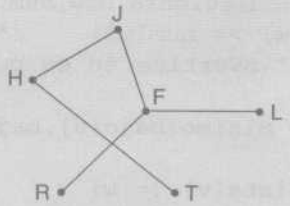


Figura 17.21. Grafo no dirigido.

a) Describir G formalmente en términos del conjunto de nodos, V, y del conjunto A de arcos.
b) Encontrar el grado de cada nodo.

17.2. Dado el grafo dirigido, G, de la Figura 17.22.

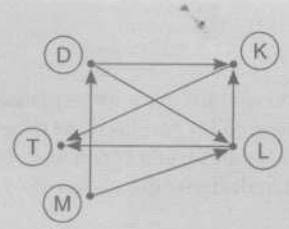


Figura 17.22. Grafo dirigido.

a) Describir G formalmente en términos de su conjunto de nodos, V, y de su conjunto A de arcos.
b) Encontrar el grado de entrada y el grado de salida de cada vértice.
c) Escribir la secuencia de vértices que forman los caminos simples del vértice M al vértice T.

17.3. Dado el grafo no dirigido, G, de la Figura 17.23:

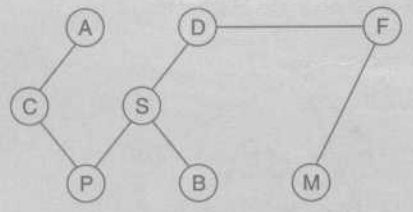


Figura 17.23. Grafo no dirigido.

a) Escribir la secuencia de vértices que forman los caminos simples del vértice A al vértice F.
b) Encontrar el camino más corto (en cuanto a número de aristas) del vértice C al vértice D.
c) ¿Es un grafo conexo?

17.4. Dado el grafo dirigido y valorado, G, de la Figura 17.24.

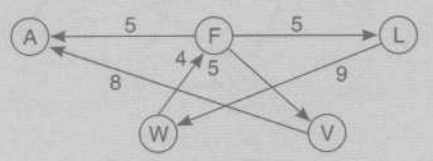


Figura 17.24. Grafo valorado.

a) Escribir la matriz de pesos del grafo.
b) Representar el grafo mediante listas de adyacencia.

17.5. Un grafo está formado por los vértices V = {A, B, C, D E}; su matriz de adyacencia, suponiendo los vértices numerados del 0 al 4 respectivamente es:

$$M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

a) Dibujar el grafo que le corresponde.
b) Representar el grafo mediante listas de adyacencia.

17.6. 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.

a) Dibujar un grafo de 6 nodos biconexo.
b) Determinar si los grafos de las Figuras 17.21 y 17.22 son biconexos.

17.7. Dado el grafo no dirigido del Ejercicio 17.5 realizar el recorrido en profundidad a partir del vértice C.

17.8. Dado el grafo no dirigido del Ejercicio 17.5 realizar el recorrido en anchura a partir del vértice C y la longitud de los caminos mínimos a los demás vértices.

17.9. En la Figura 17.25 se representan dos grafos dirigidos. Teniendo en cuenta que un grafo dirigido se considera *acíclico* si no tiene ciclos, también se denominan **gda.** Indicar si los grafos de la figura son **gda,** y en caso de no serlo buscar los ciclos.

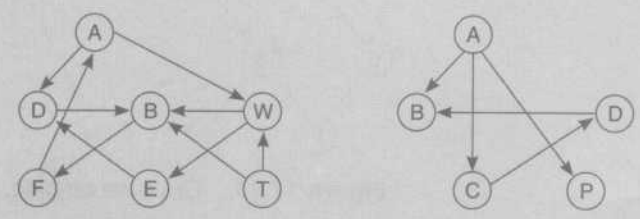


Figura 17.25. Grafos dirigidos.

17.10. Dado el grafo de la Figura 17.26, encontrar la componentes fuertemente conexas.

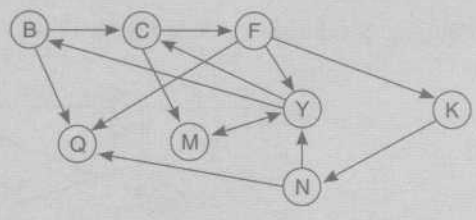


Figura 17.26. Grafo dirigido.

17.11. Dado el grafo `G` de la Figura 17.27:

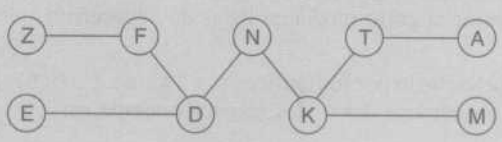


Figura 17.27. Grafo no dirigido.

a) Escribir la matriz de adyacencia del grafo.
b) Escribir la matriz de caminos de `G`.

17.12. Dado un grafo dirigido en el que los vértices son números enteros positivos y el par `(x,y)` es una arista si `x-y` es múltiplo de `3`:

a) Representar el grafo formado por los vértices 3 al 14.
b) Determinar el grado de entrada y el grado de salida de cada vértice.

17.13. En los grafos no dirigidos de las Figuras 17.23 y 17.27:

a) Dibujar los correspondientes árboles de expansión. Asociar a cada nodo su *numeración* `num()` y `bajo()`.
b) Encontrar los puntos de articulación.

17.14. Dibujar un grafo dirigido en el que los vértices son números enteros positivos del 3 al 15 para cada una de las siguientes relaciones:

a) `v` es adyacente de `w` si `v+2w` es divisible entre 3.
b) `v` es adyacente de `w` si `10v+w < v*w`.

PROBLEMAS

17.1. Diseñar un grafo valorado con los siguientes requisitos:

- Consta de 10 vértices que son números aleatorios del 11 al 99.
- Dos vértices `v`, `w`, están relacionados si `v+w` es múltiplo de 3.

a) Escribir un programa en el que se genere un grafo con las condiciones descritas. El grafo se ha de representar con un matriz de adyacencia.
b) Añadir al programa las funciones necesarias para determinar la matriz de caminos utilizando las potencias de la matriz de adyacencia.

17.2. 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.

a) Escribir un programa que represente el grafo en listas de adyacencia.
b) Añadir al programa el código necesario para realizar un recorrido en anchura desde un vértice dado.

17.3. Una región está formada por 12 comunidades. Se establece la relación de desplazamiento de personas en las primeras horas del día. Así la comunidad A está relacionada con la comunidad B si desde A se desplazan `n` personas a B, de igual forma puede haber relación entre B y A si se desplazan `m` personas de B hasta A.

a) Escribir un programa que represente el grafo descrito mediante listas de adyacencia.
b) Determinar si el grafo formado tiene *fuentes* o *sumideros*.

17.4. Dado el grafo descrito en el Problema 17.3, escriba un programa con el grafo representado 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.

17.5. Un grafo en el que los vértices son regiones y los arcos relacionan dos regiones entre las cuales hay un flujo de emigrantes (tiene factor de peso), está representado mediante una lista directorio que contiene a cada uno de los vértices y de las que sale una lista circular con los vértices adyacentes. Ahora se quiere representar el grafo mediante una matriz de pesos, de tal forma que si entre dos vértices no hay arco el de la matriz contiene el valor 0, y si entre dos vértices hay arco su posición contiene el factor de peso que le corresponde.

Escribir las funciones necesarias de modo que partiendo de la representación mediante listas se obtenga la representación mediante la matriz de pesos.

17.6. Representar la información sobre un centro de enseñanza que se enumera a continuación:

- 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. Hacer uso de estructuras de datos dinámicas.

17.7. 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, calcular el bosque abarcador, y los arcos de avance, retroceso y cruzado.

17.8. Se tiene un grafo no dirigido poco denso. Se elige la representación en memoria mediante listas de adyacencia. Escribir un programa en el que se de entrada a los vértices del grafo y sus arcos y se determine si el grafo tiene ciclos. En caso positivo, listar los vértices que forman un ciclo.

17.9. Supóngase un grafo no dirigido y conexo. Escribir un programa que encuentre un camino que vaya a través de todas las aristas exactamente una vez en cada dirección.

17.10. 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 en C que lea un grafo e imprima todos sus caminos hamiltonianos, si los hay.

17.11. Escribir un programa que compruebe si un grafo dirigido, leído del teclado tiene circuitos (ciclos) mediante el siguiente algoritmo:

1. Obtener los sucesores de todos los vértices.
2. Buscar un vértice sin sucesores y *tachar* (eliminar) ese vértice donde aparezca (conjuntos de sucesores).
3. Se continúa el proceso en el paso 2 siempre que haya algún vértice sin sucesores.
4. Si todos los vértices del grafo han sido eliminados, no tienen circuitos.

18

GRAFOS II: ALGORITMOS

OBJETIVOS

Con el estudio de este capítulo usted podrá:

- Calcular el camino mínimo desde un vértice al resto de los vértices.
- Determinar si hay camino entre cualquier par de vértices de un grafo.
- Implementar en C los algoritmos sobre grafos manejando la matriz de adyacencia.
- Relacionar objetos de forma que se representen con un grafo y aplicar los algoritmos que determinan su conectividad.
- *Saber cuándo se aplica un algoritmo de expansión de coste mínimo y cuando un camino de coste mínimo.*
- Conocer el problema del flujo máximo en una red.

CONTENIDO

18.1. Ordenación topológica.
18.2. Matriz de caminos: algoritmo de Warshall.
18.3. Caminos mínimos con un solo origen: algoritmo de Dijkstra.
18.4. Caminos mínimos para todo vértice: algoritmo de Floyd.
18.5. *Árbol de expansión de coste mínimo.*
18.6. Grafos para representar redes de flujo.

RESUMEN

EJERCICIOS

PROBLEMAS

INTRODUCCIÓN

Existen numerosos problemas que se pueden modelar en términos de grafos. Ejemplo de ello es la *planificación de las tareas que completan un proyecto, encontrar las rutas de menor longitud* entre dos puntos geográficos, calcular el camino más rápido en un transporte, determinar el flujo máximo que puede llegar desde una *fuente* a, por ejemplo, una urbanización.

La resolución de estos problemas requiere examinar todos los nodos y las aristas del grafo que representa al problema. Los algoritmos imponen implícitamente un orden en estas visitas: el nodo más próximo o las aristas más cortas, y así sucesivamente; no todos los algoritmos requieren un orden concreto en el recorrido del grafo.

Según lo anterior, se estudian en este capítulo el concepto de ordenación topológica, los problemas del camino más corto, junto con el concepto de árbol de expansión de coste mínimo; también se estudia uno de los problemas típicos de flujos, el *flujo máximo*. Estos algoritmos han sido desarrollado, por grandes investigadores y en su honor se conocen por sus nombres: Dijkstra, Warshall, Prim, Kruscal, Ford-Fulkerson ...

18.1. ORDENACIÓN TOPOLÓGICA

Una de las aplicaciones de los grafos es modelar las relaciones que existen entre las diferentes tareas, *hitos*, que deben de terminarse para dar por terminado un proyecto. Entre las tareas hay relaciones de precedencia: una tarea `r` precede a la tarea `t` si es necesario que se complete `r` para poder empezar `t`. Estas relaciones de precedencia se representan mediante un grafo dirigido en el que los vértices son las tareas o *hitos* y hay una arista del nodo `r` al nodo `t` si el inicio de la tarea `t` depende de la terminación de `r`. Una vez se dispone del grafo interesa obtener una planificación de las tareas que constituyen el proyecto; en definitiva, encontrar la *ordenación topológica* de los vértices que forman el grafo.

El grafo que representa estas relaciones de precedencia es un grafo dirigido *acíclico*, de tal forma que si hay un camino de `u` a `v`, entonces, en la ordenación topológica `v` es posterior a `u`. Es inmediato que el grafo no puede tener ciclos cuando representa relaciones de precedencia, de existir un ciclo significaría que si `r` y `t` son vértices del ciclo, `r` depende de `t`, a su vez `t` depende de la terminación de `r`.

A tener en cuenta

La ordenación topológica se aplica sobre grafos dirigidos sin ciclos (acíclico). Es una ordenación lineal, tal que si `v` es anterior a `w` entonces hay un camino de `v` a `w`. La ordenación topológica no se puede realizar en grafos con ciclos.

La Figura 18.1 muestra un grafo acíclico que modela la estructura de *prerrequisitos* de 8 cursos. Una arista cualquiera `(r,s)` significa que el curso `r` debe de completarse antes de empezar el curso `s`. Por ejemplo, el curso `M21` se puede empezar sólo cuando terminen los cursos `E11` y `T12`; en ese sentido `E11` y `T12` son *prérequisitos* de `M21`.

Una ordenación topológica de estos cursos es cualquier secuencia de cursos que cumple los requerimientos (*prerrequisitos*). Entonces para un grafo dirigido acíclico no tiene por qué existir una única ordenación topológica.

Del grafo de requisitos de la Figura 18.1 se obtienen estas ordenaciones topológicas:

```
E11 - T12 - M12 - C22 - R23 - S31 - S32 - T41
T12 - E11 - R23 - C22 - M21 - S31 - S32 - T41
```

Un grafo `G` dirigido y sin ciclos se denomina un *gda* o grafo acíclico. Los *gda* son útiles para la representación de *ordenaciones parciales*. Una *ordenación parcial* `R` en un conjunto `C` es una relación binaria de precedencia tal que:

1. Para todo `u` ∈ `C`, `u` no está relacionado con sí mismo, `u R u` es falso, por tanto, la relación `R` es *irreflexiva*.
2. Para todo `u, v, w` ∈ `C`, si `u R v` y `v R w` entonces `u R w` es verdadero. Por tanto, la relación `R` es *transitiva*.

Tal relación `R` sobre `C` se llama ordenación parcial de `C`. La relación de inclusión en conjuntos es un ejemplo de ordenación parcial. Un grafo `G` sin ciclos se puede considerar un conjunto parcialmente ordenado.

Nota

Se puede probar que un grafo es *acíclico* realizando una búsqueda en profundidad, de tal forma que si se encuentra un arco de *retroceso* en el árbol de búsqueda, el grafo tiene al menos un ciclo.

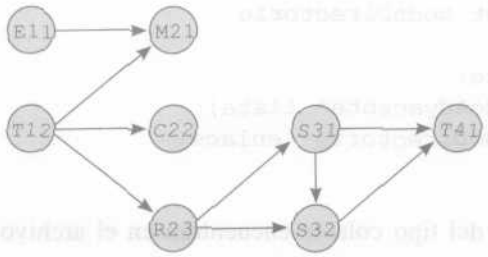


Figura 18.1. Grafo dirigido de prerrequisitos.

18.1.1. Algoritmo para obtener una ordenación topológica

El algoritmo más sencillo para encontrar la secuencia lineal de vértices que forman la ordenación, busca en primer lugar un vértice (una tarea) sin *predecesores* o sin *prerrequisitos*, esto es, que no tenga arcos de entrada. Este vértice, `v`, pasa a formar parte de la ordenación `T`; a continuación, todos los arcos que salen de `v` son eliminados (eliminación lógica), debido a que el prerrequisito `v` ya se ha satisfecho. La estrategia se repite, se toma otro vértice `w` sin arcos incidentes, se incorpora a la ordenación `T` y se eliminan todos los arcos que salen de él; así sucesivamente hasta completar la ordenación.

Precisando un poco más la estrategia, el que un vértice `v` no tenga arcos incidentes se expresa con el grado de entrada, `gradoEntrada(v)==0`. Eliminar los arcos que salen de `v`, implica que `gradoEntrada(w)` de todos los vértices `w` adyacentes de `v` disminuyen en `1`.

Por consiguiente, el algoritmo comienza calculando el grado de entrada de cada vértice del grafo. En una estructura de tipo Cola se guardan los vértices cuyo grado entrada es `0`. El elemento frente de la cola `v` pasa a formar parte de la ordenación `T`; a continuación, se disminuye en `1` el grado de entrada de los adyacentes de `v`, y aquellos vértices cuyo grado de entrada ha pasado a ser `0` se meten en la cola; así sucesivamente hasta que la cola esté vacía.

> **A tener en cuenta**
>
> Si al aplicar el algoritmo de ordenación topológica hay vértices del grafo que aún no han pasado a formar parte de la ordenación y la cola está vacía, entonces el grafo tiene ciclos.

18.1.2. Implementación en C del algoritmo de ordenación topológica

Lo primero que se debe hacer, al codificar un algoritmo de grafos, es la representación del grafo: hay que elegir entre matriz de adyacencia y listas de adyacencia. En el caso de que el grafo tenga relativamente pocos arcos, la matriz de adyacencia tiene muchos ceros, es una matriz *esparcida*; por ello su representación es con listas de adyacencia. En el caso de grafos dirigidos *densos* se prefiere, por eficiencia, la matriz de adyacencia.

Independientemente de la representación se utiliza el tipo abstracto cola, para almacenar los vértices con grado de entrada `0`. Además, se ha de escribir la función que calcula el grado de entrada para un vértice dado.

La codificación que a continuación se escribe supone el grafo representado con listas de adyacencia, obtiene la ordenación topológica de un grafo de `N` vértices que se escribe por pantalla, a la vez se guarda en el vector `T[]`. Para simplificar, los vértices se suponen identificados por un entero, en el rango de `0` a `N-1`. En el Apartado 17.3 se pueden consultar los tipos de datos y las operaciones de *TAD Grafo*, con la pequeña modificación del identificador del vértice:

```
typedef struct nodoDirectorio
{
   int vertice;
   struct nodoAdyacente* lista;
   struct nodoDirectorio* enlace;
}NodoGrafo;
```

Las operaciones del tipo cola se encuentran en el archivo `cola.h`, los elementos de la cola ahora son vértices.

```
#include "grafo.h"
typedef int TipoElemento;
#include "cola.h"
#define N 15

typedef struct nodoAdyacente
{
   struct nodoDirectorio* destino;
   struct nodoAdyacente* enlace;
} NodoLista;

                /* función que devuelve el grado de entrada del vértice v */

int gradoEntrada (NodoGrafo* g, int v)
{
   int cuenta;
   int origen;

   cuenta = 0;
   while (g)
   {
      origen = g -> vertice;
      if (adyacente(g,origen,v))                /* arco incidente a v */
         cuenta++;
      g = g -> enlace;
   }
   return cuenta;
}

                /* función para obtener una ordenación topológica */
void ordenTopologica (NodoGrafo* g, int T[])
{
   int arcosInciden[N];
   int v, w, j;
   NodoGrafo* pu, pw;
   NodoLista* lad;
   Cola cola;

   puts("\n Ordenación topológica: ");

                /* grado de entrada de cada vértice */
```

```
    for (v = 0; v < N; v++)
      arcosInciden[v] = gradoEntrada(g,v);

    crearCola(&cola); j = 0;
    for (v = 0; v < N; v++)
      if (arcosInciden[v] == 0)
        insertar(&cola,v);

    while (!colaVacia(cola))
    {
      w = quitar(&cola);
      printf("%2d ",w);
      T[j++] = w;
      pw = direccion(grafo,w);          /* dirección(puntero) al nodo */
      lad = pw -> lista;              /* puntero a lista de adyacencia */
                  /* Decrementa grado entrada de vértices adyacentes */
      while (lad)
      {

        pu = lad -> destino;
        v = pu -> vertice;
        arcosInciden[v] --;
        if (arcosInciden[v] == 0)
          insertar(&cola,v);
        lad = lad -> enlace;
      }
    }
  }
```

A tener en cuenta

La complejidad del algoritmo *ordenación topológica* de un grafo, representado con listas de adyacencia, es `O(a+n)`, siendo `a` el número de arcos y `n` el de vértices. Si la representación del grafo es con una matriz de adyacencia, la complejidad es $O(n^2)$.

18.2. MATRIZ DE CAMINOS: ALGORITMO DE WARSHALL

En el Apartado 17.6, se ha estudiado la forma de obtener la matriz de caminos, calculando las sucesivas potencias de la matriz de adyacencia. El método es poco eficiente debido a los sucesivos productos matriciales a realizar. Warshall propuso un algoritmo más eficiente para calcular la matriz de caminos (también llamado *cierre transitivo*).

Se parte de un grafo `G` de `n` vértices, representado por su matriz de adyacencia `A`, y se desea encontrar la matriz de caminos `P` del grafo `G`. La estrategia que sigue el algoritmo consiste en definir, a nivel lógico, una secuencia de matrices *n-cuadradas* P_0, P_1, P_2, P_3 ... P_n; los elementos de cada una de las matrices $P_k[i,j]$ tienen el valor `0` si no hay camino, `1` si hay camino del vértice `i` y al `j`.

La matriz P_0 es la matriz de adyacencia; los elementos de P_1 son iguales a 1 si lo son los de P_0, y los que son 0 en P_0 pasan a 1 en P_1 si se puede formar un camino entre el par de vértices, con la ayuda del vértice 1. En definitiva, $P_1[i,j] = 1$ si lo es $P_0[i,j]$, o bien, hay un camino:

La matriz P_2 se forma a partir de P_1, añadiendo el vértice 2 para poder formar camino entre dos vértices todavía no conectados. La diferencia entre dos matrices consecutivas P_k y P_{k-1} viene dada por la incorporación del vértice de orden k para estudiar si es posible formar camino del vértice i y al j con la ayuda del vértice k. La descripción de cada matriz:

$$P_0[i,j] = \begin{cases} 1 & \text{si hay un arco del vértice } i \text{ al } j. \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_1[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por ningún otro} \\ & \text{vértice a no ser por el vértice } 1. \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_2[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por} \\ & \text{otro vértice a no ser por los que están comprendidos} \\ & \text{entre los vértices } 1 \ .. \ 2. \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_3[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por} \\ & \text{otro vértice a no ser por los que están comprendidos} \\ & \text{entre los vértices } 1 \ .. \ 3. \\ 0 & \text{en otro caso.} \end{cases}$$

Se puede observar que en cada paso se incorpora un nuevo vértice, el que coincide con el índice de la matriz P, a los anteriores para poder formar camino.

$$P_k[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por} \\ & \text{otro vértice a no ser por los que están comprendidos} \\ & \text{entre los vértices } 1 \ .. \ k. \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_n[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por} \\ & \text{otro vértice a no ser por los que están comprendidos} \\ & \text{entre los vértices } 1 \ .. \ n. \text{ En la formación de esta última matriz} \\ & \text{intervienen todos los vértices.} \\ 0 & \text{en otro caso.} \end{cases}$$

Según estas definiciones, P_0 es la matriz de adyacencia del grafo. Y al ser un grafo de n vértices, la matriz P_n es la matriz de caminos.

El algoritmo de Warshall encuentra una relación recurrente entre los elementos de la matriz P_k y los elementos de la matriz P_{k-1}. Para que cualquier elemento $P_k[i,j] = 1$ ha de ocurrir uno de estos dos casos:

1. Exista un camino simple de v_i a v_j del que pueden formar parte los vértices de índice 1 al k-1 (v_1 a v_{k-1}); por consiguiente, que el elemento $P_{k-1}[i,j] = 1$.
2. Exista un camino simple de v_i a v_k y otro camino simple de v_k a v_j de los que pueden formar parte los vértices de índice 1 al k-1; por tanto, esto equivale a:

$(P_{k-1}[i,k] = 1)$ y $(P_{k-1}[k,j] = 1)$

$v_i \rightarrow \ldots \rightarrow v_j$ (1) $\quad v_i \rightarrow \ldots \rightarrow v_k \rightarrow \ldots \rightarrow v_j$ (2)

Camino de v_i a v_j *Camino* $v_i - v_k - v_j$

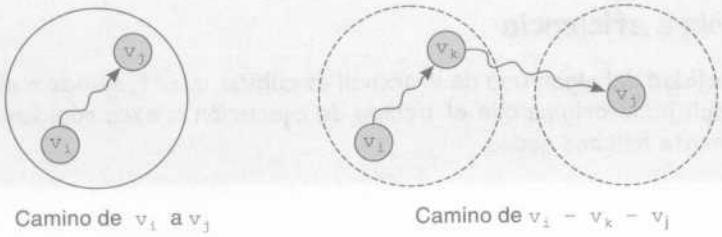


Figura 18.2. Posibilidad de camino de v_i a v_j.

En definitiva, Warshall encuentra una relación recurrente entre la matriz P_{k-1} y P_k que permite, a partir de la matriz conocida P_0, que es la matriz de adyacencia, encontrar P_n, que es la matriz de caminos. La relación para encontrar los elementos de P_k puede expresarse a la manera de una operación lógica:

$$P_k[i,j] = P_{k-1}[i,j] \text{ or } (P_{k-1}[i,k] \text{ and } P_{k-1}[k,j])$$

18.2.1. Implementación del algoritmo de Warshall

El algoritmo de Warshall, en seudocódigo, para un grafo G de n vértices, representado por su matriz de adyacencia A y cuya salida es la matriz de caminos P que se muestra a continuación:

```
{Inicializar P}
desde i <- 1 hasta n hacer
   desde j <- 1 hasta n hacer
      P[i,j] <- A[i,j]

   fin desde
fin desde

{ A nivel lógico, se obtienen las sucesivas matrices P0, P1, P2,
P3, ..., Pn-1, Pn que es la matriz de caminos del grafo}
desde k - 1 hasta n hacer
   desde i<- 1 hasta n hacer
      desde j<- 1 hasta n hacer

         P[i,j] <- P[i,j] or (P[i,k] and P[k,j])
      fin desde
   fin desde
```

```
    fin desde
fin algoritmo
```

La correspondiente codificación del algoritmo, en C, es inmediata. Sólo hay que considerar que los índices de los vértices están en el rango de `0` a `n-1`.

Nota sobre eficiencia

La complejidad del algoritmo de Warshall es cúbica, $O(n^3)$, siendo n el número de vértices. Esta complejidad origina que el tiempo de ejecución crezca rápidamente para grafos de relativamente muchos nodos.

18.3. CAMINOS MÍNIMOS CON UN SOLO ORIGEN: ALGORITMO DE DIJKSTRA

Uno de los problemas que se plantean con relativa frecuencia es determinar la longitud del camino más corto entre un par de vértices de un grafo. Así por ejemplo, determinar la ruta que en menor tiempo nos lleva desde un punto (nuestra casa) a un conjunto de puntos de la ciudad. Para este tipo de problemas se considera un grafo dirigido y *valorado* (factor de peso) en el que cada arco (v_i, v_j) tiene asociado un coste c_{ij}. La longitud del camino más corto es la suma de los costes de los arcos que forman el camino. Matemáticamente, si $v_1, v_2 \ldots v_k$ es la secuencia de vértices que forman un camino:

$$\textit{Longitud de camino} = \sum_{i=1}^{k-1} c_{i,i+1}$$

y se pretende encontrar el camino de longitud o coste mínimo.

Otro problema relativo a los caminos entre dos vértices v_i, v_k, es encontrar aquel de menor número de arcos para ir de v_i a v_k. En definitiva, se trata de encontrar el camino de longitud mínima en un grafo no valorado. Este problema se resuelve con una *búsqueda en anchura* a partir del vértice de partida (se puede consultar el Apartado 17.4).

El algoritmo que se describe a continuación encuentra el camino mínimo desde un vértice origen, en un grafo con factor de peso positivo, se denomina algoritmo de Dijkstra.

18.3.1. Algoritmo de Dijkstra

Dado un grafo dirigido `G = (V,A)` valorado y con factores de peso no negativos, uno de los algoritmos más sencillos y eficientes para determinar el camino más corto desde un vértice al resto de los vértices del grafo es el algoritmo de Dijkstra. Éste es un ejemplo típico de algoritmo *ávido* (*voraz*) que selecciona en cada paso la solución más óptima en aras de resolver el problema.

Recordemos que se trabaja con un grafo valorado, donde cada arco (v_i, v_j) tiene asociado un coste c_{ij}. De tal forma que si v_1 es el vértice origen, y $v_1, v_2 \ldots v_k$ es la secuencia de vértices que forman el camino de v_1 a v_k, entonces $\sum_{i=1}^{k-1} c_{i,i+1}$ es la longitud del camino.

El algoritmo *voraz* de Dijkstra en cada paso selecciona un vértice `v`, cuya distancia es desconocida, entre los que tiene la distancia más corta al vértice origen `s`; entonces el camino más corto de `s` a `v` ya es conocido y *marca* el vértice `v` para no volver a investigarlo. Así sucesivamente se van

marcando nuevos vértices hasta que estén todos marcados; en ese momento es conocida la distancia mínima del origen `s` al resto de los vértices.

Estudiemos con más detalle cómo realizar esta estrategia *voraz*. En un conjunto `F` se tienen los vértices cuya distancia más corta respecto al origen `s` ya es conocida. En el vector `D` se almacena la distancia más corta (coste mínimo) desde el origen `s` a cada vértice del grafo. Inicialmente `F` contiene, únicamente, el origen `s` y los elementos de `D`, D_i el coste o peso de los arcos (s, v_i); si no hay arco de s a i se asume el *coste* ∞. En cada paso se agrega algún vértice `v` de los que todavía no están en `F`, esto es de `V-F`, que tenga el menor valor `D(v)`. Además se actualizan los valores `D(w)` para aquellos vértices `w` que todavía no están en `F`, $D(w) = D(v)+c_{v,w}$ cuando este nuevo valor de `D(w)` sea menor que el anterior `D(w)`. De esta forma la incorporación de `v` implica que hay un camino de `s` a `v` cuyo coste mínimo es `D(v)`.

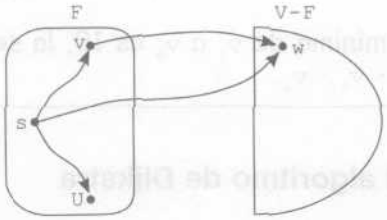


Figura 18.3. Al incorporar `v` la distancia de s a `w`, `D[w]` puede mejorar.

Para reconstruir el camino de *coste mínimo* que lleva de `s` a cada vértice `v` del grafo se almacena el último vértice que hizo el coste mínimo para cada uno de los vértices. Entonces, asociado con cada vértice resultan de interés dos datos: la mejor distancia o coste mínimo y el último vértice del camino con el que se minimizó el coste.

El Ejemplo 18.1 muestra los sucesivos pasos que se siguen en este algoritmo para encontrar los caminos mínimos desde el vértice origen, v_1, al resto de los vértices del grafo.

Ejemplo 18.1

En el grafo dirigido y con pesos no negativos, de la Figura 18.4, calcular el coste mínimo de los caminos desde el vértice 1 a los otros vértices del grafo, aplicando el algoritmo de Dijkstra.

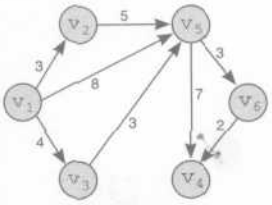


Figura 18.4. Grafo dirigido con factor de peso positivo.

La matriz de adyacencia con los pesos de los arcos, considerando ∞ el peso cuando no existe arco es:

$$C = \begin{vmatrix} \infty & 3 & 4 & \infty & 8 & \infty \\ \infty & \infty & \infty & \infty & 5 & \infty \\ \infty & \infty & \infty & \infty & 3 & \infty \\ \infty & \infty & \infty & 7 & \infty & 3 \\ \infty & \infty & \infty & 2 & \infty & \infty \end{vmatrix}$$

Los sucesivos valores del conjunto F (vértices *marcados*), vértice v incorporado y el vector distancia, D (..., D[2], D[3], ...), que se obtienen en cada paso se representan en forma tabular:

Paso	F	v	D[2]	D[3]	D[4]	D[5]	D[6]
Inicial	1		3	4		8	
1	1, 2	2	3	4		8	
2	1, 2, 3	3	3	4		7	
3	1, 2, 3, 5	5	3	4	14	7	10
4	1, 2, 3, 5, 6	6	3	4	12	7	10
5	1, 2, 3, 5, 6, 4	4	3	4	12	7	10

Ejemplo, el camino mínimo de v_1 a v_6 es 10, la secuencia de vértices que hacen el camino mínimo: $v_1 - v_3 - v_5 - v_6$.

18.3.2. Codificación del algoritmo de Dijkstra

La codificación que se realiza considera que el grafo está representado con una matriz de pesos o costes. Ésta contiene el coste de cada arco; si no hay arco entre dos vértices, la posición de la matriz contiene INFINITO (constante con un valor inalcanzable). Como cada vértice tiene asociado dos informaciones, último vértice en el camino mínimo y la distancia mínima, se define una estructura que los agrupa y una tabla de tantas posiciones como vértices. El conjunto F de vértices se representa de la forma más sencilla, un array de ceros o unos, de tal forma que si el vértice i se incluye en F, entonces F[i] se pone a 1. En esta codificación, se supone que el número de vértices es n, no pudiendo superar la constante N. Además, el vértice origen es el de índice 0.

```
#define INFINITO 1.0E+38
#define N 25

int F[N];
double matPesos[N][N];

typedef struct
{
   int ultimo;
   double distancia;
} EstadoVertice;

EstadoVertice D[N];

int minimo(int F[], EstadoVertice D[], int n);
void caminoMinimos(EstadoVertice D[], double matPesos[][N], int n);

int minimo(int F[], EstadoVertice D[], int n)
{
   double mx;
   int j, v;

   mx = INFINITO;
```

```
    for (j = 1; j < n; j++)
     if (!F[j] &&(mx >= D[j].distancia))
     {
       mx = D[j].distancia;
       v = j;
     }
    return v;
  }

void caminoMinimos(EstadoVertice D[], double matPesos[][N], int n)
{
  int v, w, i;
  int s = 0;                                        /* vértice origen */

            /* valores iniciales de tabla D. Se marca en F el origen */
  F[s] = 1;
  for (i = 1; i < n; i++)
  {
    F[i] = 0;
    D[i].distancia = matPesos[0][i];
    D[i].ultimo = 0;
  }
                                  /* Pasos para marcar los n-1 vértices */
  for (i = 1; i < n; i++)
  {
      v = minimo(F,D,n);   /* selecciona vértice no marcado de menor
                                                          distancia */
    F[v] = 1;
                          /* actualiza distancia de vértices no marcados */
    for (w = 1; w < n; w++)
     if (!F[w])
       if ((D[v].distancia+matPesos[v][w]) < D[w].distancia)
       {
          D[w].distancia = D[v].distancia + matPesos[v][w];
          D[w].ultimo = v;
       }
  }
}
```

El tiempo de ejecución del algoritmo permite deducir que tiene una complejidad cuadrática, $O(n^2)$, debido a los dos bucles anidados de orden n (número de vértices). Ahora bien, en el caso de que el número de arcos, a, fuera mucho menor que n^2, puede mejorarse su ejecución representando el grafo con listas de adyacencia, y organizando los vértices no marcados en una cola, de igual forma que se hizo en el algoritmo de longitud de camino en un grafo sin peso. Entonces puede obtenerse un tiempo $O(a*\log n)$.

18.4. CAMINOS MÍNIMOS PARA TODO VÉRTICE: ALGORITMO DE FLOYD

En algunas aplicaciones puede resultar interesante determinar el camino mínimo entre todos los pares de vértices de un grafo dirigido valorado. Considerando como vértice origen cada uno de los

vértices en el algoritmo de Dijkstra, el problema puede resolverse. Hay otra alternativa más elegante y más directa propuesta por Floyd que recibe el nombre de *algoritmo de Floyd*.

Sea G un grafo dirigido valorado, G = (V,A), en el que los vértices están numerados consecutivamente a partir del vértice inicial (vértice 1, o bien, vértice 0, según se desee). Se supone que el grafo está representado por la matriz de pesos, de tal forma que todo arco (v_i, v_j) tiene asociado un peso c_{ij}; si no existe arco $c_{ij} = \infty$. Además, cada elemento de la diagonal, c_{ii}, se hace igual a 0.

El objetivo del algoritmo de Floyd es encontrar una nueva matriz, D, de $n_x n$ elementos tal que cada elemento, D_{ij}, contenga el coste mínimo de los caminos de v_i a v_j.

El proceso que sigue este algoritmo tiene los mismos pasos que el algoritmo de Warshall para encontrar la matriz de caminos. Se generan, a nivel lógico, la secuencia de matrices D_o, D_1, D_2, ..., D_k, ..., D_n de tal forma que se parte de la matriz de pesos, D_o, ya conocida, y añadiendo en cada paso un nuevo vértice, a partir del vértice inicial, se obtiene recurrentemente una nueva matriz que recoge las posibles mejoras, en cuanto a la minimización buscada, producidas por la posibilidad de formación de nuevos caminos más cortos con el vértice incorporado. El significado de cada matriz:

$$D_0[i,j] = \begin{cases} c_{ij} & \text{coste (peso) del arco del vértice } i \text{ al vértice } j \\ \infty & \text{si no hay arco.} \end{cases}$$

$D_1[i,j]$ = mínimo($D_0[i,j]$, $D_0[i,1]+D_0[1,j]$).
El menor de los *costes* entre el *anterior* camino de i a j y la suma de los *costes* de caminos de i a 1 y de 1 a j.

$D_2[i,j]$ = mínimo($D_1[i,j]$, $D_1[i,2]+D_1[2,j]$).
El menor de los *costes* entre el *anterior* camino mínimo de i a j y la suma de los costes de caminos de i a 2 y 2 a j.

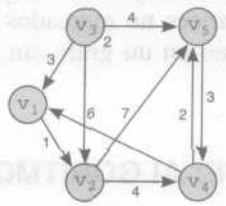
En cada paso se incorpora un nuevo vértice para determinar si hace el camino menor entre un par de vértices.

$D_k[i,j]$ = mínimo($D_{k-1}[i,j]$, $D_{k-1}[i,k]+D_{k-1}[k,j]$).
El menor de los *costes* entre el anterior camino mínimo de i a j y la suma de los *costes* de caminos de i a k y k a j.

De esta forma, recurrentemente se añade en cada paso un nuevo vértice para determinar si se consigue un nuevo camino mínimo, hasta llegar al último vértice y obtener la matriz D_n que es la matriz de caminos mínimos del grafo.

Ejemplo 18.2

En la Figura 18.5 se muestra un grafo dirigido con factor de peso y la correspondiente matriz de pesos. Aplicar el algoritmo de Floyd para obtener, en los sucesivos pasos, la matriz de caminos mínimos.



a)

$$C = \begin{vmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & \infty & 4 \\ 6 & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

b)

Figura 18.5. *a)* Grafo dirigido valorado de 5 vértices. *b)* Matriz de pesos del grafo.

El grafo consta de cinco vértices, por lo que se forman cinco matrices: D_1, D_2, ... D_5 que es la matriz de caminos mínimos. Cada una de ellas:

$$D_1 = \begin{vmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & \infty & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 1 ha cambiado $D_1(4,2)$ ya que $C(4,1)+C(1,2) < C(4,2)$.

$$D_2 = \begin{vmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 2 ha habido varios cambios, así $D_2(1,4)$: $D_1(1,2) + D_1(2,4) < D_1(1,4)$. También cambian $D_1(1,5)$, $D_1(2,5)$ y $D_1(3,4)$.

$$D_3 = \begin{vmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 3 no ha habido cambios; al vértice 3 no llega ningún arco.

$$D_4 = \begin{vmatrix} 0 & 1 & \infty & 5 & 7 \\ 10 & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ 9 & 10 & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice 4 cambian los elementos: $D_3(1,5)$, $D_3(2,1)$, $D_3(5,1)$ y $D_3(5,2)$.

$$D_5 = \begin{vmatrix} 0 & 1 & \infty & 5 & 7 \\ 10 & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ 9 & 10 & \infty & 3 & 0 \end{vmatrix}$$

La incorporación del vértice 5 no genera cambios.

La matriz D_5 es la matriz de caminos mínimos. Dado cualquier par de vértices, se puede conocer la longitud del camino más corto que hay entre ellos.

A tener en cuenta

Tanto el algoritmo de Dijkstra como el de Floyd permiten obtener los caminos mínimos, pero sólo se pueden aplicar en grafos valorados con *factor de peso* positivo, que son los más frecuentes.

18.4.1. Codificación del algoritmo de Floyd

Esta implementación añade una sentencia en la que se guarda el índice del último vértice que ha conseguido que el camino sea mínimo desde el vértice v_i al v_j, para cada par de vértices del grafo. Para ello se utiliza una matriz de vértices, en caso de que el camino entre dos vértices sea directo, el arco que los une tiene el valor clave –1. Además, se considera el grafo representado con la matriz de pesos o costes, de igual manera que se ha hecho en el algoritmo de Dijkstra. Se considera que los vértices están numerados de `0` a `n-1`.

```
#define INFINITO 1.0E+38
#define N 25
double costes[N][N];

void todoCaminoMinimo(double costes[][N], double D[][N],
                                          int traza[][N],int n)
{
  int i,j,k;

      /* matriz inicial es la de pesos.
         Clave -1, indica que camino mas corto es el arco. */

  for (i = 0; i < n; i++)
     for (j = 0; j < n; j++)
     {
        D[i][j] = costes[i][j];
        traza[i][j]= -1;
     }
        /* Camino mínimo de un vertice a si mismo: 0 */

  for (i = 0; i < n; i++) D[i][i] = 0;

  for (k = 0; k < n; k++)
     for (i = 0; i < n; i++)
       for (j = 0; j < n; j++)
         if ((D[i][k] + D[k][j]) < D[i][j]) /* nuevo mínimo */
          {
            D[i][j] = D[i][k] + D[k][j];
            traza[i][j]= k;
          }
}
```

Eficiencia del algoritmo

El tiempo de ejecución del algoritmo crece rápidamente a medida que crece el número de nodos del grafo. La complejidad es $O(n^3)$, donde `n` el número de vértices.

18.4.2. Recuperación de los vértices del camino

En la matriz `traza` se ha guardado el último vértice que origina que el camino del vértice `i` al `j` sea de coste mínimo. Para obtener la sucesión de vértices que determinan el camino hay que

«volver hacia atrás», y así obtener la secuencia de vértices que forman el camino. La mejor forma de *retroceder* hasta alcanzar el vértice de partida es con llamadas recursivas, teniendo en cuenta que la clave `-1`, en `traza[i][j]`, indica que el camino es un arco.

La función `recuperaCamino()` recibe el par de vértices y la matriz `traza`, deja de hacer llamadas recursivas cuando `traza[i][j]== -1`.

```
void recuperaCamino(int vi, int vj, int traza[][N])
{
   int vk;
   vk = traza[vi][vj];
   if (vk != -1)
   {
      recuperaCamino(vi,vk,traza);
      printf("%d - ",vk);
      recuperaCamino(vk,vj,traza);
   }
}
```

18.5. ÁRBOL DE EXPANSIÓN DE COSTE MÍNIMO

Los grafos no dirigidos se emplean para modelar relaciones simétricas entre *entes*, los vértices del grafo. Cualquier arista `(v,w)` de estos grafos es igual que `(w,v)`, por lo que las aristas en un grafo no dirigido se dicen formadas por un par no ordenado de vértices. Como consecuencia directa, la representación de un grafo no dirigido da lugar a matrices simétricas.

Una propiedad que normalmente interesa conocer de un grafo no dirigido es «si para todo par de vértices hay un camino que los une», en definitiva, si el grafo es *conexo*. A los grafos conexos también se les denomina *red conectada*. El problema del *árbol de expansión de coste mínimo* consiste en buscar un árbol que abarque todos los vértices del grafo, con suma de pesos de aristas mínimo.

Un *árbol*, en una red, es un subconjunto `G'` del grafo `G` que es conectado y sin ciclos. Los árboles tienen dos propiedades importantes:

1. Todo árbol de `n` vértices contiene exactamente `n-1` aristas.

2. Si se añade una arista a un árbol entonces resulta un ciclo.

Definición

Árbol de expansión de coste mínimo: es un subconjunto del grafo que abarca a todos los vértices que están conectados; la aristas formadas por esos vértices tienen una suma de costes mínimos.

Buscar un árbol de expansión de un grafo, en una red, es también una forma de averiguar si está conectado. Todos los vértices del grafo tienen que estar en el árbol de expansión para que sea grafo conectado. La Figura 18.6 muestra un grafo no dirigido y su árbol de expansión, el grafo es una *red conectada*.

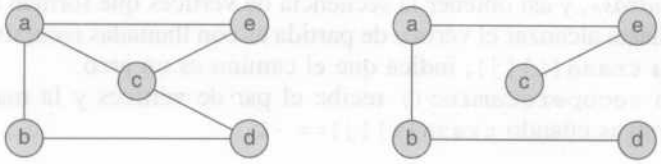


Figura 18.6. Grafo no dirigido y su árbol de expansión.

Los árboles de expansión se aplican en el diseño de redes de comunicación en todas sus vertientes. Por un momento pensemos en el siguiente problema: *De los pueblos que forman parte del Ayuntamiento de Siguenza, cada par de ellos está conectado por un camino vecinal; el peso de un camino directo entre dos pueblos viene dado por la distancia en kms. Se quiere construir una red de carriles bici de coste mínimo, de tal forma que cada par de pueblos esté comunicado a un coste mínimo.*

El planteamiento del problema: dado un grafo no dirigido ponderado y conexo, encontrar el subconjunto del grafo compuesto por todos los vértices, con conexión entre cada par de vértices, sin ciclos y que cumpla que la suma de los pesos de las aristas sea mínimo. Simplemente, encontrar el *árbol de expansión de coste mínimo.*

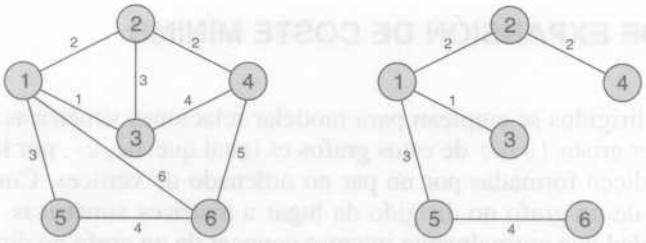


Figura 18.7. Grafo valorado y su árbol de expansión mínimo.

18.5.1. Algoritmo de Prim

El algoritmo de Prim encuentra el árbol de expansión mínimo de un grafo no dirigido; para ello realiza sucesivos pasos siguiendo la metodología clásica de los algoritmos voraces de hacer en cada paso «lo mejor que se pueda hacer». En este problema, *lo mejor* es incorporar al árbol una nueva arista del grafo de menor longitud.

El punto de partida es un grafo `G = (V,A)` no dirigido conectado (una red) y `c(i,j)` es el peso o coste asociado al arco (v_i, v_j). Supóngase que los vértices están numerados: `V = {1,2,3,...,n}`. Se utiliza el conjunto `W` para contener los vértices que han pasado a formar parte del árbol de expansión.

El algoritmo arranca asignando un vértice inicial al conjunto `W`, por ejemplo el vértice `1`; `W = {1}`. A partir del vértice inicial el árbol de expansión crece, añadiendo en cada pasada, al conjunto `W`, otro vértice `z` de `V` todavía no incluido en `W` (perteneciente a `V-W`) tal que si `u` es un vértice de `W`, la arista `(u,z)` es la *más corta*, la de *menor coste*. El proceso termina cuando todos los vértices del grafo están en `W`, y en ese momento se ha formado el árbol de expansión con todos los vértices y, además, es mínimo porque en cada pasada se ha añadido la menor arista.

La Figura 18.8 muestra un grafo conectado, formado por 10 vértices y numerados a partir del vértice `1`. Inicialmente se incorpora al conjunto `W` el vértice `1`, `W={1}`. De las aristas que forma parte el vértice `1` la de menor coste es `(1,2)`, por lo que se añade a `W` el vértice `2`, `W = {1, 2}`. La siguiente arista mínima formada por cualquier vértice de `W` y los vértices no incorporados es `(2,4)`, y se añade a `W` el vértice `4`, `W={1, 2, 4}`. La siguiente pasada añade el vértice `5` al ser la arista

(4,5) la mínima; una nueva pasada incorpora el vértice 8 ya que la arista (5,8) es la de menor coste, W={1, 2, 4, 5, 8}. A continuación se incorpora el vértice 7 ya que la arista (5,7) es la de menor coste, W={1, 2, 4, 5, 8, 7}; le siguen los vértices 9, 10, 3 y termina el algoritmo con el vértice 6, W={1, 2, 4, 5, 8, 7, 9, 10, 3, 6}. La Figura 18.9 muestra el árbol de expansión de coste mínimo que se ha formado.

Observación

En los sucesivos pasos del algoritmo de Prim, el conjunto de nodos que forma W constituyen una componente conexa sin ciclos ya que las aristas elegidas tienen un vértice en W y el otro en los restantes vértices, V-W.

Otra forma de expresar el algoritmo de Prim: partiendo de un vértice inicial u se toma la arista menor (u,v) que no forme un ciclo, y así iterativamente se toman nuevas aristas de menor peso (z,w), *sin formar ciclos y formando en todo momento una componente conexa.*

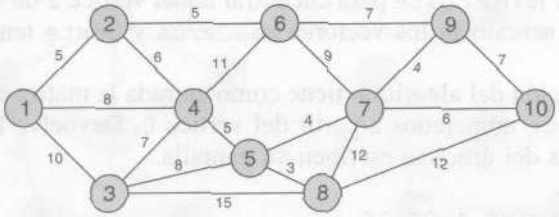


Figura 18.8. Grafo valorado conexo: representa una red telefónica.

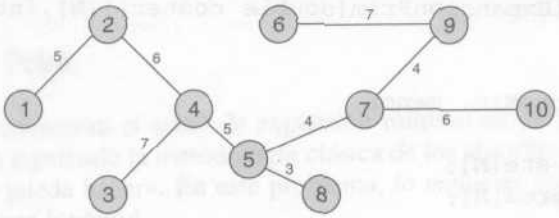


Figura 18.9. Árbol de expansión del grafo de la Figura 18.8.

El algoritmo puede expresarse:

```
arbolExpansion Prim(G, T)
{ G : grafo
  T : conjunto de arcos del árbol de coste mínimo}
var locales
  W: < conjunto de vértices >;
  u,w: vértices;
Inicio
  T <- {}
  V <- {1..n}
  u <- 1
  W <- {u}
  mientras W<> V hacer
```

```
        <Encontrar v de V-W tal que (u,v) sea mínimo>
        W <- W+{v}
        T <- T+{(u,v)}
    fin_mientras
fin_arbolExpansion
```

18.5.2. Codificación del algoritmo de Prim

La codificación parte de que es conocida la matriz de pesos que representa al grafo. Para el problema de encontrar, en cada pasada, la arista de menor *peso* que une un vértice de `W` y el otro de `V-W` se utilizan dos vectores:

`masCerca`, tal que `masCerca[i]` contiene el vértice de `W` de menor *coste* respecto el vértice `i` de `V-W`.

`coste`, tal que `coste[i]` contiene el peso de la arista `(i, masCerca[i])`.

En cada paso se revisa `coste` para encontrar aquel vértice `z` de `V-W` que sea el más cercano a `W`. A continuación se actualizan los vectores `masCerca` y `coste` teniendo en cuenta que `z` ha sido añadido a `W`.

La implementación del algoritmo tiene como entrada la matriz de pesos de un grafo no dirigido conexo de `n` vértices, numerados a partir del vértice `0`. Devuelve la suma de las aristas del árbol mínimo. Las aristas del árbol se escriben en pantalla.

```
#define INFINITO 1.0E+38
#define N 15

double arbolExpansionPrim(double costes[][N],int n)
{
   int i,j;
   double longMin, menor;
   int z;
   double coste[N];
   int masCerca[N];
   int W[N];

   for (i = 0; i < n; i++) W[i] = 0;                /* conjunto vacío */
   longMin = 0.0;
         /* vértice inicial del árbol: vértice 0 */
   W[0] = 1;
         /* inicialmente, el coste de cada vértice V-[0] es la arista (0,i) */

   for (i = 1; i < n; i++)
   {
      coste[i] = costes[0][i];
      masCerca[i] = 0;
   }

   for (i = 1; i < n; i++)
   { /* busca vértice z de V-W mas cercano, de menor longitud
        de arista, a algún vértice de W */
```

```
        menor = coste[1];
        z = 1;
        for (j = 2; j < n; j++)
        if (coste[j] < menor)
         {
           menor = coste[j];
           z = j;
         }

        longMin += menor;
        printf("\t %d -> %d\n",masCerca[z],z);
           /* vértice z es añadido al conjunto W */
        W[z] = 1;
        coste[z] = INFINITO;
           /* ajuste de coste[] del resto de vértices comparando
              con matriz de pesos, debido a que se ha añadido
              z a W. */
        for (j = 1; j < n; j++)
          if ((costes[z][j] < coste[j]) && !W[j])
           {
            coste[j] = costes[z][j];
            masCerca[j] = z;
           }
    }
    return longMin;
}
```

18.5.3. Algoritmo de Kruskal

Kruskal propone otra estrategia para encontrar el árbol de expansión de coste mínimo de un grafo, `G = (V,A)`, no dirigido y valorado que es conexo. El árbol de expansión se inicia con los mismos vértices `V` del grafo `G` pero sin aristas; se puede afirmar que cada vértice es una componente conexa en sí mismo. El algoritmo construye componentes conexas cada vez mayores examinando las aristas del grafo en orden creciente del *peso*. Si la arista conecta dos vértices que se encuentran en dos componentes conexas distintas, entonces se añade la arista al árbol `T`. En el proceso, las aristas que conectan dos vértices pertenecientes a la misma componente se descartan, ya que darían lugar a un ciclo si se le añadiera al árbol de expansión al estar en la misma componente. Cuando todos los vértices están en un sólo componente, `T`, éste es el árbol de expansión de coste mínimo del grafo `G`.

El algoritmo de Kruskal asegura que el árbol no tiene ciclos ya que una arista se añade si está en dos componentes distintas; además es de coste mínimo ya que examina las aristas en orden creciente de sus pesos.

La Figura 18.7 muestra un grafo y su árbol de expansión. La lista de aristas en orden creciente de sus pesos:

```
{(1,3), (1,2), (2,4), (1,5), (2,3), (3,4), (5,6), (4,5), (1,6)}
```

La primera arista que se toma es `(1,3)`, Figura 18.10*a*, a continuación las aristas `(1,2)`, `(2,4)` y `(1,5)`, en las que se cumple que los vértices forman parte de dos componentes

conexas diferentes, Figura 18.10*b*. La siguiente arista es `(2,3)`, que como puede observarse en la Figura 18.10*b* sus vértices están en la misma componente, por tanto, se descarta. Le sigue la arista `(3,4)`, que por el mismo motivo es descartada. En la siguiente arista, `(5,6)`, sus vértices están en dos componentes diferentes, pasa a formar parte del árbol y el algoritmo termina ya que se han alcanzado todos los vértices.

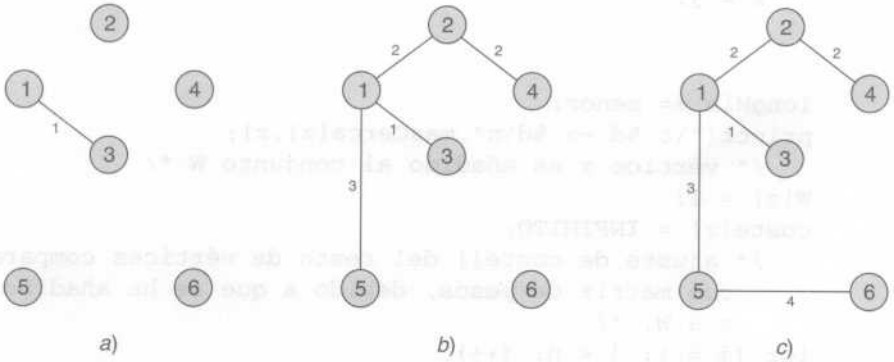


Figura 18.10. Formación del árbol de expansión del grafo de la Figura 18.7.

18.5.4. Codificación del algoritmo de Kruscal

La implementación de este algoritmo exige, en primer lugar, ordenar las aristas en orden creciente de sus pesos, para así poder examinarlas en orden creciente. Las aristas se representan con una estructura, `arco`, en la que se agrupan los dos vértices y el peso asociado.

Cada vez que se incorpora una arista al árbol hay que unir las dos componentes conexas en que se encuentran; la forma de hacerlo es con la operación unión de conjuntos. Por consiguiente, las componentes conexas se pueden considerar que son conjuntos de vértices, que a su vez son representados con un array de `n` elementos (los vértices se consideran numerados a partir de vértice `0`); al iniciar el proceso con `n` componentes conexas se definen `n` conjuntos, según se vayan uniendo componentes irán disminuyendo el número de componentes hasta quedar una única componente.

```
#define N 15

typedef struct
{
  int u;
  int v;
  double peso;
} Arco;

double arbolExpansionKruskal(double costes[][N],int n, Arco T[])
{
  Arco arcos[N*N];
  Arco actual;
  int componentes[N][N];
  int cu,cv,cn, numArcos,i,j,k;
```

```
    double longMin;
    k = numArcos = 0;
    longMin = 0.0;

    numArcos = inicial(costes,arcos,n); /* obtiene los arcos del
                                           grafo en orden creciente de peso */
                /* n componentes iniciales, una componente por vértice */
    for (i = 0; i < n; i++)
    {
      for (j = 0; j < n; j++)
       componentes[i][j] = 0;
      componentes[i][i] = 1;
    }

    cn = n;
    while (cn > 1 &&(k < numArcos) )
    {
      actual = arcos[k++];
      cu = numComponente(actual.u,componentes,n);
                /* componente conexa en que se encuentra cada vértice */
      cv = numComponente(actual.v,componentes,n);

      if (cu != cv)
      {
        longMin += actual.peso;
        printf("\t %d -> %d\n",actual.u,actual.v);
        combinaComponente(cu,cv,componentes,n);
        T[n - cn] = actual; /* guarda arcos que forman el arbol */
       cn--;
      }
    }
    return longMin;
  }


  int numComponente(int v, int componentes[][N], int n)
  {
    int sw = 0;
    int i = 0;

    while (i < n && !sw)
      if (componentes[i][v])
        sw = 1;
      else
        i++;
    return (sw ? i : -1);
  }
  void combinaComponente(int cu, int cv, int componentes[][N],int n)
```

```
{
  /* Une componente cu con componente cv. La componente resultante queda
     en la posicion cu; la componente cv queda vacia */
  int j;


  for (j = 0; j < n; j++)
    if (componentes[cv][j] == 1)
    {
     componentes[cu][j] = 1;
     componentes[cv][j] = 0;
    }
}
```

18.6. GRAFOS PARA REPRESENTAR REDES DE FLUJO

En la vida diaria aparece con mucha frecuencia el concepto abstracto de *red* aplicado en muchos sectores de la economía y, en general, de la sociedad: *red eléctrica, red de computadoras, red comercial* ..., ***red de redes;*** éstos son algunos de los ejemplos cotidianos. Muchas otras situaciones se pueden modelar mediante redes que tienen nodos (vértices) por los que entra *flujo* procedente de otros nodos, y a su vez sale *flujo* en dirección a otros nodos. El *flujo* se mueve por las aristas que unen un par de nodos; dichas aristas están caracterizadas por el máximo *flujo* que pueden transportar, que es la capacidad de la arista.

La idea primaria de flujo se obtiene del significado popular: *forma de enviar objetos de un lugar a otro*. Así entre los centros de producción y los centros de distribución hay un flujo de mercancías. Entre los lugares de residencia y los centros de trabajo se produce un flujo de personas. Un sistema de tuberías para transporte de agua, cada arco es un tubo y el «peso» representa la capacidad de la tubería en *litros/minuto* y los nodos son puntos de unión de los diversos tubos.

Desde esta idea intuitiva de *flujo* se pasa a la formulación matemática de *flujo*: se llama flujo a la función `Fij` definida en el conjunto de arcos `A`, que verifica las propiedades:

1. $\mathtt{Fij} \geq 0 \quad \forall(\mathtt{i},\mathtt{j}) \in \mathtt{A}$

2. $\sum_{j \in \alpha_i} \mathtt{Fij} - \sum_{j \in \beta_i} \mathtt{Fji} = 0 \quad \mathtt{i} \in \mathtt{V}, \mathtt{i} \neq \mathtt{S}, \mathtt{i} \neq \mathtt{T}$

   $\alpha_i \equiv$ conjunto vértices conectados con el vértice `i` mediante aristas que salen de `i`.
   $\beta_i \equiv$ conjunto vértices conectados con el vértice `i` mediante aristas que entran en `i`.
   `S` $\equiv$ nodo fuente o inicial del flujo.
   `T` $\equiv$ nodo sumidero o destino del flujo.

3. $\mathtt{Fij} \leq \mathtt{Uij} \quad \forall(\mathtt{i},\mathtt{j}) \in \mathtt{A}$

La segunda propiedad impone el principio de conservación de la cantidad total de flujo (ley de Kirchoff). Las cotas inferior y superior de los valores del flujo están reflejadas en la primera propiedad (flujo mayor o igual que cero) y en la tercera (el flujo no puede superar la capacidad, U_{ij}, de la arista).

18.6.1. Planteamiento del problema

Sobre una red de flujos pueden plantearse diversos problemas. Así, puede quererse *maximizar* la cantidad transportada desde un vértice origen a un vértice destino. Otro problema típico es conocer el modo más económico de enviar una determinada cantidad de objetos desde un origen a un destino en un sistema de distribución. Un grafo con factor de *peso* para representar una red de flujo *es una estructura ideal para modelar estas situaciones.*

Ejemplo 18.2

Planificación de rutas alternativas para circulación en horas punta desde una ciudad dormitorio al centro de la ciudad, donde se ubican las oficinas.

Supóngase un punto de origen con fuerte tráfico, por ejemplo, una ciudad dormitorio, y otro punto que representa el centro, la zona comercial de la gran ciudad. Entre ambos puntos se pueden establecer diversas rutas de entrada, cada una de ellas con una capacidad de tráfico máxima, dato que es conocido y expresado en miles de vehículos por hora. La Figura 18.11 representa, de manera muy simple, la red compuesta de rutas que parten del vértice 1, S, y tienen como destino el vértice T.

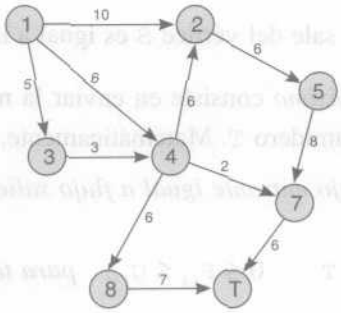


Figura 18.11. Modelo de red viaria de tráfico.

Un problema que se puede plantear es encontrar cuál es el *flujo máximo* de vehículos que pueden desplazarse desde el punto origen al destino en la hora punta, y conocer cuál será el índice de utilización de cada una de las rutas de entrada. Esta característica conduce a resolver un problema de optimización de la red viaria.

Ejemplo 18.3

Aplicación práctica: Red de tuberías para llevar el agua a una población.

Considérese un sistema de tuberías de transporte de agua; cada arco representa una cañería y el *factor de peso* es la capacidad en litros/minuto. Los vértices representan nodos en los que la cañería se une están unidas y el agua es transportada de un nodo a otro. Dos nodos especiales, S y T son la fuente de emisión del agua y el punto de consumo respectivamente.

La Figura 18.12 muestra una sencilla red de transporte de agua, los *pesos* de las aristas representan la capacidad de cada tubería en l/min.

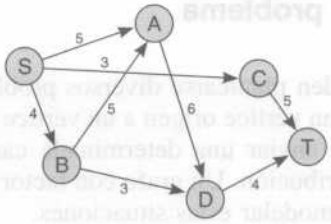


Figura 18.12. Red de flujos.

El objetivo es *maximizar* la cantidad de agua que fluye desde el nodo fuente hasta el punto de consumo, que es el nodo *sumidero*.

18.6.2. Formulación matemática

Sea la red `G = (V, A)` con un único vértice fuente `S` y un único sumidero `T`, donde asociada a cada arco `(i, j)` hay una magnitud U_{ij} que representa la capacidad máxima del arco.

$$\sum_{x \in V} F(S, X) = Vf = \sum_{x \in V} F(X, T)$$

La cantidad de *flujo* que sale del vértice `S` es igual a la cantidad que entra en el *sumidero* `T`, y esta cantidad es *Vf*.

El problema del *flujo máximo* consiste en enviar la mayor cantidad posible de flujo desde el vértice fuente `S` al vertice sumidero `T`. Matemáticamente, maximizar el flujo que llega a `T`, $\sum_{i \neq t} F_{it}$ *sujeto a las condiciones (flujo entrante igual a flujo saliente):*

$$\sum_{j} F_{ij} - \sum_{j} F_{ij} = 0;\ i \neq T \qquad 0 \leq F_{ij} \leq U_{ij},\ \textit{para toda arista}\ (i, j) \in A$$

La función *objetivo* es el sumatorio de los flujos que llegan al vértice sumidero `T`, que es, precisamente, la cantidad que se desea hacer máxima. Esta función de flujo es el denominado *flujo óptimo*.

En la Figura 18.12 se puede observar que del nodo *fuente* `S`, todas las aristas son salientes, el máximo flujo que puede emerger es la suma de capacidades de las aristas, `4+3+5`. Al nodo sumidero `T` llegan dos aristas, por consiguiente el máximo flujo que puede llegar es `5+4`. La cuestión es cómo distribuir el flujo por los nodos intermedios para que llegue el máximo flujo a `T`.

18.6.3. Algoritmo del aumento del flujo: algoritmo de Ford-Fulkerson

Este algoritmo es uno de los más sencillos y a la vez eficientes para determinar el *flujo máximo* en una red, partiendo de un nodo fuente `S` y teniendo como destino el nodo *sumidero* `T`.

La idea básica de algoritmo es la siguiente: se parte de una función de flujo, *flujo cero*, e iterativamente se van obteniendo mejoras del flujo. La mejora se da en la medida que el flujo desde `S` hasta `T` aumenta, teniendo en cuenta las condiciones que ha de cumplir la función de flujo, que en lenguaje natural:

- Flujo que entra a un nodo ha de ser igual al flujo que sale (principio de conservación).
- En todo momento el flujo no puede superar la capacidad de la arista.

Las aristas de la red pueden clasificarse en tres categorías:

- *No modificable*: arista cuyo flujo no puede aumentarse ni disminuirse, por tener capacidad cero o tener un *coste* prohibitivo.
- *Incrementable*: arista cuyo flujo puede aumentarse; transporta un flujo inferior a su capacidad.
- *Reducible*: arista cuyo flujo puede ser reducido.

Con estas categorías podemos establecer las siguientes mejoras de la función de flujo desde el nodo S al nodo T:

1.ª *Forma de mejora*: Encontrar un camino P del vértice *fuente* S al *sumidero* T, con todas las aristas *incrementables*, tal que el flujo a través de cada arista del camino es menor que la capacidad:

$$F(v_i, v_j) < U_{i,j} \quad \forall (i,j) \in P$$

Entonces el flujo se puede mejorar en las unidades:

$$\text{mínimo } \{(U_{i,j} - F_{i,j}) \quad \forall (i,j) \in P\}$$

2.ª *Forma de mejora*: Encontrar un camino P' del nodo *sumidero* T al nodo *fuente* S formado por aristas *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:*

$$\text{mínimo } \{F_{i,j} \ \forall (i,j) \in P'\}$$

3.ª *Forma de mejora*: Existe un camino P_1 desde S hasta algún nodo v con aristas *incrementables*, un camino P_2 desde un nodo w hasta el *sumidero* T con aristas *incrementables*, y un camino P_3 desde w a v con flujo *reducible*. Entonces el flujo a lo largo del camino w-v puede ser reducido y el flujo desde S a T puede ser incrementado en las unidades de flujo igual al mínimo de estas dos cantidades:

$$\textit{mínimo } \{(U_{i,j} - F_{i,j}), \ \forall (i,j) \in P1, \textit{ o bien } \in P2\}$$

$$\text{mínimo } \{ F_{i,j} \ \forall (i,j) \in P_3\}$$

El algoritmo lo que hace es aplicar estas formas de mejora de aumento de flujo hasta que ya no es posible aumentar el flujo al nodo *sumidero*.

En el contexto de redes de flujo se utiliza el concepto de *cadena*. Una *cadena* que une los vértices v_i, v_j, es una sucesión de arcos (aristas) con origen en el vértice i y final en el vértice j, tal que dos arcos sucesivos tienen un vértice común, aunque no necesariamente el vértice final de un arco es el inicial del siguiente.

Definición

Una cadena, o *semicamino*, desde el nodo v al nodo w es la secuencia de nodos $v = x_1, x_2, \ldots, x_n = w$ tal que para todo $1 \leqslant i \leqslant n$, se cumple que (x_{i-1}, x_i) o (x_i, x_{i-1}) es una arista.

De inmediato se puede afirmar que todo camino `P` de `v` a `w` es una cadena, aunque no toda cadena es un camino. En la red de la Figura 18.13 pueden encontrarse diversas cadenas de `S` a `T`; una de ellas: `S - A - B - D - T`.

A cualquier cadena de la red de `S` a `T` a la que se le pueda aplicar una de las formas de mejora de flujo será una *cadena de aumento de flujo*.

A tener en cuenta

El objetivo del algoritmo de aumento de flujo es encontrar *cadenas de aumento de flujo* en la red y aumentar el flujo en la cantidad determinada por cada una de las distintas formas de *mejora*. La solución óptima se encuentra cuando ya no hay más *cadenas de aumento de flujo*.

18.6.4. Ejemplo de mejora de flujo

Considérese la red de fluido de la Figura 18.13, el *factor de peso* de las aristas representa la capacidad, en l/min, de las tuberías que unen los nodos.

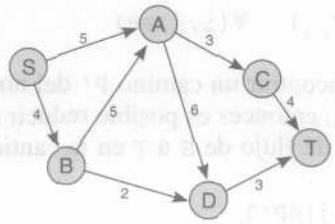


Figura 18.13. Red de flujos con capacidad en cada arista.

En la red se pueden encontrar estos caminos desde el nodo fuente, `S`, al nodo *sumidero*, `T`, P_1: `S-A-C-T`, P_2: `S-B-D-T` y P_3: `S-A-D-T` todos con arcos *incrementables*, a los que se les puede aplicar la *primera forma de mejora*. Por el camino P_1 el flujo puede ser mejorado en el `minimo{(5-0), (3-0), (4-0)}`, esto es, en 3 unidades; por el camino P_2 el flujo es mejorable en el `minimo{(4-0), (2-0), (3-0)}`, que son 2 unidades. Antes de obtener la mejora en el flujo del camino P_3 es preciso observar la Figura 18.14; cada arista tiene asociada la capacidad y el flujo actual.

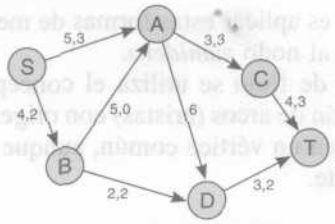


Figura 18.14. Red de flujos con la capacidad y el flujo actual en cada arista.

Entonces, por el camino P_3 el flujo es mejorable en: `minimo{(5-3), (6-0), (3-2)}`, que es igual a `1` unidad de flujo. Con esta nueva mejora de flujo al *sumidero* `T` llegan `6` unidades de flujo. La Figura 18.15 muestra la función de flujo óptimo después de las sucesivas *mejoras* aplicadas sobre los caminos de `S` a `T`.

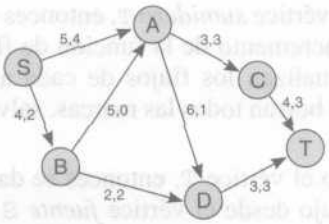


Figura 18.15. Flujo óptimo en la red, con la capacidad y el flujo en cada arista.

No sólo se pueden obtener mejoras de flujo con los caminos formados por arcos incrementables; en la Figura 18.16a se observa una sencilla red con la función de flujo después de la mejora obtenida con aristas incrementables. Ahora analicemos la secuencia de nodos `S-Y-X-T`; está formada por aristas hacia delante, `(S,Y)`, `(X,T)`; y la arista `(X,Y)`, que tiene sentido contrario, arista *hacia atrás*, ya que no une el vértice final, `Y`, con el siguiente. Por consiguiente, no es un camino de `S` a `T` sino que forma lo que se ha denominado una *cadena*, que está formada por aristas incrementables y por aristas reducibles `(X,Y)`. El aumento de flujo se obtiene aplicando la *tercera forma de mejora, mínimo de*:

`mínimo {(4-0), (3-0)}` se corresponde con las aristas *incrementables*.
`mínimo{(4)}` se corresponde con la arista reducible `(X,Y)`.

Este mínimo es igual a 3 unidades. En la Figura 18.16*b* se muestra la función de flujo después de la actualización debida a esta mejora.

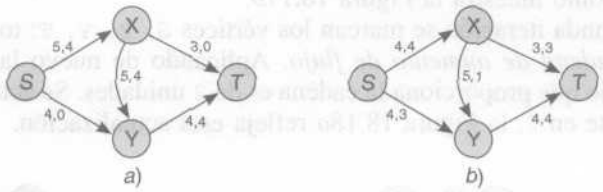


Figura 18.16. Red con mejora de flujo en un camino y en una cadena.

18.6.5. Pasos que sigue el algoritmo

Los pasos que sigue el algoritmo de aumento de flujo persiguen encontrar *cadenas de aumento de flujo* son:

1. Se parte de un flujo inicial, $F_{ij} = 0$ para todas las aristas de la red. Determina aristas que son *incrementables* y aristas *reducibles*. Marca el vértice fuente `S`.
2. Repetir hasta que sea marcado el vértice *sumidero* `T`, o bien no sea posible marcar más vertices:

   2.1. Si un vértice `i` está marcado, existe la arista `(i,j)` y es tal que `(i,j)` es una arista *incrementable*, entonces marcar vértice `j`.

   2.2. Si un vértice `i` está marcado, existe la arista `(j,i)` y es tal que `(j,i)` es una arista *reducible*, entonces marcar el vértice `j`.

3. Si ha sido marcado el vértice *sumidero* `T`, entonces se ha obtenido una *cadena de aumento de flujo*. Se procede al incremento de la función de flujo en la cantidad de aumento permitido por la cadena. Se actualizan los flujos de cada arista con las unidades de aumento de la cadena. Por último, se borran todas las marcas, salvo la del vértice `S` y se repiten las acciones a partir del paso 2.
4. Si no ha sido marcado el vértice `T`, entonces se da por finalizado el algoritmo ya que no es posible enviar más flujo desde el vértice *fuente* `S` hasta el vértice *sumidero* `T`.

Ejemplo 18.3

Dada la red de flujos de la Figura 18.17*a*, en la que los pesos de las aristas representan la capacidad y el flujo actual, aplicar los pasos que propone el algoritmo de Ford-Fulkerson para encontrar el flujo máximo.

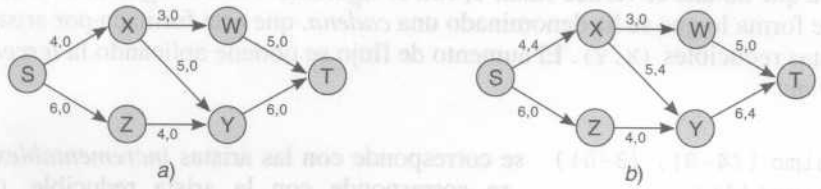


Figura 18.17. *a*) Red de flujos. *b*) La red después de la primera mejora de flujo.

En un primer paso se marcan los vértices `S, X, Y, T`; todas las aristas son *incrementables*: es una *cadena de aumento de flujo*. El incremento de flujo es igual a `minimo{(C_ij- F_ij) ∀ (i,j) ∈ cadena }`, en esta cadena el aumento es `4`. A continuación se actualizan los flujos de cada arista con esa magnitud como muestra la Figura 18.17*b*.

En una segunda iteración se marcan los vértices `S, Z, Y, T`; todas las aristas son *incrementables*, es otra *cadena de aumento de flujo*. Aplicando de nuevo la forma de mejora de flujo, el aumento de flujo que proporciona la cadena es de `2` unidades. Se actualizan los flujos de cada arista aumentando este en `2`, la Figura 18.18*a* refleja esta actualización.

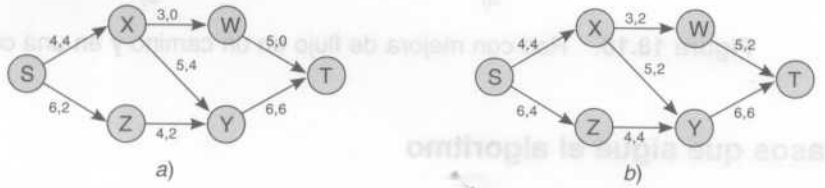


Figura 18.18. *a*) La red después de una segunda mejora. *b*) Flujo óptimo en la red.

En una tercera iteración son marcados los vértices `S, Z, Y, X, W, T`; las aristas `(S,Z)`, `(Z,Y)` son *incrementables*, la arista `(X,Y)` es *reducible* y las aristas `(X,W)`, `(W,T)` *incrementables*. Es una tercera *cadena de aumento de flujo*. El incremento de flujo de la cadena es el mínimo de estas dos magnitudes:

`mínimo {(6-2), (4-2), (3-0),(5-0)}` de las aristas *incrementables*.
`mínimo{(4)}` se corresponde con la arista *reducible* `(X,Y)`.

Entonces el nuevo aumento de flujo es 2, una vez actualizadas las aristas, la función de flujo queda como muestra la Figura 18.18b.

En esta red ya no es posible encontrar mas *cadenas de aumento de flujo*. Por tanto, el flujo máximo que llega a T es de 8 unidades. El análisis de la solución proporciona información muy valiosa para valorar en qué aristas conviene aumentar la capacidad para que aumente más el flujo que llega a T, y en qué aristas se puede disminuir la capacidad sin alterar el flujo que llega al *sumidero*.

18.6.6. Algoritmo Ford-Fulkerson en seudocódigo

En primer lugar, proponen unos tipos de datos que permitan implementar fácilmente los pasos que sigue el algoritmo. Para *marcar* un nodo, una vez que éste ha sido colocado en una cadena, se utiliza el vector `encadena[]`, de tal forma que `encadena[nodo]== 1` indica que el nodo está en la cadena y está en proceso de formación. El vector `finvia[]` se utiliza para conocer cuál es el último nodo de la *cadena en formación, de esa forma se tiene el* `nodo` *a partir del cual seguir la formación de la cadena; así* si `finvia[nodo]== 1` es que nodo está al final de la cadena en formación.

Para obtener correctamente la secuencia de nodos que forman la cadena, el vector `precede[]` permite obtener dicha secuencia: los índices del vector son los nodos (se supone numerados) y `precede[nodo`$_i$`]` contiene el nodo precedente de nodo$_i$ en la cadena de la que forma parte.

Para determinar si las aristas son hacia adelante, hacia T, el vector `adelante[]` es tal que `adelante[nodo]` es 1 (*true*) si la arista tiene ese sentido, en definitiva si la arista es `(precede[nodo`$_i$`], nodo`$_i$`)`. A medida que la cadena se va formando, las unidades de flujo acumuladas hasta un nodo `i` se guardan en flujos[`nodo`$_i$]; se ha de tener en cuenta que ese flujo ha de ser el mínimo entre el flujo alcanzado, en el nodo anterior de la cadena, y la diferencia entre capacidad y flujo de la arista si ésta es *incrementable*, o bien el flujo actual si la arista es *reducible*.

La capacidad de una arista y su flujo actual se encuentran en las posiciones correspondientes de las matrices `C` y `F`, `C[i][j]` y `F[i][j]`.

Seudocódigo del algoritmo de aumento de flujo

```
Inicio {S, T son los nodos origen y sumidero}

  Repetir

     Para todos los nodos:
       finvia[nodo] ← false
       encadena[nodo] ← false
     fin

     finvia[S]← true
     encadena[S]← true

     mientras (no encadena[T]) y (exista nd,,finvia[nd]) hacer
       finvia[nd] ← false
       mientras (exista nodo i tal que (no encadena[i]) y
              (F[nd,i]< C[nd,i])y adyacente(nd,i)) hacer

            {i nuevo nodo de la cadena, arista (nd,i) es incrementable}
            encadena[i], finvia[i], adelante[i] ← true
            precede[i] ← nd
```

```
                    {guarda el mínimo de las arista de la cadena }
                si flujos[nd]< (C[nd,i]- F[nd,i]) entonces
                    flujos[i] ← Flujos[nd]
                sino
                    flujos[i] ← C [nd,i] - F[nd,i]
                fin_si
            fin_mientras

                {A continuación busca aristas hacia atrás reducibles }
            mientras (exista nodo i tal que (no Encadena[i] y
                    F[i,nd]> 0 y adyacente(nd,i))) hacer
        {i nuevo nodo de la cadena, arista (i,nd) es reducible}
                encadena[i], finvia[i] ← true
                adelante[i] ← false
                precede[i] ← nd
                    {guarda el mínimo de las arista de la cadena }
                si flujos[nd]< F[i,nd] entonces
                    flujos[i] ← flujos[nd]
                sino
                    flujos[i] ← F[i,nd]
                fin_si
            fin_mientras
        fin_mientras                        {bucle de búsqueda de cadena}

          si encadena[T] entonces
        {Aumentar el flujo de aristas de la cadena en la cantidad flujo[T]}
          sino
           <El flujo ya es máximo>
          fin_si

      Hasta no encadena[T]
    Fin del algoritmo
```

RESUMEN

Si G es un *grafo dirigido sin ciclos*, entonces un orden topológico de G es un listado secuencial de todos los vértices de G; tal que para todos los vértices v, w ∈ G si existe una arista desde v a w entonces v precede a w en el listado secuencial. El término *acíclico* se utiliza con frecuencia para representar que un grafo no tiene ciclos.

La ordenación topológica es un recorrido solamente aplicable a grafos dirigidos acíclicos, que cumple con la propiedad de que un vértice sólo se visita si han sido visitados todos sus predecesores dentro del grafo. En un orden topológico, cada vértice debe aparecer antes que todos los vértices que son sus sucesores en el *grafo dirigido*.

Dado un grafo G, una de las operaciones clave es la búsqueda de caminos mínimos que sean lo más *cortos* posible, donde la longitud de un camino se define como la suma de los *costes* de las aristas que lo componen. Uno de los problemas que se plantean en un grafo cuyas aristas tienen asociado una magni-

tud, el *coste* o *factor de peso*, es calcular el coste del camino mínimo desde un vértice al resto de vértices. Una de las soluciones más eficiente a este problema es el *algoritmo de Dijkstra*: es un algoritmo *voraz* que permite obtener la distancia más corta desde un vértice origen a los demás vértices en un grafo cuyas aristas tengan *coste positivo*; además, se obtiene la secuencia de vértices que forman el camino mínimo.

El *algoritmo de Warshall* es importante ya que, para todo par de vértices del grafo, indica si hay un camino entre ellos. Con una estructura similar al anterior, el *algoritmo de Floyd* se utiliza para encontrar los caminos mínimos entre todos los pares de vértices de un grafo; también ofrece la posibilidad de obtener la traza de esos caminos.

Un *árbol de expansión* es un árbol que contiene a todos los vértices de una red. En un árbol no hay ciclos; es una forma de averiguar si la red está conectada. Uno de los problemas más comunes que se plantean sobre las redes es encontrar aquel *árbol de coste mínimo*. El *algoritmo de Prim* y el *algoritmo de Kruskal* resuelven el problema de encontrar el árbol de expansión mínimo.

Los grafos también se aplican en modelar problemas de fluidos. En una red de flujo hay un vértice *fuente* y un vértice destino, *sumidero*, al que se dirige el flujo de objetos. Entre el fuente y el sumidero hay un conjunto de nodos intermedios y de *aristas dirigidas* por las que puede pasar un máximo de fluido, la capacidad de la arista. La cuestión que se plantea es cómo distribuir el flujo por las aristas para que llegue el máximo de fluido al vértice *sumidero*. El *algoritmo de Ford-Fulkerson* obtiene una solución al problema en base de obtener iterativamente *cadenas de aumento de flujo*.

EJERCICIOS

18.1. Dada la red de la Figura 18.19, encontrar una ordenación topológica.

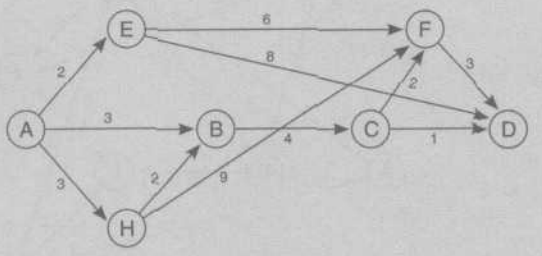


Figura 18.19. Grafo dirigido.

18.2. En la red de la Figura 18.19 los arcos representan actividades y el factor de peso el tiempo necesario para realizar dicha actividad (un diagrama *Pert*). Cada vértice `v` de la red representa el tiempo que tardan todas las actividades que terminan en `v`. El ejercicio consiste en asignar a cada vértice `v` de la red el tiempo necesario para que todas las actividades que terminan en `v` se puedan realizar y se llamará `tn(v)`. Una estrategia que se puede seguir: asignar tiempo `0` a los vértices sin predecesores; si a todos los predecesores de `v` se les ha asignado tiempo, entonces `tn(v)` es el máximo, para cada predecesor, de la suma de tiempo del predecesor con el factor de peso del arco desde ese predecesor hasta `v`.

18.3. Considerando de nuevo la red de la Figura 18.19 y teniendo en cuenta el tiempo de cada vértice, `tn(v)`, calculado en el Ejercicio 18.2, se quiere calcular el tiempo límite en que todas las actividades que terminan en `v` pueden ser completadas, sin atrasar la terminación de todas las actividades; a este tiempo se le denomina `tl(v)`. La estrategia a seguir para este cálculo: asignar `tn(v)` a todos los

vértices `v` sin sucesores, si todos los sucesores de `v` tienen tiempo asignado, entonces `tl(v)` es el mínimo, entre todos los sucesores, de la diferencia entre el tiempo asignado al sucesor, `tl(v')`, y el factor de peso desde `v` hasta el sucesor `v'`.

18.4. Una ruta crítica de una red es un camino desde un vértice que no tiene predecesores hasta otro vértice que no tiene sucesores, tal que para todo vértice `v` del camino se cumple que `tn(v) = tl(v)`. Encontrar las rutas críticas de la red de la Figura 18.19.

18.5. La Figura 18.20 muestra una red conectada. Encontrar (dibujar) un árbol de expansión de coste mínimo aplicando los pasos que propone el *algoritmo de Prim*.

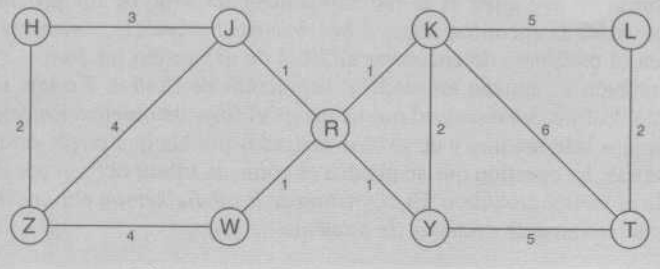


Figura 18.20. Red conectada.

18.6. Encontrar el árbol de expansión de coste mínimo de la red conectada de la Figura 18.20, siguiendo los pasos del *algoritmo de Kruskal*.

18.7. El grafo de la Figura 18.21 es dirigido y las aristas tienen asociado un *coste*. Determinar el camino más corto desde el vértice `A` al resto de vértices del grafo.

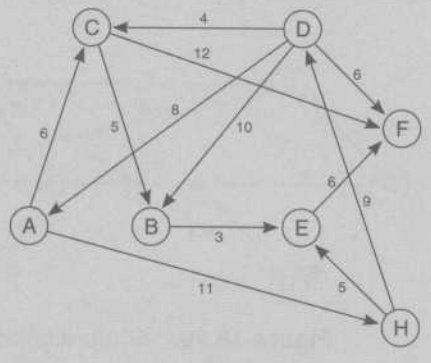


Figura 18.21. Grafo dirigido con factor de peso.

18.8. Dado el grafo de la Figura 18.21 encontrar el camino más corto desde el vértice `D` al resto de vértices. Se ha de seguir los pasos del *algoritmo de Dijkstra*, e incluir la secuencia de vértices que forman el camino.

18.9. En el grafo de la Figura 18.21 obtener los caminos más cortos entre todos los pares de vértices. Aplicar, paso a paso, el *algoritmo de Floyd*.

18.10. Dibujar un grafo dirigido con factor de peso en el que algún arco tenga factor de peso negativo. Al aplicar el *algoritmo de Dijkstra* desde un vértice origen, se ha de obtener algún resultado erróneo.

18.11. Escribir las modificaciones necesarias para que, teniendo como base el *algoritmo de Dijkstra*, se calculen los caminos mínimos entre todos los pares de vértices del grafo. ¿Cuál es la eficiencia de este algoritmo?

18.12. Tanto el *algoritmo de Prim* como el de *Kruscal* resuelven el problema de determinar el árbol de expansión mínimo de una red conectada. ¿Se pueden aplicar estos algoritmos si alguna arista tiene factor de peso negativo?

18.13. Para resolver el problema de encontrar los caminos mínimos desde un vértice origen al resto de los vértices, con aristas de coste negativo, se puede aplicar un algoritmo propuesto por Bellman-Ford. Este algoritmo no resuelve el problema si el grafo tiene ciclos. En el caso de existir ciclos con aristas con coste negativo, el algoritmo lo indica y da por terminada la búsqueda.

Suponiendo un grafo de `n` vértices, considerando el vértice `1` como el origen, siendo `P` la matriz de pesos, el algoritmo de Bellman-Ford en seudocódigo:

```
inicio
   desde v ( 2 hasta n hacer
     d(v) ← ∞
   fin_desde
   s ← 1                                  { vértice origen }
   d(s) ← 0
   desde i ← 1 hasta n - 1 hacer
     para cada arista (u,v) hacer
       si d(v) > d(u) + P(u,v) entonces
         d(v) ← d(u) + P(u,v)
       fin_si
   fin_desde

   { prueba de ciclos con alguna arista de peso negativo }

   para cada arista (u,v) hacer
     si d(v) > d(u) + P(u,v) entonces
       Error ← true
     fin_si

            { en el array d están los caminos mínimos desde
                          el vértice s al resto de vértices }

fin
```

Como ejercicio se pide dibujar dos grafos con alguna arista que tenga factor de peso negativo, y aplicar el algoritmo descrito. En un grafo hacer que haya un ciclo con una arista de peso negativo.

18.14. Determinar, en el grafo de la Figura 18.22, los vértices que son *puntos de articulación*.

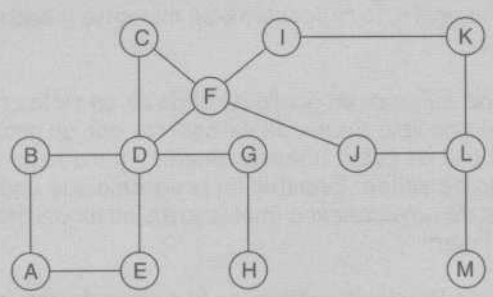


Figura 18.22. Grafo no dirigido con puntos de articulación.

18.15. Un *circuito de Euler*, en un grafo dirigido, es un ciclo en el cual toda arista es *visitada* exactamente una vez. Se puede demostrar que un grafo dirigido tiene un *circuito de Euler* si y solo si es *fuertemente conexo* y para todo vértice el grado de entrada es igual al grado de salida. Dibujar un grafo dirigido en el que se pueda encontrar un *circuito de Euler*.

PROBLEMAS

18.1. Dado un grafo dirigido sin ciclos y factor de peso que representa un *Pert*, escribir un programa que tenga como entrada la matriz de pesos y calcule:

a) El tiempo `tn(v)`.
b) El tiempo `tl(v)`.

La definición de estas magnitudes se encuentra en los Ejercicios 18.2 y 18.4.

18.2. Un grafo que representa a una red es poco densa; en un programa se quiere implementar el grafo mediante *listas de adyacencia*. El programa ha de encontrar las *rutas críticas* del grafo.

18.3. Se tiene una red (un *Pert*) representada con su matriz de pesos. Escribir un programa que calcule el mínimo tiempo en el que todo trabajo se puede terminar, si tantas tareas como sea posible son realizadas en paralelo. El programa debe escribir el tiempo en el que se inicia y se termina toda actividad en la red.

18.4. Escribir un programa para que dada una red determine el tiempo mínimo de realización del trabajo si como máximo se pueden realizar *n* actividades (de las posibles) en paralelo. El programa tiene como entrada la matriz de pesos del grafo, la salida del programa muestra el tiempo de inicio y el de finalización de cada actividad.

18.5. En cada iteración que realiza el *algoritmo de Dijkstra* se selecciona aquel vértice para el cual el camino es mínimo desde el vértice origen. Implementar el algoritmo teniendo en cuenta que si en una iteración hay dos vértices en los que coincide la distancia mínima se selecciona aquel con el menor número de arcos.

18.6. Tomando como base el *algoritmo de Dijkstra*, implementar un algoritmo que encuentre los caminos mínimos desde todo vértice `v` a un mismo vértice destino `d`, en un grafo *G* representado por su matriz de pesos.

18.7. 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. Utilizar el *algoritmo de Bellman-Ford* del Ejercicio 18.13.

18.8. Dado un grafo no dirigido con factor de peso, escribir un programa que tenga como entrada dicho grafo, lo represente en memoria y encuentre el árbol de expansión de coste mínimo.

18.9. Un *circuito de Euler* en un grafo dirigido es un ciclo en el cual toda arista es visitada exactamente una vez. Se puede demostrar que un grafo dirigido tiene un *circuito de Euler* si y sólo si es fuertemente conexo y para todo vértice el grado de entrada es igual al grado de salida. Escribir un programa que implemente un grafo dirigido mediante *listas de adyacencia* e implemente un algoritmo para encontrar, si existe, un *circuito de Euler*.

18.10. El *algoritmo de Warshall* determina la matriz de caminos de un grafo; la implementación con la matriz de adyacencia es adecuada en el caso de que el grafo sea lo sufi-

cientemente *denso*. Implementar el algoritmo para un grafo *poco denso*, representado con listas de adyacencia.

18.11. Se quiere escolarizar una zona rural compuesta de 4 poblaciones: Centenera, Lupiana, Atanzón y Pinilla. Para ello se va a construir un único centro escolar en la población que mejor coste desplazamiento educativo tenga, mínimo de la función z_i:

$z_i = \sum p_j d_{ij}$; donde p_j es la población escolar del pueblo j y d_{ij} es la distancia mínima del pueblo j al pueblo i.

Escribir un programa que tenga como entrada los datos de la población escolar y la distancia, en kilómetros, entre los pueblos. El programa determinará el pueblo en el que conviene ubicar el centro escolar.

18.12. Un grafo `G` simula una red de centros de distribución. Cada centro dispone de una serie de artículos y un stock de ellos, representado mediante una estructura lineal ordenada respecto al código de artículo. Entre los centros hay conexiones que no tienen por qué ser bidireccionales. Escribir un programa en el que se represente el grafo dirigido ponderado (el coste de las aristas es la distancia, en kilómetros, entre los dos vértices). El programa tiene que resolver el problema de que un centro no tenga un artículo `z` y lo requiera, entonces el centro más cercano que disponga de `z` lo suministra.

18.13. Europa consta de `n` capitales de cada uno de los estados, cada par de ellas está conectada, o no, por vía aérea. En caso de estar conectadas (se supone por vuelo directo) se conocen la distancia en millas y el precio del vuelo. Las conexiones, aun siendo bidireccionales, no siempre tienen la misma distancia debido a las distintas rutas. Escribir un programa en el que se simule el modelo mediante un grafo, y se resuelva el problema: dada una cantidad de euros `E`, y un par de capitales `(c1, c2)`, averiguar la ruta más corta que se ajuste al valor `E`.

18.14. La ciudad dormitorio Alcohete está conectada a través de una red de carreteras, pasando por poblaciones intermedias, con el centro de la gran ciudad. Cada conexión entre dos nodos soporta un máximo de vehículo por hora, que es la *capacidad de la arista*. Escribir un programa que simule esta situación mediante una *red de flujos*; la entrada de datos ha de ser los nodos de que consta la red, incluyendo el origen (ciudad dormitorio) y el destino, y la capacidad de cada conexión entre dos nodos. El programa implementará el algoritmo de aumento de *flujo de Ford-Fulkerson*, para calcular el máximo de vehículos/hora que pueden llegar al centro y cómo es su distribución por las distintas conexiones (calzadas).

APÉNDICES

APÉNDICE A

LENGUAJE ALGORÍTMICO UPSAM VERSIÓN 2.0

CONTENIDO

A.1. Elementos del lenguaje.
A.2. Estructura de un programa.
A.3. Estructuras de control.
A.4. Programación modular.
A.5. Archivos.
A.6. Variables dinámicas.
A.7. Programación orientada a objetos.
A.8. Palabras reservadas.

A.1. ELEMENTOS DEL LENGUAJE

A.1.1. Identificadores

Se pueden formar con cualquier carácter alfabético regional (no necesariamente ASCII estándar), dígitos (0-9) y el símbolo de subrayado (_), debiendo empezar siempre por un carácter alfabético. Los nombres de los identificadores son sensibles a mayúsculas y se recomienda que su longitud no sobrepase los 50 caracteres.

A.1.2. Comentarios

Existen dos tipos de comentarios. Para comentarios de una sola línea, se utilizará la doble barra inclinada (//). Este símbolo servirá para ignorar todo lo que aparezca hasta el final de la línea. Los comentarios también podrán ocupar más de una línea utilizando los caracteres { y }, que indicarán respectivamente el inicio y el final del comentario. Todos los caracteres incluidos entre estos dos símbolos serán ignorados.

A.1.3. Tipos de datos estándar

Datos numéricos

- **Enteros.** Se considera entero cualquier valor numérico sin parte decimal, independientemente de su rango. Para la declaración de un tipo de dato entero se utiliza la palabra reservada `entero`.
- **Reales.** Se considera real cualquier valor numérico con parte decimal, independiente de su rango o precisión. Para la declaración de un tipo de dato real se utiliza la palabra reservada `real`.

Datos lógicos

Se utiliza la palabra reservada `lógico` en su declaración.

Datos de tipo carácter

Se utiliza la palabra reservada `carácter` en su declaración.

Datos de tipo cadena

Se utiliza la palabra reservada `cadena` en su declaración. A no ser que se indique lo contrario, se consideran cadenas de longitud variable. Las cadenas de caracteres se consideran como un tipo de dato estándar pero estructurado (se podrá considerar como un array de caracteres).

A.1.4. Constantes de tipos de datos estándar

Numéricas enteras

Están compuestas por los dígitos (`0..9`) y los signos + y – utilizados como prefijos.

Numéricas reales

Los números reales en coma fija, utilizan el punto como separador decimal, además de los dígitos (`0..9`), y el carácter de signo (+ y –).

Para los reales en coma fija, la mantisa podrá utilizar los dígitos (`0..9`), el carácter de signo (+ y –) y el punto decimal (.). El exponente se separará de la mantisa mediante la letra `E` y la mantisa estará formada por el carácter de signo y los dígitos.

Lógicas

Sólo podrán contener los valores `verdad` y `falso`.

De carácter

Cualquier carácter válido del juego de caracteres utilizados delimitados por los separadores ' o ".

De cadena

Secuencia de caracteres válidos del juego de caracteres utilizados, delimitados por los separadores ' o ".

A.1.5. Operadores

Operadores aritméticos

Operador	Significado
–	Menos unario
–	Resta
+	Más unario
*	Multiplicación
/	División real
div	División entera
mod	Resto de la división entera
^ (**)	Exponenciación

El tipo de dato de una expresión arimética depende del tipo de dato de los operandos y del operador. Con los operadores +, −, * y ^, el resultado es entero si los operandos son enteros. Si alguno de los operandos es real el resultado será de tipo real. La división real (/) devuelve siempre un resultado real. Los operadores `mod` y `div` devuelven siempre un resultado de tipo entero.

Operadores de relación

Operador	Significado
=	Igual a
<	Menor que
>	Mayor que
<=	*Menor o igual que*
>=	Mayor o igual que
<>	Distinto de

Los operandos deben ser del mismo tipo y el resultado es de tipo lógico.

Operadores lógicos

Operador	Significado
no	Negación lógica
y	Multiplicación lógica (verdadero si los dos operandos son verdaderos)
o	Suma lógica (verdadero si alguno de los operandos es verdadero)

Los operandos deben ser de tipo lógico y devuelven un operando de tipo lógico.

Operadores de cadena

Operador	Significado
+	Concatenación de cadenas
&	Concatenación de cadenas

Trabajan con operandos de tipo cadena o carácter y el resultado siempre será de tipo cadena.

Prioridad de operadores

Primarios	( ) [] *Paréntesis en expresiones o en llamadas a procedimientos o funciones. Corchetes en índices de arrays*
Unarios	−, +, no
Multiplicativos	*, /, div, mod, y
Aditivos	+, −, o
De cadena	&, +
De relación	=, <, >, <=, >=, <>

A.2. ESTRUCTURA DE UN PROGRAMA

```
algoritmo <nombre_del_algoritmo>
//Secciones de declaraciones
[const
  //declaraciones de constantes]
[tipos
  //declaraciones de tipos]
[var
  //declaraciones de variables]
//Cuerpo del programa
inicio
  ...
fin
```

A.2.1. Declaración de tipos de datos estructurados

Arrays

```
array[<dimensión>...] de <tipo_de_dato> : <nombre_del_tipo>
```

`<dimensión>` es un subrango con el índice del límite inferior y el límite superior. Por ejemplo, **array**`[5..20]` **de entero** declararía un array de 16 elementos enteros. Pueden aparecer varios separados por comas para declarar arrays de más de una dimensión.

`<tipo_de_dato>` es el identificador de cualquier tipo de dato estándar o definido por el usuario.

`<nombre_del_tipo>` es un identificador válido que se utilizará para referenciar el tipo de dato.

El acceso a un elemento de un array se realizará indicando su índice entre corchetes. El índice será una expresión entera.

Registros

```
registro : <nombre_del_tipo>
  <tipo_de_dato> : <nombre_del_campo>
  ...
fin_registro
```

`<tipo_de_dato>` es el identificador de cualquier tipo de dato estándar o definido por el usuario.

`<nombre_del_tipo>` es un identificador válido que se utilizará para referenciar el tipo de dato.

`<nombre_del_campo>` es un identificador válido que se utilizará para referenciar el campo del registro.

El acceso a un campo de una variable de tipo registro se realizará utilizando el carácter punto (.), por ejemplo, `MiRegistro.MiCampo`.

Archivos secuenciales

```
archivo_s de <tipo_de_dato> : <nombre_del_tipo>
```

`<tipo_de_dato>` es el identificador de cualquier tipo de dato estándar o definido por el usuario.

`<nombre_del_tipo>` es un identificador válido que se utilizará para referenciar el tipo de dato.

Archivos directos

```
archivo_d de <tipo_de_dato> : <nombre_del_tipo>
```

<tipo_de_dato> es el identificador de cualquier tipo de dato estándar o definido por el usuario.

<nombre_del_tipo> es un identificador válido que se utilizará para referenciar el tipo de dato.

A.2.2. Declaración de constantes

Se realiza dentro de la sección de declaraciones de constantes.

```
<nombre_de_constante> = <expresión>
```

<nombre_de_constante> es un identificador válido que se utilizará para referenciar la constante.

<expresión> es una expresión válida. El tipo de la constante será el tipo de dato que devuelva la expresión.

A.2.3. Declaración de variables

Se realiza dentro de la sección de declaraciones de variables.

```
<tipo_de_dato> : <nombre_de_variable>[=<expresión>]...
```

<tipo_de_dato> es el identificador de cualquier tipo de dato estándar o definido por el usuario.

<nombre_de_variable> es un identificador válido que se utilizará para referenciar la variable. En una declaración es posible declarar varias variables separadas por comas.

Es posible inicializar la variable en la declaración, <expresión> es una expresión válida del tipo de dato de la variable.

A.2.4. Biblioteca de funciones

Funciones aritméticas

Función	Significado
abs(x)	Devuelve el valor absoluto de la expresión numérica *x*
aleatorio()	Devuelve un número aleatorio real mayor o igual que 0 y menor que 1
arctan(x)	Devuelve la arco tangente de *x*
cos(x)	Devuelve el coseno de *x*
entero(x)	Devuelve el primer valor entero menor que la expresión numérica *x*
exp(x)	Devuelve el valor e^x
ln(x)	Devuelve el logaritmo neperiano de *x*
log10(x)	Devuelve el logaritmo en base 10 de *x*
raiz2(x)	Devuelve la raíz cuadrada de *x*
sen(x)	Devuelve el seno de *x*
trunc(x)	Trunca (elimina los decimales) de la expresión numérica *x*

Funciones de cadena

Función	Significado
longitud(c)	Devuelve el número de caracteres de la cadena c
posición(c,sc)	Devuelve la posición de la primera aparición de la subcadena sc en la cadena c
subcadena(c,ini[,long])	Devuelve una subcadena de la cadena c formada por todos los caracteres a partir de la posición ini. Si se incluye el argumento long, devuelve sólo los primeros long caracteres a partir de la posición ini

Funciones de conversión de número a cadena

Función	Significado
código(car)	Devuelve el código ASCII del carácter car
aCarácter(x)	Devuelve el carácter correspondiente al código ASCII x
valor(c)	Convierte la cadena c a un valor numérico. Si el contenido de la cadena c no puede convertirse a un valor numérico (contiene caracteres alfabéticos, signos de puntuación inválidos, etc.), devuelve 0
aCadena(x)	Convierte a cadena el valor numérico x

Funciones de información

Función	Significado
tamaño_de(<variable>)	Devuelve el tamaño en bytes de la variable

A.2.5. Procedimientos de entrada/salida

leer(<lista_de_variables>) lee una o más variables desde la consola del sistema.
escribir(<lista_de_expresiones>) escribe una o más expresiones en la consola del sistema.

A.2.6. Instrucción de asignación

```
<variable> ← <expresión>
```

Primero evalúa el valor de la expresión y lo asigna a la variable. La variable y la expresión deben ser del mismo tipo de dato.

A.3. ESTRUCTURAS DE CONTROL

A.3.1. Estructuras selectivas

Estructura selectiva simple y doble

```
si <expresión_lógica> entonces
  <acciones>
[si_no
  <acciones>
fin_si
```

Estructura selectiva múltiple

```
según_sea <expresión> hacer
  <lista_de_valores> : <acciones>
  ...
[si_no
  <acciones>]
fin_según
```

<expresión> puede ser cualquier expresión valida.

<lista_de_valores> será uno o más valores separados por comas de mismo tipo que <expresión>. La estructura verifica si el valor de la expresión concide con alguno de los valores *de la primera lista de valores; si esto ocurre realiza las acciones correspondientes y el flujo de* control sale de la estructura, en caso contrario evalúa la siguiente lista. Las acciones de la claúsula **si_no** se ejecutará si ningún valor coincide con la <expresión>.

A.3.2. Estructuras repetitivas

Estructura mientras

```
mientras <expresión_lógica> hacer
  <acciones>
fin_mientras
```

Estructura repetir

```
repetir
  <acciones>
hasta_que <expresión_lógica>
```

Estructura desde

```
desde <variable> ← <valor_inicial> hasta <valor_final>
          [incremento | decremento <valor_incremento>] hacer
  <acciones>
fin_desde
```

<variable> puede ser cualquier variable en la que se pueda incrementar o decrementar su valor, es decir, todas las numéricas, las de tipo carácter y las lógicas.

<valor_inicial> es una expresión con el primer valor que toma la variable del bucle. Debe ser del mismo tipo que la variable del bucle.

<valor_final> es una expresión con el último valor que toma la variable del bucle. Debe ser del mismo tipo que la variable del bucle. El bucle finaliza cuando la variable toma un valor mayor que este valor inicial.

<valor_incremento> es una expresión con el valor en el que se incrementará o decrementará la variable del bucle al final de cada iteración.

A.4. PROGRAMACIÓN MODULAR

A.4.1. Cuestiones generales

El ámbito de las variables declaradas dentro de un módulo (procedimiento o función) es local, y el tiempo de vida de dicha variable será el tiempo de ejecución del módulo.

A.4.2. Procedimientos

Declaración

```
procedimiento <nombre_procedimiento>([<lista_parámetros_formales>])
[declaraciones locales]
inicio
  ...
fin_procedimiento
```

<nombre_procedimiento> debe ser un identificador válido.

<lista_parámetros_formales> son uno o más grupos de parámetros separados por punto y coma. Cada grupo de argumentos se define de la siguiente forma:

```
{E | E/S} <tipo_de_dato> : <lista_de_parámetros>
```

E indica que el paso de parámetros se realiza por valor.
E/S indica que el paso de parámetros se realiza por referencia.
<tipo_de_dato> es un tipo de dato estándar o definido previamente por el usuario.
<lista_de_parámetros> es uno o más identificadores válidos separados por comas.

Llamada a procedimientos

```
[llamar_a] <nombre_procedimiento>([<lista_parámetros_actuales>])
```

La lista de parámetros actuales es una o varias variables o expresiones separadas por comas que deben coincidir en número, orden y tipo con la lista de parámetros formales de la declaración.

A.4.3. Funciones

Declaración

```
<tipo_de_dato> :función <nombre_función>([<lista_parámetros_formales>])
[declaraciones locales]
inicio
  ...
  devolver(<expresión>)
fin_función
```

<tipo_de_dato> es un tipo de dato estándar o definido previamente por el usuario. Se trata del tipo del dato que devuelve la función.

<nombre_función> debe ser un identificador válido.

<lista_parámetros_formales> son uno o más grupos de parámetros separados por punto y coma. Cada grupo de argumentos se define de la siguiente forma:

```
{E | E/S} <tipo_de_dato> : <lista_de_parámetros>
```

E indica que el paso de parámetros se realiza por valor.
E/S indica que el paso de parámetros se realiza por referencia.
<tipo_de_dato> es un tipo de dato estándar o definido previamente por el usuario.
<lista_de_parámetros> es uno o más identificadores válidos separados por comas.
<expresión> es el valor de retorno de la función. Debe coincidir con el tipo de dato de la declaración.

Llamada a funciones

```
<nombre_función>([<lista_parámetros_actuales>])
```

La lista de parámetros actuales es una o varias variables o expresiones separadas por comas que deben coincidir en número, orden y tipo con la lista de parámetros formales de la declaración. Al devolver un valor y no existir funciones que no devuelven valores (funciones `void` de C o Java), la llamada debe hacerse siempre dentro de una expresión.

A.5. ARCHIVOS

A.5.1. Archivos secuenciales

Apertura del archivo

```
abrir(<variable_tipo_archivo,<modo_apertura>,<nombre_archivo>)
```

`<var_tipo_archivo>` es una variable de tipo archivo secuencial.

`<modo_apertura>` indica el tipo de operación que se realizará con el archivo. En el caso de archivos secuenciales será:

- **`lectura`** coloca el puntero al siguiente registro al comienzo del archivo y sólo realiza operaciones de lectura. El archivo debe existir previamente.
- **`escritura`** coloca el puntero al siguiente registro al comienzo del archivo y sólo realiza operaciones de escritura. Si el archivo no existe, primero crea un archivo vacío. Si el archivo existe, sobrescribe los datos que tenga.
- **`añadir`** coloca el puntero al siguiente registro en la marca de final de archivo y sólo realiza operaciones de escritura.

`<nombre_archivo>` es una expresión de cadena con el nombre que el sistema dará al archivo.

Cierre del archivo

```
cerrar(<lista_variables_tipo_archivo>)
```

Cierra el archivo o archivos abiertos previamente.

Entrada/salida

```
leer(<variable_tipo_archivo>, <variable>)
```

Lee del archivo abierto para lectura representado por `<variable_tipo_archivo>` el siguiente registro. El tipo de la variable debe coincidir con el tipo base del archivo definido en la declaración del tipo de dato.

```
escribir(<variable_tipo_archivo>, <expresión>)
```

Escribe en el archivo abierto para escritura y representado por la variable de tipo archivo el valor de la expresión. El tipo de la expresión debe coincidir con el tipo base del archivo definido en la declaración del tipo de dato.

A.5.2. Archivos de texto

Se considera el archivo de texto como un tipo especial de archivo compuesto de caracteres o cadenas. La declaración de un tipo de dato de tipo archivo de texto sería por tanto:

```
archivo_s de carácter : <nombre_tipo>
archivo_s de cadena : <nombre_tipo>
```

La lectura de un carácter único en un archivo de texto se haría de la forma **leer**(<variable_tipo_carácter>), que leería el siguiente carácter del archivo. La lectura de una variable de tipo cadena (**leer**(<variable_tipo_cadena>)), leería todos los caracteres hasta el final de línea.

La escritura de datos en un archivo de texto también se podrá hacer carácter a carácter (**escribir**(<variable_tipo_carácter>)) o línea a línea (**escribir**(<variable_tipo_cadena>)).

La detección del final de línea en un archivo de texto cuando se lee carácter a carácter se realizaría con la función **fdl**:

```
fdl(<variable_tipo_archivo>)
```

La función **fdl** devuelve el valor lógico **verdad**, si el último carácter leído es el carácter de fin de línea.

A.5.3. Archivos directos

Apertura del archivo

```
abrir(<variable_tipo_archivo,<modo_apertura>,<nombre_archivo>)
```

<variable_tipo_archivo> es una variable de tipo archivo directo.

<modo_apertura> indica el tipo de operación que se realizará con el archivo. En el caso de archivos directos será:

- **lectura** coloca el puntero al siguiente registro al comienzo del archivo y sólo realiza operaciones de lectura. El archivo debe existir previamente.
- **escritura** coloca el puntero al siguiente registro al comienzo del archivo y sólo realiza operaciones de escritura. Si el archivo no existe, primero crea un archivo vacío. Si el archivo existe, sobrescribe los datos que tenga.
- **lectura/escritura** coloca el puntero al comienzo del archivo y permite operaciones tanto de lectura como de escritura.

<nombre_archivo> es una expresión de cadena con el nombre que el sistema dará al archivo.

Cierre del archivo

```
cerrar(<lista_variables_tipo_archivo>)
```

Cierra el archivo o archivos abiertos previamente.

Acceso secuencial

leer(<variable_tipo_archivo>, <variable>)

Lee del archivo abierto para lectura representado por <variable_tipo_archivo> el siguiente registro. El tipo de la variable debe coincidir con el tipo base del archivo definido en la declaración del tipo de dato.

escribir(<variable_tipo_archivo>, <expresión>)

Escribe en el archivo abierto para escritura y representado por la variable de tipo archivo el valor de la expresión. El tipo de la expresión debe coincidir con el tipo base del archivo definido en la declaración del tipo de dato.

Acceso directo

leer(<variable_tipo_archivo>, <posición>, <variable>)

Lee el registro situado en la posición relativa <posición> y guarda su contenido en la variable.

escribir(<variable_tipo_archivo>, <posición>, <variable>)

Escribe el contenido de la variable en la posición relativa <posición>.

A.5.4. Consideraciones adicionales

Detección del final del archivo

Al cerrar un archivo abierto para escritura se coloca después del último registro la marca de fin de archivo. La función **fda** permite detectar si se ha llegado a dicha marca.

fda(<variable_tipo_archivo>)

Devuelve el valor lógico **verdad** si se ha intentado hacer una lectura secuencial después del último registro.

Determinar el tamaño del archivo

La función **lda** devuelve el número de bytes del archivo.

lda(*<nombre_archivo>*)

<nombre_archivo> es el nombre del archivo físico.
Para determinar el número de registros de un archivo se puede utilizar la expresión:

lda(<nombre_archivo>) / **tamaño_de**(<tipo_base_archivo>)

Otros procedimientos

borrar(<nombre_archivo>)

Elimina del disco el archivo representado por la expresión de cadena <nombre_archivo>. El archivo debe estar cerrado.

renombrar(<nombre_archivo>,<nuevo_nombre>)

Cambia el nombre al archivo <nombre_archivo> por el de <nuevo_nombre>. El archivo debe estar cerrado.

A.6. VARIABLES DINÁMICAS

Declaración de tipos de datos dinámicos

puntero_a <tipo_de_dato> : <nombre_del_tipo>

Declara el tipo de dato <nombre_del_tipo> como un puntero a variables de tipo <tipo_de_dato>.

El valor constante **nulo** indica una referencia a un puntero nulo.

Referencia al contenido de una variable dinámica

<variable_dinamica>↑

Asignación y liberación de memoria con variables dinámicas

reservar(<variable_dinámica>)

Reserva espacio en memoria para una variable del tipo de dato del puntero y hace que la variable dinámica apunte a dicha zona.

liberar(<variable_dinamica>)

Libera el espacio de memoria apuntado por la variable dinámica. Dicha variable queda con un valor indeterminado.

A.7. PROGRAMACIÓN ORIENTADA A OBJETOS

A.7.1. Clases y objetos

Declaración de clases

```
clase <nombre_de_clase>
  //Declaración de atributos
  //Declaración de constructores y métodos
fin_clase
```

<nombre_de_clase> es un identificador válido.

Declaración de tipos de referencias

<nombre_de_clase> : <nombre_de_referencia>

<nombre_de_clase> es el nombre de una clase previamente declarada.

<nombre_de_referencia> es un identificador válido que se utilizará para referenciar a un objeto de dicha clase.

La declaración de una referencia a una clase se hará en la sección de declaraciones de variables o tipos de datos de un algoritmo, o dentro de la sección de variables de otra clase.

Instanciación de clases

```
nuevo <nombre_de_constructor>([<argumentos_constructor>])
```

La declaración **nuevo** reserva espacio para un nuevo objeto de la clase a la que pertenece el constructor y devuelve una referencia a un objeto de dicha clase. `<nombre_de_constructor>` tendrá el mismo nombre de la clase a la que pertenece. La llamada al constructor puede llevar argumentos para la inicialización de atributos (véase más adelante en el apartado de constructores).

La instanciación se puede realizar en una sentencia de asignación, o en la propia declaración de la referencia, dentro de la sección de declaraciones del algoritmo.

```
MiObjeto ← nuevo MiClase(arg1,arg2,arg3) //Dentro del código ejecutable
MiClase : MiObjeto = nuevo MiClase(arg1,arg2,arg3) //En la sección var
```

Referencias a miembros de una clase

```
NombreReferencia.nombreDeMiembro //Para atributos
NombreReferencia.nombreDeMiembro([listaParamActuales]) //Para métodos
```

Constructores

```
constructor <nombre_de_clase>[<lista_parametros_formales>])
//Declaración de variables locales
inicio
  //Código del constructor
fin_constructor
```

Existe un constructor por omisión sin argumentos al que se le llama mediante `<nombre_de_clase()>`. Al igual que con los métodos se admite la sobrecarga dentro de constructores, distinguiéndose los distintos constructores por el orden, número y/o tipo de sus argumentos.

Puesto que la misión de un constructor es inicializar una instancia de una clase, la `<lista_de_parámetros_formales>` sólo incluyen argumentos de entrada, por lo que se puede omitir la forma en la que se pasan los argumentos.

Destructores

No se considera la existencia de destructores. Las instancias se consideran destruidas cuando se ha perdido una referencia a ellas (*recolector de basura*).

Visibilidad de las clases

Se consideran todas las clases como públicas, es decir, es posible acceder a los miembros de cualquier clase declarada en cualquier momento.

Referencia a la instancia desde dentro de la declaración de una clase

Es posible hacer referencia a una instancia desde dentro de una clase con la palabra reservada **instancia** que devuelve una referencia a la instancia de la clase que ha realizado la llamada al método. De esta forma, **instancia**.`UnAtributo` haría referencia al valor del atributo `UnAtributo` dentro de la instancia actual.

A.7.2. Atributos

Declaración de atributos

La declaración de los atributos de una clase se realizará dentro de la sección de declaraciones **var** de dicha clase.

```
var
    [const] [privado|publico|protegido] [estático]
            <tipo_de_dato> : <nombre_atributo> [= <valor_inicial>]
    ...
```

<nombre_atributo> puede ser cualquier identificador válido.

<tipo_de_dato> puede ser cualquier tipo de dato estándar, definido por el usuario u otra clase declarada con anterioridad.

Es posible dar un valor inicicia al atributo mediante una expresión de inicialización que deberá ser del mismo tipo de dato que el atributo.

Visibilidad de los atributos

Por omisión se considera a los atributos privados, es decir, sólo son accesibles por los miembros de la clase. Para que pueda ser utilizado por los miembros de otras clases se utilizará el modificador **público.** El modificador **protegido** se utiliza para que sólo pueda ser utilizado por los miembros de su clase y por los de sus clases hijas.

Atributos de clase (estáticos)

Un atributo que tenga el modificador **estático** no pertenece a ninguna instancia de la clase, sino que será común a todas ellas. Para hacer referencia a un atributo de una clase se utilizará el nombre de la clase seguido del nombre del atributo (`MiClase.MiAtributoEstático`).

Atributos constantes

El modificador **const** permite crear atributos constantes que no se modificarán durante el tiempo de vida de la instancia.

A.7.3. Métodos

Declaración de métodos

La declaración de métodos se realizará dentro de la clase después de la declaración de atributos sin indicar ninguna sección especial.

```
[estático] [abstracto] [público|privado|protegido] <tipo_de_retorno>
  método <nombre_del_método>(<lista_de_parámetros_formales>)
//declaración de variables
inicio
  //Código
  [devolver(<expresión>)]
fin_método
```

<nombre_del_método> es un identificador válido.

<tipo_de_retorno> es cualquier tipo de dato estándar, estructurado o una referencia a un objeto. La declaración **devolver** se utiliza para indicar el dato de retorno que devuelve el método

que debe coincidir con el tipo de retorno que aparece en la declaración. Si el método no devuelve valores se utilizará la palabra reservada **nada** y no aparecerá la palabra **devolver.**

La lista de parámetros formales se declararía igual que en los procedimientos y funciones. El paso de argumentos se realizará como en los procedimiento y funciones normales.

Las variables locales se declararán en la sección **var** entre la cabecera del método y su cuerpo.

Visibilidad de los métodos

Por omisión, se consideran los métodos como públicos, es decir, es posible acceder a ellos desde cualquier lugar del algoritmo. Para que pueda ser utilizado sólo por miembros de su clase se utilizará el modificador **privado,** en el caso de los procedimientos, o **privada,** en el caso de las funciones. El modificador **protegido** o **protegida** se utiliza para que sólo pueda ser utilizado por los miembros de su clase y por los de sus clases hijas.

Métodos estáticos

Un método que tenga el modificador **estático** o **estática** no pertenece a ninguna instancia de la clase, sino que será común a todas ellas. Para hacer referencia a un método de una clase se utilizará el nombre de la clase seguido del nombre del método (`MiClase.MiMétodoEstático()`).

Sobrecarga de métodos

Se permite la sobrecarga de métodos, es decir, la declaración de métodos con el mismo nombre pero con funcionalidades distintas. Para que en la llamada se pueda distinguir entre los métodos sobrecagados, número, orden o tipo de sus argumentos deben cambiar.

Ligadura de métodos

La ligadura de la llamada de un método con el método correspondiente se hace **siempre** de forma dinámica, es decir, en tiempo de ejecución, con lo que se permite la existencia de **polimorfismo.**

A.7.4. Herencia

clase <clase_derivada> **hereda_de** <superclase>

<clase_derivada> es un identificador válido.
<superclase> es una clase declarada anteriormente.
La clase derivada:

- Hereda todos los métodos y atributos de la superclase accesibles (atributos públicos y protegidos y métodos públicos y protegidos) presentes sólo en la superclase.
- Sobreescribre todos los métodos y atributos de la superclase accesibles (atributos públicos y protegidos y métodos públicos y protegidos) presentes en ambas clases.
- Añade todos los métodos y atributos presentes sólo en la clase derivada.

Es posible acceder a atributos de la superclase o ejecutar sus métodos mediante la palabra reservada **super.**

- Referencia a un miembro de la superclase `super.nombreMiembro()`.
- Referencia al constructor de la superclase: `super()`.

Clases y métodos abstractos

Clases en las que algunos o todos los miembros no tienen implementación, por lo que no pueden instanciarse directamente. Servirán de clase base para clases derivadas.

```
abstracta clase <clase_base>
```

Aquellos métodos sin implementación se podrían declarar sin inicio ni fin de método.

```
abstracta TipoDato: función NombreMétodo([paramFormales])
abstracto procedimiento NombreMétodo([paramFormales])
```

En estos casos, la clases hijas deberían implementar el método.

Herencia múltiple

No se considera la existencia de herencia múltiple.

A.8. PALABRAS RESERVADAS

Símbolo, palabra	Traducción/Significado
-	Menos unario (negativo)
-	Resta
&	Concatenación
←	Operador de asignación
↑	Referencia a una variable apuntada
*	Multiplicación
.	Cualificador de acceso a registros o a miembros de una clase
.	Separador de decimales
/	División real
//	Comentario de una sola línea. Ignora todo lo que aparezca a continuación de la línea
[]	Índice de array
**	Exponenciación
{	Inicio de comentario multilínea. Ignora todo lo que aparezca hasta encontrar el carácter de final de comentario (})
}	Fin de comentario multilínea. Ignora todo lo que aparezca desde el carácter de inicio de comentario ({)
'	Comilla simple, delimitador de datos de tipo carácter o cadena
"	Comilla doble, delimitador de datos de tipo carácter o cadena
+	Más unario (positivo)
+	Suma
+	Concatenación
<	Menor que
<=	Menor o igual que
<>	Distinto de
=	Igual a
>	Mayor que
>=	Mayor o igual que
abrir	Abre un archivo
abs(x)	Devuelve el valor absoluto de la expresión numérica x
abstracto	Declaración de métodos abstractos (sin implementación)
aleatorio()	Devuelve un número aleatorio real mayor o igual que 0 y menor que 1
algoritmo	Inicio del pseudocódigo
añadir	Modo de apertura de un archivo
arctan(x)	Devuelve la arco tangente de x

(continúa)

(*continuación*)

Símbolo, palabra	Traducción/Significado
archivo_d	Declaración de archivos directos
archivo_s	Declaración de archivos secuenciales
array	Declaración de arrays
borrar	Borra un archivo del disco
cadena	`string`
cadena(x)	Convierte a cadena el valor numérico x
carácter	`char`
carácter(x)	Devuelve el carácter correspondiente al código ASCII x
cerrar	Cierra un archivo
cerrar	Cierra uno o más archivos abiertos
clase	Inicio de la declaración de una clase
código(car)	Devuelve el código ASCII del carácter `car`
const	Inicio de la sección de declaraciónes de constantes
const	Declaración de atributos constantes en la definición de clases
constructor	Inicio de la declaración de un constructor
cos(x)	Devuelve el coseno de x
decremento	Decremento en estructuras repetitivas `desde`
desde	Inicio de estructura repetitiva `desde`, `for`
devolver	Indica el valor de retorno de una función
div	División entera
e	Exponente
e	Paso de argumentos por valor
e/s	Paso de argumentos por referencia
entero	`integer`, `int`, `long`, `byte`, etc.
entero(x)	Devuelve el primer valor entero menor que la expresión numérica x
entonces	`then`
escribir	Escribe una o más expresiones en un dispositivo de salida (consola, archivo, etc.)
escritura	Modo de apertura de un archivo
estático	Declaración de atributos o métodos de clase o estáticos
exp(x)	Devuelve el valor e^x
falso	Falso, `false`
fda	Fin de archivo
fdl	Fin de línea
fin	Fin de algoritmo
fin_clase	Final de la declaración de una clase
fin_constructor	Fin de la declaración de un constructor
fin_desde	Fin de estructura repetitiva `desde`
fin_función	Fin de la declaración de una función
fin_mientras	Fin de estructura repetitiva `mientras`
fin_procedimiento	Fin de un procedimiento
fin_registro	Fin de la declaración de registro
fin_según	Fin de estructura selectiva múltiple
fin_si	`end if`, fin de estructura selectiva simple
función	Inicio de la declaración de una función
hacer	`do`
hasta	`to`
hasta_que	Fin de estructura repetitiva `repetir`
hereda_de	Indica que una clase derivada hereda miembros de una superclase
incremento	Incremento en estructuras repetitivas `desde`
inicio	Inicio del código ejecutable de un algoritmo, módulo, constructor, etc.
instancia	Referencia a la instancia actual de la clase donde aparece
lda	Devuelve la longitud en bytes de un archivo
lectura	Modo de apertura de un archivo
lectura/escritura	Modo de apertura de un archivo
leer	Lee una o más variables desde un dispositivo de entrada (consola, archivo, etc.)
liberar	Libera el espacio asignado a una variable dinámica

(*continúa*)

(*continuación*)

Símbolo, palabra	Traducción/Significado
ln(x)	Devuelve el logaritmo neperiano de x
log10(x)	Devuelve el logaritmo en base 10 de x
longitud(c)	Devuelve el número de caracteres de la cadena c
llamar_a	Instrucción de llamada a un procedimiento
mientras	while, inicio de estructura repetitiva mientras
mod	Módulo de la división entera
nada	Tipo de retorno de métodos que no devuelven valores, void
no	Not
nuevo	Reserva espacio en memoria para un objeto de una clase y devuelve una referencia a dicho objeto
nulo	Constante de puntero nulo
o	Or
posición(c,sc)	Devuelve la posición de la primera aparición de la subcadena sc en la cadena c
privado	Modificador de acceso privado a un atributo o método
procedimiento	Inicio de la declaración de un procedimiento
protegido	Modificador de acceso a un atributo o método que permite el acceso a los miembros de su clase y de las clases hijas
público	Modificador de acceso público a un atributo o método
puntero_a	Declaración de tipos de datos de asignación dinámica
raiz2(x)	Devuelve la raíz cuadrada de x
real	float, double, single, real, etc.
registro	record, inicio de la declaración de registro
renombrar	Cambia el nombre de un archivo
repetir	repeat, inicio de estructura repetitiva repetir
reservar	Reserva espacio en memoria para una variable dinámica
según_sea	Inicio de estructura selectiva múltiple, case, select case, switch
sen(x)	Devuelve el seno de x
si	Inicio de estructura selectiva simple/doble, if
si_no	else
subcadena(c,ini[,long])	Devuelve una subcadena de la cadena c formada por todos los caracteres a partir de la posición ini. Si se incluye el argumento long, devuelve sólo los primeros long caracteres a partir de la posición ini
super	Permite el acceso a miembros de la superclase.
tamaño_de(x)	Devuelve el tamaño en bytes de la variable x
tipos	Inicio de la sección de declaraciónes de tipos de datos
trunc(x)	Trunca (elimina los decimales) de la expresión numérica x
valor(c)	Convierte la cadena c a un valor númérico. Si el contenido de la cadena c no puede convertirse a un valor numérico (contiene caracteres alfabéticos, signos de puntuación inválidos, etc), devuelve 0
var	Inicio de la sección de declaraciones de variables, o de la declaración de atributos de una clase
verdad	Verdadero, true
y	and

APÉNDICE B
GUÍA DE SINTAXIS C

CONTENIDO

B.1. Elementos básicos de un programa
B.2. Estructura de un programa C
B.3. El primer programa C
B.4. Palabras reservadas ANSI C
B.5. Directivas del preprocesador
B.6. Archivos de cabecera
B.7. Definición de macros
B.8. Comentarios
B.9. Tipos de datos
B.10. Variables
B.11. Expresiones y operadores
B.12. Funciones de entrada y salida
B.13. Sentencias de control
B.14. Funciones
B.15. Estructuras de datos
B.16. Cadenas
B.17. Estructuras
B.18. Uniones
B.19. Campos de bits
B.20. Punteros
B.21. Preprocesador de C

B.1. ELEMENTOS BÁSICOS DE UN PROGRAMA

El lenguaje C fue desarrollado en Bell Laboratories para su uso en investigación y se caracteriza por un gran número de propiedades que lo hacen ideal para usos científicos y de gestión.

Una de las grandes ventajas del lenguaje C es ser *estructurado*. Se pueden escribir bucles que tienen condiciones de entrada y salida claras y se pueden escribir funciones cuyos argumentos se verifican siempre para su completa exactitud.

Su excelente biblioteca estándar de funciones, convierten a C en uno de los mejores lenguajes de programación que los profesionales informáticos pueden utilizar.

B.2. ESTRUCTURA DE UN PROGRAMA C

Un programa típico en C se organiza en uno o más *archivos fuentes* o *módulos*. Cada archivo tiene una estructura similar con comentarios, directivas de preprocesador, declaraciones de variables y funciones y sus definiciones. Normalmente se sitúan cada grupo de funciones y variables relacionadas en un único archivo fuente. Dentro de cada archivo fuente, los componentes de un programa suelen colocarse en un determinado modo estándar. La Figura B.1 (página siguiente) muestra la organización típica de un archivo fuente en C.

Figura B.1. Organización de un programa C.

Los componentes típicos de un archivo fuente del programa son:

1. El archivo comienza con algunos comentarios que describen el propósito del módulo e información adicional tal como el nombre del autor y fecha, nombre del archivo. Los comentarios comienzan con /* y terminan con */.
2. Órdenes al preprocesador, conocidas como *directivas del preprocesador*. Normalmente incluyen archivos de cabecera y definición de constantes.
3. Declaraciones de variables y funciones visibles en todo el archivo. En otras palabras, los nombres de estas variables y funciones se pueden utilizar en cualquiera de las funciones de este archivo. Si se desea limitar la visibilidad de las variables y funciones *sólo* a ese módulo, ha de poner delante de sus nombres el prefijo `static`; por el contrario, la palabra reservada `extern` indica que los elementos se declaran y definen en otro archivo.
4. El resto del archivo incluye definiciones de las funciones (su cuerpo). Dentro de un cuerpo de una función se pueden definir variables que son locales a la función y que sólo existe en el código de la función que se está ejecutando.

B.3. EL PRIMER PROGRAMA C ANSI

```
#include <stdio.h>
int main ()
{
  printf ("¡Hola mundo!");
  return 0;
}
```

B.4. PALABRAS RESERVADAS ANSI C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

B.5. DIRECTIVAS DEL PREPROCESADOR

El *preprocesador* es la parte del compilador que realiza la primera etapa de traducción o compilación de un archivo C ANSI en instrucciones de máquina. El preprocesador procesa el archivo fuente y actúa sobre las órdenes, denominadas *directivas de preprocesador*, incluidas en el programa. Estas directivas comienzan con el signo de libra # (almohadilla). Normalmente, el compilador invoca automáticamente al preprocesador antes de comenzar la compilación. Se puede utilizar el preprocesador de tres formas distintas para hacer sus programas más modulares, más legibles y más fáciles de personalizar:

1. Se puede utilizar la directiva `#include` para insertar el contenido de un archivo en su programa.
2. Mediante la directiva `#define`, se pueden definir macros que permiten reemplazar una cadena por otra. Se puede utilizar la directiva `#define` para dar nombres significativos a constantes numéricas, mejorando la legibilidad de sus archivos fuente.
3. Con directivas tales como `#if`, `#ifdef`, `#else` y `#endif`, pueden compilar sólo partes de su programa. Se puede utilizar esta característica para escribir archivos fuente con código para dos o más sistemas, pero compilar sólo aquellas partes que se aplican al sistema informático en que se compila el programa.

B.6. ARCHIVOS DE CABECERA

Directivas tales como `#include <stdio.h>` indica al compilador que lea el archivo *stdio.h* de modo que sus líneas se sitúan en la posición de la directiva. C ANSI soporta dos formatos para la directiva `#include`:

1. `#include <stdio.h>`
2. `#include "demo.h"`

El primer formato de `#include` lee el contenido de un archivo —el archivo estándar de C, *stdio.h*—. El segundo formato lee el archivo encerrado entre las dobles comillas que está en el directorio actual.

B.7. DEFINICIÓN DE MACROS

Una macro define un símbolo equivalente a una parte de código C y se utiliza para ello la directiva `#define`. Se puede representar constantes tales como `PI`, `IVA` y `BUFFER`.

```
#define PI 3.14159
#define IVA 16
#define BUFFER 1024
```

que toman los valores `3.14159`, `16` y `1024`, respectivamente. Una macro también puede aceptar un parámetro y reemplazar cada ocurrencia de ese parámetro con el valor proporcionado cuando la macro se utiliza en un programa. Por consiguiente, el código que resulta de la expansión de una macro puede cambiar dependiendo del parámetro que se utilice cuando se ejecuta la macro. Por ejemplo, la macro siguiente acepta un parámetro y expande a una expresión diseñada para calcular el cuadrado del parámetro

```
#define cuadrado (x) ((x)*(x))
```

B.8. COMENTARIOS

El compilador ignora los comentarios encerrados entre los símbolos /* y */.

```
/* Mi primer programa */
```

Se pueden escribir comentarios multilínea:

```
/* Mi segundo programa C
escrito el día 15 de Agosto de 2000
en Carchelejo - Jaén - España */
```

Los comentarios no pueden anidarse. La línea siguiente no es legal:

```
/*Comentario /* comentario interno */ externo */
```

B.9. TIPOS DE DATOS

Los tipos de datos básicos incorporados a C son *enteros*, *reales* y *carácter*.

Tabla B.1. Tipos de datos enteros

Tipo de dato	Tamaño en bytes	Tamaño en bits	Valor mínimo	Valor máximo
signed char	1	8	–128	127
unsigned char	1	8	0	255
signed short	2	16	–32.768	32.767
unsignet short	2	16	0	65.535
signed int	2	16	–32.768	32.767
unsigned int	2	16	0	65.535
signed long	4	32	–2.147.483.648	2.147.483.647
unsigned long	4	32	0	4.294.967.295

El tipo char se utiliza para representar caracteres o valores integrales. Las constantes de tipo char pueden ser caracteres encerrados entre comillas ('A', 'b', 'p'). Caracteres no imprimibles (tabulación, avance de página...) se pueden representar con secuencias de escape ('\t', '\f').

Tabla B.2. Secuencias de escape

Carácter	Significado	Código ASCII
\a	Carácter de alerta (timbre)	7
\b	Retroceso de espacio	8
\f	Avance de página	12
\h	Nueva línea	10
\r	Retorno de carro	13
\t	Tabulación (horizontal)	9
\v	Tabulación (vertical	11
\\	Barra inclinada	92
\?	Signo de interrogación	63
\'	Comilla	39
\"	Doble comilla	34
\nnn	Carácter cuyo ordinal es el número octal	–
\xnn	Carácter cuyo ordinal es el número hexadecimal	–
'\0'	Carácter nulo (terminación de cadena)	–

Tabla B.3. Tipos de datos en coma flotante

Tipo de dato	Tamaño en bytes	Tamaño en bits	Valor mínimo	Valor máximo
`float`	4	32	3.4E-38	3.4E+38
`double`	8	64	1.7E-308	1.7E+308
`long double`	10	80	3.4E-4932	1.1E+4932

Todos los números sin un punto decimal en programas C se tratan como enteros y todos los números con un punto decimal se consideran reales de coma flotante de doble precisión. Si se desea representar números en base 16 (hexadecimal) o en base 8 (octal), se precede al número con el carácter `'OX'` para hexadecimal y `'0'` para octal. Si se desea especificar que un valor entero se almacena como un entero largo se debe seguir con una `"L"`.

```
025   /* octal 25 o decimal 21 */
0x25  /* hexadecimal 25 o decimal 37 */
250L  /* entero largo 250 */
```

B.10. VARIABLES

Todas las variables en C se declaran o definen antes de que sean utilizadas. Una *declaración* indica el tipo de una variable. Si la declaración produce también almacenamiento (se inicia), entonces es una *definición*.

B.10.1. Nombres de variables en C

Los nombres de variables en C constan de letras, números y carácter subrayado. Pueden ser mayúsculas o minúsculas o una mezcla de tamaños. El tamaño de la letra es significativo; las variables siguientes son *todas diferentes*.

```
Temperatura          TEMPERATURA          temperatura
```

A veces se utilizan caracteres subrayados y mezcla de mayúsculas y minúsculas para aumentar la legibilidad:

```
Dia_de_Semana     DiaDeSemana     Nombre_ciudad     PagaMes

int x;                  /* declara x variable entera  */
char nombre, conforme;  /* declara nombre, conforme de tipo char  */

int x, no = 0;          /* definen las variables x y no  */
float total = 42.125;   /* define la variable total  */
```

Se pueden declarar variables múltiples del mismo tipo de dos formas: así una declaración

```
int v1; int v2; int v3; int v4;
```

o bien

```
int v1;
int v2;
int v3;
int v4;
```

pudiéndose declarar también de la forma siguiente:

```
int v1, v2, v3, v4;
```

C no soporta tipos de datos lógicos, pero mediante enteros se pueden representar: 0 significa *falso*; distinto de cero significa *verdadero* (cierto).

La palabra reservada `const` permite definir determinadas variables con valores constantes, que no se pueden modificar. Así, si se declara

```
const int z = 4350;
```

y si se trata de modificar su valor

```
z = 3475
```

el compilador emite un mensaje de error similar a *Cannot modify a const object in function main* («No se puede modificar un objeto const en la función main»). Las variables declaradas como `const` pueden recibir valores iniciales, pero no pueden modificarse su valor con otras sentencias.

B.10.2. Variables tipo `char`

Las variables de tipo `char` (carácter) pueden almacenar caracteres individuales. Por ejemplo, la definición

```
char car = 'M';
```

declara una variable `car` y le asigna el valor ASCII del carácter `M`. El compilador convierte la constante carácter `'M'` en un valor entero (`int`), igual al código ASCII de `'M'` que se almacena a continuación en el byte reservado para `car`.

Dado que los caracteres literales se almacenan internamente como valores `int`, se puede cambiar la línea.

```
char car;
```

por

```
int car;
```

y el programa funcionará correctamente.

B.10.3. Constantes de cadena

Las cadenas de caracteres constan de cero o más caracteres separados por dobles comillas. La cadena se almacena en memoria como una serie de valores ASCII de tipo `char` de un solo byte y se termina con un byte cero, que se llama carácter nulo:

```
"Sierra Mágina en Jaén"
```

Además de los caracteres que son imprimibles, se pueden guardar en constantes cadena, códigos de escape, símbolos especiales que representan códigos de control y otros valores ASCII no

imprimibles. Los códigos de escape se representan en la Tabla B.2 como un carácter único, almacenado internamente como un valor entero y compuesto de una barra inclinada seguida por una letra, signo de puntuación o dígitos octales o hexadecimales. Por ejemplo, la declaración

```
char c = '\n';
```

asigna el símbolo nueva línea a la variable `c`. En los PC, cuando se envía un carácter `'\n'` a un dispositivo de salida, o cuando se escribe `'\n'` en un archivo de texto, el símbolo nueva línea se convierte en un retorno de carro y un avance de línea.

B.10.4. Tipos enumerados

El tipo `enum` es una *lista ordenada* de elementos como constantes enteras. A menos que se indique lo contrario, el primer miembro de un conjunto enumerado de valores toma el valor 0, pero se pueden especificar valores. La declaración:

```
enum diasSemana {Lunes, Martes, Miercoles, Jueves, Viernes,
                 Sabado, Domingo };
```

significa que `Lunes = 0`, `Martes = 1`, etc. Sin embargo, si se hace `Viernes = 10`, entonces `Lunes` sigue siendo `0`, `Martes` es igual a `2`, etc.; pero ahora `Viernes = 11`, `Sabado = 12`, etc.

Un tipo enumerado se puede utilizar para declarar una variable

```
enum diasSemana Laborable;
```

y a continuación utilizarla con

```
Laborable = Jueves;
```

o bien

```
Laborable = Sabado;
if (Laborable >= Viernes)
  printf ( "Hoy no es laborable \n",Laborable);
```

B.11. EXPRESIONES Y OPERADORES

Las expresiones son operaciones que realiza el programa.

```
a+b+c;
```

Tabla B.4. Operadores aritméticos

Operador	Descripción	Ejemplo
*	Multiplicación	(a * b)
/	División	(a / b)
+	Suma	(a + b)
-	Resta	(a - b)
%	Módulo	(a % b)

Tabla B.5. Operadores relacionales

Operador	Descripción	Ejemplo
<	Menor que	(a < b)
<=	Menor que o igual	(a <= b)
>	Mayor que	(a > b)
>=	Mayor o igual que	(a >= b)
==	Igual	(a == b)
!=	No igual	(a != b)

Tabla B.6. Operadores de incremento y decremento

Operador	Descripción	Ejemplo
++	Incremento en i	++i, i++
--	Decremento en i	--j, j--

Ejemplos

```
i = i+1;      /*Sumar uno a i  */
i++;          /*Igual que anterior  */

i = i-1:      /*Resta uno a i  */
i--;          /*Igual que anterior  */
```

Tabla B.7. Operadores de manipulación de bits

Operador	Descripción	Ejemplo
&	**AND** bit a bit	C = A & B;
|	**OR** inclusiva bit a bit	C = A | B;
^	**OR** exclusiva bit a bit	C = A ^ B;
<<	Desplazar bits a izquierda	C = A << B;
>>	Desplazar bits a derecha	C = A >> B;
~	Complemento a uno	C = ~B

B.11.1. Operadores de asignación

Los operadores de asignación son binarios y combinaciones de operadores y del signo = utilizado para abreviar expresiones:

```
A = B          /* asigna el valor de B a A  */
C = (A = B)    /* C y A son iguales a B  */
C = A = B      /* asigna B a A y a C  */
```

A = A + 45;	*equivale a*	A + = 45;

El compilador puede generar código más eficiente, recurriendo a operadores de asignación compuestos del tipo `* =`, `+ =`, etc. Cada operador compuesto (*op*) reduce la expresión en pseudocódigo:

```
a = a op b
```

a la forma abreviada

```
a op = b;
```

Tabla B.8. Operadores de asignación

Operador	Descripción		Ejemplo
`=`	Operación de asignación simple		`a = b;`
`*=`	`z *= 10;`	*equivale a*	`z = z * 10;`
`/=`	`z /= 5;`	*equivale a*	`z = z / 5;`
`%=`	`z %= 2;`	*equivale a*	`z = z % 2`
`+=`	`z += 4;`	*equivale a*	`z = z + 4;`
`-=`	`z -= 5;`	*equivale a*	`z = z - 5;`
`<<=`	`z <<= 3;`	*equivale a*	`z = z << 3;`
`>>=`	`z >>= 4;`	*equivale a*	`z = z >> 3;`
`&=`	`z &= j;`	*equivale a*	`z = z & j;`
`^=`	`z ^= j;`	*equivale a*	`z = z ^ j;`
`|=`	`z |= j;`	*equivale a*	`z = z | j;`

B.11.2. Operador serie

El operador en serie, la coma, indica una serie de sentencias ejecutadas de izquierda a derecha. Se utiliza normalmente en bucles `for`. Por ejemplo:

```
for (cuenta=1; cuenta<100; ++cuenta, ++lineasporpagina);
```

produce el incremento de la variable `cuenta` y de la variable `lineasporpagina` cada vez que se ejecuta el bucle (se realiza una iteración).

B.11.3. Prioridad (precedencia) de operadores

Las expresiones C constan de diversos operandos y operadores. En expresiones complejas, las subexpresiones con operadores de prioridad (precedencia) más alta se evalúan antes que las subexpresiones con operadores de menor prioridad.

Tabla B.9. Orden de evaluación y prioridad de operadores (asociatividad)

Nivel	Operadores	Orden de evaluación
1	`( ) . [ ] >`	izquierda-derecha
2	`* & ~ ! ++ -- +-` *(conversión de tipo sizeof)*	derecha-izquierda
3	`* / %`	izquierda-derecha

(*continúa*)

Tabla B.9. *(continuación)*

Nivel	Operadores	Orden de evaluación
4	+ -	izquierda-derecha
5	<< >>	izquierda-derecha
6	< <= > >=	izquierda-derecha
7	= = !=	izquierda-derecha
8	&	izquierda-derecha
9	^	izquierda-derecha
10	|	izquierda-derecha
11	&&	izquierda-derecha
12	||	izquierda-derecha
13	?:	derecha-izquierda
14	= *= /= += -= %= <<= >>= &= ^= |=	derecha-izquierda
15	,	izquierda-derecha

B.12. FUNCIONES DE ENTRADA Y SALIDA

Las funciones `printf()` y `scanf()` permiten comunicarse con un programa. Se denominan funciones de E/S. `printf()` es una función de salida y `scanf()` es una función de entrada y ambas utilizan una cadena de control y una lista de argumentos.

B.12.1. `printf`

La función `printf` escribe en el dispositivo de salida, los argumentos de la lista de argumentos. Requiere el archivo de cabecera *stdio.h*. La salida de `printf` se realiza con formato y su formato consta de una cadena de control y una lista de datos o variables.

```
printf (cadena de control [, item1, item2,...itemn]);
```

El primer argumento es la *cadena de control* (o formato); determina el formato de escritura de los datos. Los argumentos restantes son los datos o variables de datos a escribir

```
printf ("Esto es una prueba %d\n", prueba);
```

La cadena de control tiene tres componentes: texto, identificadores y secuencias de escape. Se puede utilizar cualquier texto y cualquier número de secuencias de escape. El número de identificadores ha de corresponder con el número de variables o valores a escribir.

Los identificadores de la cadena de formato determinan como se escriben cada uno de los argumentos.

```
printf ("Mi pueblo favorito es Cazorla%s", msg);
```

cada identificador comienza con un signo porcentaje (`%`) y un código que indica el formato de salida de la variable.

Tabla B.10. Códigos de identificadores

Identificador	Formato
%d	Entero decimal
%c	Carácter simple
%s	Cadena de caracteres
%f	Coma flotante (decimal)
%e	Coma flotante (notación exponencial)
%g	Usa el %f o el %e más corto
%u	Entero decimal sin signo
%o	Entero octal sin signo
%x	Entero hexadecimal sin signo

Las secuencias de escape son las indicadas en la Tabla B.2.

```
printf ("Mi flor favorita es la %s\n", msg);
printf ("La temperatura es %f grados centigrados \n",
                    centigrados);
```

Ejemplo B.1

```
#include <stdio.h>
#define PI 3.141593
#define SIERRA "Sierra Mágina"
int main (void)
{
  printf ("El valor de pi es %f.\n", PI);
  printf ("/%s/ \n", SIERRA);
  printf ("/%25s/ \n", SIERRA);
  return 0;
}
```

La salida producida al ejecutar el programa es

```
El valor de pI es 3.141593.
/Sierra Magina/
/          SierraMagina/
```

B.12.2. scanf

La función `scanf()` *es la función de entrada con formato. Esta función se puede utilizar para* introducir números con formato de máquina, caracteres o cadena de caracteres, a un programa:

```
scanf ("%f", &fahrenheit);
```

El formato general de la función `scanf()` es una cadena de formato y uno o más variables de entrada. La cadena de control consta sólo de identificadores.

Tabla B.11. Identificadores de formato de `scanf`

Identificador	Formato
`%d`	Entero decimal
`%c`	Carácter simple
`%s`	Cadena de caracteres
`%f`	Coma flotante
`%e`	Coma flotante
`%lf`	Doble precisión
`%u`	Entero decimal sin signo
`%o`	Entero octal sin signo
`%x`	Entero hexadecimal sin signo
`%h`	Entero corto

Un ejemplo típico de uso de `scanf` es

```
printf ("Introduzca ciudad y provincia : ");
scanf ("%s %s", ciudad, provincia);
```

Otros ejemplos de uso de `scanf`:

```
scanf ("%d", &cuenta);
scanf ("%s%*c", direccion)
scanf ("%d%d", &r, &c);
scanf ("%lb%bf", &x, &y);
```

B.13. SENTENCIAS DE CONTROL

Una sentencia consta de palabras reservadas, expresiones y otras sentencias. Cada sentencia termina con un punto y coma (;).

Un tipo especial de sentencia, la *sentencia compuesta* o *bloque*, es un grupo de sentencias encerradas entre llaves ({...}). El cuerpo de una función es una sentencia compuesta. Una sentencia compuesta puede tener variables locales.

B.13.1. Sentencia `if`

1.
```
if (expresión)
      sentencia;
```
o bien
```
if (expresión) sentencia;
```

2. Si la sentencia es compuesta

```
if (expresión) {
      sentencia1;
      sentencia2;
      ...
}
```
o bien
```
if (expresión)
{
      sentencia1;
      sentencia2;
      ...
}
```

3.
```
if (expresión == valor)
      sentencia;
```

4.
```
if (expresión != 0)
      sentencia;
```
equivale a
```
if (expresión)
      sentencia;
```

Nota

`(expresión != 0)` y `(expresión)` son equivalentes, ya que cualquier valor distinto de cero representa cierto.

B.13.2. Sentencia `if-else`

1.
```
if (expresión)
        sentencia1;
else
        sentencia2;
```

2.
```
if (expresión) {
        sentencia1;
        sentencia2;
        ...
} else{
        sentencia3;
        sentencia4;
        ...
}
```

o bien

```
if (expresión)
{
        sentencia1;
        sentencia2;
}
else
{
        sentencia3;
        sentencia4;
}
```

Sentencias `if` anidadas

1.
```
if (expresión1)
        sentencia1;
else if (expresión2)
        sentencia2;
else
        sentencia3;
```

2.
```
if (expresión1)
        sentencia1;
else if (expresión2)
        sentencia2;
else if (expresión3)
        sentencia3;
else if (expresiónN)
        sentenciaN;
else
        SentenciaPorOmision;        /* opcional */
```

B.13.3. Expresión condicional (`?:`)

Expresión condicional es una simplificación de una sentencia `if-else`. Su sintaxis es:

```
expresión1 ? expresión2 : espresión3;
```

que equivale a

```
if (expresión1)
  expresión2;
else
  expresión3;
```

Ejemplo B.2

```
if (opcion == 'S')
  premio = 1000;
else
  premio = 0;
```

equivale a

```
premio =(opcion == 'S')? 1000 : 0;
```

B.13.4. Sentencia `switch`

La sentencia `switch` realiza una bifurcación múltiple, dependiendo del valor de una expresión.

```
switch (expresión) {
case valor1:
  sentencia1;          /*se ejecuta si expresión igual a valor1 */
  break;               /* salida de sentencia switch      */
case valor2:
  sentencia2;
  break;
case valor3:
  sentencia3;
  break;
  ...
default:
  sentenciaporomision;  /* se ejecuta si ningún valor coincide con
                           expresión */
}
```

Ejemplo B.3

```
switch (op)
{
case 'a':
  func1();
  break;
case 'b':
  func2();
  break;
case 'H':
  printf ("Hola \n");
default:
  printf ("Salida \n");
};
```

B.13.5. Sentencia `while`

```
while (expresión)
    sentencia;
```

o bien

```
while (expresión) {
    sentencia1;
    sentencia2;
        ...
}
```

B.13.6. Sentencia `do-while`

```
do {
    sentencia;
    ...
} while (expresión);...
```

o bien

```
do {
    sentencia1;
    sentencia2;
} while (expresión)
```

Las sentencias `do-while` monosentencias se pueden escribir también así:

```
do sentencia; while (expresión);
```

B.13.7. Sentencia `for`

1. `for (expresión1; expresión2; expresión3) {`
   `sentencia;`
   `}`

2. `for (expresión1; expresión2; expresión3) sentencia;`

La sentencia `for` equivale a

```
expresión1;
while (expresión2) {
    sentencia;
    expresión3;
}
```

Ejemplo B.4

a)
```
for (i = 1; i <= 100; i++)
    printf ("i = = %d\n", i);
```

b)
```
for (i = 0, suma = 0; i <= 100; suma + = i, i++);
```

B.13.8. Sentencia nula

La *sentencia nula* representada por un punto y coma no hace nada. Se utilizan sentencias nulas en bucles, cuando todo el proceso se hace en las expresiones del bucle en lugar del cuerpo. Por ejemplo, localizar el byte cero que marca el final de una cadena:

```
char cad [80] = "Prueba";
int i;

for (i = 0; cad [i] != '\0'; i++);
    /* sentencia nula */
```

B.13.9. Sentencia `break`

La sentencia `break` se utiliza para salir incondicionalmente de un bucle `for`, `while`, `do-while` o de una sección `case` de una sentencia `switch`.

```
for (;;) {
   ...
   if (expresion)
      break;
}
```

B.13.10. Sentencia `continue`

La sentencia `continue` salta en el interior del bucle hasta el principio del bucle para proseguir con la *ejecución*, dejando sin ejecutar las líneas restantes después de la sentencia `continue` hasta el final del bucle. `continue` es similar a la sentencia `break`.

```
while (expresion1) {
    ...
    if (expresion2)
     continue;
     ...
}
```

B.13.11. Sentencia `goto`

Una sentencia `goto` dirige el programa a una sentencia específica que tiene etiqueta utilizada en dicha sentencia `goto`. Las etiquetas deben terminar con un símbolo dos puntos.

```
salto:
...
if (expresion) goto salto;
```

B.14. FUNCIONES

Las funciones son los bloques de construcción de programas C. Una *función* es una colección de declaraciones y sentencias. Cada programa C tiene al menos una función: la función `main`. Ésta es la función donde comienza la ejecución de un programa C. La biblioteca ANSI C contiene gran cantidad de funciones estándar.

B.14.1. Sentencia `return`

La sentencia `return` detiene la ejecución de la función actual y devuelve control a la función llamadora. La sintaxis es:

```
return expresion
```

en donde el valor de **expresión** se devuelve como valor de la función.

```
#include <stdio.h>
int main()
{
   printf ("Sierra de Cazorla y \n");
   printf ("Sierra Magina \n")N
   printf ("son dos hermosas sierras andaluzas \n");
   return 0;
}
```

B.14.2. Prototipos de funciones

En ANSI C se debe declarar una función antes de utilizarla. La declaración de la función indica al compilador el tipo de valor que devuelve la función y el número y tipos de argumentos que acepta.

El *prototipo de una función* es el nombre de la función, la lista de sus argumentos y el tipo de dato que devuelve.

```
int SeleccionarMenu(void);
double Area(int x, int y);
void salir(int estado);
```

B.14.3. El tipo `void`

Si una función no devuelve nada, ni acepta ningún parámetro, ANSI C ha incorporado el tipo de dato `void`, que es útil para declarar funciones que no devuelven nada y para describir punteros que pueden apuntar a cualquier tipo de dato.

```
void exit (int estado);
void CuentaArriba (void);
void CuentaAbajo (void*);
void memoria (int n);
```

Errores típicos de funciones

1. *Ningún retorno de valor.* La función carece de una sentencia `return`. Si una función termina sin ejecutar `return`, devuelve un valor impredecible, que puede producir errores serios.
2. *Retornos omitidos.* Si hay funciones que tienen sentencias `if`, hay que asegurarse de que existe una sentencia `return` por cada camino de salida posible.
3. *Ausencia de prototipos. Las funciones que carecen de prototipos se consideran devuelven* `int`, incluso aunque estén definidas para devolver valores de otro tipo. Como regla general, declarar prototipos para todas las funciones.
4. *Efectos laterales.* Este problema se produce normalmente por una función que cambia el valor de una o más variables globales.

B.14.4. Detener un programa con `exit`

Un medio para terminar un programa sin mensajes de error en el compilador es utilizar la sentencia

```
return valor;
```

donde *valor* es cualquier expresión entera.

Otro medio para detener un programa es llamar a `exit`. La ejecución de la sentencia

```
exit(0);
```

termina el programa inmediatamente y cierra todos los archivos abiertos.

B.15. ESTRUCTURAS DE DATOS

Los tipos complejos o estructuras de datos en C son: *arrays, estructuras, uniones, cadenas* y *campos de bits.*

B.15.1. Arrays

Todos los arrays en C comienzan con el índice `[0]`.

```
int notas[25];          /* declara array de 25 elementos */
float lista[10][25];    /* declara array de 10 por 25
                           dimensiones */
```

El número de elementos de un array se puede determinar dividiendo el tamaño del array completo por el tamaño de uno de los elementos:

```
noelementos = sizeof(lista) /sizeof (lista[0])
```

Un *array de caracteres* se puede iniciar cuando se declara con una cadena:

```
char nombre[ ] = "Sierra Magina";
```

Un *array* se puede iniciar sólo con una expresión constante en ANSI C.

```
int listamenor [2][2] = {{25, 4}, {100, 75}};
int digitos[ ] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, };
```

Un *array de punteros* a caracteres se puede iniciar con

```
char*colores[] = {"verde", "rojo", "amarillo", "rosa", "azul"};
```

B.16. CADENAS

Las *cadenas* son simplemente arrays de caracteres. Las cadenas se terminan siempre con un valor nulo (`'\0'`):

```
char cadena[80];      /* declara una cadena de 80 caracteres */
char mensaje[] = "Carchelejo está en Sierra Mágina";
char frutas[] [10] = {"naranja", "platano", "manzana"};
```

B.17. ESTRUCTURAS

Las **estructuras** son colecciones de datos, normalmente de tipos diferentes que actúan como un todo. Puede contener tipos de datos simples (carácter, float, array y enumerado) o compuestos (estructuras, arrays o uniones):

```
struct coordenada {
  int x;
  int y;
};
struct signatura {
  char titulo [40];
  char autor [30];
  int paginas;
  int anyopubli;
};
```

Una variable de tipo estructura se puede declarar así:

```
struct coordenada punto;
struct signatura librosinfantiles;
```

Para asignar valores a los miembros de una estructura se utiliza la notación punto (.):

```
punto.x = 12;
punto.y = 15;
```

Existe otro modo de declarar estructuras y variables estructuras:

```
struct coordenada {
  int x;
  int y;
} punto;
```

Para evitar tener que escribir `struct` cada vez que se declara la estructura, se puede usar `typedef` en la declaración:

```
typedef struct coordenada {
  int x;
  int y;
}Coordenadas;
```

y a continuación se puede declarar la variable `punto`:

```
Coordenadas punto;
```

y utilizar la notación punto (.):

```
punto.x = 12
punto.y = 15;
```

B.18. UNIONES

Las uniones son casi idénticas a las estructuras en su sintaxis. Las uniones proporcionan un medio de almacenar más de un tipo en una posición de memoria. Se define un tipo unión:

```
union tipodemo {
  short x;
  long l;
  float f;
};
```

y se declara una variable con

```
union tipodemo demo;
```

y se puede utilizar

```
demo.x = 345;
```

o bien

```
demo.l = 324567;
```

La unión anterior se puede iniciar así:

```
union tipodemo { short x; long l; float f; } demo = {75};
```

B.19. CAMPOS DE BITS

Los *campos de bits* se utilizan con frecuencia para poner enteros en espacios más pequeños de los que el compilador pueda normalmente utilizar y son, por consiguiente, dependientes de la implementación. Una estructura de campos de bits especifica el número de bits que cada miembro ocupa. Una declaración de un estructura `persona`:

```
struct persona {
  unsigned edad;
  unsigned sexo;
  unsigned hijos;
};
```

y de una estructura de campos de bits:

```
struct persona {
  unsigned edad: 7;     /* 0..127 */
  unsigned sexo: 1;     /0 = varon, 1 = hembra */
  unsigned hijos: 4;    /* 0..15 */
}
```

B.20. PUNTEROS

El **puntero** es un tipo de dato especial que contiene la dirección de otra variable. Un puntero se declara utilizando el asterisco (*) delante de un nombre de variable:

```
float *longitudOnda;        /* puntero a datos float */
char *indice;               /* puntero a datos char */
int *p;                     /* puntero a datos int */
```

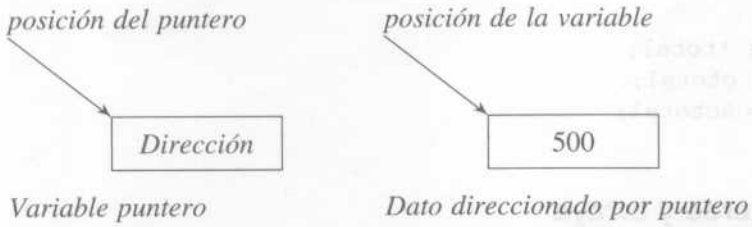


Figura B.2. Un puntero es una variable que contiene una dirección.

B.20.1. Declaración e indirección de punteros

```
int m;          /* una variable entera un */
int *p          /* un puntero p apunta a un valor entero */
p = &m;         /*se asigna a p la dirección de la variable m */
```

El operador de indirección (*) se utiliza para acceder al valor de la dirección contenida en un puntero.

```
float *longitudOnda;
longitudOnda = &cable1;
*longitudOnda = 40.5;
```

B.20.2. Punteros `nulos` y `void`

Un puntero nulo apunta a ninguna parte específica; es decir, no direcciona a ningún dato válido en memoria:

```
fp = NULL;          /* asigna Nulo a fp */
fp = 0;             /* asigna Nulo a fp */

if (fp != NULL)     /* el programa verifica si el puntero
                       es válido */
```

Al igual que una función `void` que no devuelve ningún valor, un puntero `void` apunta a un tipo de dato no especificado

```
void *noapuntafijo;
```

B.20.3. Punteros a valores

Para utilizar un puntero que apunte a un valor, se utiliza el operador de indirección (*) delante de la variable puntero:

```
double *total;
*total = 34750.75;
```

El operador de dirección (&) obtiene la dirección de la variable:

```
double *total;
double ctotal;
total = &ctotal;
```

B.20.4. Punteros y arrays

Si un programa declara

```
float *punterolista;
float listafloat [50];
```

entonces a `punterolista` se puede asignar la dirección del principio de `listafloat` (el primer elemento):

```
punterolista = listafloat;
```

o bien

```
punterolista = &listafloat [0];
listafloat[4]          es lo mismo que          *(listafloat + 4)
listafloat[2]          es lo mismo que          *(listafloat + 2)
```

B.20.5. Punteros a punteros

Los punteros pueden apuntar a otros punteros (*doble indirección*).

```
char **lista;      /* se declara un puntero a otro
                      puntero char */
```

equivale a un array de punteros a `char`

```
char *lista[];
```

y

```
char ***ptr        /* un puntero a un puntero a un puntero a char */
```

B.20.6. Punteros a funciones

```
float *ptr1;          /* ptr apunta a un valor float */
float *mifun(void);   /* la función mifunc devuelve un
                         puntero a un valor float */
```

La declaración

```
int (*ptrafuncion(void));
```

crea un puntero a una función que devuelve un entero y la declaración

```
float (*ptrafuncion1)(int x, int y);
```

declara `ptrafuncion1` como un puntero a una función que devuelve un valor `float` y requiere dos argumentos enteros.

B.21. PREPROCESADOR DE C

El preprocesador de C es un conjunto de sentencias específicas, denominadas *directivas*, que se ejecutan al comenzar el proceso de compilación.

Una directiva comienza con el símbolo # como primer carácter que indica al preprocesador que ha de ejecutarse una acción específica.

B.21.1. Directiva `#define`

Se utiliza para asociar identificadores con una secuencia de caracteres, que pueden ser una palabra reservada, una constante, una sentencia o una expresión. En el caso de que el identificador represente sentencias o expresiones se denomina *macro*.

Sintaxis **#define** <identificador>[(parámetro)] <texto>

Ejemplos

1. `#define PI 3.141592`
2. `#define cuadrado(x) x*x`
3. `#define area_circulo(r) PI*cuadrado(r)`

1. Se declara la constante `PI`. Cualquier aparición de `PI` en el programa será sustituida por el valor asociado: `3.141592`.
2. Cuando `cuadrado(x)` aparece en una línea del programa, el preprocesador la sustituye por `x*x`.
3. La tercera directiva declara una operación matemática, una fórmula, que incluye las macros `cuadrado(x)` y `PI`. Calcula el área de un círculo.

Otros ejemplos

```
#define LONGMAX 81
#define SUMA(x,y) (x)+(y)
```

El preprocesador sustituirá en el archivo fuente que contiene a estas dos macros por la constante `81` y la expresión `(x)+(y)`, respectivamente. Así:

```
float vector[LONGMAX];        define un vector de 81 elementos.
printf("%d",SUMA(5,7));       escribe la suma de 5+7.
```

Nota

Al ser una directiva del preprocesador no se pone al final ; (punto y coma).

B.21.2. Directiva `#error`

Esta directiva está asociada a la compilación condicional. En caso de que se cumpla una condición se escribe un mensaje.

Sintaxis **`#error`** `texto`

El `texto` se escribe como un mensaje de error por el preprocesador y termina la compilación.

B.21.3. Compilación condicional

La compilación condicional permite que ciertas secciones de código sean compiladas dependiendo de las condiciones señaladas en el código. Por ejemplo, el código puede desearse que no se compile si se utiliza un modelo de memoria específico o si un valor no está definido. Las directivas para la compilación condicional tienen la forma de sentencias `if`, son:

`#if, #elif, #ifndef, #ifdef, #endif.`

Directiva `#if, #elif, #endif`

Permite seleccionar código fuente para ser compilado. Se puede hacer selección simple o selección múltiple.

Formato 1

```
#if expresión_constante
. . .
#endif
```

Se evalúa el valor de la *expresión_constante*. Si el resultado es distinto de cero, se procesa todas las líneas de programa hasta la directiva `#endif`; en caso contrario se saltan automáticamente y no se procesan por el compilador.

Formato 2

```
#if expresión_constante1
. . .
#elif expresión_constante2
. . .
#elif expresión_constantenn
. . .
#else
. . .
#endif
```

Se evalúa el valor de la *expresión_constante*$_1$. Si el resultado es distinto de cero, se procesan todas las líneas de programa hasta la directiva `#elif`. En caso contrario, se evalúa la *expresión_constante*$_2$; si es distinta de cero, se procesan todas la líneas hasta la siguiente `#elif` o `#endif`. En caso contrario, se sigue evaluando la siguiente expresión, hasta que una sea distinta de cero, o bien, procesarse las líneas incluidas después de `#else`.

Ejemplo

```
#if VGA
   puts("Se está utilizando tarjeta VGA.");
#else
   puts("hardware de gráficos desconocido.");
#endif
```

Se escribe `"Se está utilizando tarjeta VGA."` si VGA es distinto de cero; en caso contrario se escribe la segunda frase.

Directiva `#ifdef`

Permite seleccionar código fuente según esté definida una macro (`#define`)

Formato

```
#ifdef identificador
. . .
#endif
```

Si `identificador` está definido previamente con `#define identificador` entonces se procesan todas las líneas de programa hasta `#endif`; en caso contrario no se compilan esas líneas. Al igual que la directiva `#if`, las directivas `#elif` y `#else` se pueden utilizar conjuntamente con `#ifdef`.

Directiva `#ifndef`

Ahora, la selección de código fuente se produce si no está definida una macro.

Formato

```
#ifndef identificador
. . .
#endif
```

Si `identificador` no está definido previamente con `#define identificador` entonces se procesan todas las líneas de programa hasta `#endif`; en caso contrario no se compilan esas líneas. Al igual que la directiva `#if`, las directivas `#elif` y `#else` se pueden utilizar conjuntamente con `#ifndef`.

Ejemplo

```
#ifndef _PAREJA
#define _PAREJA
   struct pareja {
   . . .
   }
   . . .
#endif
```

En el ejemplo, si no está definida la macro `_PAREJA` se define y se procesa el código fuente que está escrito hasta `#endif`. Se evita que se procese el código más de una vez.

B.21.4. Directiva `#include`

Permite añadir código fuente escrito en un archivo al archivo que actualmente se está escribiendo.

Formato 1 `#include` *"nombre_archivo"*
El archivo *nombre_archivo* se incluye en el módulo fuente. El preprocesador busca en el directorio o directorios especificado en el path del archivo. Normalmente se busca en el mismo directorio que contiene al archivo fuente. Una vez que se encuentra se incluye el contenido del archivo en el programa, en el punto preciso que aparece la directiva `#include`.

Formato 2 `#include` *<nombre_archivo>*
El preprocesador busca el archivo especificado sólo en el directorio establecido para contener los archivos de inclusión.

B.21.5. Directiva `#line`

Formato `#line` *constante "nombre_archivo"*
La directiva produce que el compilador trate las líneas posteriores del programa como si el nombre del archivo fuente fuera *"nombre_archivo"* y como si el número de línea de todas las líneas posteriores comenzara por *constante*. Si *"nombre_archivo"* no se especifica, el nombre de archivo especificado por la última directiva `#line`, o el nombre del archivo fuente se utiliza (si ningún archivo se especificó previamente).

La directiva `#line` se utiliza principalmente para controlar el nombre del archivo y el número de línea que se visualiza siempre que se emite un mensaje de error por el compilador.

B.21.6. Directiva `#pragma`

Formato `#pragma` *nombre_directiva*
Con esta directiva se declaran directivas que usa el compilador de C. Si al compilarse el programa con otro compilador de C, éste no reconoce la directiva, se ignora.

B.21.7. Directiva `#undef`

Formato `#undef` *identificador*
El *identificador* especificado es el de una macro definida con `#define`, con la directiva `#undef` el *identificador* se convierte en no definido por el preprocesador. Las directivas posteriores `#ifdef`, `#ifndef` se comportarán como si el identificador nunca estuviera definido.

B.21.8. Directiva

Formato `#`
Es la directiva nula, el preprocesador ignora la línea que contiene la directiva `#`.

B.21.9. Identificadores predefinidos

Los identificadores siguientes están definidos para el preprocesador:

Identificador	Significado
`--LINE--`	Número de línea actual que se compila.
`--FILE--`	Nombre del archivo actual que se compila.
`--DATE--`	Fecha de inicio de la compilación del archivo actual.
`--TIME--`	Hora de inicio de la compilación del archivo, en el formato `"hh:mm:ss"`.
`--STDC--`	Constante definida como 1 si el compilador sigue el estándar ANSI C.

APÉNDICE C
PALABRAS RESERVADAS DE C Y C++

PALABRAS RESERVADAS C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

PALABRAS RESERVADAS C++

asm	double	new	template
auto	else	operator	this
bool	enum	private	throw
break	extern	protected	try
case	float	public	typedef
catch	for	register	union
char	friend	return	unsigned
class	goto	short	using
const	if	signed	virtual
continue	inline	sizeof	void
default	int	static	volatile
delete	long	struct	while
do	namespace	switch	

➤ asm

Se utiliza para incluir directamente código ensamblador en su programa C++. El uso correcto de `asm` depende de la implementación.

Sintaxis:

```
1. asm <instruccion en ensamblador>;
2. asm <instrucción -1>; asm <instrucion-2>;...
3. asm {
        secuencia de instrucciones en ensamblador
}
```

Ejemplos:

```
asm push ds;
asm {
    pop ax
    inc ax
    push ax
}
```

➤ auto

Es un especificador de almacenamiento para variables temporales. Estas variables se crean al introducirse en un bloque de sentencias y se destruyen al salir de él. Las variables locales de una función tienen clase de almacenamiento `auto` por defecto (*omisió*).

➤ bool

Tipo lógico (*boolean*) que toma valores verdadero (*true*) o falso (*false*) que puede contener los literales **true** y **false.** Sustituye al sistema tradicional de C que considera el valor cero como falso y distinto de cero como verdadero.

➤ break

`break` permite salir del bucle `do`, `while` o `for` más interno. También se puede utilizar para salir de una sentencia `switch`. Un ejemplo de `break` en un bucle es:

```
while (Z < 10) {
  cin >> Z;
  if (Z < 0) break;      /* salir si Z es negativo */
   printf ("Hola mundo, yo sigo");
}
```

➤ case

Sirve para etiquetar los diferentes casos de la sentencia `switch`.

Sintaxis:

```
case <valor> : <sentencia>;
...
break;
```

Ejemplo:

```
switch(numero)
{
   case 2 + 5: cout << "Es 7"; break;
   case 9 : cout << "Es 9"; break;
   default : printf ("N: 7 ni 9");
}
```

➤ catch

Se utiliza como mecanismo de tratamiento de excepciones. Su propósito es interceptar una excepción generada por `throw`.

Sintaxis:

```
catch (<excepcion> {
          // código que manipula excepciones
}
```

➤ char

Tipo de dato incorporado en C. Se puede utilizar `char` para declarar variables carácter (tienen un rango de 0 a 255 en código ASCII).

➤ class

Palabra que representa la definición de una clase. Una clase contiene variables miembros (datos) y funciones miembro que operan sobre esas variables; asimismo, una clase tiene un conjunto de especificadores que controlan el acceso a los miembros de la clase (`private`, `public` y `protected`).

Sintaxis:

```
class nombre_clase: (especificadores de acceso) clase_base
{
   // miembros privados por defecto
protected:
   // miembros privados se pueden heredar
public:
   // miembros públicos
} [lista de objetos];
nombre_clase clase_1, clase_2,...;
```

Si no se especifica ningún especificador de acceso, el acceso por omisión (por defecto) es `private`. La *`lista_de_objetos`* es opcional, de modo que si no se especifica, la declaración de una clase no crea objetos de esa clase y se necesita definir el objeto como si creara una determinada variable.

➤ const

Es un calificador de tipo utilizado para indicar que la variable que le sigue no puede ser modificada por el programa. Esto significa que no se puede asignar un valor a esa variable, incrementarla o decrementarla; sin embargo, se puede inicializar a un valor dado, cuando se declara.

Sintaxis:

```
const <nombre> = <valor>;
```

Ejemplo:

```
const int edad = 46;
```

➤ continue

Se utiliza `continue` para la ejecución de un bucle. Es equivalente a ejecutar una sentencia `goto` al final del bucle. Esta sentencia afecta al bucle más interno en el que aparece.

Sintaxis:

```
continue;
```

Ejemplo:

```
for (j = 0, j < 100; j++)
{
  if (j==10) continue;
  suma += j;
}
```

➤ default

Se utiliza en la sentencia `switch` para marcar el código que se ejecutará cuando ninguna de las etiquetas de `case` se corresponde con la expresión `switch`.

Sintaxis:

```
default:<sentencia>;...
```

➤ delete

El operador de asignación dinámica `delete` se utiliza para liberar la memoria apuntada por su argumento (asignada utilizando `new`).

Sintaxis:

1. `delete <puntero>;`
2. `delete <[]> <expresión de conversión de tipo>`

Ejemplo:

```
delete objeto_prueba;
delete[] indicadores;        // borra cada uno de los
                             // elementos indicadores
```

➤ do

Se utiliza un `while` para construir bucles iterativos en los cuales la/s instrucción/es del cuerpo del bucle se ejecutan hasta que la condición se evalúa a 0 (falso).

Sintaxis:

```
do {
    <sentencias>
} while (condición);
```

Como la condición se evalúa al final del bucle, las sentencias se ejecutan al menos una vez.

Ejemplo:

```
do {
   suma += j;
   j++;
} while (j <= 100);
```

➤ double

Especificador de tipo de dato `double` que declara variables y arrays de coma flotante de doble precisión.

Sintaxis:

```
double <nombre_variable>;
```

Ejemplo:

```
double*v;
int n;
printfc "n=";
scanf ("%d", fn);
v = newdouble [n];
```

➤ else

Se utiliza con `if` para controlar el flujo de ejecución con sentencias `if` cuya sintaxis es:

```
if (expresion) sentencia_1 else sentencia_2;
```

donde *sentencia_1* se ejecuta si *expresión* es distinta de `0` (verdadero) y *sentencia_2* se ejecuta si *expresión* es igual a `0` (falso).

➤ enum

Tipo de datos de valor entero que puede tomar sus valores de una lista de constantes enumerados.

Ejemplo:

```
enum dialaborable = {lunes, martes, miercoles, jueves, vienres};
enum meses = {Enero = 1, Febrero, Marzo, Abril, Mayo, Junio};
enum dialaborable d;
enum meses m[2];
```

➤ extern

Especificador de clase de almacenamiento utilizado para indicar al compilador que una variable se declara en otra parte del programa. Si una declaración de variable comienza con la palabra reservada `extern` no es una definición, es decir, especifica el tipo y nombre de la variable e implica que una definición de esta variable se encuentra en otra parte del programa. Sin esta palabra reservada, es una definición; cada definición de variable es al mismo tiempo declaración de variable.

```
extern int n;        /* declaración de n (no definición) */
```

Este especificador se suele utilizar (en archivos compilados) separadamente, que comparten los mismos datos globales y se enlazan juntos. Así, por ejemplo, si `test` se declara en otro archivo, como entero, la declaración siguiente se utilizará en otros archivos:

```
extern int test;
```

No utilice `extern` con frecuencia. Existen otros métodos para pasar información entre funciones. En general, utilice `extern` sólo para referirse a variables globales que se definen en otra parte. Debido a que esta característica crea espacio permanente para las variables, la mayoría de las variables locales no son `extern`.

➤ extern "c"

El prefijo `extern "c"` permite que un programa C++ pueda llamar expresamente a una función C. Si se declara una función externa, un compilador C++ supondrá que es una función C y no producirá errores de compilación. La solución es especificar el enlace como `"c"` cuando se declara la función:

```
// función C
extern "c" void Mensaje(void)
```

Si existen diferentes funciones se pueden declarar en una sola declaración entre llaves:

```
extern "c" {
  void Mensaje(void);
  void Listo(void);
}
```

Estas definiciones se agruparán en un archivo de cabecera, pero no se puede declarar en otra parte dentro del archivo de cabecera con enlace `"c"`.

```
extern "c" {
  // cabeceras de bibliotecas C
  #include "dibujar.h"
  #include "impresora.h"
}
```

➤ false

Literal *boolean* de valor cero.

Ejemplo:

```
bool b1;
bool b2 = false;
bool b3 (true);
int i1 = true
bool function()
{
  // ...
}
```

➤ float

`float` es un especificador de tipos de datos utilizados para declarar variables de coma flotante.

Ejemplo:

```
float p = -2.5F;
float v[] = {1.F, 2.F, 3.F, 4.F, 5.F};
```

➤ for

El bucle `for` permite inicializar e incrementar variables contadores.

Sintaxis:

```
for (inicializacion; condicion; incremento) {
     <sentencias>
}
```

Si el bloque de sentencias sólo contiene una, no son necesarias las llaves (`{,}`). Si la condición es falsa, al comenzar el bucle éste no se ejecutará ni una sola vez.

Se puede omitir cualquiera de las tres expresiones de `for`, pero deben dejarse los puntos y comas (`;`). Si se omite *condición*, ésta se considerará como verdadera.

El bucle infinito es `for (;;)` y equivale a `while(1)`.

Se puede utilizar el operador coma (`,`) para poner varias expresiones en el interior de las diferentes partes de `for`:

Ejemplo:

```
for(i = 0, j = n-1; i < n; i++, j--)
  a[i] = a[j];
```

➤ friend

La palabra reservada `friend` se utiliza para garantizar el acceso a una parte privada (`private`) de una clase, por una función que no es miembro de ella y a las partes protegidas (`protected`) de una clase de la cual no se derivan. Las funciones amigas pueden ser *amigas* de más de una clase.

Ejemplo:

```
class Persona {
  // ...
public:
  // ...
  friend void funcamiga(int x, float y);
};
```

➤ goto

Produce un salto en la ejecución del programa a una etiqueta de la función actual. Su uso no está recomendado más que para situaciones excepcionales, tales como la salida directa del bucle más interior perteneciente a una serie de bucles anidados.

Sintaxis:

```
goto <etiqueta>;
...
etiqueta:
```

➤ if

La palabra reservada `if` se utiliza para ejecutar código sólo bajo ciertas condiciones. Se puede utilizar `if` sola o con `else`. Si las sentencias sólo son una, no se necesitan las llaves.

Sintaxis:

```
if (condicion)          if (condicion) {
                           <sentencia;...>
   <sentencias1>
else                    }
   <sentencia;...> else {
   <sentencias2>
                        }
```

Si la condición (expresión) se evalúa a cualquier valor distinto de 0, entonces se ejecutarán *sentencias_1* y si no, se ejecutarán *sentencias_2*.

Ejemplo:

```
1. if(a==x)        2. if(x > 1)        3. if(x > 1) {
     aux = 3;         if(y == 2)           if(x > 1) {
   aux = 5;             z == 5;              z == 5;
                                         } else
                                            z = 10;
                                         }
```

➤ inline

El especificador `inline` instruye al compilador para sustituir las llamadas a funciones con el código del cuerpo de la función. Esta sustitución se denomina *expansión en línea* . El uso de las funciones en línea incrementa el tamaño del programa objeto, pero puede aumentar la velocidad de ejecución, eliminando las operaciones auxiliares implicadas en llamadas a funciones. Las funciones `inline` son similares a las macros, aunque una función `inline` es mucho más segura. La ventaja del uso de funciones `inline` frente a las macros con argumentos está en la lista de argumentos; con las funciones `inline` hay comprobación de los tipos de los argumentos, en las macros no hay comprobación sino que simplemente se sustituyen.

Sintaxis:

1. Función en línea autónoma:

```
inline <tipo> <nombre_funcion> (<arg1>,...)
      {definicion funcion;}
```

2. Función en línea definida de una clase:

```
<tipo> <nombre_funcion> {definicion funcion;}
```

Existen dos métodos de utilizar funciones en línea:

1. Declarar una función independiente precediendo a la declaración con la palabra reservada `inline`. Se deben declarar y definir antes de que se haga cualquier llamada a función.
2. Declarar una función miembro de una clase como implícitamente *en línea* .

Ejemplo:

```
inline double Cuenta::Balance()
{
  return positivo;
}
inline int triple(int Z) {return Z * 3;}
```

➤ int

Especificador de tipo para variables y arrays de enteros. Los cualificadores `short` y `long` se pueden utilizar para declarar un entero del tamaño deseado.

Sintaxis:

```
int <nombre_variable>;
```

Ejemplo:

```
int j, x[100];
```

➤ long

Especificador de tipo de datos para declarar variables enteros, que ocupa dos veces más bytes que los enteros de tipo `int`.

➤ namespace

Define un ámbito (rango), un espacio de nombre. En el ámbito puede haber clases, funciones...

```
namespace Acarc {
  class mus {...};
  class poker {...};
  ...
}
```

➤ new

El operador `new` asigna memoria dinámica y devuelve un puntero del tipo apropiado a él.

Sintaxis:

```
var_p = new tipo;
```

`var_p` es una variable puntero que recibirá la descripción de la memoria asignada y *`tipo`* es el tipo de dato que la memoria contiene. El operador `new` asigna automáticamente memoria para contener un elemento de datos del tipo especificado. Si la asignación solicitada falla, `new` devuelve un puntero nulo.

Ejemplo:

```
1. double * q;        // asigna memoria suficiente para contener
   q = new double;    // un double
2. var_p = new tipo(inicializador);
```

Se inicializa la memoria asignada, especificando un inicializador (valor asignado).

```
3. var_p = new tipo[tamaño];
```

Asigna un array de una dimensión.

```
4. var_p = new tipo[tamaño1][tamaño2]...
```

Asigna arrays multidimensionales.

➤ operator

En C++ se pueden sobrecargar las funciones y los operadores. La palabra reservada `operator` se utiliza para crear funciones de operadores sobrecargados. Se pueden sobrecargar los operadores con relación a una clase.

Sintaxis:

```
nombre_clase::operator opr(lista_parametros)
```

La lista de parámetros contiene un parámetro cuando se sobrecarga un operador unitario, y dos parámetros cuando se sobrecarga un operador binario (en este caso, el operando de la izquierda se pasa en el primer parámetro, y el operando de la derecha se para en el segundo parámetro). Por ejemplo, una clase `string` puede definir el operador == como sigue:

```
class string
{
public:
   // ...
         int operator == (const string & s) const;
   // ...
};
```

➤ private

El especificador de acceso `private` se utiliza para declarar elementos privados de una clase; estos miembros no son accesibles a ninguna función distinta de las funciones miembro de esa clase.

Sintaxis:

```
class nombre {
  // ...
private:
  // ... miembros privados
};
```

Los miembros de una clase son privados por defecto.

```
class consumidor {
  char nombre[30],      // privado por omisión
  calle[40],
  ciudad[20],
  provincia[30];

public:
  void ver_datos(void);
  void leer_datos(void);
private:
  int edad, salario;      // también son privados
};
```

En declaración de clases derivadas también se utiliza `private class` *tipo_clase_derivada*`:public|private|protected)...` *tipo_clase_base*`{...}`.

En este caso, `private` provoca que todos los miembros públicos y protegidos de la clase base se vuelvan miembros privados de la clase derivada, y todos los miembros de la clase base privados permanecen en ella.

➤ protected

El especificador de acceso `protected` marca el comienzo de los miembros de la clase a los que sólo se puede acceder por los propios miembros de la clase y por las funciones miembro de todas las clases derivadas.

Sintaxis:

```
class nombre {
  // ...
protected:
  // miembros protegidos (sólo disponibles en clases derivadas)
};
```

Ejemplo:

```
class datos_act {
  char nombre[40]
  ...
  protected:
    long num_ss;
  ...
};      // num_ss es protegido

class datos_nuevos:protected datos_act {
...
};      // datos_nuevos puede acceder a num_ss
```

➤ public

El especificador de acceso `public` especifica aquellos miembros que son accesibles públicamente; cualquier función puede acceder a estos miembros.

Sintaxis:

```
class <nombre_clase> {
...
public:
  // miembros públicos, disponibles a todas las funciones
  // interiores o exteriores a la clase
};
```

Cuando se incluye el especificador `public` en una clase derivada, todos los miembros públicos de la clase base se convierten en miembros públicos de la clase derivada, y todos los miembros protegidos de la clase base se convierten en miembros protegidos de la clase derivada. En todos los casos los miembros privados de la clase base permanecen privados, es decir, no se heredan.

➤ register

`register` es un especificador de almacenamiento para tipos de datos enteros, utilizados para informar al compilador de que el acceso a los datos debe ser tan rápido como sea posible. El compilador almacenará los datos enteros en un registro de la CPU, en lugar de situarlos en memoria.

Sintaxis:

```
register <tipo> <nombre_variable>
```

Ejemplo:

```
register int j;
```

➤ return

La sentencia `return` se utiliza para detener la ejecución de la función actual y devolver el control al llamador. Si la función devuelve un valor utiliza una sentencia

```
return expresion;
```

para devolver el valor representado por la `expresion`.

Ejemplo:

```
int max(int a, int b)
{
  if(a >= b)
    return a;
  else
    return b;
}
```

➤ short

`short` es un calificador de tamaño para variables enteras con y sin signo; al menos ocupa dos bytes, `short` significa en realidad `signed short int`.

➤ signed

El modificador de tipo `signed` se utiliza para indicar que los datos almacenados en un tipo entero (`int` o `char` tienen signo). Los valores `int`, `long` y `short`, por defecto, tienen signo.
Ejemplo:

```
signed char;      /* puede tomar valores de -127 a +127 */
unsigned char;    /* puede tomar valores de 0 a 255 */
```

➤ sizeof

El operador `sizeof` determina el número de bytes que se utilizan para almacenar una variable particular o tipo de datos. Así, cuando se aplica a una variable, `sizeof` devuelve el tamaño del objeto referenciado, y cuando se aplica a un tipo, `sizeof` devuelve el tamaño total de un objeto de ese tipo.

```
int j;
longitud = sizeof(int);        /* valor 2 */
printf ("%d", sizeof j);       /* visualiza 2*/
```

➤ static

`static` es un modificador de tipo de datos que instruye al compilador para crear almacenamiento permanente para la variable local que le precede. Si se declara una variable de tipo `static`, tendrá almacenamiento permanente y retiene su valor a lo largo de toda la vida del programa (y, por consiguiente, entre llamadas a funciones).

`static` se utiliza también para ocultar datos y funciones de otros módulos o archivos.

Sintaxis:

```
static <tipo> <nombre_variable>
```

Ejemplo:

```
static int indice = 0;
```

➤ struct

La palabra reservada `struct` se utiliza para representar el tipo de datos estructura que reagrupa variables y/o funciones.

Sintaxis:

```
struct nombre_estructura {
    tipo elemento1;
    tipo elemento2;
    ...
};
struct nombre_estructura e1,e2;

struct empleado {
    int id;
    char nombre[40];
    void leer_infor(int i, char *n);
}
empleado e;     /* declara empleado e */
```

➤ switch

La sentencia `switch` se utiliza para realizar una bifurcación múltiple, dependiendo del valor de una expresión.

Sintaxis:

```
swtich (expresion)
{
  case <valor> : sentencia;...
  ...
  default : sentencia;...
};
```

Se debe utilizar `break` para separar el código de una etiqueta de `case` de otra. Una etiqueta `default` marca el código que se ejecutará si ninguna de las etiquetas de `case` se corresponde con el valor de la expresión.

Ejemplo:

```
switch(x) {
   case 'A':
          cout << "caso A\n";
          break;
   case 'B':
   case 'C':
          cout << "caso C\n";
          break;
   default:
          cout << "fin\n";
          break;
};
```

➤ template

La palabra reservada `template` se utiliza para crear funciones y clases genéricas. Define una plantilla definirá una familia de clases o funciones. Por ejemplo, una plantilla de clases para una clase `Pila` permitirá crear pilas de diferentes tipos de datos, tales como `int`, `float` o `char*`. En esencia, para crear una plantilla de funciones o clases, se definirá una clase o función con un *parámetro* . Por esta razón, las plantillas se conocen también como *tipos parametrizado* o *tipos genéicos* .

Una función *plantilla* adopta el formato:

```
template <class tipo de dato>
tipo nombre_función(lista de parámetros)
{
  // cuerpo de la función
}
```

El siguiente programa crea una función genérica que intercambia los valores de dos variables con independencia de sus tipos.

```
#include <iostream.h>

template <class Z>      // plantilla
void intercambio (Z &a, Z &b)
{
   Z aux;

   aux = a;
   a = b;
   b = aux;
}

main()
{
   int i = 50, j = 30;
   float x = 34.25, y = 16.45;
   intercambio(i, j);  // se intercambian enteros
   intercambio(x, y);  // se intercambian reales

   cout << "valores de i, j actuales" << i << "" << J << endl;
   cout << "valores de x, y actuales" << x << "" << y << endl;
   return 0;
}
```

Una declaración de clase genérica o clase plantilla es:

```
template <class tipo_de_dato>
class nombre_clase {
  // ...
};
```

Una instancia de una clase genérica se instanciará con:

```
nombre_clase <tipo> objeto;
```

El siguiente es un ejemplo de una clase genérica `Pila`:

```
template <class Tipo>
class Pila {
public:
    Pila();
    Boolean meter(const Tipo); // poner un elemento en la pila
    Boolean sacar(Tipo&);      // quitar un elemento de la pila
private:
    tipo Elementos[MaxElementos];   // elementos de la pila
    int la_cima;                    // cima de la pila
};
```

Instancias del tipo `Pila` son:

```
Pila <int> pila_ent;          // pila de enteros
Pila <char> pila_car;         // pila de caracteres
```

La sección de implementación de la `Pila` se especifica de un modo similar utilizando la construcción `template`. Ésta se utiliza para especificar cuál es el nombre que actúa como un parámetro tipo antes de especificar la declaración de cada función miembro.

```
template <class Tipo>
Pila <Tipo> :: Pila()
{
  la_cima = -1;
}

template <class Tipo>
Boolean Pila <Tipo> :: meter(const Tipo item)
{
  if(la_cima <MaxElementos-1) {
        Elementos[++la_cima] = item;
        return TRUE;
  } else {
        return FALSE;
  }
}

template <class Tipo>
Boolean Pila <Tipo>:: quitar(Tipo& item)
{
  if(la_cima < 0) {
        return FALSE;
  } else {
        item = Elementos[la_cima --];
        return TRUE;
  }
}
```

Una clase parametrizada puede tener muchos tipos como parámetros. Por ejemplo, una clase `Pila` puede tener dos tipos parametrizado utilizando

```
template <class Tipo1, class Tipo2>
  class Pila {
  ...
}
```

➤ this

`this` es un puntero al objeto actual, que significa «la dirección del objeto para el cual esta función fue llamada».

Sintaxis:

```
this
```

Ejemplo:

```
return this;
return *this;
```

Si se desea devolver el propio objeto, en lugar de un puntero al mismo, se debe utilizar `*this`.

➤ throw

La sentencia `throw` se utiliza para llamar al mecanismo de tratamiento de excepciones.

➤ try

Indica el comienzo de un bloque de manejadores de excepciones. Un bloque `try` tiene el formato:

```
try
      sentencia compuesta
      lista de manejadores
void test()
{
    try {
          ...
          throw i;
    }
    catch(int n)
    {
    if (i > o)      // maneja valores positivos
          ...
          return;
    }
    else {          // maneja parcialmente i <= 0
          ...
          throw;    // rethrow
    }
}
```

➤ true

Valor verdadero (cierto). Uno de los dos valores enumerados del tipo `bool`. El literal `true(1)` se representa por un valor distinto de cero (normalmente 1).

```
enum Boolean { FALSE, TRUE };

bool boolean;
int i1 = true;
bool b3 (true);
```

➤ typedef

`typedef` se utiliza para dar un nuevo nombre a un tipo de dato existente. Esta característica mejora la legibilidad de un programa.

Sintaxis:

```
typedef tipo_existente nuevo_nombre;
typedef float real;
typedef struct {
  float x, y;
{ punto;
```

➤ union

Tipo de dato que se utiliza para asignar almacenamiento de diferentes elementos de datos en la misma posición. La declaración de `union` es la misma que la de `struct`, con la diferencia de que en una unión todos los elementos de datos de la declaración comparten la misma posición de almacenamiento.

Sintaxis:

```
union <nombre_union>
{
  <tipo> <nombre_miembro>
  ...
}
```

C++ dispone también de una unión anónima, cuyo formato es:

```
union {lista_de_miembros};
```

➤ unsigned

El calificador de tipos `unsigned` se utiliza para tipos de datos enteros (`char`, `int`, `short`, `int` y `long int`) que informa al compilador que la variable se utilizará para almacenar sólo valores no negativos. Este calificador doble el máximo valor que se pueda almacenar en esa variable.

Sintaxis:

```
unsigned <tipo entero> <nombre_variable>
```

Ejemplo:

```
unsigned char lista[1000];
unsigned t;      // equivale a unsigned int t
```

➤ using

Declaración `using` y directiva `using`. Una declaración using permite a un cliente tener acceso a todos los nombres del espacio de nombres.

```
using namespace mcd;
using LOTOL::juego;
using namespace std;
```

➤ virtual

Especificador de una función que declara una función miembro de una clase que se redefinirá por una clase derivada.

```
class B {
public:
   int i;
   virtual void imprimir_t() const
         { cout << i << "interior a B" << endl; }
};
```

➤ void

`void` es un tipo de dato que se utiliza para indicar la no existencia de un valor de retorno o argumentos en una declaración y definición de una función. Se puede utilizar también `void*` para declarar un puntero a cualquier tipo de objeto dado.

Sintaxis:

```
void func(void);
```

Ejemplo:

```
void una_funcion(void *ptr);
```

➤ volatile

El calificador de tipo `volatile` se utiliza para indicar que la variable que sigue a `volatile` se puede modificar por factores externos al control del programa. Por ejemplo, las variables que se cambian por hardware, tales como reloj de tiempo real, interrupciones u otras entradas, se declararán `volatile`.

➤ while

La sentencia `while` permite construir un bucle cuyas sentencias interiores se ejecutan hasta que una *condición* o *expresión* se hace falsa (cero).

Sintaxis:

```
while (condicion) {
    <sentencias;...>
}
```

Un bucle `while` con una sola sentencia en su cuerpo se escribe así:

```
while (condicion) sentencia;
```

Ejemplo:

```
// suma de 10 numeros
suma = 0;
while(j <= 10)
{
  suma += j
  j++;
}
j = 0;
```

BIBLIOGRAFÍA

[AHU88] Aho, Alfred V.; Hopcroft, John E. y Ullman Jeffrey D.: *Estructuras de datos y algoritmos*, Addison-Wesley, 1988.

[BRBR97] Brassard, G., y Bratley, P.: *Fundamentos de algoritmia*, Prentice-Hall, 1997.

[DALI88] Dale, N., y Lilly S.: *Pascal plus Data Structures*, McGraw-Hill, 1988.

[FRAN94] Franch Gutierrez, Xavier. *Estructura de datos. Especificación, diseño e implementación,* Ediciones UPC, 1994.

[GIFOR01] Gilberg, Richard F. y Forouzan, Behrouz A.: *Data Structures: A Pseudocode Approach with C++,* Pacific Grove, USA, Brooks/Cole, 2001.

[HOSA90] Horowitz, Ellis y Sartaj, Sahni: *Data Structures in Pascal*, Third edition, New York, Computer Science Press, 1990.

[HEIL97] Heileman, Gregory L.: *Estructuras de datos, algoritmos y programación orientada a objetos,* McGraw-Hill, 1997.

[JOYA03] Joyanes Aguilar, Luis: *Fundamentos de programación*, tercera edición McGraw-Hill, 2003.

[JOZA98] Joyanes, L. y Zahonero I.: *Estructura de Datos. Algoritmos, abstracción y objetos*, McGraw-Hill, 1998.

[JZFSC99] Joyanes, L., Zahonero, I., Fernández, M., Sánchez, L., y Centenera, P.: *Estructuras de datos. Libro de problemas*, McGraw-Hill, 1999.

[KNUTH85] Knuth D. E.: *El arte de programar ordenadores. Algoritmos fundamentales, Volumen 1*, Editorial Reverte, 1985.

[KRUSE94] Kruse, Robert L.: *Data Structures and Program Design*, Prentice-Hall, 1994.

[LIPS87] Lipschutz Symour: *Estructura de datos*, McGraw-Hill, 1987.

[SHAF97] Shaffer, Clifford: *A Practical Introduction to Data Structures and Algorithm Analysis,* Upper Saddle River, Prentice Hall, 1997.

[SALM91] Salmon, William J.: *Structures and Abstractions*, M. A. Irwinm, 1991.

[SHAF92] Shaffer, Clifford A.: *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice-Hall, 1992.

[STAN95] Standish, Thomas A.: *Data Structures, Algorithms & Software Principles in C*. Reading, Massachusetts, 1995.

[TAAU86] Tanembaum Aaron, M., y Augestin, Moshe: *Data Structures Using* Pascal, Prentice-Hall, 1986.

[WIRTH86] Wirth Niklaus: *Algoritmos + Estructuras de Datos = Programas*, Ediciones del Castillo, 1986.

ÍNDICE

operaciones, 8
`exit`, 611-612
Expresión aritmética, 325-326
evaluación, 325,327, 332-333
algoritmo de, 320, 333
notaciones, 325, 329
prefija, 326
postfija, 326, 329
Expresión condicional, 607
`extern`, 506, 629
Exploración de direcciones, 396, 408
análisis, 398-399
cuadrática, 398-400, 403, 408
doble dirección, 399, 403
lineal, 397-399, 403, 408

Factor de carga, 387, 390-391, 398, 401, 404, 408
`false`, 4-5, 76, 105, 127, 209, 630
Fibonacci, 124, 126, 199, 230-231, 234
Ficheros, 200 *(véase archivos)*
abrir, 202
organización, 200
FIFO, 339, 363
`File`, 201-202, 205, 208, 222
`float`, 56, 630
Flujo, 201, 204, 539, 560-565
`for`, 609-610, 630
`friend`, 631
Función *Hash* (dispersa), 387-390, 395, 403, 408
aritmética modular, 391-392, 400, 408
mitad del cuadrado, 392
multiplicación, 393-394, 404-405
plegamiento, 391-392

Gda, 510, 541
Grafo, 507-510, 514, 540
árbol de expansión mínimo, 553-555, 557-558, 569
biconexo, 530
cadena, 563-567, 569
camino, 509, 522-523, 526-528, 534, 541, 544
bucle(ciclo), 509, 534
longitud, 509-510, 526, 528-529
mínimo, 549-552
simple, 509, 544
componentes conexas, 522-524, 530, 555, 557-558
componentes fuertemente conexas, 522-524
completo, 510
conexo, 510, 522, 534, 553
coste mínimo, 546-547, 550, 552-553, 569
definiciones, 508-509
dirigidos, 508, 511-512, 520, 522, 533, 540, 568
disperso, 512
Ford y **Fulkerson**, 562, 567, 569
flujo, 560-561
fuertemente conexo, 510, 523-524
grado, 509
de entrada, 509, 541
de salida, 509
lista de adyacencia, 510, 512-517, 521, 533-534, 541, 543, 549
matriz de adyacencia, 510-512, 514, 522, 527-530, 533, 541, 543, 545
matriz de caminos, 529-530, 543-545, 550
matriz valorada(de pesos), 511-512
no dirigidos, 508-509, 511, 522, 530, 533, 553
puntos de articulación, 530-532, 534
realización con listas de adyacencia, 515, 519
realización con matriz de adyacencia, 519
recorrido, 517, 519, 522, 534
en anchura, 517-520, 522, 534, 546
realización, 519
en profundidad, 517, 520-523, 530-531, 534
realización, 521
representación, 510
TAD, 514, 541
valorado, 508-509, 512-514, 546, 549, 551, 557

Heapsort, 186, 371, 379, 381-384
Hoare, 180

Identificador, 577
`inline`, 631
Ingeniería del software, 17
Instanciación, 242, 589
Interfaz, 12, 243, 245, 249
Iteración, 14, 128-130, 157

Jerarquía, 263, 268

Kirchoff, 560
Knuth, 167

Lenguaje algorítmico, 577
ámbito, 583
archivo, 585-587
atributo, 590
cadena, 578-579, 586
clases, 588-592
comentario, 577
constantes, 578
constructor, 589
declaración, 580-581, 586, 588, 590
destructor, 589
estructuras de control, 582
repetitivas, 583
selectivas, 582-583
funciones, 581, 584-585

aritméticas, 581
cadena, 582
conversión, 582
herencia, 591-592
método, 590-592
operadores, 578-579
palabras reservadas, 592-594
procedimiento, 584
tipos de datos, 577, 584
dinámicos, 588
estructurados, 580
variable dinámica, 588
visibilidad, 588
LIFO, 311-312, 335, 358
Ligadura, 264, 591
estática, 264
dinámica, 264
tiempo, 264
Lista circular, 273, 299, 306, 353-354
eliminar, 300
insertar, 300
operaciones, 302
TAD, 299
Lista doblemente enlazada, 273, 292, 306, 357
eliminación de nodos, 295
inserción, 294-295
Lista enlazada, 166, 186, 241, 271-273, 279, 306, 321, 350, 403,415, 513-514
acceso, 277
borrado, 274, 289
búsqueda, 287-288
especificación formal, 274
implementación con arrays, 273
implementación con punteros, 273
inserción, 274, 281, 284-286
operaciones, 275
recorrido, 275
TAD lista, 272, 274-275, 404
vacía, 274, 278-279, 282
Lista ordenada, 139, 289-290
búsqueda, 290
implementación, 290

Macro, 597, 617, 619
`malloc`, 94
Memoria, 50, 199, 257, 350
Mergesort, 139-140
algoritmo, 139
Mezcla, 139, 199
dos listas, 139
Modularidad, 239, 243, 246, 262-263, 268
Módulo, 240-242, 244, 246, 248, 252, 267
acoplamiento, 246-247
cohesión, 246-247
diseño, 246
estructura, 246
implementación, 244
interfaz, 244-245, 247
Montículo, 367, 371-372, 375, 379, 384
complejidad, 375-376, 378
condición, 373-374, 376-380
implementación, 375, 378, 381
maximal, 380-381, 384
operaciones, 374
eliminar mínimo, 377
insertar, 374, 376

`namespace`, 633
`new`, 633
Nodo, 272-275, 292-293, 416
Notación-O, 21-22, 31, 35, 41, 50
propiedades, 40, 42, 44, 383
`NULL`, 202, 204, 207, 274, 277-278, 284, 287, 295, 306, 396, 400-402, 404, 407, 425, 427, 487-488, 515

Objetos, 240-241, 261, 267
Ocultación, 9, 11, 241-245, 247, 249, 262
operador, 578, 602
`.`, 93, 105, 115
`->`, 93-94, 105, 115, 278-279
`&`, 103
`<<`, 111
`>>`, 111
aritméticos, 578
coma, 603
lógicos, 579
prioridad, 579, 603
`operator`, 634
Orden de función, 36-39, 45
Ordenación, 45, 165-167, 195
binsort, 186, 189, 195
burbuja, 167, 174, 178, 195
de archivos, 195
de listas, 195
hundimiento, 174
inserción, 167, 173, 178, 195
intercambio, 168
interna, 166-167, 213
megesort, 139, 195
por montículos, 186, 371, 379-380, 384
quicksort, 180, 186, 195
rápida, 180, 183
radixsort, 188, 195
selección, 167, 171, 178, 195
Shell, 178, 186, 195
Ordenación externa, 166-167, 199, 213
éficiencia, 221,
método polifásico, 229, 231, 234-235
mezcla directa, 214-215, 222
mezcla equilibrada, 221-223, 234-235
mezcla natural, 219-222, 235

Ordenación topológica, 540, 543, 568
algoritmo, 541
complejidad, 543
overflow, 5, 158

Palabras reservadas, 596, 623
Paradigma, 268
Parámetros
arrays, 71, 76
cadena, 76
de entrada, 25
de salida, 25
por referencia, 71, 103-104
por valor, 103
Persistencia, 243, 264-265
Pila, 312, 335, 341, 428, 517, 520-521
cima, 312-313, 315, 318, 324, 335, 349
especificación formal, 313
evaluación de expresiones aritméticas, 335
implementación, 313, 317, 322
con arrays, 313
con listas enlazadas, 313, 321
insertar, 312-313, 315, 317
notación de expresiones, 326
polaca, 326
polaca inversa, 326
postfija, 326-327, 335
prefija, 326, 335
llena, 314, 319, 321
quitar, 312-313, 315, 318, 324
sacar, 312
utilización, 312, 313
vacía, 314, 316, 318, 335
TAD, 315-316, 321-322, 327, 360, 500, 521
Polimorfismo, 263-264, 268, 591
Postcondición, 24-25
Precondición, 24-26
Preprocesador, 596-597, 617, 620-621
compilación condicional, 618
directivas, 597, 617-619
`printf`, 604-605
Problemas de programación
resolución, 121
Prototipo, 256-257, 315, 321, 352, 357, 471, 611
Pseudocódigo, 12, 14-16,27, 187
Puntero, 57, 201, 271-272, 276-277, 299, 425, 615-616
a función, 617
a puntero, 616
asignación, 257
declaración, 615
indirección, 615-616
nulo, 615
operador, 94
variable, 281, 284, 287
`void`, 615

Razón áurea, 393
Recursión, 121, 127-128, 130, 157
Recursividad, 121, 132, 141
algoritmos recursivos, 44, 129, 134, 157
caso base, 124, 127, 129-130, 132-134, 139, 158
condición salida, 123, 125, 128, 130, 134, 138
condición terminación, 128, 138, 158
directa, 121, 125, 157
divide y vencerás, 132
indirecta, 121, 125-126, 157
infinita, 125, 130, 158
problema de la selección óptima, 153-154, 158
ocho reinas, 44, 145
salto del caballo, 44, 142
Redes, 530, 533-534, 560, 563, 569
Registro, 6-7, 9, 88, 90, 166, 580
Rendimiento, 34
`return`, 610-611
Reutilización, 239, 265-266
Rotación, 463, 467, 471-472, 474-476, 478
doble, 463, 465-466, 472
propiedad, 467
simple, 463-465, 474

`scanf`, 604-606
Secuencia, 14, 201, 249
Secuencias de escape, 598, 604
Selección, 14
Sentencias, 606-607
compuesta, 606
nula, 609
Shell, 178
`sizeof`, 59, 75, 92, 108, 115, 211, 280, 636
Software, 240, 266-267, 269
ciclo de vida, 267
sistemas de, 262, 267
`static`, 596
`stdin`, 200
`stderr`, 200
`stdout`, 200
`strcpy`, 63-64
`switch`, 608, 610, 625, 639
etiquetas `case`, 625, 639
etiquetas `default`, 625, 639

Tablas de dispersión *(Hash)*, 387-389, 396, 408
operaciones, 388, 396-397, 408
buscar, 389-390, 396-398, 402, 404, 407
eliminar, 389-390, 396-398, 402, 406
insertar, 389-390, 396-398, 401
realización, 399
enlazada, 403, 408
TAD, 9-10, 12, 27, 239, 250-251, 255-256, 268, 311
especificación, 10, 252-254, 268
implementación, 9-11, 252, 255, 257, 268

interfaz, 250-252, 256, 268
operaciones, 12, 242, 250-254
ventajas, 252
`template`, 640-641
Teoría de los restos, 346-347, 363, 399, 408
`this`, 642
`throw`, 642
Tipos agregados, 6-7, 9
Tipos de datos, 4, 9, 14, 27, 55, 239, 248, 250, 577, 598
primitivos, 4, 7
numéricos, 4-5, 577
Tipos abstractos de datos (véase TAD), 4, 9-10, 239, 241-242, 245, 250-251, 256, 269
Tipos enumerados, 6, 601
Torres de Hanoi, 132, 136-137, 160
`true`, 4-5, 106, 143, 153, 209, 624, 643
`typedef`, 87, 109-110, 115, 256, 276, 613, 643

Underflow, 5
`Unicode`, 6
`union`, 104-105, 107, 614, 643
anónima, 643
tamaño, 104, 108
UPSAM, 14, 577
USB, 19, 167
`using`, 644

Variables, 271, 599-600
declaración, 599
definición, 599
dinámica, 271, 588
Vector, 55, 145
Verificación, 23-24
`virtual`, 644
Visibilidad, 241
`void`, 611, 644
`volatile`, 644

`while`, 609-610, 645

Algoritmos y estructuras de datos. Una perspectiva en C

Luis Joyanes Aguilar • Ignacio Zahonero Martínez

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.

El hecho de utilizar el lenguaje 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.

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

1. Cada estructura de datos tiene sus costes y sus beneficios. Los programadores y diseñadores necesitan una comprensión rigurosa y completa de cómo evaluar los costes y beneficios para adaptarse a los nuevos retos que afronta la construcción de la aplicación. Estas propiedades requieren un conocimiento o comprensión de los principios del análisis de algoritmos y también una consideración práctica de los efectos significativos del medio físico empleado.

2. Los temas relativos a costes y beneficios se consideran dentro del concepto de elemento de compensación.

3. 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.

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.

5. El método didáctico que sigue nuestro libro ya lo hemos seguido en otras obras nuestras. Busca preferentemente enseñar 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.

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.

9 788448 140779

http://www.mcgraw-hill.es

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

A Subsidiary of The McGraw-Hill Companies

ISBN: 84-481-4077-X