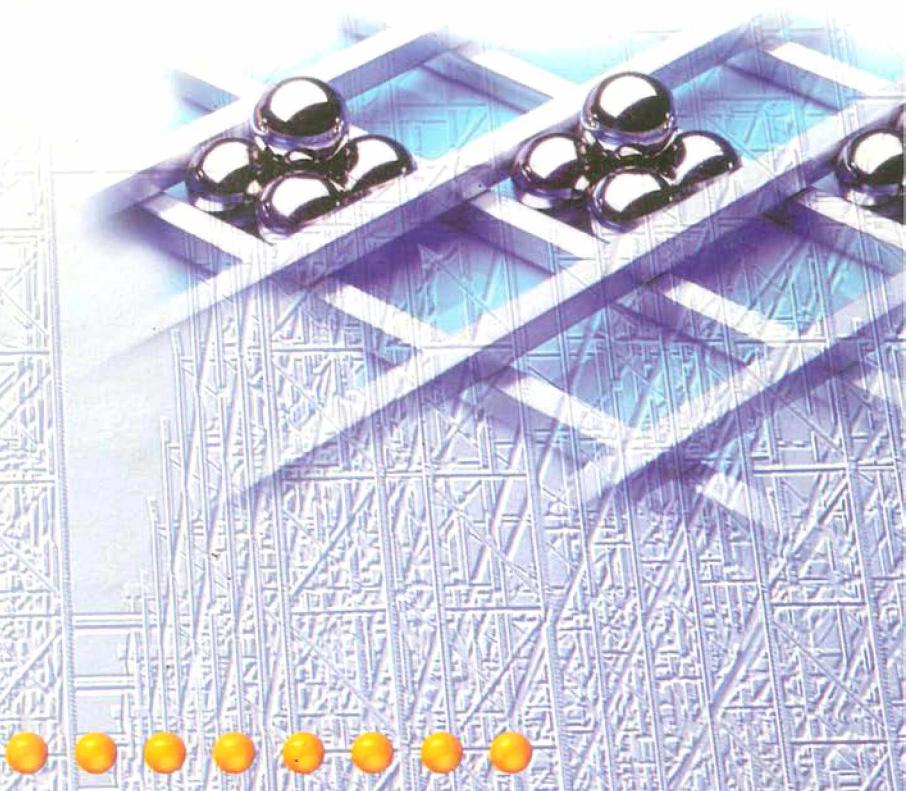


Estructura de Datos

Algoritmos, abstracción y objetos



Luis Joyanes Aguilar
Ignacio Zahonero Martínez

LIBRO DEL PROGRAMA

RTAC

DISTRIBUYE: BIS COSTA RICA TEL.: 296-3102

ESTRUCTURA DE DATOS

Algoritmos, abstracción y objetos

**CONSULTORES EDITORIALES
ÁREA DE INFORMÁTICA Y COMPUTACIÓN**

Antonio Vaquero Sánchez
Catedrático de Lenguajes y Sistemas Informáticos
Escuela Superior de Informática
Universidad Complutense de Madrid
ESPAÑA

Gerardo Quiroz Vieyra
Ingeniero en Comunicaciones y Electrónica
por la ESIME del Instituto Politécnico Nacional
Profesor de la Universidad Autónoma Metropolitana
Unidad Xochimilco
MÉXICO

ESTRUCTURA DE DATOS

Algoritmos, abstracción y objetos

**Luis Joyanes Aguilar
Ignacio Zahonero Martínez**

Departamento de Lenguajes y Sistemas Informáticos
y de Ingeniería de Software
Escuela Universitaria de Informática
Universidad Pontificia de Salamanca en Madrid



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

ESTRUCTURA DE DATOS. Algoritmos, abstracción y objetos

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

**DERECHOS RESERVADOS © 1998, respecto a la primera edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1.ª planta
Basauri, 17
28023 Aravaca (Madrid)**

ISBN: 84-481-2042-6
Depósito legal: M. 16.148-1999

Editora: Concepción Fernández Madrid
Diseño de cubierta: Design Master Dima
Compuuesto en: Puntographic, S. L.
Impreso en: Cobra, S. L.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

A los benjamines de nuestras familias, Luis y Paloma

Contenido

Prólogo	xxi
---------------	-----

PARTE I. ABSTRACCIÓN Y PROGRAMACIÓN. INTRODUCCIÓN A LA INGENIERÍA DE SOFTWARE

Capítulo 1. Ingeniería de software: introducción a la metodología de construcción de grandes programas	3
1.1. Abstracción	4
1.2. Resolución de problemas y programación	4
1.3. Herramientas para la resolución de problemas	5
1.3.1. Diseño descendente	5
1.3.2. Abstracción procedural	6
1.3.3. Abstracción de datos	7
1.3.4. Ocultación de la información	7
1.3.5. Programación orientada a objetos	7
1.4. Factores en la calidad del software	8
1.5. El ciclo de vida del software	9
1.5.1. Análisis	10
1.5.2. Diseño	11
1.5.3. Implementación (codificación)	12
1.5.4. Pruebas e integración	13
1.5.5. Verificación	13
1.5.6. Mantenimiento	14
1.5.7. La <i>obsolescencia</i> : programas obsoletos	15
1.5.8. Iteración y evolución del software	15
1.6. Métodos formales de verificación de programas	16
1.6.1. Aserciones (<i>assertos</i>)	16
1.6.2. Precondiciones y postcondiciones	17
1.6.3. Reglas para prueba de programas	17
1.6.4. Invariantes de bucles	19
1.6.5. Etapas a establecer la exactitud (corrección) de un programa	22
1.7. Principios de diseño de sistemas de software	23
1.7.1. Modularidad mediante diseño descendente	23
1.7.2. Abstracción y encapsulamiento	24
1.7.3. <i>Modificabilidad</i>	25

1.7.4.	Comprensibilidad y fiabilidad	26
1.7.5.	Interfaces de usuario	26
1.7.6.	Programación segura contra fallos.....	26
1.7.7.	Facilidad de uso	27
1.7.8.	Eficiencia	27
1.7.9.	Estilo de programación, documentación y depuración...	28
1.8.	Estilo de programación	28
1.8.1.	Modularizar un programa en subprogramas	29
1.8.2.	Evitar variables globales en subprogramas	29
1.8.3.	Usar nombres significativos para identificadores	30
1.8.4.	Definir constantes con nombres	31
1.8.5.	Evitar el uso de goto	31
1.8.6.	Uso adecuado de parámetros valor/variable.....	31
1.8.7.	Uso adecuado de funciones	32
1.8.8.	Tratamiento de errores	32
1.8.9.	Legibilidad	33
1.9.	La documentación	35
1.9.1.	Manual del usuario	35
1.9.2.	Manual de mantenimiento (documentación para programadores)	36
1.9.3.	Documentación interna	37
1.9.4.	Documentación externa	37
1.9.5.	Documentación del programa	37
1.10.	Depuración	38
1.10.1.	Localización y reparación de errores.....	39
1.10.2.	Los equipos de programación	41
1.11.	Diseño de algoritmos	42
1.12.	Pruebas (<i>testing</i>)	42
1.12.1.	Errores de sintaxis (de compilación)	44
1.12.2.	Errores en tiempo de ejecución	45
1.12.3.	Errores lógicos	45
1.12.4.	El depurador	46
1.13.	Eficiencia	46
1.13.1.	Eficiencia <i>versus</i> legibilidad (claridad)	49
1.14.	Transportabilidad	49
Resumen	49	
Ejercicios	50	
Problemas	51	
Capítulo 2. Construcción de grandes programas: módulos <i>versus</i> unidades	53	
2.1. Concepto de unidad	54	
2.2. Estructura de una unidad	54	

2.2.1.	Cabecera de la unidad	55
2.2.2.	Sección de interfaz	55
2.2.3.	Sección de implementación	56
2.2.4.	Sección de iniciación (<i>inicialización</i>)	57
2.2.5.	Ventajas de las unidades	57
2.3.	Creación de unidades	57
2.3.1.	Construcción de grandes programas	61
2.3.2.	Uso de unidades	61
2.3.3.	Declaraciones/información pública y privada	62
2.3.4.	Cláusula <i>uses</i> en la sección de implementación	63
2.4.	Utilización de unidad estándar	65
2.4.1.	Unidad <i>System</i>	66
2.4.2.	Unidad <i>Crt</i>	66
2.4.3.	Unidad <i>DOS</i>	67
2.4.4.	Unidad <i>Printer</i>	68
2.4.5.	Unidad <i>Graph</i>	68
2.5.	Situación de las unidades en sus discos: ¿dónde busca Turbo Borland Pascal las unidades?	69
2.6.	Identificadores idénticos en diferentes unidades	71
2.7.	Síntesis de unidades	71
2.7.1.	Estructura de una unidad	72
2.7.2.	Excepciones en la escritura de unidades	73
2.7.3.	Compilación de unidades	74
2.8.	Otros métodos de estructurar programas: inclusión, solapamientos y encadenamiento	74
2.8.1.	Los archivos de inclusión o incluidos (<i>include</i>)	74
2.8.2.	Los solapamientos (<i>overlays</i>)	75
2.8.3.	La unidad Overlay (generación de solapamientos)	76
2.8.4.	Encadenamiento de archivos compilados	79
	Resumen	79
	Ejercicios	79
	Referencias bibliográficas	82
Capítulo 3.	Abstracción de datos: tipos abstractos de datos y objetos	83
3.1.	El papel (el rol) de la abstracción	84
3.1.1.	La abstracción como un proceso mental natural	84
3.1.2.	Historia de la abstracción del software	85
3.1.3.	Procedimientos	86
3.1.4.	Módulos	86
3.1.5.	Tipos abstractos de datos	87
3.1.6.	Objetos	88
3.2.	Un nuevo paradigma de programación	88

3.3.	Modularidad	89
3.3.1.	La estructura de un módulo	90
3.3.2.	Reglas de modularización	91
3.4.	Diseño de módulos	94
3.4.1.	Acoplamiento de módulos	94
3.4.2.	Cohesión de módulos	95
3.5.	Tipos de datos	96
3.6.	Abstracción en lenguajes de programación	97
3.6.1.	Abstracciones de control	97
3.6.2.	Abstracciones de datos	98
3.7.	Tipos abstractos de datos (TAD)	99
3.7.1.	Ventajas de los tipos abstractos de datos	101
3.7.2.	Implementación de los TAD	101
3.8.	Tipos abstractos de datos en Turbo Borland Pascal	102
3.8.1.	Aplicación del tipo abstracto de dato <i>pila</i>	104
3.9.	Orientación a objetos	105
3.9.1.	Abstracción	107
3.9.2.	Encapsulación	107
3.9.3.	Modularidad	107
3.9.4.	Jerarquía	108
3.9.5.	Polimorfismo	108
3.9.6.	Otras propiedades	109
3.10.	Reutilización de software	110
3.11.	Lenguajes de programación orientados a objetos	111
3.11.1.	Clasificación de los lenguajes orientados a objetos	112
3.12.	Desarrollo tradicional frente a desarrollo orientado a objetos	116
3.13.	Beneficios de las tecnologías de objetos	118
	Resumen	120
	Ejercicios	120

PARTE II. FUNDAMENTOS BÁSICOS DE ESTRUCTURAS DE DATOS Y TIPOS ABSTRACTOS DE DATOS

Capítulo 4.	Estructuras de datos dinámicas: punteros	125
4.1.	Estructuras de datos dinámicas	125
4.2.	Punteros (<i>apuntadores</i>)	127
4.2.1.	Declaración de punteros	127
4.3.	Operaciones con variables puntero: procedimientos New y Dispose	130
4.3.1.	Procedimiento New	130
4.3.2.	Variables de puntero con registros	134
4.3.3.	Iniciación y asignación de punteros	135

4.3.4. Procedimiento <i>Dispose</i>	137
4.3.5. Constante <i>nil</i>	138
4.3.6. Naturaleza dinámica de los punteros	138
4.3.7. Comparación de punteros	139
4.3.8. Paso de punteros como parámetros	141
4.4. El tipo genérico puntero (<i>Pointer</i>)	142
4.5. La asignación de memoria en Turbo Borland Pascal	142
4.5.1. El montículo (<i>heap</i>) y los punteros	143
4.5.2. Métodos de asignación y liberación de memoria	145
4.5.3. <i>New</i> y <i>Dispose</i>	145
4.5.4. <i>Mark</i> y <i>Release</i>	146
4.5.5. <i>GetMem</i> y <i>FreeMem</i>	146
4.5.6. <i>MemAvail</i> y <i>MaxAvail</i>	147
Resumen	148
Ejercicios	149
Problemas	152
Capítulo 5. Listas enlazadas: el TAD <i>lista</i> enlazada	153
5.1. Especificación formal del tipo abstracto de datos <i>lista</i>	153
5.2. Implementación del TAD <i>lista</i> con estructuras estáticas	156
5.3. Implementación del TAD <i>lista</i> mediante variables dinámicas	159
5.4. Iniciar una lista enlazada	162
5.5. Búsqueda en listas enlazadas	162
5.6. Operaciones de dirección: siguiente, anterior, último	163
5.7. Inserción de un elemento en una lista	164
5.8. Supresión de un elemento de una lista	166
5.9. Recorrido de una lista	168
5.10. Lista ordenada	168
5.10.1. Implementación de una lista ordenada	168
5.10.2. Búsqueda en lista ordenada	170
Resumen	175
Ejercicios	176
Problemas	177
Capítulo 6. Listas doblemente enlazadas	179
6.1. Especificación de lista doblemente enlazada	179
6.2. Implementación de una lista doblemente enlazada mediante variables dinámicas	180
6.2.1. Creación de nodos en la lista	181
6.2.2. Eliminación de nodo	184
6.3. Una aplicación resuelta con listas doblemente enlazadas	186
6.4. Implementación de una lista circular mediante variables dinámicas	195

6.5.	Implementación de una lista circular mediante variables dinámicas	196
6.6.	Implementación de listas circulares con doble enlace	200
Resumen		203
Ejercicios		204
Problemas		204
Capítulo 7. Pilas: el TAD <i>Pila</i>.....		207
7.1.	Especificación formal del tipo abstracto de datos <i>pila</i>	207
7.2.	Implementación del TAD <i>Pila</i> con arrays	208
7.3.	Implementación del TAD <i>Pila</i> mediante variables dinámicas....	211
7.4.	Evaluación de expresiones aritméticas mediante pilas	215
7.4.1.	Notaciones Prefija (Polaca) y Postfija (Polaca inversa) ..	215
7.4.2.	Algoritmo para evaluaciones de una expresión aritmética ..	216
7.5.	Aplicación práctica de la evaluación de una expresión aritmética	218
7.5.1.	Unidad <i>Pilaop</i>	218
7.5.2.	Unidad <i>ExpPost</i> (transformación a postfija)	220
7.5.3.	Evaluación de la expresión en postfija	222
Resumen		224
Ejercicios		225
Problemas		225
Capítulo 8. Colas y colas de prioridades: el TAD <i>cola</i>		227
8.1.	Especificación formal del tipo abstracto de datos <i>cola</i>	228
8.2.	Implementación del TAD <i>cola</i> con arrays lineales	229
8.3.	Implementación del TAD <i>cola</i> con arrays circulares	231
8.4.	Implementación del TAD <i>cola</i> con listas enlazadas	235
8.5.	Implementación del TAD <i>cola</i> con listas circulares	237
8.6.	Bicolas	239
8.7.	Colas de prioridades	244
8.8.	Implementación de colas de prioridades	245
8.8.1.	Implementación mediante una única lista	245
8.8.2.	Implementación mediante una lista de n colas	246
8.9.	Implementación de un problema con colas de prioridades	247
Resumen		256
Ejercicios		257
Problemas		258

PARTE III. ESTRUCTURAS DE DATOS AVANZADAS

Capítulo 9. Recursividad: algoritmos recursivos	263	
9.1.	Recursividad	263
9.2.	Cuándo no utilizar la recursividad	264
9.2.1.	Eliminación de la recursividad	266

9.3.	Algoritmos «divide y vencerás»	267
9.3.1.	Torres de Hanoi	267
9.3.2.	Traza de un segmento	269
9.4.	Implementación de procedimientos recursivos mediante pilas.....	270
9.4.1.	El problema de las Torres de Hanoi resuelto sin recursividad.....	271
9.5.	Algoritmos de vuelta atrás (<i>backtracking</i>)	277
9.5.1.	Solución del problema «Salto del caballo» con esquema iterativo	281
9.5.2.	Problema de las ocho reinas	286
9.5.3.	Solución no recursiva al problema de las ocho reinas	288
9.5.4.	Problema de la mochila	290
9.5.5.	Problema del laberinto	293
9.5.6.	Generación de permutaciones de n elementos	296
9.6.	Problema de la selección óptima	297
9.6.1.	El viajante de comercio	298
9.7.	Problema de los matrimonios estables	298
	Resumen	306
	Ejercicios	306
	Problemas.....	308
	Capítulo 10. Árboles binarios	311
10.1.	Concepto de árbol	312
10.2.	Árboles binarios	314
10.2.1.	Terminología	315
10.2.2.	Nivel de un nodo y altura de un árbol	316
10.2.3.	Árboles binario, lleno y completo	318
10.2.4.	Recorrido de un árbol binario	320
10.3.	Árboles de expresión	320
10.4.	Construcción de un árbol binario	322
10.5.	Recorrido de un árbol	324
10.5.1.	Recorrido <i>enorden</i>	325
10.5.2.	Recorrido <i>preorden</i>	326
10.5.3.	Recorrido <i>postorden</i>	326
10.5.4.	Implementación de los algoritmos de recorrido	328
10.6.	Aplicación de árboles binarios: evaluación de expresiones	329
10.6.1.	Notación postfija: notación polaca	329
10.6.2.	Árbol de expresión	330
10.6.3.	Transformación de expresión infija a postfija	331
10.6.4.	Creación de un árbol a partir de la expresión en postfija	331
10.6.5.	Evaluación de la expresión en postfija	332
10.6.6.	Codificación del programa de evaluación	334

10.7.	Árbol binario de búsqueda	334
10.7.1	Creación de un árbol binario	336
10.8.	Operaciones en árboles binarios de búsqueda	340
10.8.1.	Búsqueda	341
10.8.2.	Inserción	341
10.8.3.	Eliminación	342
10.8.4.	Recorrido de un árbol	344
10.8.5.	Determinación de la altura de un árbol	345
	Resumen	346
	Ejercicios	347
	Problemas	349
	Referencias bibliográficas	350
 Capítulo 11. Árboles equilibrados		 353
11.1.	Eficiencia de la búsqueda en un árbol binario	353
11.2.	Árbol binario equilibrado (AVL)	354
11.3.	Creación de un árbol equilibrado de N claves	355
11.4.	Inserción en árboles equilibrados: rotaciones	356
11.4.1.	Inserción de un nuevo nodo	357
11.4.2.	Proceso a seguir: rotaciones	357
11.4.3.	Formación de un árbol equilibrado	360
11.4.4.	Procedimiento de inserción con balanceo	363
11.5.	Eliminación de un nodo en un árbol equilibrado	366
11.6.	Manipulación de un árbol equilibrado	372
	Resumen	376
	Ejercicios	377
	Problemas	377
 Capítulo 12. Árboles B		 379
12.1.	Definición de un árbol B	379
12.2.	Representación de un árbol B de orden m	380
12.3.	Proceso de creación en un árbol B	381
12.4.	Búsqueda de una clave en un árbol B	384
12.5.	Algoritmo de inserción en un árbol B de orden m	385
12.6.	Recorrido en un árbol B de orden m	389
12.7.	Eliminación de una clave en un árbol B de orden m	390
12.8.	TAD árbol B de orden m	397
12.9.	Realización de un árbol B en memoria externa	400
	Resumen	414
	Ejercicios	415
	Problemas	415

Capítulo 13. Grafos. Representación y operaciones	417
13.1. Grafos y aplicaciones.....	417
13.2. Conceptos y definiciones	418
13.2.1. Grado de entrada, grado de salida	419
13.2.2. Camino	420
13.3. Representación de los grafos	421
13.3.1. Matriz de adyacencia	421
13.3.2. Listas de adyacencia	424
13.4. TAD grafo	426
13.4.1. Realización con matriz de adyacencia	426
13.4.2. Realización con listas de adyacencia	428
13.5. Recorrido de un grafo	431
13.5.1. Recorrido en anchura	431
13.5.2. Realización del recorrido en anchura	433
13.5.3. Recorrido en profundidad	434
13.5.4. Recorrido en profundidad de un grafo	435
13.5.5. Realización del recorrido en profundidad.....	435
13.6. Componentes conexas de un grafo	437
13.7. Componentes fuertemente conexas de un grafo dirigido	439
13.8. Matriz de caminos. Cierre transitivo	442
13.9. Puntos de articulación de un grafo	446
Resumen	450
Ejercicios	451
Problemas	453
Capítulo 14. Algoritmos fundamentales con grafos	455
14.1. Ordenación topológica	456
14.2. Matriz de caminos: algoritmo de Warshall	459
14.3. Problema de los caminos más cortos con un solo origen: algoritmo de Dijkstra	461
14.3.1. Algoritmo de la longitud del camino más corto	462
14.4. Problema de los caminos más cortos entre todos los pares de vértices: algoritmo de Floyd	467
14.4.1. Recuperación de caminos	469
14.5. Concepto del flujo	470
14.5.1. Planteamiento del problema	471
14.5.2. Formulación matemática	472
14.5.3. Algoritmo del aumento del flujo: algoritmo de Ford y Fulkerson	472
14.5.4. Ejemplo de mejora de flujo	474
14.5.5. Esquema del algoritmo de aumento de flujo	476
14.5.6. Tipos de datos y pseudocódigo	478
14.5.7. Codificación del algoritmo de flujo máximo: Ford-Fulkerson ..	479

14.6.	Problema del árbol de expansión de coste mínimo	483
14.6.1.	Definiciones	483
14.6.2.	Árboles de expansión de coste mínimo	483
14.7.	Algoritmo de Prim y Kruskal	484
14.8.	Codificación. Árbol de expansión de coste mínimo	488
Resumen	493	
Ejercicios	493	
Problemas	496	

PARTE IV. ARCHIVOS Y ORDENACIÓN

Capítulo 15.	Ordenación, búsqueda y mezcla	501
15.1.	Introducción	502
15.2.	Ordenación	502
15.3.	Ordenación por burbuja	503
15.3.1.	Análisis	503
15.3.2.	Procedimientos de ordenación	507
15.3.3.	Algoritmo de burbuja mejorado (refinamiento)	507
15.3.4.	Programación completa de ordenación por burbuja ..	508
15.3.5.	Análisis de la ordenación por burbuja	509
15.4.	Ordenación por selección	510
15.5.	Ordenación por inserción	512
15.5.1.	Algoritmo de inserción binaria	514
15.5.2.	Comparación de ordenaciones cuadráticas.....	514
15.6.	Ordenación Shell	514
15.7.	Ordenación rápida (<i>quicksort</i>)	517
15.7.1.	Análisis de la ordenación rápida	522
15.8.	Ordenación por mezcla (<i>mergesort</i>)	522
15.9.	Método de ordenación por montículos (<i>heapsort</i>)	525
15.9.1.	Montículo	525
15.9.2.	Procedimiento empuja	528
15.9.3.	Montículo inicial	529
15.10.	Método de ordenación <i>binsort</i>	529
15.11.	Método de ordenación <i>radix-sort</i>	533
15.12.	Búsqueda lineal	536
15.12.1.	Análisis	536
15.12.2.	Eficiencia de la búsqueda lineal	539
15.13.	Búsqueda binaria	540
15.13.1.	Eficiencia de la búsqueda binaria	542
15.14.	Búsqueda binaria recursiva	544
15.14.1.	Función búsqueda	544
15.14.2.	Procedimiento BusquedaBin.....	545

15.15. Mezcla	546
Resumen	548
Ejercicios	548
Problemas	549
Capítulo 16. Análisis de algoritmos	553
16.1. Medida de la eficiencia de algoritmos	553
16.1.1. Análisis del orden de magnitud	554
16.1.2. Consideraciones de eficiencia	555
16.1.3. Análisis de rendimiento	555
16.1.4. Tiempo de ejecución	555
16.2. Notación <i>O-grande</i>	556
16.2.1. Descripción de tiempos de ejecución	557
16.2.2. Definición conceptual	557
16.2.3. Funciones tipo monomio	558
16.2.4. Órdenes de funciones polinómicas	559
16.2.5. Órdenes exponenciales y logarítmicas	559
16.2.6. Bases de logaritmos distintas	560
16.2.7. Inconvenientes de la notación <i>O-grande</i>	562
16.3. La eficiencia de los algoritmos de búsqueda	563
16.3.1. Búsqueda secuencial	563
16.3.2. Búsqueda binaria	564
16.4. Análisis de algoritmos de ordenación	565
16.4.1. Análisis de la ordenación por selección	565
16.4.2. Análisis de la ordenación por burbuja	566
16.4.3. Análisis de la ordenación por inserción	567
16.4.4. Ordenación por mezcla	568
16.4.5. Ordenación rápida	569
16.4.6. Análisis del método <i>Radix Sort</i>	570
16.5. Tablas comparativas de tiempos	571
Resumen	572
Ejercicios	573
Problemas	577
Referencias bibliográficas	579
Capítulo 17. Archivos (ficheros). Fundamentos teóricos	581
17.1. Noción de archivo. Estructura jerárquica	581
17.1.1. Campos	582
17.1.2. Registros	583
17.2. Conceptos, definiciones y terminología	584
17.3. Organización de archivos	587
17.3.1. Organización secuencial	588
17.3.2. Organización directa	589

17.4.	Operaciones sobre archivos	589
17.4.1.	Creación de un archivo	590
17.4.2.	Consulta de un archivo	590
17.4.3.	Actualización de un archivo	591
17.4.4.	Clasificación de un archivo	592
17.4.5.	Reorganización de un archivo	592
17.4.6.	Destrucción de un archivo	592
17.4.7.	Reunión, fusión de un archivo	592
17.4.8.	Rotura/estallido de un archivo	593
17.5.	Gestión de archivos	593
17.6.	Mantenimiento de archivos	594
17.6.1.	Altas	595
17.6.2.	Bajas	595
17.6.3.	Modificaciones	596
17.6.4.	Consulta	596
17.6.5.	Operaciones sobre registros	596
17.7.	Procesamiento de archivos. Problemas resueltos	596
	Resumen	613
	Ejercicios	614
	Problemas	615
Capítulo 18.	Tratamiento de archivos de datos	617
18.1.	Archivos de texto	617
18.2.	Cómo aumentar la velocidad de acceso a archivos	622
18.2.1.	Archivos con función de direccionamiento <i>hash</i>	622
18.2.2.	Funciones <i>hash</i>	623
18.2.3.	Resolución de colisiones	625
18.2.4.	Realización con encadenamiento	636
18.3.	Archivos indexados	645
18.3.1.	Archivo indexado de una fonoteca	645
18.3.2.	Programa de manipulación de archivos indexados	650
	Resumen	652
	Ejercicios	653
	Problemas	653
Capítulo 19.	Ordenación externa	655
19.1.	Ordenación externa. Métodos de ordenación	655
19.2.	Método de mezcla directa (mezcla simple)	656
19.2.1.	Codificación	657
19.3.	Método de mezcla equilibrada múltiple	661
19.3.1.	Tipos de datos	661
19.3.2.	Cambio de finalidad de archivo	662
19.3.3.	Control del número de tramos	662
19.3.4.	Codificación	663

19.4.	Método polifásico de ordenación externa	667
19.4.1.	Ejemplo con tres archivos	667
19.4.2.	Ejemplo con cuatro archivos	669
19.4.3.	Distribución inicial de tramos	670
19.4.4.	Mezcla polifásica <i>versus</i> mezcla múltiple	671
19.4.5.	Algoritmo de mezcla	671
19.4.6.	Codificación	672
	Resumen	678
	Ejercicios	678
	Problemas	679
	Referencias bibliográficas	679

PARTE V. PROGRAMACIÓN ORIENTADA A OBJETOS

Capítulo 20.	Objetos: Conceptos fundamentales y programación orientada a objetos	683
20.1.	La estructura de los objetos: sintaxis	684
20.1.1.	Encapsulamiento	691
20.2.	Secciones pública y privada	691
20.2.1.	Declaración de una base	692
20.2.2.	Declaración de un objeto	693
20.2.3.	Implementación de los métodos	693
20.3.	Definición de objetos mediante unidades	694
20.3.1.	Uso de un tipo objeto	695
20.4.	La herencia	696
20.4.1.	La herencia con <i>inherited</i>	702
20.5.	Los métodos	705
20.6.	Objetos dinámicos	705
20.6.1.	Asignación de objetos dinámicos	705
20.6.2.	Liberación de memoria y destructores	707
20.6.3.	La cláusula <i>Self</i>	709
20.7.	Polimorfismo	714
20.7.1.	Constructores y destructores en objetos dinámicos	717
20.7.2.	Anulación de métodos heredados	718
20.8.	Constructores y destructores	721
20.9.	Los procedimientos <i>New</i> y <i>Dispose</i> en POO	724
20.10.	Mejoras en programación orientada a objetos	725
20.10.1.	La ocultación mediante <i>public</i> y <i>private</i>	726
	Resumen	727
	Errores típicos de programación	729
	Ejercicios	730
	Problemas	730
	Referencias bibliográficas	730

Apéndices

Apéndice A.	Vademécum de Matemáticas para la resolución de algoritmos numéricos	733
Apéndice B.	Unidades estándar de Turbo Borland Pascal 7	739
	B.1. Las unidades estándar	740
	B.2. La unidad <i>System</i>	740
	B.3. La unidad <i>Printer</i>	741
	B.4. La unidad <i>DOS</i>	742
	B.5. Procedimientos y funciones de la unidad <i>DOS</i>	744
	B.6. La unidad <i>Crt</i>	753
	B.7. La unidad <i>Strings</i> : funciones.....	760
	Resumen	764
Apéndice C.	El editor de Turbo Pascal 7.0	765
Apéndice D.	El entorno integrado de desarrollo de Turbo Pascal 7.0	769
Apéndice E.	Depuración de sus programas en Turbo Pascal	783
Apéndice F.	Mensajes y códigos de error.....	797
Apéndice G.	Guía de referencia Turbo Borland Pascal	805
Apéndice H.	Guía del usuario ISO/ANSI Pascal Estándar	829
Índice		849

PRÓLOGO

Estructura de datos como disciplina informática y de computación

Una de las disciplinas clásicas en todas las carreras relacionadas con la Informática o las Ciencias de la Computación es *Estructura de datos*. El estudio de las estructuras de datos es casi tan antiguo como el nacimiento de la programación, y se ha convertido en materia de estudio obligatoria en todos los *curriculum* desde finales de los años sesenta y sobre todo en la década de los setenta cuando apareció el lenguaje Pascal de la mano del profesor Niklaus Wirth, y posteriormente en la década de los ochenta con la aparición de su obra —ya clásica— *Algorithms and Data Structures* en 1986.

Ha sido en la década de los ochenta cuando surgieron los conceptos clásicos de estructuras de datos, y aparecieron otros nuevos que se han ido consolidando en la segunda mitad de esa década, para llegar a implantar la característica distintiva de la década de los noventa, *la orientación a objetos*. Una razón para que esta característica se haya convertido en núcleo fundamental de la programación que se realiza en los noventa, y que se seguirá realizando en el tercer milenio, ha sido que un objeto encapsula el diseño o implementación de un tipo abstracto de datos (**TAD**). Además de encapsulación de datos, los objetos también encapsulan herencia; por ejemplo, una cola es un tipo de lista con restricciones específicas, de modo que el diseño del tipo de dato *cola* puede heredar algunos de los diseños de métodos (operaciones) del tipo de dato *Lista*.

Así pues, *Estructura de datos. Algoritmos, abstracción y objetos* trata sobre el estudio de las estructuras de datos dentro del marco de trabajo de los tipos abstractos de datos (**TAD**) y bajo la óptica del análisis y diseño de los algoritmos que manipulan esos tipos abstractos de datos u objetos.

Objetivos y características

La filosofía con la que se ha construido el libro reside en el desarrollo modular de programas que, hoy día, conduce de modo inequívoco a los principios de ingeniería de software que facilitan la construcción de grandes programas: *abstracción, legibilidad, comprensibilidad y reutilización o reusabilidad del software*. Estos principios se consiguen mediante especificación del software, que supone, a su vez, la separación entre la espe-

cificación y la implementación (unidades en Pascal, paquetes en Ada 95, clases en C++, etcétera) y conduce a la construcción de bibliotecas de componentes (clases y objetos) que facilitan la construcción de software mediante la reutilización de software ya existente en bibliotecas de funciones y/o componentes.

El lenguaje utilizado en el libro es Pascal y, en particular, la versión Turbo Borland que soporta el concepto de unidad (módulo de compilación separada) y de clases y objetos, y en consecuencia facilita la construcción de tipos abstractos de datos y la programación orientada a objetos.

Para conseguir esos principios que rigen la construcción de programas siguiendo las modernas teorías de la ingeniería de software, es preciso fijarse una serie de objetivos parciales, cuya consecución lleva al objetivo final: *«Enseñanza, aprendizaje y dominio de los conceptos de estructuras de datos y sus algoritmos de manipulación»*.

En consecuencia, los diferentes objetivos generales que busca el libro son:

- Introducir y describir en profundidad las estructuras de datos clásicas (*pilas, colas, listas, árboles, grafos y archivos*) que se encuentran en casi todos los programas de computadora.
- Enseñar a los estudiantes y lectores autodidactas, el uso de técnicas de análisis y diseño con las que se pueden evaluar y validar algoritmos.
- Proporcionar ejemplos de principios de ingeniería de software y técnicas de programación que incluyen abstracción de datos y programación orientada a objetos.
- Introducción al uso adecuado de las características específicas de Turbo Borland Pascal, especialmente aquellas que permiten la compilación separada: unidades, tanto predefinidas —incorporadas en el compilador— como definidas por el usuario.

El lenguaje elegido para implementar los diferentes algoritmos incluidos en el libro ha sido, como ya se ha comentado, Pascal. La razón esencial ha sido *pragmática*: Pascal sigue siendo, y estamos seguros que así seguirá, el lenguaje idóneo para enseñar a los estudiantes de Computación, Informática y restantes Ingenierías, las técnicas de programación estructurada. Cientos de miles, han sido los estudiantes que han aprendido Pascal y ese ha sido su primer lenguaje de programación, aunque hoy en el campo profesional haya sido desplazado por otros lenguajes tales como C, C++, Java o Visual Basic.

En cualquier forma, siempre hemos tenido la idea de que los lenguajes de computadora no son sólo un mecanismo de descripción de algoritmos o de programas de computadoras, sino también un vehículo de concepción, reflexión y análisis, en la resolución de problemas con computadora. Por ello, pensamos que Pascal, como cualquier otro lenguaje estructurado o incluso un pseudolenguaje, como puede ser un pseudocódigo, servirá para conseguir el rigor y el formalismo que el profesor quiera conseguir con sus alumnos, o el propio estudiante se marque siguiendo las directrices de sus maestros o profesores.

Desde un punto de vista científico, el libro se ha escrito apoyándose en las obras de las autoridades más reconocidas en el campo de algoritmos y estructuras de datos, fundamentalmente D. E. Knuth y N. Wirth, y las inestimables obras de A. V. Aho, J. Hopcroft, J. D. Ullman, E. Horowitz, D. Sahni, B. Liskov, entre otros, y que el lector encon-

trará en las referencias bibliográficas que acompañan a algunos capítulos o en la bibliografía final del libro.

El libro como texto de referencia universitaria y profesional

Estructura de datos es una disciplina académica que se incorpora a todos los planes de estudios de carreras universitarias de Ingeniería Informática, Ingeniería en Sistemas Computacionales y Licenciatura en Informática, así como suele ser también frecuente la incorporación en planes de estudios de informática en Formación Profesional o Institutos Politécnicos. Suele considerarse también *Estructuras de datos* como una extensión de las asignaturas de Programación, en cualquiera de sus niveles. Por estas circunstancias han nacido numerosas iniciativas de incorporación de esta disciplina a los diferentes planes de estudios.

Este es el caso de España. Los planes de estudios de Ingeniería en Informática, Ingeniería Técnica en Informática de Sistemas e Ingeniería Técnica en Informática de Gestión incluyen, todos ellos, una materia troncal llamada *Estructura de datos*, que luego se convierte en asignaturas anuales o semestrales. El Consejo de Universidades no sólo obliga a las universidades a incluir esta asignatura como materia troncal, y en consecuencia asignatura obligatoria, sino que además recomienda «descriptores» para la misma. Así estos descriptores son: «tipos abstractos de datos, estructuras de datos y algoritmos de manipulación, ficheros (*archivos*)». En esta edición se ha dejado fuera del contenido del libro el descriptor bases de datos, por entender que suele constituir parte de una asignatura específica que aún formando parte de *Estructura de datos* suele tener entidad propia o complementaria. De igual forma la organización internacional ACM recomienda un currículum para la carrera de *Computer Science*, equivalente a las carreras universitarias españolas ya citadas. Asimismo, todos los países de Iberoamérica han incluido también la asignatura *Estructura de datos* en sus *currículos*. Así nos consta por nuestras numerosas visitas profesionales a universidades e institutos tecnológicos de México, Cuba, Venezuela, Guatemala, etc., donde hemos podido examinar sus planes de estudios e intercambiar experiencias, tanto con profesores (*maestros*) como con estudiantes.

Estructuras de datos: Algoritmos, abstracción y objetos. Siguen las recomendaciones dadas por el Consejo de Universidades de España para las carreras de Ingeniería Informática, tanto la superior como la técnica, y se ha adaptado en la medida de lo posible al currículum recomendado por la ACM, en particular el curso C2 (*Curriculum '78*, CACM 3/79 y CACM 8/85). Asimismo se adapta a cursos típicos sobre conocimiento de algoritmos y estructura de datos, así como de los lenguajes de programación, según se describe en el informe Computing Curricula 1991 del ACM/IEEE-CS (Commun ACM 34, 6 junio 1991) *Joint Curriculum Task Force*. El libro trata de ajustarse a las directrices dadas para *Estructura de datos y Análisis de algoritmos* propuestos en ese informe y se puede usar como materia fundamental y/o complementaria de un curso normal de ciencias de la computación para graduados y no graduados.

El texto se ha diseñado para un curso de dos cuatrimestres (semestres), de algoritmos y estructuras de datos. Su división —selección del capítulo— y adscripción a uno u otro

cuatrimestre, creemos deben ser decididos por el profesor (maestro) en función de su experiencia y de los objetivos marcados. Podría servir para un cuatrimestre siempre que hubiese una selección adecuada del profesor y se tuviera en cuenta la formación previa del estudiante.

Requisitos

El único requisito previo para los estudiantes que emplean este texto es un curso de uno o dos semestres, de programación con diseño de algoritmos mediante pseudocódigos, o lenguajes estructurados, tales como FORTRAN, Pascal, C, C++ o Java. Los estudiantes que han estudiado sus cursos de programación sin utilizar el lenguaje Pascal, pueden seguir este libro junto con alguno de los libros de texto que se listan en la bibliografía final del libro o en las referencias bibliográficas de los capítulos 1 y 2. No obstante, se han incluido varios apéndices sobre Pascal y Turbo Pascal que pensamos será suficiente, al menos, a nivel de sintaxis para refrescar o adquirir aquellos conocimientos básicos necesarios para seguir la calificación de los diferentes algoritmos del libro.

Depuración

Todos los programas y algoritmos del texto se han probado y depurado. Además de nuestra tarea de puesta a punto de programas, diferentes profesores nos han ayudado en esta tarea. Este es el caso de Matilde Fernández, Lucas Sánchez, Jesús Pérez, Isabel Torralvo y M.^a del Mar García, todos ellos profesores de la Facultad de Informática y Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca en el campus de Madrid. Su entusiasmo y el cariño puesto en esta actividad ha sido desbordante y siempre aportaron sugerencias valiosas. Por supuesto, que cualquier error que todavía pudiera existir, es absoluta responsabilidad de los autores.

Los ejercicios y problemas de los diferentes capítulos varían en tipo y dificultad. Algunos son reiterativos para confirmar la comprensión de los textos. Otros implican modificaciones a programas o algoritmos presentados. Se han incluido gran cantidad de problemas de programación con diferentes niveles de complejidad que confiamos sean complementos eficientes a la formación conseguida por el lector.

Contenido

Este libro se ha dividido en cinco partes y ocho apéndices.

Parte I: *Abstracción y Programación: Introducción a la ingeniería de software.* Esta parte contiene los capítulos 1, 2 y 3 y describe una introducción a la metodología de construcción de grandes programas, una comparación entre los conceptos de módulos y unidades, así como el importante concepto de abstracción de datos.

Parte II: *Fundamentos básicos de estructuras de datos y tipos abstractos de datos.* Esta parte contiene la descripción del importante tipo de dato llamado **puntero**, junto con la descripción de las estructuras de datos más típicas: *listas enlazadas, listas doblemente enlazadas, pilas y colas*.

Parte III: *Estructuras de datos avanzadas.* Las estructuras *árboles binarios, equilibrado y B, grafos*, junto con el estudio de *algoritmos recursivos* y de manipulación de las citadas estructuras de datos, son el tema central de esta parte.

Parte IV: *Archivos y ordenación.* En esta parte se tratan los importantes algoritmos de manipulación de *ordenación, búsqueda y mezcla*, junto con el concepto de *archivo* y su tratamiento; otro concepto importante que se estudia en esta parte, es el *Análisis de Algoritmos*, como técnica indispensable para conseguir algoritmos eficientes.

Parte V: En esta parte se introduce al lector en *programación orientada a objetos*, apoyada en el concepto de objeto como extensión de un tipo abstracto de datos.

Una descripción detallada de los diferentes capítulos se proporciona a continuación. El capítulo 1 es una introducción a la metodología de construcción de software, así como las técnicas de ingeniería de software. Se describen las herramientas usuales para la resolución de problemas junto con las diferentes etapas del ciclo de vida del software. Asimismo se enumeran los principios de diseño de sistemas de software, junto con normas para un estilo de programación, documentación, depuración y pruebas.

En el capítulo 2 se describe el concepto de *módulo* y cómo se puede implementar en Turbo Pascal, mediante unidades. Se describe el concepto de unidad, el modo de creación y de utilización de las mismas. El importante concepto de abstracción de datos y su implementación mediante tipos abstractos de datos.

En el capítulo 3 se comienza el estudio de la orientación a objetos así como los diferentes lenguajes de programación orientados a objetos, junto al importante concepto de *reutilización de software*.

El capítulo 4 proporciona una introducción a las estructuras de datos dinámicas. El capítulo describe el *tipo de dato puntero* en Pascal y muestra cómo utilizar la asignación dinámica de memoria.

El tipo abstracto de dato *Lista*, se estudia en el capítulo 5. La implementación más usual son las *listas enlazadas* y por esta razón se analizan sus algoritmos de manipulación. Uno de los casos más frecuentes en el diseño y construcción de estructura de datos es la ordenación de elementos. El capítulo describe el caso particular de las listas enlazadas.

Otros tipos de listas muy utilizadas en la manipulación de estructuras de datos son las *listas doblemente enlazadas* y las *listas circulares*. Sus algoritmos de implementación se estudian en el capítulo 6.

Las *pilas* y las *colas* son otros tipos de datos muy usuales en cualquier programa de software. Por esta razón se estudian en detalle junto con los procedimientos y funciones necesarios para poder utilizarlas. Estas estructuras de datos son listas, por lo que su

implementación se realizará, no sólo con arrays, sino fundamentalmente mediante listas enlazadas o circulares. Los tipos abstractos de datos pila y cola se describen en los capítulos 7 y 8.

El capítulo 9 se dedica a la descripción de la recursividad así como al uso de procedimientos y funciones recursivas. La *recursividad* es una propiedad esencial en el diseño de técnicas algorítmicas, por esta circunstancia se hace un estudio exhaustivo y detallado de los algoritmos recursivos clásicos y aquellos otros más complejos. Así se analizan los típicos algoritmos de *divide* y *vencerás*, como es el popular algoritmo de las *Torres de Hanoi*, hasta los *algoritmos de vuelta atrás* (*backtracking*) más famosos y eficientes tales como: «el problema de las ocho reinas», «el problema de la mochila» o el «problema de los matrimonios estables».

El capítulo 10 introduce al tipo de dato *Árbol binario*. Los árboles binarios son estructuras de datos recursivas y la mayoría de sus propiedades se pueden definir recursivamente. Los árboles binarios tienen numerosas aplicaciones; entre ellas una de las más usuales es la evaluación de expresiones. En el capítulo se describen también los descendientes de los árboles binarios, como es el caso de los árboles binarios de búsqueda. Las operaciones en los árboles binarios de búsqueda son motivo de análisis a lo largo del capítulo.

Otro tipo de extensiones árboles son los complejos y eficientes *árboles B* y *árboles AVL* o *equilibrados*. Sus algoritmos de manipulación se describen en los capítulos 11 y 12.

Los grafos son uno de los tipos de datos más clásicos y de mayor utilidad en el diseño de problemas complejos. La implementación y los algoritmos de manipulación de grafos se estudian en el capítulo 13. Sin embargo, un estudio completo de grafos y de la teoría de grafos requiere mucho más que un capítulo; por esta circunstancia, se examinan aspectos avanzados en el capítulo 14. Así, se describen algoritmos complejos tales como los *algoritmos de Dijkstra, Warshall, Floyd o Kruskal*.

Los métodos clásicos de ordenación, búsqueda y mezcla, tanto los más sencillos hasta otros complejos, se describen en el capítulo 15. Así se analizan los métodos de ordenación por burbuja, selección, inserción, *Shell*, *quicksort* (rápido), *mergesort* (mezcla) o *heapsort* (montículo), junto con la ordenación binaria (*binsort*) o radix (*Radix-sort*).

Las técnicas de análisis de algoritmos y la medida de la eficiencia de los mismos se analizarán en el capítulo 16. La notación *O grande* utilizada en el análisis de algoritmos se describe en dicho capítulo 16.

La estructura de datos clásica en cuanto a organización de los datos son los archivos (ficheros). La organización de archivos junto con las operaciones sobre los mismos se describen en el capítulo 17. En el capítulo 18 se describen los archivos de texto y los archivos indexados, junto con las técnicas de direccionamiento aleatorio (*hash*).

La ordenación externa de estructuras de datos junto con los métodos de implementación correspondiente se estudian en el capítulo 19.

El capítulo 20 realiza una descripción del tipo de dato objeto junto con la implementación de las propiedades clásicas que soportan tales como herencia y polimorfismo.

Los apéndices complementan el contenido teórico del libro. El Apéndice A es un *vademécum* de fórmulas matemáticas que se requieren normalmente en la construcción

de algoritmos no sólo numéricos sino de gestión. El concepto de módulo o unidad en Turbo Pascal es motivo de estudio en el apéndice B, donde se describen las unidades estándar del compilador. El apéndice C describe el editor de texto del compilador así como la descripción de su funcionalidad. El apéndice E describe los métodos de depuración de programas en Turbo Borland Pascal y los mensajes y códigos de error se incluyen en el apéndice F. El apéndice G es una amplia guía de referencia de sintaxis del compilador de Turbo Borland Pascal. El apéndice H es una guía de usuario de ISO/ANSI Pascal pensado en aquellos lectores que utilicen un compilador de Pascal estándar distinto del compilador Turbo Borland Pascal.

AGRADECIMIENTOS

Este libro no es sólo fruto del trabajo de los autores, tal y como sucede con la creación de la mayoría de los libros técnicos. Muchas personas han colaborado de una u otra manera para que este libro vea la luz.

Entre estas personas queremos destacar aquellas que más han influido en la versión última. Los profesores de *Estructuras de datos*, **Lucas Sánchez, Matilde Fernández y Jesús Pérez**, del Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software de la Facultad de Informática y de la Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca (UPSA), con sus aportaciones y sugerencias fruto de su larga experiencia como profesores de la asignatura *Estructura de datos* en el campus de Madrid, han leído y revisado las primeras pruebas de impresión de este libro, y nos han dado ideas y sugerencias que han mejorado notablemente el libro. Las profesoras M.^a Mar García e Isabel Torralvo y el profesor Joaquín Abeger, todos del mismo departamento que los profesores anteriores, también han leído las galeras del texto y han detectado erratas y depurado muchos programas del libro.

Además de los profesores anteriores, otros profesores de la Facultad y Escuela Universitaria de Informática de UPSA han ayudado y apoyado a que esta obra sea una realidad: Paloma Centenera, Luis Villar y Ángel Hermoso.

A todos deseamos agradecer nuestro reconocimiento por su arduo trabajo. No sólo nos han demostrado que son colegas, sino y sobre todo son amigos. Nuestras gracias más sinceras a todos ellos, sin su apoyo y trabajo esta obra no sería la misma.

Naturalmente, no seríamos justos si no reconociéramos la inmensa ayuda que ha supuesto y seguirá suponiendo las sugerencias, consejos y críticas que nuestros numerosos alumnos de todos estos años pasados nos han hecho tanto en España como en Latinoamérica. Son el mejor aliciente que cualquier profesor puede tener. En nuestro caso ha sido una realidad vivida día a día. Nuestro agradecimiento eterno.

Una última reflexión

Estructura de datos es una disciplina clave en los currículum universitarios de informática y computación. En nuestra obra hemos volcado nuestra experiencia de más de diez

años en la enseñanza universitaria; nuestra esperanza es que hayamos conseguido transmitir dicha experiencia, y que nuestros lectores puedan no sólo iniciarse en el aprendizaje de las estructuras de los datos, sino llegar a dominar su concepto, organización, diseño y manipulación de los algoritmos correspondientes. El mayor deseo de los autores sería poder conseguir los objetivos del libro, que su aprendizaje sea gradual y que lleguen a dominar este vital e importante campo informático de las estructuras de datos.

Madrid, julio de 1998

Los autores

*Abstracción
y programación.
Introducción a la
ingeniería de software*

Ingeniería de software: introducción a la metodología de construcción de grandes programas

CONTENIDO

- 1.1. Abstracción.
- 1.2. Resolución de problemas de programación.
- 1.3. Herramientas para la resolución de problemas.
- 1.4. Factores en la calidad del software.
- 1.5. El ciclo de vida del software.
- 1.6. Métodos formales de verificación de programas.
- 1.7. Principios de diseños de sistemas de software.
- 1.8. Estilo de programación.
- 1.9. La documentación.
- 1.10. Depuración.
- 1.11. Diseño de algoritmos.
- 1.12. Pruebas (*testing*).
- 1.13. Eficiencia.
- 1.14. Transportabilidad (*portabilidad*).

RESUMEN.

EJERCICIOS.

PROBLEMAS.

La producción de un programa se puede dividir en diferentes fases: *análisis, diseño, codificación y depuración, prueba y mantenimiento*. Estas fases se conocen como *ciclo de vida del software*, y son los principios básicos en los que se apoya la ingeniería del software. Deben considerarse siempre todas las fases en el proceso de creación de programas, sobre todo cuando éstos son grandes proyectos. La *ingeniería de o del software* trata de la creación y producción de programas a gran escala.

1.1. ABSTRACCIÓN

Los seres humanos se han convertido en la especie más influyente de este planeta, debido a su capacidad para abstraer el pensamiento. Los sistemas complejos, sean naturales o artificiales, sólo pueden ser comprendidos y gestionados cuando se omiten detalles que son irrelevantes a nuestras necesidades inmediatas. El proceso de excluir detalles no deseados o no significativos, al problema que se trata de resolver, se denomina **abstracción**, y es algo que se hace en cualquier momento.

Cualquier sistema de complejidad suficiente se puede visualizar en diversos *niveles de abstracción* dependiendo del propósito del problema. Si nuestra intención es conseguir una visión general del proceso, las características del proceso presente en nuestra abstracción constará principalmente de generalizaciones. Sin embargo, si se trata de modificar partes de un sistema, se necesitará examinar esas partes con gran nivel de detalle. Consideremos el problema de representar un sistema relativamente complejo tal como un coche. El nivel de abstracción será diferente según sea la persona o entidad que se relaciona con el coche: conductor, propietario, fabricante o mecánica.

Así, desde el punto de vista del conductor sus características se expresan en términos de sus funciones (acelerar, frenar, conducir, etc.); desde el punto de vista del propietario sus características se expresan en función de nombre, dirección, edad; la mecánica del coche es una colección de partes que cooperan entre sí para proveer las funciones citadas, mientras que desde el punto de vista del fabricante interesa precio, producción anual de la empresa, duración de construcción, etc. La existencia de diferentes niveles de abstracción conduce a la idea de una *jerarquía de abstracciones*.

Las soluciones a problemas no triviales tiene una jerarquía de abstracciones de modo que sólo los objetivos generales son evidentes al nivel más alto. A medida que se desciende en nivel los aspectos diferentes de la solución se hacen evidentes.

En un intento de controlar la complejidad, los diseñadores del sistema explotan las características bidimensionales de la jerarquía de abstracciones. La primera etapa al tratar con un problema grande es seleccionar un nivel apropiado a las herramientas (*hardware* y *software*) que se utilizan para resolverlo. El problema se descompone entonces en subproblemas que se pueden resolver independientemente de modo razonable.

1.2. RESOLUCIÓN DE PROBLEMAS DE PROGRAMACIÓN

El término **resolución del problema** se refiere al proceso completo de tomar la descripción del problema y desarrollar un programa de computadora que resuelva ese problema. Este proceso requiere pasar a través de muchas fases: desde una buena comprensión del problema a resolver, hasta el diseño de una solución conceptual, para implementar la solución con un programa de computadora.

Realmente **¿qué es una solución?** Normalmente, una **solución** consta de dos componentes: *algoritmos* y *medios para almacenar datos*. Un **algoritmo** es una especificación concisa de un método para resolver un problema. Una acción que un algoritmo realiza con frecuencia es operar sobre una colección de datos. Por ejemplo, un algoritmo puede

tener que poner menos datos en una colección, quitar datos de una colección o realizar preguntas sobre una colección de datos. Cuando se construye una solución, se deben organizar sus colecciones de datos de modo que se pueda operar sobre los datos fácilmente en la manera que requiera el algoritmo.

Sin embargo, no sólo se necesita almacenar los datos en **estructuras de datos** sino también operar sobre esos datos. De este modo una entidad fundamental será el **tipo abstracto de datos (TAD)** (*Abstract Data Type, ADT*), que es una colección de datos y un conjunto de operaciones que actúan sobre esos datos. Por ejemplo, supongamos que se necesita almacenar una colección de nombres de futbolistas de modo que permita una búsqueda rápida de un nombre dado. El algoritmo de búsqueda binaria que se estudiará en el capítulo 15 permitirá buscar en una lista (un array) eficientemente, si el array o lista está ordenado. Por consiguiente, una solución a este problema es almacenar los nombres ordenados en un array y utilizar un algoritmo de búsqueda binaria para buscar en el array un nombre especificado. *Un TAD que visualice el array ordenado junto con el algoritmo de búsqueda binaria resolverá el problema.*

1.3. HERRAMIENTAS PARA LA RESOLUCIÓN DE PROBLEMAS

Diferentes herramientas ayudan al programador y al ingeniero de software a diseñar una solución para un problema dado. Algunas de estas herramientas son *diseño descendente, abstracción procedimental, abstracción de datos, ocultación de la información, recursión o recursividad y programación orientada a objetos*.

1.3.1. Diseño descendente

Cuando se escriben programas de tamaño y complejidad moderada, nos enfrentamos a la dificultad de escribir dichos programas. La solución para resolver estos problemas y, naturalmente, aquellos de mayor tamaño y complejidad, es recurrir a la **modularidad** mediante el **diseño descendente**. ¿Qué significa diseño descendente y modularidad? La filosofía del diseño descendente reside en que se descompone una tarea en sucesivos niveles de detalle. Para ello se divide el programa en **módulos independientes**—procedimientos, funciones y otros bloques de código—, como se observa en la Figura 1.1.

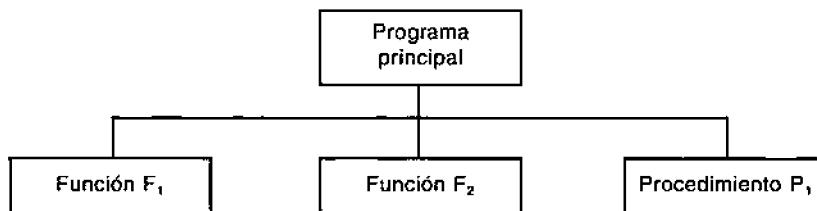


Figura 1.1. Un programa dividido en módulos independientes.

El concepto de solución modular se aprecia en la aplicación de la Figura 1.2, que busca encontrar la nota media de un conjunto de notas de una clase de informática. Existe un módulo del más alto nivel que se va refinando en sentido descendente para encontrar módulos adicionales más pequeños. El resultado es una jerarquía de módulos; cada módulo se refina por los de bajo nivel que resuelve problemas más pequeños y contiene más detalles sobre los mismos. El proceso de refinamiento continúa hasta que los módulos de nivel inferior de la jerarquía sean tan simples como para traducirlos directamente a procedimientos, funciones y bloques de código en Pascal o C, que resuelven problemas independientes muy pequeños. De hecho, cada módulo de nivel más bajo debe ejecutar una tarea bien definida. Estos módulos se denominan *altamente cohesivos*.

Cada módulo se puede dividir en subtareas. Por ejemplo, se puede refinar la tarea de leer las notas de una lista, dividiéndolo en dos subtareas. Por ejemplo, se puede refinar la tarea de leer las notas de la lista en otras dos subtareas: *pedir al usuario una nota* y *situar la nota en la lista*.

1.3.2. Abstracción procedural

Cada algoritmo que resuelve el diseño de un módulo equivale a una caja negra que ejecuta una tarea determinada. Cada caja negra especifica *lo que hace* pero *no cómo lo hace*, y de igual modo cada caja negra conoce cuantas cajas negras existen y lo que hacen.

Normalmente, estas cajas negras se implementan como subprogramas. Una **abstracción procedural** separa el propósito de un subprograma de su implementación. Una vez que un subprograma se haya escrito o codificado, se puede usar sin necesidad de conocer su cuerpo y basta con su nombre y una descripción de sus parámetros.

La modularidad y abstracción procedural son complementarios. La modularidad implica romper una solución en módulos; la abstracción procedural implica la especificación de cada módulo *antes* de su implementación en Pascal. El módulo implica que se puede cambiar su algoritmo concreto sin afectar el resto de la solución.

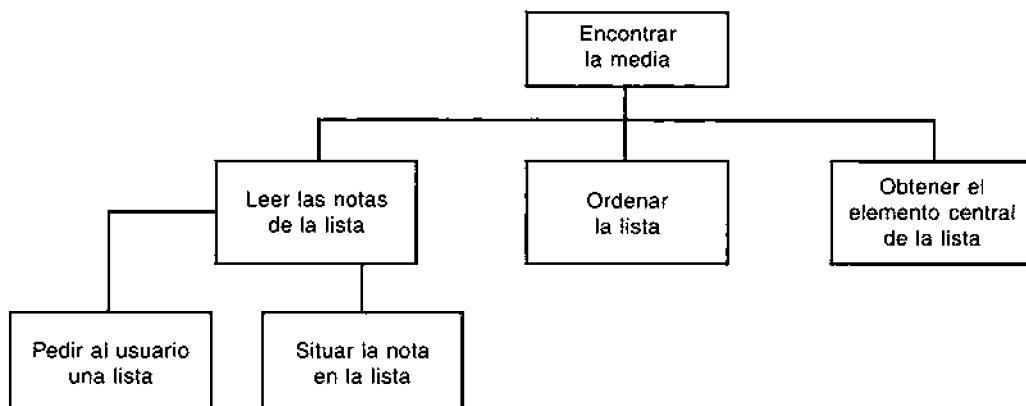


Figura 1.2. Diagrama de bloques que muestra la jerarquía de módulos.

La abstracción procedimental es esencial en proyectos complejos, de modo que se pueden utilizar subprogramas escritos por otras personas, sin necesidad de tener que conocer sus algoritmos.

1.3.3. Abstracción de datos

La abstracción procedural significa centrarse en lo que hace un módulo en vez de en los detalles de cómo se implementan los detalles de sus algoritmos. De modo similar, la **abstracción de datos** se centra en las operaciones que se ejecutan sobre los datos en vez de cómo se implementarán las operaciones.

Como ya se ha comentado antes, **un tipo abstracto de datos (TAD)** es una colección de datos y un conjunto de operaciones sobre esos datos. Tales operaciones pueden añadir nuevos datos, o quitar datos de la colección, o buscar algún dato. Los otros módulos de la solución **conocerán** qué operaciones puede realizar un TAD. Sin embargo, no conoce **cómo** se almacenan los datos ni **cómo** se realizan esas operaciones.

Cada TAD se puede implementar utilizando **estructuras de datos**. Una estructura de datos es una construcción que se puede definir dentro de un lenguaje de programación para almacenar colecciones de datos. En la resolución de un problema, los tipos abstractos de datos soportan algoritmos y los algoritmos son parte de los que constituye un TAD. Para diseñar una solución, se deben desarrollar los algoritmos y los TAD al unísono.

1.3.4. Ocultación de la información

La abstracción identifica los aspectos esenciales de módulos y estructura de datos que se pueden tratar como cajas negras. La abstracción es responsable de sus vistas externas o *públicas*, pero también ayuda a identificar los detalles que debe *ocultar* de la vista pública (*vista privada*). El principio de **ocultación de la información** no sólo oculta los detalles dentro de la caja negra, sino que asegura que ninguna otra caja negra pueda acceder a estos detalles ocultos. Por consiguiente, se deben ocultar ciertos detalles dentro de sus módulos y TAD y los hacen inaccesibles a otros módulos y TAD.

Un usuario de un módulo no se preocupa sobre los detalles de su implementación y, al contrario, un desarrollador de un módulo o TAD no se preocupa sobre sus usos.

1.3.5. Programación orientada a objetos

Los conceptos de modularidad, abstracción procedural, abstracción de datos y ocultación de la información conducen a la programación orientada a objetos, basada en el módulo o tipo de dato **objeto**.

Las prioridades fundamentales de un tipo son: **encapsulamientos, herencia y polimorfismo**. El **encapsulamiento** es la combinación de datos y operaciones que se pueden ejecutar sobre esos datos en un objeto. En Turbo Borland Pascal el encapsulamiento en un objeto se codifica mediante una *unidad*.

Herencia es la propiedad que permite a un objeto transmitir sus propiedades a otros objetos denominados descendientes; la herencia permite la reutilización de objetos que

se hayan definido con anterioridad. El **polimorfismo** es la propiedad que permite decidir en tiempo de ejecución la función a ejecutar, al contrario que sucede cuando no existe polimorfismo en el que la función a ejecutar se decide previamente y sin capacidad de modificación, en tiempo de ejecución.

1.4. FACTORES EN LA CALIDAD DEL SOFTWARE

La construcción de software requiere el cumplimiento de numerosas características. Entre ellas se destacan las siguientes:

Eficiencia

La eficiencia de un software es su capacidad para hacer un buen uso de los recursos que manipula.

Transportabilidad (portabilidad)

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

Verificabilidad (facilidad de verificación)

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

Integridad

La *integridad* es la capacidad de un software a proteger sus propios componentes contra los procesos que no tenga el derecho de acceder.

Facilidad de utilización

Un software es fácil de utilizar si se puede comunicar consigo de manera cómoda.

Corrección (exactitud)

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

Robustez

Capacidad de los productos software de funcionar incluso en situaciones anormales.

Extensibilidad

Facilidad que tienen los productos de adaptarse a cambios en su especificación. Existen dos principios fundamentales para conseguir esto:

- diseño simple;
- descentralización.

Reutilización

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

Compatibilidad

Facilidad de los productos para ser combinados con otros.

1.5. EL CICLO DE VIDA DEL SOFTWARE

Existen dos niveles en la construcción de programas: aquellos relativos a pequeños programas (los que normalmente realizan programadores individuales) y aquellos que se refieren a sistemas de desarrollo de programas grandes (*proyectos de software*) y que, generalmente, requieren un equipo de programadores en lugar de personas individuales. El primer nivel se denomina *programación a pequeña escala*; el segundo nivel se denomina *programación a gran escala*.

La programación en pequeña escala se preocupa de los conceptos que ayudan a crear pequeños programas —aquellos que varían en longitud desde unas pocas líneas a unas pocas páginas—. En estos programas se suele requerir claridad y precisión mental y técnica. En realidad, el interés mayor desde el punto de vista del futuro programador profesional está en los programas de gran escala que requiere de unos principios sólidos y firmes de lo que se conoce como *ingeniería de software* y que constituye un conjunto de técnicas para facilitar el desarrollo de programas de computadora. Estos programas o mejor proyectos de software están realizados por equipos de personas dirigidos por un director de proyectos (analista o ingeniero de software) y los programas pueden tener más de 100.000 líneas de código.

El desarrollo de un buen sistema de software se realiza durante el *ciclo de vida* que es el periodo de tiempo que se extiende desde la concepción inicial del sistema hasta su eventual retirada de la comercialización o uso del mismo. Las actividades humanas relacionadas con el ciclo de vida implican procesos tales como análisis de requisitos, diseño, implementación, codificación, pruebas, verificación, documentación, mantenimiento y evolución del sistema y obsolescencia. En esencia el ciclo de vida del software comienza con una idea inicial, incluye la escritura y depuración de programas, y continúa durante años con correcciones y mejoras al software original¹.

¹ Carrasco, Hellman y Verof: *Data structures and problem solving with Turbo Pascal*. The Benjamin/Cummings Publishing, 1993, pág. 210.

El ciclo de vida del software es un proceso iterativo, de modo que se modificarán las sucesivas etapas en función de la modificación de las especificaciones de los requisitos producidos en la fase de diseño o implementación, o bien una vez que el sistema se ha implementado, y probado, pueden aparecer errores que será necesario corregir y depurar, y que requieren la repetición de etapas anteriores.

La Figura 1.3 muestra el ciclo de vida de software y la disposición típica de sus diferentes etapas en el sistema conocido como *ciclo de vida en cascada*, que supone que la salida de cada etapa es la entrada de la etapa siguiente.

1.5.1. Análisis

La primera etapa en la producción de un sistema de software es decidir exactamente *qué* se supone ha de hacer el sistema; esta etapa se conoce también como *análisis de requisitos* o *especificaciones* y por esta circunstancia muchos tratadistas suelen subdividir la etapa en otras dos:

- *Análisis y definición del problema.*
- *Especificación de requisitos.*

La parte más difícil en la tarea de crear un sistema de software es definir cuál es el problema y a continuación especificar lo que se necesita para resolverlo. Normalmente la definición del problema comienza analizando los requisitos del usuario, pero estos requisitos, con frecuencia, suelen ser imprecisos y difíciles de describir. Se deben especificar todos los aspectos del problema, pero con frecuencia las personas que describen el problema no son programadores y eso hace imprecisa la definición. La fase de especificación requiere normalmente la comunicación entre los programadores y los futuros usuarios del sistema e iterar la especificación, hasta que tanto el especificador como los usuarios estén satisfechos de las especificaciones y hayan resuelto el problema normalmente.

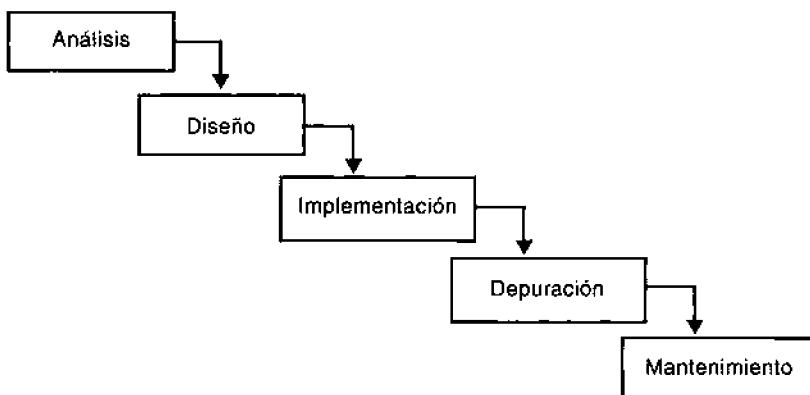


Figura 1.3. Ciclo de vida del software.

En la etapa de especificaciones puede ser muy útil para mejorar la comunicación entre las diferentes partes implicadas construir un prototipo o modelo sencillo del sistema final; es decir, escribir un programa prototípico que simule el comportamiento de las partes del producto software deseado. Por ejemplo, un programa sencillo —incluso ineficiente— puede mostrar al usuario la interfaz propuesta por el analista. Es mejor descubrir cualquier dificultad o cambiar su idea original antes que la programación se encuentre en estado avanzado o, incluso, terminada. El *modelado de datos* es una herramienta muy importante en la etapa de definición del problema. Esta herramienta es muy utilizada en el diseño y construcción de bases de datos.

Tenga presente que el usuario final, normalmente, no conoce exactamente lo que desea haga el sistema. Por consiguiente, el analista de software o programador, en su caso, debe interactuar con el usuario para encontrar lo que el usuario *deseará* haga el sistema. En esta etapa se debe responder a preguntas tales como:

- ¿Cuáles son los datos de entrada?
- ¿Qué datos son válidos y qué datos no son válidos?
- ¿Quién utilizará el sistema: especialistas cualificados o usuarios cualesquiera (sin formación)?
- ¿Qué interfaces de usuario se utilizarán?
- ¿Cuáles son los mensajes de error y de detección de errores deseables? ¿Cómo debe actuar el sistema cuando el usuario cometa un error en la entrada?
- ¿Qué hipótesis son posibles?
- ¿Existen casos especiales?
- ¿Cuál es el formato de la salida?
- ¿Qué documentación es necesaria?
- ¿Qué mejoras se introducirán —probablemente— al programa en el futuro?
- ¿Cómo debe ser de rápido el sistema?
- ¿Cada cuánto tiempo ha de cambiarse el sistema después que se haya entregado?

El resultado final de la fase de análisis es una *especificación de los requisitos del software*.

- Descripción del problema previa y detalladamente.
 - Prototipos de programas pueden clasificar el problema.

1.5.2. Diseño

La especificación de un sistema indica *lo que* el sistema debe *hacer*. La etapa de diseño del sistema indica *cómo* ha de hacerse. Para un sistema pequeño, la etapa de diseño puede ser tan sencilla como escribir un algoritmo en pseudocódigo. Para un sistema grande, esta etapa incluye también la fase de diseño de algoritmos, pero incluye el diseño e interacción de un número de algoritmos diferentes, con frecuencia sólo bosquejados, así como una estrategia para cumplir todos los detalles y producir el código correspondiente.

Es preciso determinar si se pueden utilizar programas o subprogramas que ya existen o es preciso construirlos totalmente. El proyecto se ha de dividir en módulos utilizando los principios de diseño descendente. A continuación se debe indicar la interacción entre módulos; un diagrama de estructuras proporciona un esquema claro de estas relaciones².

En este punto, es importante especificar claramente no sólo el propósito de cada módulo, sino también el *flujo de datos* entre módulos. Por ejemplo, se debe responder a las siguientes preguntas: ¿Qué datos están disponibles al módulo antes de su ejecución? ¿Qué supone el módulo? ¿Qué hacen los datos después de que se ejecuta el módulo? Por consiguiente, se deben especificar en detalle las hipótesis, entrada y salida para cada módulo. Un medio para realizar estas especificaciones es escribir una *precondición*, que es una descripción de las condiciones que deben cumplirse al principio del módulo, y una *postcondición*, que es una descripción de las condiciones al final de un módulo. Por ejemplo, se puede describir un procedimiento que ordena una lista (un array) de la forma siguiente:

```
procedure ordenar (A, n)
(Ordena una lista en orden ascendente
 precondición: A es un array de N enteros, 1<= n <= Max.
 postcondición: A[1] <= A[2] <...<= A[n], n es inalterable)
```

Por último, se puede utilizar pseudocódigo³ para especificar los detalles del algoritmo. Es importante que se emplee bastante tiempo en la fase de diseño de sus programas. El resultado final de diseño descendente es una solución que sea fácil de traducir en estructuras de control y estructuras de datos de un lenguaje de programación específico, en nuestro caso, Turbo Borland Pascal.

El gasto de tiempo en la fase de diseño será ahorro de tiempo cuando se escriba y depura su programa.

1.5.3. Implementación (codificación)

La etapa de *implementación (codificación)* traduce los algoritmos del diseño en un programa escrito en un lenguaje de programación. Los algoritmos y las estructuras de datos realizadas en pseudocódigo han de traducirse a un lenguaje que entienda la computadora.

² Para ampliar sobre este tema de diagramas de estructuras, puede consultar estas obras nuestras: *Fundamentos de programación*, 2.^a edición, McGraw-Hill, 1992; *Problemas de metodología de la programación*, McGraw-Hill, 1992, o bien la obra *Pascal y Turbo Pascal. Un enfoque práctico*, de Joyanes, Zahonero y Hermoso, en McGraw-Hill, 1995.

³ Para consultar el tema del *pseudocódigo*, véanse las obras: *Fundamentos de programación. Algoritmos y estructuras de datos*, 2.^a edición, McGraw-Hill, 1996, de Luis Joyanes, y *Fundamentos de programación. Libro de problemas*, McGraw-Hill, 1996, de Luis Joyanes, Luis Rodríguez y Matilde Fernández.

La codificación ha de realizarse en un lenguaje de programación. Los lenguajes clásicos más populares son PASCAL, FORTRAN, COBOL y C; los lenguajes orientados a objetos más usuales son C++, Java, Visual BASIC, Smalltalk, etc.

La codificación cuando un problema se divide en subproblemas, los algoritmos que resuelven cada subproblema (tarea o módulo) deben ser codificados, depurados y probados independientemente.

Es relativamente fácil encontrar un error en un procedimiento pequeño. Es casi imposible encontrar todos los errores de un programa grande, que se codificó y comprobó como una sola unidad en lugar de como una colección de módulos (procedimientos) bien definidos.

Las reglas del sangrado (*indentación*) y buenos comentarios facilitan la escritura del código. El *pseudocódigo* es una herramienta excelente que facilita notablemente la codificación.

1.5.4. Pruebas e integración

Cuando los diferentes componentes de un programa se han implementado y comprobado individualmente, el sistema completo se ensambla y se integra.

La etapa de pruebas sirve para mostrar que un programa es correcto. Las pruebas nunca son fáciles. Edgar Dijkstra ha escrito que mientras que las pruebas realmente muestran la *presencia* de errores, nunca puede mostrar su *ausencia*. Una prueba con «éxito» en la ejecución significa sólo que no se han descubierto errores en esas circunstancias específicas, pero no se dice nada de otras circunstancias. En teoría el único modo que una prueba puede mostrar que un programa es correcto, es verificar si *todos* los casos posibles se han intentado y comprobado (es lo que se conoce como *prueba exhaustiva*); es una situación técnicamente imposible incluso para los programas más sencillos. Supongamos, por ejemplo, que se ha escrito un programa que calcule la nota media de un examen. Una prueba exhaustiva requerirá todas las combinaciones posibles de marcas y tamaños de clases; puede llevar muchos años completar la prueba.

La fase de pruebas es una parte esencial de un proyecto de programación. Durante la fase de pruebas se necesita eliminar tantos errores lógicos como pueda. En primer lugar, se debe probar el programa con datos de entrada válidos que conducen a una solución conocida. Si ciertos datos deben estar dentro de un rango, se deben incluir los valores en los extremos finales del rango. Por ejemplo, si el valor de entrada de n cae en el rango de 1 a 10, se ha de asegurar incluir casos de prueba en los que n esté entre 1 y 10. También se deben incluir datos no válidos para comprobar la capacidad de detección de errores del programa. Se han de probar también algunos datos aleatorios y por último intentar algunos datos reales.

1.5.5. Verificación

La etapa de pruebas ha de comenzar tan pronto como sea posible en la fase de diseño y continuará a lo largo de la implementación del sistema. Incluso aunque las pruebas son herramientas extremadamente válidas para proporcionar la evidencia de que un programa es correcto y cumple sus especificaciones, es difícil conocer si las pruebas realizadas

son suficientes. Por ejemplo, ¿cómo se puede conocer que son suficientes los diferentes conjuntos de datos de prueba o que se han ejecutado todos los caminos posibles a través del programa?

Por esas razones se ha desarrollado un segundo método para demostrar la corrección o exactitud de un programa. Este método, denominado *verificación formal*, implica la construcción de pruebas matemáticas que ayudan a determinar si los programas hacen lo que se supone han de hacer. La verificación formal implica la aplicación de reglas formales para mostrar que un programa cumple su especificación: la verificación. La verificación formal funciona bien en programas pequeños, pero es compleja cuando se utiliza en programas grandes. La teoría de la verificación requiere conocimientos matemáticos avanzados y por otra parte se sale fuera de los objetivos de este libro; por esta razón sólo hemos constatado la importancia de esta etapa.

La prueba de que un algoritmo es correcto es como probar un teorema matemático. Por ejemplo, probar que un módulo es exacto (correcto) comienza con las precondiciones (axiomas e hipótesis en matemáticas) y muestra que las etapas del algoritmo conducen a las postcondiciones. La verificación trata de probar con medios matemáticos que los algoritmos son correctos.

Si se descubre un error durante el proceso de verificación, se debe corregir su algoritmo y posiblemente se han de modificar las especificaciones del problema. Un método es utilizar *invariantes* (una condición que siempre es verdadera en un punto específico de un algoritmo), lo que probablemente hará que su algoritmo contenga pocos errores *antes* de que comience la codificación. Como resultado se gastará menos tiempo en la depuración de su programa.

1.5.6. Mantenimiento

Cuando el producto software (el programa) se ha terminado, se distribuye entre los posibles usuarios, se instala en las computadoras y se utiliza (*producción*). Sin embargo, y aunque *a priori* el programa funcione correctamente, el software debe ser mantenido y actualizado. De hecho, el coste típico del mantenimiento excede, con creces, el coste de producción del sistema original.

Un sistema de software producirá errores que serán detectados, casi con seguridad, por los usuarios del sistema y que no se descubrieron durante la fase de prueba. La corrección de estos errores es parte del mantenimiento del software. Otro aspecto de la fase de mantenimiento es la mejora del software añadiendo más características o modificando partes existentes que se adapten mejor a los usuarios.

Otras causas que obligarán a revisar el sistema de software en la etapa de mantenimiento son las siguientes: 1) Cuando un nuevo *hardware* se introduce, el sistema puede ser modificado para ejecutarlo en un nuevo entorno; 2) Si cambian las necesidades del usuario, suele ser menos caro y más rápido, modificar el sistema existente que producir un sistema totalmente nuevo. La mayor parte del tiempo de los programadores de un sistema se gasta en el mantenimiento de los sistemas existentes y no en el diseño de sistemas totalmente nuevos. Por esta causa, entre otras, se ha de tratar siempre de diseñar programas de modo que sean fáciles de comprender y entender (legibles) y fáciles de cambiar.

1.5.7. La obsolescencia: programas obsoletos

La última etapa en el ciclo de vida del software es la evolución del mismo, pasando por su vida útil hasta su *obsolescencia* o fase en la que el software se queda anticuado y es preciso actualizarlo o escribir un nuevo programa sustitutorio del antiguo.

La decisión de dar de baja un software por obsoleto no es una decisión fácil. Un sistema grande representa una inversión enorme de capital que parece, a primera vista, más barato modificar el sistema existente, en vez de construir un sistema totalmente nuevo. Este criterio suele ser, normalmente, correcto y por esta causa los sistemas grandes se diseñan para ser modificados. Un sistema puede ser productivamente revisado muchas veces. Sin embargo, incluso los programas grandes se quedan obsoletos por caducidad de tiempo al pasar una fecha límite determinada. A menos que un programa grande esté bien escrito y adecuado a la tarea a realizar, como en el caso de programas pequeños, suele ser más eficiente escribir un nuevo programa que corregir el programa antiguo.

1.5.8. Iteración y evolución del software

Las etapas de vida del software suelen formar parte de un ciclo o bucle, como su nombre sugiere, y no son simplemente una lista lineal. Es probable, por ejemplo, que durante la fase de mantenimiento tenga que volver a las especificaciones del problema para verificarlas o modificarlas.

Obsérvese en la Figura 1.4 que las diferentes etapas rodean al núcleo documentación. La documentación no es una etapa independiente como se puede esperar sino que está integrada en todas las etapas del ciclo de vida del software.



Figura 1.4. Etapas del ciclo de vida del software cuyo núcleo aglutinador es la documentación.

1.6. MÉTODOS FORMALES DE VERIFICACIÓN DE PROGRAMAS

Aunque la verificación formal de programas se sale fuera del ámbito de este libro, por su importancia vamos a considerar dos conceptos clave, *asertos* (afirmaciones) y *precondiciones/postcondiciones invariantes* que ayuden a documentar, corregir y clarificar el diseño de módulos y de programas.

1.6.1. Aserciones (*asertos*)⁴

Una parte importante de una verificación formal es la documentación de un programa a través de *asertos* o *aserciones*, sentencias lógicas acerca del programa que se declaran «verdaderas». Un aserto se escribe como un comentario y describe lo que se supone sea verdadero sobre las variables del programa en ese punto.

Una *aserción (aserto)* es una frase sobre una condición específica en un cierto punto de un algoritmo o programa.

EJEMPLO 1.1

El siguiente fragmento de programa contiene una secuencia de sentencias de asignación, seguidas por un aserto:

```
A := 10;           {aserto : A es 10}
X := A;           {aserto : X es 10}
Y := X + A;       {aserto : Y es 20}
```

La verdad de la primera afirmación, {A es 10}, sigue a la ejecución de la primera sentencia con el conocimiento de que 10 es una constante. La verdad de la segunda afirmación, {X es 10}, sigue de la ejecución de X := A con el conocimiento de que A es 10. La verdad de la tercera afirmación, {Y es 20}, sigue de la ejecución Y := X + A con el conocimiento de que X es 10 y A es 10. En este segmento del programa se utilizan afirmaciones como comentarios para documentar el cambio en una variable de programa después que se ejecuta cada sentencia de afirmación.

La tarea de utilizar verificación formal es probar que un segmento de programa cumple su especificación. La afirmación final se llama *postcondición* (en este caso, {Y es 20}) y sigue a la presunción inicial o *precondición* (en este caso, {10 es una constante}) después que se ejecuta el segmento de programa.

* Este término se suele traducir también por *afirmaciones* o *declaraciones*. El término *aserto* está más extendido en la jerga informática, y al igual que *aserto*, están los dos términos admitidos por el DRAE.

1.6.2. Precondiciones y postcondiciones

Las precondiciones y postcondiciones —ya citadas anteriormente— son afirmaciones sencillas sobre condiciones al principio y al final de los módulos. Una *precondición* de un procedimiento es una afirmación lógica sobre sus parámetros de entrada; se supone que es *verdadera* cuando se llama al procedimiento. Una *postcondición* de un procedimiento puede ser una afirmación lógica que describe el cambio en el *estado del programa* producido por la ejecución del procedimiento; la postcondición describe el efecto de llamar al procedimiento. En otras palabras, la postcondición indica que será verdadera después que se ejecute el procedimiento.

EJEMPLO 1.2

```
{Precondiciones y postcondiciones del procedimiento LeerEnteros}
procedure LeerEnteros (Min, Max:Integer; var N: Integer);
{
    Lectura de un entero entre Min y Max en N
    Pre : Min y Max son valores asignados
    Post: devuelve en N el primer valor del dato entre Min y Max
          si Min <= Max es verdadero; en caso contrario
          N no está definido.
}
```

La precondición indica que los parámetros de entrada Min y Max se definen antes de que comience la ejecución del procedimiento. La postcondición indica que la ejecución del procedimiento asigna el primer dato entre Min y Max al parámetro de salida siempre que $\text{Min} \leq \text{Max}$ sea verdadero.

Las precondiciones y postcondiciones son más que un método para resumir acciones de un procedimiento. La declaración de estas condiciones debe ser la primera etapa en el diseño y escritura de un procedimiento. Es conveniente en la escritura de algoritmos de procedimientos, se escriba la cabecera del procedimiento que muestra los parámetros afectados por el procedimiento así como unos comentarios de cabecera que contienen las precondiciones y postcondiciones.

- **Precondición:** Predicado lógico que debe cumplirse al comenzar la ejecución de una operación.
- **Postcondición:** Predicado lógico que debe cumplirse al acabar la ejecución de una operación; siempre que se haya cumplido previamente la precondición correspondiente.

1.6.3. Reglas para prueba de programas

Un medio útil para probar que un programa P hace lo que realmente ha de hacer es proporcionar *aserciones (asertos)* que expresen las condiciones antes y después de que P sea ejecutada. En realidad las aserciones son como sentencias o declaraciones que pueden ser o bien *verdaderas* o bien *falsas*.

La primera aserción, la *precondición*, describe las condiciones que han de ser verdaderas antes de ejecutar P.

La segunda aserción, la *postcondición*, describe las condiciones que han de ser verdaderas después de que P se ha ejecutado (suponiendo que la precondición fue verdadera antes). El modelo general es:

```
(precondición)  {= condiciones lógicas que son
                  verdaderas antes de que P se ejecute}
(postcondición) {= condiciones lógicas que son verdaderas
                  después de que P se ejecute}
```

EJEMPLO 1.3

El procedimiento OrdenarSeleccion (A, m, n) ordena a los elementos del array A[m..n] en orden descendente. El modelo correspondiente puede escribirse así:

```
{m ≤ n}           {precondición: A ha de tener al menos 1 elemento}
OrdenarSeleccion (A,m,n)           {programa de ordenación a ejecutar}
{A[m] ≥ A[m+1] ≥ ... ≥ A[n]}     {postcondición: elementos de A en orden
                                    descendente}
```

PROBLEMA 1.1

Encontrar la posición del elemento mayor de una lista con indicación de precondiciones y postcondiciones.

```
function EncontrarMax (var A:Lista; m, n: Integer): Integer;
{precondición : m < n}
{postcondición : devuelve posición elemento mayor en A[m..n]}
  var i, j : Integer;

begin
  i := m;
  j := n; {aserción}
  repeat
    i := i + 10;
    if A[i] > A[j] then j := i;
  until i = n;
  EncontrarMax := j; {devuelve j como elemento mayor}
end;
```

PROBLEMA 1.2

Ordenar por el método selección con precondiciones y postcondiciones:

```
procedure OrdenarSeleccion (var A:Lista; m,n : Integer);
{precondición : m ≤ n}
{postcondición : A[m..n] está ordenado tal que
  A[m] ≥ A[m+1] ≥ ... ≥ A[n]}
```

```

var PosicionMax, Aux : Integer;
begin
  if m < n then
  begin
    PosicionMax := EncontrarMax(A,m,n);
    (Intercambiar A[m] ↔ A[PosicionMax])
    Aux := A[m];
    A[m] := A[PosicionMax];
    A[PosicionMax] := Aux;
    OrdenarSeleccion (A,m+1,n);
    (produce : A[m+1] ≥ A[m+2] ≥ ...≥ A[n])
  end {if}
  (Aserción final: A[m] ≥ A[m+1] ≥ ...≥ A[n])
end {OrdenarSeleccion}

```

1.6.4. Invariantes de bucles

Una *invariante de bucle* es una condición que es verdadera antes y después de la ejecución de un bucle. Las invariantes de bucles se utilizan para demostrar la corrección (exactitud) de algoritmos iterativos. Utilizando invariantes, se pueden detectar errores antes de comenzar la codificación y por esa razón reducir tiempo de depuración y prueba.

EJEMPLO 1.4

Un bucle que calcula la suma de los n primeros elementos del array (lista) A:

```

{calcular la suma de A[1], A[2],...,A[n]}
{aserción n >= 14}
Suma := 0;
j := 1;
while j <= n do
begin
  Suma := Suma + A[j];
  j := succ(j)
end

```

Antes de que este bucle comience la ejecución Suma es 0 y j es 1. Después que el bucle se ha ejecutado una vez, Suma es A[1] y j es 2.

Invariante del bucle Suma es la suma de los elementos A[1] a A[j+1].

Un invariante es un predicado que cumple tanto antes como después de cada iteración (vuelta) y que describe la misión del bucle.

EJEMPLO 1.5

Invariante de un bucle que sume n enteros (n entero positivo); i, Suma, n son de tipo entero.

20 Estructura de datos

```
{aserción n>= 1} {precondición}
suma := 0;
i := 1;
while i <= n do
begin
    Suma := Suma + i;
    i := i + 1
end
{aserción: Suma es 1+2+3+...+n-1+n} {postcondición}
```

Una invariante del bucle puede ser:

```
{invariante: i <= n y Suma es 1+2+..i-1}
```

lo que significa: i debe ser menor que o igual que n y después de cada pasada o vuelta, Suma es igual a la suma de todos los enteros positivos menores que i.

En la verificación de programas la invariante del bucle se utiliza para probar que el bucle cumple su especificación. Para nuestro propósito, se puede utilizar el invariante del bucle para documentar el comportamiento del mismo y se situará justo antes del cuerpo del bucle.

```
{Suma de enteros 1 a n}
{precondición : n >= 1}
Suma := 0;
i := 1;
while i <= n do
    {invariante: i <= n+1 y Suma es 1+2+...i-1}
    begin
        Suma := Suma + i;
        i := i+1
    end;
{postcondición : Suma es 1+2+3+..n-1+n}
```

Invariantes de bucle como herramientas de diseño

Otra aplicación de los invariantes de bucle es la especificación del bucle: iniciación, condición de repetición y cuerpo del bucle.

EJEMPLO 1.6

Si la variante de un bucle es:

```
{invariante : i <= n y Suma es la suma de todos los números leídos del teclado}
```

Se puede deducir que:

```
Suma := 0.0;           {iniciación;
i := 0;                 {condición/prueba del bucle}
i < n
Read (Item);
Suma := Suma + Item;   {cuerpo del bucle}
i := i + 1;
```

Con toda esta información es una tarea fácil escribir el bucle de suma:

```
Suma := 0.0;
i := 0;
while i < n do  {y, toma los valores 0,1,2,3,...n-1}
    Read (Item);
    Suma := Suma + Item;
    i := i + 1
end;
```

EJEMPLO 1.7

En los bucles for es posible declarar también invariantes, pero teniendo presente la particularidad de esta sentencia: la variable de control del bucle es indefinida después que se sale del bucle, por lo que para definir su invariante se ha de considerar que dicha variable de control se incrementa antes de salir del bucle y mantiene su valor final.

```
{precondición n >= 1}
Suma := 0;
for i := 1 to n do
    {invariante : i <= n+1 y Suma es 1+2+...i-1}
    Suma := Suma + i;
{postcondición: Suma es 1+2+3+..n-1+n}
```

PROBLEMA 1.3

Escribir un bucle controlado por centinela que calcule el producto de un conjunto de datos.

```
{Calcular el producto de una serie de datos}
{precondición : centinela es constante}
Producto := 1;
Writeln ('Para terminar, introduzca', Centinela:1);
Writeln ('Introduzca número:');
ReadLn (Numero);
while Numero <> Centinela do
    {invariante: Producto es el producto de todos los valores leídos en
     Número y ninguno era el Centinela}
begin
    Producto := Producto * Numero;
    WriteLn ('Introduzca número siguiente:');
    ReadLn (Numero)
end;
{postcondición: Producto es el producto de todos los números leídos en
Número antes del centinela}
```

1.6.5. Etapas a establecer la exactitud (corrección) de un programa

Se pueden utilizar invariantes para establecer la corrección (exactitud) de un algoritmo iterativo. Supongamos el algoritmo ya estudiado anteriormente.

```
(calcular la suma de A[1], A[2],...,A[n])
Suma := 0;
j := 1;
while j <= N do
begin
  Suma := Suma + A[j];
  j     := succ(j)
end
{invariante: Suma es la suma de los elementos A[1] a A[j-1]}
```

Los siguientes cuatro puntos han de ser verdaderos⁵:

1. **El invariante debe ser inicialmente verdadero**, antes de que comience la ejecución por primera vez del bucle. En el ejemplo anterior, Suma es 0 y j es 1 inicialmente. En este caso, el invariante significa que Suma contiene la suma de los elementos A[1] a A[0], que es verdad ya que no hay elementos en este rango.
2. **Una ejecución del bucle debe mantener el invariante**. Esto es si el invariante es verdadero antes de cualquier iteración del bucle, entonces se debe demostrar que es verdadero después de la iteración. En el ejemplo, el bucle añade A[j] a Suma y a continuación incrementa j en 1. Por consiguiente, después de una ejecución del bucle, el elemento añadido más recientemente a Suma es A[j-1]; esto es, el invariante que es verdadero después de la iteración.
3. **El invariante debe capturar la exactitud del algoritmo**. Esto es, debe demostrar que si el invariante es verdadero cuando termina el bucle, el algoritmo es correcto. Cuando el bucle del ejemplo termina, j contiene N + 1 y el invariante es verdadero: Suma contiene la suma de los elementos A[1] a A[j-1], que es la suma que se trata de calcular.
4. **El bucle debe terminar**. Esto es, se debe demostrar que el bucle termina después de un número finito de iteraciones. En el ejemplo, j comienza en 1 y a continuación se incrementa en 1 en cada ejecución del bucle. Por consiguiente, j eventualmente excederá a N con independencia del valor de N. Este hecho y la característica fundamental de while garantizan que el bucle terminará.

La identificación de invariantes de bucles ayuda a escribir bucles correctos. Se representa el invariante como un comentario que precede a cada bucle. En el ejemplo anterior:

```
{Invariante: 1 <= j <= N+1 y Suma = A[1]+...+A(j-1)}
while j <= N do
```

⁵ Carrasco, Helman y Verof, *op. cit.*, pág. 15.

1.7. PRINCIPIOS DE DISEÑO DE SISTEMAS DE SOFTWARE

El diseño de sistemas de software de calidad requiere el cumplimiento de una serie de características y objetivos. En un sentido general los objetivos a conseguir que se consideran útiles en el diseño de sistemas incluyen al menos los siguientes principios:

1. Modularidad mediante diseño descendente.
2. Abstracción y ocultamiento de la información.
3. Modificabilidad.
4. Comprensibilidad y fiabilidad.
5. Interfaces de usuario.
6. Programación segura contra fallos.
7. Facilidad de uso.
8. Eficiencia.
9. Estilo de programación.
10. Depuración.
11. Documentación.

1.7.1. Modularidad mediante diseño descendente

Un principio importante que ayuda a tratar la complejidad de un sistema es la modularidad. La descomposición del problema se realiza a través de un diseño descendente que a través de niveles sucesivos de refinamiento se obtendrán diferentes módulos. Normalmente los módulos de alto nivel especifican qué acciones han de realizarse mientras que los módulos de bajo nivel definen cómo se realizan las acciones.

La programación modular tiene muchas ventajas. A medida que el tamaño de un programa crece, muchas tareas de programación se hacen más difíciles. La diferencia principal entre un programa modular pequeño y un programa modular grande es, simplemente, el número de módulos que cada uno contiene, ya que el trabajo con programas modulares es similar y sólo se ha de tener presente el modo en que unos módulos interactúan con otros.

La modularidad tiene un impacto positivo en los siguientes aspectos de la programación:

- **Construcción del programa.** La descomposición de un programa en módulos permite que los diversos programadores trabajen de modo independiente en cada uno de sus módulos. El trabajo de módulos independientes convierte la *tarea de escribir un programa grande en la tarea de escribir muchos programas pequeños*.
- **Depuración del programa.** La depuración de programas grandes puede ser una tarea enorme, de modo que se facilitará esa tarea, al centrarse en la depuración de pequeños programas más fáciles de verificar.
- **Legibilidad.** Los programas grandes son muy difíciles de leer, mientras que los programas modulares son más fáciles de leer.

- **Eliminación de código redundante.** Otra ventaja del diseño modular es que se pueden identificar operaciones que suceden en muchas partes diferentes del programa y se implementan como subprogramas. Esto significa que el código de una operación aparecerá sólo una vez, produciendo como resultado un aumento en la legibilidad y modificabilidad.

1.7.2. Abstracción y encapsulamiento

La complejidad de un sistema puede ser gestionado utilizando *abstracción*. La abstracción es un principio común que se aplica en muchas situaciones. La idea principal es definir una parte de un sistema de modo que puede ser comprendido por sí mismo (esto es, como una unidad) sin conocimiento de sus detalles específicos y sin conocimiento de cómo se utiliza esta unidad a un nivel más alto.

Existen dos tipos de abstracciones: *abstracción procedimental* y *abstracción de datos*. La mayoría de los lenguajes de programación soportan este tipo de abstracción. Es aquella en que se separa el propósito de un subprograma de su implementación. Una vez que se ha escrito un subprograma, se puede utilizar sin necesidad de conocer las peculiaridades de sus algoritmos. Suponiendo que el subprograma esté documentado adecuadamente, se podrá utilizar con sólo conocer la cabecera del mismo y sus comentarios descriptivos; no necesitará conocer su código.

La modularidad —tratada anteriormente— y la abstracción procedural se complementan entre si. La modularidad implica la rotura de una solución en módulos; la abstracción procedural implica la especificación de cada módulo claramente *antes* de que se implemente en Pascal. De hecho, lo importante es poder utilizar los subprogramas predefinidos, tales como `Writeln`, `Sqrt`, etc., o bien los definidos por el usuario sin necesidad de conocer sus algoritmos.

El otro tipo de abstracción es la *abstracción de datos*, soportada hoy día por diversos lenguajes Turbo Borland Pascal, C++, Ada-83, Ada-95, Modula-2, etc. El propósito de la abstracción de datos es aislar cada estructura de datos y sus acciones asociadas. Es decir, se centra la abstracción de datos en las operaciones que se realizan sobre los datos en lugar de cómo se implementan las operaciones. Supongamos, por ejemplo, que se tiene una estructura de datos `Clientes`, que se utiliza para contener información sobre los clientes de una empresa, y que las operaciones o acciones a realizar sobre esta estructura de datos incluyen *Insertar*, *Buscar* y *Borrar*. El *módulo*, *objeto* o tipo *abstracto de datos*, `TipoCliente` es una colección de datos y un conjunto de operaciones sobre esos datos. Tales operaciones pueden añadir nuevos datos, buscar o eliminar datos. Estas operaciones constituyen su *interfaz*, mediante la cual se comunica con otros módulos u objetos.

Un *tipo abstracto de datos* (TAD) se implementará mediante **unidades** en Turbo Borland Pascal. Por su importancia se dedicará un capítulo completo a tratar más detenidamente el concepto de un TAD, su diseño e implementación práctica (capítulo 3).

Otro principio de diseño es la **ocultación de la información**. El propósito de la ocultación de la información es hacer inaccesible ciertos detalles que no afecten a los otros módulos del sistema. Por consiguiente, el objeto y sus acciones constituyen un sistema cerrado, cuyos detalles se ocultan a los otros módulos.

La abstracción identifica los aspectos esenciales de módulos y estructura de datos, que se pueden tratar como cajas negras. La abstracción indica especificaciones funcionales de cada caja negra; es responsable de su vista externa o *pública*. Sin embargo, la abstracción ayuda también a identificar detalles de lo que se debe *ocultar* de la vista pública —detalles que no están en las especificaciones pero deben ser *privados*—. El principio de *ocultación de la información* no sólo oculta detalles dentro de la caja negra sino que también asegura que ninguna otra caja negra pueda acceder a estos detalles ocultos. Por consiguiente, se deben ocultar ciertos detalles dentro de sus módulos y TAD, y hacerlos inaccesibles a los restantes módulos y TAD.

La abstracción de datos y su expresión más clara el tipo abstracto de datos, se implementa en Turbo Borland Pascal con **unidades**.

1.7.3. **Modificabilidad**

La *modificabilidad* (facilidad de modificación) se refiere a los cambios controlados de un sistema dado. Un sistema se dice que es *modificable* si los cambios en los requisitos pueden adecuarse bien a los cambios en el código. Es decir, un pequeño cambio en los requisitos en un programa modular normalmente requiere un cambio pequeño sólo en algunos de sus módulos; es decir, cuando los módulos son independientes (esto es, débilmente acoplados) y cada módulo realiza una tarea bien definida (esto es, altamente *cohesivos*). La *modularidad aísla las modificaciones*.

Las técnicas más frecuentes para hacer que un programa sea fácil de modificar son: uso de subprogramas y uso de constantes definidas por el usuario.

El uso de procedimientos tiene la ventaja evidente, no sólo de eliminar código redundante sino también hace el programa resultante más modificable. Normalmente será un signo de mal diseño de un programa que pequeñas modificaciones a un programa requieran su reescritura completa. Un programa bien estructurado en módulos será modificable más fácilmente; es decir, si cada módulo resuelve sólo una pequeña parte del problema global, un cambio pequeño en las especificaciones del problema normalmente sólo afectará a unos pocos módulos y en consecuencia eso facilitará su modificación.

Las constantes definidas por el usuario o con nombre son otro medio para mejorar la modificabilidad de un programa.

EJEMPLO 1.8

Los límites del rango de un array suelen ser definidos mejor mediante constantes con nombre que mediante constantes numéricas. Así, la declaración típica de un array y su proceso posterior mediante un bucle es:

```
type TipoPuntos = array [1..100] of integer;
for i := 1 to 100 do
  proceso de los elementos
```

El diseño más eficiente podría ser:

```
const NumeroDeItems = 100;
type TipoPunto = array [1..NumeroDeItems] of integer;
for i := 1 to NumeroDeItems do
proceso de los elementos
```

ya que cuando se desee cambiar el número de elementos del array sólo sería necesario cambiar el valor de la constante `NumeroDeItems`, mientras que en el caso anterior supondrá cambiar la declaración del tipo y el índice de bucle, mientras que en el segundo caso sólo el valor de la constante.

1.7.4. Comprensibilidad y fiabilidad

Un sistema se dice que es *comprendible* si refleja directamente una visión natural del mundo⁶. Una característica de un sistema eficaz es la *simplicidad*. En general, un sistema sencillo puede ser comprendido más fácilmente que uno complejo.

Un objetivo importante en la producción de sistemas es el de la fiabilidad. El objetivo de crear programas fiables ha de ser crítico en la mayoría de las situaciones.

1.7.5. Interfaces de usuario

Otro criterio importante a tener presente es el diseño de la interfaz del usuario. Algunas directrices a tener en cuenta pueden ser:

- En un entorno interactivo, se ha de tener en cuenta las preguntas posibles al usuario y sobre todo aquellas que solicitan entradas de usuario.
- Es conveniente que se realicen eco de las entradas de un programa. Siempre que un programa lee datos, bien de usuario a través de un terminal o de un archivo, el programa debe incluir los valores leídos en su salida.
- Etiquetar (rotular) la salida con cabeceras y mensajes adecuados.

1.7.6. Programación segura contra fallos

Un programa es seguro contra fallos cuando se ejecuta razonablemente por cualquiera que lo utilice. Para conseguir este objetivo se han de comprobar *los errores en datos de entrada y en la lógica del programa*.

Supongamos un programa que espera leer datos enteros positivos pero lee ,25. Un mensaje típico a visualizar ante este error suele ser:

Error de rango

⁶ Tremblay, Donrek y Bunt: *Introduction to Computer Science. An Algorithmic approach*, McGraw-Hill, 1989, pág. 440.

Sin embargo, es más útil un mensaje tal como éste:

```
-25 no es un número válido de años
Por favor vuelva a introducir el número
```

Otras reglas prácticas a considerar son:

- No utilizar tipos subrango para detectar datos de entrada no válidos. Por ejemplo, si se desea comprobar que determinados tipos nunca sean negativos, se pueden cambiar las definiciones de tipo global a:

```
type TipoNoNeg = 0..maxint; {tipo nuevo}
  TipoMillar = Bajo..Alto; {permanece el mismo}
  TipoTabla = array[TipoMillar] of TipoNoNeg;
    {un array de este tipo contiene sólo enteros no negativos}
```

- Comprobar datos de entrada no válidos

```
ReadLn (Grupo, Número)
...
if Número >= 0
  then agregar Número a total
  else manejar el error.
```

- Cada subprograma debe comprobar los valores de sus parámetros. Así, en el caso de la función SumaIntervalo que suma todos los enteros comprendidos entre m y n .

```
function SumaIntervalo {m,n:Integer} : Integer;
{
  precondition : m y n son enteros tales que m <= n
  postcondición: Devuelve SumaIntervalo = m+(m+1)+...+n
                  m y n son inalterables
}
var Suma, Indice : Integer;
begin
  Suma := 0;
  for Indice := m to n do
    Suma := Suma + Indice;
  SumaIntervalo := Suma
end;
```

1.7.7. Facilidad de uso

La *utilidad* de un sistema se refiere a su facilidad de uso. Esta propiedad ha de tenerse presente en todas las etapas del ciclo de vida, pero es vital en la fase de diseño e implementación o construcción.

1.7.8. Eficiencia

El objetivo de la eficiencia es hacer un uso óptimo de los recursos del programa. Tradicionalmente, la eficiencia ha implicado recursos de tiempo y espacio. Un sistema eficiente

es aquel cuya velocidad es mayor con el menor espacio de memoria ocupada. En tiempos pasados los recursos de memoria principal y de CPU eran factores clave a considerar para aumentar la velocidad de ejecución. En el año 1997 las CPU (UCP, Unidad Central de Proceso) típicas de los PC eran Pentium de 166 y 200 MHz. En el año 1988 son usuales los Pentium de 233 Mhz y Pentium II de 266 Mhz y son ya muy frecuentes; las memorias centrales usuales son 16 Mbytes o 32 Mbytes, aunque básicamente los tamaños de memorias tradicionales para trabajos profesionales ya son de un mínimo de 64 Mbytes; el factor eficiencia ya no se mide con los mismos parámetros de memoria y tiempo. Hoy día debe existir un compromiso entre legibilidad, modificabilidad y eficiencia, aunque, con excepciones, prevalecerá la legibilidad y modificabilidad.

1.7.9. Estilo de programación, documentación y depuración

Estas características hoy día son claves en el diseño y construcción de programas, por esta causa dedicaremos por su especial importancia tres secciones independientes para tratar estos criterios de diseño.

1.8. ESTILO DE PROGRAMACIÓN

Una de las características más importantes en la construcción de programas, sobre todo los de gran tamaño, es el estilo de programación. La buena calidad en la producción de programas tiene relación directa con la escritura de los mismos, su legibilidad y comprensibilidad. Un buen estilo de programación suele venir con la práctica, pero el requerimiento de unas reglas de escritura del programa, al igual que sucede con la sintaxis y reglas de escritura de un lenguaje natural humano, debe buscar esencialmente que no sólo sean legibles y modificables por las personas que lo han construido sino también —y esencialmente— puedan ser leídos y modificados por otras personas distintas. No existe una fórmula mágica que garantice programas legibles, pero existen diferentes reglas que facilitarán la tarea y con las que prácticamente suelen estar de acuerdo casi todos, desde programadores novatos a ingenieros de software experimentados.

Naturalmente las reglas de estilo para construir programas claros, legibles y fácilmente modificables, dependerá del tipo de programación y lenguaje elegido. En nuestro caso y dado que el tipo de programación es estructurado y el lenguaje es Pascal/Turbo Borland Pascal, nos centraremos en este enfoque, pero estas reglas serán fácilmente extrapolables a otros lenguajes estructurados tales como C, C++, Ada, Modula-2, etc.

Reglas de estilo de programación

1. Modularizar un programa en partes coherentes (uso amplio de subprogramas).
2. Evitar variables globales en subprogramas.
3. Usar nombres significativos para identificadores.
4. Definir constantes con nombres al principio del programa.
5. Evitar el uso del `goto` y no escribir nunca código *spaghetti*.

6. Escribir subrutinas cortas que hagan una sola cosa y bien.
7. Uso adecuado de parámetros variable.
8. Usar declaraciones de tipos.
9. Presentación (comentarios adecuados).
10. Manejo de errores.
11. Legibilidad.
12. Documentación.

1.8.1. Modularizar un programa en subprogramas

Un programa grande que resuelva un problema complejo siempre ha de dividirse en módulos para ser más manejable. Aunque la división no garantiza un sistema bien organizado será preciso encontrar reglas que permitan conseguir esa buena organización.

Uno de los criterios clave en la división es la independencia; esto es, el acoplamiento de módulos; otro criterio es que cada módulo debe ejecutar una sola tarea, una función relacionada con el problema. Estos criterios fundamentalmente son *acoplamiento* y *cohesión de módulos*, aunque existen otros criterios que no se tratarán en esta sección.

El *acoplamiento* se refiere al grado de interdependencia entre módulos. El grado de acoplamiento se puede utilizar para evaluar la calidad de un diseño de sistema. Es preciso minimizar el acoplamiento entre módulos, es decir, minimizar su interdependencia. El criterio de acoplamiento es una medida para evaluar cómo un sistema ha sido modularizado. Este criterio sugiere que un sistema bien modularizado es aquel en que las interfaces sean claras y sencillas.

Otro criterio para juzgar un diseño es examinar cada módulo de un sistema y determinar la fortaleza de la ligadura (enlace) dentro de ese módulo. La fortaleza interna de un módulo, esto es, lo fuertemente (estrictamente) relacionadas que están entre sí las partes de un módulo, se conoce como propiedad de *cohesión*. Un modelo cuyas partes estén fuertemente relacionadas con cada uno de los otros se dice que es fuertemente cohesivo. Un modelo cuyas partes no están relacionadas con otras se dice que es cohesivo débilmente.

Los módulos de un programa deben estar débilmente acoplados y fuertemente cohesionados.

Como regla general es conveniente utilizar subprogramas ampliamente. Si un conjunto de sentencias realiza una tarea recurrente, repetitiva, identificable, debe ser un subprograma. Sin embargo, una tarea no necesita ser recurrente para justificar el uso de un subprograma.

1.8.2. Evitar variables globales en subprogramas

Una de las principales ventajas de los subprogramas es que pueden implementar el concepto de módulo aislado. El aislamiento se sacrifica cuando un subprograma accede a

variables globales, dado que los efectos de sus acciones producen los efectos laterales indeseados, normalmente.

En general, el uso de variables globales con subprogramas no es correcto. Sin embargo, el uso de la variable global, en sí, no tiene porqué ser perjudicial. Así, si un dato es inherentemente importante en un programa al que casi todo subprograma debe acceder al mismo, entonces ese dato ha de ser global por naturaleza.

1.8.3. Usar nombres significativos para identificadores

Los identificadores que representan los nombres de módulos, subprogramas, funciones, tipos, variables y otros elementos, deben ser elegidos apropiadamente para conseguir programas legibles. El objetivo es usar interfaces *significativas* que ayuden al lector a recordar el propósito de un identificador sin tener que hacer referencia continua a declaraciones o listas externas de variables. Hay que evitar abreviaturas crípticas.

Identificadores largos se deben utilizar para la mayoría de los objetos significativos de un programa, así como los objetos utilizados en muchas posiciones, tales como, por ejemplo, el nombre de un programa usado frecuentemente. Identificadores más cortos se utilizarán estrictamente para objetos locales: así, i, j, k, son útiles para índices de arrays en un bucle, variables contadores de bucle, etc., y son más expresivos que Indice, VariableDeControl, etc.

Los identificadores deben utilizar letras mayúsculas y minúsculas. Cuando un identificador consta de dos o más palabras, cada palabra debe comenzar con una letra mayúscula. Una excepción son los tipos de datos definidos por el usuario que suelen comenzar con una letra minúscula. Así identificadores idóneos son:

SalarioMes NombreMensajeUsuario MensajesDatosMal

Algunas reglas que se pueden seguir son:

- Usar nombres para nombrar objetos de datos tales como variables, constantes y tipos. Utilizar Salario mejor que APagar o Pagar.
- Utilizar verbos para nombrar procedimientos. LeerCaracter, LeerSigCar y CalcularSigMov son procedimientos que realizan estas acciones mejor que SigCar o SigMov (siguiente movimiento).
- Utilizar formas del verbo «ser» o «estar» para funciones lógicas. SonIguales, EsCero, EsListo y EsVacio se utilizan como variables o funciones lógicas.

```
if SonIguales (A, B)
```

Los nombres de los identificadores de objetos deben sugerir el significado del objeto al lector del programa.

1.8.4. Definir constantes con nombres

Se deben evitar constantes explícitas siempre que sea posible. Por ejemplo, no utilizar 7 para el día de la semana o 3.141592 para representar el valor de la constante π . En su lugar, es conveniente definir constantes con nombre que permiten Pascal C..., tal como:

```
Const Pi = 3.141592;
Const NumDiasSemana = 7;
Const Longitud = 45;
```

Este sistema tiene la ventaja de la facilidad para cambiar un valor determinado bien por necesidad o por cualquier error tipográfico

```
Const Longitud = 200;
Const Pi = 3.141592654;
```

1.8.5. Evitar el uso de goto

Uno de los factores que más contribuyen a diseñar programas bien estructurados es un flujo de control ordenado que implica los siguientes pasos:

1. El flujo general de un programa es adelante o directo.
2. La entrada a un módulo sólo se hace al principio y se sale sólo al final.
3. La condición para la terminación de bucles ha de ser clara y uniforme.
4. Los casos alternativos de sentencias condicionales han de ser claros y uniformes.

El uso de una sentencia `goto` casi siempre viola al menos una de estas condiciones. Además es muy difícil verificar la exactitud de un programa que contenga una sentencia `goto`. Por consiguiente, en general, se debe evitar el uso de `goto`. Hay, sin embargo, *raras* situaciones en las que se necesita un flujo de control excepcional, tales casos incluyen aquellos que requieren o bien que un programa termine la ejecución cuando ocurre un error, o bien que un subprograma devuelva el control a su módulo llamador. La inclusión en Turbo Borland Pascal de sentencias `halt` y `exit` hacen innecesario —en cualquier caso— el uso de `goto`.

1.8.6. Uso adecuado de parámetros valor/variable

Un programa interactúa —se comunica— de un modo controlado con el resto del programa mediante el uso de parámetros. Los *parámetros valor*, que son declarados por defecto cuando no se especifica la palabra reservada `var`, pasa los valores al subprograma, pero ningún cambio que el programa hace a estos parámetros se refleja en los parámetros reales de retorno a la rutina llamadora. La comunicación entre la rutina llamadora y el subprograma es de un solo sentido; por esta causa en el caso de módulos aislados se deben utilizar *parámetros valor siempre que sea posible*.

¿Cuándo es adecuado usar *parámetros variable*? La situación más evidente es cuando un procedimiento necesita devolver valores a la rutina llamadora. Sin embargo, si el procedimiento necesita devolver sólo un único valor, puede ser más adecuado usar una función. Si una función no es adecuada, entonces utilizar parámetros variables.

Es conveniente utilizar un parámetro variable para comunicar un valor de retorno del subprograma a la rutina llamadora. Sin embargo, los parámetros variable cuyos valores permanecen inalterables hacen el programa más difícil de leer y más propenso a errores si se requieren modificaciones. La situación es análoga a utilizar una constante en lugar de una variable cuyo valor nunca cambia. Por consiguiente, se debe alcanzar un compromiso entre legibilidad y modificabilidad por un lado y eficiencia por otro. A menos que exista una diferencia significativa en eficiencia, se tomará generalmente el aspecto de la legibilidad y modificabilidad.

1.8.7. Uso adecuado de funciones

Pascal le proporciona un sistema para realizar tareas distintas a las funciones primitivas incorporadas al lenguaje. El mecanismo de llamada a un procedimiento o función definida por el usuario se puede activar desde cualquier punto de un programa en el que se necesite ese subprograma.

En el caso de una función, ésta se debe utilizar siempre que se necesite obtener un único valor. Este uso corresponde a la noción matemática de función. Por consiguiente, es muy extraño que una función realice una tarea diferente de devolver un valor y no debe hacerlo.

Una función no debe hacer nada sino devolver el valor requerido. Es decir, una función nunca tiene un *efecto lateral*.

¿Qué funciones tienen potencial para producir efectos laterales?

- **Funciones con variables globales.** Si una función referencia a una variable global, presenta el peligro de un posible efecto lateral. *En general, las funciones no deben asignar valores a variables globales.*
- **Funciones con parámetros variables.** Un parámetro variable es aquel en que su valor cambiará dentro de la función. Este efecto es *un efecto lateral*. *En general, las funciones no deben utilizar parámetros variables.* Si se necesitan parámetros variables utilizar procedimientos.
- **Funciones que realizan entrada/salida (E/S).** Las E/S son efectos laterales. Las funciones no deben realizar E/S.

1.8.8. Tratamiento de errores

Un programa diseñado ante fallos debe comprobar errores en las entradas y en su lógica e intentar comportarse bien cuando los encuentra. El tratamiento de errores con frecuen-

cia necesita acciones excepcionales que constituirán un mal estilo en la ejecución normal de un programa. Por ejemplo, el manejo de funciones puede implicar el uso de funciones con efectos laterales.

Un subprograma debe comprobar ciertos tipos de errores, tal como entradas no válidas o parámetros valor. ¿Qué acción debe hacer un subprograma cuando se encuentra un error? Un sistema puede, en el caso de un procedimiento, presentar un mensaje de error y devolver un indicador o bandera lógica a la rutina llamadora para indicarle que ha encontrado una línea de datos no válida; en este caso, el procedimiento deja la responsabilidad de realizar la acción apropiada a la rutina llamadora. En otras ocasiones, es más adecuado que el propio subprograma tome las acciones pertinentes —por ejemplo— cuando la acción requerida no depende del punto en que fue llamado el subprograma.

Si una función maneja errores imprimiendo un mensaje o devolviendo un indicador, viola las reglas contra efectos laterales dadas anteriormente.

Dependiendo del contexto, las acciones apropiadas pueden ir desde ignorar los datos erróneos hasta continuar la ejecución para terminar el programa. En el caso de un error fatal que invoque la terminación, una ejecución de `halt` puede ser el método más limpio para abortar. Otra situación delicada se puede presentar cuando se encuentra un error fatal en estructuras condicionales `if-then-else` o repetitivas `while`, `repeat`. La primera acción puede ser llamar a un procedimiento de diagnóstico que imprima la información necesaria para ayudarle a determinar la causa del error; pero después de que el procedimiento ha presentado toda esta información, se ha de terminar el programa. Sin embargo, si el procedimiento de diagnóstico devuelve el control al punto en el que fue llamado, debe salir de muchas capas de estructuras de control anidadas. *En este caso la solución más limpia es que la última sentencia del procedimiento de diagnóstico sea `halt`.*

1.8.9. Legibilidad

Para que un programa sea fácil de seguir su ejecución (la *traza*) debe tener una buena estructura y diseño, una buena elección de identificadores, buen sangrado y utilizar líneas en blanco en lugares adecuados y una buena documentación.

Como ya se ha comentado anteriormente se han de elegir identificadores que describan fielmente su propósito. Distinguir entre palabras reservadas, tales como `for` o `procedure`, identificadores estándar, tal como `real` o `integer` e identificadores definidos por el usuario. Algunas reglas que hemos seguido en el libro son:

- Las palabras reservadas se escriben en minúsculas negritas (en letra *courier*, en el libro).
- Los identificadores, funciones estándar y procedimientos estándar en minúsculas con la primera letra en mayúsculas (`WriteLn`).
- Los identificadores definidos por el usuario en letras mayúsculas y minúsculas. Cuando un identificador consta de dos o más palabras, cada palabra comienza con una letra mayúscula (`LeerVector`, `ListaNumeros`).

Otra circunstancia importante a considerar en la escritura de un programa es el sangrado o *indentación* de las diferentes líneas del mismo. Algunas reglas importantes a seguir para conseguir un buen estilo de escritura que facilite la legibilidad son:

- Los bloques deben ser sangrados suficientemente para que se vean con claridad (3 a 5 espacios en blanco puede ser una cifra aceptable).
- En una sentencia compuesta, las palabras `begin-end` deben estar alineadas:

```
begin
  <sentencial>
  <sentencia2>

  .
  .
  .
  <sentencian>
end
```

- *Sangrado consistente*. Sangrar siempre el mismo tipo de construcciones de igual manera. Algunas propuestas pueden ser:

bucles while/repeat/for

<pre>while <condición> <sentencia></pre>	<pre>while <condición> do begin <sentencia> end</pre>
	<pre>while <condición> do begin <sentencias> end</pre>

Sentencias if-then-else

<pre>if <condición> then <sentencial> else <sentencia2></pre>	<pre>if <condición> then <sentencial> else <sentencia2></pre>
<pre>if <condición> then <sentencias> else <sentencias></pre>	
<pre>if <condición> then begin <sentencias> end else begin <sentencias> end</pre>	<pre>if <cond1> then <acción1> else if <cond2> then <acción2> else <cond2> if ... </pre>

```

if <cond1> then                                inadecuada
  <acción1>
else if <cond2> then
  <acción2>
else if <cond3> then
  <acción3>

```

1.9. LA DOCUMENTACIÓN

Un programa (un paquete de software) de computadora necesita siempre de una documentación que permita a sus usuarios aprender a utilizarlo y mantenerlo. La documentación es una parte importante de cualquier paquete de software y, a su vez, su desarrollo es una pieza clave en la ingeniería de software.

Existen tres grupos de personas que necesitan conocer la documentación del programa: programadores, operadores y usuarios. Los requisitos necesarios para cada uno de ellos suelen ser diferentes, en función de las misiones de cada grupo:

programadores	<i>manual de mantenimiento del programa</i>
operadores	<i>manual del operador</i>
	operador: persona encargada de correr (ejecutar) el programa, introducir datos y extraer resultados
usuario	<i>manual del usuario</i>
	usuario: persona o sección de una organización que explota el programa, conociendo su función, las entradas requeridas, el proceso a ejecutar y la salida que produce

En entornos interactivos como el caso de Turbo Borland Pascal, las misiones del usuario y operador suelen ser las mismas. Así pues, la documentación del programa se puede concretar a:

- **manual del usuario,**
- **manual de mantenimiento.**

1.9.1. Manual del usuario

La documentación de un paquete (programa) de software suele producirse con dos propósitos: «uno, explicar las funciones del software y describir el modo de utilizarlas (*documentación del usuario*, que está diseñada para ser leída por el usuario del programa); dos, describir el software en sí para poder mantener el sistema en una etapa posterior de su ciclo de vida (*documentación del sistema o de mantenimiento*)»⁷.

⁷ Brookshear, Glen J.: *Introducción a las ciencias de la computación*, Addison-Wesley, 1995, pág. 272.

La documentación de usuario es un instrumento comercial importante. Una buena documentación de usuario hará al programa más accesible y asequible. Hoy día es una práctica habitual que muchos creadores de programas contratan escritores técnicos para elaborar esta parte del proceso de producción de un programa. Esta documentación adopta la forma de un manual que presenta una introducción a las funciones más utilizadas del software, una sección que explica cómo instalar el programa y una sección de referencia que describe los detalles de cada función del software. Es frecuente que el manual se edite en forma de libro, aunque cada vez es más frecuente incluirlo además, o en lugar, del libro en el propio programa y suele denominarse *manual de ayuda en línea*.

La documentación del sistema o manual de mantenimiento es por naturaleza más técnica que la del usuario. Antiguamente esta documentación consistía en los programas fuente finales y algunas explicaciones sobre la construcción de los mismos. Hoy día esto ya no es suficiente y es necesario estructurar y ampliar esta documentación.

La documentación del sistema abarca todo el ciclo de vida del desarrollo del software, incluidas las especificaciones originales del sistema y aquéllas con las que se verificó el sistema, los diagramas de flujo de datos (**DFD**), diagramas entidad-relación (**DER**), diccionario de datos y diagramas o cartas de estructura que representan la estructura modular del sistema.

El problema más grave que se plantea es la construcción práctica real de la documentación y su continua actualización. Durante el ciclo de vida del software cambian continuamente las especificaciones, los diagramas de flujo y de E/R (Entidad/Relación) o el diagrama de estructura; esto hace que la documentación inicial se quede obsoleta o incorrecta y por esta causa la documentación requiere una actualización continua de modo que la documentación final sea lo más exacta posible y se ajuste a la estructura final del programa.

El manual de usuario debe cubrir al menos los siguientes puntos:

- Órdenes necesarias para cargar el programa en memoria desde el almacenamiento secundario (disco) y arrancar su funcionamiento.
- Nombres de los archivos externos a los que accede el programa.
- Formato de todos los mensajes de error o informes.
- Opciones en el funcionamiento del programa.
- Descripción detallada de la función realizada por el programa.
- Descripción detallada, preferiblemente con ejemplos, de cualquier salida producida por el programa.

1.9.2. Manual de mantenimiento (documentación para programadores)

El manual de mantenimiento es la documentación requerida para mantener un programa durante su ciclo de vida. Se divide en dos categorías:

- documentación interna,
- documentación externa.

1.9.3. Documentación interna

Esta documentación cubre los aspectos del programa relativos a la sintaxis del lenguaje. Esta documentación está contenida en los *comentarios*, encerrados entre llaves {} o bien paréntesis/asteriscos (**). Algunos tópicos a considerar son:

- Cabecera de programa (nombre del programador, fecha de la versión actual, breve descripción del programa).
- Nombres significativos para describir identificadores.
- Comentarios relativos a la función del programa como en todo, así como los módulos que comprenden el programa.
- Claridad de estilo y formato [una sentencia por línea, *indentación* (sangrado)], líneas en blanco para separar módulos (procedimientos, funciones, unidades, etc.).
- Comentarios significativos.

Ejemplos

```
var
    Radio... {entrada, radio de un círculo}
    {Calcular Área}
    Área := Pi * radio * radio;
```

1.9.4. Documentación externa

Documentación ajena al programa fuente, que se suele incluir en un manual que acompaña al programa. La documentación externa debe incluir:

- Listado actual del programa fuente, mapas de memoria, referencias cruzadas, etc.
- Especificación del programa: documento que define el propósito y modo de funcionamiento del programa.
- Diagrama de estructura que representa la organización jerárquica de los módulos que comprende el programa.
- Explicaciones de fórmulas complejas.
- Especificación de los datos a procesar: archivos externos incluyendo el formato de las estructuras de los registros, campos, etc.
- Formatos de pantallas utilizados para interactuar con los usuarios.
- Cualquier indicación especial que pueda servir a los programadores que deben mantener el programa.

1.9.5. Documentación del programa

Un programa *bien documentado* es aquel que otras personas pueden leer, usar y modificar. Existen muchos estilos aceptables de documentación y, con frecuencia, los temas a incluir dependerán del programa específico. No obstante, señalamos a continuación algunas características esenciales comunes a cualquier documentación de un programa:

1. Un comentario de cabecera para el programa que incluye:
 - a) Descripción del programa: propósito.
 - b) Autor y fecha.
 - c) Descripción de la entrada y salida del programa.
 - d) Descripción de cómo utilizar el programa.
 - e) Hipótesis sobre tipos de datos esperados.
 - f) Breve descripción de los algoritmos globales y estructuras de datos.
 - g) Descripción de las variables importantes.
2. Comentarios breves en cada módulo similares a la cabecera del programa y que contenga información adecuada de ese módulo, incluyendo en su caso precondiciones y postcondiciones. Describir las entradas y cómo las salidas se relacionan con las entradas.
3. Escribir comentarios inteligentes en el cuerpo de cada módulo que expliquen partes importantes y confusas del programa.
4. Describir claramente y con precisión los modelos de datos fundamentales y las estructuras de datos seleccionadas para representarlas así como las operaciones realizadas para cada procedimiento.

Aunque existe la tendencia entre los programadores y sobre todo entre los principiantes a documentar los programas como última etapa, esto no es buena práctica, lo idóneo es documentar el programa a medida que se desarrolla. La tarea de escribir un programa grande se puede extender por períodos de semanas o incluso meses. Esto le ha de llevar a la consideración de que lo que resulta evidente ahora puede no serlo de aquí a dos meses; por esta causa, documentar a medida que se progresá en el programa es una regla de oro para una programación eficaz.

Regla

Asegúrese de que siempre se corresponden los comentarios y el código. Si se hace un cambio importante en el código, asegúrese de que se realiza un cambio similar en el comentario.

1.10. DEPURACIÓN

Una de las primeras cosas que se descubren al escribir programas es que un programa raramente funciona correctamente la primera vez. La ley de Murphy «si algo puede ser incorrecto, lo será» parece estar escrita pensando en la programación de computadoras.

Aunque un programa funcione sin mensajes de error y produzca resultados, puede ser incorrecto. Un programa es correcto sólo *si se producen resultados correctos para todas las entradas válidas posibles*. El proceso de eliminar errores —*bugs*— se denomina depuración (*debugging*) de un programa.

Cuando el compilador detecta un error, la computadora visualiza *un mensaje de error*, que indica se ha producido un error y cuál puede ser la causa posible del error. Desgraciadamente, los mensajes de error son, con frecuencia, difíciles de interpretar y son, a veces, engañosos. Los errores de programación se pueden dividir en tres clases: **errores de compilación** (sintaxis), **errores en tiempo de ejecución** y **errores lógicos** (véase apartado 1.12).

1.10.1. Localización y reparación de errores

Aunque se sigan todas las técnicas de diseño dadas a lo largo del libro y en este capítulo, en particular, y cualquier otra que haya obtenido por cualquier otro medio (otros libros, experiencias, cursos, etc.), es prácticamente imposible e inevitable que su programa carezca de errores. Afortunadamente los programas modulares, claros y bien documentados son ciertamente más fáciles de depurar que aquellos que no lo son. Es recomendable utilizar técnicas de seguridad contra fallos, que protejan contra ciertos errores e informen de ellos cuando se encuentran.

Con frecuencia el programador, pero sobre todo el estudiante de programación, está convencido de la bondad de sus líneas de programa, sin pensar en las múltiples opciones que pueden producir los errores: el estado incorrecto de una variable lógica, la entrada de una cláusula *then* o *else*, la salida imprevista de un bucle por un mal diseño de su contador, etc. El enfoque adecuado debe ser seguir la traza de la ejecución del programa utilizando las facilidades de depuración de Turbo Borland Pascal o añadir sentencias *Write* que muestren cuál fue la cláusula ejecutada. En el caso de condiciones lógicas, si la condición es falsa cuando se espera que es verdadera —como el mensaje de error puede indicar—, entonces el siguiente paso es determinar cómo se ha convertido en falsa.

¿Cómo se puede encontrar el punto de un programa en que algo se ha convertido en una cosa distinta a lo que se había previsto? En Turbo Borland Pascal se puede hacer el seguimiento de la ejecución de un programa o bien paso a paso a través de las sentencias del programa o bien estableciendo puntos de ruptura (*breakpoint*). Se puede examinar también el contenido de una variable específica, bien estableciendo inspecciones/observaciones (*watches*) o bien insertando sentencias *Write* temporales (véase el Apéndice). La clave para una buena depuración es sencillamente utilizar estas herramientas que indiquen lo que está haciendo el programa.

La idea principal es localizar sistemáticamente los puntos del programa que causan el problema. La lógica de un programa implica que ciertas condiciones sean verdaderas en puntos diferentes del programa (recuerde que estas condiciones se llaman *invariantes*). Un error (*bug*) significa que una condición que pensaba iba a ser verdadera no lo es. Para corregir el error, se debe encontrar la primera posición del programa en la que una de estas condiciones difiera de sus expectativas. La inserción apropiada de puntos de ruptura, y de observación o inspección o sentencias *Write* en posiciones estratégicas de un programa —tal como entradas y salidas de bucles, estructuras selectivas y subprogramas— sirven para aislar sistemáticamente el error.

Las herramientas de diagnóstico han de informarles si las cosas son correctas o equivocadas antes o después de un punto dado del programa. Por consiguiente, después de

ejecutar el programa con un conjunto inicial de diagnósticos se ha de poder seguir el error entre dos puntos. Por ejemplo, si el programa ha funcionado bien hasta la llamada al procedimiento o función P1, pero algo falla cuando se llama al procedimiento P2, nos permite centrar el problema entre estos dos puntos, la llamada a P2 y el punto concreto donde se ha producido el error en P2. Este método es muy parecido al de *aproximaciones sucesivas*, es decir, ir acotando la causa posible de error hasta limitarla a unas pocas sentencias.

Naturalmente, la habilidad para situar los puntos de ruptura, de observación o sentencias Write, dependerá del dominio que se tenga del programa y de la experiencia del programador. No obstante, le damos a continuación algunas reglas prácticas que le faciliten su tarea de depuración.

1.10.1.1. Uso de sentencias Write

Las sentencias Write pueden ser muy adecuadas en numerosas ocasiones. Tales sentencias sirven para informar sobre valores de variables importantes y la posición en el programa en que las variables toman esos valores. Es conveniente utilizar un comentario para etiquetar la posición.

```
{Posición una}
WriteLn ('Está situado en posición una del procedimiento Test');
WriteLn ('A=', a, 'B = ', b, 'C = ', c);
```

1.10.1.2. Depuración de sentencias if-then-else

Situar una parte de ruptura antes de una sentencia if-then-else y examinar los valores de las expresiones lógicas y de sus variables. Se pueden utilizar o bien puntos de ruptura o sentencias Write para determinar qué alternativa de la sentencia if se toma:

```
(Examinar valores de <condición> y variables antes de if)
if <condición>
  then
    begin
      WriteLn ('Condición verdadera: siga camino');
      ...
    end
  else
    begin
      WriteLn ('Condición falsa: siga camino');
      ...
    end;
```

1.10.1.3. Depuración de bucles

Situar los puntos de ruptura al principio y al final del bucle y examinar los valores de las variables importantes:

```

{Examinar valores de m y n antes de entrar al bucle}
for i := m to n do
begin
{Examinar los valores de i y variables importantes final bucle}
end;
{Examinar los valores de m y n después de salir del bucle}

```

1.10.1.4. Depuración de subprogramas

Las dos posiciones clave para situar los puntos de ruptura son al principio y al final de un subprograma. Se deben examinar los valores de los parámetros en estas dos posiciones utilizando o bien sentencias Write o ventanas de inspección u observación (*watches*).

1.10.1.5. Lecturas de estructuras de datos completos

Las variables cuyos valores son arrays u otras estructuras puede ser interesante examinarlas. Para ello se recurre a escribir rutinas específicas de volcado (presentación en pantalla o papel) que ejecuten la tarea. Una vez diseñada la rutina se llama a ella desde puntos diferentes según interesa a la secuencia de flujo de control del programa y los datos que sean necesarios en cada caso.

1.10.2. Los equipos de programación

En la actualidad es difícil y raro que un gran proyecto de software sea *implementado* (realizado) por un solo programador. Normalmente, un proyecto grande se asigna a un equipo de programadores, que por anticipado deben coordinar toda la organización global del proyecto.

Cada miembro del equipo es responsable de un conjunto de procedimientos, algunos de los cuales pueden ser utilizados por otros miembros del equipo. Cada uno de estos miembros deberá proporcionar a los otros las especificaciones de cada procedimiento, condiciones *pretest* o *posttest* y su lista de parámetros formales; es decir, la información que un potencial usuario del procedimiento necesita conocer para poder ser llamado.

Normalmente, un miembro del equipo actúa como bibliotecario, de modo que a medida que un nuevo procedimiento se termina y comprueba, su versión actualizada sustituye la versión actualmente existente en la biblioteca. Una de las tareas del bibliotecario es controlar la fecha en que cada nueva versión de un procedimiento se ha incorporado a la librería, así como asegurarse de que todos los programadores utilizan la versión última de cualquier procedimiento.

Es misión del equipo de programadores crear bibliotecas de procedimientos, que posteriormente puedan ser utilizadas en otras aplicaciones. Una condición importante deben cumplir los procedimientos *estar comprobados y ahorro de tiempo/memoria*.

1.11. DISEÑO DE ALGORITMOS

Tras la fase de análisis, para poder solucionar problemas sobre una computadora, debe conocerse cómo diseñar algoritmos. En la práctica sería deseable disponer de un método para escribir algoritmos, pero, en la realidad, no existe ningún algoritmo que sirva para realizar dicha escritura. El diseño de algoritmos es un proceso creativo. Sin embargo, existen una serie de pautas o líneas a seguir que ayudarán al diseño del algoritmo (Tabla 1.1).

Tabla 1.1. Pautas a seguir en el diseño de algoritmos

1. Formular una solución precisa del problema que debe solucionar el algoritmo.
2. Ver si existe ya algún algoritmo para resolver el problema o bien se puede adaptar uno ya existente (*algoritmos conocidos*).
3. Buscar si existen técnicas estándar que se puedan utilizar para resolver el problema.
4. Elegir una estructura de datos adecuada.
5. Dividir el problema en subproblemas y aplicar el método a cada uno de los subproblemas (*diseño descendente*).
6. Si todo lo anterior falla, comience de nuevo en el paso 1.

De cualquier forma, antes de iniciar el diseño del algoritmo es preciso asegurarse que el programa está bien definido:

- Especificaciones precisas y completas de las entradas necesarias.
- Especificaciones precisas y completas de la salida.
- ¿Cómo debe reaccionar el programa ante datos incorrectos?
- ¿Se emiten mensajes de error? ¿Se detiene el proceso?, etc.
- Conocer cuándo y cómo debe terminar un programa.

1.12. PRUEBAS (*TESTING*)

Aunque muchos programadores utilizan indistintamente los términos *prueba* o *comprobación* (*testing*) y *depuración*, son, sin embargo, diferentes. La *comprobación* (pruebas) se refiere a las acciones que determinan si un programa funciona correctamente. La *depuración* es la actividad posterior de encontrar y eliminar los errores (*bugs*) de un programa. Las pruebas de ejecución de programas —normalmente— muestran claramente que el programa contiene errores, aunque el proceso de depuración puede, en ocasiones, resultar difícil de seguir y comprender.

Edgar Dijkstra ha escrito que mientras las pruebas muestran efectivamente la presencia de errores, nunca pueden mostrar su *ausencia*. Una prueba (test) con éxito significa solamente que ningún error se descubrió en las circunstancias particulares probadas, pero no dice nada sobre otras circunstancias. En teoría, el único medio de comprobar

que un programa es correcto es probar *todos* los casos posibles (realizar una *prueba exhaustiva*), situación técnicamente imposible, incluso para los programas más simples. Consideremos un caso sencillo: calcular la media aritmética de las temperaturas de un mes dado; una prueba exhaustiva requerirá todas las posibles combinaciones de temperaturas y días de un mes: tarea ardua, laboriosa y lenta.

No obstante, el análisis anterior no significa que la comprobación sea imposible; al contrario, existen diferentes metodologías formales para las comprobaciones de programas. Una filosofía adecuada para pruebas de programas incluye las siguientes consideraciones:

1. Suponer que su programa tiene errores hasta que sus pruebas muestren lo contrario.
2. Ningún test simple de ejecución puede probar que un programa está libre de error.
3. Trate de someter al programa a pruebas duras. Un programa bien diseñado manipula entradas «con elegancia». Por este término se entiende que el programa no produce errores en tiempo de ejecución ni produce resultados incorrectos; por el contrario, el programa, en la mayoría de los casos, visualizará un mensaje de error claro y solicita de nuevo los datos de entrada.
4. Comenzar la comprobación antes de terminar la codificación.
5. Cambiar sólo una cosa cada vez.

La prueba de un programa ocurre cuando se ejecuta un programa y se observa su comportamiento.

Cada vez que se ejecuta un programa con algunas entradas, se prueba a ver cómo funciona el trabajo para esa entrada particular. Cada prueba ayuda a establecer que el programa cumpla las especificaciones dadas.

Selección de datos de prueba

Cada prueba debe ayudar a establecer que el programa cumple las especificaciones dadas. Parte de la ciencia de *ingeniería de software* es la construcción sistemática de un conjunto de entradas de prueba que es idóneo a descubrir errores.

Para que un conjunto de datos puedan ser considerados como buenos datos de prueba, sus entradas de prueba necesitan cumplir dos propiedades.

Propiedades de buenos datos de prueba

1. Se debe conocer qué salida debe producir un programa correcto para cada entrada de prueba.
2. Las entradas de prueba deben incluir aquellas entradas que probablemente originen más errores.

Se deben buscar numerosos métodos para encontrar datos de prueba que produzcan probablemente errores. El primer método se basa en identificar y probar entradas denominadas *valores externos*, que son especialmente idóneos para causar errores. Un **valor externo** o límite de un problema en una entrada produce un tipo diferente de comportamiento. Por ejemplo, suponiendo que se tiene una función `ver_hora` que tiene un parámetro `hora` y una precondición:

Precondición: Hora está comprendido en el rango 0-23

Los dos valores límites de `ver_hora` son hora igual a 0 (dado que un valor menor de 0 es ilegal) y hora igual a 23 (dado que un valor superior a 23-24... es ilegal). Puede ocurrir que la función se comporte de modo diferente para horario matutino (0 a 11) o nocturno (12 a 23), entonces 11 y 12 serán valores extremos. Si se espera un comportamiento diferente para hora igual a 0, entonces 1 es un valor extremo. En general no existe una definición precisa de valor extremo, pero debe ser aquel que muestre un comportamiento límite en el sistema.

Valores de prueba extremos

Si no se pueden probar todas las entradas posibles, probar al menos los valores extremos. Por ejemplo, si el rango de entradas legales va de cero a un millón, asegúrese probar la entrada 0 y la entrada 1.000.000. Es buena idea considerar también 0, 1 y -1 como valores límites siempre que sean entradas legales.

Otra técnica de prueba de datos es la denominada **perfilador** que básicamente considera dos reglas:

1. Asegúrese de que cada línea de su código se ejecuta al menos una vez para algunos de sus datos de prueba. Por ejemplo, puede ser una porción de su código que maneje alguna situación rara.
2. Si existe alguna parte de su código que a veces se salte totalmente, asegúrese, en ese caso, que existe al menos una entrada de prueba que salte realmente esta parte de su código. Por ejemplo, un bucle en el que el cuerpo se ejecute, a veces, cero veces. Asegúrese de que hay una entrada de prueba que produce que el cuerpo del bucle se ejecute cero veces.

1.12.1. Errores de sintaxis (de compilación)

Un *error de sintaxis* o *en tiempo de compilación* se produce cuando existen errores en la sintaxis del programa, tales como signos de puntuación incorrectos, palabras mal escritas, ausencia de separadores (signos de puntuación), o de palabras reservadas (ausencia

de un *end*). Si una sentencia tiene un error de sintaxis, no puede ser traducida y su programa no se ejecutará.

Cuando se detecta un error, Turbo Borland Pascal carga automáticamente el archivo fuente, sitúa el cursor en el error y visualiza un mensaje de error.

44 Field identifier expected

Normalmente, los mensajes de error son fáciles de encontrar. El siguiente ejemplo presenta dos errores de sintaxis: el punto y coma que falta al final de la primera linea y la palabra *WritaLn* mal escrita, debería ser *WriteLn*.

```
Suma := 0
for I := 0 to 10 do
  Suma := Suma + A[I];
WritaLn (Suma/10);
```

1.12.2. Errores en tiempo de ejecución

Los errores en tiempo de ejecución —o simplemente de ejecución— (*runtime error*) suceden cuando el programa trata de hacer algo imposible o ilógico. Los errores de ejecución sólo se detectan en la ejecución. Errores típicos son: la división por cero, intentar utilizar un subíndice fuera de los límites definidos en un array, etc.

x := 1/N produce un error si *N = 0*

Los mensajes de error típicos son del tipo:

Run-Time error nnn at xxxx:yyyy

nnn	<i>número de error en ejecución</i>
xxxx:yyyy	<i>dirección del error en ejecución (segmento y desplazamiento)</i>

Los errores de ejecución se dividen en cuatro categorías:

- errores DOS, 1-99 (*números de mensaje*)
- errores I/O, 100-149
- errores críticos, 150-199
- errores fatales, 200-255

1.12.3. Errores lógicos

Los errores lógicos son errores del algoritmo o de la lógica del programa. Son difíciles de encontrar porque el compilador no produce ningún mensaje de error. Se producen cuando el programa es perfectamente válido y produce una respuesta.

Calcular la media de todos los números leídos del teclado

```

Suma := 0;
for i := 0 to 10 do
begin
  ReadLn (Num);
  Suma := Suma + Num
end;
Media := Suma / 10;

```

La media está calculada mal ya que existen once números (0 a 10) y no diez como se ha escrito.

Si se desea escribir la sentencia:

```
Salario := Horas * Tasa;
```

y se escribe:

```
Salario := Horas + Tasa;
```

Es un error lógico (+ por *) ya que *a priori* el programa funciona bien, y sería difícil, por otra parte, a no ser que el resultado fuese obvio, detectar el error.

1.12.4. El depurador

Turbo Borland Pascal tiene un *programa depurador* disponible para ayudarle a depurar un programa; el programa depurador le permite ejecutar su programa, una sentencia cada vez, de modo que se pueda ver el efecto de la misma. El depurador imprime un diagnóstico cuando ocurre un error de ejecución, indica la sentencia que produce el error y permite visualizar los valores de variables seleccionadas en el momento del error. Asimismo, se puede seguir la pista de los valores de variables seleccionadas durante la ejecución del programa (*traza*), de modo que se pueda observar cómo cambian estas variables mientras el programa se ejecuta. Por último se puede pedir al depurador que detenga la ejecución en determinados puntos (*breakpoints*); en esos momentos se pueden inspeccionar los valores de las variables seleccionadas a fin de determinar si son correctas.

El depurador tiene la gran ventaja de posibilitar la observación de los diferentes valores que van tomando las variables dentro del programa.

1.13. EFICIENCIA

La *eficiencia* de un programa es una medida de cantidad de recursos consumidos por el programa. Tradicionalmente, los recursos considerados han sido el tiempo de ejecución y/o el almacenamiento (ocupación del programa en memoria). Mientras menos tiempo se utilice y menor almacenamiento, el programa será más eficiente.

El tiempo y almacenamiento (memoria) de la computadora suelen ser costosos y por ello su ahorro siempre será importante. En algunos casos la eficiencia es críticamente

importante: control de una unidad de vigilancia intensiva de un hospital —un retardo de fracciones de segundo puede ser vital en la vida de un enfermo—, un programa de control de roturas en una prensa hidráulica —la no detección a tiempo podría producir grandes inundaciones—, etc. Por el contrario, existirán otros casos en los que el tiempo no será factor importante: control de reservas de pasajeros en una agencia de viajes.

La mejora del tiempo de ejecución y el ahorro en memoria se suelen conseguir con la mejora de los algoritmos y sus programas respectivos. En ocasiones, un simple cambio en un programa puede aumentar la velocidad de ejecución considerablemente. Como muestra de ello analicemos el problema siguiente desde el punto de vista de tiempo de ejecución

Buscar en un array o lista de enteros una clave dada (un entero)

```
ArrayLista : Lista[Primero..Ultimo] of integer

ALGORITMO Buscar elemento t
  J := Primero;
  while (T <> Lista [J] and (J < Ultimo) do
    J := J + 1;
  if T = Lista [J] then
    WriteLn ('el elemento', T, 'está en la lista')
  else
    WriteLn ('el elemento', T, 'no está en la lista')
```

El bucle va comprobando cada elemento de la lista hasta que encuentra el valor de T o bien se alcanza el final de la lista sin encontrar T.

Supongamos ahora que la lista de enteros está ordenada.

45 73 81 120 160 321 450

En este caso el bucle puede ser más eficiente si en lugar de la condición

(T <> Lista[J]) and (J < Ultimo)

se utiliza

(T > Lista[J]) and (J < Ultimo)

Ello se debe a que si T es igual a Lista [J], se ha encontrado el elemento, y si T es menor que Lista [J], entonces sabemos que T será más pequeño que todos los elementos que le siguen. Tan pronto como se pruebe un valor de T y resulte menor que su correspondiente Lista [J], esta condición será falsa y el bucle se terminará. De este modo, y como término medio, se puede ahorrar alrededor de la mitad del número de iteraciones.

En el caso de que T no existe en la lista, el número de iteraciones de ambos algoritmos es igual, mientras que si T no existe en la lista, el algoritmo 2 con (T > Lista [J]) reducirá el número de iteraciones en la mitad y por consiguiente será más eficiente.

El listado y ejecución del programa con los dos procedimientos muestran cómo con un simple cambio (el operador $<$ por el operador $>$) se gana notablemente en eficiencia ya que reduce el tiempo de ejecución.

```
program Eficiencia;
{comparar dos algoritmos de búsqueda}
const
  Primero = 1;
  Ultimo = 10;
type
  Indice = Primero..Ultimo;
  Items = array [Indice] of integer;
var
  Lista : Items;
  J, T : integer;

procedure Busqueda1 (L : Items, T : Integer);
var
  I : Indice;
begin
  I := Primero;
  while (T<> L[I]) and (I < Ultimo) do
    I := I+1;
  if T = L[I] then
    WriteLn ('el elemento ',T,'está en la lista')
  else
    WriteLn ('el elemento ',T,'no está en la lista');
  WriteLn ('Búsqueda terminada');
  WriteLn (I,'iteraciones')
end;

procedure Busqueda2 (L : Items, T : Integer);
var
  I : Indice;
begin
  I:= 1;
  while (T > L[I]) and (I < Ultimo) do
    I := I + 1;
  if T = L[I] then
    WriteLn ('el elemento ', T,'está en la lista')
  else
    WriteLn ('el elemento ',T, 'no está en la lista');
  WriteLn ('Búsqueda terminada');
  WriteLn (I, 'iteraciones')
end;

begin {programa principal}
  WriteLn ('Introduzca 10 enteros en orden ascendente:');
  for J := Primero to Ultimo do
    Read(Lista[J]);
  ReadLn;
  WriteLn('Introducir número a buscar:');
  ReadLn(T);
  Busqueda1(Lista,T);
  Busqueda2(Lista,T)
end.
```

Ejecución

```
Introduzca 10 enteros en orden ascendente:
2   5   8   12   23   37   45   89   112   234
Introducir número a buscar:
27
el elemento 27 no está en la lista
Busqueda1 terminada en
10 iteraciones
el elemento 27 no está en la lista
Busqueda2 terminada en
6 iteraciones
```

1.13.1. Eficiencia *versus* legibilidad (claridad)

Las grandes velocidades de los microprocesadores (unidades centrales de proceso) actuales, junto con el aumento considerable de las memorias centrales (cifras típicas usuales superan siempre los 32/64 MB), hacen que los recursos típicos tiempo y almacenamiento no sean hoy día parámetros fundamentales para la medida de la eficiencia de un programa.

Por otra parte es preciso tener en cuenta que —a veces— los cambios para mejorar un programa puede hacerlo más difícil de comprender: poco legibles o claros. En programas grandes la legibilidad suele ser más importante que el ahorro en tiempo y en almacenamiento en memoria. Como norma general, cuando la elección en un programa se debe hacer entre claridad y eficiencia, generalmente se elegirá la claridad o la legibilidad del programa.

1.14. TRANSPORTABILIDAD (*PORTABILIDAD*)

Un programa es *transportable* o *portable* si se puede trasladar a otra computadora sin cambios o con pocos cambios apreciables. La forma de hacer un programa transportable es elegir como lenguaje de programación la versión estándar del mismo, en el caso de Pascal: **ANSI/IEEE** estándar e **ISO** estándar.

Turbo Borland Pascal no sigue todas las normas del estándar y además reúne un conjunto de características no disponibles en Pascal estándar. Esto significa que si desea transportar sus programas a Pascal estándar, deberá evitar en los mismos todas las características propias de Turbo.

RESUMEN

La ingeniería de software trata sobre la creación y producción de programas a gran escala. El ciclo de vida del software consta de las fases:

- Análisis de requisitos y especificación.
- Diseño.
- *Implementación* (codificación y depuración).

- Prueba.
- Evolución.
- Mantenimiento.

Si se desea crear buenos programas en Turbo Borland Pascal, los conceptos de unidades y programación modular serán vitales en el desarrollo de los mismos.

EJERCICIOS

- 1.1. ¿Cuál es el invariante del bucle siguiente?

```
Indice:= 1;
Suma := A[1];
while Indice < N hacer
begin
  Indice := Succ (Indice);
  Suma := Suma + A[Indice]
end;
```

- 1.2. Considere el programa siguiente que interactivamente lee y escribe el número de identificación, nombre, edad y salario (en millares de pesetas) de un grupo de empleados. ¿Cómo se puede mejorar el programa?

```
program Test;
var x1, x2, x3, i : integer;
    Nombre : array[1..10] of char;
begin
  while not eof do
  begin
    Read(x1);
    for i := 1 to 8 do
      Read (Nombre[i]);
    ReadLn (x2,x3);
    WriteLn (x1,Nombre,x2,x3)
  end
end;
```

- 1.3. ¿Cuál es el error de la siguiente función? ¿Cómo puede resolverse?

```
function Tangente (x : real):real;
begin
  Tangente := sin(x)/cos(x)
end;
```

- 1.4. Escribir una función que devuelve el total de dígitos (n) distintos de cero en un número entero arbitrario (Número). La solución debe incluir un bucle while con el siguiente invariante de bucle válido.

```
{invariante:
  0 <= Total <= n y Número se ha dividido por 10 veces Total}
y ésta será la aserción válida (postcondición)
```

```
{postcondición : Total es n}
```

1.5. Escribir un invariante de bucle para el siguiente segmento de código

```

Producto := 1;
Contador := 2;
while Contador < 10 do
begin
    Producto := Producto * Contador;
    Contador := Contador + 1
end;

```

PROBLEMAS

- 1.1.** Escribir un programa que lea una lista de 100 o menos enteros y a continuación realice las siguientes tareas: visualizar la lista ordenada desde el mayor hasta el menor; visualizar lista en orden inverso, calcular la media; calcular la mediana; listar las listas en orden creciente y en orden decreciente, mostrando la diferencia de cada valor con la media y la mediana. Es conveniente que cada tarea se realice con un módulo y luego se integren todos en un programa.
- 1.2.** A la entrada de un aparcamiento, un automovilista retira un ticket en el cual está indicado su hora de llegada. Antes de abandonar el aparcamiento se introduce el ticket en una máquina que le indica la suma a pagar. El pago se realiza en una máquina automática que devuelve cambio.

Escribir un programa que simule el trabajo de la máquina. Se supone que:

- La duración del estacionamiento es siempre inferior a 24 horas.
 - La máquina no acepta más que monedas de 200, 100, 50, 25, 5, 2 y 1 peseta.
 - Las tarifas de estacionamiento están definidas por tramos semihorarios (1/2 hora).
- 1.3.** Escribir un programa que contenga una serie de opciones para manipular un par de matrices (suma, resta y multiplicación). Cada procedimiento debe validar sus parámetros de entrada antes de ejecutar la manipulación requerida de datos.
 - 1.4.** Un viajante de comercio debe visitar N ciudades. Parte de una de ellas y debe volver a su punto de partida. Conoce la distancia entre cada una de las ciudades y desea hacer un viaje lo más corto posible. El sistema elegido de viaje es el siguiente:

- A partir de cada ciudad, elige como ciudad siguiente la más próxima entre las ciudades que no ha visitado.

Se suponen datos: la lista de ciudades, las distancias (en kilómetros) entre cada una de ellas y el nombre de la ciudad de partida. Elegir un algoritmo que calcule el recorrido más corto.

- 1.5.** Se tiene la lista de una clase que contiene un nombre de estudiante y las notas de cinco exámenes. Se desea escribir un programa que visualice la media de cada alumno, la nota media más alta y la nota media más baja, junto con los nombres correspondientes.
- 1.6.** Se desea diseñar un programa que permita adiestrar a un niño en cálculos mentales. Para ello el niño debe elegir entre las cuatro operaciones aritméticas básicas; la computadora le presentará la operación correspondiente entre dos números, y el niño debe introducir desde el teclado el resultado. El niño dispone de tres tentativas. Caso de acertar, la computadora debe visualizar «Enhorabuena», y en caso de fallo «Lo siento, inténtalo otra vez».

- 1.7. Escribir un programa que construya un directorio telefónico interactivo que contiene una colección de nombres y números de teléfonos. Las características del directorio han de ser: 1) poder insertar una nueva entrada en el directorio; 2) recuperar una entrada del directorio; 3) cambiar una entrada del directorio, y 4) borrar una entrada del directorio. *Nota:* El directorio debe estar ordenado alfabéticamente.

Construcción de grandes programas: módulos versus unidades

CONTENIDO

- 2.1. Concepto de unidad.
 - 2.2. Estructura de una unidad.
 - 2.3. Creación de unidades.
 - 2.4. Utilización de unidad estándar.
 - 2.5. Situación de las unidades en sus discos. ¿Dónde busca Turbo Borland Pascal las unidades?
 - 2.6. Identificadores idénticos en diferentes unidades.
 - 2.7. Síntesis de unidades.
 - 2.8. Otros métodos de estructurar programas: inclusión, solapamiento y encadenamiento.
- RESUMEN.**
EJERCICIOS.
REFERENCIAS BIBLIOGRÁFICAS.

Una de las características más interesantes de Turbo Borland Pascal 5/6/7 es la posibilidad de descomponer un programa grande en productos más pequeños que se pueden compilar independientemente. Estos módulos se denominan *unidades* y eliminan el inconveniente de las versiones anteriores a 4.0 de la limitación de la memoria ocupada por programas ejecutables a 64 K. Los lectores que hayan utilizado Modula-2 encontrarán familiar el concepto unidad, ya que *unidad* (en Turbo Borland Pascal) y *módulo* (en Modula-2) representan el concepto de compilación separada y la posibilidad de construir grandes programas enlazando unidades compiladas separadamente para construir un programa ejecutable. Como resultado de ello, cuando se realiza un cambio en el código fuente, sólo se tiene que compilar el segmento al que afecta la modificación.

2.1. CONCEPTO DE UNIDAD

Una **unidad** es un conjunto de constantes, tipos de datos variables, procedimientos y funciones. Cada unidad es como un programa independiente Pascal o bien una biblioteca de declaraciones que se pueden poner en un programa y que permiten que éste se pueda dividir y compilar independientemente. Una unidad puede utilizar otras unidades y contiene una parte que puede contener instrucciones de iniciación.

Una unidad contiene uno o más procedimientos, funciones constantes definidas (y a veces otros elementos). Se puede compilar, probar y depurar una unidad independientemente de un programa principal. Una vez que una unidad ha sido compilada y depurada, no necesita compilarse más veces, Turbo Borland Pascal se encarga de enlazar la unidad al programa que utiliza esa unidad, empleando siempre en esta tarea menor tiempo que en la propia compilación. Los procedimientos, funciones y constantes que se definen en una unidad pueden ser utilizados por cualquier futuro programa que escriba sin tener que ser declarados en el programa.

Las unidades tienen una estructura similar a los programas y en consecuencia requieren un formato estricto, y es preciso declararlas por el usuario como si se tratara de un programa o subprograma.

Turbo Borland Pascal proporciona siete unidades estándar para el uso del programador: *System*, *Graph*, *DOS*, *Crt*, *Printer*, *Turbo3* y *Graph3*. Las cinco primeras sirven para escribir sus programas, y las dos últimas para mantener compatibilidad con programas y archivos de datos creados con la versión 3.0 de Turbo Borland Pascal. Las siete unidades están almacenadas en el archivo TURBO.TPL (biblioteca de programas residente propia del programa Turbo Borland Pascal). La versión 7.0 introdujo dos nuevas unidades: *WinDOS* y *Strings*.

2.2. ESTRUCTURA DE UNA UNIDAD

Una unidad está constituida de cuatro partes: **cabecera de la unidad**, sección de **interface** (**interfaz**), sección **implementation** (**implementación**) y sección **initialization** (**inicialización**)¹.

Formato

```
unit <identificador>;
interface
  uses <lista de unidades>;           {opcional}
    {declaraciones públicas objetos exportados}
implementation
  {declaraciones privadas}
  {definición de procedimientos y funciones públicas}
begin
  {código de inicialización}      {opcional}
end.
```

¹ Aceptamos el término *inicialización* por su extensa difusión en la jerga informática, aun a sabiendas que dicho término no es aceptado por el DRAE y su expresión correcta sería *iniciación*.

2.2.1. Cabecera de la unidad

La *cabecera de la unidad* comienza con la palabra reservada **unit**, seguida por el nombre de la unidad (identificador válido)². Es similar a una cabecera de programa, donde la palabra reservada **unit** reemplaza a la palabra reservada **program**. Por ejemplo, para crear una unidad denominada **MiUnidad** se ha de utilizar la cabecera de la unidad: **unit MiUnidad**.

El nombre de la unidad es arbitrario pero debe coincidir con el nombre del archivo que contiene. Por ejemplo, si la unidad se denomina **Test**.

Unit Test;

el archivo que contiene el programa fuente de la unidad se debe llamar **Test.PAS**. Cuando Turbo Borland Pascal compila la unidad, le asigna la extensión **TPU** (**Turbo Borland Pascal Unit**). Si el nombre de la unidad es diferente del nombre del archivo, el programa principal no podrá encontrar el archivo **TPU**.

Una unidad puede utilizar otras unidades siempre que se incluyan en la cláusula **uses** que aparece inmediatamente después de la palabra **interface**, y separadas por comas.

2.2.2. Sección de interfaz

La sección de *interface* (interfaz) es la parte de la unidad que sirve para conectar dicha unidad con otras unidades y programas. Esta sección se conoce como «la parte pública» de la unidad, ya que todos los objetos que figuren en esta sección son *visibles* desde el exterior o *exportables*. Las restantes unidades y programas tienen acceso a la información contenida en la sección de *interface*.

En la interfaz de la unidad se pueden declarar constantes, tipos de datos, variables y procedimientos. Los procedimientos y funciones visibles a cualquier programa que utilice la unidad se declaran aquí, pero sus cuerpos reales —implementaciones— se encuentran en la sección de implementación. La sección de interfaz indica a un programador cuáles procedimientos y funciones se pueden utilizar en un programa. La sección de implementación indica al compilador cómo implementarlos.

Los identificadores declarados en la sección de interfaz pueden ser referenciados por un cliente de la unidad, de modo que se consideran *visibles*.

Ejemplo de declaración

```
unit Rayo;
interface
  uses DOS, Graph, Crt; {se utilizan las unidades DOS, Graph y Crt}
var
  a, b, c : integer;
function Exponencial (a, b : integer) : real;
procedure Dividir (x, y : integer; var cociente : integer);
```

² El identificador puede tener de uno a ocho caracteres más un punto y una extensión tal como .PAS.

En la sección de *interface* se pueden declarar variables y elementos globales y otras unidades.

Observaciones

1. La unidad Rayo puede utilizar cualquier procedimiento o función presente en las unidades estándar Dos, Graph o Crt.
2. Se declaran tres variables globales; a, b, c (están disponibles a otras unidades y programas que utilicen la unidad Rayo). Se debe extremar el cuidado a la hora de declarar variables globales en la sección de interfaz de la unidad.

2.2.3. Sección de implementación

La tercera parte de una unidad, denominada **implementation** (*implementación*), es estrictamente «privada»; su contenido no es exportable. Sólo los procedimientos o funciones que aparecen en la sección **interface** pueden ser invocados desde el exterior de la unidad.

Esta sección contiene el cuerpo de los procedimientos y funciones declarados en la sección de *interface*.

```
unit Rayo;

interface
  function Exponencial (A, B : integer) : real;
  procedure Dividir (X, Y:integer; var Cociente : integer);

implementation
  function Exponencial (A, B : integer) : real;
  var
    P, I : integer;
  begin
    P := 1;
    for I := 1 to B do
      P := P * A;
    Exponencial := P
  end;

  procedure Dividir (X, Y : integer; var Cociente : integer);
  begin
    Cociente := X div Y
  end;
.
end.
```

Nótese que la declaración de una unidad está terminada por la palabra reservada **end** y un punto.

Las variables globales pueden ser declaradas dentro de una sección de *implementación*, pero estas variables serán globales a la unidad solamente y no accesibles a cualquier otra unidad o programa.

2.2.4. Sección de iniciación (*inicialización*)

La cuarta parte de una unidad, llamada *iniciación*, puede contener instrucciones pero puede igualmente estar vacía. Estas instrucciones sirven, por ejemplo, para inicializar variables o abrir archivos. La ejecución de estas instrucciones se efectúa en el momento del lanzamiento o ejecución de un programa que utiliza la unidad antes de la ejecución de la primera instrucción del cuerpo del programa.

En la sección de iniciación se inicializa cualquier estructura de datos (variables) que utilice la unidad y las hace disponibles (a través del *interface*) al programa que las utiliza. Comienza con la palabra reservada `begin` seguida por una secuencia de sentencias y termina con `end`. (Al igual que en los programas ordinarios se requiere un punto a continuación de `end`).

Cuando un programa que utiliza esta unidad se ejecuta, la sección de iniciación se llama antes que el cuerpo del programa se ejecute.

2.2.5. Ventajas de las unidades

La noción de unidad acentúa el carácter estructurado y modular de Turbo Borland Pascal para el diseño de grandes programas. Las unidades que representan módulos independientes pueden ser compiladas aisladamente, con independencia de un programa. Estas propiedades facilitan la creación de librerías, estándar o personalizadas (por el usuario).

Turbo Borland Pascal dispone de varias unidades estándar—predefinidas—conectadas a tareas específicas: gráficos, gestión de pantalla, etc. Cualquier usuario puede, a su vez, crear sus propias unidades, enriqueciendo así los recursos básicos. En general una unidad puede llamar a otras unidades.

2.3. CREACIÓN DE UNIDADES

Una unidad tiene una estructura muy similar a la de un programa.

La cláusula `uses` de la parte de *interface* sólo es necesaria en el caso de que la unidad actual llame a otras unidades. Todos los procedimientos y funciones en la sección de interfaz deben ser definidos en la sección de implementación.

El contenido de la parte de iniciación es optativo; si figuran en ellas instrucciones, se ejecutarán inmediatamente antes del principio de la ejecución del programa que utiliza esta unidad.

La cabecera de los procedimientos y funciones declaradas en la parte de *interface* deben ser idénticos a la cabecera de las mismas funciones definidas en la parte de implementación; sin embargo, es posible escribir la cabecera en forma abreviada en la sección de implementación.

Una vez que se dispone el código fuente de una unidad, se compila de igual forma que un programa, pero el archivo obtenido no es ejecutable directamente. Se trata de un archivo objeto con la extensión **TPU** (Turbo Borland Pascal Unit). Así, por ejemplo, si la unidad se graba con el nombre `Demo.Pas`, se guardará con el nombre `Demo.TPU`.

```
unit utilidad  ← nombre de unidad
           ha de corresponder
           con nombre de archivo
```

Cabecera de la unidad

```
interface
  uses Crt;    ← opcional
  procedure Frase(Texto : string);
```

Sección de interfaz

```
implementation
  uses Printer; ← opcional
  var
    MiVar : Integer
  procedure Frase;
  begin
    ClrScr;
    GotoXY ((80-Length (Texto)) div 2, 1)
    Write (Texto);
  end;
```

Sección de implementación

```
begin
  MiVar := 0
end.
```

Sección de iniciación

Figura 2.1. Estructura de una unidad.

Para utilizar una unidad en su programa, debe incluir una sentencia `uses` para indicar al compilador que está utilizando esa unidad.

```
program Prueba;
uses demo;
```

Turbo Borland Pascal espera que el nombre del archivo fuente tenga el mismo nombre de la unidad. Por ejemplo, la unidad `demo` se encuentra en el archivo `demo.tpu`. Si el nombre de la unidad fuera `demostra.TPU`, Turbo Borland Pascal tratará de encontrar un archivo `demostra.TPU` para evitar estas contrariedades se puede utilizar una directiva del compilador (`$U`), que permite especificar el nombre del archivo en que se encuentra la unidad. Así, por ejemplo, si la unidad `utilidad` se encuentra en el archivo `VARIOS.PAS` un programa puede llamar de la forma siguiente:

```
program Test;
uses
  Crt, {$U VARIOS.PAS} utilidad;
```

La directiva debe preceder inmediatamente al identificador de la unidad.

Cuando el compilador encuentra la cláusula **uses nombre-unidad**, busca la unidad *nombre-unidad* sucesivamente en:

- Biblioteca de base TURBO.TPL.
- El archivo *nombre-unidad.TPU* en el directorio actual, después en el directorio reservado a las unidades y especifica en el menú «Options» del entorno de trabajo.

Pasos de compilación de una unidad

1. Edición de la unidad.
2. Seleccionar la orden Compile/Compile (o pulse ALT-C).

Una vez sin errores, conviene activar la opción Compile a disk y volver a compilar. Turbo Borland Pascal crea un archivo con extensión TPU. Este archivo se puede dejar o mezclarlo con TURBO.TPL.

EJEMPLO 2.1

Escribir una unidad que conste a su vez de un procedimiento para intercambiar los valores de dos variables, así como calcular su valor máximo.

```
unit Demol;
interface
  procedure Intercambio (var I, J : integer);
  function Maximo (I, J : integer) : integer;
implementation
  procedure Intercambio;
  var
    Aux : integer;
  begin
    Aux := I;
    I := J;
    J := Aux
  end;

  function Maximo;
  begin
    if I>J
      then Maximo := I
      else Maximo := J
  end;
end.
```

Tras editar el programa, guárdelo en disco con el nombre Demol.PAS, a continuación compilar en disco. El código objeto resultante será Demol.TPU. Un programa que utiliza la unidad Demol es:

```

program Prueba;
uses
  Demol;
var
  X, Y : integer;
begin
  Write ('Introducir dos números enteros');
  ReadLn (X, Y);
  Intercambio (X,Y); WriteLn (X,'', Y);
  WriteLn ('el valor máximo es'. Maximo(X, Y));
end.

```

El programa Prueba utiliza la unidad Demol y, en consecuencia, puede disponer del procedimiento *Intercambio* y la función *Maximo*.

EJEMPLO 2.2

Construcción de una unidad con un procedimiento Intercambio y una función Potencia o Exponenciación (x^n ; x: real, n: entero).

```

unit Calculo;

interface          {sección de interfaz}
  procedure Intercambio (var X, Y : integer);
  {intercambia los valores de X e Y}
  function Potencia (X : real; n : integer) : real;
  {devuelve X a la potencia n-esima}

implementation      {sección de implementación}
  procedure Intercambio (var X, Y : integer);
  var
    Aux : Integer;
  begin
    Aux := X;
    X := Y;
    Y := Aux
  end;  {Intercambio}

  function Potencia (X : real; n : integer) : real;
  var
    I       : integer;
    Producto : real;
  begin {Potencia}
    Producto := 1;
    for I := 1 to n do
      Producto := Producto * X;
    Potencia := Producto
  end;  {Potencia}
end.  {Cálculo}

```

El procedimiento *Intercambio* y la función *Potencia* de la sección de implementación pueden ser definidos de un modo abreviado.

```
procedure Intercambio;
function Potencia;
```

Sin embargo, el método abreviado para declarar no es un estilo muy recomendado.

Reglas de compilación

Una unidad se compila tal como se compila un programa completo. La compilación se realiza en memoria. Sin embargo, antes que la unidad pueda ser utilizada por un programa o por otra unidad, debe ser compilado en disco. Normalmente el archivo que contiene una unidad debe tener el mismo nombre que la unidad.

2.3.1. Construcción de grandes programas

La característica de compilación independiente de las unidades facilita considerablemente la programación modular. Un programa grande se divide en unidades que agrupan procedimientos y funciones afines. Así, un programa puede ser dividido en diferentes módulos o unidades: A, B, C y D, y la estructura del programa podría ser ésta:

```
program Prueba;
uses
  Dos, Crt, Printer,
  A, B, C, D;
  (declaraciones de procedimientos y funciones del programa)
begin {programa principal}
end.
```

La compilación independiente de las unidades permite también superar el tamaño del código fuente de un programa de 64 K que viene limitado por el tamaño de un segmento del microprocesador 8086 (64 K); esta característica significa que el programa principal y cualquier segmento dado no pueden exceder un tamaño de 64 K.

Turbo Borland Pascal manipula las unidades de forma que el límite superior de almacenamiento ocupado por un programa puede llegar al límite máximo de memoria que puede soportar la máquina y el sistema operativo: 640 K en la mayoría de los PC. Sin la posibilidad de construcción de unidades, de Turbo Borland Pascal 4.0 y 5.0, los programas están limitados a 64 K.

Turbo Borland diferencia unidades de programas por las palabras reservadas con las que comienzan y actúa de acuerdo con ellas.

2.3.2. Uso de unidades

Las unidades creadas por el usuario se almacenan con la extensión TPU, mientras que las unidades estándar se almacenan en un archivo especial (TURBO .TPL) y se cargan automáticamente en memoria junto con el propio Turbo Borland Pascal.

El uso de una unidad o diferentes unidades añade muy poco tiempo (normalmente menos de un segundo) al tiempo de compilación de su programa. Si las unidades están cargadas en un archivo de disco independiente, se pueden necesitar unos segundos para leerlas del disco.

Un programa puede utilizar más de una unidad. En este caso todas las unidades se listan en la sentencia `uses`, separadas por comas, como en el siguiente ejemplo:

```
program Prueba;
uses Objetos, Unidad2;
```

Sólo las unidades realmente utilizadas en el programa necesitan ser listadas. Si `Unidad2` utiliza una unidad llamada `ObViejos`, pero el programa no incluye una llamada a algo definido en `ObViejos`, no es necesario listar `ObViejos` en la sentencia `uses`.

Las unidades pueden ser listadas en cualquier orden. Aunque no exigible, una buena regla a seguir es listar cada unidad antes de todas aquellas que la utilicen. Así, por ejemplo, si `Unidad2` utiliza `Objetos`, es buena práctica listar `Objetos` antes que `Unidad2`.

Si una unidad utiliza otra unidad, debe contener una cláusula `uses`. Por ejemplo, continuemos suponiendo que `Unidad2` utiliza las dos unidades `Objetos` y `ObViejos`, entonces `Unidad2` debe incluir una cláusula `uses` que indique al compilador su uso. La cláusula `uses` (si se necesita) se incluye normalmente en la sección de `interface`.

```
unit Unidad2;
interface
  uses ObViejos, Objetos;
```

2.3.3. Declaraciones/información pública y privada

La sección de `interface` no está limitada a contener sólo procedimientos y cabeceras. Puede contener declaraciones de tipos, constantes u otras clases.

Todos los objetos declarados en la sección de `interface` es información **pública** y puede ser referenciada en cualquier programa (o unidad) que utilice la unidad. La información que se desea ocultar del programa principal debe ser situada en la sección de **implementación**.

Además de los procedimientos y funciones cuyas cabeceras aparecen en la sección de `interface` de una unidad, ésta puede tener también otros procedimientos, funciones, constantes y/o otras declaraciones. Estas declaraciones se llaman, con frecuencia, *declaraciones privadas*. Son locales a la unidad y no se pueden ver referenciadas por cualquier programa (u otra unidad) que utilice la unidad. Por ejemplo, si un programa utiliza la unidad `Financa`, puede utilizar el procedimiento `Uno`, el procedimiento `Dos` y la constante `Tasa`. Sin embargo, no puede referenciar la constante `Banda` o la función `Conversión` que aparece en la sección de implementación. `Banda` y `Conversión` son locales a la unidad `Financa`. Pueden ser utilizadas en la unidad `Financa`, pero no tienen significado fuera de esa unidad.

```

unit Financa;
  (declara procedimiento para convertir capitales)

interface
  const
    Tasa = 4.25;
  procedure Uno(Capital : real);
  procedure Dos(Capital : real);
implementation
  const
    Banda = '**';
  function Conversion(Capital : real) : real;
  var
    FactorConversion : integer;
    CantidadPagar   : real;
  begin
    FactorConversion := 65;
    CantidadPagar   := Capital * FactorConversion;
    Conversion       := CantidadPagar
  end;

  procedure Uno (Capital : real);
  begin
    Write ('el interés es igual a :');
    WriteLn (Capital * 4.25);
  end;

  procedure Dos (Capital : real);
  begin
    WriteLn ('el interés es igual a :', Capital * 6);
  end;
end. {fin de Financa}

```

La sección de interface de una unidad es la sección *pública* y la sección de implementación es la sección *privada*.

Un programa que utilice la unidad puede referenciar cualquier cosa de la parte pública (interface), pero no puede referenciar nada en la parte de implementación. Este proceso de ocultación de la información es muy útil en el diseño de grandes programas.

2.3.4. Cláusula uses en la sección de implementación

En las versiones a partir de la 5.0 una cláusula **uses** se puede situar en la sección de implementación. Esta propiedad permite ocultación adicional de la información. Por ejemplo, supóngase que la unidad P utiliza la unidad Q. Esta cláusula **uses** puede ser situada en la sección de interface o en la sección de implementación. Si la cláusula **uses** se utiliza en la sección de implementación, entonces las declaraciones dadas en la unidad Q sólo pueden situarse en la sección de implementación de la unidad P. Ellas no pueden situarse en la sección interface de la unidad P.

Cuando se utiliza una cláusula **uses** en la sección de implementación, se sitúa inmediatamente después de la palabra reservada **implementation**.

```

unit P;
interface
  uses A, B, C;
  .
implementation
  uses Q;

```

2.3.4.1. Uso circular de unidades

La versión 5.0 permite el uso circular de unidades. Es posible a la unidad P utilizar la unidad Q y a la unidad Q utilizar la unidad P: *se pueden construir unidades mutuamente dependientes*. En ambos casos la cláusula uses se debe situar en la sección de implementación.

Si se escriben unidades con referencia circular, se debe utilizar la orden **Make** del menú **Compile** en lugar de la orden **Compile**.

La orden **Make** gestiona las unidades, compilando automáticamente cualesquiera unidades que necesiten ser compiladas o recompiladas.

EJEMPLO 2.3

El programa circular utiliza una unidad denominada Visualiz, que incorpora un procedimiento denominado EscribirEnPosicionXY, que a su vez se llama a otro procedimiento denominado VerError que pertenece a una unidad denominada Errores, que contiene una llamada a la unidad Visualizar.

```

program Circular;
{visualizar texto utilizando EscribirEnPosicionXY}
uses
  Crt, Visualiz;
begin
  ClrScr; {limpieza de la pantalla}
  EscribirEnPosicionXY (1, 1, 'Test');
  EscribirEnPosicionXY (90, 90,'Fuera de pantalla');
  EscribirEnPosicionXY (10, 2,'Retorno a pantalla')
end.

```

El procedimiento EscribirEnPosicionXY tiene tres parámetros: **x**, **y**, **mensaje**. **x**, **y**, coordenadas del punto en la pantalla de texto (de 25 × 80); **mensaje**, frase a visualizar en la posición (**x**, **y**).

Si las coordenadas (**x**, **y**) son válidas, es decir, $1 \leq x \leq 80$, $1 \leq y \leq 25$ se escribe el mensaje en la pantalla; si **x** e **y** no son válidos (la posición queda fuera de la pantalla), se visualiza un mensaje de error.

```

unit Visualiz;
interface
  procedure EscribirEnPosicionXY (X,Y:integer; Mensaje:string);
implementation
  uses Crt, Error;
  procedure EscribirEnPosicionXY (X,Y:integer; Mensaje:string);
  begin
    if (X in [1..80]) and (Y in [1..25])
    then
      begin
        GotoXY (X, Y,);
        Write (Mensaje)
      end
    else
      VerError ('Coordenadas XY fuera de rango')
  end
end.

```

El procedimiento VerError visualiza el mensaje 'Error Fatal' en la línea 25 de la pantalla.

```

unit Error;
interface
  procedure VerError (Cadena : string)
implementation
  uses Visualiz;
procedure VerError (Cadena : string)
begin
  EscribirEnPosicionXY (1,25,cadena)
end;
end.

```

Como habrá observado en la cláusula `uses` de las secciones de implementación, existen referencias mutuas (llamadas reciprocas) de las unidades posibles porque Turbo Borland Pascal puede compilar ambas secciones de interface completas.

Si se utiliza `Make` para compilar una de las dos unidades que referencia a la otra unidad, se compilarán ambas unidades.

2.4. UTILIZACIÓN DE UNIDAD ESTÁNDAR

El archivo TURBO.TPL, que contiene todas las unidades estándar, se carga en memoria central a la vez que el propio compilador y está disponible en cualquier momento, con la ayuda de la cláusula `uses`.

Un programa debe contener la cláusula `uses`, situada inmediatamente después de la cabecera, cuando utiliza objetos declarados en una unidad.

```

program Test;
uses
  Crt, Proceso;
const
  Mayor = 100;
type
  Palabra = string [2];
var
  Expresion : real;
  .
  .

```

Recordemos que las unidades estándar son:

```
System Overlay Dos Crt Printer Graph Graph3 Turbo3
```

En este capítulo se hará una breve introducción a las diferentes unidades. Todas las unidades contienen una serie de objetos que están declarados en la sección de *interface* (constantes, variables, tipos, procedimientos y funciones) y que son exportables o visualizables desde cualquier programa o unidad que utilice dichas unidades.

2.4.1. Unidad System

Esta unidad contiene todos los procedimientos y funciones estándar de Turbo Borland Pascal relativas a entradas/salidas, cadena de caracteres, cálculos en coma flotante, gestión de memoria, etc.

La unidad *System* tiene una configuración especial, se enlaza automáticamente en la compilación de cada programa y no precisa ser referenciada por la cláusula **uses system**.

2.4.2. Unidad Crt

Esta unidad proporciona un conjunto específico de declaraciones para entrada y salida: constantes, variables, procedimientos y funciones, que permiten el acceso al control de los modos de pantalla, al teclado, a los colores, al posicionamiento del cursor, etc.

La mayoría de los programas Turbo Borland Pascal que hacen uso de la pantalla para representaciones de salida recurren a la unidad *Crt*. Algunos procedimientos típicos son:

ClrScr	Borra la pantalla.
KeyPressed	Detecta la pulsación de una tecla.
Sound	Hace sonar el altavoz interno.
Window	Define una ventana de texto en la pantalla.

EJEMPLO 2.4

El siguiente programa permite comprobar la velocidad de presentación de los caracteres en la pantalla, creando una ventana de 10.000 caracteres aleatorios que se repiten tres veces.

```

program Velocidad_Caracteres;
uses
  Crt;
var
  j, k : integer;
begin
  Randomize;
  ClrScr;
  DirectVideo := true;
  window (1,1,40 + Random(2), 10+Random(15));
  for j := 1 to 2 do
  begin
    ClrScr;
    GotoXY (1, 1);
    if i = 1 then
      WriteLn ('acceso directo a memoria');
    else
      WriteLn ('no acceso directo a memoria');
    delay (200);
    for k := 1 to 10000 do
      Write (chr (Random (128) + 32));
    DirectVideo := not DirectVideo
  end
end.

```

2.4.3. Unidad DOS

Esta unidad contiene declaraciones, constantes, tipos, variables, procedimientos y funciones relacionadas con el sistema operativo DOS y la gestión de archivos. Los subprogramas que constituyen esta unidad no existen en Pascal estándar. Esta unidad no necesita ninguna otra unidad en su declaración.

Algunos procedimientos importantes son:

GetTime SetTime DiskSize GetAttr ...

y rutinas de bajo nivel:

MsDos Intr

que permiten invocar directamente cualquier llamada MS-DOS o interrupción del sistema.

Los registros del microprocesador son los tipos de datos que se asignan a los parámetros de las rutinas de bajo nivel *MsDos* e *Intr*.

EJEMPLO 2.5

Visualizar la hora en la esquina superior derecha de la pantalla.

```

program Hora;
uses
  Crt, Dos;

```

```

var
  Horas, Minutos, Segundos, Centesimas : Word;
begin
  CirScr:
  while not keyPressed do
  begin
    GotoXY (64, 1);
    GetTime (Horas, Minutos, Segundos, Centesimas);
    {hora reloj interno}
    WriteLn (horas:2, ':', minutos:2, ':', segundos:2, '.', centesimas:2)
  end
end.

```

2.4.4. Unidad *Printer*

La unidad *Printer* está concebida para facilitar la tarea del programador cuando ha de utilizar una impresora como dispositivo de salida. Permite enviar la salida estándar de Pascal a la impresora utilizando *Write* y *WriteLn*.

La inclusión de *Printer* en su programa define *Lst* como una variable de texto. Esta unidad requiere a su vez de la unidad *Crt*.

EJEMPLO 2.6

```

program impresora;
uses Printer;
var
  A, B, C : integer;
begin
  WriteLn (Lst,'este texto aparece en la impresora');
  Read (A, B);
  C := A + B;
  WriteLn (Lst, C : 6)
end.

```

Nota

Es posible imprimir en impresora sin llamar a la unidad *Printer*.

2.4.5. Unidad *Graph*

Esta unidad contiene las constantes, tipos, variables, procedimientos y funciones relacionadas con los gráficos, que permiten hacer uso de las capacidades gráficas de su PC.

Algunas de las rutinas a las que se tiene acceso con esta unidad son:

- Reconocimiento automático de un gran número de tipos de pantallas y adaptadores gráficos.
- Escrituras de texto con diferentes tipos y tamaños de caracteres.

- Desplazamiento de objetos gráficos.
- Trazado de figuras geométricas e histogramas.

2.5. SITUACIÓN DE LAS UNIDADES EN SUS DISCOS: ¿DÓNDE BUSCA TURBO BORLAND PASCAL LAS UNIDADES?

Las unidades existen físicamente en una de las dos formas siguientes: como archivo independiente con su nombre y una extensión TPU (Turbo Borland Pascal Unit), o como parte de un archivo llamado TURBO.TPL. Pueden existir muchos archivos con extensión TPU, pero sólo puede haber un archivo .TPL (Turbo Borland Pascal Library).

TURBO.TPL se carga en la memoria de la computadora siempre que se arranca el programa TURBO Borland Pascal, de modo que todas las unidades predefinidas, tales como **Crt**, **Dos**, etc., están siempre disponibles. No importa cuál sea la unidad de disco activa o en qué directorio está situado, su programa puede utilizar cualquiera de estas unidades y el programa TURBO encontrará la unidad. Si se conoce de antemano que algunas de las unidades residentes en TURBO.TPL no se van a utilizar, se pueden eliminar con la *utilidad* llamada TPUMOVER.EXE. Esta operación hará más pequeña TURBO.TPL y permitirá que la memoria sea utilizada para otras cosas; es decir, se pueden añadir las unidades del usuario al archivo TURBO.TPL.

Las unidades que diseña el programador, el sistema TURBO espera encontrarlas en el directorio activo de la unidad de disco activa. Esto significa que si tiene una unidad con procedimientos que utiliza con frecuencia, deberá transferir la unidad cada vez que cambia el directorio, o en caso contrario perderá el acceso a esa unidad cuando no esté en el directorio activo. Es preciso indicar entonces al sistema TURBO en qué directorio están las unidades.

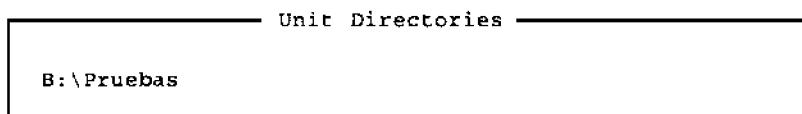
Desgraciadamente no se puede situar un especificador de unidad o de camino en la sentencia **uses**. Las sentencias siguientes no son válidas:

```
uses Crt, Dos, C:\UnidadesUsuario;
uses Crt, C:\Prueba\UnidadesUsuario;
```

Algunos métodos para indicar al sistema TURBO en qué directorio están las unidades son:

- Si los archivos de su unidad están en el directorio actual (por ejemplo, C:\Practicas) y el compilador está en C:\TURBO, se puede invocar el compilador desde C:\Practicas, el compilador las encontrará.
- Supongamos que están todas las unidades del usuario en el directorio Pruebas en la unidad de disco B y se desea indicar al sistema que busque siempre allí para encontrar las unidades. Las operaciones a realizar son:
 1. Activar el menú **Options**.
 2. Elegir la orden **Directories** de ese menú. Se produce una lista con los tipos de directorios que se pueden especificar.

3. Elegir de esa lista **Unit Directories** y aparecerá una ventana en la pantalla. Teclee B:\Pruebas y, tras pulsar RETURN, aparece:



El sistema ya está preparado para buscar en ese directorio las unidades que haya escrito. Cuando se encuentra una cláusula **uses**, el sistema busca la unidad en TURBO.TPL, si no la encuentra, busca en el directorio activo de la unidad de disco activa. Si no se encuentra en ninguna de esas posiciones, entonces busca en el directorio que se referenció como **Unit Directories**, es decir, en el ejemplo Pruebas, en la unidad B. En caso negativo el sistema produce un error.

- Se pueden utilizar directorios múltiples. En este caso se pueden listar todos los directorios separados por puntos y comas:

```
unit directories:B:\Practicas; B:\Pruebas; C:\DemoUno
```

En esta ocasión TURBO busca, como siempre, en TURBO.TPL y a continuación en el directorio activo para encontrar una unidad. Si no tiene éxito la búsqueda, se busca en el directorio Practicas de la unidad B. Si no existe ninguna unidad, entonces se busca en Pruebas de la unidad B. Si tampoco tiene éxito en la búsqueda, entonces se busca en el directorio DemoUno de la unidad C. Si tampoco tiene éxito la búsqueda, se producirá un mensaje de error.

La directiva **\$U** tiene prioridad sobre unidades del directorio actual o de los directorios especificados en el menú **Options**, de modo que si tiene una unidad idéntica en su directorio actual, **\$U** permite seleccionar una unidad del mismo nombre almacenada en cualquier parte de su disco.

TURBO supone que el nombre de la unidad y el nombre del archivo que contiene a la unidad son idénticos (por ejemplo, unidad Demo y archivos Demo.Pas y Demo.TPU).

Reglas de funcionamiento

Cuando aparece un nombre de unidad en una directiva **uses**, Turbo busca esta unidad (módulo) sucesivamente en:

1. Entre las unidades residentes en la biblioteca TURBO.TPL.
2. En el directorio actual.
3. Los directorios citados en el menú **Options**.

El programa TPUMOVER

Las unidades estándar residen en la biblioteca TURBO.TPL y se cargan automáticamente cuando arranca Turbo Borland Pascal. Al objeto de ahorrar memoria se pueden transferir unidades poco utilizadas, tales como *Turbo3* y *Graph*, fuera del archivo TURBO.TPL, mediante la utilidad TPUMOVER. En otras palabras, esta utilidad se puede utilizar para añadir o borrar desde TURBO.TPL.

2.6. IDENTIFICADORES IDÉNTICOS EN DIFERENTES UNIDADES

Es válido tener identificadores idénticos dentro de dos unidades y utilizar ambas unidades en el mismo programa. Es decir, es posible tener una unidad llamada *Pantalla* que contenga un procedimiento denominado *ClrScr*, de igual nombre que el procedimiento, ya conocido, de borrado de la pantalla de la unidad estándar *Crt*.

```
uses DOS, Crt, Pantalla;
```

En este caso se dispondría de dos procedimientos *ClrScr*. ¿Cuál utilizar? Turbo Borland Pascal utiliza el *último* procedimiento *ClrScr* que se encuentra al explorar las unidades en la sentencia *uses*. En este ejemplo explora primero *Crt* y luego *Pantalla*, y por consiguiente, se emplea el procedimiento *ClrScr* escrito por el usuario. Si se intercambian los nombres de las unidades

```
uses DOS, Pantalla, Crt;
```

el compilador utilizará ahora el procedimiento estándar *ClrScr*.

Un sistema de diferenciar los identificadores idénticos es utilizar el identificador de la unidad y un punto.

```
program Pruebas;
uses
  Dos, Crt, Pantalla;
begin
  Crt.ClrScr; {limpieza de la pantalla de texto}
  ClrScr;      {limpieza de cualquier otra pantalla de texto}
  ...
end.
```

Con este método es posible referenciar y diferenciar identificadores idénticos en unidades diferentes. En el caso de que ningún nombre de unidad preceda al identificador o referencia, se asigna la última unidad encontrada en la exploración de unidades en la sentencia *uses*.

2.7. SÍNTESIS DE UNIDADES

Una vez visualizadas todas las características de las unidades, vamos a realizar una síntesis de las mismas al objeto de que el lector pueda asimilar totalmente y comenzar a modularizar sus programas con la máxima eficiencia a partir de ese momento.

Un programa puede disponer de constantes, definiciones de tipos y subprogramas precompilados, incluyendo una sentencia uses después de la cabecera de un programa.

2.7.1. Estructura de una unidad

Una unidad es un módulo. No es un programa ejecutable. Existen unidades estándar (por ejemplo, Crt; si desea acceder a procedimientos de esa unidad como ClrScr, Window, debe incluir mediante una cláusula uses la unidad Crt: uses Crt) y unidades creadas por el usuario.

Una unidad tiene una estructura ya conocida que contiene una *cabecera*, una parte **interface**, una parte **implementation** y, evidentemente, una parte de *iniciación*.

Los identificadores que aparecen en la sección de *interface* son visibles desde cualquier programa que utiliza la unidad, pero los detalles de su estructura interna sólo se pueden conocer en la sección de *implementación*.

Desde el punto de vista profesional, usted puede proteger su código fuente, distribuyendo en la documentación del programa sólo la sección de *interface* y reservándose los algoritmos y el código fuente de la sección de *implementación*.

La cabecera de los procedimientos y funciones declaradas en la sección de *interface* debe ser idéntica a la cabecera de las mismas funciones definidas en la sección de implementación; sin embargo, es posible escribir la cabecera en forma abreviada en la sección de implementación. Los dos modelos de la unidad Base que siguen después en los modelos son correctos y equivalentes.

```

unit <identificador>; {cabecera obligatoria}
{el identificador se utiliza como nombre del archivo
de la unidad ya compilada; la extensión es .TPU}

interface
  uses <lista de unidades> {opcional}
  {declaraciones públicas o visibles son opcionales}
  const .....
  type .....
  var .....

  procedure ... {sólo cabecera}
  function ... {sólo cabecera}

implementation
  {declaraciones privadas: locales a la unidad}
  uses <lista de unidades>
  const .....
  type .....
  var .....

```

```

procedure .... {cabecera y cuerpo del procedimiento}
function .... {cabecera y cuerpo de la función}
[begin           {selección de inicialización: opcional}
.
.
.
end.            {obligatoria, fin de implementación}

```

Modelo 1

```

unit Base;
interface
  uses
    Crt;
  procedure Mensaje (Texto : string);
implementation
  procedure Mensaje;
  begin
    ClrScr;
    GotoXY ((80 - Length (Texto)) div 2, 4);
    Write (Texto)
  end;
begin
  ...
end.

```

Modelo 2

```

unit Base;
interface
  uses
    Crt;
  procedure Mensaje (Texto : string);
implementation
  procedure Mensaje (Texto: string);;
  begin
    ClrScr;
    GotoXY ((80 - Length (Texto)) div 2, 4);
    Write (Texto)
  end;
begin
  ...
end.

```

2.7.2. Excepciones en la escritura de unidades

Al escribir unidades debe tener en cuenta estas limitaciones:

- La declaración **forward** de los subprogramas no está permitida (no es necesaria) en una unidad.

- Si un subprograma se declara como **inline**, su definición de código debe aparecer en la sección de interface y el identificador del subprograma no debe aparecer en la sección *implementación*.
- Si un subprograma es declarado **external**, su identificador no debe aparecer en la sección *implementation*.

2.7.3. Compilación de unidades

La compilación de una unidad se efectúa del mismo modo que la compilación de un programa, con la diferencia de que al compilar el programa se obtiene un archivo autónomo ejecutable (**.EXE**) y al compilar una unidad se obtiene un archivo en código objeto (no ejecutable) como resultado, con extensión **.TPU** (Turbo Borland Pascal Unit), aunque con el mismo nombre que el archivo de código fuente. Este código objeto de un archivo **.TPU** sólo es enlazable (*linkable*) por el compilador de Turbo Borland Pascal y no es compatible con archivos **.OBJ** creados por otros compiladores.

Para crear unidades que pueda automáticamente recompilar con las opciones **Make** y **Build** de Turbo Borland Pascal se deben seguir estas reglas:

- El identificador de unidad (especificado) en la cabecera debe ser el mismo que el nombre del archivo fuente (sin incluir la extensión **.PAS**).
- El nombre del archivo fuente debe tener una extensión (**.PAS**).
- El archivo **.TPU** debe estar en el directorio actual o en un directorio especificado con las opciones **Options/Directories/Unit** directories. El código del programa fuente debe estar en el directorio actual.

2.8. OTROS MÉTODOS DE ESTRUCTURAR PROGRAMAS: INCLUSIÓN, SOLAPAMIENTOS Y ENCADENAMIENTO

Además de las unidades, existen dos métodos para estructurar o construir grandes programas a partir de Turbo Borland Pascal 5.0: *los archivos de inclusión (include)* y *los solapamientos (overlays)*. Aunque es necesario hacer constar que si bien en las versiones 3.0 y anteriores eran imprescindibles, su utilidad ha decaído en 5.0, 6.0 y 7.0 en beneficio de las unidades.

2.8.1. Los archivos de inclusión o incluidos (include)

Como ya conoce el lector, la directiva *Incluir Archivos (include)* (**\$I**) permite incluir código fuente en el programa que está en memoria, a partir de archivos externos de discos. El código del programa ejecutable actúa como si el código incluido estuviera físicamente presente en el archivo que se está compilando.

La directiva **\$I<nombrearchivo>** permite incluir el archivo denominado **<nombrearchivo>** durante la compilación. Éste es un método para dividir su código fuente en segmentos más pequeños. Cuando el compilador encuentra la directiva

`$I <nombrearchivo>`, abre el archivo del disco `<nombrearchivo>` y comienza a compilarlo. El archivo `<nombrearchivo>` no se lee realmente en memoria como un todo; el compilador lee simplemente linea a linea, compila la linea a código máquina y a continuación lee la siguiente linea.

Utilizar unidades, no archivos de inclusión, para bibliotecas de subprogramas.

El uso actual de los archivos de inclusión se suele reducir a archivos que contienen procedimientos o funciones sencillas; o incluso la definición de uno o más tipos de datos; como ejemplos típicos: procedimientos o funciones de búsqueda, ordenación, etc.

2.8.2. Los solapamientos (*overlays*)

Si se desarrollan grandes aplicaciones que deban correr sobre sistemas con recursos de memoria limitados (640 K o menos), la técnica de solapamientos es indispensable.

Los solapamientos (*overlays*) son partes de un programa que comparten una zona de memoria común. Sólo las partes del programa que se requieren para una función dada residen en memoria al mismo tiempo; se cargan (sobreescreiben) una sobre otra durante la ejecución. Los solapamientos pueden reducir significativamente los requisitos totales de memoria de un programa en tiempo de ejecución. De hecho, con solapamientos se pueden ejecutar programas que son mucho más grandes que la memoria total disponible, ya que sólo partes del programa residen en memoria en un instante dado.

La idea básica del solapamiento es ésta: se divide un programa en partes y se prepara de modo que diferentes partes compartan el mismo lugar en memoria. Cuando el programa principal comienza la ejecución, un solapamiento (segmento de programa) se carga en una zona libre (*slot*) de la memoria. Cuando se requiere el segundo solapamiento, se carga en memoria en la misma zona que el primer solapamiento, superponiendo el primer solapamiento. Posteriormente, cuando un tercero, cuarto o quinto solapamiento se requieren, simplemente se cargan en la parte superior de cualquier solapamiento que hubiese ocupado anteriormente la zona del solapamiento. De este modo, se pueden manejar megabytes de código que pueden correr sobre memorias de 640 K, 256 K o incluso menos. Simplemente necesita dividir esos megabytes de código en un número suficiente de solapamientos.

2.8.2.1. Gestión de los solapamientos

El sistema de solapamientos de la versión 7.0 y anteriores está basado en unidades. La gestión de los mismos se realiza a nivel de unidad; ésta es la parte más pequeña de un programa que se puede hacer en un solapamiento. Cualquier número de unidades puede ser especificado como solapamiento, lo que significa que cada uno ocupará la misma región de memoria cuando se cargan.

Todas las unidades especificadas como solapamientos (*overlays*) se enlazan en un archivo independiente, con el mismo nombre que el programa principal, pero con una

extensión .OVR. En otras palabras, si un programa con solapamientos MIPROG.PAS se compila, Turbo Borland Pascal genera un archivo de solapamiento (MIPROG.OVR) y un archivo ejecutable (MIPROG.EXE). El archivo .EXE contiene las partes estáticas (no recubiertas) del programa y el archivo .OVR contiene todas las unidades de solapamiento que se intercambiarán en la memoria durante la ejecución del programa.

Existe una sola zona de memoria para solapamientos, se denomina **memoria intermedia de solapamiento** (*overlay buffer*), que reside en memoria entre el segmento de la pila (*stack*) y el segmento para variables dinámicas (*heap*). Inicialmente el tamaño de la memoria intermedia de solapamiento se hace tan grande como sea necesaria, de modo que puede contener la unidad de solapamiento más grande; posteriormente puede incrementarse su tamaño, si así se desea. Si no existe bastante memoria disponible cuando se carga la unidad, se visualizarán mensajes de error:

```
Program too big to fit in memory
Not enough memory to run program
```

El gestionador de solapamientos tiene la posibilidad de cargar el archivo recubierto en memoria ampliada o expandida (EMS) cuando exista espacio disponible suficiente. Si la librería en tiempo de ejecución detecta memoria EMS (*Expanded Memory Specification*) libre, el archivo de solapamiento se sitúa en la memoria EMS RAM.

La Figura 2.2. muestra el sistema de funcionamiento de los solapamientos.

2.8.3. La unidad Overlay (generación de solapamientos)

Turbo Borland Pascal proporciona un gestionador de archivos que manipulan acceso a solapamientos en una unidad llamada **Overlay**. Si se trata de utilizar solapamientos, se debe incluir **Overlay** en su sentencia **Uses**, *antes de todas* las unidades a recubrir. Para diseñar programas con solapamientos se deben seguir las siguientes reglas:

1. Una unidad especificada como solapamiento debe ser compilada con la directiva \$O. Esta directiva define una unidad como un solapamiento.

La directiva \$O tiene dos formatos: un formato de tipo comutador y un formato de parámetro.

- *formato comutador* { \$O+ }
 { \$O- }
- *formato parámetro* { \$Oparámetro }

\$O+ situada en la unidad del archivo fuente, se *activa* su uso como un solapamiento, pero no se requiere su uso como solapamiento; puede ser enlazada, todavía, normalmente si se desea.

\$O xmodem indica al programa principal que la unidad llamada Xmodem se trate como un solapamiento.

Las directivas \$O deben ser situadas después de la cláusula uses.

```

program Mortimer;
{$F+}
uses overlay, Dos, Crt, Circulos, Xmodem, Kermit, Parser;
{$O Xmodem}
{$O Kermit}
{$O Parser}

```

2. En el ejemplo anterior se observa la directiva **\$F+**. La razón es que todas las unidades utilizadas por el programa, así como el propio programa, deben ser compiladas con la directiva **Far Calls** activada (**{\$F+}**). Por esta causa, **\$F+** se sitúa inmediatamente después de la sentencia **program**.
3. Inicializar el gestionador (manipulador) de solapamientos. El código de iniciación se debe situar antes de la primera llamada a una rutina de solapamiento, y en particular al principio de la parte de sentencias del programa. La llamada se realiza con **OvrInit**, cuyo formato es **OvrInit (nombrearchivo)**.

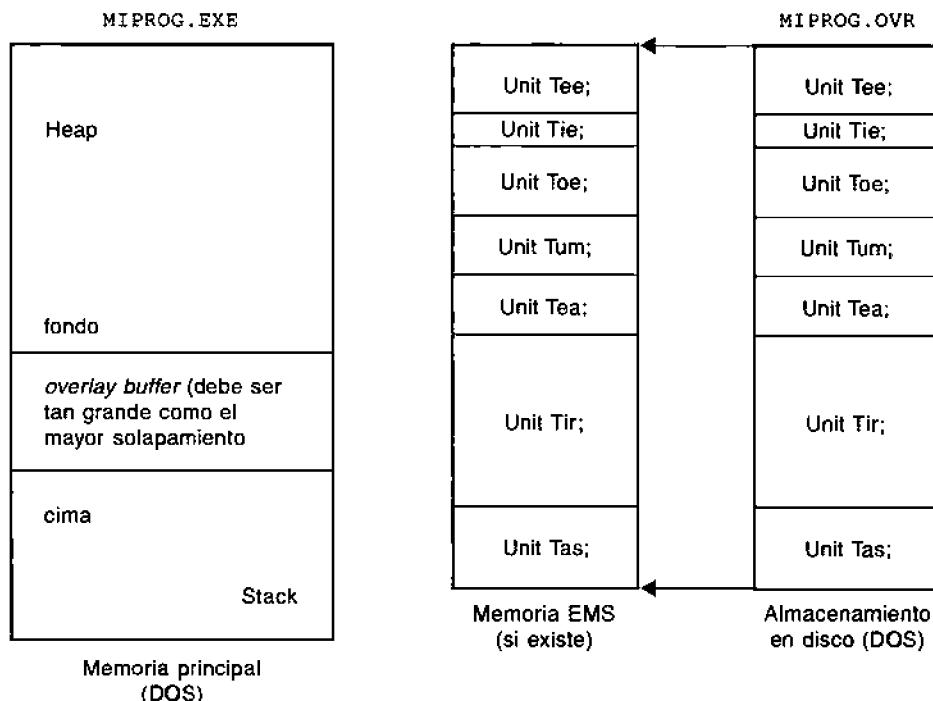


Figura 2.2. Archivos y memorias de solapamiento.

Ejemplo

```

begin
  OvrInit ('MIPROG.OVR')
end;

```

2.8.3.1. Constantes y variables de la unidad Overlay

```
const
  OvrOk          = 0;
  OvrError       = -1;
  OvrNotFound    = -2;
  OvrNoMemory    = -3;
  OvrIoError     = -4;
  OvrNoEMSDriver = -5;
  OvrNoEMSSMemory= -6;
var
  OvrResult : integer;
```

2.8.3.2. Procedimientos y funciones de la unidad Overlay

<i>Procedimientos</i>	<i>Funciones</i>
OvrInit	OvrGetBuf
OvrInitEMS	
OvrSetBuf	
OvrClearBuf	

OvrClearBuf

Este procedimiento permite borrar temporalmente la memoria intermedia de solapamiento; esta operación se debe realizar cuando se necesite utilizar la memoria que ocupa.

OvrClearBuf

OvrGetBuf

Devuelve el tamaño actual de la memoria intermedia de solapamiento, en *VarLongInt*.

VarLongInt := OvrGetBuf

OvrInit

Inicializa el gestionador de solapamientos y abre el archivo de solapamiento.

OvrInit (nombrearchivo)

OvrInitEMS

Verifica que existe un controlador **EMS** y suficiente memoria **EMS** para contener el archivo de solapamiento.

OvrInitEMS

OvrSetBuf

Establece el tamaño de la memoria intermedia de solapamiento (*overlay buffer*).

```
OvrSetBuf (VarLongInt)
```

VarLongInt variable de tipo *longint*

2.8.4. Encadenamiento de archivos compilados

Turbo Borland Pascal produce archivos .EXE en lugar de los archivos más simples .COM que producirá la versión 3; por esta razón no puede soportar encadenamiento de programas con datos compartidos. Se puede utilizar el procedimiento Exec de Turbo Borland Pascal 5, 6 y 7 para transferir el control de un programa a otro, pero cada programa tendrá su propio segmento de datos.

RESUMEN

Una unidad es una colección de procedimientos y funciones de propósito general, guardadas en disco de forma compilada. Las unidades se pueden utilizar para construir su propia librería de procedimientos y funciones predefinidas. Esta librería puede ser utilizada en diferentes proyectos y por diferentes equipos de programadores.

Las unidades refuerzan el concepto de abstracción de datos. Esto se consigue situando los detalles sobre procedimientos *implementados* en una unidad, y por consiguiente *ocultando* estos detalles.

Sintaxis de una unidad

```
unit <nombre>
interface
uses <lista de unidades utilizadas por esta unidad>
      (se omite si no se utiliza ninguna unidad)
      <cabezas de procedimientos y funciones>
      <otras declaraciones>
implementation
      <declaraciones completas de procedimientos y funciones>
begin  {no se necesita si no existen sentencias de inicialización}
      .
      .
end.
```

EJERCICIOS

- 2.1. Diseñar una unidad Lib1 que defina las tres funciones siguientes, que puedan ser utilizadas posteriormente por cualquier programa:

```

function Mín (X1,X2 : integer) : integer;
{devuelve el elemento más pequeño de X1 y X2}
function Máx (X1,X2 : integer) : integer;
{devuelve el elemento mayor de X1 y X2}
function Media (X1,X2 : integer): real;
{devuelve la media de X1 y X2}

```

Escribir un programa que haga uso de la unidad.

- 2.2. Construir una unidad que contenga las siguientes funciones estadísticas de un vector x de n elementos.

- Desviación media ($DM = \frac{1}{n} \sum_{i=1}^n |x_i - \bar{x}|$)

$$\text{donde } \bar{x} = \frac{1}{n} \sum_{i=1}^n x_i = \frac{1}{n} (x_1 + x_2 + \dots + x_n)$$

$||$ es el valor absoluto

- Media cuadrática ($MC = \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2}$)
- Media armónica ($MA = \frac{n}{\sum_{i=1}^n \left(\frac{1}{x_i}\right)}$)

- 2.3. Escribir una unidad llamada Grados que incluya funciones para convertir grados sexagesimales y centesimales a radianes. Diseñar un programa que permita calcular los valores de las funciones trigonométricas seno, coseno y tangente de una tabla de valores angulares dados en grados sexagesimales y centesimales (incremento del ángulo en la tabla, 1 grado).
- 2.4. Dadas dos matrices, A y B , donde

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \quad \text{y} \quad B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & & & \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix}$$

Diseñar una unidad que contenga los procedimientos o funciones: *producto* y *suma* de A y B .

- 2.5. Diseñar una unidad que soporte las siguientes dos funciones de tiempo:

- *Fecha* (convierte los parámetros numéricos proporcionados por el procedimiento *GetDate* en una cadena con el formato DiaSemana Dia Mes Año: ds DD MMAA).
- *Hora* (convierte valores numéricos proporcionados por el procedimiento *GetTime* en una cadena de formato: 'hh mm ss').

Nota

Los procedimientos `GetTime` y `GetDate` están predefinidos en la unidad DOS.

- 2.6. Escribir una unidad que manipule tipos de datos abstractos Enteros grandes (enteros de al menos 2 dígitos). Debe proporcionar números positivos y negativos, y las operaciones de suma, resta, multiplicación y división.
- 2.7. Escribir una unidad cuyos subprogramas básicos deben ser:
 - Operación traspuesta de una matriz.
 - Multiplicaciones de dos matrices cuadradas.
 - Suma de dos matrices cuadradas.
- 2.8. Crear una unidad que se componga de procedimientos y funciones que realicen las tareas siguientes:
 - Hacer el cursor visible o invisible.
 - Emitir un sonido (pitido «bip»).
 - Borrar la parte de pantalla situada entre dos líneas.
 - Convertir las letras de una cadena de caracteres en mayúsculas.
 - Convertir un valor numérico en una cadena de caracteres.
 - Insertar caracteres.
- 2.9. Escribir una unidad que contenga los procedimientos de cambios de base de numeración: paso de base 2 a base 16, base 2 a base 10 y base 10 a base 2.
- 2.10. Escribir una unidad que contenga procedimientos y funciones que efectúen las tareas siguientes:
 - Desplazar una cadena de caracteres en la pantalla, de izquierda a derecha o viceversa y de velocidad ajustable.
 - Visualizar una ventana.
 - Mover el cursor por la pantalla.
- 2.11. Escribir la unidad `Prueba` que contenga dos funciones para asegurar divisiones correctas y evitar la división por cero, que produce un error irrecuperable en la ejecución del programa. *Aplicación práctica:* Escribir un programa `DemoPrueba` que utilice las funciones creadas en la unidad `Prueba`.
- 2.12. Crear una unidad `Aleatori` que contenga tres funciones que proporcionen valores generados aleatoriamente de tres tipos diferentes: enteros, cadenas y lógicos. Las funciones a definir se denominan `EnterosA`, `CadenasA`, `LogicosA`.

<code>EnterosA</code>	Proporciona un entero comprendido en un intervalo de valores específicos Min y Max. <code>Enteros (Min y Max).</code>
<code>CadenasA</code>	<code>EnterosA(5, 50)</code> genera un entero aleatorio entre 5 y 50. Proporciona una cadena de letras mayúsculas arbitrarias.
<code>LogicosA</code>	<code>CadenaA(n)</code> n , número de letras. Proporciona un valor lógico aleatorio.

- 2.13. Escribir un programa que visualice seis líneas que contengan cada una de ellas un entero entre 300 y 400, una cadena de ocho letras y un valor true o false, todos ellos aleatorios.

325	MNRHQANT	true
346	LOTVRSTN	false
371	MSPBVTRL	true
382	NZJMKSB	false
343	FGRILTRP	true

Nota: Utilice la unidad Aleatori.

- 2.14. El programa siguiente lee un archivo de entrada (programa fuente en Pascal) y lo convierte en otro archivo de salida (programa fuente en Pascal) con las palabras reservadas de Turbo Borland Pascal en negritas y mayúsculas.

REFERENCIAS BIBLIOGRÁFICAS

1. Borland: *Reference Guide. Turbo Borland Pascal 5.0.* Capítulo 13.
Si desea trabajar con solapamientos, este capítulo le complementará las ideas expuestas en la sección 2.8.
2. Duntzman, Jeff: *Complete Turbo Borland Pascal.* Third Edition. Scott Foreman, 1989.
Es una excelente obra para profundizar en Turbo Borland Pascal. Recomendada para usuarios expertos y como ampliación de conocimientos tras la lectura de esta obra.
3. Joyanes, Luis: *Turbo/Borland Pascal 7. Iniciación y Referencia,* Madrid, McGraw-Hill, 1997.

Abstracción de datos: tipos abstractos de datos y objetos

CONTENIDO

- 3.1. El papel (el rol) de la abstracción.
- 3.2. Un nuevo paradigma de programación.
- 3.3. Modularidad.
- 3.4. Diseño de módulos.
- 3.5. Tipos de datos.
- 3.6. Abstracción en lenguajes de programación.
- 3.7. Tipos abstractos de datos.
- 3.8. Tipos abstractos de datos en Turbo Borland Pascal.
- 3.9. Orientación a objetos.
- 3.10. Reutilización de software.
- 3.11. Lenguajes de programación orientados a objetos.
- 3.12. Desarrollo tradicional frente a orientado a objetos.
- 3.13. Beneficios de las tecnologías de objetos (TO).

RESUMEN.

EJERCICIOS.

En este capítulo examinaremos 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 TAD soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado *objeto*.

3.1. 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. Para comprender lo mejor posible la importancia de las técnicas orientadas a objetos, revisemos cuáles han sido los diferentes mecanismos utilizados por los programadores para controlar la complejidad. Entre todos ellos se destaca la *abstracción*. Como describe Wulf: «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».

En esencia, *la abstracción es la capacidad para encapsular y aislar la información, del diseño y ejecución*. En otro sentido, las técnicas orientadas a objetos se ven como resultado de una larga progresión histórica que comienza en los procedimientos y sigue en los módulos, tipos abstractos de datos y objetos.

3.1.1. La abstracción como un proceso mental natural

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 lo mismo que el diseño de software; aunque el desarrollo de software es único, el diseño de software 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 nueve 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 nueve dígitos se organizan en una única entidad (el número de teléfono). 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.

3.1.2. 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 abstracciones 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 fue aumentar los niveles de abstracción. En esencia, la *abstracción* supone la capacidad de

encapsular y aislar la información de diseño y ejecución. En un determinado sentido, las técnicas orientadas a objetos pueden verse como un producto natural de una larga progresión histórica que va desde las estructuras de control, pasando por los procedimientos, los módulos, los tipos abstractos de datos y los objetos.

En las siguientes secciones describiremos los mecanismos de abstracción que han conducido al desarrollo profundo de los objetos: *procedimientos, módulos, tipos abstractos de datos y objetos*.

3.1.3. Procedimientos

Los procedimientos y 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 el interfaz visible a su trabajo, es decir, cuatro rutinas: meter, sacar, 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 Pascal existe el ámbito local y global. 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.

3.1.4. 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—; asimismo 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.

3.1.5. 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 ellas 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 apartados 2 y 3. Los tipos abstractos de datos se implementan con *módulos* en Modula-2 y *paquetes* en CLU o Ada.

3.1.6. 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.

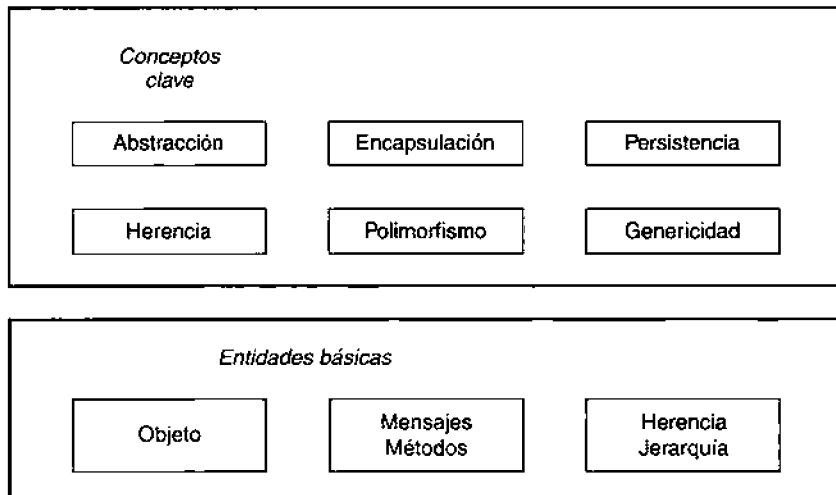


Figura 3.1. Principios básicos de la orientación a objetos.

3.2. UN NUEVO PARADIGMA DE PROGRAMACIÓN

La *programación orientada a objetos* (POO)¹ se suele conocer como un nuevo *paradigma* de programación. Otros paradigmas conocidos son: el *paradigma de la programación imperativa* (con lenguajes tales como Pascal o C), el *paradigma de la progra-*

¹ En inglés, OOP (Object Oriented Programming).

mación lógica (PROLOG) y el paradigma de la programación funcional (Lisp). El significado de *paradigma*² (*paradigma*, en latín; *paradigma*, en griego) en su origen significaba un ejemplo ilustrativo, en particular enunciado modelo que mostraba todas las inflexiones de una palabra. En el libro *The Structure of Scientific Revolutions* el historiador Thomas Kuhn³ [Kuhn 70] describía un paradigma como un conjunto de teorías, estándar y métodos que juntos representan un medio de organización del conocimiento: es decir, un medio de visualizar el mundo. En este sentido la programación orientada a objetos es un nuevo paradigma. La orientación a objetos fuerza a reconsiderar nuestro pensamiento sobre la computación, sobre lo que significa realizar computación y sobre cómo se estructura la información dentro del computador.

Jenkins y Glasgow observan que «la mayoría de los programadores trabajan en un lenguaje y utilizan sólo un estilo de programación. Ellos programan en un paradigma forzado por el lenguaje que utilizan. Con frecuencia, no se enfrentan a métodos alternativos de resolución de un problema, y por consiguiente tienen dificultad en ver la ventaja de elegir un estilo más apropiado al problema a manejar». Bobrow y Stefik definen un estilo de programación como «un medio de organización de programas sobre la base de algún modelo conceptual de programación y un lenguaje apropiado para hacer programas en un estilo claro». Sugiere que existen cuatro clases de estilos de programación:

- Orientados a procedimientos *Algoritmos*
- Orientados a objetos *Clases y objetos*
- Orientados a lógica *Expresado en cálculo de predicados*
- Orientados a reglas *Reglas if-then*

No existe ningún estilo de programación idóneo para todas las clases de programación. La orientación a objetos se acopla a la simulación de situaciones del mundo real.

En POO, las entidades centrales son los objetos, que son tipos de datos que encapsulan con el mismo nombre estructuras de datos y las operaciones o algoritmos que manipulan esos datos.

3.3. 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.

² Un ejemplo que sirve como modelo o patrón: *Dictionary of Science and Technology*, Academic Press, 1992.

³ Kuhn, Thomas S.: *The Structure of Scientific Revolution*, 2.^a ed., University of Chicago Press, Chicago, 1970.

Si 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.

3.3.1. 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 implantadas estáticamente llamadas *atributos*, eventualmente asociadas 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».

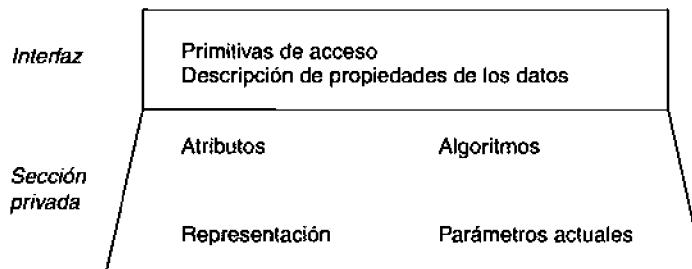


Figura 3.2. Estructura de un módulo.

3.3.2. 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++ se conocen como *clases* y en Turbo Borland Pascal, los módulos se denominan *unidades*. La sintaxis de las unidades diferencia entre el interfaz y la implementación del módulo. Las dependencias entre unidades se pueden declarar sólo en un interfaz del módulo. Ada va más lejos y define el *paquete* en dos partes: la especificación del paquete y el cuerpo del paquete. Al contrario que Object Pascal, Ada permite que la conexión entre módulos se declaren independientemente en la especificación y en el cuerpo de un paquete.

Interfaces adecuados

En la estructuración de un programa en unidades es beneficioso que existan *pocos interfaces* y que estos sean pequeños. Es conveniente que existan pocos enlaces entre los diferentes módulos en que se descompone un programa. *El interfaz de un módulo* es la parte del módulo (datos, procedimientos, etc.) que es visible fuera del módulo.

Los interfaces deben ser también *pequeños* (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 (Figura 3.3).

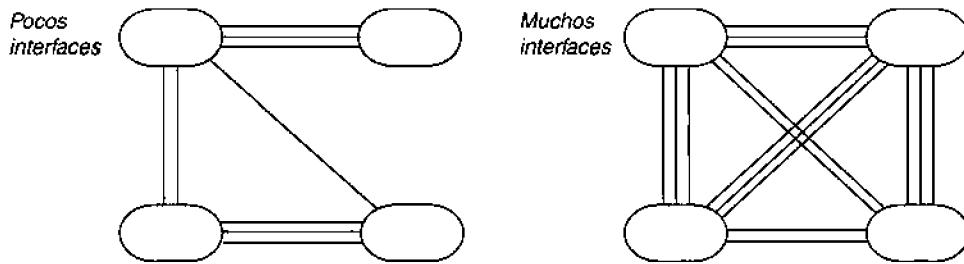


Figura 3.3. Interfaces adecuados (pocos-muchos).

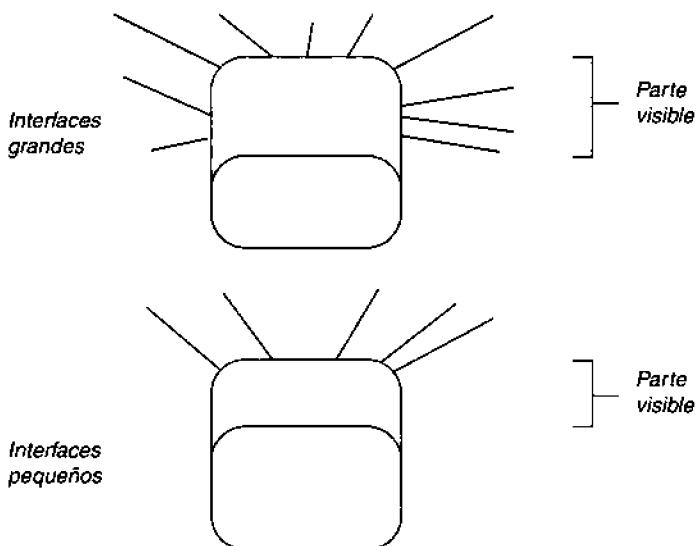


Figura 3.4. Interfaces adecuados (grandes-pequeños).

Interfaces explícitos

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*⁴, 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.

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.

⁴ 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, núm. 7, noviembre-diciembre 1993.

3.4. 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 Booch⁵ define la modularidad como *la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados*.

3.4.1. 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.

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 3.1). 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.

⁵ Booch, Grady: *Object-Oriented Design with applications*, Benjamin Cummings, 1991, pág. 52.

Tabla 3.1. Clasificación del acoplamiento del módulo

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

3.4.2. 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 3.1 muestra la clasificación de acoplamientos de módulos y su grado de acoplamiento. La Tabla 3.2 muestra los grados de cohesión: baja cohesión (no deseable) y alta cohesión (deseable), así como los diferentes tipos de cohesión.

Idealmente se buscan módulos altamente cohesivos y débilmente acoplados.

Tabla 3.2. Clasificación de cohesión de módulos

Tipos de cohesión	Grado de cohesión	Grado de mantenimiento
Por coincidencia	Bajo	Bajo
Lógica		
Temporal		
Por procedimientos		
Por comunicaciones		
Secuencial		
Funcional		
Informacional	Alto	Alto

3.5. TIPOS DE DATOS

Todos los lenguajes de programación soportan algún tipo de dato. Por ejemplo, el lenguaje de programación convencional Pascal soporta tipos base tales como enteros, reales y caracteres, así como tipos compuestos tales como *arrays* (vectores y matrices) y registros. Los tipos abstractos de datos extienden la función de un tipo de datos; ocultan la implementación de las operaciones definidas por el usuario asociadas con el tipo de datos. Esta capacidad de ocultar la información permite el desarrollo de componentes de software reutilizables y extensibles.

Un tipo de dato es un conjunto de valores, y un conjunto de operaciones definidas por 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 son. 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)*, y por ejemplo en C y C++ son entre otros: *int*, *char* y *float*.

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, Pascal soporta registros y arrays.

En lenguajes convencionales tales como C, Pascal, etc., las operaciones sobre un tipo de dato son composiciones de constructores de tipo y operaciones de tipos bases.

Operaciones = Operaciones constructor + Operaciones base

Algunos tipos de constructores incluyen registros, arrays, listas, conjuntos, etc.

3.6. 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.

3.6.1. 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 amplian 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. Las abstracciones 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 Pascal estándar para establecer abstracción procedimental es el subprograma (procedimientos y funciones). 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 Pascal 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.

3.6.2. Abstracción 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 de uno 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 simi-

lar 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. 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 la *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.

3.7. TIPOS ABSTRACTOS DE DATOS (TAD)

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 Turbo Borland Pascal un 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.

$$\boxed{\text{TAD} = \text{Representación (datos)} + \text{Operaciones (funciones y procedimientos)}}$$

Las operaciones desde un enfoque orientado a objetos se suelen denominar *métodos*. La estructura de un tipo abstracto de dato (*clase*), desde un punto de vista global, se compone del interfaz y de la implementación (Figura 3.5).

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 esos interfaces son *privados*.

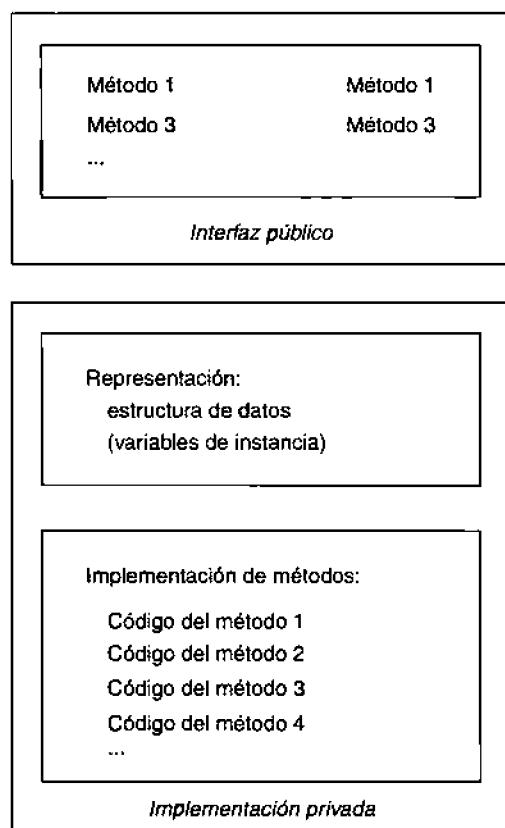


Figura 3.5. Estructura de un tipo abstracto de datos (TAD).

3.7.1. 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. Es similar a la definición que daremos en el capítulo siguiente de objeto y, de hecho, están unidos íntimamente. Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

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

3.7.2. Implementación de los TAD

Los lenguajes convencionales, tales como Pascal, permiten la definición de nuevos tipos y la declaración de procedimientos y funciones para realizar operaciones sobre objetos de los tipos. Sin embargo, tales lenguajes no permiten que los datos y las operaciones asociadas sean declaradas juntas 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 Borland Pascal	<i>unidad, objeto</i>
Modula-2	<i>módulo</i>
Ada	<i>paquete</i>
C++	<i>clase</i>
Java	<i>clase</i>

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.

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*»).

3.8. TIPOS ABSTRACTOS DE DATOS EN TURBO BORLAND PASCAL

Una *pila* es una de las estructuras de datos más utilizadas en el mundo de la compilación. Una *pila* es un tipo de dato clásico utilizado frecuentemente para introducir al concepto de tipo abstracto de datos; es una lista lineal de elementos en la que los elementos se añaden o se quitan por un solo extremo de la lista. La *pila* almacena elementos del mismo tipo y actúan sobre ella las operaciones clásicas de Meter y Sacar elementos en dicha *pila*, teniendo presente la estructura lógica LIFO (*último en entrar, primero en salir*).

En Turbo Borland Pascal, los TAD se implementan mediante estructuras tipo *unidad*. Recordemos que una *unidad* es una biblioteca de funciones y procedimientos que pueden ser utilizados por cualquier programa con la condición de incluir el nombre de la unidad en la cláusula uses de su sintaxis.

```
unit <nombree unidad>;
interface
  <clausula uses>
  <constantes, tipos y variables publicas>
  <cabeceras de procedimientos y funciones publicas>
implementation
  <clausula uses>
  <constantes, tipos y variables privadas>
  <procedimientos/funciones privadas y cuerpos de
    procedimientos/funciones publicas>
begin
  <secuencia de sentencias para inicializacion>
end.
```

Pila

Estructura de datos

Almacena una serie de elementos del tipo *elemento*. La *pila* está inicialmente vacía y los elementos se meten o sacan en la *pila* por el mismo extremo.

Operaciones (Procedimientos que actúan sobre la estructura de datos).

Meter
Sacar
Crear
Destruir
Pilavacia

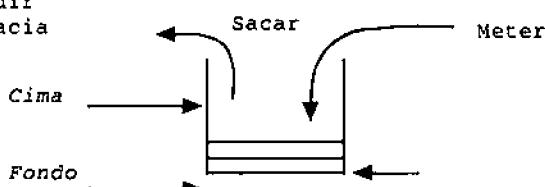


Figura 3.6. Estructura de datos Pila.

La implementación de una *pila* con capacidad para 1.000 elementos del tipo entero es:

```

unit Pila;
interface
  const
    MaxPila = 1000;
  type
    TipoElemento = integer;
    ListaElementos = array [1..MaxPila] of TipoElemento;
    tipo = record
      Elems : ListaElementos;
      Cima : integer;
    end;

  procedure Crear (var S:tipo);
  (* S se inicializa y se limpia o vacía *)

  procedure Destruir (var S:tipo);
  (* Se libera memoria asignada a S *)
  (* S no está inicializada *)

  procedure Meter (var S:tipo; Item: tipoElemento);
  (* Se añade un elemento a la cima de la pila *)

  procedure Sacar (var S:tipo; var Item:tipoElemento);
  (*quitar un elemento de la pila*)

  function PilaVacia (S:tipo):boolean;
  (*devuelve true si S es vacía; false en caso contrario*)

implementation

  procedure Crear (var S:tipo);
  begin
    S.Cima := 0;
  end;

  procedure Destruir (var S:tipo);
  begin
    (*no hace nada*)
  end;

  procedure Meter (var S:tipo; Item:TipoElemento);
  begin
    S.Cima := S.Cima + 1;
    S.Elems[S.Cima] := Item;
  end;

  procedure Sacar (var S:tipo; var Item:TipoElemento);
  begin
    Item := S.Elems[S.Cima];
    S.Cima := S.Cima-1
  end;

```

```

function PilaVacia (S:tipo):boolean;
begin
  PilaVacia := (S.Cima=0);
end;
end.

```

Una vez que se ha implementado el tipo de dato Pila, éste puede ser utilizado en cualquier programa con tal de invocar en la cláusula uses a dicho tipo de dato Pila.

3.8.1. Aplicación del tipo abstracto de dato *pila*

El siguiente programa lee una secuencia de enteros, uno por línea, tales como éstos:

```

100
4567
-20
250

```

y genera la misma secuencia de números en orden inverso (los saca de la pila).

```

250
-20
4567
100

```

Esta tarea se consigue metiendo números en la pila y a continuación vaciando dicha pila.

```

program Numeros;
uses
  Pila;
var
  S:Pila.Tipo;

procedure LeerYAlmacenarNumeros(var NumPila:Pila.Tipo);
  var
    Aux:Pila.TipoElemento;
begin
  while not eof do
  begin
    readln(Aux);
    Pila.Meter(NumPila, Aux);
  end;
end;

procedure VerNumeros(var NumPila: Pila.Tipo);
  var
    Aux: Pila.TipoElemento;
begin
  while not Pila.Pilavacia(NumPila) do

```

```

begin
  Pila.Sacar(NumPila, Aux);
  WriteLn(Aux);
end;      (* while *)
end;      (* VerNumeros *)

begin      (* programa principal *)
  Pila.Crear(S)
  LeerYAlmacenarNumeros(S);
  VerNumeros(S);
  Pila.Destruir(S);
end.      (* fin de principal *)

```

Al igual que se ha definido el tipo *Pila* con estructuras estáticas tipo *array*, se podía haber realizado con estructuras dinámicas tales como listas enlazadas. De igual modo se pueden implementar otros tipos de datos abstractos tales como, por ejemplo, las colas que se utilizan en muchas aplicaciones: sistemas operativos, sistemas de comunicaciones, etc.

3.9. ORIENTACIÓN A OBJETOS

La orientación a objetos puede describirse como el *conjunto de disciplinas (ingeniería) que desarrollan y modelan 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 modelación 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 3.7 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 que codifica el problema y la transformación se 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.

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.

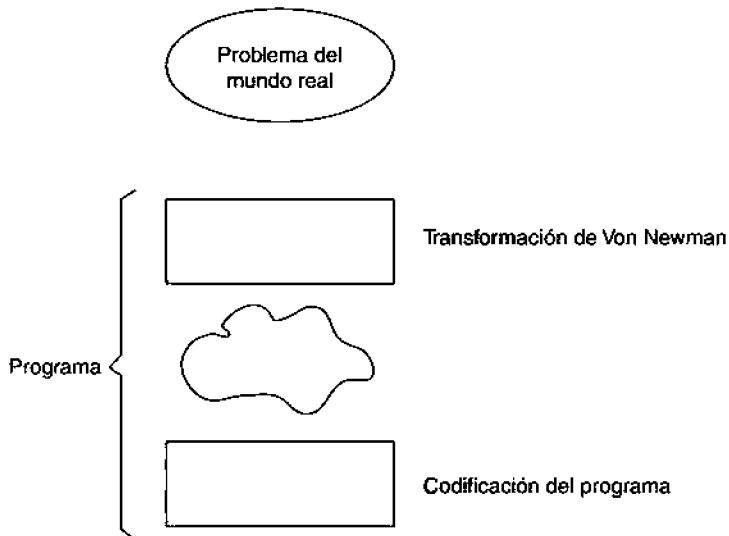


Figura 3.7. Construcción de software.

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 modelo⁶ 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.

⁶ Booch, Grady: *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, 1994.

3.9.1. Abstracción

La *abstracción* es uno de los medios más importante 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.

3.9.2. Encapsulación

La *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.

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*. La *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.

3.9.3. 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 las 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.

3.9.4. 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].

No se debe 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.

3.9.5. 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 bombón o cara-

meo 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 a 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.

3.9.6. Otras propiedades

El modelo objeto ideal no sólo tiene las propiedades anteriormente citadas al principio del apartado, sino que es conveniente 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 LPOO, 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 esta segunda mitad de la década de los noventa.

3.10. 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 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 a objetos proporcionan un mecanismo para construir *componentes de software* reutilizables que posteriormente puedan ser interconectados entre sí y formar grandes proyectos de software⁷.

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: Turbo Visión de Turbo Borland Pascal, OLE 2.0 de C++, jerarquía de clases Smalltalk, clases MacApp para desarrollo de interfaces gráficos de usuario en Object Pascal, disponibles en Apple, la colección de clases de Objective-C, etc.

⁷ 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*.

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* o *navegadores* («*browser*»), para localizar clases de interés y depuradores interactivos para ayudar al programador.

3.11. 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. Asimismo, 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, 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 interactivo de usuario basado en menús.

Entre los lenguajes orientados a objetos que se han desarrollado a partir de los 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 han aparecido 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, seguramente aumentará 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 nueva versión Ada-95 ya soporta herencia y polimorfismo.

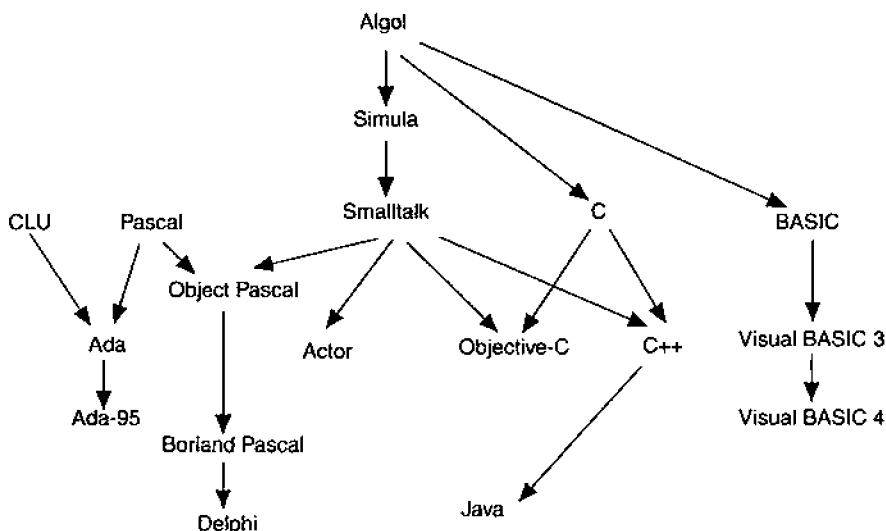


Figura 3.8. Evolución de los lenguajes orientados a objetos.

En los últimos años han aparecido lenguajes con soporte de objetos que cada vez se están popularizando más: Clipper 5.2 (ya en claro desuso), Visual BASIC, etc.

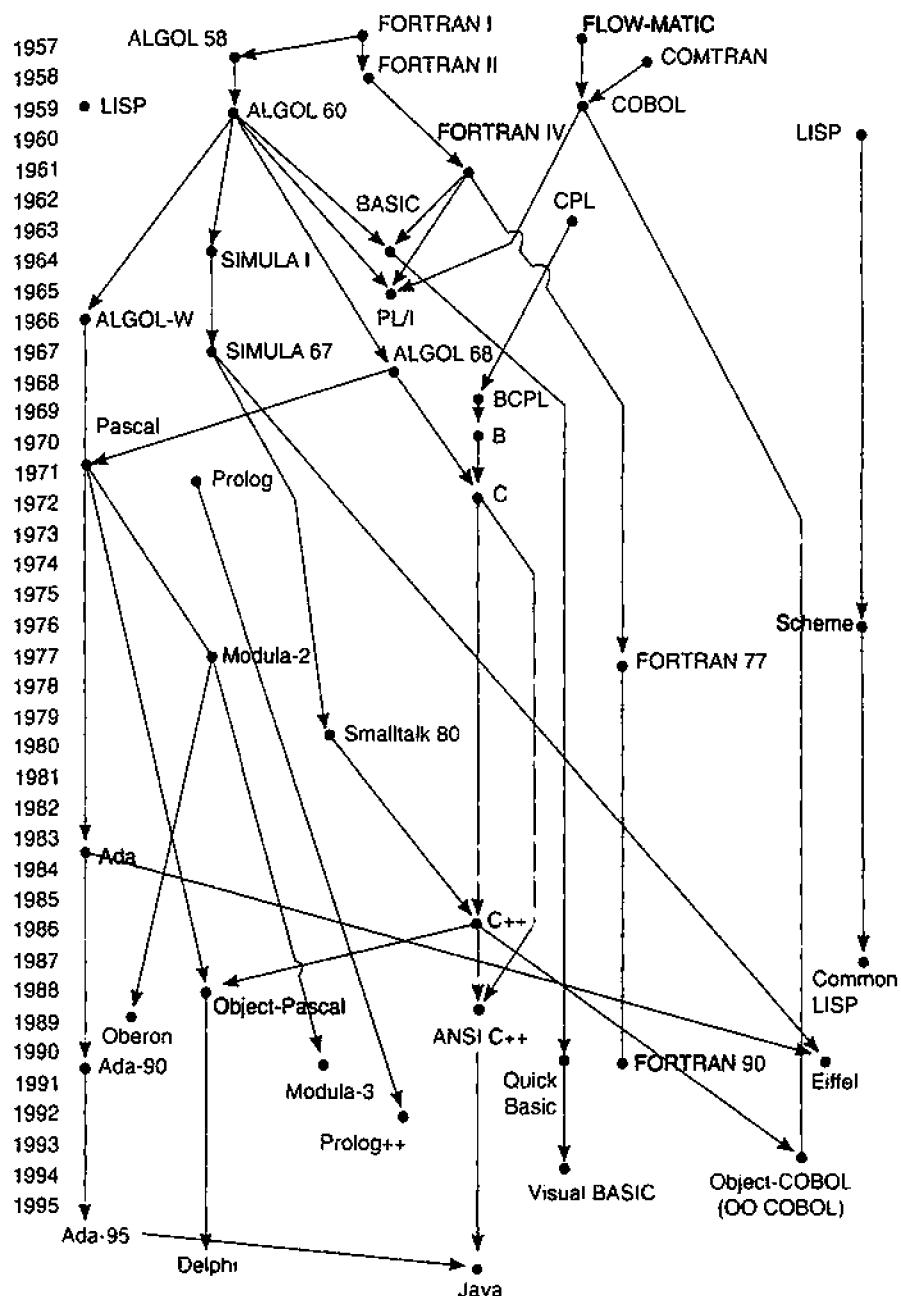
De cualquier forma, existe un *rey* actual en los lenguajes orientados a objetos: C++. La normalización por ANSI y AT&T de la versión 3.0 y las numerosas versiones de diferentes fabricantes, tales como Borland C++ 4.0/4.5/5, Turbo C++ 3.0/3.1 y 4.5, Visual C++ 4.1/4.2/5, Symantec 6.0/7.0, etc., hacen que en la segunda mitad de los noventa, C++ será el lenguaje orientado a objetos más popular y utilizado en el mundo de la programación. A C++ se le ha unido recientemente Java, un lenguaje evolución de C++, creado específicamente para programar en Internet pero que puede ser utilizado también en aplicaciones de propósito general.

La evolución de los lenguajes orientados a objetos se muestra en la Figura 3.8, 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).

3.11.1. Clasificación de los lenguajes orientados a objetos

Existen varias clasificaciones de lenguajes de programación orientados a objetos, atendiendo a criterios de construcción o características específicas de los mismos. Una clasificación ampliamente aceptada y difundida es la dada por Wegner y que se ilustra en la Figura 3.10⁸.

⁸ Wegner, Peter [1987]: *Dimensions of Object-Based Languages Design*. Número especial de SIGPLAN Notices, 1987.

Figura 3.9. Genealogía de los lenguajes de objetos según Sebesta⁹.

⁹ Esta tecnología ha sido extraída y modificada ligeramente con lenguajes de los noventa de Robert W. Sebesta: *Concepts of Programming Languages*, Addison-Wesley, 2.^a ed., 1993.

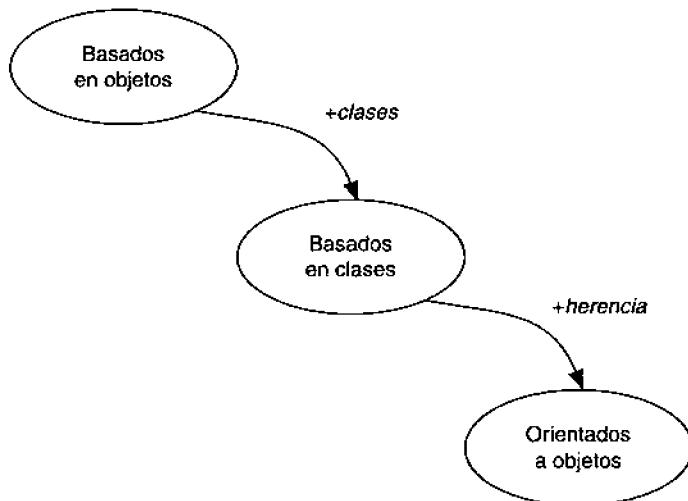


Figura 3.10. Clasificación de lenguajes OO de Wegner.

La clasificación de Wegner divide los lenguajes en tres categorías:

1. Lenguajes **basados en objetos** que soportan objetos. Es decir, disponen de componentes caracterizados por un conjunto de operaciones (comportamiento) y un estado.
2. Lenguajes **basados en clases** que implican objetos y clases. Es decir, disponen de componentes tipo clase con operaciones y estado común. Una clase de un objeto se construye con una «*interfaz*» que especifica las operaciones posibles y un «*cuerpo*» que implementa dichas operaciones.
3. Lenguajes **orientados a objetos** que además de objetos y clases ofrecen mecanismos de herencia entre clases. Esto es, la posibilidad de derivar operaciones y atributos de una clase (superclase) a sus subclases.

La definición anterior, pese a su antigüedad sigue teniendo vigencia. Existen otras clasificaciones similares, pero con la inclusión de la propiedad de *polimorfismo* en la categoría 3, como requisito para ser lenguaje orientado a objetos.

De cualquier forma, hoy día es posible ampliar esa clasificación de acuerdo a criterios puramente técnicos y hacer una nueva clasificación de la categoría 3:

- 3.1. *Lenguajes orientados a objetos puros*. Soportan en su totalidad el paradigma de orientación a objetos:

Smalltalk Eiffel Simula

- 3.2. *Lenguajes orientados a objetos híbridos.* Soportan en su totalidad el paradigma de orientación a objetos sobre un núcleo de lenguaje híbrido:

C++ (*extensión* de C: Borland C++, Microsoft C++, Turbo C++, Visual C++, Symantec, Watcom...).
 Objective-C (*extensión* de C).
 Object COBOL (*extensión* de COBOL).
 Object Pascal (*extensión de Pascal*: Turbo Borland Pascal).
 Visual Object (*extensión* de Clipper).
 Delphi (*extensión* de Turbo Borland Pascal 7.0).
 Java (*extensión mejorada* de C++).
 Ada-95 (*extensión* de Ada-83).

De cualquier forma, Meyer, creador del lenguaje Eiffel, proporciona unos criterios para considerar la «bondad»¹⁰ de un lenguaje orientado a objetos, cuyos complementos configuran, *de hecho*, una nueva clasificación. En este sentido, los criterios recogidos en [Meyer 88] son los siguientes:

1. La modularización de los sistemas ha de realizarse mediante estructuras de datos apropiadas.
2. Los objetos se describen como la implementación de tipos abstractos de datos.
3. La memoria se gestiona (administra) automáticamente.
4. Existe una correspondencia entre tipos de datos no elementales y clases.
5. Las clases se pueden definir como extensiones o restricciones de otras clases ya existentes mediante herencia.
6. Soportan polimorfismo y ligadura dinámica.
7. Existe herencia múltiple y repetida.

De acuerdo con los criterios de Meyer recogemos en la Tabla 3.3 el cumplimiento de dichos criterios en los lenguajes OO y basados en objetos más populares.

Tabla 3.3. Criterios de Meyer en lenguajes OO y basados en objetos

Criterios	Ada 83	Ada 95	C++	Eiffel	Smalltalk
1. Modularización	Sí	Sí	Sí	Sí	Sí
2. Tipos abstractos de datos	Sí	Sí	Sí	Sí	Sí
3. Gestión automática de memoria	Sí	Sí	<i>En parte</i>	Sí	Sí
4. Sólo clases	Sí	Sí	<i>En parte</i>	Sí	Sí
5. Herencia	No	Sí	Sí	Sí	Sí
6. Polimorfismo (ligadura dinámica)	No	Sí	Sí	Sí	Sí
7. Herencia múltiple y repetida	No	No	Sí	Sí	No

¹⁰ Meyer, Bertrand: *Object Oriented Software Construction*, Englewood Cliffs NJ, Prentice-Hall, 1988.

3.12. DESARROLLO TRADICIONAL FRENTE A DESARROLLO 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 esta sección 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:

$$\text{Datos} + \text{Algoritmos} = \text{Programas}$$

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 compartmentaliza 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 *sólo en datos* o *sólo en procedimientos* es similar a un lenguaje (idónea) 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 les 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.

3.13. BENEFICIOS DE LAS TECNOLOGÍAS DE OBJETOS

Una pregunta que hoy día se hacen muchos informáticos es: *¿Cuál es la razón para introducir métodos de TO en los procesos de desarrollo?* La principal razón, sin lugar a dudas, son los beneficios de dichas TO: aumento de la fiabilidad y productividad del desarrollador. La fiabilidad se puede mejorar debido a que cada objeto es simplemente «una caja negra» con respecto a objetos externos con los que debe comunicarse. Las estructuras de datos internos y métodos se pueden refinar sin afectar a otras partes de un sistema (Figura 3.11).

Los sistemas tradicionales, por otra parte, presentan, con frecuencia, efectos laterales no deseados. Las tecnologías de objetos ayudan a los desarrolladores a tratar la complejidad en el desarrollo del sistema.

La productividad del desarrollador se puede mejorar debido a que las clases de objetos se pueden hacer reutilizables de modo que en cada subclase o instancia de un objeto se puede utilizar el mismo código de programa para la clase. Por otra parte, esta productividad también aumenta debido a que existe una asociación más natural entre objetos del sistema y objetos del mundo real.

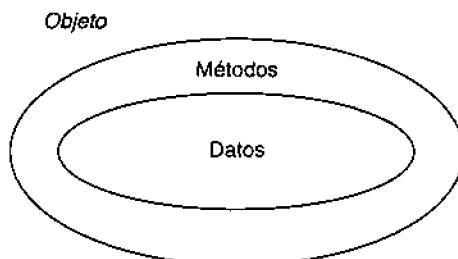


Figura 3.11. El objeto como *caja negra*.

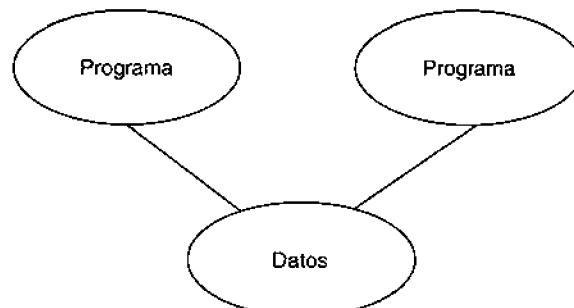


Figura 3.12. Proceso tradicional de datos.

Taylor¹¹ considera que los beneficios del modelado y desarrollo de objetos son:

1. Desarrollo más rápido.
2. Calidad más alta.
3. Mantenimiento más fácil.
4. Coste reducido.
5. Incremento en escalabilidad.
6. Mejores estructuras de información.
7. Incremento de adaptabilidad.

Sin embargo, Taylor¹² también considera algunos inconvenientes, aunque algunos de ellos ya han sido superados o al menos reducido su impacto.

1. Inmadurez de la tecnología (hoy día ya no se puede considerar así).
2. Necesidades de estándares (el grupo OMG es una realidad).
3. Necesidad de mejores herramientas.
4. Velocidad de ejecución.
5. Disponibilidad de personal cualificado.
6. Coste de conversión.
7. Soporte para modularidad a gran escala.

La Figura 3.13 muestra los beneficios genéricos de las tecnologías de objetos.

- *Reutilización.* Las clases se construyen a partir de otras clases.
- *Sistemas más fiables.*
- *Proceso de desarrollo más rápido.*
- *Desarrollo más flexible.*
- *Modelos que reflejan mejor la realidad.*
- *Mejor independencia e interoperatividad de la tecnología.*
- *Mejor informática distribuida y cliente-servidor.*
- *Bibliotecas de clases comerciales disponibles.*
- *Mejor relación con los clientes.*
- *Mejora la calidad del producto software terminado.*

Figura 3.13. Beneficios de las tecnologías de objetos.

¹¹ Taylor, David A.: *Object-Oriented Technology Reading*, MA: Addison-Wesley, 1992, páginas 103-107.

¹² Ibid., págs. 108-113.

RESUMEN

Este capítulo examina el concepto fundamental de la orientación a objetos, el tipo abstracto de datos. 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.

En la mayoría de los lenguajes de programación orientados a objetos, los tipos abstractos de datos se implementan mediante *clases* (*unidades* en Pascal, *módulos* en Modula-2, *paquetes* en Ada).

En este capítulo se considera también una introducción a los métodos de desarrollo orientados a objetos. Se comienza con una breve revisión de los problemas encontrados en el desarrollo tradicional de software, que condujeron a la crisis del software y que se han mantenido hasta los años actuales. El nuevo modelo de programación se apoya esencialmente en el concepto de objetos.

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.

Los conceptos de la programación orientada a objetos se examinan en el capítulo; si no ha leído hasta ahora nada sobre tecnologías de objetos, deberá examinar con detenimiento todos los elementos conceptuales del capítulo.

EJERCICIOS

- 3.1. Construir un tipo abstracto lista enlazada de nodos que contienen enteros.
- 3.2. Diseñar un tipo abstracto de datos pila de números enteros y que al menos soporte las siguientes operaciones:

<i>Borrar:</i>	Eliminar todos los números de la pila.
<i>Copiar:</i>	Hace una copia de la pila actual.
<i>Meter:</i>	Añadir un nuevo elemento en la cima de la pila.
<i>Sacar:</i>	Quitar un elemento de la pila.
<i>Longitud:</i>	Devuelve un número natural igual al número de objetos de la pila.
<i>Llena:</i>	Devuelve <i>verdadero</i> si la pila está llena (no existe espacio libre en la pila).
<i>Vacia:</i>	Devuelve <i>verdadero</i> si la pila está vacía y <i>falso</i> en caso contrario.
<i>Igual:</i>	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; <i>falso</i> en caso contrario.

- 3.3. Crear un tipo abstracto Cola que sirva para implementar una estructura de datos cola.
- 3.4. Crear un TAD para representar:
 - Un vector (representación gráfica y operaciones).
 - Una matriz y sus diferentes operaciones.
 - Un número complejo y sus diferentes operaciones.
- 3.5. Crear un TAD que represente un dato tipo cadena (*string*) y sus diversas operaciones: cálculo, longitud, buscar posición de un carácter dado, concatenar cadenas, extraer una subcadena, etc.

*Fundamentos básicos
de estructuras de datos
y tipos abstractos de datos*

Estructuras de datos dinámicas: punteros

CONTENIDO

- 4.1. Estructuras de datos dinámicas.
- 4.2. Punteros (*apuntadores*).
- 4.3. Operaciones con variables puntero: los procedimientos new y dispose.
- 4.4. El tipo genérico puntero (*pointer*).
- 4.5. La asignación de memoria en Turbo Pascal.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Los punteros son el último de los tipos de datos incorporados a Pascal. *Un puntero* es un tipo de dato simple que contiene la dirección de una variable o estructura en vez de un valor de dato. Los punteros tienen dos propósitos principales: hacer los programas más eficientes y construir estructuras muy complejas. En el capítulo se examina también la gestión dinámica de memoria y el modo en que se puede manipular. Se verá cómo utilizar variables dinámicas para construir estructuras de datos que pueden crecer y disminuir conforme se ejecuta el programa.

Las estructuras de datos dinámicas son una colección de elementos (denominados *nodos*) que son nodos, en contraste a un array que siempre contiene almacenamiento para un número fijo de elementos.

4.1. ESTRUCTURAS DE DATOS DINÁMICAS

Todas las variables y estructuras de datos que se han considerado hasta este momento han sido estáticas. Con una variable *estática*, la cantidad de memoria (*espacio*) ocupada debe ser declarado por anticipado y no puede ser incrementado durante la ejecución del programa si se necesitara más espacio de memoria.

Un array de requisitos es estático dado que la cantidad exacta de memoria se fija por la declaración del tamaño del array. Esta falta de flexibilidad puede ser una desventaja notable. Así, por ejemplo, si en un array de registros se declara para un tamaño máximo de 1.000 registros, el programa no funcionará si se deben almacenar más de 1.000 registros en ese array. Por otra parte, si el tamaño máximo declarado para un array es mucho mayor que el total de espacio de memoria requerido, el programa utilizará ineficientemente la memoria, dado que la cantidad de memoria especificada en la declaración se reservará, incluso aunque sólo se utilice una pequeña parte.

Los *punteros* (apuntadores)¹ permiten la creación de estructuras de datos *dinámicas*: estructuras de datos que tienen capacidad de variar en tamaño y ocupar tanta memoria como utilicen realmente. Las variables que se crean y se destruyen durante la ejecución se llaman *variables dinámicas* (también *anónimas*). Así, durante la ejecución de un programa, puede haber una posición de memoria específica asociada con una variable dinámica y posteriormente puede no existir ninguna posición de memoria asociada con ella.

Pascal proporciona los métodos para asignar y liberar espacio de memoria utilizando *punteros* y los procedimientos predefinidos *new* y *dispose*.

Al contrario que las *estructuras de datos estáticas*, tales como arrays cuyos tamaños y posiciones de memoria asociados se fijan en tiempo de compilación, las *estructuras dinámicas de datos* se amplían (expanden) o reducen (contraen) a medida que se requiera durante la ejecución y cambia sus posiciones de memoria asociada.

Una estructura de datos dinámica es una colección de elementos llamados *nodos* de la estructura —normalmente de tipo registro— que se enlazan o encadenan juntos. Este enlace se establece asociando con cada nodo un puntero que apunta al nodo siguiente de la estructura.

Existen diferentes tipos de estructuras dinámicas de datos, siendo las más notables y significativas las *listas enlazadas*, *los árboles* y *los grafos*.

Las estructuras de datos dinámicas son útiles especialmente para almacenar y procesar conjuntos de datos cuyos tamaños cambian durante la ejecución del programa, por ejemplo, el conjunto de trabajos que se han introducido en una computadora y están esperando su ejecución o el conjunto de nombres de pasajeros y asignación respectiva de asientos de un vuelo de avión determinado. En este capítulo se estudiarán los punteros y los procedimientos *new* y *dispose*, así como el método que se puede utilizar

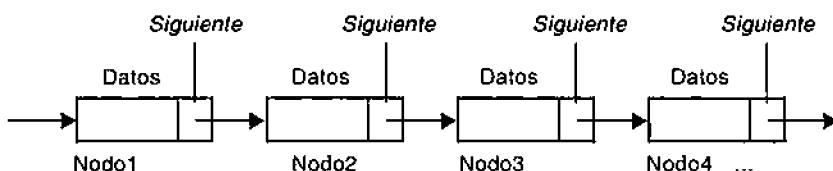


Figura 4.1. Representación de una lista enlazada.

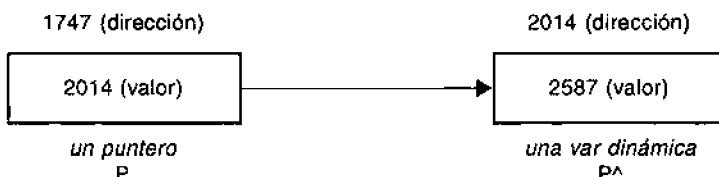
¹ En Latinoamérica, el término utilizado para definir este concepto es *apuntador*.

para construir y procesar estructuras dinámicas de datos, junto con la estructura dinámica de datos, por excelencia, la *lista enlazada*.

4.2. PUNTEROS (APUNTADORES)

En una computadora cada posición de memoria tiene una dirección y un valor específico almacenado en esa posición. Se han utilizado nombres de variables en lugar de direcciones porque los nombres son más fáciles de recordar. Para almacenar un nuevo valor en memoria se asigna a una variable, y la computadora envía una dirección a la memoria seguida por el valor a almacenar en esa posición.

El tipo *puntero* es un tipo de datos simple en Pascal; es simple debido a que no se puede romper en otros componentes pequeños como es el caso de un array o un registro. Los punteros en esencia son un *tipo especial de variable* (estática) que se utiliza para almacenar la dirección de memoria de otra variable, o lo que es igual, su valor es una dirección de una posición de memoria donde está almacenada otra variable. Las variables utilizadas para almacenar direcciones en vez de valores convencionales se denominan *variables puntero* o simplemente *puntero* (apuntador).



Al definir un tipo puntero se debe indicar el tipo de valores que se almacenarán en las posiciones designadas por los punteros. La razón es que los diferentes tipos de datos requieren diferentes cantidades de memoria para almacenar sus constantes, una variable puntero puede contener una dirección de una posición de memoria adecuada sólo para un tipo dado. Por esta razón se dice que un puntero *apunta* a una variable particular, es decir, a otra posición de memoria.

Una variable tipo puntero contiene la dirección de la posición de otra variable.

4.2.1. Declaración de punteros

Un puntero es una variable que se utiliza para almacenar la dirección de otra variable. Sin embargo, el puntero en sí no es normalmente lo que le interesa más; sino que el interés reside en la celda (dirección de memoria) apuntada por él. Es decir, es preciso diferenciar entre las dos entidades implicadas en el apuntamiento: *la variable puntero* (quién hace el apuntamiento) y la *variable apuntada* (a quién se apunta).

Un tipo de dato puntero se especifica utilizando el símbolo de circunflejo (^) seguido por un identificador de tipo. Su formato es:

```
^identificador-tipo
```

Se puede declarar un puntero a una variable carácter, un puntero a un array de enteros, un puntero a un registro o un puntero a cualquier otro tipo. En general, se pueden declarar variables puntero que apunten a cualquier tipo de dato, incluso otros punteros. La declaración de una variable a un puntero se realiza con el formato siguiente:

```
var
  Nombre-variable: ^identificador-tipo
```

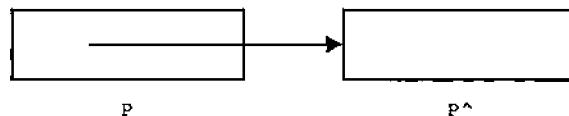
```
var
  Ptr: ^TipoElemento
```

La variable `Ptr` apunta a un tipo de dato `TipoElemento`

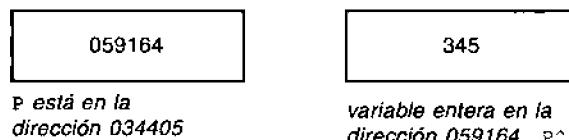
EJEMPLO 4.1

```
var
  P: ^Integer;
```

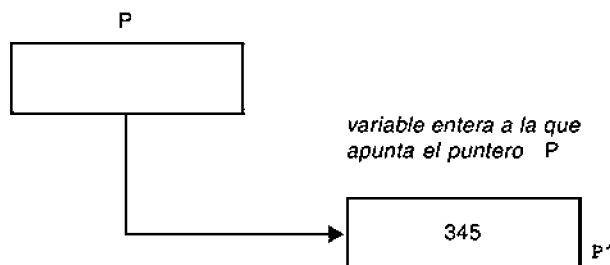
Los punteros se pueden representar mediante diagramas gráficos:



Por ejemplo, la variable `P` contiene 059164 que es la dirección de memoria donde está la variable entera apuntada 345.



Un puntero *apunta hacia* (*a*) otra variable cuando contiene la dirección de esa variable.



Es muy útil crear un identificador de tipo para los tipos de datos puntero. Se puede crear dicho identificador y utilizar ese identificador para crear variables punteros.

```
type
  nombre-tipo: ^identificador-tipo;
var
  nombre-var-ptr: nombre-tipo;
```

EJEMPLO 4.2

Supongamos que se desean almacenar números reales utilizando aplicación de memoria dinámica. Las variables A y B son variables punteros que apuntan a datos reales.

```
type
  PunteroReal = ^Real;                                real
var
  A, B: PunteroReal;                                 A → [ ] A^
                                                       B → [ ] B^
                                                       real
1. type
   Tipo_Puntero = ^integer;
var
  P: Tipo_Puntero;
2. var
  P: ^Integer;
3. type cad40 = string[40];    declara tres variables puntero:
  var ptr: ^cad40;           ptr apunta a valores cadena
  q1, q2: ^real;             q1, q2 apuntan a valores reales
4. type
   PunteroReal = ^Real;      puntero a reales
var
  P: PunteroReal;
  R: Real;                  P es una variable puntero de tipo
                           PunteroReal
type
  PunteroEnt = ^integer;
```

```

5. var
   Primero: ^Real;
   Sig: PunteroEnt;

```

4.3. OPERACIONES CON VARIABLES PUNTERO: LOS PROCEDIMIENTOS *NEW* Y *DISPOSE*

Los punteros se crean con las declaraciones citadas:

```

type
  PunteroReal = ^Real;
var
  P: PunteroReal;

```

P es una variable puntero de tipo PunteroReal que apunta a posiciones que contienen reales. La posición de memoria designada por el valor de la variable puntero P se representa por $P^$. La Figura 4.2 representa la relación entre P y $P^$.

Como $P^$ designa una posición de memoria, se puede utilizar como cualquier otra variable Pascal. Se pueden asignar valores a $P^$ y utilizar valores de $P^$ en expresiones tal como cualquier otra variable. Si P apunta a posiciones que contienen reales, $P^$ es una variable real. Así, en la Figura 4.2 el valor de $P^$ es 3.500.

Sin embargo, estas operaciones no se pueden realizar directamente tras la declaración, debido a que el objeto o dirección apuntada $P^$ no tiene existencia. Antes de que un programa utilice un puntero, se requiere primero espacio para el tipo de datos objeto de la dirección del puntero. En otras palabras, un programa debe inicializar sus punteros —su declaración no basta—; para iniciar un puntero se debe utilizar el procedimiento New.

4.3.1. Procedimiento *New*

La declaración de una variable puntero P no crea una celda de memoria para apuntar a ella. El procedimiento (sentencia) New se utiliza para crear tal celda de memoria P; es decir el procedimiento New crea una nueva variable dinámica y establece que una variable puntero apunte a ella.

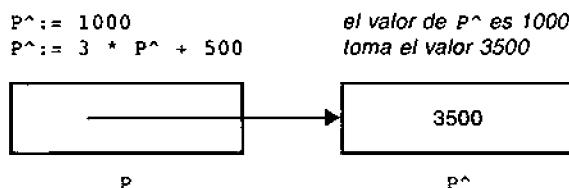
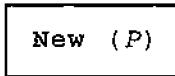


Figura 4.2. Diferencia entre P y $P^$.

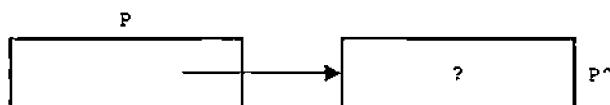
Formato


p variable puntero

La sentencia **New (P)**; llama al procedimiento **New** que asigna almacenamiento para un valor del tipo determinado y sitúa la dirección de esta celda de memoria en la variable puntero **P**. Una vez que se asigna almacenamiento para un valor de tipo determinado al que está apuntando **P**, se puede almacenar un valor en esa celda de memoria y manipularlo. La posición exacta de la memoria de esta celda particular es indiferente.

La nueva variable creada es la ya conocida con el símbolo **P^**. La llamada a **New** exige que exista suficiente espacio de memoria libre en el *montículo* —la pila de variables dinámicas— (*heap*) para asignar a la nueva variable. En caso contrario se producirá un error en tiempo de ejecución.

Se puede representar el valor de una variable puntero por una flecha dirigida a una celda de memoria. El diagrama



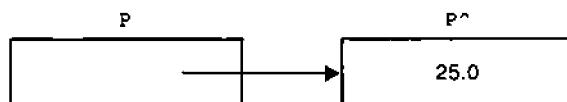
muestra que la variable puntero **P** apunta a una celda de memoria cuyo contenido se desconoce. Esta es la sentencia que existe inmediatamente después de que se ejecuta **New (P)**.

El símbolo **P^** se utiliza para referenciar a la celda de memoria apuntada por la variable puntero **P**. El símbolo **^** (circunflejo) se denomina *operador de indirección o de desreferencia*.

La sentencia de asignación

P^ := 25.0

almacena el valor real 25.0 en la posición (celda) de memoria apuntada por **P**.



Como **P^** se considera una variable, podrá ser visualizado su valor

Write (P^:10:2); *visualiza el valor 25.0*

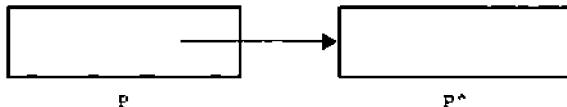
Atención

<code>P := 25.0;</code>	<i>asignación no válida</i>
<code>Write (P:10:2);</code>	<i>sentencia no válida</i>

Ambas sentencias son *no válidas* debido a que no se puede almacenar un valor de tipo Real en una variable puntero P, ni se puede visualizar el valor (una dirección) de una variable puntero.

Funcionamiento de New (P)

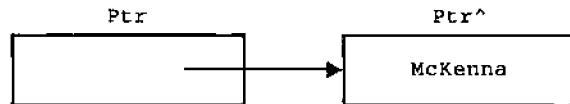
1. Se asigna espacio de memoria conveniente para almacenar tipos de datos correspondientes.
2. La dirección de esta posición de memoria se almacena en P; se crea una *variable dinámica* P^* .

**EJEMPLO 4.3**

A la posición o celda de memoria a que apunta un puntero se puede acceder situando un símbolo ^ después del nombre de la variable puntero. Por ejemplo,

```
new (Ptr);
Ptr^ := 'McKenna';
```

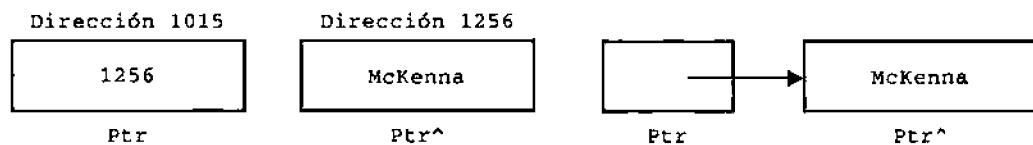
La sentencia `new (Ptr)` crea una celda de memoria vacía (o con basura) a la que apunta `Ptr`. La segunda sentencia sitúa el valor 'McKenna' en esa celda de memoria. Un diagrama gráfico sería:

**Implementación interna**

Internamente, los punteros se implementan teniendo en cuenta las direcciones de memoria a las que se apunta. Por ejemplo, supongamos que la dirección de `ptr` es 1015 y que la celda de memoria que crea `new(ptr)` está en la dirección 1256. Por consiguiente, el efecto de

```
new (ptr);
ptr^:= 'McKenna';
```

se puede dibujar a nivel de direcciones de memoria, como se muestra a la izquierda, o con el diagrama más sencillo de la derecha.



Atención

Observe que en la declaración de un puntero, el circunflejo está situado a la izquierda de su tipo, mientras que en una sentencia en que se accede a la celda de memoria a que se está apuntando, el circunflejo aparece a la derecha.

EJEMPLO 4.4

Realizar el seguimiento del segmento de programa.

```
var
  1. Ptr1: ^char;
  2. Ptr2: ^integer;
begin
  3. New (Ptr1);
  4. Ptr1^:= 'B';
  5. New (Ptr2);
  6. Ptr2^:= 86;
```

EJEMPLO 4.5

Deducir la salida del siguiente programa.

```
program Puntero;
type
  Print = ^integer;
var
  I, J: Print;
  N: integer;
begin
  New (I);
  New (J);
  N:= 5;
  I^:= N;
  writeln (I^);
  J:= I;
  J^:= -7;
  writeln (J^)
end.
```

EJEMPLO 4.6

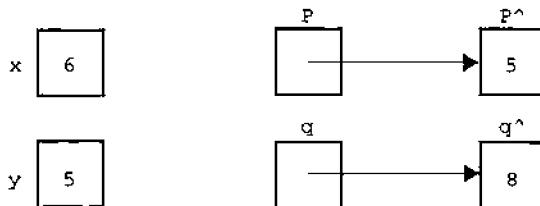
¿Qué se imprime tras ejecutar el siguiente programa?

```
program PruebaSimple;
var p, q: ^integer;
    x, y: integer;
begin
  new(p);
  p^:= 5;
  x:= p^ + 1;
  y:= p^;
  new (q);
  q^:= y + 3;
  writeln (x, ', ', y, ', ', p^, ', ', q^)
end.
```

Solución

6 5 5 8

La distribución gráfica es:

**4.3.2. Variables de puntero con registros**

Las variables de puntero pueden contener la dirección de tipos de datos simples o datos de tipo compuesto. En las estructuras dinámicas de datos es frecuente el uso de registros como elementos. Se suelen conocer con el nombre de *nodos*, aunque en realidad el término se extiende tanto a datos simples como compuestos. Por su importancia, y aunque la manipulación es idéntica a la ya estudiada, consideraremos un ejemplo con variables dinámicas tipo registro.

EJEMPLO 4.7

¿Cuál es la salida de este programa?

```
program Test;
type
  Estudiante = record
    Letra: char;
    Edad: integer
  end;
```

```

PuntEstu = ^ Estudiante;
var
  P1, P2: PuntEstu;
begin
  New (P1);
  P1^.Edad := 1;
  P1^.Letra := 'A';
  WriteLn (P1^.Edad, P1^.Letra);
  New (P2);
  P2^.Edad := 2;
  P2^.Letra := 'B';
  WriteLn (P2^.Edad, P2^.Letra);
  P1 := P2;
  P2^.Edad := 3;
  P2^.Letra := 'C';
  WriteLn (P1^.Edad, P1^.Letra, P2^.Edad, P2^.Letra)
end.

```

Ejecución

1A
2B
3C3C

4.3.3. Iniciación y asignación de punteros

Antes de intentar referirse a valores de variables es preciso, como siempre, inicializar las variables. Los pasos a dar se muestran en las siguientes sentencias:

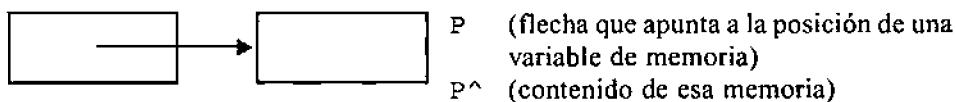
```

var
  P : ^integer;
  .
  .
  .
New (P);
  .
  .
  .
P^ := 17;

```

En un puntero (P) se deben considerar dos valores:

- *Puntero (P)* Su valor es la posición (dirección) de una variable *integer*.
- *P[^]* El valor de la variable *integer* propiamente dicho.



El procedimiento *new* inicializa el puntero con la dirección de una variable *integer*. La variable *integer*, *P[^]*, a la que *P* apunta debe también ser inicializada (*P[^] := 17*).

Recuerde

`Write (P^)` sentencia válida
`Write (P)` sentencia no válida (error)

Asignación

Las variables dinámicas pueden tener más de un puntero que esté apuntando a ellas. También los punteros pueden ser modificados de modo que apunten a variables dinámicas diferentes en momentos diferentes.

Estos cambios se pueden realizar con el operador de asignación (`:=`). Estudiemos las asignaciones posibles.

$$(1) \quad \boxed{p := q}$$

$$(2) \quad \boxed{p^ := q^}$$

- (1) Los punteros p y q apuntan a la misma posición; por consiguiente, $p^$ y $q^$ designan la misma posición y tienen el mismo valor.
- (2) El valor de $q^$ se asigna a $p^$; después de la asignación, $p^$ y $q^$ tienen el mismo valor, sin embargo, $p^$ y $q^$ permanecen en distintas posiciones de memoria.

Advertencia

No se puede asignar un puntero de un tipo que apunte a un puntero de un tipo diferente.

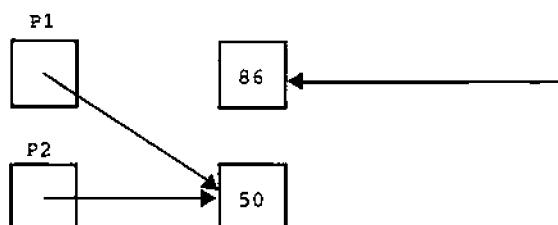
EJEMPLO 4.8

P1 y P2 son punteros de tipo entero. ¿Qué se imprimirá tras la ejecución del siguiente segmento de programa? y ¿cómo será la distribución en memoria?

```
new (p1);
new (p2);
P1^ := 86;           No se puede acceder a este valor
P2^ := 50;
P1 := P2;
WriteLn (P1^, ',', P2^);
```

Solución

50 50



4.3.4. Procedimiento Dispose

El procedimiento `dispose` libera la posición de memoria ocupada por una variable dinámica.

Formato

```
dispose(P)
```

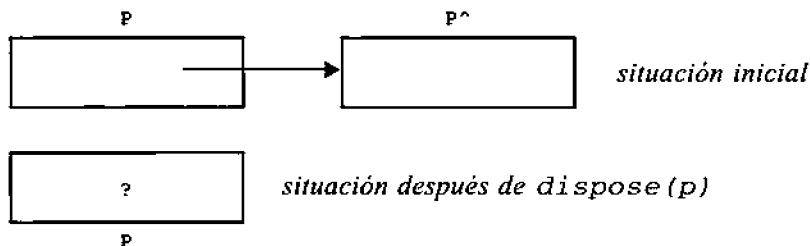
P variable puntero

`Dispose` destruye la variable referenciada por *P* y devuelve su zona de memoria a la pila dinámica (*heap*). Después de una llamada a `dispose`, el valor de *P* se vuelve indefinido y cualquier intento de referenciar a *P*[^] producirá un error.

En resumen, `Dispose` produce las siguientes acciones:

1. La posición de memoria cuya dirección se almacena en *P* se libera de la memoria montículo (*heap*). En otras palabras, *P*[^] deja de existir.
2. *P* se vuelve indefinido.

La representación gráfica es:



EJEMPLO 4.9

```

1. var
   P, Q: ^integer;
begin
2.   New (P);
3.   P^ := 8;
4.   Q := P;
5.   Q^ := 5;
6.   Dispose (P)
end

```

4.3.5. Constante *nil*

Pascal proporciona una constante predefinida, *nil* (nulo). La constante *nil* se utiliza para dar un valor a una variable puntero que no apunta a ninguna posición, *nil* puede ser asignada a un puntero de cualquier tipo.

p := nil

Como *nil* no apunta a ninguna posición de memoria, una referencia a *p*[^] es ilegal si el valor de *p* es *nil*.

```
p := nil;
p^ := 5;      es ilegal ya que se instruye a la computadora a almacenar 5 en la
               posición apuntada por p, pero debido a la primera sentencia (p:=nil),
               p no apunta a ninguna dirección
```

EJEMPLO 4.10

El segmento de programa

```
var
  p: ^char;
begin
  p:= nil;
  if p = nil
    then
      WriteLn ('el puntero P no apunta a nada')
    else
      WriteLn ('P se queda indefinido apunta');
```

proporciona la salida

el puntero P no apunta a nada

Un puntero puede que no apunte a ninguna posición: *nil*.

4.3.6. Naturaleza dinámica de los punteros

Las variables a las que se apuntan no están declaradas. Únicamente las variables a las que se apunta se declaran. Así, una sentencia como:

```
New(P);
```

se puede ejecutar muchas veces dentro de un programa simple; cada vez que se ejecuta, crea una nueva celda de memoria. En consecuencia no existe límite declarado para saber cuántas celdas de memoria se pueden crear utilizando unos pocos punteros.

Los arrays se dice que son *estáticos* debido a que la cantidad máxima de espacio de memoria que se puede utilizar debe ser declarado por anticipado. Los punteros son *dinámicos* ya que no hay tal límite previo, el espacio de memoria se crea durante la ejecución del programa.

4.3.7. Comparación de punteros

Los punteros pueden ser comparados *sólo* en expresiones de igualdad. Así, en el caso de

```
var
  PunteroA, PunteroB: ^integer;
```

las comparaciones:

```
if (PunteroA = PunteroB)
  then < sentencia >;
if (PunteroA <> PunteroB)
  then < sentencia >;
```

No se puede comparar la ordenación de punteros con los operadores de relación <, >, <= y >=.

No existe ningún problema para comparar los objetos direccionados por los punteros. La línea siguiente es válida.

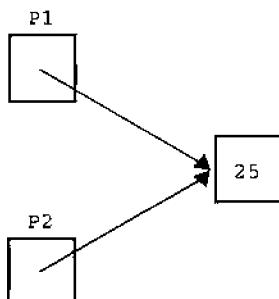
```
if (PunteroA^ <= PunteroB^)
  then < sentencia >
```

Si dos punteros P1 y P2 son del mismo tipo, el efecto de

```
P1 := P2;
```

redirige P1 de modo que apunta a la celda de memoria apuntada por P2. Después de la ejecución de P1 := P2, ambos punteros apuntan a la misma celda.

Se dice que dos punteros son iguales si ambos apuntan precisamente al mismo elemento de datos. En un diagrama ambos apuntan a la misma caja.



Advertencia

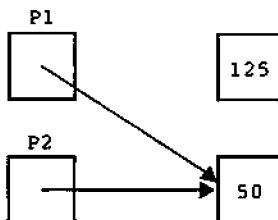
No se puede asignar un puntero de un tipo para apuntar a un puntero de un tipo diferente.

EJEMPLO 4.11

P1 y P2 son punteros de tipo integer. ¿Cuál es la salida del siguiente segmento de programa?

```
new (P1);
new (P2);
P1^:= 125;
P2^:= 50;
P1:= P2;
WriteLn (P1^, ' ', P2^);
```

Después de la ejecución del programa, un plano de memoria podría ser éste:



La salida será

50 50

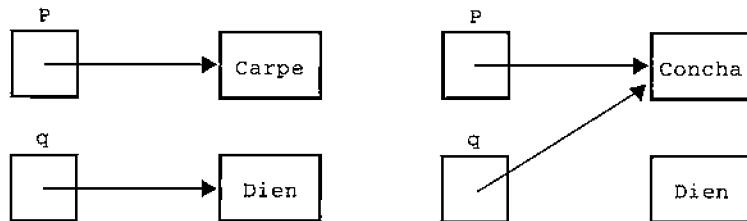
Obsérvese que el valor 125 no se puede acceder. Al valor 50 puede accederse con P1 o con P2.

EJEMPLO 4.12

Suponga que p y q son ambos de tipo cadena (^string). ¿Qué se imprimirá al ejecutar este segmento de programa?

```
new (p);
new (q);
p^ := 'Carpe';
q^ := 'Dien';
q := p;
q^ := 'Concha';
WriteLn (p^); -
WriteLn(q^);
```

La traza de memoria es



La salida será

Concha
Concha

4.3.8. Paso de punteros como parámetros

El paso de parámetros es, como conoce el lector, una técnica poderosísima para procesar información. El paso de punteros se realiza como cualquier otro tipo de dato, con la salvedad de que cuando se pasa un puntero como un parámetro, se está pasando el puntero, no el objeto al que apunta. En el caso de una lista enlazada, el paso de la cabeza o cabecera de la lista no pasa la lista en sí, sólo el puntero a su primer elemento.

El caso más sencillo es cuando se pasa el puntero como un parámetro `var`. Al igual que en otros casos similares, todos los cambios realizados a los parámetros formales se harán también al parámetro real. Normalmente se utiliza un parámetro `var` cuando se espera que el procedimiento cambie el puntero.

Más complejo es el caso en el que un puntero se pasa por valor. En este caso, la computadora hace una copia legal del parámetro real de modo que cualquier cambio hecho al parámetro formal no se reflejará de nuevo. Por ejemplo, supongamos que en el procedimiento de escritura `VerNumeros` (imprime los números almacenados en una pila) en lugar de utilizar un puntero temporal para moverse a través de la lista, se utiliza el propio parámetro.

```

procedure VerNumeros (Cabeza: PtrLista);
begin
  while Cabeza < > nil do
  begin
    WriteLn (Cabeza^.Datos);
    Cabeza:= Cabeza^.Siguiente;
  end;
end;

```

El procedimiento funciona, pero ¿se destruye la lista cuando se cambia `Cabeza`? La respuesta es *no* ya que el parámetro `valor` protege el parámetro real del cambio.

4.4. EL TIPO GENÉRICO PUNTERO (*POINTER*)

Turbo Pascal permite un tipo especial de definición de puntero: «*genérico*» o «*no tipificado*». Difiere del puntero estándar en que no tiene un tipo base, no está definido como un puntero hacia algún tipo, sino simplemente como una variable de tipo *pointer*.

EJEMPLO 4.13

```
var
  enlace: pointer;
  ...
  enlace:= lis; (la variable lis figura en declaraciones precedentes)
```

La dirección contenida en la variable *lis* está asignada a la variable *enlace*.

Los punteros genéricos están especialmente concebidos para la programación de bajo nivel, para la cual Turbo Pascal ofrece buenas posibilidades.

4.5. LA ASIGNACIÓN DE MEMORIA EN TURBO BORLAND PASCAL

Turbo Pascal divide la memoria de su computadora en cuatro partes: el segmento de código, el segmento de datos, el segmento pila (*stack*) y el segmento *montículo* o *almacenamiento dinámico* (*heap*). Técnicamente la pila y el montículo no están totalmente separados, pero funcionan como entidades separadas.

El segmento de datos está claramente dedicado al almacenamiento de datos, pero en los otros tres segmentos también pueden almacenarse datos. La Figura 4.3 muestra el mapa de memoria simplificada de Turbo Pascal 7.0. Cada módulo (que incluye el programa principal y cada unidad) tiene su propio segmento de código.

El programa principal ocupa el primer segmento de unidades (en orden inverso de como están listadas en la cláusula *uses*) y el último segmento de código está ocupado por la biblioteca en tiempo de ejecución.

El tamaño de un segmento de código no puede exceder de 64 K, pero el tamaño total del código está limitado sólo por la memoria disponible. El segmento de datos contiene todas las constantes con tipo seguidas por todas las variables globales. El tamaño del segmento de la pila no puede exceder de 64 K (el tamaño por defecto es 16 K, que se pueden modificar con la directiva *\$M*).

El *buffer* o memoria intermedia de solapamiento (*overlay*) se utiliza por la unidad estándar *Overlay* para almacenar código recubierto. Si el programa no tiene *solapamiento*, el tamaño de la memoria intermedia del solapamiento es cero.

La Figura 4.4 (modificación de la 4.3) muestra cómo queda la memoria cuando un programa arranca, y en ella se observa que todas las variables locales se almacenan en la pila (*stack*) y las variables globales (también llamadas estáticas) se almacenan en el segmento de datos. El código y el segmento de datos están localizados en la parte baja de la memoria y la pila (*stack*) y el almacenamiento dinámico o montículo (*heap*) ocupan la zona alta de la memoria.

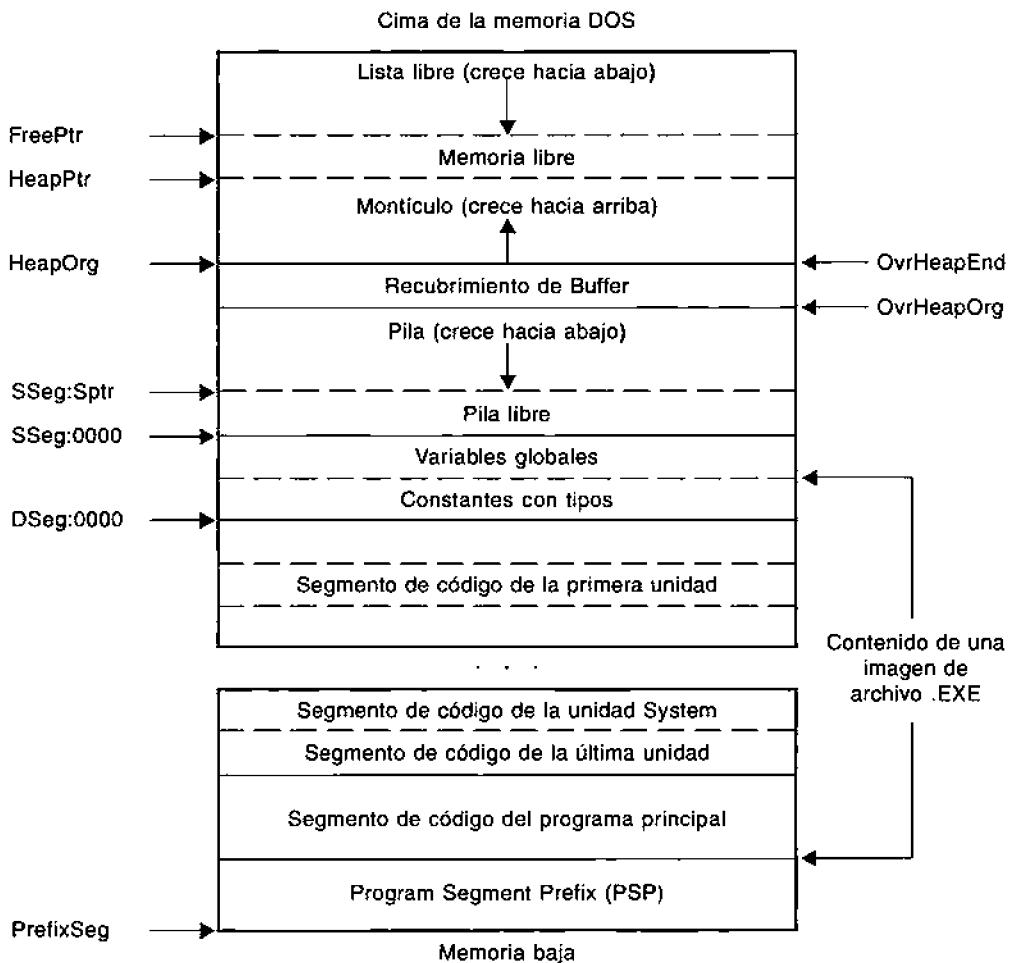


Figura 4.3. Mapa de memoria de Turbo Pascal 7.0.

El diagrama de la Figura 4.4 muestra también que la pila crece hacia abajo en la memoria y el montículo crece hacia arriba en la memoria. Aunque la pila y el montículo comparten la misma zona de la memoria, ellas nunca deben solaparse (recubrirse).

La mayoría de las variables que se declaran en Turbo Pascal son estáticas, su tamaño se fija en tiempo de compilación y no pueden variar. Por el contrario, el montículo almacena variables dinámicas.

4.5.1. El montículo (*heap*) y los punteros

El montículo o *heap* (*pila de variables dinámicas* o *almacenamiento dinámico*) almacena *variables dinámicas*, esto es, las variables asignadas a través de los procedimientos

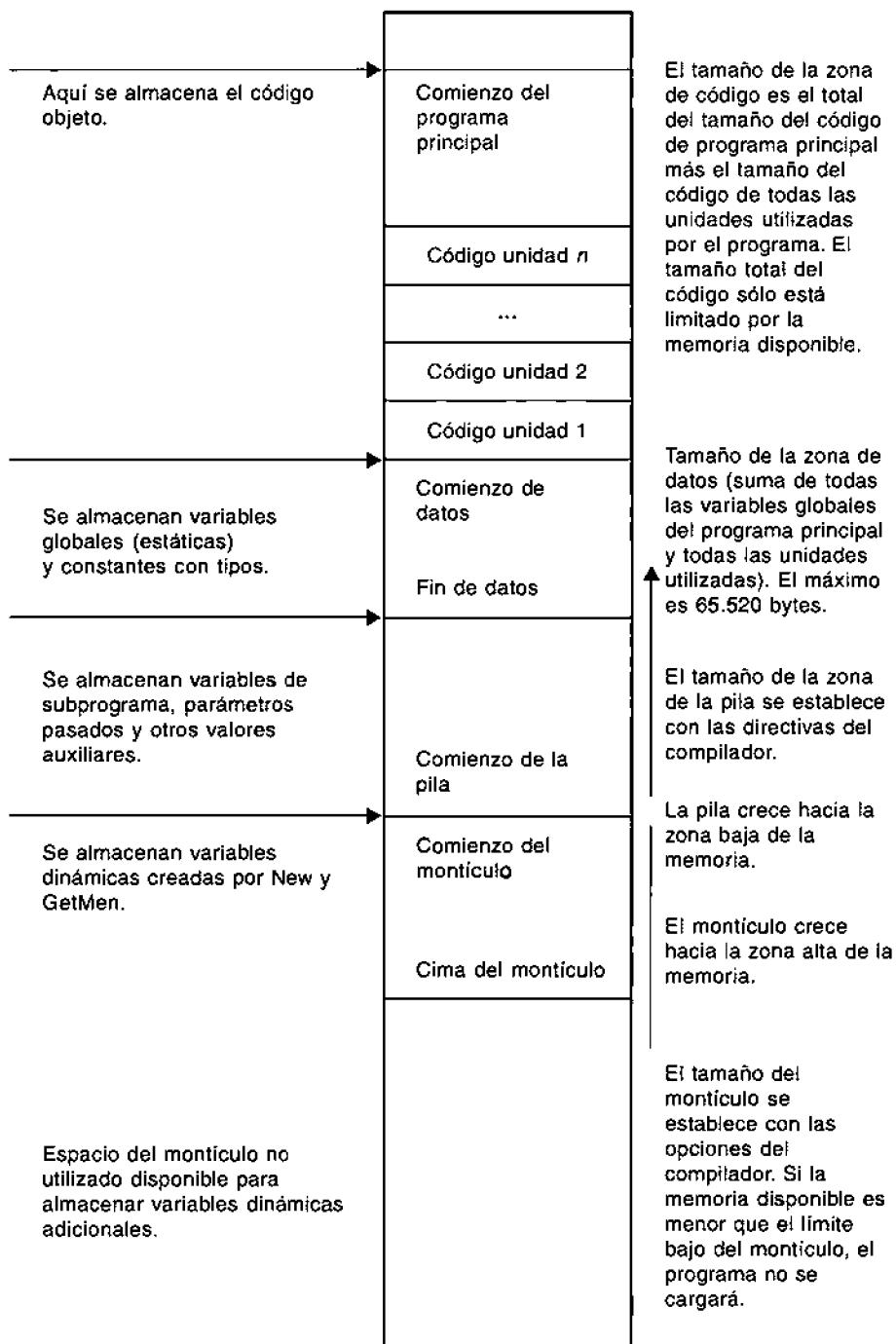


Figura 4.4. Mapa práctico de memoria de Turbo Pascal 7.0.

estándar *New* y *GetMem*. El montículo puede crecer o disminuir en el segmento correspondiente, ya que utiliza tipos de datos dinámicos: los *punteros*, que pueden crear o liberar variables dinámicas mientras el programa se está ejecutando. En resumen, las variables puntero pueden utilizar y reutilizar la memoria montículo.

El tamaño real del montículo depende de los valores mínimos y máximos que pueden fijarse con la directiva del compilador \$M. El tamaño mínimo es de 0 bytes, y el máximo por defecto es de 640 Kb; esto significa que por defecto el montículo ocupará toda la memoria restante (640 Kb viene definida por la máxima memoria direccionable por el DOS, aunque los procesadores 8086/88 tienen diecisésis segmentos que por un valor de 64 K de RAM resultaría 1.048.560 bytes = 1 Megabyte).

El límite inferior del montículo se almacena en la variable *HeapOrg*, y el límite o cuota superior (límite inferior de la memoria libre) se almacena en la variable *HeapPtr*. Cada vez que una variable dinámica se asigna en el montículo (vía *New* o *GetMem*), el gestor de la pila mueve *GetPtr* hacia arriba el tamaño de la variable.

El uso de las variables puntero en el montículo ofrece dos ventajas principales. Primero se amplía la cantidad total de espacio de datos disponibles en un programa; el segmento de datos está limitado a 64 K, pero el montículo, como ya se ha citado, está limitado sólo por la cantidad de **RAM** en su computadora. La segunda ventaja es permitir que su programa se ejecute con menos memoria. Por ejemplo, un programa puede tener dos estructuras de datos muy grandes, pero sólo una de ellas se utiliza en cada momento. Si estas estructuras de datos se declaran globalmente, ellas residen en el segmento de datos y ocupan memoria en todo momento. Sin embargo, si estas estructuras de datos se definen como punteros, se pueden poner en el montículo y quitarse cuando no se necesiten, reduciendo, por consiguiente, los requisitos de memoria.

4.5.2. Métodos de asignación y liberación de memoria

Los subprogramas que gestionan el montículo o almacenamiento dinámico son:

<i>Asignación dinámica de memoria</i>	<i>Espacio ocupado en memoria</i>
New	Dispose
Mark	Release
Getmem	FreeMem
	MaxAvail
	MemAvail

4.5.3. *New* y *Dispose*

Cuando **Dispose** libera o destruye una variable dinámica, puede dejar un agujero en el montículo. Si hace uso frecuente de **New** y **Dispose**, puede llegar a producirse una acumulación de espacio de memoria inservible «basura» (*garbage*). Algunos sistemas Pascal proporcionan rutinas específicas para comprimir esa información inservible (*garbage collection*), Turbo Pascal no las soporta, pero sí utiliza un sistema de gestión de la memoria montículo muy sofisticada que minimiza la pérdida de esos espacios inservibles. Los procedimientos **Mark** y **Release** proporcionan un método alternativo de gestión del montículo que eliminan la necesidad de la operación «eliminar basura». Sin embargo, hay un inconveniente: **Dispose** no es compatible con **Mark** y **Release**.

4.5.4. **Mark** y **Release**

Mark y **Release** son una alternativa a utilizar **New** y **Dispose** para asignar memoria dinámicamente.

El procedimiento **Mark** registra la dirección de la parte superior del montículo en una variable puntero. El procedimiento **Release** devuelve el montículo a un estado dado (reestructura la dirección).

Mark (varpunt)

Release (varpunt)

varpunt variable puntero de cualquier tipo puntero

Nota

Recuerde que **Dispose** y **Release** son métodos incompatibles de recuperación de memoria. Puede elegir utilizar uno u otro, nunca utilizar ambos en el mismo programa.

4.5.5. **GetMem** y **FreeMem**

Un tercer método de asignar memoria es **GetMem** y **FreeMem**. Se asemejan a **New** y **Dispose** en que asignan o liberan memoria, pero **GetMem** y **FreeMem** pueden especificar cuánta memoria se desea asignar con independencia del tipo de variable que está utilizando. Se pueden asignar bloques de memoria en la pila de una unidad de datos cuyo tamaño no se conoce en tiempo de compilación.

GetMem crea una nueva variable dinámica del tamaño especificado y pone la dirección del bloque en una variable puntero.

Formato

GetMem (varpunt, tamaño)

varpunt variable puntero de cualquier tipo puntero
tamaño expresión de tipo word

FreeMem libera una variable dinámica de un tamaño dado.

Formato

FreeMem (varpunt, tamaño)

Las variables asignadas con **GetMem** se liberan con **FreeMem**.

EJEMPLO 4.14

```
GetMem (y, 100);
FreeMem (y, 100);
```

Nota

El número de bytes especificado en *FreeMem* debe concordar con el especificado en *GetMem*. No se debe utilizar *Dispose* en lugar de *FreeMem*.

4.5.6. MemAvail y MaxAvail

El conocimiento de cuánta memoria está realmente disponible para variables dinámicas puede ser crítico. Turbo Pascal lleva un registro de cuánta memoria queda en el montículo, de modo que si se solicita más de la disponible, se generará un error en tiempo de ejecución. La solución es no crear una variable dinámica que sea mayor que la memoria disponible en el montículo. Turbo Pascal proporciona dos funciones para medir la memoria disponible: **MemAvail** y **MaxAvail**.

MaxAvail devuelve el tamaño del bloque libre contiguo más grande del montículo, correspondiente al tamaño de la variable dinámica más grande que se puede asignar a la vez. (El valor devuelto es un entero largo.)

Formato


MaxAvail

MemAvail devuelve un valor (del tipo entero largo) que indica, en bytes, la cantidad total de memoria disponible para variables dinámicas.

Formato


MemAvail

EJEMPLOS 4.15

```
begin
  Writeln (MemAvail, 'bytes disponibles');
  Writeln('el bloque libre más grande es:', MaxAvail,'bytes')
end
```

Programa

En el archivo Marathon van a almacenarse los datos personales de los participantes del Marathon Jerez-97.

Análisis. Este sencillo problema nos sirve para escribir un ejemplo de utilización de variable puntero y variable apuntada o referenciada. El puntero va a referenciar a un registro con los campos necesarios para representar los datos personales. Es evidente que podríamos utilizar una variable estática de tipo registro, pero es un ejemplo.

Codificación

```

program Marathon (input, output);
uses crt;
type
  registro = record
    nombre, apellido : string[25];
    dirección       : string[30];
    ciudad          : string[20];
    edad            : integer;
    marca           : record
      n, m, sg: integer
    end;
  Ptrreg = ^registro;
var
  R : Ptrreg; F: file of registro;

procedure Datos (var Pr: Ptrreg);
begin
  new (Pr);
  with Pr^do
  begin
    write('Nombre Apellido:'); readln (nombre, apellido);
    write('Direccion ciudad:');readln (dirección, ciudad);
    write ('Edad:');readln (edad);
    write('mejor Marca:');readln(marca.h,marca.m,marca.sg)
  end
end;
begin
  assign (F,'carrera.Dat');
  rewrite (F);
  writeln('Escribe los datos de cada participante');
  writeln('Para acabar proceso:Crln Z');
  while not eof do
  begin
    Datos(R);
    write(F,R^)
  end
end.

```

RESUMEN

Un puntero es una variable que se utiliza sólo para almacenar la dirección de memoria de otra variable. Un puntero «apunta» a otra variable cuando contiene una dirección de la variable.

Pascal tiene métodos especiales para declaración, iniciación y manipulación de punteros. La razón principal para utilizar punteros es que ellos hacen posible las *estructuras de datos dinámicas*, que les permiten crecer, decrecer e incluso desaparecer mientras se están ejecutando.

Una de las cosas peculiares sobre punteros es que sólo un puntero constante está predefinido: la constante `nil` (una palabra reservada) significa «puntero a nada».

En Turbo Pascal, un puntero se puede crear apuntando a una variable o a una estructura de datos de un solo modo: utilizando el procedimiento estándar de Pascal denominado `new` que crea tal variable o estructura de datos. La sentencia

```
new (CarPtr);
```

llama al procedimiento `new` y se ejecutan estas dos tareas:

- Asigna una nueva variable del tipo al que apunta `CarPtr`.
- Carga la dirección de la nueva variable en `CarPtr`.

El procedimiento `dispose` libera el espacio de memoria ocupada por una variable dinámica. La sentencia

```
dispose (p)
```

destruye la variable `p` y devuelve la zona de memoria ocupada al montículo (`heap`).

Turbo Pascal permite un tipo especial de definición de punteros: «genérico» o «no tipificado». Su diferencia con el puntero estándar es que no tiene un tipo base, no se define como un puntero hacia algún tipo, sino simplemente como una variable de tipo `pointer`. El compilador Turbo Pascal incorpora un conjunto de procedimientos que permiten gestionar dinámicamente la memoria, además de `new` y `dispose`. Estos procedimientos son: `GetMem`, `FreeMem`, `Mark`, `MaxAvail` y `MemAvail`.

EJERCICIOS

4.1. Suponga que se tienen las declaraciones:

```
type
  Indice = 0..9;
  IndicePuntero = ^Indice;
var
  I : Indice
  IPtr: IndicePuntero
```

- ¿Qué contendrá `IPtr`?
- Si se ejecuta el código

```
New (IPtr);
IPtr^:=5,
I := 4
```

¿qué contiene `IPtr`? ¿Qué contendrá `IPtr^`?

4.2. Despues de ejecutar el siguiente código

```
type
  Cosa = Integer;
  CosaPuntero = ^Cosa;
var
  T, TT : Cosa;
  TPtr, TTPtr : CosaPuntero;
```

```

begin
  TPtr := nil;
  New (TPtr);

```

¿Cuál de las variables siguientes contienen basura (información no válida o impredecible)?

- a) TPtr
- b) T
- c) TT
- d) TPtr^
- e) TTPtr^
- f) TTPtr

4.3. Supongamos que:

```

var
  IP : ^Char;

```

¿Es legal la llamada `New (Ip^)`? ¿Y qué hace `New (Ip)`? Explique sus respuestas.

4.4. ¿Qué imprime el siguiente programa?

```

program Mackoy;
type
  PunteroC = ^Char;
var
  P1, P2 : PunteroC;
begin
  New (P1);
  New (P2);
  P1^:= 'A';
  P2^:='B';
  P1 := P2;
  writeln (P1^);
  writeln (P2^);
end.

```

4.5. ¿Cuál es la salida de este programa?

```

Program SierraDeCazorla;
Const
  ESPACIO = " ";
type
  Cadena 30 = string [30];
  Entrada =
    record
      Nombre : Cadena 30;
      Numero : Integer;
    end;
  EntradaPtrTipo = ^Entrada;
var
  P1, P2 : EntradaPtrTipo;
begin
  new (P1);
  P1^.Nombre := 'manzanas';
  writeln (P1^.Nombre, ESPACIO, P1^.Número);
  new (P2);

```

```

P2^.Nombre := 'chumberas';
P2^.Numero :=52;
WriteLn (P1^.Nombre, ESPACIO, P1^.Numero);
WriteLn(P2^.Nombre, ESPACIO, P2^.Numero;
end.
```

- 4.6. Un programa contiene las siguientes declaraciones:

```

type
  Cadena25 = string [25];
  Entrada = record
    Nombre : Cadena25;
    Numero : Integer;
  end;
  PtrTipo = ^Entrada;
var
  P1, P2 : PtrTipo;
```

¿Cuál es el significado de las siguientes sentencias?

- a) P1^
- b) ^P1
- c) Cadena25^
- d) ^Cadena25
- e) P1 := P2
- f) P1^ = P2
- g) P1 := P2^

- 4.7. Dadas las declaraciones siguientes

```

type
  Cadena=string[20];
  PtrCad=^Cadena;
var
  C: Cadena;
  Pc,Pch: PtrCad;
```

señalar los errores y las correcciones necesarias.

```

begin
  C:='Vivienda';
  new(Pc); Pc:=C;
  new(Pd); Pd^:=Pc^;
```

- 4.8. Escribir las declaraciones necesarias para declarar un tipo puntero a un vector de 20 elementos. A continuación escribir un procedimiento con el argumento el tipo puntero y que dé entrada a los 20 elementos del vector. En el procedimiento se ha de reservar memoria para el vector.
- 4.9. Dado el vector definido en 4.8, escribir una función que tenga como entrada el puntero al vector y devuelva la suma de los elementos positivos.
- 4.10. Dado el vector definido en el problema 4.6, escribir una función que tenga como entrada la dirección del vector (puntero) y devuelva un puntero a otro vector del mismo tamaño sin elementos duplicados.

PROBLEMAS

- 4.1. Escribir las declaraciones necesarias para definir un tipo de datos puntero a un registro de estudiante con los campos Apellido-1, Apellido-2, Nombre, Fecha-nacimiento y notas. Los tres primeros campos definirlos de tipo puntero a cadena de 25 caracteres. El campo Fecha puntero a un registro con los campos día, mes y año. Y el campo Notas puntero a un array de 10 elementos.
- 4.2. Escribir un procedimiento que devuelva los datos correspondientes a un registro de estudiante definido en 4.1. En el procedimiento se ha de reservar memoria para cada uno de los campos del registro.
- 4.3. Definir un vector de registro del tipo definido en 4.1. Escribir un procedimiento en el que se dé entrada a los registros de los estudiantes de una clase que tiene un máximo de 40 alumnos. Utilizar el procedimiento definido en 4.2.
- 4.4. Definir un archivo cuyos elementos van a ser los registros definidos en 4.1, con la salvedad de que en vez de los campos puntero, los campos sean del tipo «apuntado». Escribir un procedimiento que tome como entrada el vector generado en 4.7 y escriba los registros en el archivo.
- 4.5. Definir de nuevo el registro enunciado en 4.5 para escribir el procedimiento de entrada de campos del registro con la particularidad de que se reserve memoria con el procedimiento GetMem. El número de bytes a reservar se debe ajustar al tamaño real del campo.
- 4.6. Una ecuación de segundo grado depende de tres parámetros de tipo real: a, b, c. Escribir un programa que tenga como datos de entrada los coeficientes de la ecuación de segundo grado y como salida las raíces de la ecuación. Definir los parámetros a, b, c como punteros a números reales de tal forma que en el procedimiento de entrada se reserve memoria para ellos.

Listas enlazadas: el TAD lista enlazada

CONTENIDO

- 5.1. Especificación formal del tipo abstracto de datos *lista*.
- 5.2. Implementación del TAD *lista* con estructuras dinámicas.
- 5.3. Implementación del TAD *lista* mediante variables dinámicas.
- 5.4. Iniciar una lista enlazada.
- 5.5. Búsqueda en listas enlazadas.
- 5.6. Operaciones de dirección: siguiente, anterior, último.
- 5.7. Inserción de un elemento en una lista.
- 5.8. Borrado de un elemento de una lista.
- 5.9. Recorrido de una lista.
- 5.10. Lista ordenada.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Una *lista enlazada* (o lista) es una secuencia de nodos en el que cada nodo está enlazado o conectado con el siguiente. La lista enlazada es una estructura de datos dinámica cuyos nodos suelen ser normalmente registros y que no tienen un tamaño fijo.

En el capítulo se desarrollan algoritmos para insertar, buscar y borrar elementos. De igual modo se muestra el tipo abstracto de datos (TAD) que representa a las listas enlazadas.

5.1. ESPECIFICACIÓN FORMAL DEL TIPO ABSTRACTO DE DATOS LISTA

Una forma de almacenar elementos relacionados es alinearlos, formando una lista lineal que necesita un enlace por cada elemento, para referenciar al elemento sucesor.

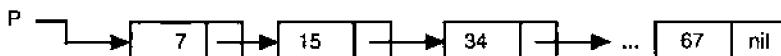


Figura 5.1. Lista enlazada de números enteros.

En la Figura 5.1 se muestra una lista enlazada de enteros a la que se accede a través del puntero P. Una lista es una estructura que 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.

Una lista es una secuencia de 0 a n elementos. A la lista de cero elementos llamaremos lista vacía. Cada elemento de una lista se denomina *nodo*. En un nodo podemos considerar que hay dos campos, *campo de información* (Info) y *campo de enlace* (Enlace) o dirección del elemento siguiente.

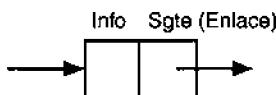


Figura 5.2. Nodo de una lista enlazada.

El campo de dirección, a partir del cual se accede a un nodo de la lista, se llama *puntero*. 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 de apuntar a ningún elemento, no debe de tener ninguna dirección, por lo que contiene un valor especial denominado puntero nulo (*nil*).

La lista vacía, aquella que no tiene nodos, tiene el puntero externo de acceso a la lista apuntando a nulo.

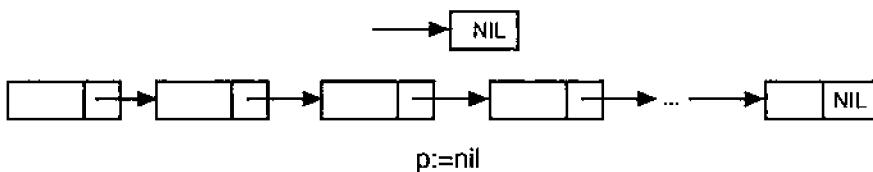


Figura 5.3. Lista enlazada sin información o inicialización de lista (lista vacía).

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 supresión (eliminación) de nodos.

Las inserciones se pueden realizar por cualquier punto de la lista. Así pueden realizarse inserciones 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 que se desea eliminar.

Especificación formal del TAD lista

Matemáticamente, una lista es una secuencia de cero o más elementos de un determinado tipo.

(a₁, a₂, a₃, ..., a_n) donde n >= 0,
si n = 0 la lista es vacía.

Los elementos de la lista tienen la propiedad de que sus elementos están ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que a_i precede a a_{i+1} para i = 1, 2, 3, 4, 5, ..., n-1 y que a_i sucede a a_{i+1} para i = 2, 3, 4, 5, 6, ..., n.

Para formar el tipo de datos abstracto Lista a partir de la noción matemática de lista, se debe definir un conjunto de operaciones con objetos de tipo Lista. Las operaciones:

Listavacia(L)	Inicializa la lista.
Esvacia(L)	Función que determina si la lista es vacía.
Inserprim(X, L)	Inserta un nodo con la información X como primer nodo de la lista L.
Inserta(X, P, L)	Inserta en la lista L un nodo con el campo X, delante del nodo de dirección P.
Inserfin(X, L)	Inserta un nodo con el campo X como último nodo de la lista L.
Localiza(X, L)	Función que devuelve la posición/dirección donde está el campo de información X. Si no está devuelve nulo.
Suprime(X, L)	Elimina de la lista el nodo que contiene a X.
Suprimedir(P, L)	Elimina de la lista el nodo cuya dirección/posición viene dada por P.
Anterior(P, L)	Función que devuelve la posición/dirección del nodo anterior a P.
Siguiente(P, L)	Función que devuelve la posición/dirección del nodo siguiente a P.
Primero(P)	Función que devuelve la posición/dirección del primer nodo de la lista L.
Ultimo(P)	Función que devuelve la posición/dirección del último nodo de la lista L.
Anula(L)	Esta operación vacía la lista L.
Visualiza(L)	Esta operación visualiza el campo de información de todos los elementos de la lista L.

Estas operaciones son consideradas como básicas en el manejo de 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, por lo que es posible que puedan aparecer otras operaciones posteriormente.

Independientemente de la implementación de las operaciones, se escribe un ejemplo de manejo de una lista. El objetivo es que dada una lista *L*, eliminar aquellos nodos repetidos. El campo de información de los nodos puede ser cualquiera (*Tipoinfo*). La función *Iguales* devuelve verdadero en el caso de que los dos parámetros de *Tipoinfo* sean iguales.

```

procedure Eliminadup(var L: Lista);
var
  P, Q: Posicion; {P posición actual en L,
                    Q posición avanzada}

begin
  P:= Primero(L);
  while P<> nulo do
    begin
      Q:= Siguiente(P,L);
      while Q <> nulo do
        if Iguales(Info(P), Info(Q)) then
          Suprimerdir(Q,L)
        else
          Q:= Siguiente(Q,L);
      P:= Siguiente(P,L)
    end;
end;

```

Para representar las listas y las correspondientes implementaciones pueden seguirse dos alternativas:

- Utilización de la estructura estática array para almacenar los nodos de la lista.
- Utilización de estructuras dinámicas, mediante punteros y variables dinámicas.

5.2. IMPLEMENTACIÓN DEL TAD *LISTA CON ESTRUCTURAS ESTÁTICAS*

En la implementación de una lista mediante arrays, los elementos de ésta se guardan en posiciones contiguas del array. Con esta realización se facilita el recorrido de la lista y la operación de añadir elementos al final. Sin embargo, la operación de insertar un elemento en una posición intermedia de la lista obliga a desplazar en una posición a todos los elementos que siguen al nuevo elemento a insertar con objeto de dejar un «hueco».

En esta realización con estructuras estáticas se define el tipo *Lista* como un registro con dos campos: el primero, es el array de elementos {tendrá un máximo número de elementos, pre-establecido}; el segundo, es un entero, que representa la posición que ocupa el último elemento de la lista. Las posiciones que ocupan los nodos en la lista son valores enteros; la posición *i*-ésima es el entero *i*.

Declaraciones:

```

const
  Max= 100;

```

```

type
  Tipoinfo= ... {tipo del campo información de cada nodo}
  Lista = record
    Elementos: array[1..Max] of Tipoinfo;
    Ultimo: integer
  end;
  Posicion= 0 .. Max;

```

El procedimiento `Listavacia(L)` inicializa la lista, para lo cual simplemente se pone a cero `Ultimo`.

```

procedure Listavacia(var L: Lista);
begin
  L.Ultimo := 0
end;

```

El procedimiento `Inserprim(X,L)`, que inserta `X` como primer nodo, debe desplazar una posición a la derecha todos los nodos, para así poder asignar `X` en la primera posición.

```

procedure Inserprim(X: Tipoinfo;var L: Lista);
var
  I: integer;
begin
  with L do
  begin
    for I:= Ultimo downto 2 do
      Elementos[I+1]:= Elementos[I];
    Elementos[1]:= X
    ultimo := ultimo + 1
  end
end;

```

El procedimiento `Inserta(X,P,L)` desplaza los elementos de la lista a partir de la posición `P` e inserta `X` en dicha posición.

```

procedure Inserta(X: Tipoinfo;P: Posicion; var L: Lista);
var
  Q: Posicion;
begin
  if L.Ultimo= Max then
    < error: lista L llena >
  else if (P> L.ultimo) or (P< 1) then
    < error: posición no existe >
  else begin
    for Q:= L.Ultimo downto P do
      (desplaza los elementos a la «derecha»)
      L.Elementos[Q+1]:= L.Elementos[Q];
    L.Ultimo:= L.Ultimo+1;
    L.Elementos[P]:= X
  end
end;

```

Para implementar la operación `Suprime(X, L)`, primero hay que encontrar la posición que ocupa X, a continuación se elimina.

```
procedure Suprime(X: Tipoinfo; var L: Lista);
var
  I: integer;
  P: integer;
begin
  P := Localiza(X, L);
  if P > 0 then
  begin
    L.Ultimo := L.Ultimo - 1;
    for I := P to L.Ultimo do
      {desplaza a la izquierda}
      L.Elementos[I] := L.Elementos[I+1];
  end
end;
```

Otra forma de escribir el procedimiento `Suprime` es aprovechar el procedimiento `Suprimedir`:

```
procedure Suprime(X: Tipoinfo; var L: Lista);
var
  P: integer;
begin
  P := Localiza(X, L);
  if P > 0 then
    Suprimedir(P, L)
end;
```

La operación `Suprimedir(P, L)`, que elimina de la lista el nodo cuya posición viene dada por P, tiene que hacer lo opuesto que la operación `Inserta`: desplazar a la «izquierda» los elementos que se encuentran a partir de la posición P:

```
procedure Suprimedir(P: Posicion; var L: Lista);
var
  Q: Posicion;
begin
  if (P > L.Ultimo) or (P < 1) then
    < error: posición no existe >
  else begin
    L.Ultimo := L.Ultimo - 1;
    for Q := P to L.Ultimo do
      {desplaza a la izquierda}
      L.Elementos[Q] := L.Elementos[Q+1]
  end
end;
```

La operación de localizar, `Localiza(X, L)` devuelve la posición de un elemento X:

```
function Localiza(X: Tipoinfo; L: Lista): Posicion;
var
```

```

Q: Posicion;
Lc: boolean;
begin
  Q:= 1;
  Lc:= false;
  while (Q<= L.Ultimo) and not Lc do
    begin
      Lc:= L.Elementos [Q]= X
      if not Lc then Q:= Q+1
    end;
  if Lc then
    Localiza:= Q
  else
    Localiza:= 0
end;

```

5.3. IMPLEMENTACIÓN DEL TAD LISTA MEDIANTE VARIABLES DINÁMICAS

Este método de implementación de listas enlazadas utiliza variables puntero que permiten crear variables dinámicas, en las cuales se almacena el campo de información y el campo de enlace al siguiente nodo de la lista. Con esta implementación se ajusta la memoria ocupada al tamaño real de la lista, no como ocurre en la realización mediante estructuras estáticas en las que es necesario prever un máximo de elementos o nodos. Ahora, cada vez que sea necesario crear un nuevo nodo se llama al procedimiento que capta la memoria necesaria de la pila de memoria libre, y se enlaza con la lista. De igual forma cuando sea necesario eliminar un nodo, la memoria ocupada por éste se devuelve a la pila de memoria libre. El proceso es dinámico, la lista crece o decrece según las necesidades.

Con esta realización se evitan los desplazamientos de elementos a la derecha o a la izquierda en las operaciones de inserción o eliminación, respectivamente. Por contra, cada vez que se quiere acceder a un nodo hay que recorrer la lista, a través de los enlaces hasta alcanzar su dirección. También hay que tener en cuenta que en esta realización de listas, cada nodo ocupa la memoria adicional del campo de enlace.

Las variables enteras contienen o pueden contener valores enteros, y sucede igual con variables reales. El tipo de variables puntero (o apuntadores) son aquellas cuyo contenido va a ser la dirección de memoria de un dato. Las variables puntero, al igual que las variables enteras o reales, son variables estáticas, ya que se crean (se reserva memoria para ellas) en tiempo de compilación.

Puntero= ^Tipoapuntado;

Con esta declaración de tipo estamos expresando que los valores que contenga una variable de tipo *Puntero* van a ser direcciones de memoria del tipo *Tipoapuntado*. Gráficamente:



Con los punteros se pueden construir toda clase de estructuras enlazadas. Estas estructuras son dinámicas; es decir, el número de elementos puede cambiar durante la ejecución del programa. Para lo cual hay procedimientos que permiten crear y destruir elementos durante la ejecución del programa, y así se ajustan a las necesidades de cada instante, cosa que no ocurre con las estructuras estáticas como los arrays.

Al trabajar con estructuras dinámicas hay que manejar dos clases diferentes de variables: *variables puntero* (direcciónan un dato) y *variables referenciadas* (son apuntadas).

```
type
  Ptrentero= ^integer;
  Ptrcadena= ^string[25];
var
  Pe: Ptrentero;
  Pc: Ptrcadena;
```

Las variables puntero Pe y Pc van a contener la dirección en memoria de un dato entero y de un dato cadena, respectivamente.

Lo habitual será que el tipo del dato al que apunta una variable puntero sea un registro, éste tendrá un campo de información y otro, u otros campo de enlace, por consiguiente del mismo tipo puntero.

```
type
  Ptrnodo = ^Persona;
  Persona = record
    Nombre: string[25];
    Direcc: string[50];
    Edad : integer;
    Enlace: Ptrnodo
  end;
```

Variables del tipo puntero Ptrnodo van a estar asociadas con variables referenciadas del tipo Persona.

```
var
  P: Ptrnodo;
```



En la declaración anterior se crea la variable estática P, cuyo valor, de momento, está indefinido. Esta variable P apuntará a una variable referenciada, todavía no creada, del tipo Persona. La creación de la variable referenciada se realiza mediante el procedimiento new

```
new(variable puntero);           new(P);
```

Este procedimiento reserva memoria, crea una variable referenciada cuyo tipo (Persona) está declarado dentro de las definiciones formales. Después de la ejecución del procedimiento new la variable puntero tiene la dirección en memoria de la nueva variable creada.

Así, P contendrá la dirección de la variable creada del tipo Persona. ¿Cómo se nombran a las variables referenciadas? Tienen el mismo nombre que las variables puntero poniendo a continuación el carácter ^ (o una flecha).

`new(P);` *P^ es la variable referenciada del tipo Persona.*

Con esta variable se pueden hacer exactamente las mismas operaciones que con una variable registro del tipo Persona.

```
new(P);
P^.Nombre:= 'Juan Marco';
readln(P^.Edad);
P^.Enlace:= nil;
```

Existe una operación inversa a new que permite liberar la memoria referenciada por una variable puntero: procedimiento **dispose**.

dispose(P); libera la memoria, devuelve esa memoria a la pila de memoria libre, asignada anteriormente con **new**.

Operaciones con variables puntero

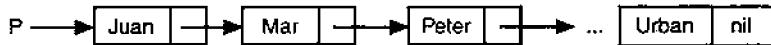
Las variables puntero se pueden asignar y comparar con los operadores =, <>. Los punteros pueden pasarse como parámetros a los subprogramas; una función puede ser de tipo puntero, es decir, puede devolver un puntero.

```
type
  Ptr= ^Elemento;
  Elemento = record
    Numero: integer;
    Enlace: Ptr
  end;
var
  P1, P2: Ptr;
begin
  P2:= P1;
  P2:= P1^.Enlace;           Se asigna el campo Enlace de la variable referenciada por P1.
  P1^.Enlace:= P2^.Enlace;
  P2:= nil;                 Es posible hacer que una variable puntero no apunte a «nada» asignando nil.
```

Después de la operación de liberar memoria, **dispose(P)**. La variable puntero se queda con un valor arbitrario, entonces conviene asignar **nil**,

```
dispose(P1);
P1:= nil;
```

Ahora ya se puede escribir la implementación de las operaciones con listas simplemente enlazadas utilizando variables dinámicas. Las operaciones sobre una lista enlazada permiten acceder a la misma mediante un puntero externo, que contiene la dirección del primer nodo de la lista.



Supóngase los tipos

```

type
  Tipoinfo= ...
  Ptrnodo= ^Nodo;
  Nodo = record
    Info: Tipoinfo;
    Sgte: Ptrnodo
  end;
    
```

5.4. INICIAR UNA LISTA ENLAZADA

Las operaciones usuales de iniciación de una lista enlazada son: *crear una lista vacía* y *comprobar si una lista está vacía*.

Crear lista

Inicia una lista sin nodos, como lista vacía.

```

procedure Listavacia(var L: Ptrnodo);
begin
  L:= nil
end;
    
```

Esvacia

Esta operación es una función que determina si la lista no tiene nodos.

```

function Esvacia(L: Ptrnodo): boolean;
begin
  Esvacia := L = nil
end;
    
```

5.5. BÚSQUEDA EN LISTAS ENLAZADAS

La operación de búsqueda en una lista enlazada, requiere una operación Localiza que permite localizar la dirección de un nodo que contenga un campo de información determinado; asimismo, se puede plantear una operación Existe que sirve para deter-

minar si hay un nodo con cierta información. La función Localiza devuelve la dirección de un nodo o nil. Los códigos fuente de ambas funciones son:

```

function Localiza(X:Tipoinfo;L:Ptrnodo): Ptrnodo;
var
  T: Ptrnodo;
begin
  T:= L;
  while (T^.Sgte<> nil) and (T^.Info<> X) do
    T:= T^.Sgte;
  if T^.Info<> X then
    Localiza:= nil
  else
    Localiza:= T
end;

function Existe(X: Tipoinfo; L:Ptrnodo): boolean;
begin
  if not Esvacia(L) then
  begin
    while (L^.Info<> X) and (L^.Sgte<> nil) do
      L:= L^.Sgte;
    Existe:= L^.Info=X
  end
  else
    Msge('Error en llamada')
end;

```

5.6. OPERACIONES DE DIRECCIÓN: SIGUIENTE, ANTERIOR, ÚLTIMO

Operaciones usuales en la construcción de algoritmos de manipulación de listas enlazadas son aquellas que permiten la búsqueda y localización de un cierto nodo. Este es el caso de las operaciones *Anterior*, *Siguiente* y *Último*.

Anterior (P, L)/Siguiente (P, L)

Con esta operación se obtiene la dirección del nodo anterior a P, o bien nil si no existe tal nodo. Para lo que hay que recorrer la lista hasta encontrar P.

```

function Anterior(P:Ptrnodo; L:Ptrnodo): Ptrnodo;
{El anterior de lista vacía, del primer nodo y de dirección no
 existente de la lista: nil}
begin
  if Esvacia(L) or (P=nil) or (L=P) then
    Anterior:= nil
  else begin
    while (L^.Sgte<> P) and (L^.Sgte<> nil) do
      L:= L^.Sgte;
    if L^.Sgte= P then
      Anterior:= L
  end;
end;

```

```

    else
      Anterior:= nil
    end
end;

```

La operación siguiente devuelve la dirección del nodo siguiente a uno dado, o bien nil si es el último nodo.

```

function Siguiente (P:Ptrnodo; L:Ptrnodo): Ptrnodo;
begin
  if Esvacia(L) or (P=nil) then
    Siguiente:=nil
  else
    Siguiente:=P^.Sgte
end;

```

Último (L)

Para obtener la dirección del último nodo se ha de recorrer toda la lista.

```

function Ultimo(L:Ptrnodo): Ptrnodo;
{ El último de lista vacía consideramos nil }
begin
  if Esvacia(L) then
    Ultimo:= nil
  else
    begin
      while L^.Sgte<> nil do
        L:= L^.Sgte;
      Ultimo:= L
    end
end;

```

5.7. INSERCIÓN DE UN ELEMENTO EN UNA LISTA

La operación de inserción de un elemento con un campo de información X siempre supone crear un nuevo nodo. Para ello se utiliza el procedimiento new. Además habrá que hacer un movimiento de enlaces. La función Crear devuelve la dirección de un nuevo nodo.

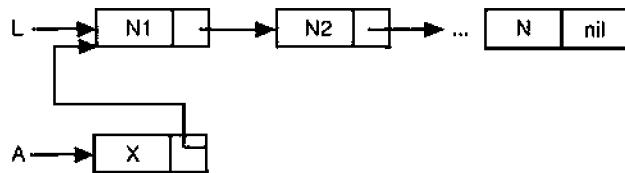
```

function Crear(X: Tipoinfo): Ptrnodo;
var
  N: Ptrnodo;
begin
  new(N);
  N^.Info:= X;
  N^.Sgte:= nil;
  Crear:= N
end;

Inserprim(X,L)

```

Añade un nodo con la información X como primer nodo de la lista

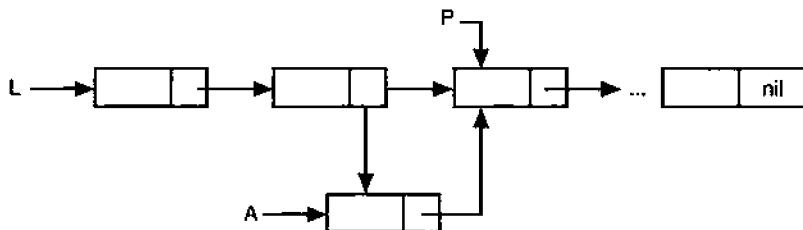


```

procedure Inserprim(X: Tipoinfo; var L: Ptrnodo);
var
  A: Ptrnodo;
begin
  A := Crear(X);
  A^.Sgte := L;
  L := A
end;

Inserta(X, P, L)
  
```

Añade a la lista L un nodo con el campo X, delante del nodo de dirección P.



```

procedure Inserta(X: Tipoinfo; P: Ptrnodo; var L:Ptrnodo);
var
  A: Ptrnodo;
begin
  A := Crear(X);
  if Esvacia(L) then
    L := A
  else if P = L then
    begin
      A^.Sgte := P;
      L := A
    end
  else begin  {Ahora es cuando se enlaza el nuevo nodo entre el nodo
               anterior y el nodo P}
    Anterior(P,L)^.Sgte := A;
    A^.Sgte := P
  end
end;
  
```

Otra forma de enlazar el nuevo nodo consiste en añadir el nuevo nodo a continuación de P e intercambiar el campo de información:

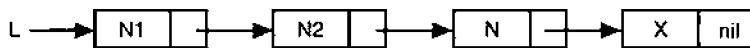
```

procedure Inserta(X: Tipoinfo; P: Ptrnodo; var L: Ptrnodo);
...
else
begin  {El nuevo nodo es enlazado a continuación de P, después se
        intercambia el campo Info}
    A^.Sgte:= P^.Sgte;
    P^.Sgte:= A;
    A^.Info:= P^.Info;
    P^.Info:= X
end;

Inserfin(X,L)

```

Añade un nodo con el campo X como último nodo de la lista.



```

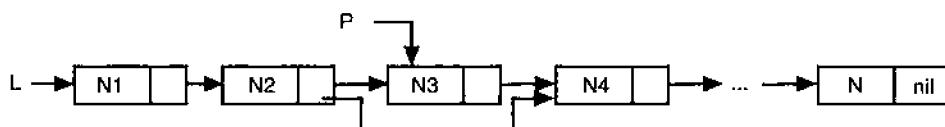
procedure Inserfin(X: Tipoinfo; var L: Ptrnodo);
var
  A: Ptrnodo;
begin
  A:= Crear(X);
  if Esvacia(L) then
    L:= A
  else
    Ultimo(L)^.Sgte:= A
end;

```

5.8. SUPRESIÓN DE UN ELEMENTO DE UNA LISTA

La operación de supresión (borrado) supone enlazar el nodo anterior con el nodo siguiente al que va a ser borrado y liberar la memoria ocupada por el nodo a borrar. Esta operación se realiza con una llamada al procedimiento `dispose`.

Con las operaciones de supresión de un nodo y `Ultimo` resulta fácil eliminar todos los nodos de una lista, quedando ésta como lista vacía.



`Suprime(X,L)` Se elimina el nodo que contiene a X.

```

procedure Suprime(X: Tipoinfo; var L: Ptrnodo);
var
  A: Ptrnodo;
begin
  A:= Localiza(X,L);
  if A < > nil then
  begin
    if A = L then {Primer nodo}
      L:= L^.Sgte
    else
      Anterior(A,L)^.Sgte:= A^.Sgte; {enlaza anterior con siguiente}
    dispose(A)
  end
end;

```

También puede codificarse haciendo una llamada a Suprimedir:

```

procedure Suprime(X: Tipoinfo; var L: Ptrnodo);
var
  A: Ptrnodo;
begin
  A:= Localiza(X,L);
  if A < > nil then
    Suprimedir(A, L)
end;

```

Suprimedir(P,L) Elimina de la lista el nodo cuya dirección viene dada por P.

```

procedure Suprimedir(P: Ptrnodo; var L: Ptrnodo);
begin
  if P = L then {Primer nodo}
  begin
    L:= L^.Sgte;
    dispose(P)
  end
  else if Anterior(P, L)<> nil then
  begin
    Anterior(P, L)^.Sgte:= Siguiente(P)
    {enlaza anterior con siguiente}
    dispose(P)
  end
end;

```

Anula(L) Esta operación libera la memoria ocupada por todos los nodos de la lista L. Se fundamenta en las operaciones de Ultimo y Suprimedir.

```

procedure Anula(var L: Ptrnodo);
begin
  while not Esvacia(L) do
    Suprimedir(Ultimo(L),L)
end;

```

5.9. RECORRIDO DE UNA LISTA

Una operación frecuente en cualquier algoritmo de manipulación de listas es el recorrido de los nodos de la lista enlazada. El qué se haga con cada nodo dependerá del problema que se esté resolviendo, ahora son visitados para escribir el campo de información.

```
procedure Visualiza(L: Ptrnodo);
var
  Q: Ptrnodo;
begin
  Q:= L;
  while Q<> nil do
  begin
    writeln(Q^.Info, ' ');
    Q:= Q^.Sgte
  end
end;
```

5.10. LISTA ORDENADA

En las listas tratadas anteriormente los elementos de las mismas están ordenadas por la posición que ocupan dentro de la lista. Si el tipo de información que representa cada elemento es un tipo ordinal, o tiene un subcampo ordinal, se puede mantener la lista ordenada respecto a dicho campo.

Al representar la lista ordenada mediante un puntero externo L, éste apuntará al primer nodo en el orden creciente de los elementos.

Dada la lista $(a_1, a_2, a_3, \dots, a_n)$ estará ordenada si $a_1 \leq a_2 \leq a_3 \leq \dots \leq a_{n-1} \leq a_n$.

La representación enlazada:



5.10.1. Implementación de una lista ordenada

La formación de una lista ordenada se basa en dos operaciones: Inserorden, añade el elemento X a la lista, manteniendo la ordenación; Posinser, operación que obtiene la posición a partir de la cual hay que añadir el elemento X en la lista para mantener la ordenación.

Posinser(X, L)

Devuelve la dirección del nodo anterior a X según la ordenación de los elementos. Nil, si es el anterior del primero.

```

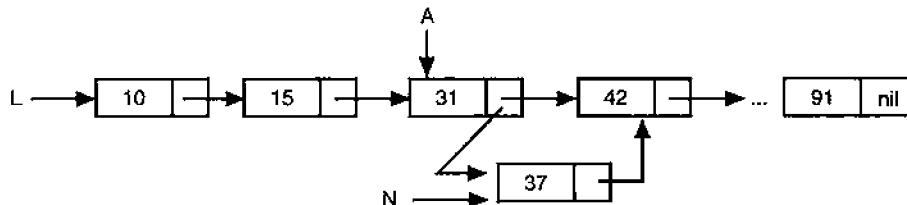
function Posinser(X:Tipoinfo; L:Ptrnodo):Ptrnodo;
var
  T: Ptrnodo;
begin
  T:= nil;
  if Not Esvacia(L) then
  begin
    while (X >= L^.Info) and (L^.Sgte<> nil) do
    begin
      T:= L;
      L:= L^.Sgte
    end;
    if X>= L^.Info then {Es el último nodo}
      T:= L;
  end
  Posinser:= T
end;

Inserorden(X,L)

```

Si la lista está vacía, el nodo se inserta como el primero de la lista. En caso contrario se presentan dos casos:

- El nodo ha de ser el primero de la lista.
- El nodo ha de situarse entre dos nodos, o bien el último. Al tener la dirección del anterior, se reajustan los enlaces.



```

procedure Inserorden(X:Tipoinfo; var L: Ptrnodo);
var
  A, N: Ptrnodo;
begin
  N:= Crear(X);
  if Esvacia(L) then
    L:= N;
  else begin
    A:= Posinser(X,L);
    if A = nil then {Se añade como primer nodo}
      begin
        N^.Sgte:= L;
        L:= N
      end
    else begin
      N^.Sgte:= A^.Sgte;
      A^.Sgte:= N
    end
  end
end;

```

5.10.2. Búsqueda en lista ordenada

La operación de búsqueda de un elemento en una lista ordenada es más eficiente que en una lista general. Para decidir que un elemento no está en la lista basta encontrar un elemento mayor, no hace falta recorrer la lista hasta el final. La función Buscorden realiza la búsqueda en una lista ordenada.

```
function Buscorden(X:Tipoinfo; L:Ptrnodo):Ptrnodo;
var
  T: Ptrnodo;
begin
  T:= L;
  while (T^.Sgte<> nil) and (T^.Info< X) do
    T:= T^.Sgte;
  if T^.Info= X then
    Buscorden:= T
  else
    Buscorden:= nil
end;
```

El resto de operaciones sobre listas son iguales ya estén o no ordenadas. Como en la operación de borrado de un elemento hay que buscar antes la posición que ocupa, es más eficiente cambiar la llamada a Localiza por Buscorden.

PROBLEMA 5.1

Como ejemplo de listas ordenadas, se escribe un programa en el que se forma una lista ordenada de N números enteros aleatorios. Una vez formada, se eliminan los elementos repetidos y se muestra la lista.

Ánalisis

La unidad ListaOrdenada contiene las operaciones de manejo de listas. El programa ListaOr contiene un procedimiento EliminarDup que sirve para eliminar elementos duplicados y otro procedimiento Mostrar que visualiza la lista. El algoritmo de EliminarDup recorre la lista nodo a nodo, comparando el nodo actual con el siguiente; si son iguales se libera la memoria del nodo siguiente, se restablecen los punteros y continúa el proceso con el siguiente nodo. La lista estará formada por números enteros generados aleatoriamente de 1 a 99. El número de nodos, lo fijamos mediante la constante Elementos = 55.

Interfaz de la unidad

```
unit ListaOrdenada;
interface
type
  Tipoinfo = integer;
  Ptrnodo = ^Nodo;
```

```

Nodo = record
  Info:Tipoinfo;
  Sgte:Ptrnodo
end;
function Posinser (X:Tipoinfo; L:Ptrnodo) : Ptrnodo;
procedure Inserorden (X:Tipoinfo, var L:Ptrnodo);
function Buscorden (X:Tipoinfo; L:Ptrnodo): Ptrnodo;
implementation
...{Código correspondiente a los procedimientos / funciones escritos
anteriormente}
end.

```

Codificación del programa

```

program ListaOr(input, output);
uses ListaOrdenada, Crt;
const
  Elementos = 55;
  M = 99;
var
  L,Q:Ptrnodo;
  Y: integer;

procedure EliminarDup(L:Ptrnodo);
var
  A,Q: Ptrnodo;
begin
  A:=L;
  while A <> nil do
    if A^.Sgte<> nil then
      if A^.Info=A^.Sgte^.Info then {Un repetido}
        begin
          Q := A^.Sgte;
          A^.Sgte := Q^.Sgte;
          dispose(Q)
        end
      else A:=A^.Sgte
      else A:=A^.Sgte
    end;
  end;

procedure Mostrar(L:Ptrnodo);
var
  Q:Ptrnodo;
begin
  Q:=L;
  while Q <> nil do
    begin
      write (Q^.Info,' ');
      Q:=Q^.Sgte
    end;
  end;

begin {Bloque ppal}
  L:=Nil; randomize;
  for Y:=1 to Elementos do
    Inserorden(Random (99)+1,L);

```

```

clrscr; {la lista es mostrada}
Mostrar(L);
{Los elementos duplicados son eliminados de la lista ordenada}
EliminarDup(L);
gotoxy(15, WhereY+1);
writeln ('Lista sin elementos duplicados');
Mostrar(L)
end.

```

PROBLEMA 5.2 (*vector dinámico*)

Se desea definir un vector de números reales de manera dinámica (*vector dinámico*), utilizando punteros para formar una lista enlazada que represente el vector. La estrategia de creación consiste en crear en una única operación («de golpe») un espacio de memoria que reserve memoria suficiente para contener mx elementos.

En la operación de asignación se van añadiendo consecutivamente los nuevos elementos en la memoria reservada y si no hubiera espacio se amplía a otros mx elementos. La operación de supresión de un elemento requiere que un espacio equivalente al ocupado por el elemento quede libre para posteriores asignaciones.

Solución

El vector se representa mediante una lista enlazada con un puntero al primer nodo y otro al último nodo con número real añadido. La estrategia de asignación es la indicada en el enunciado del problema planteado.

La unidad VectDina contiene los tipos de datos y las operaciones necesarias para cumplir los requisitos del problema.

```

unit VectDina;
interface
const
  Mx=20;
type
  Telem=real;
  Ptr=^Elem;
  ELEM=record
    Dat:Telem;
    Sgte:Ptr
  end;
  Tvector=record
    A,Ult:Ptr;
    N:integer {número de elementos del vector}
  end;

procedure Creacion (var V:Tvector);
function Direccion (V:Tvector; X:Telem):Ptr;
function Anterior (V:Tvector; W:Ptr):Ptr;
procedure Asignar (var V:Tvector; T:Telem);
procedure Borrar (var V:Tvector; T:Telem);
procedure Mostrar (V:Tvector);

```

```

implementation
uses crt;

function Dirección (V:Tvector; X:Telem):Ptr;
var P:Ptr;
  Sw:boolean;
begin
  P:=V.A; Sw:=false;
  while (P<>V.Ult) and not Sw do
  begin
    Sw:=P^.Dat=X;
    if not Sw then P:=P^.Sgte
  end;
  if not Sw then
    if V.Ult^.Dat <> X then
      Direccion:= nil
    else
      Direccion:=V.Ult
  else Direccion:=P
end;

function Anterior (V:Tvector; W:Ptr):Ptr;
var P:Ptr;
begin
  if V.A=W then
    Anterior:=nil
  else begin
    P:=V.A;
    while P^.Sgte <>W do
      P:=P^.Sgte;
    Anterior:=P
  end
end;

function Reserva:Ptr;
var
  W:Ptr;
  I:integer;
begin
  new(W); Reserva:=W;
  for I:=2 to Mx do
  begin
    new(W^.Sgte);
    W:=W^.Sgte
  end;
  W^.Sgte:=nil
end;

procedure Creacion (var V:Tvector);
begin
  V.A:=Reserva;
  V.Ult:=nil;
  V.N:=0
end;

procedure Asignar (var V:Tvector;T:Telem);

```

```

begin
  if V.Ult=nil then
  begin
    V.A^.Dat:=T;
    V.Ult:=V.A
  end
  else begin
    if V.Ult^.Sgte=nil then
      V.Ult^.Sgte:=Reserva;
    V.Ult:=V.Ult^.Sgte;
    V.Ult^.Dat:=T
  end;
  V.N:=V.N+1
end;

procedure Borrar (var V:Tvector; T:Telem);
var W,Anter,Aux:Ptr;
begin
  W:=Direccion(V,T);
  if W <> nil then
  begin
    Anter:=Anterior(V,W);
    if W=V.Ult then
      V.Ult:=Anter
    else begin
      if V.A=W then
        V.A:=V.A^.Sgte
      else
        Anter^.Sgte:=W^.Sgte;
      {Enlaza a partir de V.Ult el nodo a borrar W}
      Aux:=V.Ult^.Sgte;
      V.Ult^.Sgte:=W;
      W^.Sgte:=Aux
    end;
    V.N:=V.N-1
  end
  else writeln(T:5:1,'no puede ser eliminado.');
end;

procedure Mostrar (V:Tvector);
var
  J:integer;
  P:Ptr;
begin
  P:=V.A;
  for J:= 1 to V.N do
  begin
    write(P^.Dat:5:1, ' ');
    if (WhereX+6)>80 then writeln;
    P:=P^.Sgte
  end;
  writeln
end;
begin
end.

```

El programa Prob_vector prueba esta realización particular de un vector, que consiste en crear el vector con n números reales y a continuación insertar o borrar elementos.

```
program Prob_vector (input, output);
uses Crt, VectDina;
var
  V:Tvector;
  O:char;
  X:Telem;

procedure Iniciar (var V:Tvector);
const A=999; M=30;
var
  i:integer;
begin
  Creacion(V);
  clrscr;randomize;
  for i:=1 to M do
    Asignar(V,random(A));
  writeln('Vector inicial':44);
  Mostrar(V)
end;

begin
  Iniciar (V);
  write ('Ahora tienes las opciones:');
  writeln ('Insertar(I)/Borrar(B)/Mostrar(M)/Salir(S)');
  repeat
    write ('Elige I/B/M/S:'); readln(O);
    case O of
      'I', 'i': begin
        write ('Nuevo item:'); readln(X);
        Asignar (V,X)
      end;
      'B', 'b': begin
        write ('Item a eliminar:'); readln(X);
        Borrar (V,X)
      end;
      'M', 'm': Mostrar (V)
    end;
  until O in ['S','s'];
end.
```

RESUMEN

La estructura de datos «lista ordenada» 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 ordenadas físicamente como están en un *array*. El final de la lista se señala mediante una constante o puntero especial llamada *nil*.

La ventaja de una lista enlazada sobre un *array* es que la lista enlazada puede crecer y decrecer en tamaño y que es fácil insertar o suprimir un valor (nodo) en el centro de una lista enlazada.

La lista enlazada es una estructura muy versátil. Los algoritmos para inserción y eliminación de datos constan de dos pasos:

- Recorrer la lista desde el principio hasta que se alcanza la posición apropiada.
- Ejecutar cambios en los punteros para modificar la estructura de la lista.

El TAD lista enlazada es uno de los más utilizados en gestión de proyectos software, debido a que la estructura lista suele aparecer en la resolución de numerosos problemas.

EJERCICIOS

Todos los ejercicios y problemas a los que se hace referencia deben de considerar la lista enlazada implementada mediante punteros. La dirección de acceso a la lista está en la variable puntero L.

- 5.1. Escribir la función *Cardinal* 1 que calcule el número de nodos de una lista enlazada.
- 5.2. Escribir un procedimiento que añada un nuevo elemento a la lista L a partir del elemento *i*-ésimo.
- 5.3. Escribir un procedimiento que elimine de una lista L el nodo *i*-ésimo.
- 5.4. Escribir una función que devuelva la dirección del nodo *i*-ésimo de la lista L.
- 5.5. Escribir la función *INVERSA* que tiene como argumento de entrada la lista enlazada L, devuelva la dirección de otra lista que tenga los nodos en orden inverso, es decir, último nodo pase a ser el primero, penúltimo pase a ser el segundo, y así sucesivamente.
- 5.6. Escribir un procedimiento que tenga como argumento de entrada una lista L de números enteros, de la que se sabe que tiene nodos repetidos. El procedimiento creará otra lista cuyos nodos contendrán las direcciones de los nodos repetidos en la lista L. Definir los tipos de datos para representar ambas listas.
- 5.7. Una lista de cadenas de caracteres está ordenada alfabéticamente. Escribir un procedimiento para suprimir aquel nodo que contenga la cadena S.
- 5.8. Se quiere implementar una lista enlazada mediante arrays de tal forma que cada elemento del array contenga dos campos: campo de información y un segundo campo que llamaremos apuntador. El campo apuntador tiene la posición que ocupa el siguiente nodo de la lista, el último nodo tiene el campo apuntador a cero.
Escribir los tipos de datos para esta representación de la lista.
- 5.9. Dada la representación de la lista propuesta en el anterior ejercicio, escribir el procedimiento de insertar un nodo como primer elemento de la lista.
- 5.10. Dada la representación de la lista 5.8, escribir el procedimiento de insertar un nodo a partir del que ocupa la posición P.
- 5.11. Siguiendo con la representación propuesta en 5.8, ahora escribir la función que localice un nodo con el campo X.
- 5.12. Con la representación de lista propuesta en 5.8, escribir el procedimiento de suprimir el nodo con el campo de información X.

PROBLEMAS

- 5.1. Dada una lista enlazada de números enteros, escribir las rutinas necesarias para que dicha lista esté ordenada en orden creciente. La ordenación se ha de hacer intercambiando los punteros a los nodos.
- 5.2. Se dispone una lista enlazada ordenada con claves repetidas. Realizar un procedimiento de inserción de una clave en la lista, de tal forma que si la clave ya se encuentra en la lista la inserte al final de todas las que tienen la misma clave.
- 5.3. Dos cadenas de caracteres están almacenadas en dos listas. Se accede a dichas listas mediante los punteros L_1 , L_2 . Escribir un subprograma que devuelva la dirección en la cadena L_2 a partir de la cual se encuentra la cadena L_1 .
- 5.4. Dada una cadena de caracteres almacenada en una lista L . Escribir un subprograma que transforme la cadena L de tal forma que no haya caracteres repetidos.
- 5.5. Se quiere representar el tipo abstracto de datos conjunto de tal forma que los elementos estén almacenados en una lista enlazada. Escribir una unidad para implementar el TAD conjunto mediante listas. En la unidad deberá de contener los tipos de datos necesarios y las operaciones:
 - Conjunto vacío.
 - Añadir un elemento al conjunto.
 - Unión de conjuntos.
 - Intersección de conjuntos.
 - Diferencia de conjuntos.

Nota: Los elementos del conjunto que sean de tipo cadena.

- 5.6. Escribir un programa en el que dados dos archivos de texto F_1 , F_2 se formen dos conjuntos con las palabras respectivas de F_1 y F_2 . Posteriormente encontrar las palabras comunes a ambos y mostrarlas por pantalla. Utilizar la unidad de conjuntos del problema 5.5 para resolver este supuesto.
- 5.7. 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 pesetas.
 - Aumentar en un 3 por 100 el sueldo de los empleados que ganan más de P pesetas.
 - Dar de baja a los empleados con más de 35 años de antigüedad.
- 5.8. 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 la unidad `Lista_de_Enteros` para encapsular el tipo lista de números enteros.
 - Escribir la unidad `Lista_de_Palabras` para encapsular las operaciones que van a manejar la lista de palabras propuesta.
 - Escribir un programa que haciendo uso de la(s) unidades anteriores resuelva el problema.

- 5.9. El polinomio $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ deseamos representarlo en una lista enlazada, de tal forma que cada nodo contenga el coeficiente y el grado de un monomio. Escribir un programa que tenga como entrada los coeficientes y exponentes de cada término de un polinomio, y forme una lista enlazada para representarlo; ha de quedar en orden decreciente respecto al grado del polinomio. En el programa deben de encontrarse las operaciones:
- Evaluación del polinomio para un valor dado de x .
 - Obtención del polinomio derivada de $P(x)$.
 - Obtención del polinomio producto de dos polinomios.
- 5.10. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa para representar mediante listas un vector disperso. Y realizar las operaciones:
- Suma de dos vectores dispersos.
 - Producto escalar de dos vectores dispersos.

Listas doblemente enlazadas

CONTENIDO

- 6.1. Especificación de lista doblemente enlazada.
- 6.2. Realización de una lista doble mediante variables dinámicas.
- 6.3. Una aplicación resuelta con listas doblemente enlazadas.
- 6.4. Especificación de lista circular.
- 6.5. Realización de una lista circular mediante variables dinámicas.
- 6.6. Realización de listas circulares con doble enlace.

RESUMEN.

EJERCICIOS.

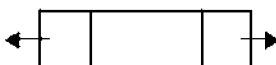
PROBLEMAS.

Las listas enlazadas se recorren en un solo sentido, normalmente en un sentido establecido. En numerosas ocasiones es deseable avanzar en cualquiera de los dos sentidos. Estas listas se denominan *listas doblemente enlazadas*. Otro tipo de listas que suelen ser también de gran utilidad son *listas circulares*.

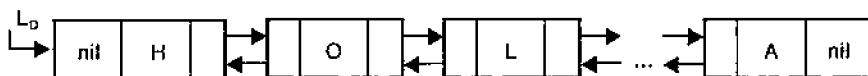
En el capítulo se examinan las especificaciones de las listas doblemente enlazadas y listas circulares, así como los algoritmos que realizan su implementación.

6.1. ESPECIFICACIÓN DE LISTA DOBLEMENTE ENLAZADA

En las listas simplemente enlazadas hay un solo sentido en el recorrido de la lista. Puede resultar útil el poder avanzar en ambos sentidos, de tal forma que los términos predecesor y sucesor no tengan significado puesto que la lista es completamente simétrica. En cada nodo de una lista doblemente enlazada existen dos enlaces, uno al siguiente nodo y otro al nodo anterior.



Un nodo de una lista doblemente enlazada puede ser considerada como un registro con tres campos: un campo información y dos campos de enlace.



La implementación de listas doblemente enlazadas se puede realizar con estructuras estáticas, o bien con estructuras dinámicas.

Las operaciones que pueden ser definidas en el TAD lista doblemente enlazada son similares a las operaciones con listas simplemente enlazadas.

Listavacia (Ld)	Inicializa la lista.
Esvacia (Ld)	Función que determina si la lista es vacía.
Inserprim (X,Ld)	Inserta un nodo con la información X como primer nodo de la lista Ld.
Inserta (X,P,Ld)	Inserta en la lista Ld un nodo con el campo X delante del nodo de dirección P.
Inserfin (X,Ld)	Inserta un nodo con el campo X como último nodo de la lista Ld.
Localiza (X,Ld)	Función que devuelve la posición/dirección donde está el campo de información X. Si no está devuelve nulo.
Suprime (X,Ld)	Elimina de la lista el nodo que contiene a X.
Suprimedir (P,Ld)	Elimina de la lista el nodo cuya dirección/posición viene dada por P.
Primer (P)	Función que devuelve la posición/dirección del primer nodo de la lista L.
Ultimo (Ld)	Función que devuelve la posición/dirección del último nodo de la lista Ld.
Anula (Ld)	Esta operación elimina/libera todos los nodos de la lista Ld.
Visualiza (Ld)	Esta operación visualiza el campo de información de todos los elementos de Ld.

Puede observarse que no aparecen las operaciones Anterior ni Siguiente. Al tener cada nodo dos enlaces, uno al siguiente y otro al anterior, y ser una lista simétrica, se dispone directamente de las direcciones de los nodos contiguos a uno dado.

Las operaciones anteriores son las operaciones básicas para manipulación de listas. Dependiendo del tipo de representación elegida y del problema a resolver podrán existir otras operaciones sobre listas.

6.2. IMPLEMENTACIÓN DE UNA LISTA DOBLEMENTE ENLAZADA MEDIANTE VARIABLES DINÁMICAS

Se utiliza el tipo puntero a un dato. Este dato tendrá un campo de información, relativo a lo que queremos almacenar, y dos campos de enlace que por tanto también serán pun-

teros. Con esta implementación, la lista doble se representa con una variable puntero Ld a un nodo extremo que por conveniencia le consideramos el primer nodo de la lista.

```
type
  Tipoinfor = ...;
  Ptrndble = ^Nododoble
  Nododoble = record
    Info: Tipoinfo;
    Sgte, Anter: Ptrndble
  end;
```

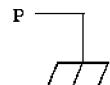
6.2.1. Creación de nodos en la lista

Las operaciones utilizadas serán *Cargar la lista* (ListaVacia), *Esvacia*, *Localiza*, *Existe*, *Ultimo*, *Inserta* e *Insertafín*.

- **Crear lista**

Inicia una lista doble sin nodos, como lista vacía.

```
procedure ListaVacia (var Ld:Ptrndble);
begin
  Ld:=nil
end;
```



- **Esvacia**

La función Esvacia devuelve verdadero si la lista está vacía y falso en caso contrario.

```
function Esvacia (Ld: Ptrndble): boolean;
begin
  Esvacia := Ld = nil
end
```

- **Localiza**

Devuelve la dirección de un nodo o nil.

```
function Localiza (X:Tipoinfo; Ld:Ptrndble):Ptrndble;
var
  T: Ptrndble;
begin
  T := Ld;
  if not Esvacia(Ld) then
    while (T^.Sgte < > nil) and (T^.Info < > X) do
      T := T^.Sgte;
    if T^.Info < > X then
      Localiza :=nil
    else
      Localiza := T
  end;
```

Una versión de Localiza es la operación Existe. Con esta operación se inspecciona si en una lista doble existe un nodo del cual se conoce su dirección.

• **Existe (P, Ld)**

```
function Existe (P: Ptrndble; Ld:Ptrndble): boolean;
begin
  if not Esvacia (Ld) and (P<> nil) then
    begin
      while (Ld<> P) and (Ld^.Sgte<> nil) do
        Ld := Ld^.Sgte;
      Existe := Ld=P
    end
  else
    Msge ('Error en llamada')
end
```

Ultimo (Ld)

La operación Ultimo obtiene la dirección del nodo que se encuentra en el otro extremo del nodo que consideramos el primero, el apuntado por Ld.

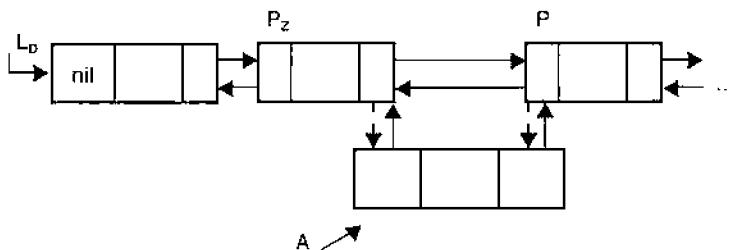
```
function Ultimo (Ld: Ptrndble) : Ptrndble;
{El último de lista vacía consideramos nil}
begin
  if Esvacia (Ld) then
    Ultimo := nil
  else begin
    while Ld^.Sgte<> nil do
      Ld := Ld^.Sgte;
    Ultimo := Ld
  end
end;
```

Cada vez que se realiza la inserción de un nodo con el campo de información X, hay que reservar memoria para dicho nodo. La operación crear realiza esta operación.

```
function Crear (X: Tipoinfo): Ptrndble;
var
  N: Ptrndble;
begin
  new (N);
  N^.Info := X;
  N^.Sgte := nil;
  N^.Anter := nil;
  Crear := N
end;
```

Inserta (X, P, Ld)

Añade a la lista Ld un nodo con el campo X, delante del nodo de dirección P.



```

procedure Inserta (X:Tipoinfo; P: Ptrndble; var Ld: Ptrndble);
var
  A: Ptrndble;
begin
  A := Crear (X);
  if Esvacia (Ld) then
    Ld := A
  else if P = Ld then {Delante del primer nodo}
  begin
    A^.Sgte := P;
    P^.Anter := A;
    Ld:= A
  end
  else if Existe (P, Ld) then
    begin {Ahora es cuando se enlaza el nuevo nodo entre el nodo
          anterior y el nodo P}
    P^.Anter^.Sgte := A;
    A^.Anter := P^.Anter;
    A^.Sgte := P;
    P^.Anter := A
  end
end;

```

Insertad (X, P, Ld)

Esta operación puede ser planteada como añadir el nodo con el campo de información X a continuación del nodo P en la lista Ld:

```

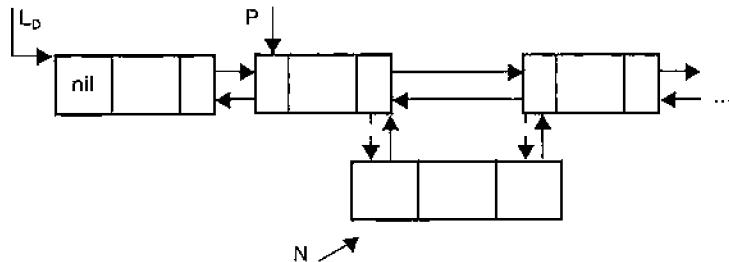
procedure Insertad (X: Tipoinfo; P: Ptrndble; var Ld: Ptrndble);
var
  N: Ptrndble;
begin
  if Esvacia (Ld) then
    Ld:= Crear (X)
  else if P = Ultimo (Ld) then
    Insertafin (X, Ld)
  else if Existe (P, Ld) then
    begin {Ahora es cuando se enlaza el nuevo nodo, a continuación del
          nodo P}
    N := Crear (X);
    P^.Sgte^.Anter := N;
    N^.Sgte := P^.Sgte;
  end
end;

```

```

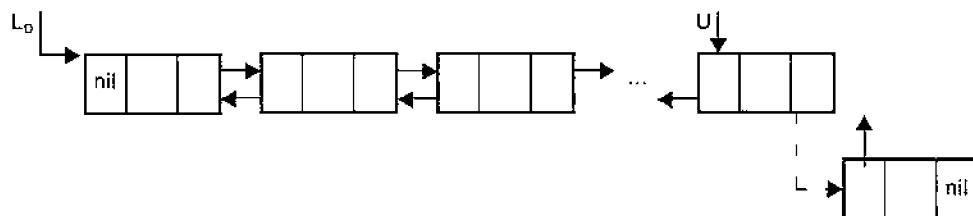
P^.Sgte := N;
N^.Anter := P
end
end;

```



Insertafin (X, Ld)

Añade un nodo con el campo X como último nodo de la lista.



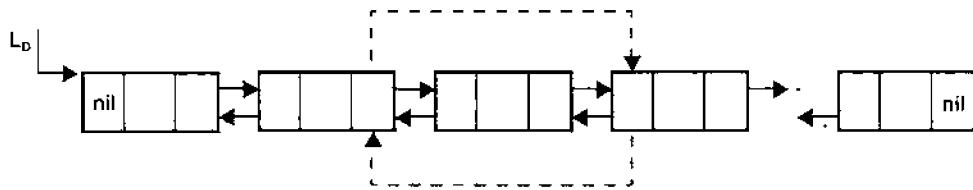
```

procedure Insertafin (X:Tipoinfo; var Ld: Ptrndble);
var
  N, U: Ptrndble;
begin
  N := Crear (X);
  if Esvacia (Ld) then
    Ld := N
  else begin
    U := Ultimo (Ld);
    U^.Sgte := N;
    N^.Anter := U
  end
end;

```

6.2.2. Eliminación de nodos

La eliminación de nodos de una lista doble la planteamos en dos operaciones muy similares. En la primera es pasado el campo de información del nodo que queremos borrar, y se elimina el primer nodo que se encuentra con dicho campo de información. En la segunda se tiene la dirección del nodo que quiere ser eliminado. Las acciones de supresión se realizan en la segunda operación.



```

procedure Suprime (X: Tipoinfo; var Ld: Ptrrndble);
var
  P: Ptrrndble;
begin
  P := Localiza (X, Ld);
  if P <> nil then
    Suprimedir (P, Ld)
  else
    Msge ('No está')
end;

procedure Suprimedir (P: Ptrrndble; var Ld: Ptrrndble);
begin
  if Existe (P, Ld) then
  begin
    if P = Ld then {Primer nodo}
      begin
        Ld := Ld^.Sgte;
        Ld^.Anter := nil
      end
    else if P < > Ultimo (Ld) then
      begin
        P^.Anter^.Sgte := P^.Sgte;
        P^.Sgte^.Anter := P^.Anter
      end
    else {Es el último nodo}
      P^.Anter^.Sgte := nil;
    dispose (P)
  end
end;

```

Anula (Ld)

Esta operación libera la memoria ocupada por todos los nodos de la lista Ld. El estado final de la lista Ld es vacía.

Una de las formas de realizar esta operación es suprimir nodos desde aquel que arbitrariamente consideramos último hasta el apuntado por Ld. Se fundamenta en las operaciones de Ultimo y Suprimedir.

```

procedure Anula (var Ld: Ptrrndble);
var
  P: Ptrrndble;
begin
  while Ld < > nil do
  begin
    P := Ultimo(Ld);

```

```

SuprimeDir(P,Ld);
end
end;

```

Visualiza (Ld)

Esta operación muestra el contenido de la lista en cualquier momento de manejo de la misma.

```

procedure Visualiza (Ld: Ptrndble);
begin
  while Ld<=> nil do
  begin
    Escribir (Ld^.Info);
    Ld := Ld^.Sgte
  end
end;

```

6.3. UNA APLICACIÓN RESUELTA CON LISTAS DOBLEMENTE ENLAZADAS

En un ambulatorio se desea asociar médicos con asegurados. Para ello se trata de formar una lista doblemente enlazada con todos los asegurados. Los datos de cada asegurado son su nombre y número de afiliación. A su vez, en otra lista doble se tendrá a los médicos del ambulatorio. Los datos de cada médico son nombre y número de teléfono (fono). La lista de médicos está ordenada alfabéticamente; además, cada nodo médico tiene un campo puntero, que referencia al nodo del primer asegurado asignado. Por consiguiente, cada nodo de asegurado tiene, además de los datos propios, un puntero al siguiente asegurado que pertenece al mismo médico; el último de los asegurados a un médico, contiene nil en el campo que forma la lista de asegurados a un médico.

La entrada de datos será interactiva; primero se introducen los datos de los médicos; a continuación, los datos de los asegurados. La asignación de los asegurados a los médicos se hará de manera aleatoria.

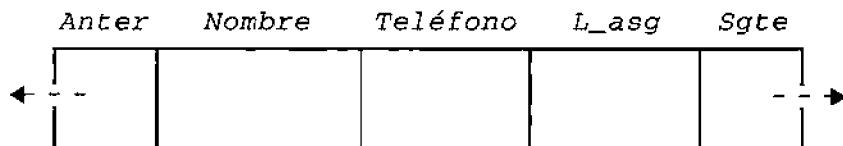
Análisis

La representación gráfica de las dos listas a crear:

Médico	Referencia	Asegurado	Referencia
Amador	6	1 Luis M	5
Castro	9	2 Marta B	
		3 Mertoli	7
		4 Tonino	
		5 Rufis A	3
		6 Rios C	8
		7 Fausto F	nil
		8 Slisa N	3
		9 Doroteo	

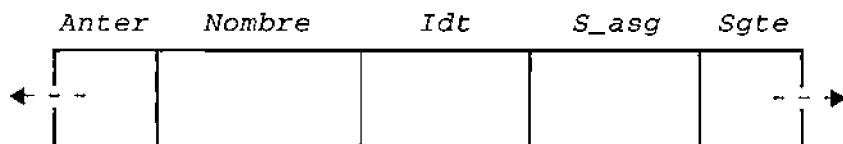
De esta representación podemos obtener que la lista de asegurados con el médico *Amador* comienza en el *nodo <6>*, *Rios C*, que le sigue *<8>*, *Silva N*, al nodo *<3>*, *Mertoli*, después *<7>*, *Fausto F*, y aquí termina.

Para la representación de datos, el nodo del médico:



L_asg: es el campo puntero al siguiente de la lista de asegurados.

El nodo del asegurado:



Los tipos de datos y operaciones con lista doble de asegurados en la unidad *ListAseg*.

Unidad ListAseg

```
unit ListAseg;
interface
type
  Cadna30:string[30];
  Cadnall:string[11];
  PtrAs=^NodoAs;
  NodoAs=record
    Nombre:Cadna 30;
    Idt: Cadnall;
    S_asg:PtrAs; {Referencia a siguiente asegurado de un mismo médico}
    Anter, Sgte:PtrAs
  end;

{Operaciones}
procedure Listavacia (var Ld:PtrAs);
function Esvacia (Ld:PtrAs) :boolean;
function Localiza (X:Cadnall; Ld:PtrAs) :PtrAs;
function Existe (P: PtrAs; Ld:PtrAs) :boolean;
function Ultimo (Ld:PtrAs) :PtrAs;
procedure Inserta (Nm:Cadna30; Id:Cadnall; P:PtrAs; var Ld:PtrAs);
procedure Insertad (Nm:Cadna30; Id:Cadnall; P:PtrAs; var Ld:PtrAs);
procedure Insertafin (Nm:Cadna30; Id:Cadnall; var Ld:PtrAs);
procedure Suprime (Id:Cadnall; var Ld:PtrAs);
procedure Supremedir(P:PtrAs; var Ld:PtrAs);
procedure Visualiza (Ld:PtrAs);
implementation
uses crt;
```

```

procedure Listavacia (var Ld:PtrAs);
begin
  Ld:=nil
end;

function Esvacia (Ld:PtrAs) :boolean;
begin
  Esvacia:=Ld=nil
end;

function Localiza (X:Cadnall;Ld:PtrAs) :PtrAs;
{Búsqueda se hace por el código de identificación}
var
  T:PtrAs;
begin
  T:=Ld;
  if not Esvacia (Ld) then
    while (T^.Sgte<>nil) and (T^.Idt<>X) do
      T:=T^.Sgte;
    if T^.Idt<>X then
      Localiza:=nil
    else
      Localiza:=T
  end;
end;

function Existe (p:PtrAs; Ld:PtrAs) :boolean;
begin
  if not Esvacia (Ld) and (P<>nil) then
    begin
      while (Ld<>p) and (Ld^.Sgte<>nil) do
        Ld:=Ld^.Sgte;
      Existe:=Ld=p
    end
    else
      writeln ('Error en llamada'); (<Msge (Error)>)
  end;
end;

function Ultimo (Ld:PtrAs) :PtrAs;
{El último de lista vacía consideramos nil}
begin
  if Esvacia (Ld) then
    Ultimo:=nil
  else begin
    while Ld^.Sgte<> nil do
      Ld:=Ld^.Sgte;
    Ultimo:=Ld
  end
end;
function Crear (Nm:Cadna30;Id:Cadnall):PtrAs;
var
  N:PtrAs;
begin
  new(N);
  N^.Nombre:=Nm;N^.Idt:=Id;N^.S_asg:=nil;
  N^.Sgte:=nil;N^.Anter:=nil;
  Crear:=N
end;

```

```

procedure Inserta(Nm:Cadna30;Id:Cadnall;P:PtrAs; var Ld:PtrAs);
{Añade a la lista Ld un nodo con el campo X, delante del nodo P}
var
  A:PtrAs;
begin
  A:=Crear(Nm, Id);
  if Esvacia (Ld) then
    Ld:=A
  else if P=Ld then {Delante del primer nodo}
  begin
    A^.Sgte:=P;
    P^.Anter:=A;
    Ld:=A
  end
  else if Existe(P, Ld) then
  begin {Ahora es cuando se enlaza el nuevo nodo entre el nodo
        anterior y el nodo P}
    P^.Anter^.Sgte:=A;
    A^.Anter:=P^.Anter;
    A^.Sgte:=P;
    P^.Anter:=A
  end
end;

procedure Insertad(Nm:Cadna30;Id:Cadnall;P:PtrAs; var Ld:PtrAs);
{Añadir nodo con el campo X a continuación del nodo P en la lista}
var
  N:PtrAs;
begin
  if Esvacia (Ld) then
    Ld:=Crear(Nm, Id)
  else if P = Ultimo(Ld) then
    Insertafin(Nm,Id,Ld)
  else if Existe(P,Ld) then
  begin {Ahora es cuando se enlaza el nuevo nodo, a continuación
        del nodo P}
    N:=Crear(Nm,Id);
    P^.Sgte^.Anter:=N;
    N^.Sgte:=P^.Sgte;
    P^.Sgte:=N;
    N^.Anter:=P
  end
end;

procedure Insertafin(Nm:Cadna30;Id:Cadnall;var Ld:PtrAs);
var
  N, U: PtrAs;
begin
  N:=Crear(Nm, Id);
  if Esvacia (Ld) then
    Ld:=N
  else begin
    U:=Ultimo (Ld);
    U^.Sgte:=N;
    N^.Anter:=U
  end
end;

```

```

{Borrado de nodos}
procedure Suprime(Id:Cadnall; var Ld: PtrAs);
var
  P: PtrAs;
begin
  P:=Localiza(Id, Ld);
  if P<> nil then
    Suprimedir (P, Ld)
  else
    writeln ('Error en la llamada.')
end;

procedure Suprimedir (P:PtrAs; var Ld:PtrAs);
begin
  if Existe (P, Ld) then
  begin
    if P=Ld then {Primer nodo}
    begin
      Ld:=Ld^.Sgte;
      Ld^.Anter:=nil
    end
    else if P<>Ultimo(Ld) then
    begin
      P^.Anter^.Sgte:=P^.Sgte;
      P^.Sgte^.Anter:=P^.Anter
    end
    else
      P^.Anter^.Sgte:=nil;
    dispose(P)
  end
end;

```

```

procedure Visualiza(Ld:PtrAs);
  procedure Escribir(A:NodoAs);
  begin
    write(A.Nombre);gotoxy(31,whereY);
    writeln(A.Idt)
  end;
begin
  while Ld<> nil do
  begin
    Escribir(Ld^);
    Ld:=Ld^.Sgte
  end;
begin
end.

```

Los tipos de datos y operaciones básicas para formar una lista doble ordenada las representamos en la unidad *ListMed*.

Unidad ListMed

```

unit ListMed;
interface

```

```

uses ListAseg;
type
  Cadna30:string[30];
  Cadnall:string[11];
  PtrMd:^NodoMd;
  NodoMd=record
    Nombre:Cadna30;
    Fono:Cadnall;{teléfono}
    L_asg:PtrAs;
    Anter,Sgte:PtrMd
  end;

  function Posinser (X:Cadna30;L:PtrMd):PtrMd;
  procedure Inserorden (M:Cadna30;F:Cadnall;var L:PtrMd);
  function Buscorden (X:Cadna30;L:PtrMd) :PtrMd;
  procedure Mostrar (L:PtrMd);
implementation

  function Posinser (X:Cadna30;L:PtrMd) :PtrMd;
  begin
    if(L<>nil) then
    begin
      while (X >= L^.Nombre) and (L^.Sgte<>nil) do
        L:=L^.Sgte;
      if X >= L^.Nombre then {Es el último nodo}
        Posinser:=L
      else
        Posinser:=L^. Anter
    end
    else {Lista vacía}
  end;

procedure Inserorden(M:Cadna30;F:Cadnall;var L:PtrMd);
var
  A, N:PtrMd;
begin
  new(N); N^.Nombre:=M; N^. Fono:=F;N^.L_asg:=nil;
  N^.Sgte:=nil; N^.Anter:=nil;
  if L=nil then
    L:=N
  else begin
    A:=Posinser(M, L);
    if A=nil then {Añadido como primer nodo}
    begin
      N^.Sgte:=L;L^.Anter:=N;
      L:=N
    end
    else begin {Añadido a partir de A}
      N^.Sgte:=A^.Sgte; N^.Anter:=A;
      if A^.Sgte<>nil then
        A^.Sgte^.Anter:=N;
      A^.Sgte:=N
    end
  end
end;
end;

```

```

function Buscorden (X:Cadna30;L:PtrMd) :PtrMd;
var
  T:PtrMd;
begin
  T:=L;
  while (T^.Sgte<>nil) and (T^.Nombre < X) do
    T:=T^.Sgte;
  if T^.Nombre=X then
    Buscorden:=T
  else
    Buscorden:=nil
end;

procedure Mostrar(L:PtrMd);
var
  Q:PtrMd;
begin
  Q:=L;
  while Q<>nil do
  begin
    writeln(Q^.Nombre,' ',Q^.Fono); Q:=Q^.Sgte
  end
end;
end.

```

Programa Ambulatorio de creación de listas

La primera acción para crear esta superestructura es dar entrada a los datos de los médicos y dar de alta los asegurados del ambulatorio. Por último, asignar (se hace aleatoriamente) cada asegurado a un médico, y formar la lista virtual de asegurados de un médico.

Realmente hay dos únicas listas, *médicos* y *asegurados*. A nivel lógico se forma una lista por cada médico para ello se utiliza el campo *S_asg* con el que va enlazando los asegurados del mismo médico.

Los nombres de los médicos y sus números de fono se encuentran en el archivo de texto *Medicos.txt*; al igual que los nombres de las personas adscritas al ambulatorio que están en el archivo *Asegurad.txt*.

Una vez formada toda esta superestructura pueden plantearse otros problemas, como, por ejemplo, dado un médico listar sus asegurados; otra operación, dado un asegurado mostrar el doctor que le pertenece, y otras más que podemos pensar de utilidad. En el programa se realiza la operación de listar todos los asegurados que tiene un médico.

```

program Ambulatorio;
uses ListAseg,ListMed,Crt;
var
  Lmed,Md:PtrMd;
  Lasg:PtrMd;
  Fmed,Fasg:Text;
  Nb:Cadna30;
  Ch:char;

```

```

procedure Medicos(var Lm:PtrMd);
var
  Nm:Cadna30; F:Cadnall;
  J:integer;
begin
  Lm:=nil;
  repeat
    readln(Fmed,Nm);
    readln(Fmed,F);
    Inserdorden(Nm,F,Lm);
  until eof(Fmed);
end;

procedure Asegurados(var Las:PtrAs);
var
  Nm:Cadna30; Id:Cadnall;
begin
  Las:=nil;
  repeat
    readln(Fasg,Nm);
    readln(Fasg,Id);
    Insertafin(Nm,Id,Las)
  until eof(Fasg)
end;

```

{Los siguientes procedimientos asignan aleatoriamente cada asegurado a un médico. Para ello se cuenta el número de médicos y con una función random, se asocia un asegurado al médico}

```

function Posicion(L:PtrMd;K:integer):PtrMd;
begin
  while K>1 do
  begin
    L:=L^.Sgte;
    K:=K-1
  end;
  Posicion:=L
end;

procedure Asocia(Asg:PtrAs;Lm:PtrMd);
var
  C:integer;
  M:PtrMd;
  function Cuantos(L:PtrMd):integer;
  var K:integer;
  begin
    K:=0;
    while L<> nil do
    begin
      L:=L^.Sgte;
      K:=K+1
    end;
    Cuantos := K
  end;

```

```

begin
  randomize;
  C:=Cuantos (Lm);if C=0 then Asg:=nil; {Para no entrar en el bucle}
  while Asg<>nil do
  begin
    M:=Posicion(Lm, random(C)+1);
    write(Asg^.Nombre);gotoxy(31,whereY);
    write('tiene asignado al doctor');
    writeln(M^.Nombre);
    {Enlazamos en la lista de asegurados, como el primero de ella}
    Asg^.S_asg:=M^.L_asg;
    M^.L_asg:=Asg;
    Asg:=Asg^.Sgte
  end
end;

procedure VerAseg(M:PtrMd);
var A:PtrAs;
begin
  {Recorre, visualizando por pantalla, los asegurados de un médico}
  A:=M^.L_asg;
  while A<>nil do
  begin
    write(A^.Nombre) ;gotoxy(31,whereY) ;writeln(A^.Idt);
    A:=A^.S_asg;
  end
end;

begin {programa Ambulatorio}
  clrscr;
  {Es creada la lista doble ordenada de médicos}
  assign (Fmed, 'Medicos.txt');reset(Fmed);
  Medicos(Lmed);
  Mostrar(Lmed);
  repeat until readkey in [#0..#255];clrscr;
  {Es creada la lista doble de asegurados}
  assign(Fasg, 'Asegurad.txt');reset(Fasg);
  Asegurados(Lasg);
  Visualiza(Lasg);
  repeat until readkey in [#0..#255];clrscr;
  {Se asocian}
  Asocia (Lasg,Lmed);
  repeat
    gotoxy(10,WhereY+2);
    write('Nombre del médico:');readln(Nb);
    Md:=Buscorden(Nb,Lmed);
    if Md<>nil then
      VerAseg(Md)
    else
      writeln('Médico no está en el ambulatorio');
    repeat write('Otro médico:');readln(Ch)
    until upcase(Ch) in ['S', 'N']
    until Ch in ['N', 'n']
  end. {Fin de programa Ambulatorio}

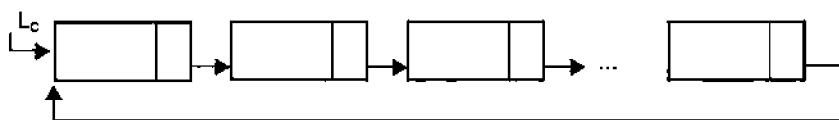
```

6.4. IMPLEMENTACIÓN DE UNA LISTA CIRCULAR MEDIANTE VARIABLES DINÁMICAS

En las listas lineales siempre hay un último nodo que tiene el campo de enlace a nil. Esto presenta el inconveniente de que dada la posición de un nodo P no se puede alcanzar cualquier otro nodo que preceda al nodo P.

Haciendo un pequeño cambio en la estructura de una lista lineal, de tal manera que no haya, al menos a nivel lógico, un último nodo sino que los nodos formen una estructura circular, podremos acceder a todos los nodos a partir de uno dado.

Se podría decir que el cambio que hay que hacer es que el último nodo en vez de apuntar a nil apunte al primer nodo. En realidad, en una estructura circular no hay



primero ni último.

Una lista circular, por su naturaleza no tiene primero ni último nodo. Sin embargo, resulta útil establecer un primer y un último nodo. Una convención es la de considerar que el puntero externo de la lista circular referencia al último nodo, y que el nodo siguiente sea el primer nodo. Todos los recorridos de la lista circular se hacen tomando como referencia el considerado último.

Sobre una lista circular pueden especificarse una serie de operaciones, y así formar el TAD lista circular. Estas operaciones coinciden con las definidas en los TAD lista y lista doblemente enlazada

Listavacia (Lc)	Inicializa la lista.
Esvacia (Lc)	Función que determina si la lista es vacía.
Primero (Lc)	Devuelve la dirección del primer nodo.
Ultimo (Lc)	Devuelve la dirección del último nodo.
Anterior (P, Lc)	Devuelve la dirección del nodo anterior a P.
Inserprim (X, Lc)	Inserta un nodo con la información X como primer nodo de la lista Lc.
Inserta (X, P, Lc)	Inserta en la lista Lc un nodo con el campo X, delante del nodo de dirección P.
Inserfin (X, Lc)	Inserta un nodo con el campo X como último nodo de la lista Lc.
Localiza (X, Lc)	Función que devuelve la posición/dirección donde está el campo de información X. Si no está devuelve nulo.
Suprime (X, Lc)	Elimina de la lista el nodo que contiene a X.
Suprimedir (P, Lc)	Elimina de la lista el nodo cuya dirección/posición viene dada por P.
Visualiza (Lc)	Muestra el campo de información de cada nodo de una lista circular.

6.5. IMPLEMENTACIÓN DE UNA LISTA CIRCULAR MEDIANTE VARIABLES DINÁMICAS

En esta implementación con punteros, hay un puntero externo *Lc* que referencia al que arbitrariamente se considera último nodo de la lista. Esta realización de lista circular a su vez es simplemente enlazada, por lo que el sentido de recorrido siempre es el mismo. Puede hacerse con dos enlaces y así podría recorrerse en ambos sentidos.

```
type
  Tipoinfo = ...;
  Ptrnlc = ^Nodolc
  Nodolc = record
    Info: Tipoinfo;
    Sgte,
    Anter: Ptrnlc
  end;
```

Las operaciones de *Listavacia*, *Esvacia* no hace falta escribirlas, son exactamente iguales que en las listas dobles. El resto de las operaciones de la especificación se detallan a continuación.

Primero (Lc)

```
function Primero(Lc: Ptrnlc): Ptrnlc;
begin
  if not Esvacia(Lc) then
    Primero := Lc^.Sgte
  else
    Primero := nil {Consideración arbitraria}
end;
```

Ultimo (Lc)

```
function Ultimo(Lc: Ptrnlc) : Ptrnlc;
begin
  Ultimo := Lc
end;
```

Anterior (P, Lc)

```
function Anterior (P: Ptrnlc; Lc: Ptrnlc) : Ptrnlc;
  {Consideraremos que el anterior de lista vacía y de dirección no
  existente de la lista: nil}
var
  A: Ptrnlc;
begin
  if Esvacia (Lc) or (P= nil) then
    Anterior:= nil
  else begin
    A:= Lc;
    while (A^.Sgte < > P) and (A^.Sgte < > Lc) do
      A :=A^.Sgte;
```

```

if A^.Sgte = P then
  Anterior := A
else
  Anterior := nil
end
end;

```

Localiza(X, Lc)

Función que devuelve la posición/dirección donde está el campo de información X. Si no está devuelve nulo.

```

function Localiza (X: Tipoinfo; Lc: Ptrnlc): Ptrnlc;
var
  T: Ptrnlc;
begin
  T := Lc;
  while (T^.Sgte < > Lc) and (T^.Info < > X) do
    T := T^.Sgte;
  if T^.Info < > X then
    Localiza := nil
  else
    Localiza := T
end;

```

Al igual que ocurre con listas doblemente enlazadas, una versión de Localiza es la operación Existe. Es planteada de tal forma que busca la existencia de un nodo, dada su dirección en una lista circular.

Existe(P, Lc)

```

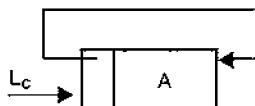
function Existe(P: Ptrnlc; Lc: Ptrnlc): boolean;
var
  T: Ptrnlc;
begin
  if not Esvacia(Lc) and (P < > nil) then
  begin
    T := Lc;
    while (Lc < > P) and (Lc^.Sgte < > T) do
      Lc := Lc^.Sgte;
    Existe := Lc = P
  end
  else
    Msge ('Error en llamada')
end;

```

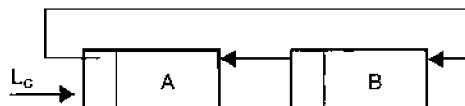
Inserta (X, P, Ld)

Añade a la lista Lc un nodo con el campo X, delante del nodo de dirección P. Para las operaciones de inserción escribimos la operación de crear un nodo para una lista circular.

```
function Crear(X: Tipoinfo): Ptrnlc;
var
  N: Ptrnlc;
begin
  new(N);
  N^.Info := X;
  N^.Sgte := N;
  Crear := N
end;
```



Al añadir un nuevo nodo:



```
procedure Insera(X:Tipoinfo; P:Ptrnlc; var Lc:Ptrnlc);
var
  A: Ptrnlc;
begin
  A:= Crear(X);
  if Esvacia(Lc) then
    Lc:= A
  else if Existe(P, Lc) then
    begin {Ahora es cuando se enlaza el nuevo nodo entre el nodo anterior y
           el nodo P}
    Anterior(P, Lc)^.Sgte := A;
    A^.Sgte := P;
  end
end;
```

Inserprim(X, Lc)

Añade un nodo con campo de información X como nodo que está a continuación del considerado el último, Lc. En el caso de que la lista esté vacía, crea la lista con dicho nodo.

```
Procedure Inserprim(X:Tipoinfo; var Lc: Ptrnlc);
var
  N: Ptrnlc;
begin
  N := Crear(X);
  if Esvacia(Lc) then
    Lc := N
  else begin
    N^.Sgte := Lc^.Sgte;
    Lc^.Sgte := N
  end
end;
```

Inserfin(X, Lc)

Añade un nodo con el campo X como último nodo de la lista Lc. De estar vacía crea la lista con dicho nodo. En cualquier caso, esta operación siempre devuelve la dirección del nuevo nodo: *el último*.

```

Procedure Inserfin(X:Tipoinfo: var Lc: Ptrnlc);
var
  N: Ptrnlc;
begin
  N := Crear(X);
  if not Esvacia(Lc) then
  begin
    N^.Sgte := Lc^.Sgte;
    Lc^.Sgte:= N
  end;
  Lc := N
end;

```

SuprimeDir(P, Lc)

Elimina de la lista el nodo cuya dirección/posición viene dada por P.

```

Procedure SuprimeDir(P: Ptrnlc; var Lc: Ptrnlc);
var
  N: Ptrnlc;
begin
  if not Esvacia (Lc) and Existe(P, Lc) then
  begin {Se dan las condiciones para borrar el nodo P}
    if Lc^.Sgte = Lc then {Lista se queda vacía}
      Lc := nil
    else if (P= Lc) then {El nuevo acceso a la lista será por el nodo
                           anterior a Lc}
      begin
        Lc := Anterior(P, Lc);
        Lc^.Sgte := P^.Sgte
      end
    else Anterior(P, Lc)^.Sgte := P^.Sgte;
    dispose(P)
  end
end;

```

Visualiza(Lc)

Recorre todos los nodos de una lista circular con el propósito de escribir el campo de información.

```

procedure Visualiza(Lc: Ptrnlc);
var
  A: Ptrnlc;
begin
  if not Esvacia(Lc) then
  begin
    A := Lc;
    repeat
      A := A^.Sgte;
      Escribir(A^.Info);
    until A = Lc;
  end
end;

```

6.6. IMPLEMENTACIÓN DE LISTAS CIRCULARES CON DOBLE ENLACE

Esta implementación de listas circulares con nodos que tienen dos punteros, permiten recorrer la lista circular en sentido del avance del reloj, o bien en sentido contrario. Los tipos de datos para realizar una lista circular con doble enlace son:

```
type
  Tipoinfo = string;
  PtrDnc = ^NodoDc;
  NodoDc = record
    Info : Tipoinfo;
    Sgte, Anter : PtrDnc
  end;
```

Las operaciones de inserción tienen que contemplar el doble enlace para formar la lista circular. Lo mismo a la hora de suprimir un nodo de la lista. La operación de recorrer la lista se puede hacer en dos direcciones o bien con el puntero `Sgte` o bien con el puntero `Anter`. A continuación se escribe el procedimiento que recorre la lista hacia adelante:

```
procedure Recorre_A (Ldc:PtrDnc);
var A:PtrDnc;
begin
  if not Esvacia (Ldc) then
  begin
    A:=Ldc;
    repeat
      A := A^.Sgte;
      Escribir(A^.Info)
    until A = Ldc;
  end
end;
```

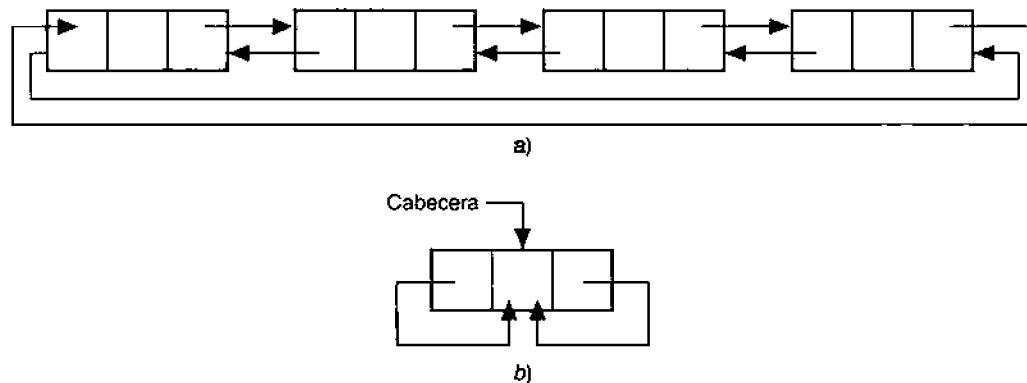


Figura 6.1. Lista circular doblemente enlazada con un nodo cabecera.

También se escribe el procedimiento para recorrer la lista desde el último nodo al «primero». Recordemos que Ldc referencia al nodo que por convenio consideramos último; se utiliza el puntero Anter.

```
procedure Recorre_D(Ldc:PtrDnc);
var D:PtrDnc;
begin
  if not Esvacia (Ldc) then
  begin
    D:=Ldc;
    repeat
      Escribir(D^.Info);
      D:=D^.Anter
    until D=Ldc
  end
end;
```

PROBLEMA 6.1. *Lista circular doblemente enlazada y ordenada*

Se trata de escribir un procedimiento que tenga como entrada una lista circular con doble enlace, siendo el campo info ordinal, y nos devuelva la lista ordenada.

En realidad se construye con dos procedimientos. El procedimiento ordenar recibe la lista, «rompe» la estructura circular, se convierte en una lista lineal y ordena la lista; el procedimiento ordencir compone la estructura circular. El nodo de la lista tiene como campo info valores enteros, y los dos campos puntero para enlazar que ahora se llaman izqdo y dcho.

```
type
  ptro=^nodo;
  nodo=record
    izdo,
    dcho:ptro;
    info:integer
  end;
var
  cab:ptro;

procedure ordenar (var cab:ptro);
var
  p,q,aux:ptro;
begin
  p:=cab;
  while p<>nil do
  begin
    begin
      q:=p^.dcho;
      while q<>nil do
        if p^.info <= q^.info then
          q:=q^.dcho
        else
          begin
            (*saca* de la lista el nodo q)
            q^.izdo^.dcho:=q^.dcho;
          end;
        end;
      p^.dcho:=q;
    end;
  end;
end;
```

```

        if q^.dcho<>nil then
            q^.dcho^.izdo:=q^.izdo;
            (añade q como anterior a p)
            aux:=q^.dcho;
            q^.dcho:=p;
            q^.izdo:=p^.izdo;
            p^.izdo:=q;
            if q^.izdo=nil then
                cab:=q
            else
                q^.izdo^.dcho:=q;
            p:=q;
            q:=aux
        end;
        p:=P^.dcho
    end
end;

procedure ordencir (var cl:ptro);
var
    aux:ptro;
begin
    if cl<>nil then
        begin
            aux:=cl^.izdo;
            aux^.dcho:=nil;
            cl^.izdo:=nil;
            ordenar (cl);
            aux:=cl;
            while aux^.dcho<>nil do
                aux:=aux^.dcho;
            aux^.dcho:=cl;
            cl^.izdo:=aux
        end
    end;
end;

```

PROBLEMA 6.2. Cadenas de caracteres mediante listas circulares

Dos cadenas de caracteres están almacenadas en dos listas circulares. Se accede a dichas listas mediante los punteros Lc1, Lc2. Escribir un subprograma que devuelva la dirección en la cadena Lc2 a partir de la cual se encuentra la cadena Lc1.

Solución

En primer lugar, se escribe una unidad que defina los tipos de datos necesarios y la función que resuelve el supuesto planteado. A esta unidad se la puede añadir otros procedimientos o funciones para crear listas circulares, visualizarlas...

```

unit Lista_ci
interface
type
    Tinfo = char;

```

```

Ptrlc = ^Nodolc;
Nodolc= record
  Info : Tinfo;
  Sgte : Ptrlc
end;
function Direccion(Lc1, Lc2:Ptrlc): Ptrlc;
implementation
function Direccion(Lc1, Lc2:Ptrlc): Ptrlc;
var
  A, A2, B: Ptrlc; Sw:boolean;
begin
  A := Lc2;
  A2:= Lc2;
  B := Lc1;
  repeat
    if A^.Info = B^.Info then
    begin
      A := A^.Sgte;Sw:=true;
      B := B^.Sgte
    end
    else
    begin
      A2:=A2^.Sgte;A:=A2;
      B := Lc1;Sw:=false
    end
  until (A = Lc2) or (B=Lc1)and Sw;
  if (B=Lc1 and Sw) then
    Direccion:= A2
  else
    Direccion := nil
  end;
end.

```

RESUMEN

Las listas doblemente enlazadas son aquel tipo de lista que se puede recorrer avanzando en cualquiera de los sentidos.

Otro tipo de lista muy eficiente es la lista circular, que es, por naturaleza, la que no tiene primero ni último nodo.

Aunque una lista enlazada circularmente tiene ventajas sobre una lista lineal pero presenta también algún inconveniente, como no poder recorrer la lista en sentido inverso. En estos casos la lista idónea es una *lista doblemente enlazada*, en la que cada nodo de la lista contiene los punteros, uno a su predecesor y otro a su sucesor.

Las listas doblemente enlazadas pueden ser o bien lineales o circulares y pueden o no contener un nodo de cabecera.

Los nodos de una lista doblemente enlazada constan de tres campos: un campo *info*, que contiene la información almacenada en el nodo, y los campos *izquierdo* y *derecho*, que contienen punteros a los nodos de cada lado.

EJERCICIOS

- 6.1. Dibujar una lista doblemente enlazada, L_d , de nombres de personas con un nodo cabecera. Escribir el procedimiento *Lista_vacía* que inicializa L_d como vacía.
- 6.2. Escribir la función *Esvacia* que devuelve cierto si la lista L_d con nodo de encabezamiento es vacía.
- 6.3. Dada la lista doble con nodo de encabezamiento L_d , escribir el procedimiento de insertar un nodo antes del nodo de dirección P .
- 6.4. En la lista doble con nodo de encabezamiento borrar el nodo que tiene como campo de información X .
- 6.5. Dada una lista doble sin nodo de encabezamiento cuya dirección de acceso L_d es el primer nodo, la información de cada nodo es una cadena de caracteres; escribir un procedimiento para visitar los nodos del primero (L_d) al último, convirtiendo las cadenas a mayúsculas; a continuación visite los nodos del último al primero (L_d) mostrando los nodos por pantalla.
- 6.6. Escribir un procedimiento que tenga como parámetro de entrada L_c . Suponer que L_c apunta al «primer» nodo de una lista circular doblemente enlazada de caracteres. El procedimiento debe de escribir los caracteres de la lista L_c en orden inverso, es decir, del último al primero.



- 6.7. Hacer lo mismo que en el ejercicio 6.6 pero con una lista simplemente enlazada.
- 6.8. Dibujar una lista circular doblemente enlazada con nodo de cabecera. El nodo de cabecera es tal que sus dos campos puntero referencian a los nodos extremos de la lista.
- 6.9. Dada la representación de lista circular propuesta en 6.8, escribir las operaciones *Lista_vacía* y la función *Esvacia*.
- 6.10. Con la representación de lista circular de 6.8, escribir las operaciones de insertar como «último» nodo e insertar como «primer» nodo.
- 6.11. Con la representación de lista circular de 6.8, escribir la función de localizar el nodo con el campo de información X y la operación de eliminar el nodo con campo de información X .

PROBLEMAS

- 6.1. Dada una lista doblemente enlazada de números enteros, escribir las rutinas necesarias para que dicha lista esté ordenada en orden creciente. La ordenación se ha de hacer intercambiando los punteros a los nodos.
- 6.2. Tenemos una lista doblemente enlazada ordenada con claves repetidas. Realizar un procedimiento de inserción de una clave en la lista, de tal forma que si la clave ya se encuentra en la lista la inserte al final de todas las que tienen la misma clave.
- 6.3. En una lista simplemente enlazada L se encuentran nombres de personas ordenados alfabéticamente. A partir de dicha lista L crear una lista doblemente enlazada LL de tal forma que el puntero de comienzo de la lista esté apuntando a la posición central. Damos por supuesto que la posición central es el nodo que ocupa la posición $n/2$, siendo n el número de nodos de la lista. Obviamente, los nodos que se encuentran a la derecha de la posición central están ordenados ascendente y los que se encuentran a la izquierda ordenados de manera decrecientemente.

- 6.4. Dada una cadena de caracteres almacenada en una lista circular Lc, escribir un subprograma que transforme la cadena Lc de tal forma que no haya caracteres repetidos.
- 6.5. En el archivo LIBROS se encuentran almacenados los datos relativos a la biblioteca municipal de Lupiana. Los campos de que consta cada registro son: Autor, Título, Número de ejemplares. Escribir un programa que realice como primera acción formar una lista doblemente enlazada ordenada respecto al campo Autor, cuya dirección de acceso sea el nodo que ocupe la posición central. Con la lista se podrán realizar estas otras operaciones:
- Mostrar todos los libros de un Autor.
 - Prestar un ejemplar de un libro, designado por Autor y Título.
 - Añadir un nuevo libro a la lista.
 - Dar de baja a todos los libros de un autor.
- Al finalizar el proceso deberá guardarse en el archivo los libros actuales.
- 6.6. Escribir un programa para realizar operaciones con vectores dispersos. Un vector disperso es aquel cuyo número de elementos es grande, sin embargo muchos de esos elementos son cero. La representación se ha de hacer mediante una lista doblemente enlazada. Cada nodo de la lista ha de tener un elemento del vector distinto de cero junto al índice del elemento. El programa debe de permitir estas operaciones:
- Dado un vector disperso, representarlo en una lista doble.
 - Dados dos vectores mediante sendas listas L1, L2, obtener el vector suma L1+L2. Podrá haber nuevas posiciones que sean cero.
 - Dados dos vectores L1, L2, obtener el vector diferencia (L1-L2). Tener en cuenta que puede haber nuevas posiciones que sean cero.
- 6.7. El polinomio $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ deseamos representarlo en una lista circular enlazada, de tal forma que cada nodo contenga el coeficiente y el grado de un monomio. Escribir un programa que tenga como entrada los coeficientes y exponentes de cada término de un polinomio, y forme una lista circular para representarlo. El puntero de acceso a la lista circular será el del nodo que contiene al término de grado n , a partir de él se accederá al término de grado 0, y así sucesivamente. En el programa deben de encontrarse las operaciones:
- Evaluación del polinomio para un valor dado de x .
 - Obtención del polinomio derivada de $P(x)$.
 - Obtención del polinomio producto de dos polinomios.
- 6.8. Se desea sumar enteros muy largos utilizando listas circulares doblemente enlazadas. Escribir una unidad donde se encuentren los tipos de datos y las operaciones para manejo de listas circulares dobles. Escribir un programa que permita la entrada de enteros largos, realice la suma y los muestre por pantalla.
- 6.9. Escribir un programa para sumar enteros largos utilizando listas circulares con doble enlace y un nodo de encabezamiento (según se indica en los ejercicios 6, 8, 9, 10 y 11). Previamente escribir la unidad con tipos de datos y operaciones para manejo de listas circulares con nodo de encabezamiento.

Pilas: el TAD Pila

CONTENIDO

- 7.1. Especificación formal del tipo abstracto de datos Pila.
- 7.2. Implementación del TAD Pila con arrays.
- 7.3. Implementación del TAD Pila mediante variables dinámicas.
- 7.4. Evaluación de expresiones aritméticas mediante pilas.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Una *pila* es una estructura de datos en la que todas las inserciones y eliminaciones de elementos se realizan por un extremo denominado *cima* de la pila. Una analogía es una pila de platos o una pila de cajas.

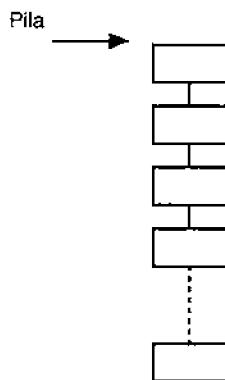
La implementación de una pila se puede realizar mediante arrays o con punteros. El inconveniente de la implementación de una pila con un array es que su tamaño máximo se debe especificar en tiempo de compilación. Para resolver este inconveniente, la implementación de una pila se ha de realizar con punteros (*apuntadores*).

El desarrollo de las pilas como tipos abstractos de datos es también otro de los motivos centrales de este capítulo. En el mismo se verá cómo utilizar el TAD Pila para resolver problemas de diferentes tipos.

7.1. ESPECIFICACIÓN FORMAL DEL TIPO ABSTRACTO DE DATOS PILA

Una *pila* es una lista ordenada de elementos en la que todas las inserciones y supresiones se realizan por un mismo extremo de la lista. En una pila el último elemento añadido es el primero en salir de la pila. Por esa razón a las pilas, se las denomina también *listas*.

Lifo (*Last input first output*, «último en entrar, primero en salir»). En la mente se tiene la imagen intuitiva de una pila. Así, si nos referimos a una pila de platos, sabemos que los platos se toman por «arriba», «por la cabeza». En la vida cotidiana se encuentran infinitud de ejemplos; así a veces se dice que «apilamos» los libros de un curso...



Las pilas crecen y decrecen dinámicamente, es una estructura de datos dinámica. En cuanto a la representación de una pila utilizando las estructuras de datos que tiene el lenguaje, existen varias formas. Al ser una pila una colección ordenada de elementos, y ser los arrays también una colección ordenada de elementos se tiene una forma de representar las pilas. En definitiva, una primera representación se realiza mediante la estructura estática array.

Una segunda forma de representar una pila es con listas enlazadas. Las listas crecen y decrecen dinámicamente, al igual que ocurre en una pila. Es una representación dinámica que utiliza punteros y variables dinámicas. Las operaciones básicas que definen el TAD Pila son las siguientes:

Pilavacia(P)	Crea una pila sin elementos.
Esvacia(P)	Devuelve verdadero si la pila P no tiene elementos.
Cima(P)	Devuelve el elemento que está en la cima de la pila.
Suprime(P)	Elimina el elemento que está en la cima de la pila.
Sacar(X, P)	Devuelve el elemento cabeza y lo suprime.
Meter(X, P)	Añade el elemento X en la pila.

7.2. IMPLEMENTACIÓN DEL TAD PILA CON ARRAYS

Un array constituye el depósito de los elementos de la pila. El rango del array debe ser lo suficientemente amplio para poder contener el máximo previsto de elementos de la pila. Un extremo del array se considera el *fondo* de la pila, que permanecerá fijo. La parte

superior de la pila, *cima*, estará cambiando dinámicamente durante la ejecución del programa. Además del array, una variable entera nos sirve para tener en todo momento el índice del array que contiene el elemento *cima*. Las declaraciones, procedimientos y funciones para representar el TAD pila forman parte de la unidad pilas.

```
const
  Maxelems = 100; {Dependerá de cada realización}
type
  Indicepila = 0..Maxelems;
  Tipoelemen = ...{Tipo de los elementos de la pila}
  Tipopila = record
    Elementos: array[1 .. Maxelems] of Tipoelemen;
    Cab: Indicepila
  end;
```

En función de esta representación se codifican las operaciones básicas definidas en TAD *pila*: *Pilavacia*, *Esvacia*, *Cima*, *Suprimir*, *Sacar*, *Meter* y *Pilallena*.

Pilavacia(P)

```
procedure Pilavacia(var P: Tipopila);
begin
  P.Cab:= 0
end;
```

Esvacia(P)

```
function Esvacia(P: Tipopila): boolean;
begin
  Esvacia:= P.Cab=0
end;
```

Cima(P)

Esta función devuelve el elemento cabeza de la pila, sin modificar la pila.

```
function Cima(P: Tipopila): Tipoelemen;
begin
  if not Esvacia(P) then
    Cima := P.Elementos[P.Cab]
end;
```

Suprime(P)

Esta operación elimina el elemento cabeza, modificando la pila.

```
procedure Suprime(var P: Tipopila);
begin
  if not Esvacia(P) then
    P.Cab:= P.Cab- 1
end;
```

Sacar(X, P)

Esta operación devuelve el elemento cabeza y lo suprime.

```
procedure Sacar(var X:Tipoelemen; var P:Tipopila);
begin
  if not Esvacia(P) then
  with P do
  begin
    X:= Elementos[Cab];
    Cab:= Cab- 1
  end
end;
```

Meter(X, P)

Esta operación añade el elemento X a la pila. Cuando se añade un elemento se dice que éste es empujado dentro de la pila.

Al estar representando la pila mediante una estructura estática puede ocurrir que no haya posiciones libres en el array. Por esta circunstancia se ha de incorporar una operación Plena que devuelve verdadero si se ha alcanzado el número máximo de elementos previstos.

```
function Plena(P: Tipopila): boolean;
begin
  Plena:= P.Cab=Maxelems
end;

procedure Meter(X: Tipoelemen; var P: Tipopila);
begin
  if not Plena(P) then
  with P do
  begin
    Cab:= Cab+ 1;
    Elementos[Cab]:= X
  end
end;
```

PROBLEMA 7.1. Utilización de una pila

Se desea invertir una cadena de caracteres y se trata de determinar si han sido invertidos correctamente dichos caracteres. Para ello se lee la cadena de caracteres; a continuación, en la siguiente línea los caracteres de la cadena invertidos.

La estrategia a seguir es sencilla, los caracteres de la cadena se almacenan en una pila. Según se van leyendo los caracteres de la cadena invertida se comparan con el carácter cima de la pila; de este modo se aprovecha la principal característica de una

pila: último en entrar primero en salir. El último carácter introducido debe ser igual que el primer carácter de la cadena invertida.

```

function Esinversa: boolean;
var
  Ch: char;
  Pila: Tipopila;
  Inv: boolean;
begin
  Pilavacia(Pila);
  writeln('Caracteres de la cadena: ');
  repeat
    read(Ch);
    Meter(Ch, Pila)
  until eoln;
  readln;
  {Los caracteres leídos son comparados con la pila}
  Inv:= true;
  while not Esvacia(Pila) and Inv do
  begin
    read(Ch);
    Inv:= Ch= Cima(Pila);
    Suprimir(Pila)
  end;
  {Consideramos que si hay más caracteres que en la pila, no ha
   sido invertida correctamente}
  Esinversa:= Inv and eoln;
  readln
end;

```

Para sacar los elementos de la pila se puede seguir la alternativa de utilizar otra variable carácter D y la operación Sacar:

```

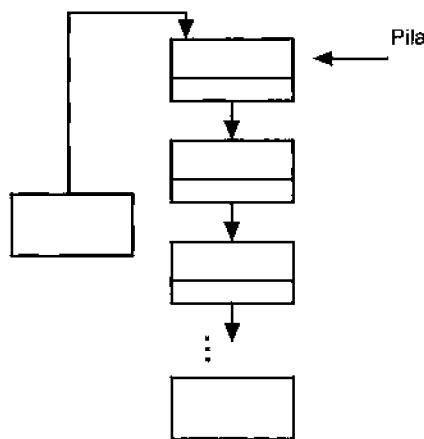
while not Esvacia(Pila) and Inv do
begin
  read(Ch);
  Sacar(D,Pila)
  Inv:= Ch = D;
end;

```

7.3. IMPLEMENTACIÓN DEL TAD PILA MEDIANTE VARIABLES DINÁMICAS

La implementación dinámica de una pila se hace almacenando los elementos como nodos de una lista enlazada, con la particularidad de que siempre que se quiera meter (*empujar*) un elemento se hará por el mismo extremo que se extraerá.

Esta realización tiene la ventaja de que el tamaño se ajusta exactamente a los elementos de la pila. Sin embargo, para cada elemento es necesaria más memoria, ya que hay que guardar el campo de enlace. En la realización con arrays hay que establecer un máximo de posibles elementos, aunque el acceso es más rápido, ya que se hace con una variable subindicada.



Los tipos de datos y operaciones en la realización con listas son muy similares a los ya expuestos en listas enlazadas *Pilavacia*, *Esvacia*, *Cima*, *Suprime*, *Sacar* y *Meter*.

```

type
  Tipoelemen= ... {Tipo de los elementos de la pila}
  Ptrpla= ^NodoPla;
  Nodopla= record
    Elemento: Tipoelemen;
    Enlace: PtrPla
  end;
  
```

Pilavacia(crear pila)

Crea una pila sin elementos.

```

procedure Pilavacia(var Pila: Ptrpla);
begin
  Pila:= nil
end;
  
```

Esvacia

Función que determina si la pila no tiene elementos.

```

function Esvacia(Pila: Ptrpla): boolean;
begin
  Esvacia:= Pila= nil
end;
  
```

Cima(Pila)

Devuelve el elemento cabeza de la pila, sin modificar la pila.

```

function Cima(Pila: Ptrpla): Tipoelemen;
begin
  if not Esvacia(Pila) then
    Cima:= Pila^.Elemento
  end;

Suprime(Pila)

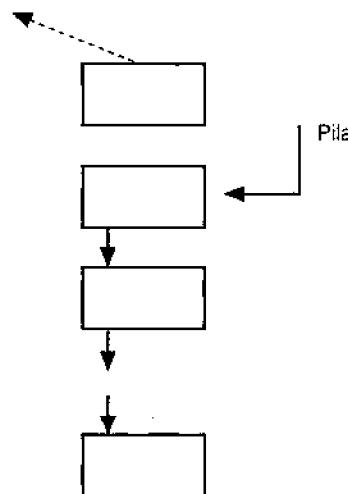
```

Con esta operación se elimina y libera el elemento cabeza, modificando la pila.

```

procedure Suprime(var Pila: Ptrpla);
var
  Q: Ptrpla;
begin
  if not Esvacia(Pila) then
  begin
    Q:= Pila;
    Pila:= Pila^.Enlace;
    dispose(Q)
  end
end;

```



```
Sacar(X, Pila)
```

Esta operación engloba a las dos anteriores. Devuelve el elemento cabeza y lo suprime.

```

procedure Sacar(var X:Tipoelemen; var Pila: Ptrpla);
var
  Q: Ptrpla;

```

```

begin
  if not Esvacia(Pila) then
    with Pila^ do
      begin
        Q:= Pila;
        X:= Elemento;
        Pila:= Enlace;
        dispose(Q)
      end
    end;
end;

```

El procedimiento se puede escribir llamando a Cima y Suprimir.

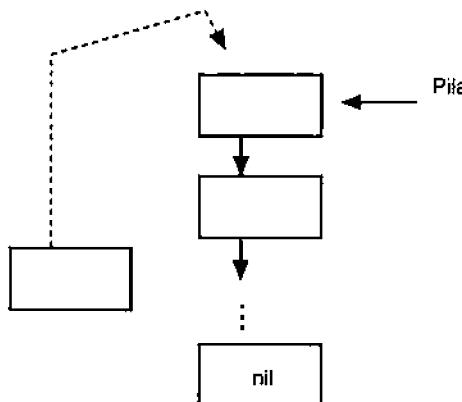
```

procedure Sacar(var X:Tipoelemen; var Pila: Ptrpla);
begin
  X:= Cima(Pila);
  Suprime(Pila)
end;

Meter(X, Pila)

```

Esta operación añade el elemento X a la pila. En esta representación con estructuras dinámicas no tiene sentido la operación pila llena, la pila crece o decrece según lo necesita.



```

procedure Meter(X: Tipoelemen; var Pila: Ptrpla);
var
  Q: Ptrpla;
begin
  new(Q);
  Q^.Elemento:= X;
  Q^.Enlace:= Pila;
  Pila:= Q
end;

```

7.4. EVALUACIÓN DE EXPRESIONES ARITMÉTICAS MEDIANTE PILAS

Una de las aplicaciones más típicas del **TAD pila** es almacenar los caracteres de que consta una expresión aritmética con el fin de evaluar el valor numérico de dicha expresión. Una *expresión aritmética* está formada por operandos y operadores. Así, la expresión

$$R = X * Y - (A + B)$$

está escrita de la forma habitual: el operador en medio de los operando. Se conoce como *notación infija*. Conviene recordar que las operaciones tienen distintos niveles de precedencia.

Paréntesis	:	()	<i>nivel mayor de prioridad</i>
Potencia	:	^	
Multipl/división	:	*, /	
Suma/Resta	:	+, -	<i>nivel menor de prioridad</i>

También suponemos que a igualdad de precedencia son evaluados de izquierda a derecha.

7.4.1. Notaciones Prefija (Polaca) y Postfija (Polaca inversa)

La forma habitual de escribir operaciones aritméticas es situar el operador entre sus dos operandos con la citada notación *infija*. Esta forma de notación obliga en muchas ocasiones a utilizar paréntesis para indicar el orden de evaluación.

$$A * B / (A + C)$$

$$A * B / A + C$$

Representan distintas expresiones al no poner paréntesis. Igual ocurre con las expresiones:

$$(A - B) ^ C + D$$

$$A - B ^ C + D$$

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 estudió).

$$\begin{aligned} A * B / (A + C) & \text{ (infija)} \rightarrow A * B / + A C \rightarrow * A B / + A C \rightarrow / * A B + A C \text{ (polaca)} \\ A * B / A + C & \text{ (infija)} \rightarrow * A B / A + C \rightarrow / * A B A + C \rightarrow + / * A B A C \text{ (polaca)} \\ (A - B) ^ C + D & \text{ (infija)} \rightarrow - A B ^ C + D \rightarrow ^ - A B C + D \rightarrow + ^ - A B C D \text{ (polaca)} \end{aligned}$$

Podemos observar que no es necesario la utilización de paréntesis al escribir la expresión en notación polaca. La propiedad fundamental de la notación polaca es que el orden en que se van a realizar las operaciones está determinado por las posiciones de los operadores y los operandos en la expresión.

Otra forma de escribir las operaciones es mediante la notación *postfija* o *polaca inversa* que coloca el operador a continuación de sus dos operandos.

$A * B / (A + C)$ (infija) $\rightarrow A * B / A C + \rightarrow AB^* / AC + \rightarrow AB^* AC + /$ (polaca inversa)
 $A * B / A + C$ (infija) $\rightarrow AB^* / A + C \rightarrow AB^* A / + C \rightarrow AB^* A / C +$ (polaca inversa)
 $(A - B) ^ C + D$ (infija) $\rightarrow AB^- ^ C + D \rightarrow AB - C ^ + D \rightarrow AB - C ^ D +$ (polaca inversa)

7.4.2. Algoritmo para evaluación de una expresión aritmética

A la hora de evaluar una expresión aritmética escrita, normalmente, en notación infija la computadora sigue dos pasos:

- 1.^o Transformar la expresión de infija a postfija.
- 2.^o Evaluar la expresión en postfija.

En el algoritmo para resolver cada paso es fundamental la utilización de pilas.

Se parte de una expresión en notación infija que tiene operandos, operadores y puede tener paréntesis. Los operandos vienen representados por letras, los operadores van a ser:

* (potenciación), $*$, $/$, $+$, $-$.

La transformación se realiza utilizando una pila en la que se almacenan los operadores y los paréntesis izquierdos. La expresión se va leyendo carácter a carácter, los operandos pasan directamente a formar parte de la expresión en postfija.

Los operadores se meten en la pila siempre que esta 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 izquierdo siempre se meten en la pila con la mínima prioridad. 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 ítems de la expresión y la pila esté vacía. Sea por ejemplo la expresión infija $A * (B + C - (D / E ^ F) - G) - H$, la expresión en postfija se va ir formando con la siguiente secuencia:

Expresión	Estado de la Pila
A	Carácter A a la expresión; carácter * a la pila.
AB	Carácter (a la pila; carácter B a la expresión.
ABC	Carácter + a la pila; carácter C a la expresión.

En este momento el estado de la pila es



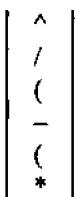
El nuevo carácter leído es «-», que tiene igual prioridad que el elemento cima de la pila «+»; en consecuencia, el estado de la pila es:



y la expresión es:

ABC +
ABC + D
ABC + DE
ABC + DEF

Carácter (a la pila; carácter D a la expresión.
Carácter / a la pila; carácter E a la expresión.
Carácter ^ a la pila; carácter F a la expresión.
Carácter) (paréntesis derecho) provoca vaciar la pila hasta un (.
La pila en este momento contiene



El nuevo estado de la pila es



y la expresión

ABC + DEF ^ /
ABC + DEF ^ / -
ABC + DEF ^ / - G
ABC + DEF ^ / - G - *
ABC + DEF ^ / - G - * H
ABC + DEF ^ / - G - * H -

Carácter - a la pila y se extrae a su vez -;
Carácter G a la expresión; carácter), son extraídos de la pila los operadores hasta un (la pila queda *
Carácter -, se saca de la pila * y se mete -
Carácter H se lleva a la expresión
Fin de entrada, se vacía la pila:

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 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 el mismo pero metido en la pila. Las prioridades son determinadas según esta tabla:

Operador	Prioridad dentro pila	Prioridad fuera pila
$^$	3	4
$* \cdot /$	2	2
$+ \cdot -$	1	1
(0	5

Obsérvese que no se trata el paréntesis derecho ya que éste provoca sacar operadores de la pila hasta el paréntesis izquierdo. El algoritmo de paso de notación infija a postfija:

1. Obtener caracteres de la expresión y repetir los pasos 2 al 4 para cada carácter.
2. Si es 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, sacar cima de la pila y pasarlo a la expresión postfija, volver a 3.
4. Si es paréntesis derecho:
 - 4.1. Sacar cima de pila y pasarlo a 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.

7.5. APLICACIÓN PRÁCTICA DE LA EVALUACIÓN DE UNA EXPRESIÓN ARITMÉTICA

Los tipos de datos, procedimientos y funciones utilizados para codificar el algoritmo tratado en el apartado anterior se realiza en las unidades Pilaop y ExpPost.

7.5.1. Unidad Pilaop

La unidad Pilaop contiene los tipos básicos para el manejo de las pilas, así como sus operaciones asociadas.

```
unit Pilaop;
interface
type
  Plaopr = ^Nodopr;
  Nodopr= record
```

```

Info: char;
Sgte: Plaopr
end;

function Pvacia(P: Plaopr ): boolean;
procedure Pcrear(var P: Plaopr );
procedure Pmeter(Ch: char; var P: Plaopr );
procedure Psacar(var Ch: char; var P: Plaopr );
function Pcima(P:Plaopr):char; {devuelve el elemento cima de la pila}
procedure Pborrar(var P: Plaopr );

implementation

function Pvacia(P: Plaopr): boolean;
begin
  Pvacia:= P=nil
end;

procedure Pcrear(var P: Plaopr );
begin
  P:=nil
end;

procedure Pmeter(Ch: char; var P: Plaopr );
var
  A: Plaopr;
begin
  new(A);
  A^.Info := Ch;
  A^.Sgte := P;
  P := A
end;

procedure Psacar(var Ch: char; var P: Plaopr );
begin
  Ch:= Pcima(P);
  Pborrar(P)
end;

function Pcima(P:Plaopr):char; {devuelve elemento cima de la pila}
begin
  if not Pvacia(P) then
    Pcima:= P^.Info
end;

procedure Pborrar(var P: Plaopr );
var
  A: Plaopr ;
begin
  if not Pvacia(P) then
  begin
    A:=P;
    P:=P^.Sgte;
    dispose(A)
  end;
end;
end.

```

7.5.2. Unidad ExpPost (transformación a postfija)

La unidad ExpPost realiza la transformación de la expresión infija a postfija, utilizando para ello la unidad Pilaop definida anteriormente.

```

unit ExpPost;
interface
uses crt, Pilaop;
type
  Treg= record
    C: char;
    Oprdor: boolean
  end;
  Tag = array[1..100] of Treg;
  {Tipo para almacenar la expresión}
  Oprdos = array['A'..'Z'] of Real;
  {para evaluar la expresión}
function Valido(C: char): boolean;
function Operando(C: char): boolean;
function Pdp(P: Pilaopr ): integer;
function Pfp(op: char): integer;
procedure Postfija(var Ar: Tag; var J: integer);
implementation

function Valido(C: char): boolean;
begin
  Valido:= C in ['A'..'Z', 'a'..'z', '^', '/', '*', '+', '-', '#', '(', ')'];
end;

function Operando(C: char): boolean;
begin
  Operando:= C in ['A'..'Z', 'a'..'z']
end;
{Prioridad del operador dentro de la Pila}
function Pdp(P: Pilaopr ): integer;
var
  op: char;
begin
  if not Pvacia(P) then
  begin
    OP:= Pcima(P);
    case op of
      '^': Pdp:= 3;
      '*', '/': Pdp := 2;
      '+', '-': Pdp := 1;
      '(': Pdp      := 0;
    end
  end
  else {Pila está vacía, es asignada la mínima prioridad}
    Pdp:= -1
end;
{Prioridad del operador en la expresión infija}

function Pfp(Op: char): integer;
begin

```

```

case Op of
  '^': Pfp := 4;
  '*', '/': Pfp := 2;
  '+', '-': Pfp := 1;
  '('': Pfp := 5;
end;
end;

{En este procedimiento se lee la expresión en notación infija. El
carácter que arbitrariamente determina el fin de la expresión es #.
La expresión en postfija es almacenada en el vector Ar}

procedure Postfija(var Ar: Tag; var J: integer);
var
  P: Plaopr ;
  It, Ch: char;
  Desapila: boolean;
begin
  Ch:=' ';
  J:= 0;
  Pcrear(P);
  while Ch <> '#' do
    begin
      repeat
        read(Ch);
        Ch:=upcase(Ch)
      until Valido(Ch);
      if operando(Ch) then {Pasa directamente a la expresión postfija}
      begin
        J:= J+1;
        Ar[J].oprdo:=false;
        Ar[J].C:= Ch
      end
      else if Ch<>'#' then {es operador}
        if Ch < > ')' then
          begin
            desapila:=true;
            while desapila do
              if Pvacia(P) or (Pfd(Ch)> Pdp(P)) then
              begin
                Pmeter(Ch, P);
                desapila := false
              end
              else if Pfd(Ch)< Pdp(P) then {hay que dasapilar}
              begin
                Psacar(It, P);
                J:= J+1;
                Ar[J].C:= It; {Ar en postfija}
                Ar[J].oprdo := true
              end
            end
            else begin {es un ')'}
              Psacar(It, P);
              repeat
                J:= J+1;
                Ar[j].c:=It;
                Ar[J].oprdo:=true;

```

```

        Psacar(It, P);
        until It=';';
    end;
end;
repeat
    Psacar(It, P);
    J:= J+1;
    Ar[J].C:= It;
    Ar[J].oprdo:= true;
    until Pvacia(P);
end; {del procedure}
end.

```

7.5.3. Evaluación de la expresión en postfija

En un vector ha sido almacenada la expresión aritmética en notación postfija. Los operandos están representados por variables de una sola letra. La primera acción que va a ser realizada es dar valores numéricos a los operandos.

Una vez que tenemos 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 Pf es el vector que contiene la expresión. El número de elementos de que consta la expresión es n .

1. Examinar Pf desde el elemento 1 hasta el n. Repetir los pasos 2 y 3 para cada elemento de Pf.
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 llamamos X e Y, respectivamente.
 - Evaluar Y & X, el resultado es Z = Y & X.
 - El resultado Z, meterlo en la pila.
 Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento cima de la pila.
5. Fin del algoritmo.

Codificación de evaluación de expresión en postfija

Las operaciones para manejar la pila de operandos están encapsuladas en la unidad Pilaopdos. Los nombres de las operaciones han sido cambiados para distinguirlas de la unidad Pilaop. La parte de implementation es la misma, por ello nos limitaremos a escribir la interfaz.

```

unit Pilaopdos;
interface
type
  Plaopdos = ^Nodopdos;
  Nodopdos= record
    Info: real;
    Sgte: Plaopdos
  end;

```

```

function Povacia(P: Plaopdos ): boolean;
procedure Pocrear(var P: Plaopdos );
procedure Pometer(var P: Plaopdos ;Ch: char);
procedure Posacar(var P: Plaopdos ;var Ch: char);
function Pocima(P: Plaopdos): char;
{devuelve el elemento cima de la pila}
procedure Poborrar(var P: Plaopdos );
{Fin de la sección de interface. La sección implementation es igual que
en la unidad Pilaop}

```

Ahora ya se puede escribir el programa completo que realiza todo el proceso. El programa `Evalua_Expresion` utiliza las unidades `Crt`, `ExpPost` y `PilaOpdos`; contiene el procedimiento `Leer_oprdos` que lee el valor numérico de los operandos, así como el procedimiento `Evalua` que implementa el algoritmos de evaluación.

```

program Evalua_Expresion;
uses
  Crt, ExpPost, PilaOpdos;
var
  V: Oprdos;
  T: Tag;
  I, J:integer;
  Valor: real;
procedure Leer_oprdos(E:Tag;N:integer; var V: Oprdos);
{En este procedimiento se asignan valores numéricos a los operandos}
var
  K: integer;
  Ch: char;
begin
  K:= 0;
  repeat
    K:= K+1;
    if not E[K].Oprdor then {Es un operando, petición de su valor
      numéricico}
      begin
        Ch:= E[K].C;
        write(Ch, '=');
        readln(V[Ch])
      end
  until K=N
end;

procedure Evalua(Pf: Tag; N: integer;
                Oper: Oprdos; Var Valor: real);
var
  Pila: Plaopdos;
  I: integer;
  Num1, Num2: real;
  Op: char;
begin
  I:= 0;
  Pocrear(Pila);
  repeat
    I:=I+1

```

```

if not Pf[I].Oprdor then
  Pometer(Oper{Pf[I].C}, Pila)
else if Pf[I].Oprdor then
begin
  Posacar(Num2, Pila);
  Posacar(Num1, Pila);
  case Pf[I].C of
    '^':Valor:= Exp(Num1*Ln(Num2)) (Potencia)
    '**':Valor:= Num1*Num2;
    '/':Valor:= Num1/Num2;
    '+':Valor:= Num1+Num2;
    '-':Valor:= Num1-Num2
  end;
  Pometer(Valor, Pila)
end;
until I= N
end;
begin
  clrscr;
  writeln('Expresión aritmética(termina #)');
  Postfija(T, J);
  writeln('Asignación de valores numéricos a los operandos');
  writeln ;
  Leer_oprdos(T, J, V);
  Evalua(T, J, V, Valor);
  writeln;
  write('Resultado de evaluación de la expresión: ', Valor);
end.

```

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 se insertan y eliminan por el mismo extremo, que se denomina *cima* de la pila. El proceso de inserción se denomina *meter* o *poner* y el de eliminación se denomina *extraer*, *quitar* o *sacar*.

Añadir un elemento a una pila se llama operación *meter* o *poner* (*push*) y eliminar un elemento de una pila es la operación *extraer*, *quitar* o *sacar* (*pop*).

El intento de poner un elemento en una pila llena produce un error conocido como *desbordamiento de la pila* (*stack overflow*). El intento de sacar un elemento de una pila vacía produce un error conocido como *desbordamiento negativo de la pila* (*stack underflow*).

Una pila puede ser implementada mediante un *array* o mediante una *lista enlazada*. Una ventaja de una implementación de una pila mediante una *lista enlazada* sobre un *array* es que, con la lista enlazada no existe límite previo en el número de elementos o entradas que se pueden añadir a la pila.

Las llamadas a procedimientos recursivos se implementan utilizando una pila (de hecho, todas las llamadas a procedimientos, sean o no recursivas, se implementan utilizando una pila).

El TAD *Pila*, al igual que el TAD *Lista* y TAD *Cola* son, posiblemente, los tipos abstractos de datos más utilizados en la gestión de proyectos software, dado que la estructura tipo pila es muy frecuente en numerosos problemas de tratamiento de información, así como en problemas matemáticos, estadísticos o financieros. Por ejemplo, se puede utilizar una pila para determinar si

una secuencia de vuelos existe entre dos ciudades. La pila mantiene la secuencia de ciudades visitadas y permite a un algoritmo de búsqueda volver hacia atrás. Para ello se sitúa la ciudad origen en la parte inferior de la pila, y la ciudad destino en la parte superior, con lo que las sucesivas ciudades de tránsito se van colocando en la secuencia correcta.

EJERCICIOS

- 7.1. Se desea implementar el TAD pila de tal forma que los elementos de la pila sean almacenados en una lista circular. El puntero externo a la lista apunta al elemento cabeza. Escribir las operaciones de meter, cima, suprimir y esvacia.
- 7.2. Supongamos que estamos trabajando en un lenguaje de programación que no tiene el tipo de dato puntero. Se plantea la resolución de un problema utilizando dos pilas de números reales. Las dos pilas se quiere guardar en un único array, de tal forma que crecen en sentido contrario, una de ellas crece desde la posición 1 del array y la otra desde la última posición del array. Escribir la estrategia a seguir para esta representación, variables necesarias para contener la posición del elemento cabeza de ambas pilas.
- 7.3. Dada la representación de dos pilas propuesta en 7.2, escribir el algoritmo de la operación `CreaPila(Pila1, Pila2)` que inicializan ambas pilas como pila vacía. De igual forma escribir las funciones `Esvaciar1` y `Esvaciar2` que determinan si las respectivas `Pila1` y `Pila2` están vacías.
- 7.4. Completar las operaciones de las pilas propuestas en 7.2: `Meter1`, `Meter2` y `Suprimir1`, `Suprimir2`. En las operaciones hay que tener en cuenta condiciones de error.
- 7.5. Utilizando una pila de caracteres, hacer un seguimiento para transformar la siguiente expresión infija a su equivalente expresión en postfija.

$$E: (X-Y)/Z^*W+(Y-Z)^*V$$

- 7.6. Aplicando el algoritmo de evaluación de una expresión, obtenga el resultado de las siguientes expresiones escritas en postfija.

- $XY+Z-YX+Z^-$
- $XYZ+*ZYX-+*$

para $X=1$, $Y=3$, $Z=2$

PROBLEMAS

- 7.1. Escribir un programa para determinar si frases (cadenas) son palíndromos. Utilizar para ello el TAD *pila* de caracteres realizado en una unidad. La entrada de datos son las frases y la salida la propia frase junto a la etiqueta «ES PALÍNDROMO», o bien «NO ES PALÍNDROMO».
- 7.2. Escribir un programa que convierta expresiones escritas en notación infija (forma habitual) a notación postfija. Cada expresión se encuentra en una línea, el programa ha de terminar cuando la línea conste de 3 asteriscos.
- 7.3. Escribir un programa que convierta expresiones escritas en notación postfija a notación infija. Cada expresión se encuentra en una línea, el programa ha de terminar cuando la línea conste de 3 asteriscos.

- 7.4. Escribir un programa que convierta expresiones escritas en notación postfija a notación prefija. Cada expresión se encuentra en una linea, el programa ha de terminar cuando la línea conste de 3 asteriscos.

- 7.5. Un mapa de carreteras podemos representarlo mediante la matriz simétrica MM de $N \times N$ elementos enteros, donde los valores l a N representan los pueblos/ciudades que aparecen en el mapa.

Los elementos de la matriz son tales que $MM(i,j) = 0$ si no hay conexión directa entre el pueblo i y el pueblo j . $MM(i,j) = d$ si hay conexión directa entre el pueblo i y el pueblo j , y su distancia es d .

Con esta representación del mapa queremos escribir un programa que simule el mapa descrito y que tenga como entrada dos pueblos (origen, destino) entre los que no hay conexión directa; decida si hay un camino que pase por los pueblos del mapa y determine la distancia de ese camino. Utilizar una pila para ir almacenando los pueblos que van formando el camino recorrido y poder volver atrás si se alcanza un pueblo desde el que no se puede proseguir la ruta, y probar con otra ruta.

- 7.6. Utilizando únicamente las operaciones básicas sobre pilas: Meter, Cima, Suprime y Esvacia, construir las operaciones que realicen las siguientes acciones:

- Asignar a X el segundo elemento desde la parte superior de la pila, dejando la pila sin sus dos elementos de la parte superior.
- Asignar a X el segundo elemento desde la parte superior de la pila, sin modificarla.
- Dado un entero positivo n , asignar a X el n -ésimo elemento desde la parte superior de la pila, dejando la pila sin sus n elementos de la parte superior.
- Dado un entero positivo n , asignar a X el n -ésimo elemento desde la parte superior de la pila, sin modificarla.
- Asignar a X el elemento fondo de la pila, dejando la pila vacía.
- Asignar a X el elemento fondo de la pila, sin modificarla.

- 7.7. Utilizando las operaciones del TAD pila, escribir una operación de copia, tal que devuelva la copia de una pila.

- 7.8. 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, siendo $\&$ el carácter separador.

- 7.9. Escribir una función para determinar si una secuencia de caracteres de entrada es de la forma:

$A \# B \# C \# \dots$

Donde cada una de las cadenas A , B , C ... son de la forma $X \& Y$, que a su vez estarán separadas por el carácter $\#$.

Colas y colas de prioridades: el TAD cola

CONTENIDO

- 8.1. Especificación formal del tipo abstracto de datos cola.
- 8.2. Implementación del TAD cola con arrays lineales.
- 8.3. Implementación del TAD cola con arrays circulares.
- 8.4. Implementación del TAD cola con listas enlazadas.
- 8.5. Implementación del TAD cola con listas circulares.
- 8.6. Bicolas.
- 8.7. Colas de prioridades.
- 8.8. Implementación de colas de prioridades.
- 8.9. Un problema complejo resuelto con colas de prioridades.
- 8.10. Problema: Suma de enteros grandes.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

El concepto de *cola* es uno de los que más abundan en la vida cotidiana. Espectadores esperando en la taquilla de un cine o de un campo de fútbol; clientes de un supermercado esperando la compra de un artículo comercial; etc. En una aplicación informática, una *cola* es una lista en la que todas las inserciones a la lista se realizan por un extremo, y todas las eliminaciones o supresiones de la lista se realizan por el otro extremo.

Las colas se llaman también estructuras FIFO (*first-in, first-out*; primero en entrar, primero en salir). Las aplicaciones de las colas son numerosas en el mundo de la computación: colas de las tareas a realizar por una impresora, acceso a *almacenamiento de disco*, o incluso, en sistemas de tiempo compartido, el uso de la UCP (Unidad Central de Proceso). En el capítulo se examinan las operaciones básicas de manipulación de los tipos de datos *cola*.

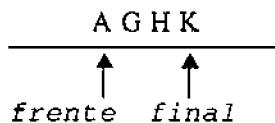
8.1. ESPECIFICACIÓN FORMAL DEL TIPO ABSTRACTO DE DATOS COLA

Una cola es una lista ordenada de elementos, en la cual las eliminaciones se realizan en un solo extremo, llamado *frente* o *principio de la cola*, y los nuevos elementos son añadidos por el otro extremo, llamado *fondo* o *final de la cola*. La Figura 8.1 muestra una estructura cola en una organización original (*a*) y sus modificaciones (*b* y *c*) después de eliminar y añadir un elemento de modo sucesivo.

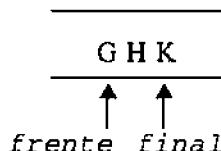
En esta estructura de datos el primer elemento que entra es el primero en salir. Por esta causa a las colas también se les llama listas **FIFO** (*first input first output*). Las operaciones básicas que definen la especificación del TAD cola son:

<i>Qcrear(Q)</i>	Crea la cola Q como cola vacía.
<i>Qvacia(Q)</i>	Nos devuelve cierto si la cola está vacía.
<i>Frente(Q)</i>	Devuelve el elemento <i>frente</i> de la cola.
<i>Qborrar(Q)</i>	Elimina el elemento <i>frente</i> de la cola.
<i>Qanula(Q)</i>	Convierte la cola en vacía.
<i>Quitar(x,Q)</i>	Elimina y devuelve el <i>frente</i> de la cola.
<i>Qponer(x,Q)</i>	Añade un nuevo elemento a la cola.

a) Estructura original



b) Estructura después de eliminar un elemento



c) Estructura después de añadir un elemento

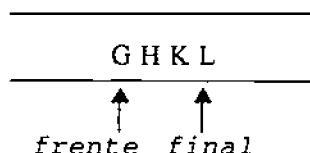


Figura 8.1. Una estructura de datos tipo cola.

Al igual que las pilas, la implementación de colas puede hacerse utilizando como «depósito» de los elementos un array o bien una lista enlazada y dos punteros a los extremos.

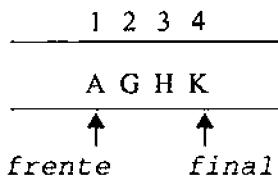
8.2. IMPLEMENTACIÓN DEL TAD COLA CON ARRAYS LINEALES

La forma más sencilla de representación de una cola es mediante arrays lineales. Se utiliza como depósito de los elementos de la cola un array unidimensional cuyo tipo es el mismo que el tipo de los elementos de la cola. Son necesarios dos índices para referenciar al elemento frente y al elemento final. El array y las dos variables índice se agrupan en el tipo registro de nombre cola.

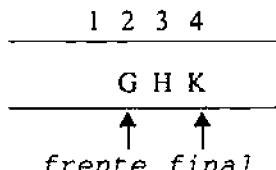
```
const
  Max = 100;
type
  Tipoelemen = ...;           // definición del tipo de elemento
  Posicion= 0..Max;
  Cola= record
    Datos: array[1.. Max] of Tipoelemen;
    Frente,
    Final: Posicion
  end;
var Q:Cola;
```

Un array es una estructura estática y por tanto tiene dimensión finita; por el contrario, una cola puede crecer y crecer sin límite, y en consecuencia se puede presentar la posibilidad de que se presente un desbordamiento. Por esa razón dentro de las operaciones de este TAD se incorpora, normalmente, la operación de verificación de que la cola está llena.

La operación de añadir un nuevo elemento en la cola comienza a partir de la posición 1 del array



Supongamos que la cola se encuentra en la situación anterior: el frente está fijo y el final de la cola es el que se mueve al añadir nuevos elementos. Si ahora se quita un elemento (evidentemente por el frente) la cola queda así:



Se ha dejado un hueco no utilizable, por lo que se desaprovecha memoria.

Una alternativa a esta situación es mantener fijo el frente de la cola al comienzo del array; este hecho supone mover todos los elementos de la cola una posición cada vez que se quiera retirar un elemento de la cola.

Estos problemas quedan resueltos con los arrays circulares que se describen más adelante. Las operaciones más importantes incluidas en la especificación de una cola con esta representación son: *Crear*, *ColaVacia*, *ColaLlena*, *Añadir* y *Quitar*.

Crear (Qcrear(Q))

La operación *Qcrear* inicializa la cola como cola vacía.

```
procedure Qcrear(var Q: Cola);
begin
  Q.Frente:= 1;
  Q.Final:= 0;
end;
```

Cola vacía (Qvacia(Q))

La operación *Qvacia* devuelve verdadero si la cola no tiene elementos.

```
function Qvacia(Q: Cola): boolean;
begin
  Qvacia:= Q.Final < Q.Frente
end;
```

Cola llena (Qllena(Q))

La operación *Qllena* devuelve verdadero si en la cola no pueden añadirse más elementos.

```
function Qllena(Q: Cola): boolean;
begin
  Qllena := Q.Final = Max
end
```

Añadir (Qponer(X,Q))

Añade el elemento X a la cola. Se modifica la cola por el extremo final.

```
procedure Qponer(X:Tipoelemen; var Q:Cola);
begin
  if not Qllena(Q) then
    with Q do
      begin
        Final:= Final+1;
        Datos [Final]:= X
      end
  end;
```

Quitar (Quitar(X,Q))

Elimina y devuelve el frente de la cola.

```
procedure Quitar(var X: Tipoelemen; var Q: Cola);
  procedure Desplazar;
  var I: Posicion;
```

```

begin
  for I:=1 to Q.Final-1 do
    Q.Datos[I]:=Q.Datos[I+1]
  end;
begin
  X:= Q.Datos[Frente];
  Desplazar
end;

```

8.3. IMPLEMENTACIÓN DEL TAD COLA CON ARRAYS CIRCULARES

La implementación de colas mediante un array lineal es poco eficiente. Cada operación de supresión de un elemento supone mover todos los elementos restantes de la cola.

Para evitar este gasto, imagínese un array como una estructura circular en la que al último elemento le sigue el primero. Esta representación implica que aun estando ocupado el último elemento del array, pueda añadirse uno nuevo detrás de él, ocupando la primera posición del array.

Para *añadir un elemento* a la cola, se mueve el índice final una posición en el sentido de las manecillas del reloj, y se asigna el elemento. Para *suprimir un elemento* es suficiente con mover el índice frente una posición en el sentido del avance de las manecillas del reloj. De esta manera, la cola se mueve en un mismo sentido, tanto si se realizan inserciones como supresiones de elementos.

Según se puede observar en la representación, la condición de cola vacía [frente = siguiente (final)] va a coincidir con una cola que ocupa el círculo completo, una cola que llene todo el array.

Para resolver el problema, una primera tentativa sería considerar que el elemento final referencia a una posición adelantada a la que realmente ocupa el elemento, en el sentido del avance del reloj.

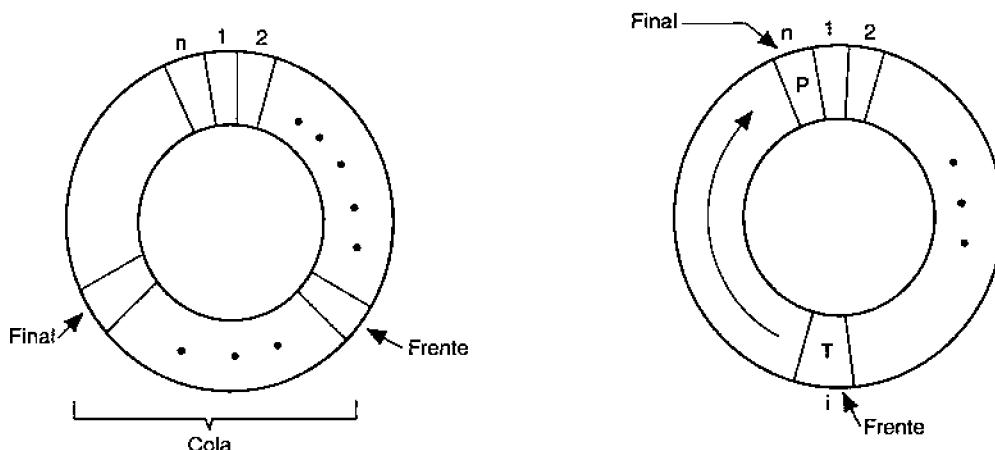
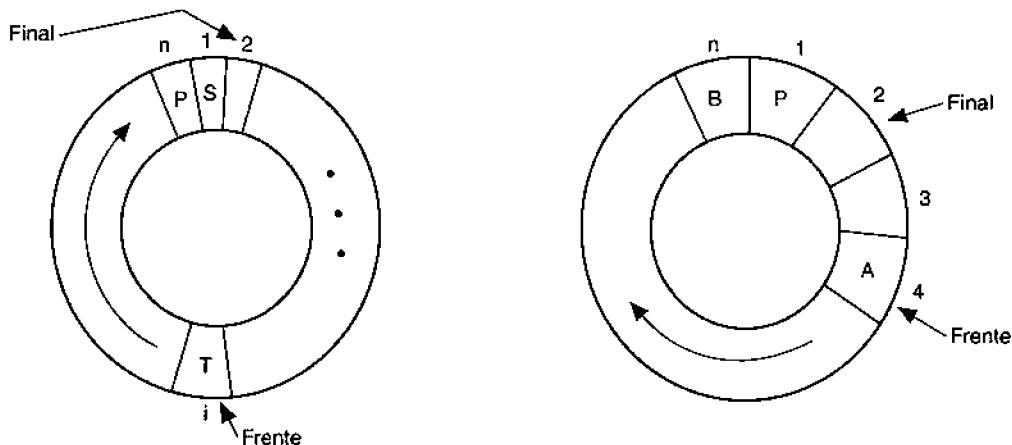


Figura 8.2. Arrays circulares.



Teniendo presente esta consideración cuando la cola estuviera llena, el índice siguiente a final será igual al frente.

Consideremos ahora el caso en que queda un solo elemento en la cola. Si en estas condiciones se suprime el elemento, la cola queda vacía, el puntero *Frente* avanza una posición en el sentido de las manecillas del reloj y va a referenciar a la misma posición que el siguiente al puntero *Final*. Es decir, está exactamente en la misma posición relativa que ocuparía si la cola estuviera llena.

Una solución a este problema es sacrificar un elemento del array y dejar que *Final* refiera a la posición realmente ocupada por el último elemento añadido. Si el array tiene *long* posiciones, no se debe dejar que la cola crezca más que *long*-1.

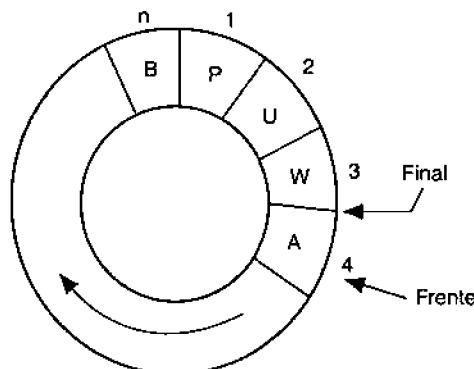


Figura 8.4. Inserción de nuevos elementos en un array circular.

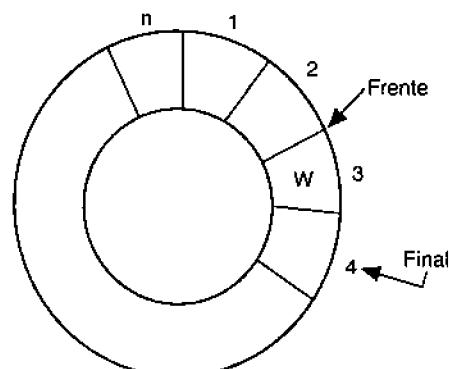


Figura 8.5. Supresión de un elemento en un array circular.

Según lo expuesto anteriormente, las declaraciones necesarias para especificar el tipo cola son:

```
const
  Long = ... ;
type
  Tipoelemen = ...;
  Posicion= 0..Long;
  Cola= record
    Elementos: array[1.. long] of Tipoelemen;
    Frente,
    Final: Posicion
  end;
```

Las operaciones definidas en el TAD cola con esta representación de array circular son: *Siguiente*, *Crear*, *ColaVacia*, *ColaLlena*, *Frente*, *Borrar* y *Quitar*.

Siguiente (J)

La función *Siguiente (J)* obtiene la posición siguiente a *J* en el sentido circular de avance, según las manecillas del reloj.

```
function Siguiente(J: integer): integer;
begin
  Siguiente:= (J mod Long) +1
end;
```

Crear (Q)

La operación de *crear* inicializa los índices de la cola de tal manera que la condición de cola vacía aplicada sobre la cola creada sea cierta.

```
procedure Qcrear(var Q: Cola);
begin
  Q.Frente:= 1;
  Q.Final:= Long;
end;
```

Cola vacía

La operación *Qvacía* devuelve verdadero si la cola no tiene elementos.

```
function Qvacía(Q: Cola): boolean;
begin
  Qvacía:= Siguiente(Q.Final)= Q.Frente
end;
```

Cola llena

La operación *Qllena* devuelve verdadero si en la cola no pueden añadirse más elementos, es decir, si están ocupadas *Long-1* posiciones del array.

```
function Qllena(Q: Cola): boolean;
begin
  Qllena := Siguiente(Siguiente(Q.Final)) = Q.Frente
end;
```

Frente (Frente(Q))

```
function Frente(Q: Cola): Tipoelemen;
begin
  if not Qvacia(Q) then
    Frente:= Q.Elementos[Q.Frente]
  end;
```

Borrar (Qborrar(Q))

Esta operación modifica la cola, elimina el elemento frente de la cola.

```
procedure Qborrar(var Q: Cola);
begin
  if not Qvacia(Q) then
    Q.Frente:= Q.Frente+1
  end;
```

Quitar (Quitar(X, Q))

Elimina y devuelve el frente de la cola. En realidad esta operación es un compendio de las operaciones Frente y Qborrar.

```
procedure Quitar(var X: Tipoelemen; var Q: Cola);
begin
  X:= Frente(Q);
  Qborrar(Q)
end;
```

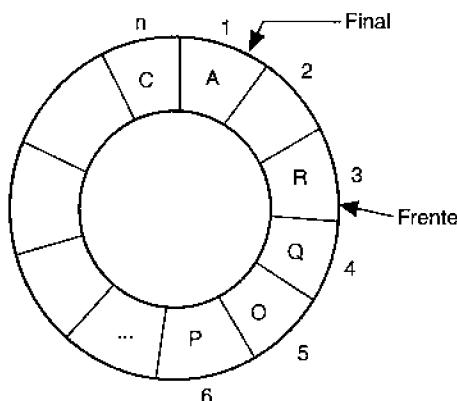


Figura 8.6. Llenado del array circular.

Qponer(X, Q)

Añade el elemento X a la cola. Se modifica la cola por el extremo final.

```
procedure Qponer(X:Tipoelemen; var Q:Cola);
begin
  if not Qllena(Q) then
    with Q do
    begin
      Final:= Siguiente(Final);
      Elementos[Final]:= X
    end
  end;
end;
```

8.4. IMPLEMENTACIÓN DEL TAD COLA CON LISTAS ENLAZADAS

Como ocurre con toda representación estática, una de las principales desventajas es que hay que prever un máximo de elementos, de ese máximo no podemos pasarnos. La realización de una cola mediante una lista enlazada permite ajustarse exactamente al número de elementos de la cola.

Esta implementación con listas enlazadas utiliza dos punteros para acceder a la lista. El puntero *Frente* y el puntero *Final*.

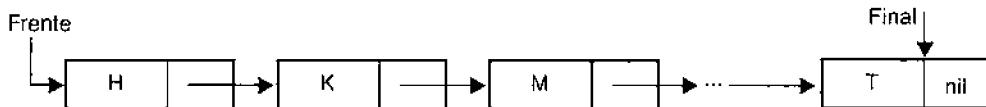


Figura 8.7. Cola implementada con lista enlazada.

El puntero *Frente* referencia al primer elemento que va a ser retirado. El puntero *Final* referencia al último nodo que fue añadido, en definitiva, al último que será retirado (lista Fifo). En esta representación no tiene sentido la operación que determina si una cola está llena (*Qllena*). Al ser una estructura dinámica crece y decrece según las necesidades. Las declaraciones para la representación de cola mediante listas:

```
type
  Tipoelemen = ...;
  Ptnodoq= ^Nodoq;
  Nodoq= record
    Info: Tipoelemen;
    Sgte: Ptnodoq
  end;
  Cola= record
    Frente,
    Final:Ptnodoq
  end;
```

Las operaciones definidas en la especificación del tipo en esta estructura son: *Crear*, *ColaVacia*, *Frente*, *Borrar*, *Quitar* y *Poner*.

Crear (Qcrear(Q))

La operación *Qcrear* inicializa la cola como Cola vacía.

```
procedure Qcrear(var Q: Cola);
begin
  Q.Frente:= nil;
  Q.Final:= nil
end;
```

Cola vacía (Qvacia(Q))

La operación *Qvacia* devuelve verdadero si la cola no tiene elemento.

```
function Qvacia(Q: Cola): boolean;
begin
  Qvacia:= Q.Frente= nil
end;
```

Frente (Frente(Q))

Devuelve el elemento *Frente* de la cola.

```
function Frente(Q: Cola): Tipoelemen;
begin
  if not Qvacia(Q) then
    Frente:= Q.Frente^.Info
  end;
```

Borrar (Qborrar(Q))

Esta operación modifica la cola, elimina el elemento frente de la cola y es liberado.

```
procedure Qborrar(var Q: Cola);
var
  A: Ptrnodoq;
begin
  if not Qvacia(Q) then
    with Q do
      begin
        A:= Frente;
        Frente := Frente^.Sgte;
        if Frente = nil Then
          Final:= nil;
        dispose(A)
      end
  end;
end;
```

Quitar (Quitar(X,Q))

Elimina y devuelve el frente de la cola.

```
procedure Quitar(var X:Tipoelemen; var Q:Cola);
begin
  X := Frente(Q);
  Qborrar(Q)
end;
```

Poner (Qponer(X,Q))

Añade el elemento X a la cola. Este elemento se añade como último nodo, por lo que se modifica la cola al cambiar *Final*.

```
function Crear (X:Tipoelemen): Ptrnodoq;
var T:Ptrnodoq;
begin
  new(T); T^.Info:= X; T^.Sgte:= nil
  Crear:= T
end;

procedure Qponer(X: Tipoelemen; var Q: Cola);
var
  N: Ptrnodoq;
begin
  N:= Crear(X); {Crea nodo con campo X, devuelve su dirección}
  with Q do
  begin
    if Qvacia(Q) then
      Frente:= N;
    else
      Final^.Sgte:= N;
    Final:= N
  end
end;
```

8.5. IMPLEMENTACIÓN DEL TAD COLA CON LISTAS CIRCULARES

Esta implementación es una variante de la realización con listas enlazadas. Al realizar una lista circular se estableció, por conveniencia, que el puntero de acceso a la lista referenciaba al último nodo, y que el nodo siguiente se considera el primero. 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*, y la eliminación por el nodo siguiente. Las declaraciones de tipos para esta representación:

```
type
  Tipoelemen = ...;
  Cola= ^Nodoq;
  Nodoq= record
    Info: Tipoelemen;
    Sgte: Cola
  end;
```

Las operaciones son similares a las realizadas con listas enlazadas: *Crear*, *ColaVacia*, *Frente*, *Borrar*, *Quitar* y *Poner*. El puntero necesario para realizar inserciones y supresiones es el mismo: *Q*. Teniendo en cuenta que *Q* referencia al último (*Final*) y *Q^.Sgte* al primero (*Frente*).

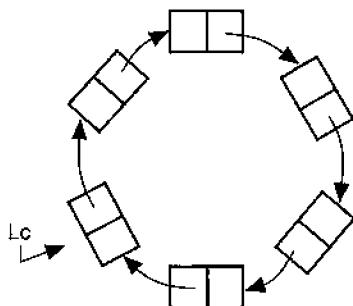


Figura 8.8. Cola implementada con una lista circular con punteros.

```

Crear (Crear(Q))
procedure Qcrear(var Q: Cola);
begin
  Q:= nil;
end;

Cola vacía (Qvacia(Q))

function Qvacia(Q: Cola): boolean;
begin
  Qvacia:= Q= nil
end;

Frente(Q)

function Frente(Q: Cola): Tipoelemen;
begin
  if not Qvacia(Q) then
    Frente:= Q^.Sgte^.Info
end;

Qborrar(Q)

procedure Qborrar(var Q: Cola);
var
  A: Cola;
begin
  if not Qvacia(Q) then
  begin
    A:= Q^.Sgte;
    if Q^.Sgte= Q then
      Q:=nil;
    else
      Q^.Sgte:=Q^.Sgte^.Sgte;
    dispose(A)
  end
end;

```

```

Quitar(X, Q)
procedure Quitar(var X:Tipoelemen; var Q:Cola);
begin
  X := Frente(Q);
  Qborrar(Q)
end;

Oponer(X, Q)
procedure Oponer(X: Tipoelemen; var Q: Cola);
var
  N: Cola;
begin
  new(N);N^.Info:=X;N^.Sgte:=N;(Crea nodo con campo X)
  if not Qvacia(Q) then
  begin
    N^.Sgte:= Q^.Sgte;
    Q^.Sgte:= N
  end
  Q:= N;
end;

```

8.6. BICOLAS

Una variación del tipo de datos cola es la estructura bicola. Una *bicola* es un conjunto ordinal de elementos en el cual se pueden añadir o quitar elementos de cualquier extremo de la misma. Es, en realidad, una *cola bidireccional*.

Los dos extremos de una bicola los llamaremos **Izquierdo** y **Derecho**, respectivamente. Las operaciones básicas que definen una bicola son:

- CrearBq (Bq):** inicializa una bicola sin elementos.
- Esvacia (Bq):** devuelve verdadero si la bicola no tiene elementos.
- InserIzq (x, Bq):** añade un elemento por extremo izquierdo.
- InserDch (x, Bq):** añade un elemento por extremo derecho.
- ElimnIzq (x, Bq):** devuelve el elemento Izquierdo y lo retira de la bicola.
- ElimnDch (x, Bq):** devuelve el elemento Derecho y lo retira de la bicola.

Para representar una bicola se puede elegir una representación estática, con arrays, o bien una representación dinámica, con punteros.

En la representación dinámica, la mejor opción es mantener la bicola con una lista doblemente enlazada: los dos extremos de la lista se representan con las variables puntero **Izquierdo** y **Derecho**, respectivamente.

En la representación estática se mantienen los elementos de la bicola con un array circular y dos variables índice del extremo izquierdo y derecho, respectivamente.

Al tipo de datos bicola se pueden imponer 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 permite la eliminación 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.

Frente:	3		LUZ	AGUA	RIO			
Final:	5	1	2	3	4	5	6	7

Al añadirse el elemento SOL por el frente

Frente:	2		SOL	LUZ	AGUA	RIO		
Final:	5	1	2	3	4	5	6	7

Puede eliminarse por el final

Frente:	2		SOL	LUZ	AGUA			
Final:	4	1	2	3	4	5	6	7

Figura 8.9. Estructura tipo Bicola.

PROBLEMA 8.1

El estacionamiento de las avionetas de un aeródromo es en linea, con una capacidad hasta 12 avionetas. Las avionetas llegan por el extremo izquierdo y salen por el extremo derecho. Cuando llega un piloto a recoger su avioneta, si ésta no está justamente en el extremo de salida (derecho), todas las avionetas a su derecha han de ser retiradas, sacar la suya y las retiradas colocadas de nuevo en el mismo orden relativo en que estaban. La salida de una avioneta supone que las demás se muevan 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) o salida (S) de avioneta. En la llegada puede ocurrir que el estacionamiento esté lleno, si es así la avioneta espera hasta que se quede una plaza libre, o hasta que se dé la orden de retirada (salida).

Análisis

El estacionamiento va a estar representado por una bicola de salida restringida. ¿Por qué? La salida siempre se hace por el mismo extremo, sin embargo la entrada se puede hacer por los dos extremos, y así contemplar la llegada de una avioneta nueva, y que tenga que entrar una avioneta que ha sido movida para que salga una intermedia. La línea de espera para entrada de una avioneta (*estacionamiento lleno*) se representa por una bicola a la que se añaden avionetas por un extremo, salen para entrar en el estacio-

namiento por el otro extremo, aunque pueden retirarse en cualquier momento si así lo decide el piloto; en este caso puede ocurrir que haya que retirar las avionetas que tiene delante, o bien si es la última retirarla por ese extremo.

Las avionetas que se mueven para poder retirar del estacionamiento una intermedia se disponen en una lista LIFO; así la última en entrar será la primera en añadirse en el extremo salida del estacionamiento y seguir en el mismo orden relativo.

La unidad `lifo(pila)` se describe sólo con `interface` ya que su implementación se ha escrito anteriormente. En la unidad `bicola` se implementan las operaciones de una bicola, la representación con array circular, además se incorpora la operación que da las plazas libres que hay en el estacionamiento. La restricción de salida se refleja en que no es válida la operación `Rem_ent` (retirada por el extremo de salida).

```
unit Lifo;
interface
type
  Avioneta=string[15]; {En realidad sería un registro con más campos
                        identificativos}
  PtrPila =^Itemp;
  Itemp = record
    Info: Avioneta;
    Sgte: PtrPila
  end;
procedure Pilavacia(var Pila: PtrPila);
function Pvacia(Pila: PtrPila): boolean;
procedure Pmeter(X: Avioneta; var Pila: PtrPila);
procedure Pcima(var X: Avioneta; Pila: PtrPila);
procedure Pborrar(var Pila: PtrPila);
procedure Psacar(var X: Avioneta; var Pila: PtrPila);
{Fin de la sección de interfaz}
```

La unidad `Bicolas` utiliza la lista LIFO:

```
unit Bicolas;
interface
uses Lifo;
const
  Long = 13;
type
  Avioneta = string[15];
  Posicion= 0..Long;
  BiCola= record
    Elementos: array[1.. Long] of Tipoelemen;
    Salida,
    Entrada: Posicion
  end;
procedure CrearBq (var Bq: BiCola);
function Esvacia(Bq: BiCola): boolean;
function EsLLena(Bq: BiCola): boolean;
procedure InsEntrada(A:Avioneta;var Bq:BiCola);
procedure InsSalida(A:Avioneta;var Bq:BiCola);
procedure EliEntrada(var A:Avioneta;var Bq:BiCola);
procedure EliSalida(var A:Avioneta;var Bq:BiCola);
```

```
procedure Retirar(A:Avioneta;var Bq:Bicola;var Ok:boolean);
function Libres(Bq:Bicola):integer;
implementation
function Siguiente(J: integer): integer;
begin
  Siguiente:= (J mod Long) +1
end;

function Anterior(J: integer): integer;
begin
  if (J-1)=0 then
    Anterior:= Long
  else
    Anterior:=J-1
end;

function Posicion(A:Avioneta;Bq:Bicola): integer;
{Determina posición que ocupa la avioneta A tomando como inicio de
búsqueda Salida}
var
  P:integer;
  Sw:boolean;
begin
  Sw:=false; P:=Bq.Salida;
  repeat
    Sw:=Bq.Elementos[P]=A;
    if not Sw then
      P:=Siguiente(P);
  until Sw or P
  if Sw then
    Posicion:=P
  else
    Posicion:=0
end;

function Libres(Bq:Bicola):integer;
begin
  Libres:=abs(Bq.Salida-Siguiente(Bq.Entrada))
end;

procedure CrearBq {var Bq: BiCola};
begin
  Bq.Entrada:= long-1;
  Bq.Salida:= Long;
end;

function Esvacia(BQ: Cola): boolean;
begin
  Esvacia:= Siguiente(BQ.Entrada)= BQ.Salida
end;

function EsLLena(BQ: Cola): boolean;
begin
  EsLLena:=Siguiente(Siguiente(BQ.Entrada))= BQ.Salida
end;
```

```

procedure InsEntrada(A:Avioneta;var Bq:Bicola);
begin
  if not EsLlena(Bq) then
    with Bq do
    begin
      Entrada:= Siguiente(Entrada);
      Elementos[Entrada]:= A
    end
  end;
end;

procedure InsSalida(A:Avioneta;var Bq:Bicola);
begin
  if not EsLlena(Bq) then
    with Bq do
    begin
      Salida:= Anterior(Salida);
      Elementos[Salida]:= A
    end
  end;
end;

procedure EliEntrada(var A:Avioneta;var Bq:Bicola);
begin
  if not Esvacia(Bq) then
    begin
      A:=BQ.Elementos[Bq.Entrada];
      Bq.Entrada:= Anterior[Bq.Entrada]
    end
  end;
end;

procedure EliSalida(var A:Avioneta;var Bq:Bicola);
begin
  if not Esvacia(Bq) then
    begin
      A:=BQ.Elementos[Bq.Salida];
      Bq.Entrada:= Siguiente[Bq.Salida]
    end
  end;
end;

procedure Retirar(A:Avioneta;var Bq:Bicola;var Ok:boolean);
{Esta operación retira una avioneta. O bien por un extremo, o bien
intermedia}
var
  D: integer;
  Pl:Ptrpila;
begin
  D:=Posicion(A,Bq);
  Ok:=true;
  if D<>0 then
    if D=Bq.Salida then
      EliSalida(A,Bq)
    else begin
      Pilavacia(Pl);
      {Se sacan elementos intermedios, desde Salida hasta D}
      repeat
        Elisalida(A,Bq);
        Pmeter(A,Pl);
      until D=Bq.Salida;
    end;
  end;
end;

```

```

until Bq.Salida=D;
ElSalida(A,Bq); {es retirado el elemento pedido}
{Son introducidos los elementos sacados}
repeat
  Psacar(A,P1);
  InsSalida(A,Bq)
  until Pvacia(P1)
end
else
  Ok:=false; {No está la avioneta}
end;
{fin de la unidad bicola}
end.

```

Por último, el programa que emula el estacionamiento es Avionetas.

```

program Avionetas(input,output);
uses crt,Lifo,Bicola;
var
  U: Avioneta;
  Ch: char;
  Bq:Bicola;
  Bqw:Bicola; {Bicola de espera de entrada en estacionamiento}
  Sw:boolean;
begin
  clrscr;
  CrearBq(Bq); CrearBq(Bqw);
  write('La entrada interactiva. son líneas con un char y ');
  writeln('número matrícula:E(llegada),S(Salida)');
  repeat
    repeat
      read(Ch);Ch:=uppercase(Ch)
      until Ch in['E', 'S'];
      readln(U);
      if Ch = 'E' then
        if not Esllena(Bq) then
          InsEntrada(U,Bq)
        else
          EnsEntrada(U,Bqw); {se añade a bicola de espera}
      else {retirada de avionetas}
        Retirar(U,Bq,Sw)
      Until not Sw
    end.

```

8.7. COLAS DE PRIORIDADES

El término *cola* sugiere la forma en que esperan ciertas personas u objetos la utilización de un determinado servicio. Por otro lado, el término *prioridad* sugiere que el servicio no se proporciona únicamente aplicando el concepto de cola (el primero en llegar es el primero en ser atendido) sino que cada persona tiene asociada una prioridad basada en un criterio objetivo.

Un ejemplo típico de organización formando colas de prioridades, es el sistema de tiempo compartido necesario para mantener un conjunto de procesos que esperan

servicio para trabajar. Los diseñadores de estos sistemas asignan cierta prioridad a cada proceso.

El orden en que los elementos son procesados y por tanto eliminados sigue estas reglas:

1. *Se elige la lista de elementos que tienen la mayor prioridad.*
2. *En la lista de mayor prioridad, los elementos se procesan según el orden de llegada; en definitiva, según la organización de una cola: primero en llegar, primero en ser procesado.*

Las colas de prioridades pueden implementarse de dos formas: mediante una única lista o bien mediante una lista de colas.

8.8. IMPLEMENTACIÓN DE COLAS DE PRIORIDADES

Para realizar la representación, se define en primer lugar el tipo de dato que representa un proceso.

```
type
  Tipo_identif=...{Tipo del identificador del proceso}
  Tipo_proceso= record
    Idt: Tipo_identif;
    Prioridad: integer
  end;
```

8.8.1. Implementación mediante una única lista

Cada proceso forma un nodo de la lista enlazada. La lista se mantiene ordenada por el campo prioridad.

La operación de añadir un nuevo nodo hay que hacerla siguiendo este criterio: *La posición de inserción es tal que la nueva lista ha de permanecer ordenada. A igualdad de prioridad se añade como último en el grupo de nodos de igual prioridad.* De esta manera la lista queda organizada de tal forma que un nodo X precede a un nodo Y si:

1. *Prioridad(X) > Prioridad(Y).*
2. *Ambos tienen la misma prioridad, pero X se añadió antes que Y.*

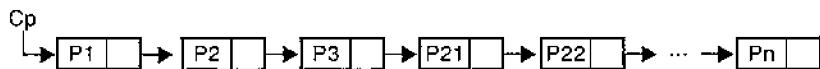


Figura 8.10. Cola de prioridad con una lista enlazada.

Los números de prioridad tienen el significado habitual: *a menor número mayor prioridad*. Esta realización presenta como principal ventaja que es inmediato determinar el siguiente nodo a procesar: siempre será el primero de la lista. Sin embargo, añadir un nuevo elemento supone encontrar la posición de inserción dentro de la lista, según el criterio expuesto anteriormente.

8.8.2. Implementación mediante una lista de n colas

Se utiliza una cola separada para cada nivel de prioridad. Cada cola puede representarse con un array circular, mediante una lista enlazada, o bien mediante una lista circular, en cualquier caso con su *Frente* y *Final*. Para agrupar todas las colas, se utiliza un array de registros. Cada registro representa un nivel de prioridad, y tiene el frente y el final de la cola correspondiente. La razón para utilizar un array radica en que los niveles de prioridad son establecidos de antemano.

La definición de los tipos de datos para esta realización:

```
const
  Max_prior=...; {Máximo número de prioridades previsto}
type
  Tipo_identif=...{Tipo del identificador del proceso}
  Tipo_proceso= record
    Idt: Tipo_identif;
    Prioridad: integer;
  end;
  Ptnodoq^Nodoq;
  Nodoq= record
    info: Tipo_proceso;
    sgte: Ptnodoq
  end;
```

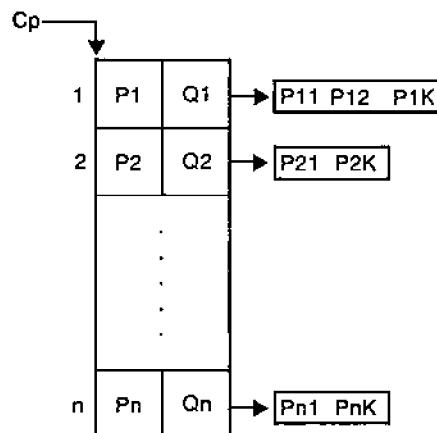


Figura 8.11. Cola de prioridad con n colas.

```

Cola = record
    Numprioridad: integer;
    frente,
    final: PTrnodoq
end;
Tabla_cp = array[1..Max_prior] of Cola;

```

El tipo Tabla_cp es el que define la lista de colas y cada cola representa una prioridad. Las acciones más importantes al manejar una cola de prioridades son las de *añadir* un nuevo elemento, con una determinada prioridad al sistema, y la acción de *retirar* un elemento para su procesamiento. Estas acciones se expresan en forma algorítmica en la realización de n colas, como se muestra a continuación.

Algoritmo para añadir nuevo elemento

Añade un elemento P que tiene un número de prioridad m.

1. Buscar el índice en la tabla correspondiente a la prioridad m.
2. Si existe y es K, poner el elemento P como final de la cola de índice K.
3. Si no existe, crear nueva cola y poner en ella el elemento P.
4. Salir.

Algoritmo para retirar un elemento

Retira el elemento de frente de la cola que tiene máxima prioridad.

1. Buscar el índice de la cola de mayor prioridad no vacía. Cola k.
2. Retirar y procesar el elemento frente de la cola k.
3. Salir.

8.9. IMPLEMENTACIÓN DE UN PROBLEMA CON COLAS DE PRIORIDADES

El objetivo del supuesto planteado es emular un sistema simple de procesamiento en tiempo compartido. Para ello, van a generarse tareas aleatoriamente con un número de prioridad de 1 a 8.

La simulación se va a realizar hasta que hayan sido finalizadas un número determinado de tareas (Max_proc). Además, cada vez que el sistema retira una tarea, ésta se procesa en su totalidad.

La cadencia es: *por cada tarea procesada llegan al sistema dos nuevas tareas, así hasta completar el proceso del máximo de tareas previstas.*

La realización es con un sistema de n colas. En la unidad colas están encapsuladas las operaciones de manejo de colas, los tipos de datos y los procedimientos para añadir y eliminar una tarea cuyos algoritmos se han expuesto anteriormente.

Unidad unitcola

```

unit unitcola;
interface
uses crt;
const
  Max_proc=100;
  Max_prior=8;
type
  Tipo_proc=record
    Tarea: string[20];
    Prior: integer
  end;
  Ptrndoq^Nodoq;
  Nodoq= record
    Info: Tipo_proc;
    sgte:Ptrndoq
  end;
  Colap= record
    Prd: integer; (Prioridad que representa la cola)
    Frente,
    Final: Ptrndoq
  end;
  Tablacolas =array [1..Max_prior] of Colap;

var
  LL: integer; (utilizada para presentación)
procedure Qcrear(var Q: Colap);
function Qvacia(Q: Colap): boolean;
procedure Qponer(X: Tipo_proc; var Q:Colap);
procedure Qborrar(var Q: Colap);
procedure Quitar(var X: Tipo_proc; var Q:Colap);
procedure CrearQp (var Tcp:Tablacolas; Max_prior:integer);
function Numcola(P: integer; var Tcp:tablacolas):integer;
function Qmaxpri(var Tcp:tablacolas): integer;
procedure presentar_proc (Q: Colap);
procedure generar_proc (var Tcp:Tablacolas;j:integer);
procedure procesar_proc(var Tcp:Tablacolas; var Enco:boolean);

implementation

function Qvacia (Q: Colapp): boolean;
begin
  Qvacia:= Q.Frente:= nil
end;

procedure Qcrear(var Q: Colap);
begin
  Q.Frente:= nil;
  Q.Final:= nil
end;

procedure Qponer(X: Tipo_proc; var Q:Colap);
var
  A: Ptrnodoq;

```

```

begin
  new(A);
  A^.Info:=X;
  A^.Sgte:=nil;
  if Qvacia(Q) then
    Q.Frente:= A
  else
    Q.Final^.Sgte:=A;
  Q.Final:= A
end;

procedure Qborrar(var Q: Colap);
var
  A:Ptrnodoq;
begin
  if (not Qvacia(Q))then
  begin
    A:=Q.Frente;
    Q.Frente:=Q.Frente^.Sgte;
    dispose(A)
  end
end;

procedure Quitar(var X: Tipo_proc; var Q:Colap);
begin
  if (not Qvacia(Q)) then
    X:= Q.Frente^.Info;
  Qborrar(Q)
end;

procedure CrearQP (var Tcp:Tablacolas;Max_prior:integer);
var
  i:integer;
begin
  for i:=1 to Max_prior do
  begin
    Qcrear(Tcp[i]);
    Tcp[i].Prd:= 0
  end
end;

function Numcola(P: integer; var Tcp:tablacolas): integer;
{Busca cola de prioridad P; Si no Existe devuelve la siguiente libre.
Todas ocupadas, devuelve 0}
var
  J: integer;
  Sw: boolean;
begin
  Sw:= false;
  J:= 0;
  while (J < Max_prior) and not Sw do
  begin
    J:= J+1;
    Sw:= (Tcp[J].Prd= P) or (Tcp[J].Prd= 0)
  end;
  if Sw then

```

```

Numcola:= J
else
  Numcola:= 0
end;

function Qmaxpri(var Tcp:tablacolas): integer;
{Busca cola de máxima prioridad. Si todas las tareas están procesadas
devuelve 0}
var
  J,K: integer;
  M: integer;
begin
  J:= 0;
  K:= 0;
  M:= Max_prior + 1; { valor de arranque }
  while J < Max_prior do
    if not Qvacia(Tcp[J]) and (Tcp[J].Prd < M) then
      begin
        K:= J;
        M:= Tcp[J].Prd
      end;
  Qmaxpri:= K
end;

procedure presentar_proc (T: ColaQp);
{Este procedimiento muestra en pantalla la tarea que va a ser procesada}
begin
  if LL = 0 then
    clrscr;
  LL:= LL+1
  write(T.Frente^.Info.Tarea);
  writeln('Prioridad:', T.Frente^.Info.Prior);
  if (LL mod 24)= 0 then
    begin
      delay(1000);
      clrscr
    end
end;

procedure generar_proc(var Tcp:Tablacolas; j:integer);
var
  N: integer;
  X: Tipo_proc;
  function Tarea(K: integer): String[20];
  const
    Nt= 9;
    Tareas = array[1.. Max_prior] of string[20] =
    ('RSET', 'COPIAR', 'ENLAZAR',
     'LEERDISCO', 'COMPILAR', 'HWAX',
     'EJECUTAT', 'ESTAT', 'TMTAX');
begin
  if K in [1.. Max_prior] then
    Tarea:= Tareas[K]
end;

```

```

begin
  N:= random(Nt) + 1; {Tarea al azar}
  X.Tarea:= Tarea(N);
  N:= random(Max_prio) + 1; {Prioridad al azar}
  X.Prior:= N;
  {Busca de Número de cola de prioridad}
  N:= Numcola(X.Prior);
  if N= 0 then
    begin
      writeln(' Overflow en tabla de colas ');
      halt(1)
    end;
  Tcp[N].Prd:= X.Prior;
  Qponer(X, Tcp[N])
  writeln('SE GENERA TAREA N°: ',J)
end;

procedure procesar_proc (var Tcp:Tablacolas; var Enco:boolean);
var
  I: integer;
begin
  I:= Qmaxpri(Tcp); {Cola de mayor prioridad}
  Enco:= I<>0;
  if I<> 0 then
    begin
      presentar_proc(Tcp[I]);
      borrar (Tcp[I])
    end
  end;
begin
  LL:= 0; {Para presentar tareas procesadas}
end.

```

Codificación del programa de gestión de colas de prioridades

```

program Colasdeprioridades;
uses
  unitcola,crt;
var
  Tablaq: Tablacolas;
  I, J: integer;
  Hm: boolean;
  C: char;
begin
  randomize;
  CrearQp (Tablaq, max_prior);
  {El bucle se realiza max_proc * 3 div 2 iteraciones para que
  se generen Max_proc tareas. Por cada tarea procesada, se
  generan dos nuevas}
  J:=0;
  for i:= 1 to (Max_proc*3 div 2) do
    if i mod 3 = 0 then
      procesar_proc (Tablaq, Hm)

```

```

else begin
    J := J + 1;
    generar_proc (Tablaq, j);
end;
repeat
    procesar_proc (Tablaq, Hm);
until not Hm
end.

```

PROBLEMA 8.2

En un archivo de texto F están almacenados números enteros arbitrariamente grandes. La disposición es tal, que hay un número entero por cada linea de F. Escribir un programa que muestre por pantalla la suma de todos los números enteros. Nota: Al resolver el problema habrá que tener en cuenta que al ser enteros grandes no pueden almacenarse en variables numéricas.

Análisis

Partimos de que los números están dispuestos de tal forma que hay uno por línea. El objetivo es obtener la suma de todos estos números y mostrarlo por pantalla.

Para leer los números, los hacemos dígito a dígito (como caracteres).

Esta lectura capta los dígitos de mayor peso a menor peso; como la suma se realiza en orden inverso, guardamos los dígitos en una pila de caracteres. Así, el último dígito que entra (la unidad) es el primero en salir.

A partir de este hecho, podemos seguir dos estrategias:

- Tener un array de enteros, de un máximo número de elemento establecido ($n = 250$, por ejemplo). La suma se hace dígito a dígito, y guardando el resultado en la misma posición del array. Esto es, sacando de la pila y sumando el dígito con el entero almacenado en la posición n del array; en la siguiente iteración se saca de la pila y se suma con la posición $n - 1$ del array, y así sucesivamente hasta que la pila esté vacía. En cada iteración hay que guardar un posible acarreo.

La alternativa descrita tiene el inconveniente de que hay que establecer un máximo de elementos del array, pudiendo ocurrir que nos quedemos «cortos».

- Otra estrategia consiste en sustituir el array de enteros por una lista enlazada. Ahora se saca un dígito de la pila y se suma con el primer elemento metido en la lista y con el acarreo. El dígito resultante se almacena en una nueva lista, a partir del último dígito insertado. En definitiva, es el típico tratamiento de una lista fifo. Con esta forma de proceder, cada vez que se accede a la lista para sumar dígito hay que liberar el correspondiente nodo.

Esta alternativa nos permite que no haya ningún tipo de restricción en cuanto al número de dígitos. Como contrapartida, el proceso es más lento, hay que realizar muchas operaciones de creación y liberación de nodos.

El programa que presentamos sigue la segunda alternativa. Para ello utilizamos dos unidades, la unidad de manejo de pilas y la unidad de manejo de listas fifo. Presentamos únicamente la sección de interface de pila, la codificación puede verse en las diversas realizaciones de pilas escritas en el capítulo anterior.

Codificación de la unidad Pila

```
unit Pila;
interface
type
  Tipoelem = char;
  PtrPila = ^Itemp;
  Itemp = record
    Info: Tipoelem;
    Sgte: PtrPila
  end;

procedure Pilavacia(var Pila: PtrPila);
function Pvacia(Pila: PtrPila): boolean;
procedure Pmeter(X: Tipoelem; var Pila: PtrPila);
procedure Pcima(var X: Tipoelem; Pila: PtrPila);
procedure Pborrar(var Pila: PtrPila);
procedure Psacar(var X: Tipoelem; var Pila: PtrPila);
{Fin de la sección de interfaz}
```

Codificación de la unidad Colas

```
unit Lfifo ;
interface
type
  Tipoinfo = integer;
  Ptrnodoq = ^Nodoc;
  Nodoc = record
    Info: Tipoinfo;
    Sgte: Ptrnodoq
  end;
  Cola= record
    Frente,
    Final: Ptrnodoq
  end;

procedure Qcrear(var C: Cola);
function Qvacia(C: Cola): boolean;
procedure Qponer(X: Tipoinfo; var C:Cola);
procedure Qborrar(var C: Cola);
function Frente(C:Cola): Tipoinfo;
procedure Quitar(var X: Tipoinfo; var C:Cola);
implementation

procedure Qcrear;
begin
  C.Frente := nil;
  C.Final := nil
end;
```

```

function Qvacia;
begin
  Qvacia:= (C.Frente = nil)
end;

procedure Qponer;
var
  A: Ptrnodoq;
begin
  new(A);
  A^.Info := X;
  A^.Sgte := nil;
  if Qvacia(C) then
    C.Frente := A
  else
    C.Final^.Sgte := A;
  C.Final:= A
end;

procedure Qborrar;
var
  A:Ptrnodoq;
begin
  if (not Qvacia(C))then
  begin
    A := C.Frente;
    C.Frente := C.Frente^.Sgte;
    dispose(A)
  end
end;

function Frente;
begin
  if (not Qvacia(C)) then
    Frente := C.Frente^.Info
end;

procedure Quitar;
begin
  X := Frente(C);
  Qborrar(C)
end;
begin
end.

```

Codificación completa del programa de gestión

```

program NumerosGrandes(output, Datos);
uses
  Crt, Pila, Lfifo;
var
  Datos: text;
  Pla : Ptppila;
  Lfo : Cola;

```

```

procedure Abrir(var F: text);
begin
  assign(Datos, 'C:Numeros.Dat');
  {I-}
  reset(F);
  {I+}
  if IOresult <> 0 then
  begin
    writeln ('Error al abrir el archivo., Salimos de ejecución ');
    halt(1)
  end
  else
    writeln('Archivo de números grandes abierto')
end;

procedure Leernumero(var Fn: text; var P: Ptrpila);
var
  C: char;
  function Esdígito(C: char): boolean;
begin
  Esdígito:= C in ['0'..'9']
end;
begin
  Pilavacia(P);
  while not eoln(Fn) do
  begin
    read(Fn, C);
    if Esdígito(C) then
      Pmeter(P, C)
    else begin
      writeln ('Error en el archivo de Números grandes'); halt(1)
    end
  end;
  readln(Fn)
end;

procedure Sumar(var P: Ptrpila; var Lfo: Cola);
var
  C: char;
  S1, S2, R: integer;
  Nq: Cola;
  function Número(C: char): integer;
begin
  Número:= ord(C) - ord('0')
end;

begin
  {R es el acarreo}
  R:= 0;
  Qrear(Nq);
  while not Pvacia(P) do
  begin
    Psacar(P, C);
    S1 := Número(C);
    S2 := 0;

```

```

if not Qvacia(Lfo) then
  Quitar(S2, Lfo);
  S2 := S1+S2+R;
  R := S2 div 10;
  S2 := S2 mod 10;
  Qponer(S2, Nq)
end;
while not Qvacia(Lfo) do
begin
  Quitar(S2, Lfo);
  S2 := S2+R;
  R := S2 div 10;
  S2 := S2 mod 10;
  Qponer(S2, Nq)
end;
if R <> 0 then Qponer(R, Nq);
Lfo:= Nq
end;

procedure Escribir(Q: Cola);
var
  N: integer;
begin
  if not Qvacia(Q) then
    begin
      Quitar(N, Q);
      Escribir(Q);
      write(N)
    end
  end;
begin
  Abrir(Datos);
  Qcrear(Lfo);
  while not eof(Datos) do
    begin
      Leernumero(Datos, Pla);
      Sumar(Pla, Lfo)
    end;
  if not Qvacia(Lfo) then
    begin
      writeln('Resultado de la suma de números grandes ');
      Escribir(Lfo)
    end
  end.
end.

```

RESUMEN

El proceso de inserción de datos se denomina *acolar* (*enquering*) y el proceso de eliminación se llama *desacolar* (*dequering*).

Una *cola* es una estructura de datos del tipo FIFO (*first-in, first-out*, primero en entrar, primero en salir); tiene dos extremos: una *cola*, donde se insertan los nuevos elementos, y una *cabecera o frontal*, de la que se borran o eliminan los elementos.

El tipo abstracto de datos cola se suele implementar mediante listas enlazadas por las mismas razones que en el caso de las pilas: facilidad para gestionar el tamaño variable o dinámico frente al caso del tamaño fijo.

Numerosos modelos de sistemas del mundo real, tanto físicos como lógicos, son del tipo cola; tal es el caso de la cola de trabajos de impresión en un servidor de impresoras, la cola de prioridades en viajes, programas de simulación o sistemas operativos. Una cola es el sistema típico que se suele utilizar como *buffer* de datos, cuando se envían datos desde un componente rápido de una computadora (memoria) a un componente lento (por ejemplo, una impresora).

Cuando se implementa una pila, sólo se necesita mantener el control de una entrada de la lista, mientras que en el caso de una cola, se requiere el control de los dos extremos de la lista.

Una cola de prioridad se puede implementar como un *array* de colas ordinarias o una lista enlazada de colas ordinarias.

EJERCICIOS

- 8.1. Se ha estudiado en el capítulo la realización de una cola mediante un array circular; una variante a esta representación es aquella en la que se tiene una variable que tiene la posición del elemento *Frente* y 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 elemento; extraer de la cola tres elementos; añadir elementos hasta que haya overflow. En todas las representaciones escribir los valores de *Frente* y *LongCola*.
- 8.2. Con la realización de 8.1 escribir las operaciones de Cola: *Qcrear*, *Qvacia*, *Qponer*, *Frente* y *Qborrar*.
- 8.3. Con la realización de una cola descrita en 8.1,2 consideremos una cola de caracteres. El array consta de 7 posiciones. Los campos *Frente*, *LongCola* actuales y el contenido de la cola:

$$\text{Frente} = 2 \quad \text{LongCola} = 3 \quad \text{Cola: } I, K, M$$

Escribir los campos *Frente*, *LongCola* y la Cola según se vayan realizando las siguientes operaciones:

- a) Se añaden los elementos F, J, G a la cola.
 - b) Se eliminan dos elementos de la cola.
 - c) Se añade el elemento A a la cola.
 - d) Se eliminan dos elementos de la cola.
 - e) Se añaden los elementos B, C, W a la cola.
 - f) Se añade el elemento R a la cola.
- 8.4. La realización de una cola mediante un array circular sacrifica un elemento del array; esto se puede evitar añadiendo un nuevo campo a la representación: *Vacio* de tipo lógico. Escribir una unidad en la que se defina el tipo de datos y se implementen las operaciones de manejo de colas.
 - 8.5. Considere la siguiente cola de nombres, representada por un array circular con 6 posiciones, los campos *Frente*, *Final* y *Vacio*:

$$\text{Frente} = 2, \text{ Final} = 4, \text{ Vacio} = \text{Falso} \quad \text{Cola: ,Mar,Sella,Licor,}$$

Escribir el contenido de la cola y sus campos según se realizan estas operaciones:

- a) Añadir Folk y Reus a la cola.
 b) Extraer de la cola.
 c) Añadir Kilo.
 d) Añadir Horche a la cola.
 e) Extraer todos los elementos de la cola.
- 8.6.** Una bicola con restricción de entrada sólo permite inserciones por uno de sus extremos, permitiendo extraer elementos por cualquier extremo. Definir e implementar las operaciones para este tipo de datos.
- 8.7.** Una bicola con restricción de salida sólo permite extraer elementos por uno de sus extremos, permitiendo insertar elementos por cualquier extremo. Definir e implementar las operaciones para este tipo de datos.
- 8.8.** Consideremos una bicola de caracteres representada en un array circular. El array consta de 9 posiciones. Los extremos actuales y el contenido de la bicola:

$$\text{Izquierdo} = 5 \quad \text{Derecho} = 7 \quad \text{Bicola: } A, C, E$$

Escribir los extremos y la bicola según se vayan realizando las siguientes operaciones:

- a) Se añaden los elementos F, J a la derecha de la bicola.
 b) Se añaden los elementos R, W, V a la izquierda de la bicola.
 c) Se añade el elemento M a la derecha de la bicola.
 d) Se eliminan dos letras a la izquierda.
 e) Se añaden los elementos K, L a la derecha de la bicola.
 f) Se añade el elemento S a la izquierda de la bicola.

PROBLEMAS

- 8.1.** Con un archivo de texto se quieren realizar estas 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 desea visualizar las palabras del archivo, empezando por la cola que contiene aquellas palabras que empiezan por la letra *a*, luego las de la letra *b*, y así sucesivamente.
- 8.2.** Una empresa de reparto de propaganda contrata a sus trabajadores por días. Cada repartidor puede trabajar varios días continuados o alternos. Los datos de los repartidores se almacenan en una lista simplemente enlazada. El programa a desarrollar contempla los siguientes puntos:
- 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.

- 8.3.** En un archivo de texto se quiere determinar todas las frases que son palíndromo. Para ello se sigue la siguiente estrategia: añadir cada carácter de la frase a una pila y a la vez a una cola. La extracción de caracteres simultánea de ambas y su comparación determina si la

frase es o no palíndromo. Escribir un programa para determinar todas las frases palíndromo del archivo de texto. Considerar que cada línea de texto es una frase.

- 8.4. En un archivo de texto se encuentran los resultados de una competición de tiro al plato, de tal forma que en cada línea se encuentra Apellido, Nombre, número de dorsal y número de platos rotos. Se desea escribir un programa que lea el archivo de la competición y determine los tres primeros. La salida ha de ser los tres ganadores y a continuación los concursantes en el orden en que aparecen en el archivo (utilizar la estructura cola).
- 8.5. El despegue de aeronaves en un aeropuerto se realiza siguiendo el orden establecido por una cola de prioridades. Hay 5 prioridades establecidas según el destino de la aeronave. Destinos de menos de 500 km tienen la máxima prioridad, prioridad 1, entre 500 y 800 km prioridad 2, entre 800 y 1.000 km prioridad 3, entre 1.000 y 1.350 km prioridad 4 y para mayores distancias prioridad 5. Cuando una aeronave recibe cierta señal se coloca en la cola que le corresponde y empieza a contar el tiempo de espera. Los despegues se realizan cada 6 minutos según el orden establecido en las distintas colas de prioridad. El piloto de una aeronave puede pasar el aviso a control de que tiene un problema, y no puede despegar por lo que pasa al final de la cola y se da la orden de despegue a la siguiente aeronave. Puede darse la circunstancia de que una aeronave lleve más de 20 minutos esperando, en ese caso pasará a formar parte de la siguiente cola de prioridad y su tiempo de espera se inicializa a cero.
Escribir un programa que simule este sistema de colas mediante una lista única, cada vez que despegue un avión saldrá un mensaje con las características del vuelo y el tiempo total de espera.
- 8.6. Resolver el problema 8.4 realizando el sistema de colas mediante un array de 5 colas.

PARTE



Estructuras de datos avanzadas

Recursividad: algoritmos recursivos

CONTENIDO

- 9.1. Recursividad.
- 9.2. Cuándo no utilizar recursividad.
- 9.3. Algoritmos divide y vence.
- 9.4. Implementación de procedimientos recursivos mediante pilas.
- 9.5. Algoritmos de vuelta atrás.
- 9.6. Problema de la selección óptima.
- 9.7. Problema de los matrimonios estables.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Un *procedimiento o función recursiva* es aquella que se llama a sí misma. Esta característica permite a un procedimiento recursivo repetirse con valores diferentes de parámetros. La recursión es una alternativa a la iteración muy elegante en la resolución de problemas, especialmente si éstos tienen naturaleza recursiva.

Normalmente, una solución recursiva es menos eficiente en términos de tiempo de computadora que una solución iterativa debido al tiempo adicional de llamada a procedimientos.

En muchos casos, la recursión permite especificar una solución más simple y natural para resolver un problema que en otro caso sería difícil. Por esta razón la *recursión* (recursividad) es una herramienta muy potente para la resolución de problemas y la programación.

9.1. RECURSIVIDAD

Un objeto recursivo es aquel que forma parte de sí mismo. Esta idea puede servir de ayuda para la definición de conceptos matemáticos. Así, la definición del conjunto

de los números naturales es aquel conjunto en el que se cumplen las siguientes características:

- *0 es un número natural.*
- *El siguiente número de un número natural es otro número natural.*

Mediante una definición finita hemos representado un conjunto infinito.

El concepto de la recursividad es muy importante en programación. La recursividad es una herramienta muy eficaz para resolver diversos tipos de problemas; existen muchos algoritmos que se describirán mejor en términos recursivos.

Suponga que dispone de una rutina *Q* que contiene una sentencia de llamada a sí misma, o bien una sentencia de llamada a una segunda rutina que a su vez tiene una sentencia de llamada a la rutina original *Q*. Entonces se dice que *Q* es una *rutina recursiva*.

Un procedimiento o función recursivos han de cumplir dos propiedades generales para no dar lugar a un bucle infinito con las sucesivas llamadas:

- *Cumplir una cierta condición o criterio base del que dependa la llamada recursiva.*
- *Cada vez que el procedimiento o función se llamen a sí mismos, directa o indirectamente, debe estar más cerca del incumplimiento de la condición de que depende la llamada.*

EJEMPLO 9.1. Secuencia de números de Fibonacci

Esta serie numérica es un ejemplo típico de cumplimiento de las dos propiedades generales de todo procedimiento recursivo. La serie de Fibonacci es:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34...

El término inicial es $a_0 = 0$, y los siguientes son: $a_1 = 1$, $a_2 = 2$, $a_3 = 3$, $a_4 = 5$, $a_5 = 8...$ observándose que cada término es la suma de los dos términos precedentes excepto a_0 y a_1 .

La serie puede ser definida recursivamente del modo siguiente:

$$\begin{aligned} \text{Fibonacci}(n) &= n \text{ si } n = 0 \text{ o } n = 1 \\ \text{Fibonacci}(n) &= \text{Fibonacci}(n-2) + \text{Fibonacci}(n-1) \text{ para } n > 1 \end{aligned}$$

Esta definición recursiva hace referencia a ella misma dos veces. Y la condición para que dejen de hacerse llamadas recursivas es que *n* sea 0 o 1. La llamada recursiva se hace en términos menores de *n*, *n-2*, *n-1*, por lo que se va acercando a los valores de los que depende la condición de terminación.

9.2. CUÁNDO NO UTILIZAR RECURSIVIDAD

La solución recursiva de ciertos problemas simplifica mucho la estructura de los programas. Como contrapartida, en la mayoría de los lenguajes de programación las llamadas recursivas a procedimientos o funciones tienen un coste de tiempo mucho mayor que sus

homólogos iterativos. Se puede, por tanto, afirmar que *la ejecución de un programa recursivo va a ser más lenta y menos eficiente que el programa iterativo que soluciona el mismo problema*, aunque, a veces, la sencillez de la estructura recursiva justifica el mayor tiempo de ejecución.

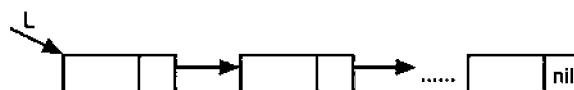
Los procedimientos recursivos se pueden convertir en no recursivos mediante la introducción de una pila y así emular las llamadas recursivas. De esta forma se puede eliminar la recursión de aquellas partes de los programas que se ejecutan más frecuentemente.

PROBLEMA 9.1

Dada una lista enlazada imprimirla en orden inverso.

Se tiene una lista enlazada que se referencia por un puntero externo al primer nodo de la lista. Se desea imprimir los nodos del último al primero. Los tipos de datos que se utilizan en la lista son:

```
type
  Tipointfo = ...;
  Puntero = ^Nodo;
  Nodo = record
    Info: Tipointfo;
    Sgte: Puntero
  end;
```



La variable L referencia al primer nodo de la lista. El problema va a ser descompuesto de forma recursiva, de tal forma que se vaya reduciendo la lista hasta llegar al último nodo. Es posible considerar la tarea a realizar dividida en los siguientes pasos:

- *Imprimir desde el segundo nodo hasta el último en orden inverso.*
- *Imprimir el primer nodo.*

Ya se ha reducido la lista en un nodo. A su vez, la primera parte se puede efectuar en dos pasos:

- *Imprimir desde el tercer nodo hasta el último en orden inverso.*
- *Imprimir el segundo nodo.*

El primer paso está formulado recursivamente. El proceso llega a su fin cuando no quedan elementos en la lista.

Codificación

```

procedure ImprimeInverso(L: Puntero);
begin
  if L <> nil then
    begin
      ImprimeInverso(L^.Sgte);
      write(L^.Info)
    end
  end;

```

9.2.1. Eliminación de la recursividad

Para eliminar la recursividad se utiliza una pila. En la pila el primer elemento que entra es el último en salir. Por consiguiente habrá que recorrer la lista, nodo a nodo. Las direcciones de los nodos (punteros) son almacenados en una pila hasta alcanzar el último nodo. Una vez alcanzado el último nodo se escribe el campo de información y aquí es donde se simula la recursividad. La vuelta atrás conseguida con la recursividad se consigue sacando de la pila las direcciones de los nodos, escribiendo la información, así hasta que quede vacía la pila.

Codificación

```

uses Pilas;
procedure ImprimeInverso (L: Puntero);
var
  Pila: Ptrpila;
  P: Puntero;
begin
  Pcrear(Pila);
  P := L;
  while P <> nil do
    begin
      Pmeter(P, Pila);
      P := P^.Sgte
    end;
  {En la pila están todas las direcciones de los nodos}
  while not Pvacia(Pila) do
    begin
      Psacar(P, Pila);
      Escribir(P^.Info)
    end
  end;

```

PROBLEMA 9.2

Multiplicación de dos números enteros por el método de la montaña rusa.

Descripción: *El método consiste en formar dos columnas, una por cada operando. Las columnas se forman aplicando repetidamente los pasos siguientes:*

- *Dividir por 2 el multiplicando. Anotar el cociente en la columna del multiplicando como nuevo multiplicando.*
- *Duplicar el multiplicador y anotarlo en la columna del multiplicador.*

Una vez hecho esto, se suman los valores de la columna del multiplicador que se correspondan con valores impares de la columna de multiplicandos. La suma es el producto.

Codificación

El problema se resuelve con una función que tiene como entrada dos enteros X, Y, y devuelve el producto aplicando el método de la montaña rusa.

```
function Producto(X, Y: longint): longint;
begin
  if X >= 1 then
    if (X mod 2) <> 0 then {Es impar}
      Producto := Y + Producto(X div 2, Y * 2)
    else
      Producto := Producto(X div 2, Y * 2)
    else
      Producto := 0
  end;
```

9.3. ALGORITMOS «DIVIDE Y VENCERÁS»

Una de las técnicas más importantes para el diseño de algoritmos recursivos es la técnica llamada «divide y vencerás». Consiste esta técnica en transformar un problema de tamaño n en problemas más pequeños, de tamaño menor que n . De modo que dando solución a los problemas unitarios se pueda construir fácilmente una solución del problema completo.

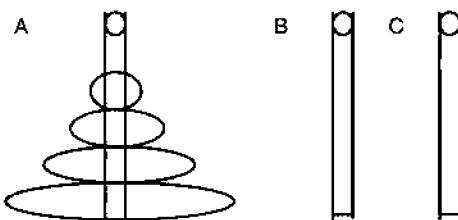
El algoritmo de búsqueda binaria es un ejemplo típico de esta técnica algorítmica. La lista ordenada de elementos se divide en dos mitades de forma que el problema de búsqueda de un elemento se reduce al problema de búsqueda en una mitad; así se prosigue dividiendo el problema hasta encontrar el elemento, o bien decidir que no se encuentra. Otro ejemplo claro de esta técnica es el método de ordenación rápido (*quick sort*).

Un algoritmo «divide y vencerás» puede ser definido de manera recursiva, de tal modo que se llama a sí mismo aplicándose cada vez a un conjunto menor de elementos. La condición para dejar de hacer llamadas es, normalmente, la obtención de un solo elemento.

9.3.1. Torres de Hanoi

Este famoso juego/acertijo que a continuación se describe, va a permitir aplicar la técnica de «divide y vencerás» en la resolución de un problema. El juego dispone de tres

postes, A, B, C; en el poste A se encuentran n discos de tamaño decreciente. El objetivo es mover uno a uno los discos desde el poste A al poste C utilizando el poste B como auxiliar. Además, nunca podrá haber un disco de mayor radio encima de otro de menor radio.



Vamos a plantear la solución de tal forma que el problema se vaya dividiendo en problemas más pequeños, y a cada uno de ellos aplicarles la misma solución. En lenguaje natural lo podemos expresar así:

1. El problema de mover n discos de A a C consiste en:

*mover los $n-1$ discos superiores de A a B
mover el disco n de A a C
mover los $n-1$ discos de B a C*

Un problema de tamaño n ha sido transformado en un problema de tamaño $n-1$. A su vez cada problema de tamaño $n-1$ se transforma en otro de tamaño $n-2$ (empleando el poste libre como auxiliar).

2. El problema de mover los $n-1$ discos de A a B consiste en:

*mover los $n-2$ discos superiores de A a C
mover el disco $n-1$ de A a B
mover los $n-2$ discos de C a B*

De este modo se va progresando, reduciendo cada vez un nivel la dificultad del problema hasta que el mismo sólo consista en mover un solo disco. La técnica consiste en ir intercambiando la finalidad de los postes, origen destino y auxiliar. La condición de terminación es que el número de discos sea 1. Cada acción de mover un disco realiza los mismos pasos, por lo que puede ser expresada de manera recursiva.

El procedimiento recursivo `TorresH` resuelve el acertijo o juego.

```
procedure TorresH (N: integer; A, B, C: char);
procedure Movimiento(N: integer; A, C: char);
begin
  writeln('Mover disco ',N,' de ',A,' a ',C)
end;
```

```

begin
  if N=1 then
    Movimiento (N, A, C)
  else begin
    TorresH (N - 1, A, C, B);
    Movimiento(N, A, C);
    TorresH(N - 1, B, A, C)
  end
end;

```

9.3.2. Trazo de un segmento

Se desea dibujar un segmento que conecta dos puntos (x_1, y_1) y (x_2, y_2) . Suponemos que siempre x_1, y_1, x_2, y_2 son ≥ 0 . Planteamos la solución de manera recursiva, utilizando la estrategia «divide y vencerás». El procedimiento consiste en dibujar el punto medio del segmento; sobre las dos mitades que determina ese punto medio volvemos a aplicar el algoritmo de dibujar su punto medio. Así hasta que, a base de dividir el segmento, se proporcione un segmento de longitud próxima a cero.

El programa Dibujar en el que se define el tipo de dato coordenada (Coord), se establecen los extremos del segmento y se dibuja.

```

program Dibujar;
uses crt;
type
  Coord = record
    X : real;
    Y : real
  end;
var
  O, F : Coord;
procedure Dib_sgmt(O, F: Coord);
var
  M: Coord;
begin
  with M do
    if (O.X+0.5) < F.X then
      begin
        X := (O.X+F.X)/2;
        Y := (O.Y+F.Y)/2;
        Dib_sgmt(O, M);
        if WhereX > 70 then
          GotoXY(5, WhereY + 1);
        write('(',X:3:1,',',Y:3:1,')');
        Dib_sgmt(M, F);
      end
    end;
begin
  Clrscr; GotoXY(5,1);
  O.X := 1; O.Y := 2;
  F.X := 12; F.Y := 7;
  Dib_sgmt(O, F)
end.

```

9.4. IMPLEMENTACIÓN DE PROCEDIMIENTOS RECURSIVOS MEDIANTE PILAS

Un procedimiento, una función contiene tanto variables locales como argumentos ficticios (*parámetros*). A través de los argumentos se transmiten datos en las llamadas a los subprogramas, o bien se devuelven valores al programa o subprograma invocante.

Además, el subprograma debe guardar la dirección de retorno al programa que realiza la llamada. En el momento en que termina la ejecución de un subprograma el control pasa a la dirección guardada.

Ahora el subprograma es recursivo, entonces además de la dirección de retorno, los valores actuales de las variables locales y argumentos deben de guardarse ya que se usarán de nuevo cuando el subprograma se reactive. Supongamos que se ha llamado al subprograma P, que tiene llamadas a si mismo, es decir, es recursivo. El funcionamiento del programa recursivo P:

- Se crea una pila para cada argumento.
- Se crea una pila para cada variable local.
- Se crea una pila para almacenar la dirección de retorno.

Cada vez que se hace una llamada recursiva a P, los valores actuales de los argumentos y de las variables locales se meten en sus pilas para ser procesadas posteriormente. Asimismo, cada vez que hay un retorno a P procedente de una llamada recursiva anterior, se restauran los valores de variables locales y argumentos de las cimas de las pilas.

Para la obtención de la dirección de retorno vamos a suponer que el procedimiento P contiene una llamada recursiva en la sentencia N. Entonces guarda en otra pila la dirección de retorno, que será la sentencia siguiente, la N + 1. De tal forma que cuando el nivel de ejecución del procedimiento P actual termine, alcance la sentencia end final, usará dicha pila de direcciones para volver al nuevo nivel de ejecución. De esta forma cuando la pila de direcciones se quede vacía volverá al programa que llamó al subprograma recursivo.

PROBLEMA 9.3

Hacemos una traza del estado de las pilas en la ejecución de función producto, por el método de la montaña rusa, para los valores de 19 y 45.

1	720	720+Prod...
2	360	Producto(..)
4	180	Producto(..)
9	90	90+Prod..
19	45	45+Prod..
Pila de X	Pila de Y	Pila de direcc.

La estrategia a seguir para emular un procedimiento recursivo P es:

1. Definir una pila para cada variable local y cada argumento, y una pila para almacenar direcciones de retorno.
2. En la sentencia *n* donde se haga la llamada recursiva a P:
 - (1) Meter en las respectivas pilas los valores actuales de las variables locales y argumentos; (2) Meter en la pila de direcciones la dirección de la siguiente sentencia.
 - Inicializar los argumentos con el valor actual de ellos y empezar la ejecución desde el principio del procedimiento.
3. Vuelta de la ejecución después de la llamada recursiva:
 - Si la pila de direcciones está vacía, devolver control al programa invocador.
 - Sacar de las pilas los valores de la cima. Llevar la ejecución a la sentencia extraída de la pila de direcciones.

9.4.1. El problema de las Torres de Hanoi resuelto sin recursividad

El problema de las Torres de Hanoi, cuya solución recursiva se ha descrito, se puede resolver de forma no recursiva siguiendo esta estrategia.

Algoritmo no recursivo Torres de Hanoi

Los argumentos conocidos son:

```
N Número de discos.  
A Varilla A  
B Varilla B  
C Varilla C
```

Para cada uno de ellos se define una pila:

```
PilaN  
PilaA  
PilaB  
PilaC
```

Además la pila de direcciones: PilaDir

```
Torre_Hanoi(N, A, B, C)
```

Crear pilas: PilaN, PilaA, PilaB, PilaC, PilaDir

Las etapas a considerar en la construcción del algoritmo Torres de Hanoi son:

1. Condición para acabar.

```
si N = 1 entonces  
  Escribir Palo A -> Palo C  
  Ir a 5  
fin si
```

2. Llamada recursiva.

a) Guardar en pilas:

```
Meter(N, PilaN)
Meter(A, PilaA)
Meter(B, PilaB)
Meter(C, PilaC)
Meter(<Paso 3>, PilaDir)
```

b) Actualizar los argumentos:

```
N ← N - 1
A ← A
Intercambiar B, C:
t ← B
B ← C
C ← t
```

c) Volver a paso 1.

3. Escribir Varilla A → Varilla C.

4. Segunda llamada recursiva:

a) Guardar en pilas:

```
Meter(N, PilaN)
Meter(A, PilaA)
Meter(B, PilaB)
Meter(C, PilaC)
Meter(<Paso 5>, PilaDir)
```

b) Actualizar los argumentos:

```
N ← N-1
C ← C
Intercambiar A, B:
t ← B
B ← A
A ← t
```

c) Volver a paso 1.

5. Este paso se corresponde con el retorno de las llamadas recursivas.

a) Si Pila vacía entonces fin de algoritmo; volver a la rutina llamadora.

b) Sacar de la pila:

```
Sacar(N, PilaN)
Sacar(A, PilaA)
Sacar(B, PilaB)
Sacar(C, PilaC)
Sacar(Dir, Piladir)
```

c) Ir al paso que está guardado en Dir.

Codificación no recursiva del problema de las Torres de Hanoi

Para realizar la codificación del algoritmo no recursivo de las Torres de Hanoi es necesario una pila de enteros (PilaN) y tres pilas de elementos char (PilaA, PilaB, PilaC). Para simular direcciones de retorno se utiliza otra pila de enteros en la que se guarda 1 o 2, según se simule la primera llamada o la segunda llamada recursiva. Los tipos de datos y las operaciones están en las unidades de pilas.

```

uses Pilachar, Pilaint;

procedure Hanoiter(N: integer; A, B, C: char);
var
  PilaA, PilaB, PilaC: Pilachar.PtrPila;
  PilaN, PilaD: Pilaint.PtrPila;
  Paso: integer;
  T: char;
begin
  Pilachar.Pcrear(PilaA);
  Pilachar.Pcrear(PilaB);
  Pilachar.Pcrear(PilaC);
  Pilaint.Pcrear(PilaN);
  Pilaint.Pcrear(PilaD);
  Pilaint.Pmeter(1,PilaD);
  while not Pilaint.Pvacia(PilaD) do
    if (N = 1) then
      begin
        Mover_disco(N,A,C);
        Pilaint.Psacar(Paso,PilaD);
        if not Pilaint.Pvacia(PilaD) then
          begin
            Pilachar.Psacar(A,PilaA);
            Pilachar.Psacar(B,PilaB);
            Pilachar.Psacar(C,PilaC);
            Pilaint.Psacar(N,PilaN)
          end
      end
    end
  else begin
    Pilaint.Psacar(Paso,PilaD);
    Case Paso of
      1: begin
        Pilachar.Pmeter(A,PilaA);
        Pilachar.Pmeter(B,PilaB);
        Pilachar.Pmeter(C,PilaC);
        Pilaint.Pmeter(N,PilaN);
        Pilaint.Pmeter(2,PilaD); {Paso actual}
        Pilaint.Pmeter(1,PilaD);
        N := N-1;
        T := B; B := C; C := T
      end;
      2: begin
        Mover_disco(N,A,C);
        {No es necesario almacenar los argumentos A, B, C, N pues no
         se utilizan}
        Pilaint.Pmeter(1,PilaD);
      end;
    end;
  end;
end;

```

```

    N := N-1;
    T := B; B := A; A := T
  end
end
end;

```

Esta solución resulta un tanto farragosa debido a la utilización de 5 pilas. Una simplificación fácil que puede hacerse es utilizar una única pila, en ésta se guarda el estado completo de los argumentos y la dirección de retorno. En la unidad PilaStat está declarado el tipo Elemento, un registro con los campos necesarios para guardar A, B, C, N y Dir, y además las operaciones de manejo de pilas.

Los tipos de datos en la unidad PilaStat:

```

type
  Elemento = record
    Pa, Pb, Pc: char;
    Nd, Dir: integer
  end;
  PtrPila = ^Itemp;
  Itemp = record
    Info: Elemento;
    Sgte: PtrPila
  end;

```

El procedimiento Hanoiter2:

```

procedure Hanoiter2(N: integer; A, B, C: char);
var
  Pila: PtrPila;
  T: Elemento;
  procedure Estado(var St: Elemento; A,B,C: char; N,D: integer);
  begin
    with St do
    begin
      Pa := A;
      Pb := B;
      Pc := C;
      Nd := N;
      Dir := D
    end
  end;
begin
  Pcrear(Pila);
  Estado(T,A,B,C,N,1);
  Pmeter(T,Pila);
  while not Pvacia(Pila) do
    with T do
    begin
      Psacar(T,Pila);
      if (Nd = 1) then
        Mover_disco(Nd, Pa, Pc)
      else
    end;
  end;

```

```

    case Dir of
    1: begin
        Dir := 2;
        Pmeter(T,Pila); {Estado actual}
        Estado(T,Pa,Pc,Pb,Nd-1,1);
        Pmeter(T,Pila);
    end;
    2: begin
        Mover_disco(Nd, Pa, Pc);
        Estado(T,Pb,Pa,Pc,Nd-1,1);
        Pmeter(T,Pila);
    end
    end
end;

```

En el programa Torrehanoi se muestran diversas soluciones de las Torres de Hanoi según el número de discos. Se aplica tanto la solución recursiva como a las dos soluciones iterativas. La salida se dirige a la impresora.

```

program Torrehanoi;
uses Pilaint, Pilachar, Pilastat, crt, printer ;
const
  Maxnumdiscos = 15;
var
  N: integer;
  A,B,C: char;

procedure mover_disco (N: integer; A,C: Char);
begin
  writeln(Lst, 'Mover disco ',N,' de ',A,' a ',C)
end;
procedure mover_torre(N: integer ;A,B,C: Char);
begin
  if N = 1 then
    mover_disco(N,A,C)
  else
    begin
      mover_torre(N-1,A,C,B);
      mover_disco(N,A,C);
      mover_torre(N-1,B,A,C)
    end;
end;

procedure Hanoiter1(N: integer; A, B, C:char);
var
  PilaA, PilaB, PilaC: Pilachar.PtrPila;
  PilaN, PilaD: Pilaint.PtrPila;
  Paso: integer;
  T: char;
begin
  Pilachar.Pcrear(PilaA);
  Pilachar.Pcrear(PilaB);
  Pilachar.Pcrear(PilaC);

```

```

Pilaint.Pcrear(PilaN);
Pilaint.Pcrear(PilaD);
Pilaint.Pmeter{1,PilaD};
while not Pilaint.Pvacia(PilaD) do
  if (N = 1) then
    begin
      Mover_disco(N,A,C);
      Pilaint.Psacar(Paso,PilaD);
      if not Pilaint.Pvacia(PilaD) then
        begin
          Pilachar.Psacar(A,PilaA);
          Pilachar.Psacar(B,PilaB);
          Pilachar.Psacar(C,PilaC);
          Pilaint.Psacar(N,PilaN)
        end
      end
    else begin
      Pilaint.Psacar(Paso,PilaD);
      Case Paso of
        1: begin
          Pilachar.Pmeter(A,PilaA);
          Pilachar.Pmeter(B,PilaB);
          Pilachar.Pmeter(C,PilaC);
          Pilaint.Pmeter{N,PilaN};
          Pilaint.Pmeter{2,PilaD}; {Paso actual}
          Pilaint.Pmeter{1,PilaD};
          N := N-1;
          T := B; B := C; C := T
        end;
        2: begin
          Mover_disco(N,A,C);
          Pilaint.Pmeter{1,PilaD};
          N := N-1;
          T := B; B := A; A := T
        end
      end
    end
  end;
end;

procedure Hanoiter2(N: integer; A, B, C:char);
var
  Pila: PtrPila;
  T: Elemento;
procedure Estado(var St: Elemento; A,B,C: char; N,D: integer);
begin
  with St do
  begin
    Pa := A;
    Pb := B;
    Pc := C;
    Nd := N;
    Dir := D
  end
end;
begin
  Pcrear(Pila);

```

```

Estado(T,A,B,C,N,1);
Pmeter(T,Pila);
while not Pvacia(Pila) do
with T do
begin
  Psacar(T,Pila);
  if (Nd = 1) then
    Mover_disco(Nd, Pa, Pc)
  else
    Case Dir of
      1: begin
          Dir := 2;
          Pmeter(T,Pila); {Estado actual}
          Estado(T,Pa,Pc,Pb,Nd-1,1);
          Pmeter(T,Pila);
        end;
      2: begin
          Mover_disco(Nd, Pa, Pc);
          Estado(T,Pb,Pa,Pc,Nd-1,1);
          Pmeter(T,Pila);
        end
      end
    end
  end;
begin
  clrscr;
  A := 'A';
  B := 'B';
  C := 'C';
  write('Número de discos?:');
  repeat
    readln(N)
  until N in [1..Maxnumdiscos];
  clrscr;
  writeln(Lst, 'Solución recursiva con ',N, 'discos');
  mover_torre(N,A,B,C);
  writeln(Lst);
  writeln(Lst, 'Solución iterativa1 con ',N,' discos');
  Hanoiter1(N,A,B,C);
  writeln;
  writeln(Lst, 'Solución iterativa2 con ', N,' discos');
  Hanoiter2(N,A,B,C)
end.

```

9.5. ALGORITMOS DE VUELTA ATRÁS (BACKTRACKING)

Esta técnica algorítmica recurre a realizar una búsqueda exhaustiva sistemática de una posible solución al problema planteado. El procedimiento general es descomponer el proceso de tanteo de una solución en tareas parciales. Cada tarea parcial se expresa frecuentemente en forma recursiva.

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. Es una prueba sistemática hasta llegar a la solución, o bien determinar que no hay solución por haberse agotado todas las opciones que probar. Aplicamos esta técnica algorítmica al conocido problema de la vuelta del caballo que se describe a continuación.

En un tablero de ajedrez de $N \times N$ casillas. Un caballo sigue los movimientos de las reglas del ajedrez. El caballo se sitúa en la casilla de coordenadas (X_0, Y_0) . El problema consiste en encontrar, si existe, un circuito que permita al caballo pasar exactamente una vez por cada una de las casillas del tablero.

La tarea básica en que va a basarse el problema es la de que el caballo realice un nuevo movimiento, o bien decidir que ya no quedan movimientos posibles. El algoritmo que exponemos a continuación trata de llevar a efecto un nuevo movimiento del caballo con el objetivo de visitar una vez todas las casillas.

```

Algoritmo Caballo;
inicio
    Repetir
        Seleccionar nuevo movimiento del caballo
        si (Está en tablero) y (No pasó ya) entonces
            Anotar movimiento en el tablero
            si (No completado tablero) entonces
                Nuevo ensayo: Caballo
                (Vuelta de llamada recursiva)
                si (No se alcanzó solución) entonces
                    Borrar anotación anterior
                fin_si
                fin_si
            fin_si
        hasta (Completado tablero) o (No más posibles movimientos)
fin

```

En los algoritmos de vuelta atrás siempre hay nuevas tentativas en busca de solución, nuevos ensayos. En el caso de que un ensayo no conduzca a alcanzar la solución, se da marcha atrás. Esto es, se borra la anotación hecha al realizarse el ensayo y se vuelve a hacer otro, en el caso de que sea posible (en el caso de un caballo, éste puede realizar hasta ocho movimientos desde una posición dada).

Para describir con más precisión el algoritmo, definimos los tipos de datos para representar el tablero y los movimientos.

```

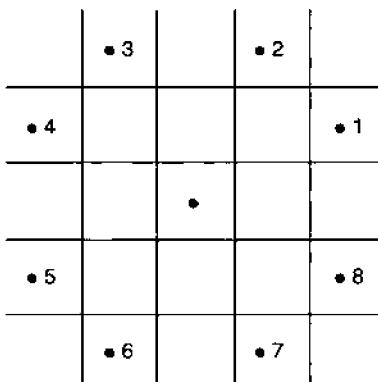
const
    N= 8;
type
    Tablero = array[1..N, 1..N] of integer;
var
    T: Tablero;

```

El tablero se representa mediante un array de enteros para guardar el número de movimiento en el que pasa el caballo. Una posición del tablero contendrá:

$$T[X, Y] \begin{cases} 0 & \text{Por la casilla } (X, Y) \text{ no pasó el caballo.} \\ i & \text{Por la casilla } (X, Y) \text{ pasó el caballo en el movimiento } i. \end{cases}$$

Para la acción de seleccionar un nuevo movimiento, hay que tener en cuenta que dada una posición el caballo puede realizar 8 posibles movimientos. En una matriz de dos dimensiones se guardan los 8 desplazamientos relativos para conseguir un siguiente salto. La figura nos muestra los desplazamientos:



La condición de que el nuevo movimiento esté en el tablero y de que no hubiera pasado anteriormente:

```
(X in [1..N]) and (Y in [1..N]) and (T[X,Y] = 0)
```

La condición «no completado tablero» se representa con el número de movimiento *i* y el total de casillas:

```
i < NxN
```

Para no penalizar en tiempo de ejecución, las variables array son definidas globales.

CODIFICACIÓN DEL ALGORITMO VUELTA DEL CABALLO CON UN MÉTODO RECURSIVO

```
const
  N = 8;
type
  Tablero = array[1..N, 1..N] of integer;
  Coord = array[1..2, 1..8] of integer;
var
  T: Tablero;
  H: Coord;
  I, J: integer;
  S: boolean;
```

```

procedure Caballo(I: integer; X, Y: integer; var S: boolean);
var
  Nx, Ny: integer;
  K: integer;
begin
  S := false;
  K := 0; {Inicializa el conjunto posible de movimientos}
repeat
  K := K+1;
  Nx := X+ H[1,K];
  Ny := Y+ H[2,K];
  if (Nx in [1..N]) and (Ny in [1..N]) then
  if T[Nx, Ny] = 0 then
    begin
      T[Nx, Ny] := I; {Anota movimiento}
      if I < N * N then
        begin {Se produce un nuevo ensayo}
          Caballo(I+1, Nx, Ny, S);
          if not S then {No se alcanza la solución}
            T[Nx, Ny] := 0 {Borrado de la anotación para probar con
                           otro movimiento}
        end
      else {Tablero completado}
        S := true
    end
  until S or (K= 8)
end;

```

En las sentencias del bloque principal se inicializa las posiciones del tablero y los desplazamientos relativos para obtener nuevas coordenadas.

```

begin
  H[1,1] := 2; H[2,1] := 1;
  H[1,2] := 1; H[2,2] := 2;
  H[1,3] := -1; H[2,3] := 2;
  H[1,4] := -2; H[2,4] := 1;
  H[1,5] := -2; H[2,5] := -1;
  H[1,6] := -1; H[2,6] := -2;
  H[1,7] := 1; H[2,7] := -2;
  H[1,8] := 2; H[2,8] := -1;
  for I := 1 to N do
  for J := 1 to N do
    T[I,J] := 0;
    {El caballo parte de la casilla (1,1)}
  T{1,1} := 1;
  Caballo(2,1,1,S);
  if S then
    for I := 1 to N do
    begin
      for J := 1 to N do
        write(T[I,J]:4);
        writeln
    end
  else
    writeln('NO SE ALCANZA SOLUCION')
end.

```

La característica principal de los algoritmos de vuelta atrás es intentar realizar pasos que se acercan cada vez más a la solución completa. Cada paso es anotado, borrándose tal anotación si se determina que no conduce a la solución, esta acción constituye una vuelta atrás. Cuando se produce una «vuelta atrás» se ensaya con otro paso (otro movimiento). En definitiva, se prueba sistemáticamente con todas las opciones posibles hasta encontrar una solución, o bien agotar todas las posibilidades sin llegar a la solución.

El esquema general de este método:

```

procedimiento EnsayarSolucion
  Inicio
    Inicializar cuenta de opciones de selección
  repetir
    Seleccionar nuevo paso hacia la solución
    si válido entonces
      Anotar el paso
      si no completada solución entonces
        EnsayarSolución a partir del nuevo paso
        si no alcanza solución completa entonces
          borrar anotación
        fin_si
      fin_si
    fin
  Hasta (Completada solución) o (No más opciones)
fin

```

Este esquema puede tener variaciones. En cualquier caso siempre habrá que adaptarlo a la casuística del problema a resolver.

9.5.1. Solución del problema «Salto del caballo» con esquema iterativo

Ampliamos el problema anterior para encontrar todas las rutas que debe seguir el caballo para completar el tablero pero utilizando pilas y así transformar el esquema recursivo en iterativo. En la unidad PilasC tenemos todas las operaciones de manejo de pilas. En la pila se almacena el número de movimiento, número de salto y posición. A continuación escribimos el código de la unidad PilasC.

```

unit PilasC;
interface
const
  N = 8;
type
  Indice = 1..N;
  Tipoelem = record
    Nummov: integer;
    Salto: integer;
    X: Indice;
    Y: Indice
  end;

```

```
PtrPila = ^Itemp;
Itemp = record
  Info: Tipoelem;
  Sgte: PtrPila
end;

procedure Pcrear(var Pila: PtrPila);
function Pvacia(Pila: PtrPila): boolean;
procedure Pmeter(var Pila: PtrPila; X: Tipoelem);
procedure Pcima(Pila: PtrPila; var X: Tipoelem);
procedure Pborrar(var Pila: PtrPila);
procedure Psacar(var Pila: PtrPila; var X: Tipoelem);
implementation

procedure Pcrear(var Pila: PtrPila);
begin
  Pila := nil
end;

function Pvacia(Pila: PtrPila): boolean;
begin
  Pvacia := (Pila = nil)
end;

procedure Pmeter(var Pila: PtrPila; X: Tipoelem);
var
  A: PtrPila;
begin
  new(A);
  A^.Info := X;
  A^.Sgte := Pila;
  Pila := A
end;

procedure Pcima(Pila: PtrPila; var X: Tipoelem);
begin
  if not Pvacia(Pila) then
    X := Pila^.Info
end;

procedure Pborrar(var Pila: PtrPila);
var
  A: PtrPila;
begin
  if not Pvacia(Pila) then
  begin
    A := Pila;
    Pila := Pila^.Sgte;
    dispose(A)
  end
end;

procedure Psacar(var Pila: PtrPila; var X: Tipoelem);
var
  A: PtrPila;
begin
  if not Pvacia(Pila) then
```

```

begin
  A := Pila;
  X := Pila^.Info;
  Pila := Pila^.Sgte;
  dispose(A)
end;
end;
begin
end.

```

Esta unidad la utilizamos para escribir el programa que iterativamente encuentra todas las rutas que puede seguir el caballo para recubrir el tablero.

```

program Cabaitef;
uses
  crt, PilaC;
type
  Incrementos = array [1..8] of integer;
  Tabla = array [indice, indice] of integer;
var
  Cp: Ptrpila;
  R: Tipoelem;
  Xn, Xp, Yn, Yp: integer;  {coordenadas casillas}
  Fila, Columna: Indice;
  Rl: set of Indice;
  IncX,
  IncY: Incrementos; {Desplazamientos relativos}
  Tablero : Tabla;
  Soluciones, Salto, NumMov: integer;
  Recorrido, Colocado : boolean;

procedure Anotar(var Nw: Tipoelem; Nm, S: integer;
                 U,V: Indice);
begin
  with Nw do
  begin
    Nummov := Nm;
    Salto := S;
    X := U;
    Y := V
  end
end;
end;

procedure EscribePila (Cp : PtrPila);
begin
  if Cp <> nil then
  begin
    EscribePila (Cp^.sgte);
    with Cp^ do
      write (info.nummov:4, info.salto:2,info.x:2,'-',info.y);
  end
end;

procedure Iniciartablero (var Tablero :Tabla);
var
  Fila, Columna : Indice;

```

```

begin
  for Fila := 1 to N do
    for Columna := 1 to N do
      Tablero[Fila, Columna] := 0
end;

procedure Incre(var Incx, Incy: Incrementos);
begin  {Desplazamientos relativos para desde una posición dada dar
        un salto de caballo}
  IncX[1] := 2; IncY[1] := 1;
  IncX[2] := 1; IncY[2] := 2;
  IncX[3] := -1; IncY[3] := 2;
  IncX[4] := -2; IncY[4] := 1;
  IncX[5] := -2; IncY[5] := -1;
  IncX[6] := -1; IncY[6] := -2;
  IncX[7] := 1; IncY[7] := -2;
  IncX[8] := 2; IncY[8] := -1;
end;

procedure Escribetablero(var Tablero :Tabla);
var
  Fila, Columna: Indice;
begin
  for Columna := N downto 1 do
    begin
      for Fila := 1 to N do
        write (Tablero[Fila, Columna]:5);
      writeln
    end;
  end;
begin
  clrscr;
  Rl := [1..N];
  Incre(IncX, Incy);
  Iniciartablero(Tablero);
  Write ('Introduzca casilla inicial: ');
  Nummov := 1;
  with R do
  begin
    readln (X,Y);
    Nummov := 1;
    Tablero[X,Y] := Nummov;
    Salto := 8
  end;
  Pcrear(Cp);
  Pmeter(Cp, R);
  Recorrido := false;
  Soluciones := 0;
  while not Pvacia(Cp)do
  begin
    if Recorrido then
    begin
      Pcima(Cp, R);
      with R do
      Tablero[X,Y] := 0;
      Salto := R.Salto;
      Pborrar(Cp);
    end;
  end;
end;

```

```

("Vuelta atrás" hasta llegar a una casilla desde la que no
se agotaron todos los saltos y así poder encontrar otra
solución)
while(Salto = 8) and not Pvacia(Cp) do
begin
  Pcpma(Cp, R);
  with R do
    Tablero[X,Y] := 0;
  Pborrar(Cp);
  Nummov := Nummov - 1;
  if not Pvacia(Cp) then
    begin
      Pcpma(Cp, R);
      Salto := R.Salto
    end
  end;
  if not Pvacia(Cp) then
  begin
    Tablero [R.X,R.Y] := 0; {Deshace anterior para búsqueda
                           de otra Solución}
    Nummov := R.Nummov - 1;
    Pborrar(Cp);
    Recorrido := false
  end
end
else Salto := 0;

if not Pvacia(Cp) then
begin
  Colocado := false;
  Pcpma(Cp, R); {Para dar un nuevo salto}
  repeat
    Salto := Salto+1;
    Xn := R.X + Incx[Salto];
    Yn := R.Y + Incy[Salto];
    if (Xn in Rl) and (Yn in Rl) then
      Colocado := Tablero [Xn, Yn] = 0;
  until Colocado or (Salto = 8);
  if Colocado then {Anota movimiento}
  begin
    Nummov := Nummov + 1;
    Tablero[Xn, Yn] := Nummov;
    Anotar(R, Nummov, Salto, Xn, Yn);
    Pmeter(Cp, R);
    if Nummov = N * N then {ruta completada}
    begin
      Recorrido := true;
      Soluciones := Soluciones + 1;
      writeln('SOLUCION ':44, Soluciones);
      Escribetablero (tablero);

    end
  end
else {No pudo colocarse Nummov+1 a partir de Nummov }
  while not Colocado and not Pvacia(Cp) do

```

```

begin {Marcha atrás: es sacado el movimiento anterior
       para nueva "prueba" con otro salto}
  Pcoma(Cp, R);
  with R do
    Tablero[X,Y] := 0;
    Salto := R.Salto;{Es el último Salto}
    Pborrar(Cp);
    if not Pvacia(Cp) then
    begin
      Pcoma(Cp, R);
      with R do
      begin
        Xp := X;
        Yp := Y
      end;
      while (Salto < 8) and not Colocado do
      begin
        Salto := Salto + 1;
        Xn := Xp + Incx[Salto];
        Yn := Yp + Incy[Salto];
        if (Xn in Rl) and (Yn in Rl) then
          if Tablero [Xn, Yn] = 0 then
          begin
            Colocado := true;
            Nummov := R.Nummov + 1;
            Tablero[Xn, Yn] :=Nummov;
            Anotar(R, Nummov, Salto, Xn, Yn);
            Pmeter(Cp, R);
          end
        end
      end { fin de if not Pvacia }
      end { fin de while not colocado }
    end {fin de if not Pvacia }
  end; {fin del mientras}

  Writeln ('Soluciones encontradas = ', Soluciones);
end.

```

A continuación son expuestos problemas típicos que son resueltos siguiendo la estrategia de los algoritmos de vuelta atrás.

9.5.2. 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 reina pueda atacar («comer») a cualquiera de las otras. En primer lugar recordamos la regla del ajedrez respecto de los movimientos de la reina. Ésta puede moverse a lo largo de la columna, fila y diagonales donde se encuentra. Una primera conclusión es que cada columna puede contener una y sólo una reina, por lo que la colocación de la reina i queda restringida a las casillas de la

columna i . Por esta coincidencia el parámetro i nos sirve de índice de columna dentro de la cual podremos colocarla en los ocho posibles valores de fila.

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 , definimos un vector entero. El contenido de una posición del vector será 0, o bien el índice de fila donde se sitúa la reina.

```
const
  N= 8; { Número de reinas }
type
  Fila = array[1..N] of integer;
```

En orden a buscar la solución completa, la tarea básica que exhaustivamente se prueba es colocar la reina i en las 8 posibles filas. La comprobación de que dicho ensayo es válido 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, hasta llegar a la solución completa, o bien determinar que en un paso no sea posible colocar la reina. Entonces, en el retroceso se coloca la reina anterior en otra fila válida para realizar un nuevo tanteo.

CODIFICACIÓN

```
const
  N= 8; { Número de reinas }
type
  Filas = array[1..N] of integer;
var
  Reinas: Filas;
  Solucion: boolean;
  I: integer;

procedure Colocar_Reinas (I: integer; var S: boolean);
var
  K: integer;
function Valido(J: integer): boolean;
{Inspecciona si la reina de la columna J es atacada por alguna reina
colocada anteriormente}
var
  R: integer;
  V: boolean;
begin
  V := true;
  for R := 1 to J-1 do
  begin
    V := V and (Reinas[R] <> Reinas[J]);
    {No esté en la misma fila}
    {No esté en alguna de las dos diagonales:}
    V := V and ((Reinas[J] + J) <> (Reinas[R] + R));
    V := V and ((Reinas[J] - J) <> (Reinas[R] - R));
  end;
  Valido := V
end;
```

```

end;
begin
  K := 0; {Inicializar posibles movimientos}
repeat
  K := K + 1;
  S := false;
  Reinas[I] := K; {Tentativa de colocar reina I en fila K, a la vez
                    queda anotado movimiento}
  if Valido(I) then
    if I < 8 then {8 puede sustituirse por N}
      begin {No completado el problema}
        Colocar_Reinas(I + 1, S);
        if not S then {Vuelta atrás}
          Reinas[I] := 0
      end
    else {Las 8 reinas colocadas}
      S := true
  until S or (K= 8)
end;

```

Observar que en la vuelta atrás se puede omitir borrar la anotación ya que en la siguiente iteración se prueba con otra fila, asignando un nuevo valor a la misma posición del vector Reinas. En el bloque principal es escrita la solución.

```

begin
  Colocar_Reinas(1, Solucion);
  if Solucion then
    for I := 1 to 8 do
      write(Reinas[I]:5)
end.

```

9.5.3. Solución no recursiva al problema de las ocho reinas

Al igual que la solución iterativa del salto del caballo, presentamos la solución iterativa de las 8 reinas con una pila para emular las llamadas recursivas. Ahora lo relevante en cada movimiento es el número de fila y de columna, ello es lo que guardamos en la pila.

A continuación escribimos el interface de la unidad PilaR.

```

unit PilaR;
interface
  const
    N = 8;
  type
    Rango = 1..N;
    PtrPila = ^Nodo;
    Nodo = record
      Nr, Col: word;
      sgte: PtrPila
    end;
  procedure Pcrear(var Pila: PtrPila);
  function Pvacia(Pila: PtrPila): boolean;

```

```

procedure Pmeter(var Pila: PtrPila; Nr, Col: word);
procedure Pcima(Pila: PtrPila; var Nr, Col: word);
procedure Pborrar(var Pila: PtrPila);
procedure Psacar(var Pila: PtrPila; var Nr, Col: word);
{Fin de la sección de interface}

```

La codificación del programa que encuentra una solución al problema.

```

program ReinaA_It;
{Síntesis N reinas sin atacarse en tablero N*N. Da una sola solución}
uses Crt, PilaR;
type
  Tipocol = array [Rango] of boolean;
  DiagonalDchaIzda = array [2..2 * N] of boolean;
  {Fila(número de reina) + Columna = cte.}
  DiagonalIzdaDcha = array [1..N..N - 1] of boolean;
  {Fila o número de reina - columna = cte.}
var
  Col, Nf, Nr, i: integer;
  Colibre: Tipocol; {No hay reina en fila j-ésima}
  Ddch: DiagonalDchaIzda; {no hay reina en Diagonal Dcha-Izda}
  Dizq: DiagonalIzdaDcha; {no hay reina en Diagonal Izda-Dcha}
  Cp: Ptrpila;
  Colocado: boolean;

procedure Listarpila(Cp: Ptrpila);
begin
  while Cp^.sgte <> nil do
    begin
      write(Cp^.Nr, '=', Cp^.Col, '');
      Cp := Cp^.Sgte
    end;
  writeln
end;

procedure Liberar(var Col: Tipocol; var Ddch: DiagonalDchaIzda;
                  var Dizq: DiagonalIzdaDcha; Nr, Co: word);
begin
  Col[Co] := true;
  Ddch[Nr + Co] := true;
  Dizq[Nr - Co] := true
end;

procedure Ocupar(var Col: Tipocol; var Ddch: DiagonalDchaIzda;
                  var Dizq: DiagonalIzdaDcha; Nr, Co: word);
begin
  Col[Co] := false;
  Ddch[Nr + Co] := false;
  Dizq[Nr - Co] := false
end;

begin
  ClrScr;
  write('Introduzca dimensión del tablero:');
  repeat

```

```

readln(Nf)
until Nf <= N;
for i := 1 to Nf do Colibre[i] := true;
for i := 2 to 2*Nf do Ddch[i] := true;
for i := 1 - Nf to Nf - 1 do Dizq[i] := true;
Pcrear(Cp); Nr := 0; Col := 0;
Pmeter(Cp, Nr, Col);
while(Nr < Nf) and not Pvacia(Cp) do
begin
  Nr := Nr + 1; Colocado := false;
  while not Colocado and (Col < Nf) do
  begin
    Col := Col + 1;
    if Colibre[Col] then
      if Ddch[Nr + Col] then
        if Dizq[Nr - Col] then
        begin
          Ocupar(Colibre, Ddch, Dizq, Nr, Col);
          Colocado := true;
          Pmeter(Cp, Nr, Col);
          Col := 0
        end
      end;
    if not Colocado then
    begin
      if Cp^.Nr <> 0 then
      begin
        Liberar(Colibre, Ddch, Dizq, Cp^.Nr, Cp^.Col);
        Nr := Cp^.Nr - 1;
        Col := Cp^.Col;
      end;
      Pborrar(Cp)
    end
  end;
  if Pvacia(Cp) then
    writeln ('NO ENCUENTRO SOLUCION')
  else
    Listarpila(Cp)
end.

if Pvacia(Cp) then
  writeln ('NO ENCUENTRO SOLUCION')
else
  Listarpila(Cp)
end.

```

9.5.4. Problema de la mochila

Un esforzado correo desea llevar en su mochila exactamente v kilogramos, se tiene para elegir un conjunto de objetos de pesos conocidos. Se desea cargar la mochila con un peso que sea igual al objetivo.

El planteamiento del problema de la mochila: dado un conjunto de pesos $p_1, p_2, p_3, \dots, p_n$ (enteros positivos), estudiar si existe una selección de pesos que totalice exactamente un valor dado como objetivo V . Por ejemplo, si $V=12$ y los pesos son $4, 3, 6, 2, 1$, se pueden elegir el primero, el tercero y el cuarto, ya que $4 + 6 + 2 = 12$. La representación de los pesos se hace con un vector de números enteros.

```

const
  N = ... {máximo de pesos previsto}
type
  Vector = array[1..N] of integer;
var
  Pesos: Vector;

```

El algoritmo para solucionar el problema tiene como tarea básica añadir un peso nuevo, probar si con ese peso se alcanza la solución, o se avanza hacia la solución. Es una búsqueda sistemática de la solución hacia adelante. Si llegamos a una situación de «impasse», en la que no se consigue el objetivo por que siempre es superado, entonces se retrocede para eliminar el peso añadido y probar con otro peso para inspeccionar si a partir de él se consigue el objetivo. Esto es, realizar otra búsqueda sistemática.

La bolsa donde se meten los pesos viene representada por un tipo conjunto. Al añadir un peso, la anotación se realiza acumulando el peso y metiendo en la bolsa, el conjunto, el índice del peso. Para borrar la anotación se opera a la inversa.

CODIFICACIÓN

```

const
  N = ...; {máximo de pesos previsto}
type
  Vector = array[1..N] of integer;
var
  Pesos: Vector;
  Bolsa: set of 1..N;
  Solucion: boolean;
  V: integer; {Objetivo}
  {V           : Es el objetivo          }
  {Candidato   : Índice del peso a añadir }
  {Cuenta      : Suma parcial de pesos   }

procedure Mochila(V: integer; Candidato:integer;
                      Cuenta: integer; var S: boolean);
begin
  if Cuenta = V then
    S := true
  else if (Cuenta < V) and (Candidato <= N) then
  begin
    {Es anotado el objeto Candidato y sigue la búsqueda}
    Bolsa := Bolsa + [Candidato];
    Mochila(V, Candidato + 1, Cuenta + Pesos[Candidato], S);
    if not S then {Es excluido Candidato, para seguir
                     tanteando con siguiente}
    begin
      Bolsa := Bolsa - [Candidato];
      Mochila(V, Candidato + 1, Cuenta, S)
    end
  end
end;

```

La llamada a Mochila desde el bloque principal transmite el objetivo V, el candidato 1 y la suma de pesos 0.

```
Mochila(V, 1, 0, Solucion);
```

Eliminación de la recursión en el problema de la mochila

Para eliminar la recursividad introducimos una pila de enteros para almacenar el candidato actual y en otra pila de números reales almacenar la suma parcial de pesos.

Las llamadas recursivas a Mochila se producen en dos puntos, por lo cual el problema de guardar la dirección de retorno lo solventamos con otra pila de enteros, en la que almacenamos tres posibles valores:

0. Indica que la llamada a Mochila procede de fuera del procedimiento.
1. Indica que es una llamada recursiva a Mochila, en la cual se incluye Pesos [Candidato] dentro de la solución.
2. Indica que es la llamada recursiva a Mochila, consecuencia de la vuelta atrás al no alcanzarse la solución completa.

Para realizar el procedimiento no recursivo de Mochila hay que incorporar la unidad Pilas para números enteros y para números reales.

```
uses Pilaint, Pilareal;
procedure Mochila (V: integer; Candidato: integer; Cuenta: integer;
                  var S: boolean);
var
  Pcandi, Pstat: Pilint;
  Pcuent: Pilreal;
  St, C: integer;
  Pso: real;
begin
  S := false;
  Pcrear(Pcandi);
  Pilareal.Pcrear(Pcuent);
  Pcrear(Pstat);
  Pmeter(0, Pstat); {Es asignado el estado inicial}
repeat
  if Cuenta = V then
    S := true
  else if (Cuenta < V) and (Candidato <= N) then
  begin
    {Es anotado el objeto Candidato y sigue la búsqueda}
    Bolsa := Bolsa + [Candidato];
    Cuenta := Cuenta + Pesos[Candidato];
    Pilareal.Pmeter(Cuenta, Pcuent);
    Candidato := Candidato + 1;
    Pmeter(1, Pstat);
  end
  else if (Cuenta > V) and (Cima(Pstat) = 0) or
         (Candidato > N) then
```

```

begin
  Candidato := Candidato - 1;
  Pborrar(Pstat);
end
else if Cima(Pstat) = 1 then
begin {Se excluye al candidato y se vuelve a probar}
  Bolsa := Bolsa - {Candidato};
  Pilareal.Psacar(Cuenta, Pcuent);
  Candidato := Candidato + 1;
  Pborrar(Pstat);
  Pmeter(2, Pstat)
end
else if Cima(Pstat) = 2 then
begin
  Candidato := Candidato - 1;
  Pborrar(Pstat)
end
until Pvacia(Pstat) or S
end;

```

{Al haber dos unidades de pilas las referencias a las rutinas de la unidad Pilareal las cualificamos con su nombre.}

9.5.5. PROBLEMA DEL LABERINTO

Se desea simular el juego del laberinto. En el laberinto hay muros por los que no se puede pasar. El viajero debe moverse desde una posición inicial hasta la salida del laberinto. El laberinto está representado por una matriz de $N \times N$, los muros son representados en la matriz por celdas que contienen el carácter 0, los caminos por celdas con el carácter 1. La salida del laberinto viene indicada por la celda que contiene una S. La entrada al laberinto es única, sus coordenadas son conocidas.

La representación de datos es:

```

const
  N = 12; {tamaño del laberinto}
  Norte = 1;
  Sur = N;
  Oeste = 1;
  Este = N;
type
  Latitud = Norte..Sur;
  Longitud = Oeste..Este;
  Laberinto = array[Latitud, Longitud] of char;

```

ALGORITMO

El viajero parte de una casilla inicial. Desde una casilla puede moverse a cuatro posibles casillas (hacia el Norte, el Oeste, el Sur o el Este). El movimiento estará permitido si en la nueva casilla no hay un «muro». Cada vez que se pasa por una casilla se marca con un '*' para así saber el camino seguido hasta la salida. Si llegamos a una situación de «impasse», es decir estamos en una casilla desde la cual ya no podemos avanzar: se desanota la casilla, se marca la casilla como si fuera un muro para no volver a pasar por

ella, se vuelve al anterior movimiento y se ensaya con un movimiento en otra dirección. El algoritmo termina cuando se alcanza la casilla de salida.

CODIFICACIÓN

```

program Juego_laberinto;
uses
  crt;
const
  N = 12;
  Norte = 1;
  Sur = N;
  Oeste = 1;
  Este = N;
type
  Latitud = Norte..Sur;
  Longitud = Oeste..Este;
  Laberinto = array[Latitud, Longitud] of char;
var
  L: Laberinto;
  Exito: boolean;
  A0, Af: Latitud;
  L0, Lf: Longitud;

procedure Entrada(var L: Laberinto);
var
  Lat: Latitud;
  Lon: Longitud;
  K, Km: integer;
begin
  Clrscr;
  writeln ('Simulación de laberinto de ',N,'x', N);
  for Lat := Norte to Sur do
    for Lon := Oeste to Este do
      L[Lat, Lon] := '1';
  repeat
    write('Número de "muros" (8..20): ');
    readln(K);
  until K in [8..20];
  Km := 0;
  randomize;
  while Km < K do
    begin
      Lat := 1 + random(Sur);
      Lon := 1 + random(Este);
      if L[Lat, Lon]= '1' then
        begin
          L[Lat, Lon] := '0';
          Km := Km + 1
        end
    end
  end;
end;

procedure VerLab(var L: Laberinto);

```

```

var
  Lat: Latitud;
  Lon: Longitud;
begin
  Clrsqr;
  for Lat := Norte to Sur do
    begin
      for Lon := Oeste to Este do
        write(L[Lat, Lon]:2);
      writeln
    end
  end;

procedure Soluclaber(A:Latitud; H:Longitud; var Q:boolean);
begin
  if L[A,H] = 'S' then {Hemos llegado a la salida}
    Q := true
  else begin
    L[A, H] := '**';
    {Tanteo sistemático: avanza probando Este, Sur, Oeste, Norte}
    if (H + 1) in [Oeste..Este] then {Al Este}
      if L[A, H + 1] = '1' then
        Soluclaber(A, H + 1, Q);
      if not Q then
        begin
          L[A, H + 1] := '0';
          if (A + 1) in [Norte..Sur] then {Al Sur}
            if L[A+1, H] = '1' then
              Soluclaber(A+1, H, Q);
          end;
        if not Q then
          begin
            L[A+1, H] := '0';
            if (H-1) in [Oeste..Este] then {Al Oeste}
              if L[A, H-1] = '1' then
                Soluclaber(A, H-1, Q);
            end;
          if not Q then
            begin
              L[A, H-1] := '0';
              if (A - 1) in [Norte..Sur] then {Al Norte}
                if L[A-1,H] = '1' then
                  Soluclaber(A-1,H,Q);
            end;
          end;
        end;
      L[A-1,H] := '0'
    end;
  end;
begin
  Entrada(L);
  VerLab(L);
  write('Casilla de entrada: ');
  readln(A0,L0);
  L[A0,L0] := 'E';
  write('Casilla de salida: ');
  readln(Af,Lf);

```

```

L[Af,Lf] := 'S';
Exito := false;
Soluclaber(A0,L0, Exito);
if Exito then
  Verlab(L)
else
  writeln(';; Horror no salimos del laberinto!!!')
end.

```

9.5.6. Generación de las permutaciones de n elementos

La resolución está basada en la ley de formación de las permutaciones de n elementos. Partiendo de las Permutaciones de 0 elementos se obtienen las Permutaciones monarias tomando 1(k) elemento y situándolo en todas las posiciones posibles: $p = 1$. Las Permutaciones binarias se obtienen tomando el 2(k) elemento y situándolo en todas las posiciones posibles: $p = 2, p = 1$. En definitiva, en el paso i el elemento i debe colocarse en las $p = i, p = i - 1, p = i - 2, \dots, p = 1$. Para ello el array A está indexado de 0 a n , guardando en la posición p el ordinal de la permutación (k) y en la posición k , ordinal de la anterior permutación. El primer valor de $p = 0$. Cuando se alcanza $k = n$ es escrito el contenido de la permutación y después el «elemento» k es colocado en A[p] para así «colocar» el elemento k en posición anterior. La solución se plantea recursivamente, aplicando una variación a la estrategia de *backtracking* para así agotar todas las «soluciones» que ahora se convierten en grupos de n elementos.

CODIFICACIÓN

```

program Generadorpermutaciones;
uses crt;
const
  Maxgrado = 10;
  B: array [0..Maxgrado] of char =
    ('X','A','B','C','D','E','F','G','H','I','J');
type
  Apuntador = 0..Maxgrado;
var
  A: array [Apuntador] of Apuntador;
  N: integer;
  X: char;
procedure procesopermutacion;
var
  q:Apuntador;
begin
  q :=0;
  while A[q] <> 0 do
  begin
    write(B[A[q]]);
    q :=A[q];
  end;
  write('':10-N)
end;
procedure permuta(K,N:Apuntador);

```

```

var
  p:Apuntador;
begin
  p :=0;
repeat
  A[K] :=A[p];
  A[p] :=K;
  if K = N then
    procesopermutacion
  else
    permuta(K+1,N);
    {Ahora se "coloca" en la anterior, representado por A[K]}
    A[p] :=A[K];
    p := A[p];
until p = 0;
writeln;
end;

begin
  clrscr;
  repeat
    write('Número de elementos a permutar:');readln(N);
    until N in [1..Maxgrado];
  writeln('Permutaciones de ':50,N,'elementos');
  A[0] := 0;
  Permuta(1, N);
  X := readkey
end.

```

9.6. PROBLEMA DE LA SELECCIÓN ÓPTIMA

En los problemas del Salto de caballo, Ocho reinas y Mochila se ha aplicado la estrategia de vuelta atrás para encontrar una única solución. Con la misma base, se ha hecho una ampliación para encontrar así todas las soluciones. Ahora no se trata de encontrar una situación fija o un valor predeterminado, sino de encontrar del conjunto de soluciones la óptima según unas restricciones definidas.

En términos reales, este es el problema del viajante que tiene que hacer las maletas seleccionando entre n artículos, aquellos cuyo valor total sea un máximo (lo óptimo, en este caso, es el máximo establecido) y su peso no exceda de una cantidad.

Seguimos aplicando la estrategia de vuelta atrás para generar todas las soluciones posibles, y cada vez que se alcance una solución guardarla si es mejor que las soluciones anteriores según la restricción definida.

```

if solucion then
  if mejor(solucion) then
    optimo :=solucion

```

La tarea básica en esta búsqueda sistemática es investigar si un objeto i es adecuado incluirlo en la selección que se irá acercando a una solución aceptable, y continuar la búsqueda con el siguiente objeto. En el caso de que lo que haya que hacer sea excluirlo

de la selección actual, el criterio para seguir con el proceso de selección actual es que el valor total todavía alcanzable después de esta exclusión no sea menor que el valor óptimo (máximo) encontrado hasta ahora. Cada tarea realiza las mismas acciones que la tarea anterior, por lo que puede expresarse recursivamente. Al estar buscando la selección óptima, hay que probar con todos los objetos del conjunto.

Consideraremos que el valor máximo alcanzable inicialmente es la suma de todos los valores de los objetos que disponemos. Debido a la restricción del peso, posiblemente el valor óptimo alcanzado sea menor.

9.6.1. El viajante de comercio

Suponemos que el viajante tiene 10 objetos. La entrada de datos es la información asociada con cada objeto: <Peso, Valor>. El peso máximo que puede ser transportado variará desde el mínimo peso hasta el peso total de los objetos, con un incremento de 3 y un máximo de 10 salidas.

El valor máximo que pueden alcanzar los objetos es la suma de los valores de cada uno, está representado por la variable *Tvalor*. El valor óptimo alcanzado en el proceso transcurrido está en la variable *Mvalor*.

Los parámetros del procedimiento recursivo son los necesarios para realizar una nueva tarea: *I*, número de objetos a probar; *Pt*, peso de la selección actual; *Va*, valor máximo alcanzable por la selección actual; y *Mvalor*, que es el valor máximo obtenido en el proceso transcurrido.

CODIFICACIÓN

```
program Optima(input,output);
uses crt;
const
  N = 10;
type
  Indice = 1..N;
  Objeto = Record
    val: integer;
    pso: integer
  end;
  Lista = array [Indice] of Objeto;
  Conj = set of Indice;
var
  A: Lista;
  Psomx, Tpeso,
  Tvalor, Mvalor,
  T, K: integer;
  Act, Opt: Conj;

procedure Objetos(var G:Lista; var Tv,Tp :integer);
var
  K:integer;
begin
  writeln;
```

```

Tv := 0; Tp := 0;
for K := 1 to N do
begin
  write('Objeto', K, '.', 'Peso y Valor:');
  readln(G[K].pso,G[K].val);
  Tv := Tv + G[K].val;
  Tp := Tp + G[K].pso
end
end;

function Min(A:Lista):integer;
var
  K,M:integer;
begin
  M := A[1].pso;
  for K := 2 to N do
    if A[K].pso < M then
      M := A[K].pso;
  Min := M
end;

procedure Solucion (S:Conj; P,V:integer);
  {Salida del conjunto de objetos}
var
  K:integer;
begin
  if S <> [ ] then
  begin
    write ('Seleccion óptima:');
    for K := 1 to N do
      if K in S then
        Write ('<',A[K].pso,'.',A[K].val,'>');
    writeln ('Peso:',P,'Valor:',V)
  end
end;
{El procedimiento recursivo, Maleta, es el que encuentra la selección
óptima}
procedure Maleta (I:indice; Pt,Va:integer;var Mvalor: integer);
var
  VaEx:integer;
begin
if Pt + A[I].pso <= Psomx then {El objeto I se incluye}
  begin
    Act := Act + [I];
    if I < N then
      Maleta (I+1,Pt+A[I].pso,Va, Mvalor)
    else if Va > Mvalor then {Todos los objetos han sido probados y}
      begin {se ha obtenido un nuevo valor óptimo}
        Opt := Act;
        Mvalor := Va
      end;
    Act := Act - [I] {Vuelta atrás para ensayar la exclusión}
  end;
{Proceso de exclusión del objeto I para seguir la búsqueda
sistemática con el objeto Y+1}

```

```

VaEx := Va - A[I].val;
{VaEx es el valor máximo que podría alcanzar la selección actual}
if VaEx > Mvalor then
  if I < N then
    Maleta (I+1, Pt, VaEx, Mvalor)
  else begin
    Opt := Act;
    Mvalor := VaEx
  end;
  {si VaEx < Mvalor es inútil seguir ensayando ya que no va a
  superar el valor óptimo actual: Mvalor}
end;

begin
  clrscr;
  writeln ('Entrada de ',N,'Objetos');
  Objetos (A, Tvalor, Tpeso);
  clrscr;
  write ('Peso:');
  for K := 1 to N do
    write (A[K].ps0:6);
  writeln;
  write ('valor:');
  for K := 1 to N do
    write (A[K].val:6);
  writeln;
  {El peso máximo irá variando desde el mínimo, hasta alcanzar el
  máximo o 10 pruebas, cada prueba aumenta el peso en 3 unidades}
  T := Min(A); Psomx := T;
repeat
  Act := [ ]; Opt := [ ];
  Mvalor := 0;
  Maleta (1,0,Tvalor,Mvalor);
  Solucion (Opt,Psomx,Mvalor);
  Psomx := Psomx+3
until (Psomx > Tpeso) or (Psomx > T+10*3)
end.

```

9.7. PROBLEMA DE LOS MATRIMONIOS ESTABLES

El planteamiento del problema es el siguiente: dados dos conjuntos A y B, disjuntos y con igual número de elementos, n , hay que encontrar un conjunto de n pares (a,b) , tales que a pertenece A, b pertenece a B y cumplan ciertas condiciones.

Una concreción de este planteamiento es el problema de los matrimonios estables. Ahora A es un conjunto de hombres y B un conjunto de mujeres. A la hora de elegir pareja, cada hombre y cada mujer tienen distintas preferencias. Se trata de formar parejas, matrimonios estables. De las n parejas, en cuanto exista un hombre y una mujer que no formen pareja pero que se prefieran frente a sus respectivas parejas, se dice que la asignación es inestable ya que tenderá a una ruptura para buscar la preferencia común. Si no existe ninguna pareja inestable se dice que la asignación es estable.

En el problema se hace la abstracción de que la lista de preferencias no cambia al hacer una asignación.

Este planteamiento caracteriza ciertos problemas en los que hay dos conjuntos de elementos y se ha de hacer una elección según una lista de preferencias. Así, pensemos en el conjunto de ofertas de vacaciones y el conjunto de turistas que quieren elegir una de ellas; la elección de facultad por los chicos de COU...

ALGORITMO

Una forma de buscar solución es aplicar la búsqueda sistemática de los algoritmos de «vuelta atrás». La tarea básica es encontrar una pareja para un hombre h según la lista de preferencias, esta tarea básica la realiza el procedimiento ensayar:

```
procedimiento Ensayar(H: TipoHombre)
var
  R: entero
  inicio
    Desde R ← 1 hasta n hacer
      <Tomar preferencia R-ésima de hombre H>
      si aceptable entonces
        <Anotar matrimonio>
        si <H no es el último hombre> entonces
          Ensayar(H+1)
        sino
          <El conjunto de matrimonios es estable>
          fin_Si
          <Cancelar el matrimonio>
          fin_si
    fin_Desde
fin
```

Ahora afrontamos la tarea de representación de datos. El Tipohombre, Tipomujer son representados por un subrango entero; por tanto, hacemos la abstracción de representar tanto un hombre como una mujer por un número que es el ordinal del rango de hombres o de mujeres. Para representar las preferencias de los hombres por las mujeres se utiliza una matriz de mujeres. Recíprocamente, las preferencias de las mujeres por los hombres son representadas por una matriz de hombres.

La solución del problema ha de ser tal que nos muestre las parejas (hombre-mujer) que forman la asignación estable. Dos vectores, uno de las parejas de los hombres y el otro de parejas de las mujeres, representan la solución.

```
const
  N = ... ; {Número de parejas}
type
  Tipohombre = 1..N;
  Tipomujer = 1..N;
  Prefhombres = array [Tipohombre,1..N] of Tipomujer;
  Prefmujeres = array [Tipomujer,1..N] of Tipohombre;
  Parhombres= array [Tipohombre] of Tipomujer;
```

```

Parmujeres= array [Tipomujeres] of Tipohombre;
var
  Phb: Prefhombres;
  Pmj: Prefmujeres;
  V: Parhombres;
  F: Parmujeres;

```

La información representada por V y F determina la estabilidad de un conjunto de matrimonios. Este conjunto se construye paso a paso casando hombre-mujer y comprobando la estabilidad después de cada propuesta de matrimonio. Para facilitar la inspección de las parejas ya formadas utilizamos un array lógico:

```
Soltera: array[Tipomujer] of boolean;
```

Soltera[J] a true quiere decir que la mujer J todavía no ha encontrado pareja. Para determinar si un hombre K ha encontrado pareja puede utilizarse otro array similar al de Soltera, o bien sencillamente si $K < H$ es que K ya encontró pareja (H es el hombre actual que busca pareja).

Con todas estas consideraciones de tipos de datos ya podemos proponer una solución más elaborada:

```

procedimiento Ensayar(H: TipoHombre)
var
  R: entero;
  M: Tipomujer;
inicio
  Desde R ← 1 hasta N hacer
    M ← Phb[H,R]           {Mujer candidata}
    si Soltera[M] y <Estable> entonces
      V[H]← M
      F[M]← H
      Soltera[M]← False
      si H < N entonces
        Ensayar(H+1)
      sino
        <Anotar, conjunto ya es estable>
      fin_si
      Soltera[M] ← True
    fin_si
  fin_desde
fin_procedimiento

```

La acción más importante que nos falta es determinar la estabilidad que ha sido expresada como <Estable>. Pues bien, la buscada estabilidad se encuentra por comparaciones entre las distintas preferencias. Las preferencias vienen dadas por el rango de 1 a N. Para facilitar el cálculo de la estabilidad son definidas dos matrices:

```
Rhm: array [Tipohombre,Tipomujer] of 1..N;
Rmh: array [Tipomujer,Tipohombre] of 1..N;
```

tal que $R_{hm}[H, M]$ contiene el orden que ocupa, o rango, de la mujer M en la lista de preferencias del hombre H . De igual forma, $R_{mh}[M, H]$ contiene el orden del hombre H en la lista de preferencias de la mujer M . Ambas matrices se determinan inicialmente a partir de las matrices de preferencias.

El predicado <Estable> parte de la hipótesis de que lo normal es la estabilidad y se buscan posibles fuentes de perturbarla. Recordemos que se está analizando la estabilidad de la posible pareja H y M , siendo M la mujer que ocupa la posición R en la lista de preferencias de H ($M \leftarrow P_{hb}[H, R]$). Hay dos posibilidades:

1. Puede haber otra mujer M_m , por la que H tenga más preferencia que M , y a su vez M_m prefiera más a H que a su actual pareja.
De forma simétrica.
2. Puede haber otro hombre H_h , por el que M tenga más preferencia que a H , y a su vez ese hombre prefiera más a M que a su actual pareja.

La posibilidad 1 se dilucida comparando los rangos $R_{hm}[M_m, H]$ y $R_{mh}[M_m, F[M_m]]$ para todas las mujeres más preferidas que M por el hombre H . Es decir, para todas las $M_m \leftarrow P_{hb}[H, I]$ tales que $I < R$. Además, todas estas mujeres tienen ya pareja, ya que si alguna estuviera soltera el hombre H la hubiera elegido ya. Esta posibilidad puede expresarse:

```

Estable ← true
I ← 1
mientras (I < R) y Estable hacer
    Mm ← P_{hb}[H, I]
    I ← I + 1
    si no soltera[Mm] entonces
        Estable ← R_{mh}[Mm, H] < R_{mh}[Mm, F[Mm]] {Es estable si prefiere más }
                                                {a su actual pareja}
    fin_si
fin_mientras

```

Para analizar la posibilidad 2 razonamos de forma simétrica. Hay que investigar a todos los hombres H_h por los cuales tiene más predilección la mujer M que a su pareja actual H :

```

I ← 1
Tope ← R_{mh}[M, H]
mientras (I < Tope) y Estable hacer
    Hh ← P_{mj}[M, I]
    I ← I+1
    si Hh < H entonces { Hh ya tiene pareja }
        Estable ← R_{hm}[Hh, M] > R_{hm}[Hh, V[Hh]] {Estable si Hh prefiere }
                                                { a su actual pareja.}
    fin_si
fin_mientras

```

La codificación de estas dos posibles fuentes de inestabilidad la realizamos en una función lógica anidada al procedimiento ensayar.

CODIFICACIÓN

```

program MatrimonioEstable;
uses
  crt;
const
  N = 9;  {Número de parejas a formar}
type
  Tipohombre = 1 .. N;
  Tipomujer = 1 .. N;
  Prefhombres = array [Tipohombre, 1..N] of Tipomujer;
  Prefmujeres = array [Tipomujer, 1..N] of Tipohombre;
  Parhombres = array [Tipohombre] of Tipomujer;
  Parmujeres = array [Tipomujer] of Tipohombre;
  Flags = array[Tipomujer] of boolean;
  Matranghb = array[Tipohombre, Tipomujer] of 1..N;
  Matrangmj = array[Tipomujer, Tipohombre] of 1..N;
var
  Phb: Prefhombres;
  Pmj: Prefmujeres;
  V: Parhombres;
  F: Parmujeres;
  Soltera: Flags;
  Rhm: Matranghb;
  Rmh: Matrangmj;
  H: Tipohombre;
  M: Tipomujer;
  R: integer;

procedure Escribir;
var
  H: Tipohombre;
begin
  write('Solución: ');
  for H := 1 to N do
    write('<', H, ',', M, ', V[H], >; ');
  writeln
end;

procedure Ensayar(H: Tipohombre);
var
  R: integer;
  M: Tipomujer;
function Estable(H:Tipohombre;M:Tipomujer;R:integer):boolean;
var
  Hh: Tipohombre;
  Mm: Tipomujer;
  I, Tope: integer;
  Es: boolean;
begin
  Es := true;
  I := 1;
  while (I < R) and Es do
  begin
    Mm := Phb[H, I];
    I := I+1;
    if Mm <= M then Es := false;
  end;
  if Es then
    Es := (R = Tope);
end;

```

```

if not Soltera[Mm] then
  Es := Rmh[Mm, H] < Rmh[Mm, F[Mm]]
end;
I := 1;
Tope := Rmh[M, H];
while (I < Tope) and Es do
begin
  begin
    Hh := Pmj[M, I];
  I := I + 1;
    if Hh < H then
      Es := Rhm[Hh, M] > Rhm[Hh, V[Hh]]
  end;
  Estable := Es
end;
begin {Ensayar}
  for R := 1 to N do
  begin
    M := Phb[H, R];
    if Soltera[M] and Estable(H, M, R) then
    begin
      V[H] := M;
      F[M] := H;
      Soltera[M] := false;
      if H < N then
        Ensayar(H + 1)
      else {Encontrada una asignación estable }
        Escribir;
      Soltera[M] := true
    end
  end
end;
begin {bloque principal}
  clrscr;
  {Entrada de rango de preferencias de hombres}
  for H := 1 to N do
  begin
    writeln('Preferencias de hombre ', H, ' según rango de 1 a ', N);
    for R := 1 to N do
    begin
      read(Phb[H, R]);
      Rhm[H, Phb[H, R]] := R
    end
  end;
  {Entrada de rango de preferencias de mujeres}
  for M := 1 to N do
  begin
    writeln('Preferencias de mujer ', M, ' según rango de 1 a ', N);
    for R := 1 to N do
    begin
      read(Pmj[M, R]);
      Rmh[M, Pmj[M, R]] := R
    end
  end;
  for M := 1 to N do
    Soltera[M] := true;
  Ensayar(1)
end.

```

RESUMEN

La recursión (recursividad) permite resolver problemas cuyas soluciones iterativas son difíciles de conceptualizar: un procedimiento o función que se llama a sí misma. Normalmente, una solución recursiva es menos eficiente en términos de tiempo y operaciones suplementarias que entrañan las llamadas extra a procedimientos; sin embargo, en numerosas ocasiones el uso de la recursión permite especificar una solución muy natural y sencilla a problemas que en caso contrario serían muy difíciles de resolver. Por esta razón, la recursividad es una herramienta muy importante y potente para la resolución de problemas con programación.

El uso apropiado de la recursividad se debe considerar antes de su utilización. Una vez tomada la decisión, el programador debe tener mucha precaución en *proporcionar condiciones de terminación para detener las llamadas recursivas*. Es muy fácil entrar en un bucle infinito si no se incluyen condiciones de terminación adecuadas.

Algunos lenguajes de programación no permiten la recursividad. Por consiguiente, un programador puede desear resolver un problema utilizando técnicas recursivas mediante el uso de pseudocódigo. Incluso, como en el caso de Pascal, que soporta recursividad, el programador puede tratar de intentar reducir las operaciones y el tiempo auxiliar implicado en el proceso recursivo simulando la recursividad.

EJERCICIOS

9.1. Suponer que la función G está definida recursivamente de la siguiente forma:

$$G(x, y) = \begin{cases} 1 & \text{si } x \leq y \\ G(x - y + 1) + 1 & \text{si } y < x \end{cases}$$

Siendo x, y enteros positivos.

- a) Encontrar el valor de $G(8,6)$.
- b) Encontrar el valor de $G(100,10)$.

9.2. Sea $H(x)$ una función definida recursivamente $\forall x > 0$, siendo x un entero:

$$H(x) = \begin{cases} 1 & \text{si } x = 1 \\ H(x / 2) + 1 & \text{si } x > 1 \end{cases}$$

- a) Encontrar el valor de $H(80)$.
- b) ¿Cómo podemos describir lo que hace esta función?

9.3. Definimos $C(n,k)$ como el número de combinaciones de n elementos agrupados de k en k , es decir, el número de los diferentes grupos que se pueden formar con k miembros dado un conjunto de n miembros para elegir. Así, por ejemplo, si $n = 4$ ($\{A,B,C,D\}$) $C(4,2) = 6$ que serán: (A,B), (A,C), (A,D), (B,C), (B,D), (C,D). matemáticamente $C(n,k)$ podemos definirla:

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

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

$$C(n,k) = C(n - 1, k - 1) + C(n - 1, k) \quad \forall n > k, k > 1$$

- a) Encontrar el valor de $C(8,5)$.
 b) Escribir una función recursiva para calcular $C(n,k)$.
- 9.4. Eliminar la recursividad de la función $C(n,k)$. Escribir la función $C(n,k)$ de forma iterativa utilizando una pila para eliminar la recursividad.
- 9.5. Dada la siguiente función recursiva:

```
function Reves(N: integer): char;
  var C: char;
begin
  read(C);
  if eoln then
    Reves := '*'
  else
    Reves := Reves(1);
  write(C)
end;
```

- a) ¿Qué hace la función?
 b) Hacer un seguimiento con esta llamada: $Reves(1)$.

- 9.6. Dado el siguiente procedimiento recursivo:

```
procedure Desconocido(N, Despl: integer);
var
  K: integer;
begin
  if N > 0 then
  begin
    Desconocido(N-1, Despl+1);
    for K := 1 to Despl do
      write(' ');
    for K := 1 to Despl do
      write(' | ');
    writeln;
    Desconocido(N-1, Despl+1)
  end
end;
```

- Hacer un seguimiento del procedimiento para la llamada $Desconocido(4,1)$.
- 9.7. Eliminar la recursividad del procedimiento escrito en 9.6. Escribir de nuevo el procedimiento $Desconocido$ del ejercicio 9.6 utilizando una pila.
- 9.8. Realizar una función recursiva que calcule la función de Ackermann definida de la siguiente forma:

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

- 9.9. Escribir la función de Ackermann eliminando la recursividad.
- 9.10. La resolución recursiva de las Torres de Hanoi ha sido realizada con dos llamadas recursivas. Volver a escribir el procedimiento de resolución con una sola llamada recursiva.

Nota: Sustituir la última llamada por un bucle repetir-hasta.

PROBLEMAS

- 9.1. En el problema de las 8 reinas se encuentran 92 soluciones diferentes. Hacer los cambios necesarios en el procedimiento de resolución para que nos muestre únicamente las soluciones no simétricas.
- 9.2. Escribir un programa que tenga como entrada una secuencia de números enteros positivos (mediante una variable entera). El programa debe de hallar la suma de los dígitos de cada entero y encontrar cuál es el entero cuya suma de dígitos es mayor. La suma de dígitos ha de ser con una función recursiva.
- 9.3. Sea A una matriz cuadrada de $n \times n$ elementos, el determinante de A podemos definirlo de manera recursiva:
- Si $n = 1$ entonces $\text{Deter}(A) = a_{1,1}$.
 - 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.

Podemos expresarlo:

$$\text{Det}(A) = \sum_{i=1}^n (-1)^{i+j} * A[i,j] * \text{Det}(\text{Menor}(A[i,j])); \text{ para cualquier columna } j$$

o

$$\text{Det}(A) = \sum_{j=1}^n (-1)^{i+j} * A[i,j] * \text{Det}(\text{Menor}(A[i,j])); \text{ para cualquier fila } i$$

Se observa que la resolución del problema sigue la estrategia de los algoritmos divide y vence.

Escribir un programa que tenga como entrada los elementos de la matriz A, y tenga como salida la matriz A y el determinante de A. eligiendo la fila 1 para calcular el determinante.

- 9.4. Escribir un programa que transforme números enteros en base 10 a otro en base B. Siendo la base B de 8 a 16. La transformación se ha de realizar siguiendo una estrategia recursiva.
- 9.5. Escribir un programa para resolver el problema de la subsecuencia creciente más larga. La entrada es una secuencia de n números $a_1, a_2, a_3, \dots, a_n$; hay que encontrar la subsecuencia más larga $a_{i1}, a_{i2}, \dots, a_{ik}$ tal que $a_{i1} < a_{i2} < a_{i3} \dots < a_{ik}$ y que $i1 < i2 < i3 < \dots < ik$. 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.
- 9.6. El sistema monetario consta de monedas de valor $p_1, p_2, p_3, \dots, p_n$ (orden creciente) pesetas. Escribir un programa que tenga como entrada el valor de las n monedas en pesetas, en orden creciente, y una cantidad X pesetas de cambio. Calcule:

- El número mínimo de monedas que se necesitan para dar el cambio X.
- Calcule el número de formas diferentes de dar el cambio de X pesetas con la p_i monedas. Aplicar técnicas recursivas para resolver el problema.

- 9.7. Dadas las m primeras del alfabeto, escribir un programa que escriba las diferentes agrupaciones que se pueden formar con n letras ($n < m$) cada una diferenciándose una agrupación de otra por el orden que ocupan las letras, o bien por tener alguna letra diferente (en definitiva, formar variaciones Vm,n).

- 9.8. En un tablero de ajedrez se coloca un alfil en la posición (x_0, y_0) y un peón en la posición $(1, j)$, siendo $1 \leq j \leq 8$. Se pretende encontrar una ruta para el peón que llegue a la fila 8 sin ser comido por el alfil. Siendo el único movimiento permitido para el peón el de avance desde la posición (i, j) a la posición $(i + 1, j)$. Si se encuentra que el peón está amenazado por el alfil en la posición (i, j) , entonces debe de retroceder a la fila 1, columna $j + 1$ o $j - 1$ $\{(1, j + 1), (1, j - 1)\}$. Escribir un programa para resolver el supuesto problema. Hay que tener en cuenta que el alfil ataca por diagonales.
- 9.9. Dados n números enteros positivos, encontrar combinación 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.
- 9.10. Dados n números, encontrar combinación con sumas o restas que más se aproxime a un objetivo Z . La aproximación puede ser por defecto o por exceso. La entrada son los n números y el objetivo y la salida la combinación más próxima al objetivo.
- 9.11. Dados n números encontrar, si existe, la combinación con los n números que mediante sumas o restas totalice exactamente el objetivo Z .
- 9.12. Dados n números, encontrar la combinación con los n números que mediante sumas o restas más se aproxime a el objetivo Z .
- 9.13. Un laberinto podemos emularlo con una matriz $n \times n$ 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.
- 9.14. Realizar las modificaciones necesarias en el problema del laberinto 9.12 para encontrar la ruta más corta. Considerando ruta más corta la que pasa por un menor número de casillas.
- 9.15. 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 tenga como entrada un par de pueblos (x,y) y encuentre 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.

- 9.16. En el programa escrito en 9.14, hacer las modificaciones necesarias para encontrar todos los caminos posibles entre el par de pueblos (x,y) .
- 9.17. Referente al problema 9.14, escribir un programa que genere una matriz P de $n \times n$ en la que cada elemento $P(i,j)$ contiene el camino más corto entre el pueblo i y el pueblo j . Utilizar únicamente técnicas recursivas.
- 9.18. El celebre presidiario Señor S quiere fugarse de la cárcel de Carabanchel, por el sistema de alcantarillado, pero tiene dos problemas:

- 1) El diámetro de la bola que arrastra es de 50 cm, resulta demasiado grande para pasar por algunos pasillos.
- 2) Los pasillos que comunican unas alcantarillas con otras tienen demasiada pendiente y sólo puede circular por ellos en un sentido.
Ha conseguido hacerse con los planos que le indican el diámetro de salida de las alcantarillas, así como el diámetro de los pasillos y el sentido de la pendiente que lo conectan.

Suponiendo que:

- 1) El Número de alcantarillas es N .
- 2) El Señor S se encuentra inicialmente en la alcantarilla 1 (dentro de la prisión).
- 3) Todas las alcantarillas (excepto la 1) tienen su salida fuera de la prisión, si bien puede que sean demasiado «estrechas» para sacar la bola fuera.
- 4) Se puede pasar de un pasillo a otro a través de las alcantarillas «estrechas» aunque a través de dichas alcantarillas no se pueda salir al exterior.
- 5) Todos los pasillos tienen un diámetro, si bien éste puede ser demasiado «estrecho» para poder pasar con la bola.

Escribir un programa que contenga al menos estos procedimientos:

- a) Un procedimiento para generar aleatoriamente los diámetros de los túneles del alcantarillado así como los diámetros de salida de las distintas alcantarillas. (Tenga en cuenta que si se puede ir de la alcantarilla i a la j por el túnel, entonces no se puede ir por el túnel desde j hasta i .)
- b) Un procedimiento de backtracking que resuelva el problema del señor S.
- c) Procedimientos para escribir en pantalla el sistema de conexiones así como la posible solución encontrada.

Árboles binarios

CONTENIDO

- 10.1. Concepto de árbol.
- 10.2. Árboles binarios.
- 10.3. Árboles de expresión.
- 10.4. Construcción de un árbol binario.
- 10.5. Recorrido de un árbol.
- 10.6. Aplicación de árboles binarios: evaluación de expresiones.
- 10.7. Árbol binario de búsqueda.
- 10.8. Operaciones con árboles binarios de búsqueda.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

REFERENCIAS BIBLIOGRÁFICAS.

En este capítulo se centra la atención sobre una estructura de datos, el *árbol*, cuyo uso está muy extendido y es muy útil en numerosas aplicaciones. Se definen formas de esta estructura de datos (árboles generales, árboles binarios y árboles binarios de búsqueda) y cómo se pueden representar en Pascal, así como el método para su aplicación en la resolución de una amplia variedad de problemas. Al igual que ha sucedido anteriormente con las listas, los árboles se tratan principalmente como estructura de datos en lugar de como tipos de datos. Es decir, nos centraremos principalmente en los algoritmos e implementaciones en lugar de en definiciones matemáticas.

Los árboles junto con los grafos constituyen estructuras de datos no lineales. Las listas enlazadas tienen grandes ventajas o flexibilidad sobre la representación contigua de estructura de datos (los *arrays*), pero tienen una gran debilidad: son listas secuenciales; es decir, están dispuestas de modo que es necesario moverse a través de ellas, una posición cada vez.

Los árboles superan estas desventajas utilizando los métodos de punteros y listas enlazadas para su implementación. Las estructuras de datos organizadas como árboles serán muy valiosas en una gama grande de aplicaciones, sobre todo problemas de recuperación de información.

10.1. CONCEPTO DE ÁRBOL

Un **árbol** (en inglés, *tree*) es una estructura que organiza sus elementos, denominados **nodos**, formando jerarquías. Los científicos utilizan los árboles generales para representar relaciones. Fundamentalmente, la relación clave es la de «*padre-hijo*» entre los nodos del árbol. Si existe una arista (rama) dirigida del nodo *n* al nodo *m*, entonces *n* es el **padre** de *m* y *m* es un **hijo** de *n*. En el árbol de la Figura 10.1, los nodos B y C son hijos del nodo A. Los hijos del mismo padre se llaman **hermanos**, por ejemplo B y C. Cada nodo de un árbol tiene al menos un parente, y existe un único nodo, denominado **raíz** del árbol, que no tiene parente. El nodo A es el **raíz** del árbol de la Figura 10.1. Un nodo que no tiene hijos se llama **hoja** del árbol. Las hojas del árbol de la Figura 10.1 son C, D, E y F.

Terminología básica raíz padre hijo hermano hoja

La relación padre-hijo entre los nodos se generaliza en las relaciones **ascendente** (antecesor) y **descendiente**. En la Figura 10.1 A es un antecesor de D, y por consiguiente D es un descendiente de A. Obsérvese que no todos los nodos están relacionados por las relaciones ascendente/descendiente: B y C, por ejemplo, no están relacionados. Sin embargo, el **raíz** de cualquier árbol es un ascendiente de todos los nodos de ese árbol. Un **subárbol** de un **árbol** es cualquier nodo del árbol junto con todos sus descendientes. Un **subárbol** de un nodo *n* es un subárbol enraizado en un hijo de *n*. Por ejemplo, la Figura 10.2 muestra un subárbol de la Figura 10.1. Este subárbol tiene a B como su **raíz** y es un subárbol del nodo A.

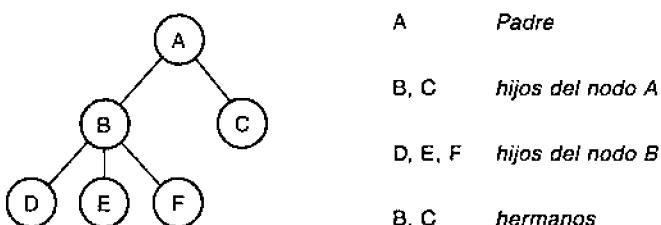


Figura 10.1. Árbol general.

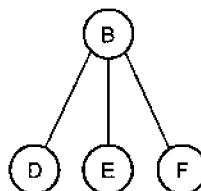


Figura 10.2. Un subárbol del árbol de la Figura 10.1.

Debido a la naturaleza jerárquica de los árboles, se puede utilizar para representar en formación que sea jerárquica por naturaleza, por ejemplo, diagramas de organizaciones, árboles genealógicos, árboles de especies animales, etc.

Terminología complementaria

Además de los términos ya citados anteriormente, existen otros también de gran importancia.

Camino	Una secuencia de nodos conectados dentro de un árbol.
Longitud del camino	Es el número de nodos menos uno ($r - 1$). Si $r > 0$, se dice que el camino es propio.
Altura del árbol ¹	Es el nivel más alto del árbol. La altura es igual a la longitud del camino desde el nodo raíz a la hoja más lejana que sea alcanzable desde él. Por ejemplo, la altura del árbol de la Figura 10.3 es 4. Un árbol que contiene sólo un raíz, tiene de altura 1.
Nivel (profundidad)	De un árbol (<i>level</i> o <i>depth</i>), es el número de nodos que se encuentra entre él y la raíz. El nodo Luis Carchejo está en el nivel 3. Por definición el número de niveles de un árbol se define como el nivel de la hoja más profunda; así, el número de niveles del árbol de la Figura 10.3 es 4. Observemos que por definición, el número de niveles de un árbol es igual a la altura por lo que pueden usarse ambas magnitudes indistintamente.
Grado (aridad)	Es el número de hijos del nodo. La <i>aridad</i> de un árbol se define como el máximo de la aridad de sus nodos.
Hermanos	Dos nodos son hermanos si tienen el mismo padre. Se llamarán <i>hermano izquierdo</i> de n y <i>hermano derecho</i> de n , respectivamente.

Un tipo especial de árbol es el denominado **árbol binario** y un árbol binario específico de gran utilidad es el **árbol binario de búsqueda**.

¹ Las definiciones de altura, profundidad y nivel, como se señala en [Franch 93], pág. 220, son contradictorias en algunos textos, que no definen algunos de estos conceptos o los definen sólo para un árbol y no para sus nodos; o bien empiezan a numerar a partir del cero y no del nodo, etc. En nuestro caso y al igual que hace Franch preferimos numerar a partir del uno para que el árbol vacío tenga una altura diferente al árbol con un único nodo.

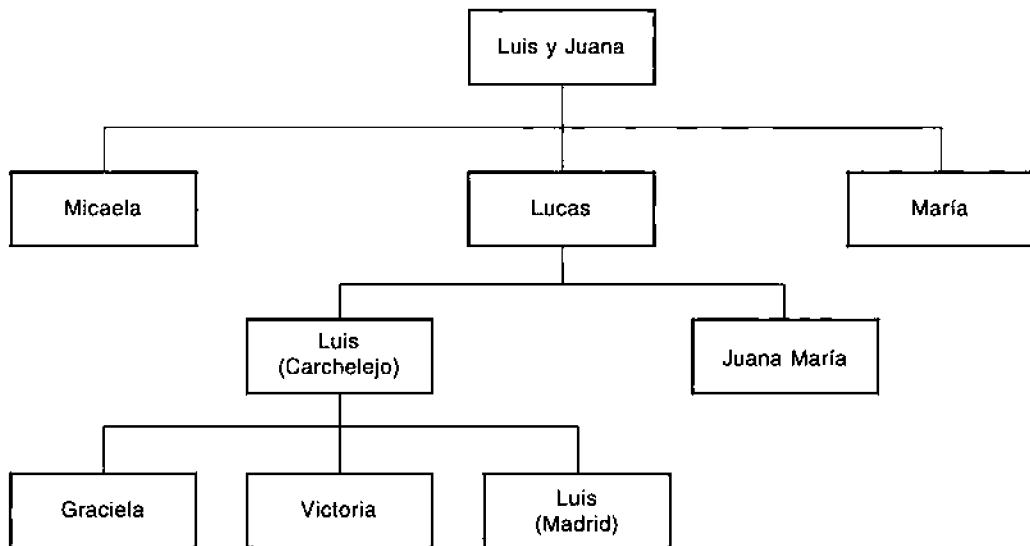


Figura 10.3. Árbol geneatógico.

10.2. ÁRBOLES BINARIOS

Un **árbol binario** es un árbol en el que cada nodo no puede tener más de dos hijos o descendientes.

En particular, un árbol binario es un conjunto de nodos que es, o bien el conjunto vacío, o un conjunto que consta de un nodo **raíz** enlazado a dos árboles binarios disjuntos denominados **subárbol izquierdo** y **subárbol derecho**. Cada uno de estos subárboles es, a su vez, un árbol binario. La Figura 10.4 muestra diversos ejemplos de árboles binarios.

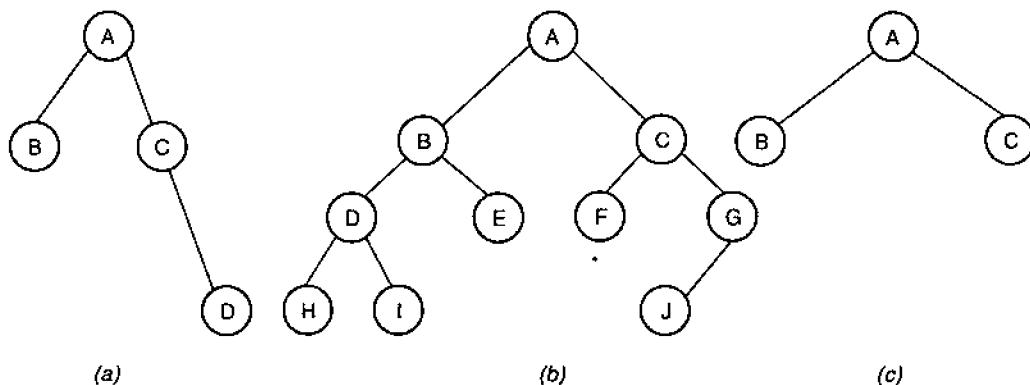


Figura 10.4. Diversos tipos de árboles binarios.

10.2.1. Terminología

En un árbol binario los hijos se conocen como *hijo izquierdo* e *hijo derecho*, lo que supone automáticamente una diferencia. Los dos árboles de la Figura 10.5 no representan el mismo árbol binario, ya que sus hijos izquierdo y derecho están en orden inverso.

Un nodo que no tiene hijos se denomina **hoja**.

Por consiguiente, los nodos H, I, E, F y J son hojas en el árbol de la Figura 10.4-b. Los nodos con descendientes se denominan **nodos interiores**.

El nodo raíz se dice que está en el nivel 1 en el árbol, los nodos B y C están en el nivel 2, y los nodos D, E, F y G están en el nivel 3. La *altura* del árbol se define como el nivel más alto del árbol. Por consiguiente, la altura del árbol (*b*) de la Figura 10.4 es 4.

Cualquier nodo sin sucesores se denomina un *nodo terminal*. Por ejemplo, los nodos H, I, E, F, J son todos hojas o nodos terminales, en el árbol 10.4 (*b*).

En la Figura 10.7, los nodos B, D y E forman el *subárbol izquierdo*. De modo similar C, F y G forman el *subárbol derecho*. Cada uno de estos subárboles es un verdadero árbol.

Los subárboles izquierdo y derecho de un árbol binario deben ser subconjuntos disjuntos de nodos. Esto es, ningún nodo puede estar en ambos subárboles.

Un árbol binario es un conjunto de nodos que es o bien vacío o consta de un nodo raíz y dos árboles binarios disjuntos denominados subárbol izquierdo y subárbol derecho.

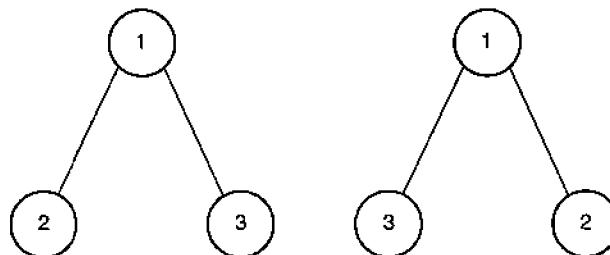


Figura 10.5. Árboles binarios distintos.

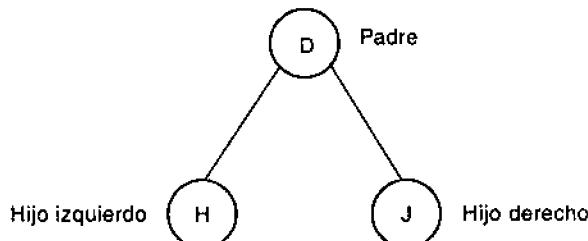


Figura 10.6. Padre e hijos de un árbol binario.

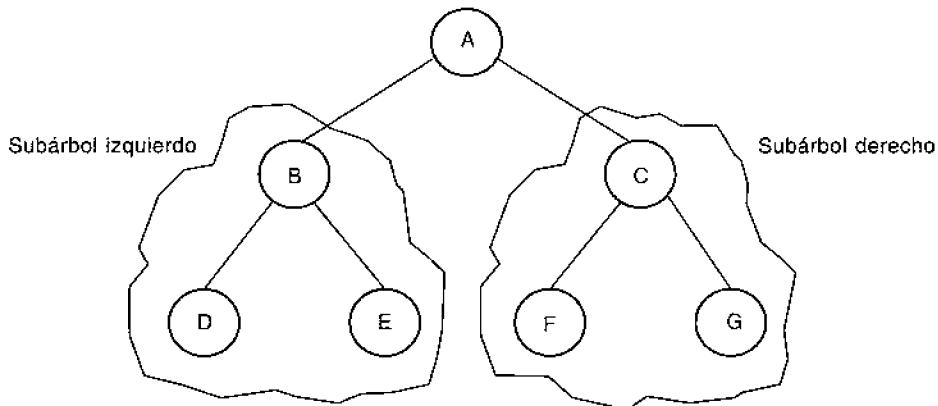


Figura 10.7. Subárboles de un árbol binario.

La definición de árbol conlleva el hecho de que un árbol binario es un tipo de estructura de datos recursiva. Esto es, cada subárbol se define como un árbol más simple. La naturaleza recursiva del árbol binario ayuda a simplificar la operación con árboles binarios.

Un **árbol binario lleno** es aquel en el que cada nodo tiene o dos hijos o ninguno si es una hoja. La Figura 10.8 (a) muestra un árbol binario lleno y, sin embargo, la Figura 10.8 (b) representa un árbol binario no lleno pero sí completo (se define posteriormente).

10.2.2. Nivel de un nodo y altura de un árbol

El **nivel** o **profundidad** de un nodo se define como una cantidad mayor en uno al número de sus ascendientes. Así, suponiendo el nivel de nodo n :

- Si n es la raíz de un árbol T , entonces está en el nivel 1.
- Si n no es la raíz de T , entonces su nivel es mayor que el nivel de su padre.

Por ejemplo, en la Figura 10.9

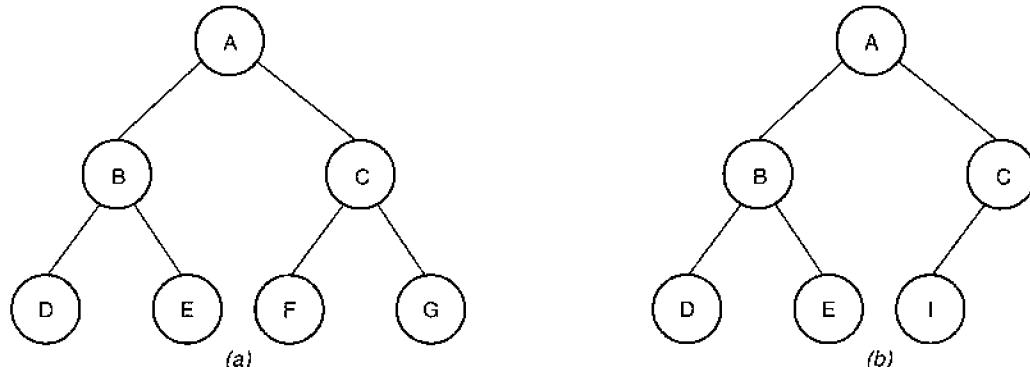


Figura 10.8. Árbol binario: (a) completo; (b) no completo.

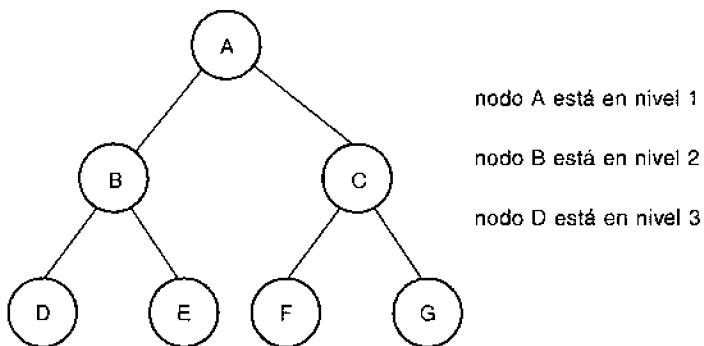


Figura 10.9. Árbol binario con diferentes niveles.

La **altura** de un árbol es el número de nodos en el camino más largo desde la raíz a una hoja; dicho de otro modo, la altura de un árbol es el número de niveles distintos. Así, en un árbol general T, en términos de los niveles de sus nodos se define como sigue:

- Si T es vacío, entonces la altura es 0.
- Si T no es vacío, entonces su altura es igual al nivel máximo de sus nodos.

Los árboles de la Figura 10.10 tienen por altura 3, 4 y 6.

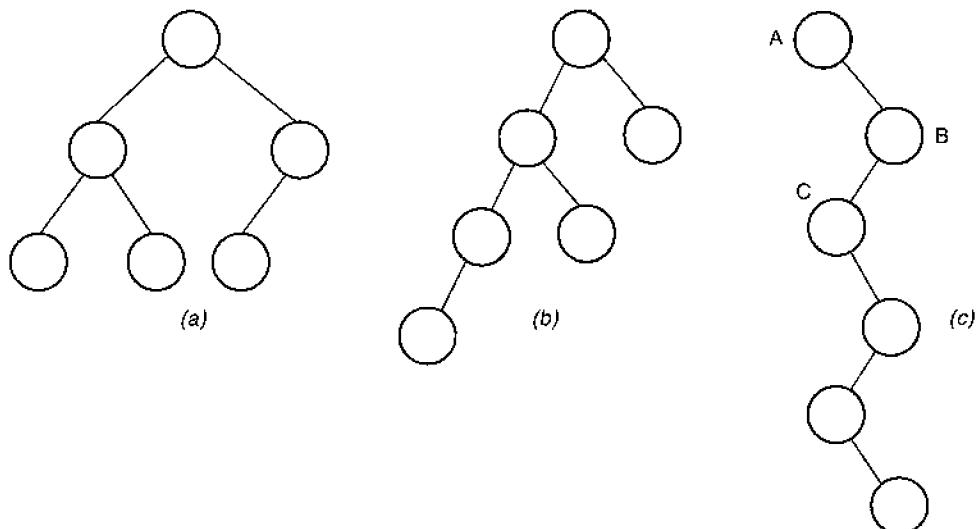


Figura 10.10. Árboles binarios con los mismos nodos pero alturas diferentes.

Definición recursiva de altura

- Si T está vacío, su altura es 0
- Si T no es un árbol binario vacío, entonces debido a que T es de la forma



La altura de T se puede definir como:

$$\text{altura } (T) = 1 + \max [\text{altura } (T_L), \text{altura } (T_R)]$$

10.2.3. Árboles binario, lleno y completo

Un **árbol binario lleno** de altura h tiene todas sus hojas a nivel h y todos los nodos que están a nivel menor que h tiene cada uno dos hijos. La Figura 10.11 representa un árbol binario lleno de altura 3.

Se puede dar una definición recursiva de árbol binario lleno:

- Si T está vacío, entonces T es un árbol binario lleno de altura 0.
- Si no está vacío y tiene altura $h > 0$, entonces T es un árbol binario lleno si los subárboles de la raíz son ambos árboles binarios llenos de altura $h - 1$.

Un **árbol binario completo** de altura h es un árbol binario que está lleno a partir del nivel $h - 1$, con el nivel h lleno de izquierda a derecha (Figura 10.12). Más formalmente, un árbol binario de altura h es completo si:

- Todos los nodos de nivel $h - 2$ y superiores tienen dos hijos cada uno.
- Cuando un nodo tiene un descendiente derecho a nivel h , todas las hojas de su subárbol izquierdo están a nivel h .

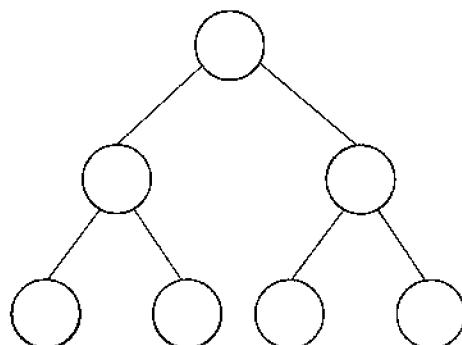


Figura 10.11. Árbol binario lleno de altura 3.

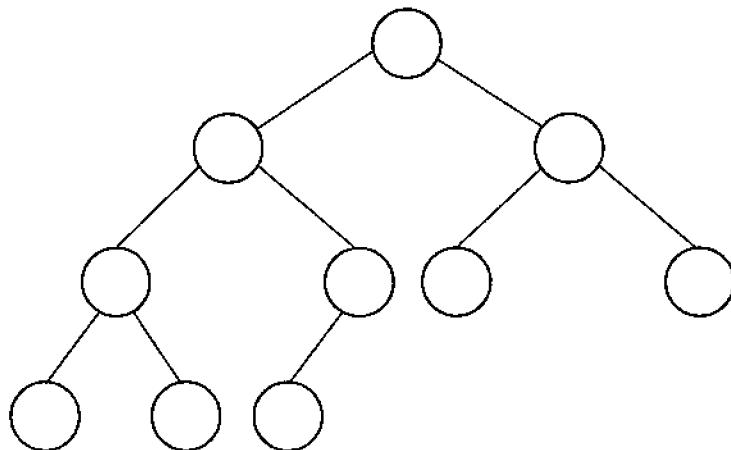


Figura 10.12. Árbol binario completo.

Si un árbol binario es lleno, es necesariamente completo.

Un árbol binario es completamente (totalmente) equilibrado si los subárboles izquierdo y derecho de cada nodo tienen la misma altura.

Un árbol binario completo es equilibrado, mientras que un árbol binario lleno es totalmente equilibrado.

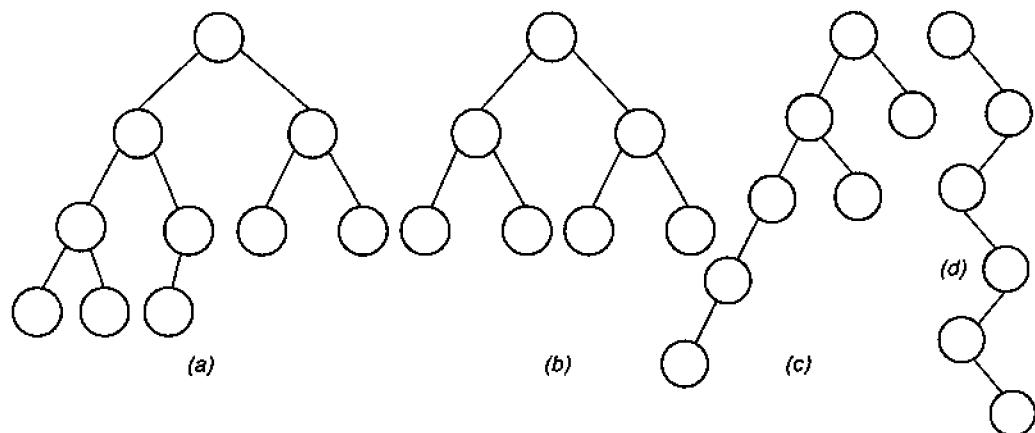


Figura 10.13. Árbol binario: (a) equilibrado, (b) completamente equilibrado; (c) y (d) árboles no equilibrados.

10.2.4. Recorrido de un árbol binario

El proceso u operación de acceder o visitar a todos los nodos (datos) de un árbol se conoce normalmente como **recorrido de un árbol**. El árbol puede ser recorrido en varios órdenes. Los tres recorridos más típicos se clasifican de acuerdo al momento en que se visita su raíz en relación con la visita a sus subárboles.

Preorden	: raíz-izquierdo-derecho
Enorden	: izquierdo-raíz-derecho
Postorden	: izquierdo-derecho-raíz

recorrido preorden

1. Visitar el raíz
2. Ir a subárbol izquierdo
3. Ir a subárbol derecho

recorrido enorden

1. Ir a subárbol izquierdo
2. Visitar el raíz
3. Ir a subárbol derecho

recorrido postorden

1. Ir a subárbol izquierdo
2. Ir a subárbol derecho
3. Visitar el raíz

En el árbol de la Figura 10.14 los posibles recorridos pueden ser:

- Recorrido *preorden* visita los nodos en el orden GDBACEFKHJIML.
- Recorrido *enorden* visita los nodos en el orden ABCDEFGIJKLHM.
- Recorrido *postorden* visita los nodos en el orden ACBFEDIJHLMKG.

10.3. ÁRBOLES DE EXPRESIÓN

Los árboles binarios se utilizan para representar expresiones en memoria; esencialmente, en compiladores de lenguaje de programación. La Figura 10.15 muestra un árbol binario de expresiones para la expresión aritmética $(a + b) * c$.

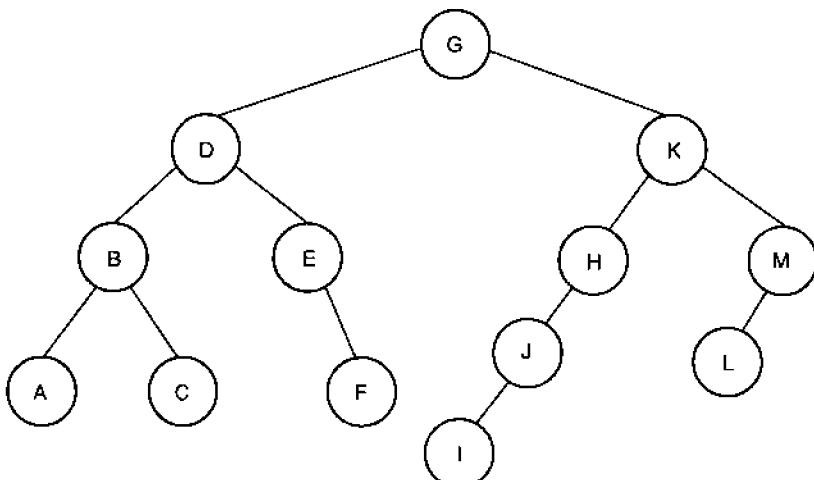


Figura 10.14. Árbol binario.

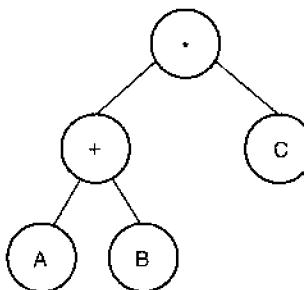


Figura 10.15. Árbol binario de expresiones que representa $(A + B) * C$.

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 10.16 se 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.

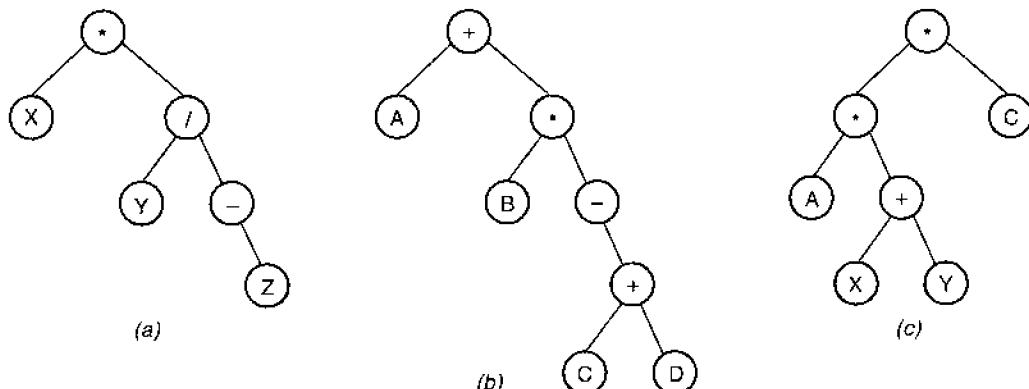
Utilizando el razonamiento anterior, se puede escribir la expresión almacenada como

$$(X + Y) * (A - B)$$

en donde se han insertado paréntesis alrededor de subexpresiones del árbol. En la Figura 10.17 aparece el árbol de la expresión $[X + (Y * Z)] * (A - B)$.

EJEMPLO 10.1

Deducir las expresiones que representan los siguientes árboles binarios



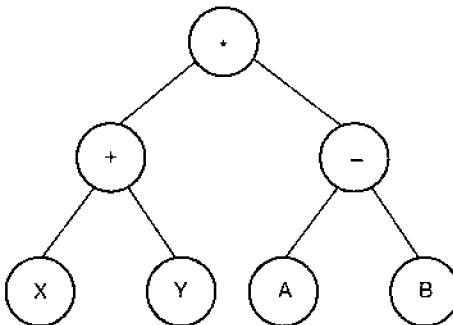


Figura 10.16. Árbol de expresión $(X+Y) * (A-B)$.

Solución

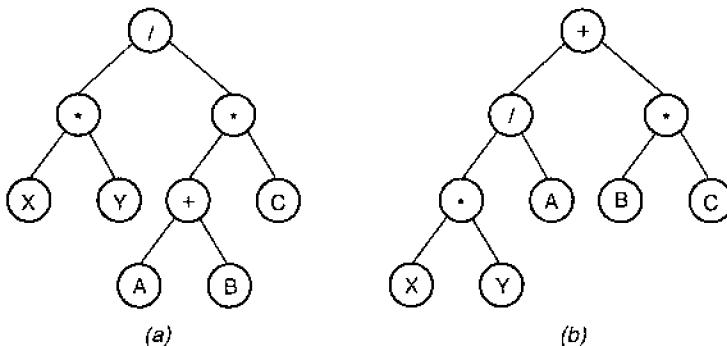
- a) $X * (Y / -Z)$
- b) $A + [(B * - (C + D))]$
- c) $[A * (X + Y)] * C$

EJEMPLO 10.2

Dibujar la representación en árbol binario de cada una de las siguientes expresiones

- a) $X * Y / [(A + B) * C]$
- b) $(X * Y / A) + (B * C)$

Solución



10.4. CONSTRUCCIÓN DE UN ÁRBOL BINARIO

Los árboles binarios se construyen de igual forma que las listas enlazadas, utilizando diferentes elementos con la misma estructura básica. Esta estructura básica es un nodo

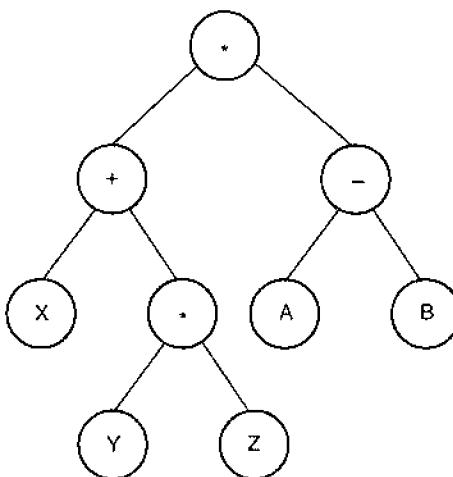


Figura 10.17. Árbol de expresión $[X + (Y * Z)] * (A - B)$.

con un espacio de almacenamiento para datos y enlaces para sus hijos izquierdo y derecho. Existe una diferencia clara con las listas y es el enlace de los nodos; esto se debe a que el árbol es bidimensional, tiene una estructura de registro más compleja y tiene muchos punteros *nil* frente a uno solo que aparece al final de una lista enlazada.

Con este formato cada nodo tiene tres elementos: *datos*, *un puntero izquierdo* y *un puntero derecho*. Los punteros son otros nodos que llevan a declaraciones muy similares a las de una lista enlazada.

```

type
  PtrArbol = ^NodoArbol

  NodoArbol = record
    Datos:NombreTipo;
    Izda, Dcha:PtrArbol; {punteros a hijos}
  end; {NodoArbol}
  
```

Añadir una hoja

Dada esta definición, un árbol se puede construir mediante llamadas sucesivas a new, cada una de las cuales asigna un nodo nuevo al árbol.

La implementación de un árbol binario comienza disponiendo al principio de una variable puntero externo T que apunta a la raíz del árbol. Si el árbol está vacío, T es nil. La Figura 10.18 ilustra esta implementación. La definición recursiva de un árbol binario conduce a que cada árbol binario no vacío conste de un subárbol izquierdo y un subárbol derecho, cada uno de los cuales es un árbol binario. Así, si T apunta a la raíz de un árbol binario, entonces T^. Izdo apunta a la raíz del subárbol izquierdo y T^. Dcho apunta a la raíz del subárbol derecho.

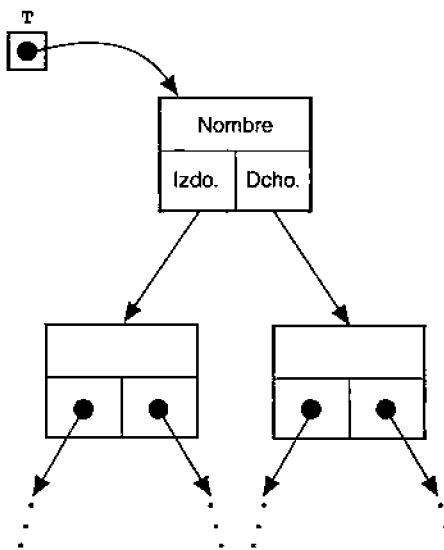


Figura 10.18. Implementación de un árbol binario.

El procedimiento para crear una hoja de un árbol y una función que comprueba si un nodo es una hoja se muestran a continuación:

```

procedure NuevaHoja(var NuevoItemPtr : PtrArbol; Num : NombreTipo);
begin
  new (NuevoItemPtr);
  NuevoItemPtr^.Datos := Num;
  NuevoItemPtr^.Izdo := nil;
  NuevoItemPtr^.Dcho := nil;
end; {NuevaHoja}

function EsHoja (unNodo : PtrArbol) : boolean;
begin
  if unNodo = nil then
    EsHoja := false;
  else
    EsHoja:=(unNodo^.Izdo = nil) and (unNodo^.Dcho = nil);
end; {EsHoja}
  
```

10.5. RECORRIDO DE UN ÁRBOL

Para visualizar o consultar los datos almacenados en un árbol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. Al contrario que las listas enlazadas, los árboles binarios no tienen realmente un primer valor, un segundo valor, tercer valor, etc. Se puede afirmar que el raíz viene el primero, pero ¿quién viene a continuación? Existen dife-

rentes métodos de recorrido de árbol, como ya se comentó en el apartado 10.2. La mayoría de las aplicaciones binarias son bastante sensibles al orden en el que se visitan los nodos, de forma que será preciso elegir cuidadosamente el tipo de recorrido.

El **recorrido** de un árbol supone visitar cada nodo sólo una vez. Las tres etapas básicas en el recorrido de un árbol binario recursivamente son:

1. Visitar el nodo (*N*)
2. Recorrer el subárbol izquierdo (*I*)
3. Recorrer el subárbol derecho (*D*)

Según sea la estrategia a seguir, los recorridos se conocen como **enorden** (*inorder*), **preorden** (*preorder*) y **postorden** (*postorder*)

preorden (nodo-izdo-dcho) (**NID**)
enorden (izdo-nodo-dcho) (**IND**)
postorden (izdo-dcho-nodo) (**IDN**)

10.5.1. Recorrido enorden

Si el árbol no está vacío, el método implica los siguientes pasos:

1. Recorrer el subárbol izquierdo (*I*).
2. Visitar el nodo raíz (*N*).
3. Recorrer el subárbol derecho (*D*).

El algoritmo correspondiente es

Enorden (A)

```
Si el arbol no esta vacio entonces
  inicio
    Recorrer el subarbol izquierdo
    Visitar el nodo raiz
    Recorrer el subarbol derecho
  fin
```

En el árbol de la Figura 10.19, 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 o 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 o raíz) y por último G. Por consiguiente, el orden del recorrido de la Figura 10.19 es D-B-E-A-F-C-G.

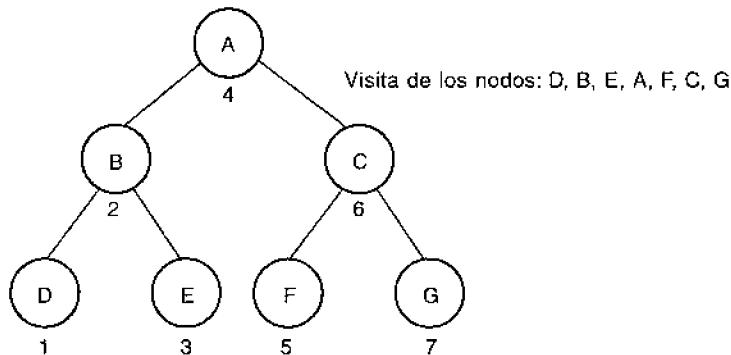


Figura 10.19. Recorrido enorden de un árbol binario.

10.5.2. Recorrido preorden

El recorrido preorden (**NID**) conlleva los siguientes pasos:

1. Visitar el raíz (**N**).
2. Recorrer el subárbol izquierdo (**I**).
3. Recorrer el subárbol derecho (**D**).

El algoritmo recursivo correspondiente es:

```

si T no es vacio entonces
    inicio
        ver los datos en el raiz de T
        preorden (subarbol izquierdo del raiz de T)
        preorden (subarbol derecho del raiz de T)
    fin

```

Si utilizamos el recorrido *preorden* del árbol de la Figura 10.20 se visita primero el raíz (nodo A). A continuación se visita el subárbol 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).

A continuación se visita 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 10.20 es A-B-D-E-C-F-G.

10.5.3. Recorrido postorden

El recorrido *postorden* (**IDN**) realiza los pasos siguientes:

1. Recorrer el subárbol izquierdo (**I**).
2. Recorrer el subárbol derecho (**D**).
3. Visitar el raíz (**N**).

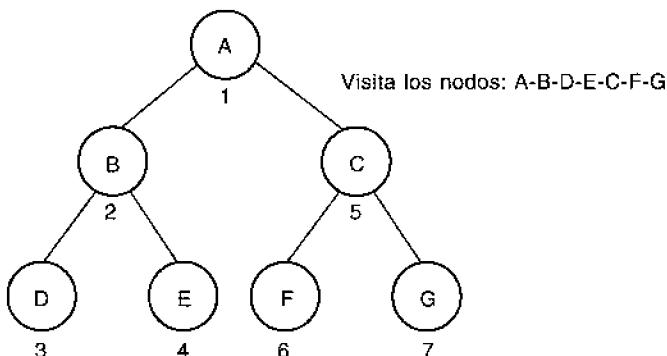


Figura 10.20. Recorrido *preorden* de un árbol binario.

El algoritmo recursivo es

```

si A no esta vacio entonces
inicio
    Postorden (subarbol izquierdo del raiz de A)
    Postorden (subarbol derecho del raiz de A)
    Visualizar los datos del raiz de A
fin
    
```

Si se utiliza el recorrido postorden del árbol de la Figura 10.21, se visita primero el subárbol izquierdo 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 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 raíz A (nodo). Así, el orden del recorrido postorden del árbol de la Figura 10.21 es D-E-B-F-G-C-A.

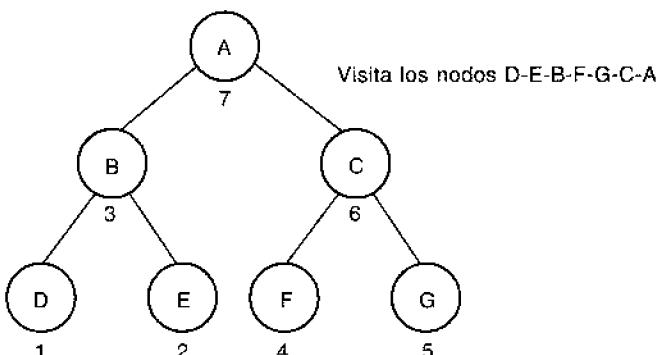
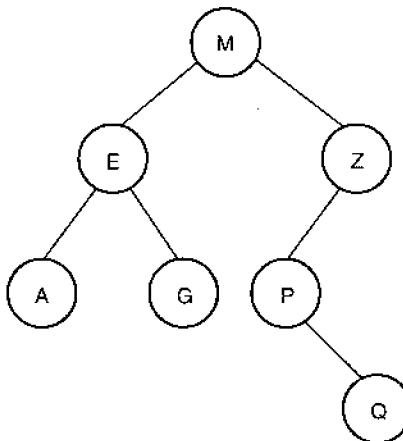


Figura 10.21. Recorrido *postorden* de un árbol binario.

EJEMPLO 10.3

Deducir los tres recorridos del árbol binario siguiente:

**Solución**

Preorden : M E A G Z P Q (NID)
Enorden : A E G M P Q Z (IND)
Postorden : A G E Q P Z M (IDN)

10.5.4. Implementación de los algoritmos de recorrido

Los métodos utilizados para recorrer un árbol binario son recursivos y por lo tanto se emplearán procedimientos recursivos para implementar los recorridos. Un árbol se representa utilizando las siguientes sentencias Pascal:

```

const LongMax = 30 {longitud máxima del nombre}
type NombreTipo = string [LongMax] ;

ptrTipo = ^nodoTipo;

nodoTipo = record
    Nombre:nombreTipo;
    HijoIzdo:ptrTipo;           {puntero a Hijo izquierdo}
    HijoDcho:ptrTipo;          {puntero a Hijo derecho}
end;
TipoArbolBin = ptrTipo;

var A : TipoArbolBin; {puntero a raíz del árbol}
  
```

Por ejemplo el procedimiento EnOrden puede ser:

```
procedure EnOrden (A : TipoArbolBin);
begin
  if A < > nil then {árbol no está vacío}
    begin
      EnOrden (A^.HijoIzdo); {operación I}
      WriteLn (A^.Nombre); {operación N}
      EnOrden (A^.HijoDcho); {operación D}
    end;
end;
```

La implementación de los recorridos *preorden* y *postorden* se realiza siguiendo el esquema algorítmico de los apartados 10.5.2 y 10.5.3, respectivamente.

10.6. APLICACIÓN DE ÁRBOLES BINARIOS: EVALUACIÓN DE EXPRESIONES

Una expresión aritmética está formada por operandos y operadores aritméticos. Así, la expresión

$$R = (A+B) * D + A * B / D$$

R está escrita de la forma habitual, el operador en medio de los operando, se conoce como notación infija. Recordemos que ya hemos realizado la aplicación de evaluar una expresión utilizando únicamente el TAD pila. Ahora va a ser utilizada tanto las pilas como el TAD árbol binario para la evaluación. La pila será utilizada para pasar la expresión de infija a postfija. La expresión en postfija será almacenada en el árbol para que la evaluación se realice utilizando el árbol binario. En primer lugar, recordamos la prioridad de operadores, de mayor a menor:

Paréntesis	:	()
Potencia	:	[^]
Multipl/división	:	* , /
Suma/Resta	:	+ , -

A igualdad de precedencia son evaluados de izquierda a derecha.

10.6.1. Notación postfija: notación polaca

La forma habitual de escribir operaciones aritméticas es situando el operador entre sus dos operandos, la llamada notación infija. Esta forma de notación obliga en muchas ocasiones a utilizar paréntesis para indicar el orden de evaluación.

$$A * B / (A + C) \qquad \qquad A * B / A + C$$

Representan distintas expresiones al no poner paréntesis. Igual ocurre con las expresiones:

$$(A - B) ^ C + D \qquad \qquad A - B ^ C + D$$

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 estudió).

$A*B/(A+C)$ (infija) $\rightarrow A*B/+AC$ $\rightarrow *AB/+AC$ $\rightarrow /*AB+AC$ (polaca)

$A*B/A+C$ (infija) $\rightarrow *AB/A+C$ $\rightarrow /*ABA+C$ $\rightarrow +/*ABAC$ (polaca)

$(A-B)^C+D$ (infija) $\rightarrow -AB^C+D$ $\rightarrow ^{-ABC}+D$ $\rightarrow +^{\wedge}-ABCD$ (polaca)

Podemos observar que no es necesario la utilización de paréntesis al escribir la expresión en notación polaca. Es la propiedad fundamental de la notación polaca, el orden en que se van a realizar las operaciones está determinado por las posiciones de los operadores y los operandos en la expresión.

Hay más formas de escribir las operaciones. Así, la notación postfija o polaca inversa coloca el operador a continuación de sus dos operandos.

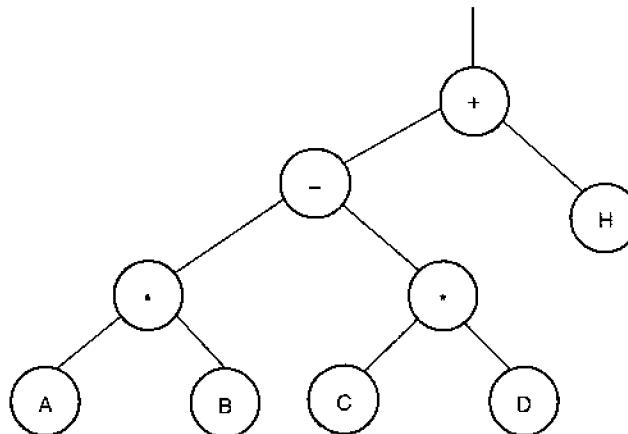
$A*B/(A+C)$ (infija) $\rightarrow A*B/AC+$ $\rightarrow AB*/AC+$ $\rightarrow AB*AC+/-$ (polaca inversa)

$A*B/A+C$ (infija) $\rightarrow AB*/A+C$ $\rightarrow AB*A/+C$ $\rightarrow AB*A/C+$ (polaca inversa)

$(A-B)^C+D$ (infija) $\rightarrow AB -^C+D$ $\rightarrow AB-C^+D$ $\rightarrow AB-C^D+/-$ (polaca inversa)

10.6.2. Árbol de expresión

Una vez que se tiene la expresión en postfija la formación del árbol binario es fácil: se crean dos nodos del árbol con respectivos operandos que se enlazan como rama izquierda y rama derecha del nodo operador. Así, la expresión $A*B-C*D+H$ se transforma en postfija: $AB*CD*-H+$ y el árbol de la expresión:



Pasos a seguir

A la hora de evaluar una expresión aritmética en notación infija se siguen estos pasos:

1. Transformar la expresión de infija a postfija.
2. Formar a partir de la expresión en postfija el árbol de expresión.
3. Evaluar la expresión utilizando el árbol.

En el algoritmo para resolver el primer paso se utiliza una pila de caracteres. Para el segundo paso se utiliza otra pila y el árbol de caracteres. Y en el tercer paso un vector con los valores numéricos de los operandos que es utilizado para evaluar la expresión en el árbol.

10.6.3. Transformación de expresión infija a postfija

Partimos de una expresión en notación infija que tiene operandos, operadores y puede tener paréntesis. Los operandos vienen representados por letras, los operadores van a ser:

\wedge (potenciación), *, /, +, - .

La transformación se realiza utilizando una pila en la que se almacenan los operadores y los paréntesis izquierdos. Esta transformación puede verse en el capítulo de Pilas, apartado 7.4, Evaluación de expresiones mediante Pilas.

10.6.4. Creación de un árbol a partir de la expresión en postfija

El árbol se forma de tal manera que en la raíz se encuentra el operador, en el subárbol izquierdo y derecho los operandos. En una pila se guardan las direcciones de los nodos que contienen a los operandos. Cuando se va a formar el nodo de un operador, se saca de la pila los dos últimos nodos metidos para formar el nodo operador enlazando la rama izquierda y la derecha con las direcciones de los nodos que se han sacado de la pila.

La unidad que encapsula el TAD pila necesario para guardar las direcciones de los nodos del árbol, también define los tipos necesarios para manejar el árbol. Esto es así porque el campo Info de los elementos de la pila es PtrA (puntero a nodos del árbol). A continuación se escribe la interfaz de la unidad pila. La sección de definición está ya escrita anteriormente.

```
unit Pilaptr;
interface
type
  PtrA = ^NodoAb;
  NodoAb = record
    C      : char;
    Izqdo,
    Drcho : PtrA
  end;
  Plaptr = ^Nodopt;
  Nodopt = record
    Info: PtrA;
    Sgte: Plaptr
  end;
```

```

function Pvacia(P: Plaptr ): boolean;
procedure Pcrear(var P: Plaptr );
procedure Pmeter(D: PtrA; var P: Plaptr);
procedure Psacar(var D: PtrA; var P: Plaptr);
function Pcima(P:Plaptr): PtrA; {devuelve el elemento cima de la pila}
procedure Pborrar(var P: Plaptr);
implementation
.....
end.

```

El siguiente procedimiento genera el árbol de expresión a partir del vector de registros que contiene la expresión en postfija.

```

procedure ArbolExpresion(var Ar:Tag; J:integer; var R:PtrA);
var {el tipo Tag está definido en la unidad ExpPost}
    I: integer;
    P: Pilaptr;
    A1, A2, A3: PtrA;
procedure Crearnodo(Iz: PtrA; Inf:char; Dr:PtrA; var A:PtrA);
begin
    new(A);
    A^.C := Inf;
    A^.Izqdo := Iz;
    A^.Drcho := Dr
end;
begin
    Pcrear(P);
    for I:= 1 to J do
        if not Ar[I].Oprdor then
            begin
                Crearnodo(nil, Ar[I].C, nil, A3);
                Pmeter(A3, P)
            end
        else begin
            Psacar(A2, P);
            Psacar(A1, P);
            Crearnodo(A1, Ar[I].C, A2, A3);
            Pmeter(A3, P)
        end;
    Psacar(A3, P); {Devuelve el árbol con la expresión}
    R:= A3
end;

```

10.6.5. Evaluación de la expresión en postfija

La primera acción que se va a realizar es dar valores numéricos a los operandos. Una vez que tenemos los valores de los operandos, la expresión es evaluada. El algoritmo de evaluación recorre el árbol con la expresión en *postorden*. De esta forma se evalúa primer operando (rama izquierda), segundo operando (rama derecha) y según el operador (raíz) se obtiene el resultado.

Es en la unidad *ExpPost* donde está definido el tipo de los Oprdos para así poder

guardar los valores numéricos de los operandos. La función real Evaluar tiene como entrada la raíz del árbol y el vector con los valores de los operandos.

```

function Evaluar (Rex:PtrA; var V: Oprdos): real;
var
  Oper1,
  Oper2 : real;
  Ch : char;
function Mayuscula(C: char): char;
begin
  if C in ['a'..'z'] then
    Mayuscula:= Chr(ord(C)+ ord('A')-ord('a'))
  else
    Mayuscula:= C
end;

function Potencia(X: real; N: integer): real;
var
  I: integer;
  T: real;
begin
  T:= 1.0;
  for I:= 1 to N do
    T:= T*X;
  if N< 0 then
    Potencia:= 1.0/T
  else
    Potencia:= T;
end;

begin
  if not Arbolvacio(Rex) then
  begin
    Ch:= Mayuscula(Rex^.C);
    if Ch in ['A' .. 'Z'] then Evaluar:= V[Ch]
    else begin
      Oper1:= Evaluar(Rex^.Izqdo, V);
      Oper2:= Evaluar(Rex^.Drcho, V);
      case Ch of
        '+': Evaluar:= Oper1+Oper2;
        '-': Evaluar:= Oper1-Oper2;
        '*': Evaluar:= Oper1*Oper2;
        '/': if Oper2<> 0 then
                  Evaluar:= Oper1/Oper2
                else begin
                  writeln('Error: división por 0');
                  exit(1)
                end;
      end;
      {El operador ^ en Pascal no existe, por lo que se realiza en una
      función. Asumimos que el exponente es entero}
      '^':Evaluar:= Potencia(Oper1,Int(Oper2));
    end;
  end
end;

```

10.6.6. Codificación del programa de evaluación

El programa que realiza todo el proceso se muestra a continuación.

En el programa incorporamos las unidades para leer y pasar la expresión de infija a postfija. Y la unidad para manejo de una pila de punteros a nodos del árbol, en esta unidad se definen los tipos de datos para manejo los nodos del árbol.

```

program Evalua_Expresion;
uses
  crt, ExpPost, Pilaptr;
var
  V: Oprdos;
  T: Tag;
  J: integer;
  R: PtrA;

procedure Leer_oprdos(E:Tag; N:integer; var V:Oprdos);
{En este procedimiento se asignan valores numéricos a los operandos}
var
  K: integer;
  Ch: char;
begin
  K := 0;
  repeat
    K:= K+1;
    if not E[K].Oprdor then {Es un operando, petición de valor numérico}
    begin
      Ch := E[K].C;
      write(Ch, '=' );
      readln(V[Ch])
    end
  until K=N
end;

procedure ArbolExpresion(var Ar: Tag; J: integer; var R: PtrA);
function Evaluar (Rex:PtrA; var V: Oprdos): real;
begin { bloque principal }
  clrscr;
  writeln('Expresión aritmética(termina #)');
  Postfija(T, J);
  ArbolExpresion(T, J, R);
  writeln('Asignación de valores numéricos a los operandos');
  writeln ;
  Leer_oprdos(T, J, V);
  writeln;
  write('Resultado de evaluación de la expresión: ');
  writeln(Evaluar(R, V):12:4)
end.

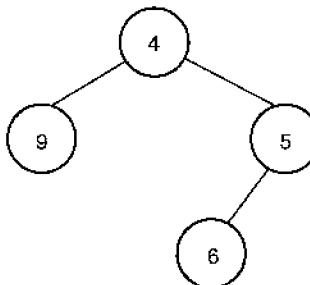
```

10.7. ÁRBOL BINARIO DE BÚSQUEDA

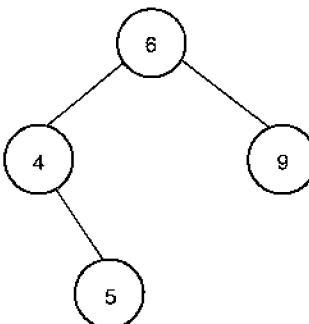
Los árboles vistos hasta ahora no tienen un orden definido; sin embargo, los árboles binarios ordenados tienen sentido. Estos árboles se denominan árboles binarios de bús-

queda, 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 los datos del nodo. Por consiguiente, este árbol *no* es un árbol binario de búsqueda,

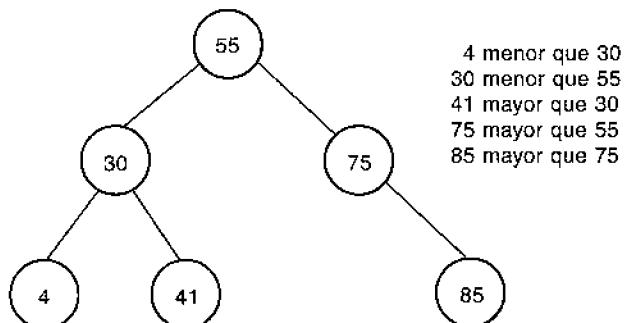


ya que 9 no es menor que 4. Sin embargo, el siguiente árbol sí es binario de búsqueda.



EJEMPLO 10.4

Árbol binario de búsqueda



10.7.1. Creación de un árbol binario

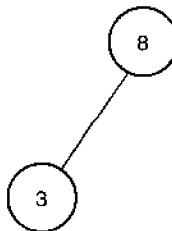
Supongamos que se desea almacenar los números

8 3 1 20 10 5 4

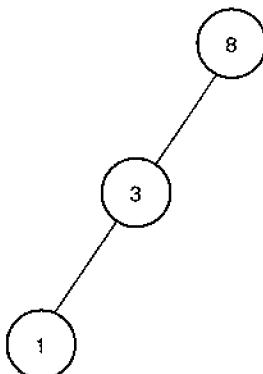
en un árbol binario de búsqueda. Siguiendo la regla, dado un nodo en el árbol todos los datos a su izquierda deben ser menores que todos los datos del nodo actual, mientras que todos los datos a la derecha deben ser mayores que dichos datos. Inicialmente el árbol está vacío y se desea insertar el 8. La única elección es almacenar el 8 en el raíz:



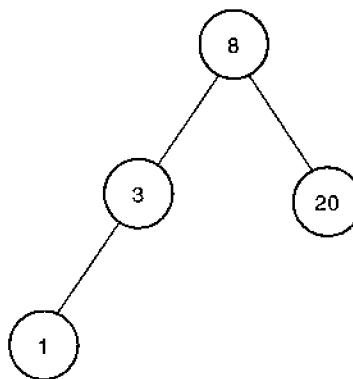
A continuación viene el 3. Ya que 3 es menor que 8, el 3 debe ir en el subárbol izquierdo



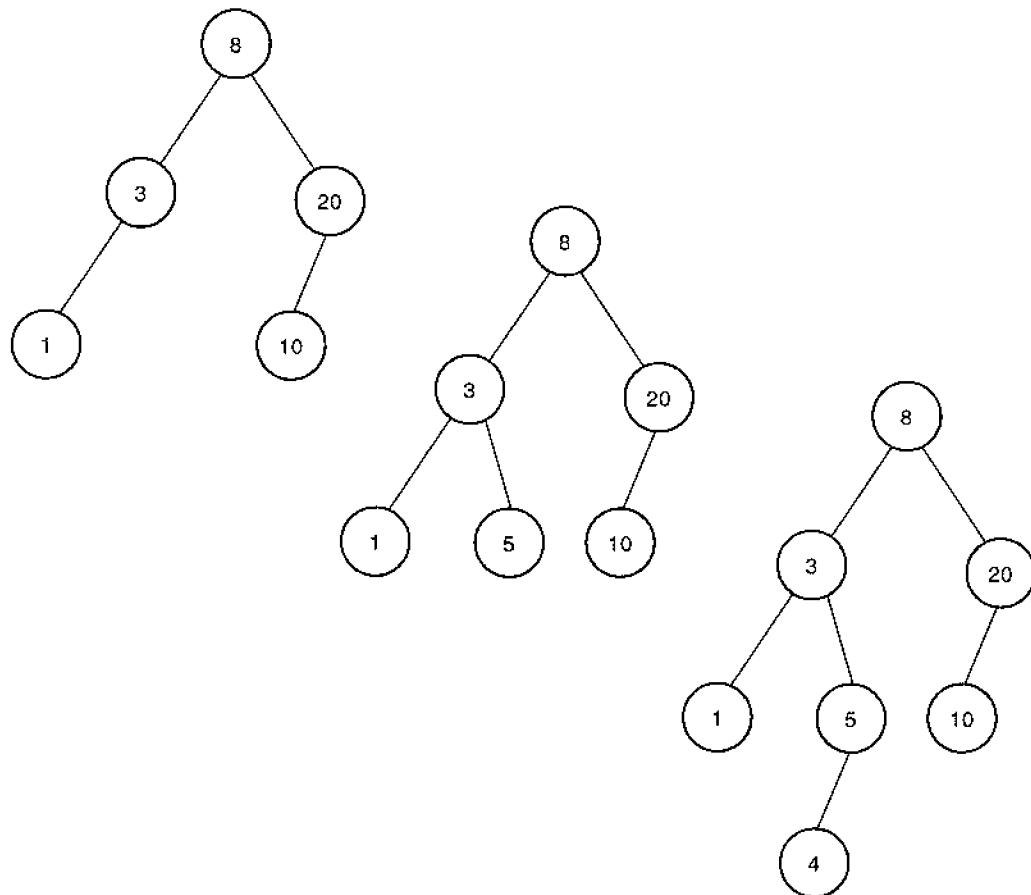
A continuación se ha de insertar 1, que es menor que 8 y que 3, por consiguiente irá a la izquierda y debajo de 3.



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.

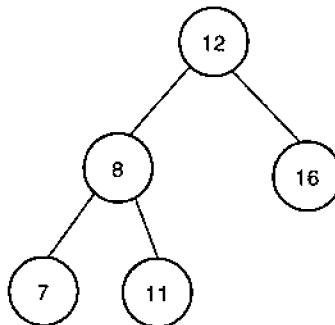


Una propiedad de los árboles binarios de búsqueda es que no son únicos para los datos dados.

EJEMPLO 10.5

Construir un árbol binario para almacenar los datos 12, 8, 7, 16 y 11.

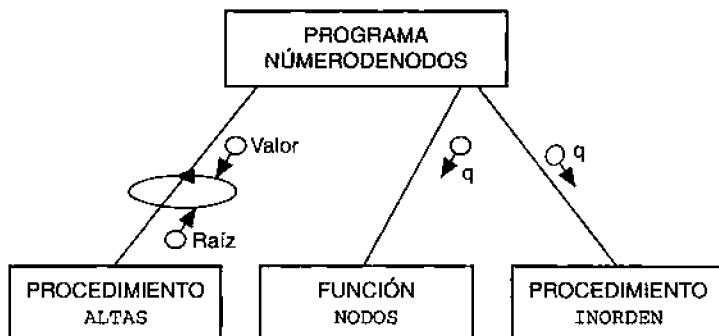
Solución



PROBLEMA 10.1²

Este programa genera un árbol binario de números enteros con un número aleatorio de nodos y en un rango de valores también aleatorio. Se utiliza un procedimiento recursivo para contar el número de nodos del árbol, otro procedimiento para mostrar los nodos ordenados ascendente y un procedimiento de altas para insertar cada nuevo número generado en su correspondiente nodo.

Diagrama de estructura



² Este problema ha sido extraído de la obra *Pascal y Turbo Pascal: Un enfoque práctico*, de Luis Joyanes, Ignacio Zahonero y Ángel Hermoso, McGraw-Hill, 1995, y con permiso de los autores.

CODIFICACIÓN

```

program NumeroDeNodos;
{Este programa genera un árbol binario de números enteros, y
posteriormente muestra y cuenta el número de nodos}
uses
  Crt;
const
  MaxNumNodos = 30;
  MaximoValor = 100;
type
  RangoNodos = 0..MaxNumNodos - 1;
  TipoClave = 1..MaximoValor;
  Puntero = ^NodoArbol;
  NodoArbol = record
    Clave : TipoClave;
    Izdo,
    Dcho : Puntero
  end;

var
  Raiz : Puntero;
  NodosGenerados,
  NodosContados,
  Indice : RangoNodos;
  Valor : TipoClave;
function Nodos (P : Puntero) : RangoNodos;
{Función recursiva para contar el número de nodos del árbol. El número de
nodos es la suma del número de nodos de cada uno de sus subárboles
izquierdo y derecho más el nodo raíz}
begin
  if P = nil then
    Nodos := 0
  else
    Nodos := 1 + Nodos (P^.Izdo) + Nodos (P^.Dcho)
end;
procedure InOrden (P : Puntero);
begin
  if P < > nil then
    begin
      InOrden (P^.Izdo);
      Write (P^.Clave : 5);
      Inorden (P^.Dcho)
    end
  end;

procedure Altas (var Root : Puntero; Valor : TipoClave);
{Inserta el nuevo valor generado, como nodo del árbol}
var
  Padre,
  Actual,
  NodoAlta : Puntero;
begin
  new (NodoAlta);
  NodoAlta^.Clave := Valor;
  NodoAlta^.Izdo := nil; {el nuevo nodo es el nodo hoja}

```

```

NodoAlta^.Dcho := nil;
Padre := nil;
Actual := Root;
while Actual <> nil do
begin
  Padre := Actual;
  if Actual^.Clave >= Valor then
    Actual := Actual^.Izdo
  else
    Actual := Actual^.Dcho
end;
  if Padre = nil then {se situará como raiz}
    Root := NodoAlta
  else
    if Padre^.Clave >= Valor then
      Padre^.Izdo := NodoAlta {se sitúa a la izquierda}
    else
      Padre^.Dcho := NodoAlta {se sitúa a la derecha}
end;

begin
  ClrScr;
  Randomize;
  NodosGenerados := Random (MaxNumNodos);
  Raiz := nil;
  Writeln ('conjunto de nodos generados al azar': 52);
  for Indice := 1 to NodosGenerados do
    begin
      Valor := Random (maximoValor) + 1;
      Write (valor:5);
      Altas (Raiz, valor) {inserción del nodo en el árbol}
    end;
  Writeln;
  Writeln;
  NodosContados := Nodos (Raiz);
  Writeln('recorrido InOrden. Nodos encontrados':65);
  InOrden (Raiz);
  Writeln;
  Writeln;
  Writeln ('Número de nodos generados: ', NodosGenerados);
  Writeln ('Número de nodos contados: ', NodosContados)
end.

```

10.8. OPERACIONES EN ÁRBOLES BINARIOS DE BÚSQUEDA

De lo expuesto hasta ahora se deduce que los árboles binarios tienen naturaleza recursiva y en consecuencia las operaciones sobre los árboles son recursivas, si bien siempre tenemos la opción de realizarlas de forma iterativa. Estas operaciones son:

- *Búsqueda* de un nodo.
- *Inserción* de un nodo.
- *Recorrido* de un árbol.
- *Supresión* de un nodo.

10.8.1. 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, o si el subárbol está vacío.
3. Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el subárbol derecho. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda con el subárbol izquierdo.

La función de búsqueda de una clave devuelve la dirección del nodo que contiene a la clave, o bien `nil` en caso de encontrarse en el árbol.

```
function busqueda (R: Ptr; X: TipoInfo): Ptr;
var Esta: boolean;
begin
  Esta:=false;
  while not Esta and (R<>nil) do
    if R^.elemento=X then
      Esta:=true
    else if R^.elemento >X then
      R:= R^.izquierdo
    else
      R:= R^.derecho;
  busqueda := R
end;
```

Se deja al lector la implementación recursiva como ejercicio práctico.

10.8.2. Inserción

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.

La implementación de la operación insertar una nueva clave en un árbol de búsqueda se realiza, en primer lugar, de forma recursiva. Siempre se inserta como nodo hoja; la posición de inserción se determina siguiendo el camino de búsqueda: izquierda para claves menores, derecha para claves mayores. Hay veces que no se permiten claves repetidas; en nuestro caso se permitirá y se inserta por la derecha.

```
procedure Insertar (var R: Ptr; X: TipoInfo);
begin
  if R = nil then
    R:= CrearNodo (X)
```

```

else if X<R^.elemento then
  Insertar (R^.izquierdo, X)
else if X>=R^.elemento then
  Insertar(R^.derecho, X)
end;

```

El mecanismo de enlace del nodo creado con el nodo antecedente se explica teniendo presente el significado de paso de parámetros por referencia (**var**).

La función *CrearNodo* reserva memoria y devuelve su dirección.

```

function CrearNodo (X: TipoInfo):Ptr;
var T: Ptr;
begin
  new(T);
  T^.elemento := X;
  T^.izquierdo := nil; T^.derecho := nil;
  CrearNodo:=T
end;

```

El procedimiento *Insertar* también puede plantearse de manera iterativa, como se muestra a continuación:

```

procedure Insertar (var R:Ptr; X:TipoInfo);
var Q, P: Ptr;
begin
  P:= R; Q:= R;
  while P<> nil do
  begin
    Q:= P; {Q tiene la dirección del nodo antecedente}
    if X< P^.elemento then P:=P^.izquierdo
    else P:= P^.derecho;
  end;
  P:= CrearNodo(X);
  if Q = nil then R:= P {Se crea el nodo raíz}
  else if X<Q^.elemento then Q^.izquierdo := P
  else if X>=Q^.elemento then Q^.derecho := P
end;

```

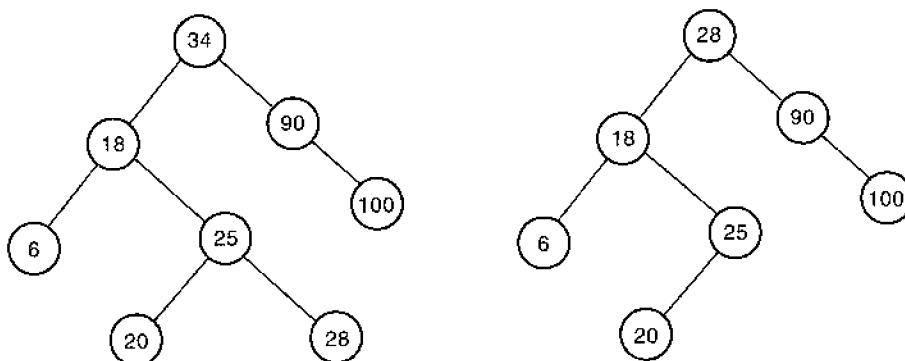
10.8.3. Eliminación

La operación de *eliminación* de un nodo es también una extensión de la operación de búsqueda, si bien es 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 de datos. 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 dos hijos, o subir a la posición que éste ocupa el nodo descendiente con la clave inmediatamente superior o inferior con objeto de mantener la estructura de árbol binario.

La eliminación de una clave y su correspondiente nodo, presenta dos casos claramente diferenciados. En primer lugar, si es un nodo hoja o tiene un único descendiente, resulta una tarea fácil, ya que lo único que hay que hacer es asignar al enlace desde el nodo padre (según el camino de búsqueda) el descendiente del nodo a eliminar (o bien, `nil`). En segundo lugar, que el nodo tenga los dos descendientes. Para mantener la estructura de árbol de búsqueda tenemos dos alternativas, reemplazar la clave a eliminar por la mayor de las claves menores, o bien reemplazar por la menor de las claves mayores. Se elige la primera alternativa, lo que supone «bajar» por la derecha en la rama izquierda del nodo a eliminar hasta llegar al nodo hoja, que será el que esté más a la derecha dentro del subárbol izquierdo de la clave a borrar.

EJEMPLO 10.6 En el árbol de la figura se quiere eliminar la clave 34. El nodo más a la derecha de su rama izquierda es 28. Se reemplaza y por último se elimina el nodo hoja 28.



En la codificación, el procedimiento anidado `Reemplazar` realiza la tarea explicada anteriormente. La codificación recursiva es la que se ajusta al planteamiento recursivo de la operación, y así se implementa:

```

procedure Eliminar (var R:Ptr; X:TipoInfo);
var Q: Ptr;
procedure Reemplazar (var N: Ptr);
begin
  if N^.derecho <> nil then
    Reemplazar (N^.derecho)  ("baja" por rama derecha)
  else begin
    Q^.elemento := N^.elemento;  {reemplaza información}
    Q := N;
    N := N^.izquierdo
  end
end;

```

```

begin
  if R=nil then writeln ('No está la clave')
  else if X<R^.elemento then
    Eliminar (R^.izquierdo, X)
  else if X>R^.elemento then
    Eliminar (R^.derecho, X)
  else begin
    Q := R;
    if Q^.derecho = nil then (No tiene dos descendientes)
      R := Q^.izquierdo
    else if Q^.izquierdo = nil then
      R := Q^.derecho
    else
      Reemplazar (Q^.izquierdo);
    dispose(Q)
  end
end;

```

10.8.4. Recorrido de un árbol

Existen dos tipos de recorrido de los nodos de un árbol: el recorrido en anchura y el recorrido en profundidad. En *el recorrido en anchura* se visitan los nodos por niveles. Para ello se utiliza una estructura auxiliar tipo cola en la que después de mostrar el contenido de un nodo, empezando por el nodo raíz, se almacenan los punteros correspondientes a sus hijos izquierdo y derecho. De esta forma si recorremos los nodos de un nivel, mientras mostramos su contenido, almacenamos en la cola los punteros a todos los nodos del nivel siguiente.

El *recorrido en profundidad* se realiza por uno de los tres métodos recursivos: *preorden*, *enorden* y *postorden*. El primer método consiste en visitar el nodo raíz, su árbol izquierdo y su árbol derecho, por este orden. El recorrido *enorden* visita el árbol izquierdo, a continuación el nodo raíz y finalmente el árbol derecho. El recorrido *postorden* consiste en visitar primero el árbol izquierdo, a continuación el derecho y finalmente el raíz.

<i>preorden</i>	Raíz	Izdo	Dcho
<i>enorden</i>	Izdo	Raíz	Dcho
<i>postorden</i>	Izdo	Dcho	Raíz

```

procedure preorden (p : ptr);
begin
  if p <> nil then
  begin
    write (p^.elemento : 6);
    preorden (p^.izquierdo);
    preorden (p^.derecho)
  end
end;

procedure en_orden (p : ptr);
begin
  if p <> nil then

```

```

begin
  en_orden (p^.izquierdo);
  Write (p^.elemento : 6);
  en_orden (p^.derecho)
end;
end;

procedure postorden (p : ptr);
begin
  if p <> nil then
    begin
      postorden (p^.izquierdo);
      postorden (p^.derecho);
      write (p^.elemento : 6);
    end;
end;

```

10.8.5. Determinación de la altura de un árbol

La *altura* de un árbol dependerá del criterio que se siga para definir dicho concepto. Así, si en el caso de un árbol que tiene nodo raíz, se considera que su altura es 1, la altura del árbol de la Figura 10.22a es 2, y la altura del árbol de la Figura 10.22b es 4. Por último, si la altura de un árbol con un nodo es 1, la altura de un árbol vacío (el puntero es `nil`) es 0.

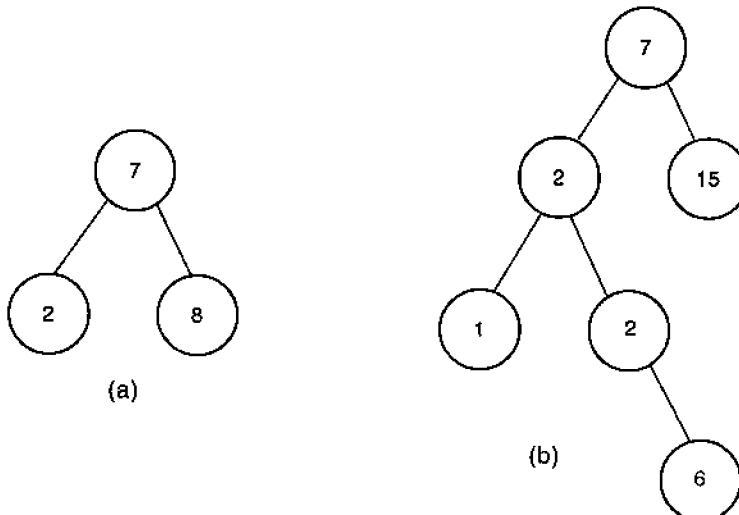


Figura 10.22. Árboles binarios de diferente altura.

Nota

La altura de un árbol es 1 más que la mayor de las alturas de sus subárboles izquierdo y derecho.

La función recursiva Altura encuentra la altura de un árbol

```

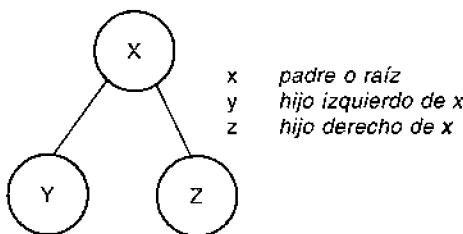
type NodoPuntero = ^Padre;
Padre = record
    info : integer;
    Izdo, Dcho : NodoPuntero
end;

function Altura (T : NodoPuntero) : Integer;
(Determina la altura de un árbol)
begin
    function Max (A, B : integer) : Integer;
    begin
        if A > B then
            Max := A
        else
            Max := B
    end; {Max}

    begin {Altura}
        if T = nil then
            Altura := 0
        else
            Altura:=1+Max(Altura(T^.Izdo), Altura(T^.Dcho))
    end; {Altura}
end;
```

RESUMEN

En este capítulo se introdujo y desarrolló la estructura de datos dinámica árbol. Esta estructura, muy potente, se puede utilizar en una gran variedad de aplicaciones de programación.

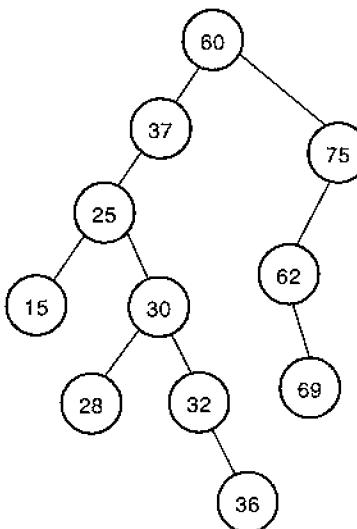


La estructura árbol más utilizada normalmente es el *árbol binario*. Un **árbol binario** es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho.

En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre, pero sí cada elemento restante tiene un parente. 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. Si el árbol A es vacío, la altura es 0. La altura del árbol siguiente es 6. El *nivel* o *profundidad* de un elemento es un concepto similar al de altura. En el árbol siguiente el nivel de 30 es 4 y el nivel de 36 es 6. 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.

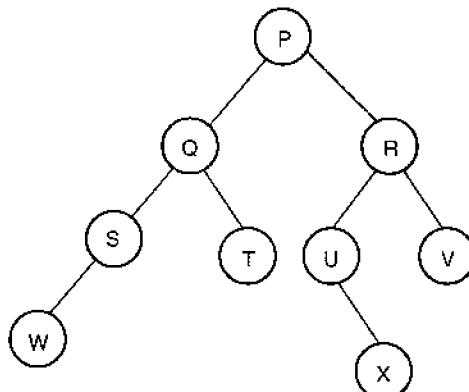


Los árboles binarios presentan dos tipos característicos: *árboles binarios de búsqueda* y *árboles binarios de expresiones*. Los árboles binarios de búsqueda se utilizan fundamentalmente para mantener una colección ordenada de datos y los árboles binarios de expresiones para almacenar expresiones.

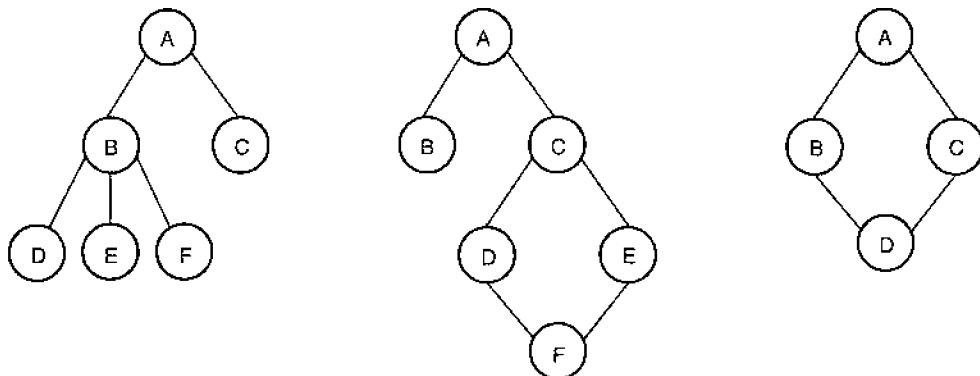
EJERCICIOS

10.1. Considérese el árbol siguiente:

- ¿Cuál es su altura?
- ¿Está el árbol equilibrado? ¿Por qué?
- Listar todos los nodos hoja.
- ¿Cuál es el predecesor inmediato (padre) del nodo U?
- Listar los hijos del nodo R.
- Listar los sucesores del nodo R.



10.2. Explicar por qué cada una de las siguientes estructuras no es un árbol binario:



10.3. Para cada una de las siguientes listas de letras:

1. M, Y, T, E, R
2. R, E, M, Y, T
3. T, Y, M, E, R
4. C, O, R, N, F, L, A, K, E, S

- 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.

10.4. Para los árboles del ejercicio 10.1, recorrer cada árbol utilizando los órdenes siguientes: NDI, DNI, DIN.

10.5. 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) $(A-B) / ((C*D) - (E/F))$

10.6. El recorrido preorden de un cierto árbol binario produce

ADFGHKLPQRWZ

y en recorrido enorden produce

GFKDLAWRQPZ

Dibujar el árbol binario.

10.7. Escribir una función recursiva que cuente las hojas de un árbol binario.

10.8. 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

- 10.9. Escribir una función que tome un árbol como entrada y devuelva el número de hijos del árbol.
- 10.10. Escribir una función booleana a la que se le pase un puntero a un árbol binario y devuelva verdadero (*true*) si el árbol es completo y falso (*false*) en caso contrario.
- 10.11. Diseñar una función recursiva que devuelva un puntero a un elemento en un árbol binario de búsqueda.
- 10.12. Diseñar una función iterativa que encuentre un elemento en un árbol binario de búsqueda.

PROBLEMAS

- 10.1. Crear un archivo de datos en el que cada linea 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 de 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

- 10.2. 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.
- 10.3. Se dispone de un árbol binario de elementos de tipo integer. Escribir funciones que calculen:
 - a) La suma de sus elementos.
 - b) La suma de sus elementos que son múltiplos de 3.
- 10.4. Escribir una función booleana IDÉNTICOS que permita decir si dos árboles binarios son iguales.
- 10.5. Escribir un programa que tenga como entrada de datos el archivo generado en el problema 10.1; de forma interactiva permita:
 1. Crear un árbol binario de búsqueda T tomando como clave el número de la Seguridad Social.

2. Añadir nuevos registros al árbol T.
 3. Eliminar un registro dada la clave,
 4. FIN (lo que supone almacenar el árbol en el archivo original).
- 10.6. Diseñar un programa interactivo que permita dar altas, bajas, listar, etc., en un árbol binario de búsqueda.
- 10.7. Construir un procedimiento recursivo para encontrar una determinada clave en un árbol binario de búsqueda.
- 10.8. Calcular el número de hojas en un árbol binario.
- 10.9. Diseñar procedimientos no recursivos que listen los nodos de un árbol en inorden, preorden y postorden.
- 10.10. Dados dos árboles binarios de búsqueda indicar mediante un programa si los árboles tienen o no elementos comunes.
- 10.11. 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.)
- 10.12. 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 $2*I + 1$. Diseñar a partir de esta representación los correspondientes procedimientos y funciones para gestionar interactivamente un árbol de números enteros. (Comente el inconveniente de esta representación de cara al máximo y mínimo número de nodos que pueden almacenarse.)
- 10.13. Una matriz de N elementos almacena cadenas de caracteres. Utilizando un árbol binario de búsqueda como estructura auxiliar ordene ascendentemente la cadena de caracteres.
- 10.14. Dado un árbol binario de búsqueda diseñe un procedimiento que liste los nodos del árbol ordenados descendentemente.
- 10.15. En tres árboles binarios de búsqueda (ORO, PLATA, COBRE) están representados los medallistas de cada una de las pruebas de una reunión atlética. Cada nodo tiene la información: nombre de la prueba, nombre del participante y nacionalidad. El árbol ORO almacena los atletas ganadores de dicha medalla, y así respectivamente con los árboles PLATA y COBRE. El criterio de ordenación de los árboles ha sido el nombre del atleta. Escribir los procedimientos/funciones necesarias para resolver este supuesto:
Dado el nombre de un atleta y su nacionalidad, del cual no se sabe si tiene medalla, encontrar un equipo de atletas de su mismo país, incluyendo al mismo que tenga una suma de puntos comprendida entre N y M . Hay que tener en cuenta que una medalla de oro son 10 puntos, plata 5 puntos y cobre 2 puntos.

REFERENCIAS BIBLIOGRÁFICAS

- Aho, Alfred V., y Ullman, Jeffrey D.: *Foundations of Computer Science*, Computer Science Press, 1992.
- Cormen, Thomas H.; Leiserson Charles, E., y Rivert Ronal, L.: *Introduction to Algorithms*, The Mit Press, McGraw-Hill, 1992.
- Carrasco, Hellman y Veroff: *Data Structures and problem solving with Turbo Pascal*, The Benjamin/Cummings, 1993.

- Collins, William J.: *Data structures. An Object-Oriented Approach*, Addison-Wesley, 1992.
- Franch Gutierrez, Xavier: *Estructura de datos. Especificación, Diseño e implementación*, Barcelona, Edicions UPC, 1994.
- Hale, Guy J., y Easton, Richard J.: *Applied Data Structures Using Pascal*, Massachusetts, Heath, 1987.
- Horowitz, Ellis, y Sartaj, Sahni: *Data Structures in Pascal*, Third edition, New York, Computer Science Press, 1990.
- Joyanes Aguilar, Luis: *Fundamentos de programación*, 2.^a edición, Madrid, McGraw-Hill, 1996.
- Joyanes, L.; Zahonero, I., y Hermoso, A.: *Pascal y Turbo Pascal. Un enfoque práctico*, Madrid, McGraw-Hill, 1995.
- Krasse, Robert L.: *Data Structures and program design*, Prentice-Hall, 1994.
- Koffman, Elliot B., y Maxim, Bruce R.: *Software Design and Data Structures in Turbo Pascal*, Addison-Wesley, 1994.
- Salmon, William J.: *Structures and abstractions*, Irwin, 1991.
- Tenembaum Aaron, M., y Angenstein Moshe: *Data structures using Pascal*, Prentice-Hall, 1986.

Árboles equilibrados

CONTENIDO

- 11.1. Eficiencia de la búsqueda en un árbol binario.
- 11.2. Árbol binario equilibrado (AVL).
- 11.3. Creación de un árbol equilibrado de N claves.
- 11.4. Inserción en árboles equilibrados. Rotaciones.
- 11.5. Eliminación de un nodo en un árbol equilibrado.
- 11.6. Programa para manejar un árbol equilibrado.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

En el Capítulo 10 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.

11.1. EFICIENCIA DE LA BÚSQUEDA EN UN ÁRBOL BINARIO

La eficiencia para una búsqueda de una clave en un árbol binario de búsqueda 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.

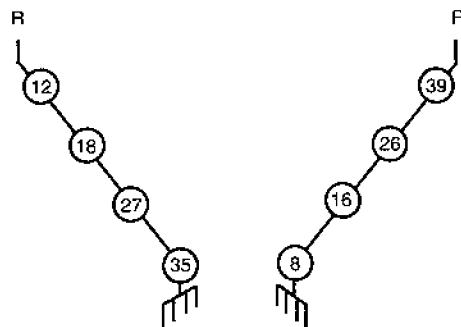


Figura 11.1. Búsqueda en un árbol binario.

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 son suficientes un máximo de $\log_2(n)$ comparaciones para obtener un elemento.

Como resumen, en los árboles de búsqueda el número promedio de comparaciones que debe de realizarse para localizar a una determinada clave es $N/2$. Esta cifra en el rendimiento de la búsqueda resulta ser poco eficiente.

Para mejorar el rendimiento en la búsqueda surgen los árboles equilibrados.

11.2. ÁRBOL BINARIO EQUILIBRADO (AVL)

La altura o profundidad de un árbol binario es el nivel máximo de sus hojas. La altura de un árbol nulo se considera cero.

Un árbol equilibrado es un árbol binario en el cual las alturas de los dos subárboles para cada nodo nunca difieren en más de una unidad. A los árboles equilibrados también se les llama árboles AVL en honor de Adelson-Velskii-Landis que fueron los primeros en proponer y desarrollar este tipo abstracto de datos.

La Figura 11.2 nos muestra un árbol AVL.

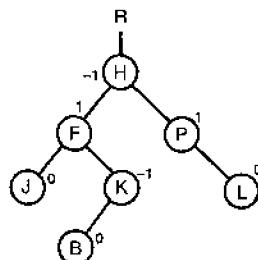


Figura 11.2. Árbol binario equilibrado (AVL).

El 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 será 1, -1 o 0. La Figura 11.2 nos muestra un árbol equilibrado con el factor de equilibrio de cada nodo.

11.3. CREACIÓN DE UN ÁRBOL EQUILIBRADO DE N CLAVES

Considerando la formación de los árboles balanceados, se parecen mucho a la secuencia de los números de Fibonacci:

$$a(n) = a(n-2) + a(n-1)$$

De igual forma, 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. A_{h-1} y A_{h-2} son árboles de Fibonacci de alturas $h-1$ y $h-2$ entonces $A_h = \langle A_{h-1}, x, A_{h-2} \rangle$ es árbol de Fibonacci de altura h .

El número de nodos A_h viene dado por la sencilla relación recurrente:

$$\begin{aligned} N_0 &= 0 \\ N_1 &= 1 \\ N_h &= N_{h-1} + 1 + N_{h-2} \end{aligned}$$

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 podemos expresar 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.

El programa *GeneraBalanceado* implementa la creación de un árbol equilibrado de n claves.

```
program GeneraBalanceado(input, output);
uses crt;
type
  Ptrae= ^Nodo;
  Nodo=record
    Clave:integer;
    Izdo,Dcho:Ptrar
  end;
var
```

```

R:Ptrae;
N:integer;

function ArbolEq(N:integer):Ptrae;
  {función para generar el árbol equilibrado de claves. Devuelve
  puntero al nodo raíz}
var
  Nuevo:Ptrae;
  Niz; Ndr:integer;
begin
  if N=0 then
    Arboleq:=nil
  else begin
    Niz:=N div 2;
    Ndr:=N-Niz-1;
    new(Nuevo);
    with Nuevo^ do
    begin
      write('Clave:'); readln(Clave);
      Izdo:=Arboleq(Niz);
      Drch:=Arboleq(Der)
    end;
    Arboleq:=Nuevo
  end
end;

procedure Dibujararbol(R:Ptrae; H:integer);
  {Dibuja los nodos del árbol. H nos permite establecer separación entre
  los nodos}
var
  I:integer;
begin
  if R<>nil then
  begin
    Dibujararbol(R^.Izdo,H+1);
    for I:=1 to H do
      write('');
    writeln(R^.Clave);
    Dibujararbol(R^.Drch,H+1)
  end
end;

begin {GeneraBalanceado}
  clrscr;
  write('¿Número de nodos del árbol?:'); readln(N);
  R:=Arboleq(N);
  clrscr;
  Dibujararbol(R,0)
end. {GeneraBalanceado}

```

11.4. INSERCIÓN EN ÁRBOLES EQUILIBRADOS: ROTACIONES

Para determinar si un árbol está equilibrado debe de manejarse información relativa al *balanceo* o factor de equilibrio de cada nodo del árbol. Por esta razón añadimos al tipo de datos que representa cada nodo un campo más: el *factor de equilibrio (Fe)*.

```

type
  Tipoinfo= ...;
  PtrAe= ^NodoAe;
  NodoAe= record
    Info: Tipoinfo;
    Fe: -1..1;
    Izqdo,
    DrCho: PtrAe
  end;

```

11.4.1. Inserción de un nuevo nodo

Se dispone ya un árbol binario equilibrado. Ahora se va a insertar un nuevo nodo, y como ocurre con las inserciones, como nodo hoja. Pueden ocurrir las siguientes circunstancias relativas al *criterio de balanceo*:

1. Las ramas izquierda (R_i) y derecha (R_d) del árbol tengan la misma altura ($h_{R_i} = h_{R_d}$), por lo que se inserte en rama izquierda o en rama derecha no va a ser causa de romper el equilibrio.
2. Las ramas izquierda y derecha del árbol tienen altura diferente ($|h_{R_i} - h_{R_d}| = 1$):
 - 2.1. Suponiendo que $h_{R_i} < h_{R_d}$ puede ocurrir:
 - 2.1.1. Si se inserta el nodo en rama izquierda entonces h_{R_i} será igual a h_{R_d} . Se ha mejorado el equilibrio.
 - 2.1.2. Si se inserta el nodo en la rama derecha entonces se rompe el criterio de equilibrio del árbol y es necesario reestructurarlo.
 - 2.2. Suponiendo que $h_{R_i} > h_{R_d}$ puede ocurrir:
 - 2.2.1. Si se inserta el nuevo nodo en la rama izquierda, entonces queda roto el equilibrio. Hay que reestructurar el árbol.
 - 2.2.2. Si se inserta el nuevo nodo en la rama derecha, entonces h_{R_d} será igual a h_{R_i} . Se ha mejorado el equilibrio.

11.4.2. Proceso a seguir: rotaciones

En todo este proceso hay que recordar que estamos trabajando con árboles binarios de búsqueda con un campo adicional que es el factor de equilibrio, por lo que el árbol resultante debe seguir siendo un árbol de búsqueda.

Para añadir un nuevo nodo al árbol, primero se baja por el árbol siguiendo el camino de búsqueda determinado por el valor de la clave, hasta llegar al lugar donde hay que insertar el nodo. Este nodo se inserta como hoja del árbol, por lo que su factor de equilibrio será 0.

Una vez hecho esto, se regresa por el camino de búsqueda marcado, calculando el factor de equilibrio de los distintos nodos que forman el camino. Este cálculo hay que hacerlo porque al añadir el nuevo nodo al menos una rama de un subárbol ha aumentado en altura. Puede ocurrir que el nuevo factor de equilibrio de un nodo viole el criterio de balanceo. Si es así, debe de reestructurarse el árbol (subárbol) de raíz dicho nodo.

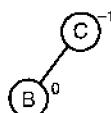
El proceso de regreso termina cuando se llega a la raíz del árbol, o cuando se realiza la reestructuración en un nodo del mismo; en cuyo caso no es necesario determinar el

factor de equilibrio de los restantes nodos, debido a que dicho factor queda como el que tenía antes de la inserción, pues el efecto de la reestructuración hace que no aumente la altura.

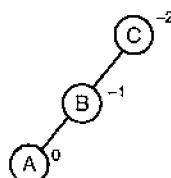
Las violaciones del factor de equilibrio de un nodo pueden darse de cuatro maneras distintas. El reequilibrio o reestructuración se realizan con un desplazamiento particular de los nodos implicados, rotando los nodos. Hay rotación derecha (D) de un nodo y rotación izquierda de un nodo (I).

Los cuatro casos de violación del balance de un nodo los mostramos en las siguientes figuras, así como la reestructuración que equilibra el árbol:

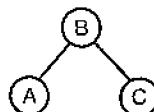
a) *El árbol original*



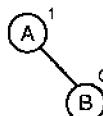
Ahora se inserta el nodo A (siguiendo el camino de búsqueda) y cambia el factor de equilibrio.



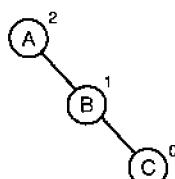
En el nodo C se ha roto el criterio de equilibrio, la reestructuración del árbol consiste en una rotación izquierda, izquierda (rotación II).



b) *El árbol original*

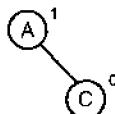


Ahora se inserta el nodo C, cambia el factor de equilibrio.

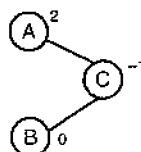


En el nodo A se ha roto el criterio de equilibrio, la reestructuración del árbol consiste en una rotación derecha, derecha (rotación DD).

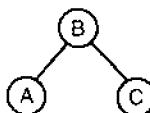
c) *El árbol original*



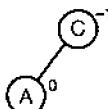
Ahora se inserta el nodo B, cambia el factor de equilibrio.



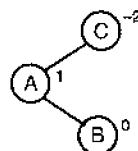
En el nodo A se ha roto el criterio de equilibrio, la reestructuración del árbol consiste en una rotación derecha, izquierda (rotación DI).



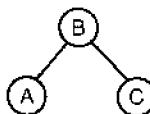
d) *El árbol original*



Ahora se inserta el nodo B (siguiendo el camino de búsqueda) y cambia el factor de equilibrio.



En el nodo C se ha roto el criterio de equilibrio, la reestructuración del árbol consiste en una rotación izquierda, derecha (rotación ID).

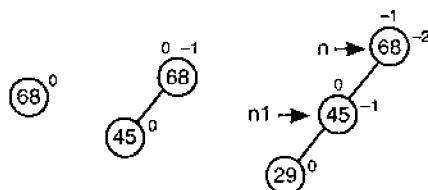


Restructurar el árbol implica mover, rotar los nodos del mismo. Las rotaciones pueden considerarse simples o compuestas. La rotación simple involucra a dos nodos, la rotación compuesta involucra a tres nodos. Las rotaciones simples se realizan, o bien por las ramas DD o II. Las rotaciones compuestas se realizan por las ramas DI o ID.

11.4.3. Formación de un árbol equilibrado

A continuación se simulan las inserciones de nodos en un árbol de búsqueda equilibrado, partiendo del árbol vacío. Por comodidad se supone que el campo clave es entero. El factor de equilibrio actual de un nodo y el nuevo al añadir un nodo se representan como superíndices de los nodos. Los punteros n , $n1$ y $n2$ referencian al nodo que viola la condición de equilibrio y a los descendientes en el camino de búsqueda.

- *Inserción de las claves 68-45-29:*

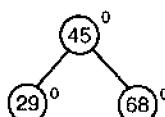


Una vez insertado el nodo con la clave 29, al regresar por el camino de búsqueda cambia los factores de equilibrio, así el del nodo 45 pasa a -1 , y en el nodo 68 se pasa a -2 . Se ha roto el criterio de equilibrio y debe de reestructurarse. Al ser los factores de equilibrio -1 y -2 debe de realizarse una rotación de los nodos II para rehacer el equilibrio. Los movimientos de los punteros para realizar esta rotación II

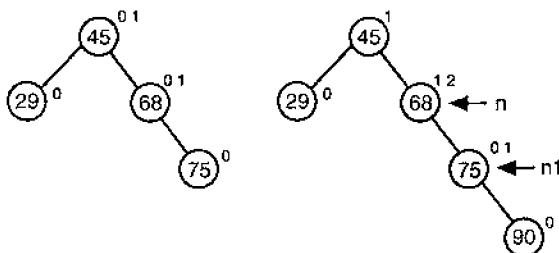
```

n^.Izqdo  ← n1^.Drcho
n1^.Drcho ← n
n          ← n1
  
```

Realizada la rotación, los factores de equilibrio serán siempre 0 en las rotaciones simples. El árbol queda de la forma siguiente:



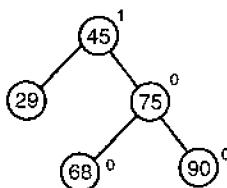
- Inserción de las claves 75 y 90



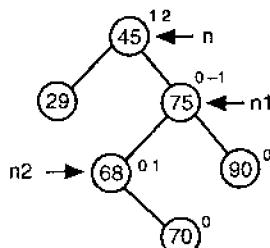
Una vez insertado el nodo con la clave 90, a la derecha del nodo 75, y regresar por el camino de búsqueda para así calcular los nuevos factores de equilibrio, se observa que dicho factor queda incrementado en 1 pues la inserción ha sido por la derecha. En el nodo con clave 68 queda roto el equilibrio. Para reestructurar se realiza una rotación DD. Los movimientos de los punteros para realizar esta rotación DD:

```
n^.Drcho  ← n1^.Izqdo
n1^.Izqdo ← n
n          ← n1
```

Una vez realizada la rotación, los factores de equilibrio de los nodos implicados será 0, como ocurre en todas las rotaciones simples, el árbol queda como sigue:



- Inserción de la clave 70



Para insertar el nodo con la clave 70 se sigue el camino: derecha de 45, izquierda de 75 y se inserta por la derecha del nodo 68. Al regresar por el camino de bús-

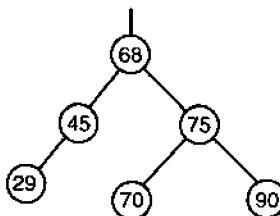
queda, los factores de equilibrio se incrementan en 1 si se fue por la rama derecha, se decrementa en 1 si se fue por la rama izquierda. En el nodo 45 el balanceo se ha roto. La rotación de los nodos para reestablecer el equilibrio es DI. Los movimientos de los punteros para realizar esta rotación DI

$n1^A.Izqdo$	\leftarrow	$n2^A.Drcho$
$n2^A.Drcho$	\leftarrow	$n1$
$n^A.Drcho$	\leftarrow	$n2^A.Izqdo$
$n2^A.Izqdo$	\leftarrow	n
n	\leftarrow	$n2$

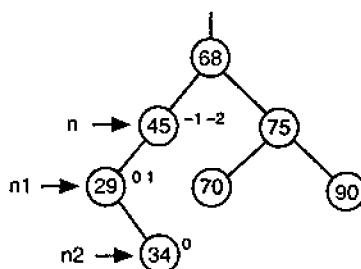
Los factores de equilibrio de los nodos implicados en la rotación depende del valor antes de la inserción del nodo referenciado por $n2$ según esta tabla:

<u>Si</u>	<u>n2^.Fe = -1</u>	<u>n2^.Fe = 0</u>	<u>n2^.Fe = 1</u>
$n^{\wedge}.Fe \leftarrow 0$	0	-1	
$n1^{\wedge}.Fe \leftarrow 1$	0	0	
$n2^{\wedge}.Fe \leftarrow 0$	0	0	

Con esta rotación el árbol quedaría



- *Inserción de la clave 34*



El camino seguido para insertar el nodo con clave 34 ha seguido el camino de izquierda de 68, izquierda de 45, derecha de 29. Al regresar por el camino de búsqueda, el factor de equilibrio del nodo 29 se incrementa en 1 por seguir el camino de la rama derecha, el del nodo 45 se decremente en 1 por seguir la rama izquierda y pasa a ser -2, se ha roto el criterio de equilibrio. La rotación de los nodos para reestablecer el equilibrio es ID.

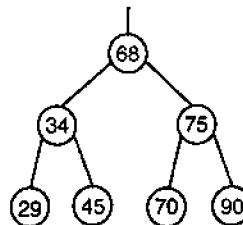
Los movimientos de los punteros para realizar esta rotación ID

```
n1^.Drcho  ← n2^.Izqdo
n2^.Izqdo  ← n1
n^.Izqdo   ← n2^.Drcho
n2^.Drcho  ← n
n          ← n2
```

Los factores de equilibrio de los nodos implicados en la rotación depende del valor antes de la inserción del nodo referenciado 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

Con esta rotación el árbol quedaría



11.4.4. Procedimiento de inserción con balanceo

En el procedimiento `Inserta_balanceado` se realizan las operaciones descritas anteriormente.

En primer lugar el recorrido baja por el camino de búsqueda hasta insertar el nuevo nodo como hoja del árbol. Una vez insertado, activa un indicador (*flag*) si ha crecido en altura para regresar por el camino de búsqueda, determinar los nuevos factores de equilibrio y si procede equilibrar.

```

function Arbolvacio(R: Ptrae): boolean;
begin
  Arbolvacio:= R= nil
end;

function Crear(X: Tipoinfo): Ptrae;
var
  Nuevo: Ptrae;

begin
  new(Nuevo);
  with Nuevo^ do
```

```

begin
  Info:=x;
  Izqdo:=nil;
  Drcho:=nil;
  Fe:=0
end;
Crear:=Nuevo
end;

procedure Rotacion_dd (var N: Ptrae; N1: Ptrae);
begin
  N^.Drcho:=N1^.Izqdo;
  N1^.Izqdo:=N;
  if N1^.Fe= 1 then
    begin
      {Si la rotación es por una inserción siempre se cumple la
       condición}
      N^.Fe:=0;
      N1^.Fe:=0
    end
  else begin
    N^.Fe:= 1;
    N1^.Fe:= -1
  end;
  N:=N1;
end;

procedure Rotacion_di (var N: Ptrae; N1: Ptrae);
{es una Rotación doble, implica el mov. de dos nodos}
var
  N2:Ptrae;
begin
  N2:=N1^.Izqdo;
  N^.Drcho:=N2^.Izqdo;
  N2^.Izqdo:=N;
  N1^.Izqdo:=N2^.Drcho;
  N2^.Drcho:=N1;
  if (N2^.Fe=1) then
    N^.Fe:=-1
  else
    N^.Fe:=0;
  if (N2^.Fe=-1) then
    N1^.Fe:=1
  else
    N1^.Fe:=0;
  N2^.Fe:=0;
  N:=N2;
end;

procedure Rotacion_ii(var N: Ptrae; N1: Ptrae);
begin
  N^.Izqdo:= N1^.Drcho;
  N1^.Drcho:=N;
  if N1^.Fe= -1 then
    begin
      {Si la rotación es por una inserción siempre se cumple la condición}

```

```

N^.Fe:=0;
N1^.Fe:=0
end
else begin
  N^.Fe:=-1;
  N1^.Fe:= 1
end;
N:=N1;
end;

procedure Rotacion_id(var N:Ptrae; N1:Ptrae);
{es una Rotación doble, implica el mov. de dos nodos}
var
  N2: Ptrae;
begin
  N2:= N1^.Drcho;
  N^.Izqdo:= N2^.Drcho;
  N2^.Drcho:= N;
  N1^.Drcho:= N2^.Izqdo;
  N2^.Izqdo:= N1;
  if (N2^.Fe=1) then
    N1^.Fe:=-1
  else
    N1^.Fe:=0;
  if (N2^.Fe=-1) then
    N^.Fe:=-1
  else
    N^.Fe:=0;
  N2^.Fe:=0;
  N:= N2;
end;

procedure insertar_balanceado (var R: Ptrae;
                               var hh: boolean; x:Tipoinfo);
{hh: activado cuando ha crecido en altura}
var
  Ni :Ptrae;
begin
  if Arbolvacio(R) then
    {se ha llegado a un nodo hoja, se crea un nuevo nodo}
    begin
      R:= crear(x);
      hh:=true {La altura del árbol cuya raíz es el ancestro ha crecido}
    end
  else
    if (x< R^.Info) then
      begin
        insertar_balanceado(R^.Izqdo, hh, x);
        if (hh) then
          {Hay que decrementar en 1 al Fe porque se
           insertó por rama izquierda}
          case R^.Fe of
            1: begin
              R^.Fe:=0;
              hh:=false
            end;

```

```

0: R^.Fe:=-1;
-1: begin {hay que reestructurar ya que
    pasaría a valer -2.Es un desequilibrio Izda}
    N1:=R^.Izqdo;
    {tipos de Rotación}
    if (N1^.Fe= -1) then
        {De nuevo desequilibrio Izda. Por lo que: Rotación ii}
        Rotacion_ii(R, N1)
    else {Rotación id}
        Rotacion_id(R,N1);
    hh:=false
end
else if (x > R^.Info) then
begin
    insertar_balanceado(R^.Drcho, hh, x);
if (hh) then
    {Al Fe hay que incrementarlo en 1 porque la
        inserción fue por rama derecha}
case R^.Fe of
    -1: begin {se reequilibra el solo}
        R^.Fe:=0;
        hh:=false
    end;
0: R^.Fe:= 1;
1: begin {Hay que reequilibrar. El
            desequilibrio es por Derecha}
    N1:=R^.Drcho;
    {tipos de Rotación}
    if (N1^.Fe = 1) then {Rotación dd}
        Rotacion_dd(R, N1)
    else {Rotación di}
        Rotacion_di(R,N1);
    hh := false;
end
end
else begin {x = R^.Info}
    writeln ('No está previsto insertar claves repetidas');
    hh:=false
end
end;

```

11.5. ELIMINACIÓN DE UN NODO EN UN ÁRBOL EQUILIBRADO

La operación de eliminación consiste en suprimir un nodo con cierta clave de un árbol equilibrado. Evidentemente, el árbol resultante debe de seguir siendo un árbol equilibrado ($|h_{Ri} - h_{Rd}| \leq 1$).

El algoritmo de eliminación sigue la misma estrategia que el algoritmo de supresión en árboles de búsqueda. Hay que añadirle las operaciones de restauración del equilibrio

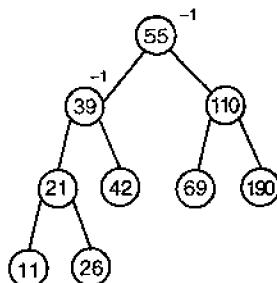
utilizadas en el algoritmo de inserción en árboles balanceados (rotación de nodos simples) (dd, ii) o dobles (di, id). Se distinguen los siguientes casos:

1. El nodo a suprimir 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, y se sustituye.

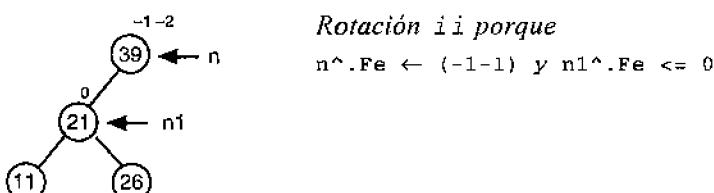
Una vez eliminado el nodo siguiendo los criterios establecidos anteriormente, se regresa por el camino de búsqueda calculando los nuevos factores de equilibrio (Fe) de los nodos visitados. Si en alguno de los nodos se viola el criterio de equilibrio, debe de restaurarse el equilibrio.

En el algoritmo de inserción, una vez que era efectuada una rotación el proceso terminaba ya que los nodos antecedentes mantenían el mismo factor de equilibrio. En la eliminación debe de continuar el proceso puesto que se puede producir más de una rotación en el retroceso realizado por el camino de búsqueda, pudiendo llegar hasta la raíz del árbol.

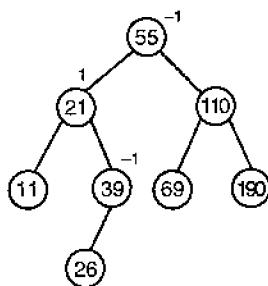
En los procedimientos se utiliza el argumento *boolean hh*, será activado cuando la altura del subárbol disminuya debido a que se haya eliminado un nodo, o bien porque al reestructurar haya quedado reducida la altura del subárbol.



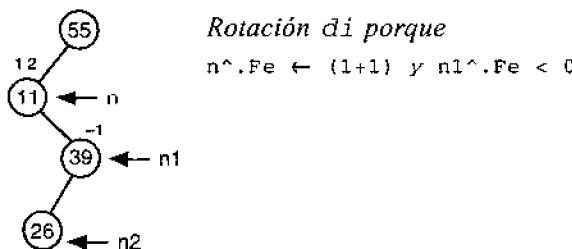
En el árbol de la figura va a ser eliminado el nodo con la clave 42: al ser un nodo hoja el borrado es simple, se suprime el nodo. Al volver por el camino de búsqueda para determinar los Fe , resulta que el Fe del nodo con clave 39 pasaría a ser -2 ya que ha decrementado la altura de la rama derecha, es violado el criterio de equilibrio. Hay que reestructurar el árbol de raíz 39.



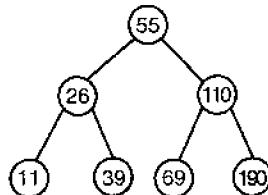
El árbol resultante es:



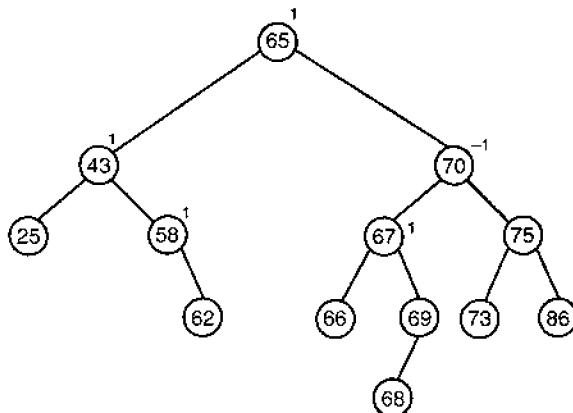
Ahora se elimina el nodo con la clave 21. Al tener dos ramas, toma el nodo más a la derecha de la rama izquierda que es el de clave 11. Al volver por el camino de búsqueda para calcular los Fe, el factor de equilibrio del nodo 11 pasaría a ser 2 y por tanto hay reestructurar el árbol de raíz 11.



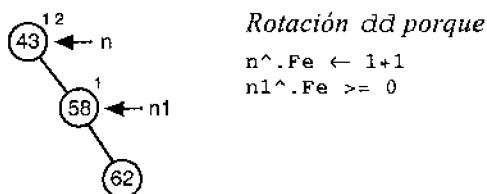
El árbol resultante es:



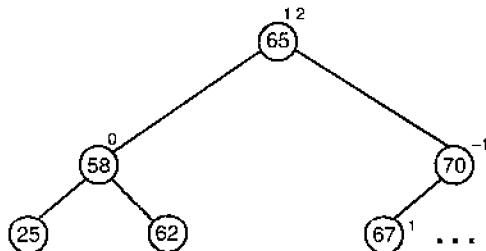
En estos dos ejemplos se observa que después de realizar la eliminación de un nodo, y cuando se regresa por el camino de búsqueda, el factor de equilibrio del nodo visitado disminuye en 1 si la eliminación se hizo por su rama derecha y se incrementa en 1 si la eliminación se hizo por su rama izquierda. Consideraremos ahora este árbol equilibrado:



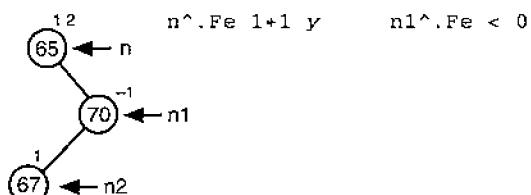
Se elimina el nodo de clave 25. Como es un nodo hoja se suprime. La supresión se hace por la rama izquierda, por lo que la altura de la rama derecha correspondiente aumenta en 1, y lo mismo ocurre con el factor de equilibrio. Los factores de equilibrio quedan:



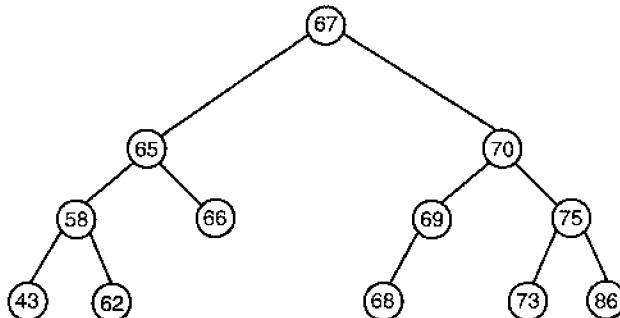
El árbol resultante es:



Al seguir regresando por el camino de búsqueda, el nodo raíz debe de incrementar su Fe con lo que pasaría a +2, por consiguiente hay que restaurar el árbol, la rotación es derecha-izquierda ya que



El nuevo árbol queda así es:



En el algoritmo de supresión se introducen dos procedimientos simétricos de equilibrado: Equilibrar1 se invoca cuando la altura de la rama izquierda ha disminuido y Equilibrar2 se invocará cuando la altura de la rama derecha haya disminuido.

En el procedimiento 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 produce es del tipo derecha-derecha, o derecha-izquierda.

```

procedure Equilibrar1(var N:Ptrae; var hh: boolean);
{hh: activado cuando ha disminuido en altura la rama izquierda
del nodo N}
var
  N1 :Ptrae;
begin
  case N^.Fe of
  -1: N^.Fe:= 0;
  0: begin
      N^.Fe:= 1;
      hh:= false
    end;
  1: begin {Hay que restaurar el equilibrio}
      N1:= N^.Drcho;
      {Es determinado el tipo de rotación}
      if N1^.Fe >= 0 then
        begin
          if N1^.Fe= 0 then
            hh:= false; {No disminuye de nuevo la altura}
            Rotacion_dd(N, N1)
          end
        else
          Rotacion_di(N, N1)
        end
      end
    end;
end;
  
```

En el procedimiento Equilibrar2 al disminuir la altura de la rama derecha, el factor de equilibrio queda decrementado en 1. De producirse una violación del criterio de equilibrio, la rotación será del tipo izquierda-izquierda, o izquierda-derecha.

```

procedure Equilibrar2(var N: Ptrae; var hh: boolean);
{hh: activado cuando ha disminuido en altura la rama derecha del nodo N}
var
  N1 :Ptrae;
begin
  case N^.Fe of
    1: N^.Fe:= 0;
    0: begin
      N^.Fe:= -1;
      hh:= false
    end;
    -1: begin {Hay que restaurar el equilibrio}
      N1:= N^.Izqdo;
      {Es determinado el tipo de rotación}
      if N1^.Fe <= 0 then
        begin
          if N1^.Fe= 0 then
            hh:= false;
          Rotacion_ii(N, N1)
        end
      else
        Rotacion_id(N, N1)
    end
  end
end;

```

A continuación son escritos los procedimientos de borrar_balanceado y el procedimiento anidado bor. El algoritmo que sigue es el mismo que el de borrado en los árboles de búsqueda sin criterio de equilibrio. La principal diferencia está en que en el momento que una rama disminuye en altura es llamado el procedimiento respectivo de equilibrar.

```

procedure borrar_balanceado(var R:Ptrae;var hh:boolean;X: Tipoinfo);

var
  Q:Ptrae;
procedure bor(var d: Ptrae; var hh: boolean);
begin
  if d^.Drcho<>nil then
    begin
      bor(d^.Drcho, hh);
      if hh then {Ha disminuido rama derecha}
        Equilibrar2 (d, hh)
    end
  else begin
    q^.info:=d^.info;
    q:=d;
    d:=d^.Izqdo;
    hh:= true
  end
end;
begin
  if not ArbolVacio(R) then
    if x< R^.info then

```

```

begin
  borrar_balanceado(R^.Izqdo, hh, x);
  if hh then
    Equilibrar1(R, hh)
end
else if x>R^.info then
begin
  borrar_balanceado(R^.Drcho, hh, x);
  if hh then
    Equilibrar2(R, hh)
end
else begin {Ha sido encontrado el nodo}
  q:=R;
  if q^.Drcho= nil then
  begin
    R:= q^.Izqdo;
    hh:= true {Disminuye la altura}
  end
  else if q^.Izqdo=nil then
  begin
    R:=q^.Drcho;
    hh:= true
  end
  else begin
    bor(q^.Izqdo, hh);
    if hh then
      Equilibrar1(R, hh)
  end;
  dispose(q);
end
end;

```

11.6. MANIPULACIÓN DE UN ÁRBOL EQUILIBRADO

Agrupamos todas las operaciones relativas a los árboles equilibrados en la unidad U_avl. De esta forma realizamos el TAD árbol binario equilibrado. Sólo presentamos la sección de interface, «interfaz» (en definitiva la parte pública).

El campo de información se supone por comodidad que es entero. Incorporamos la realización de la operación buscdir y mostrar. La operación buscdir devuelve la dirección del nodo que contiene el campo de información X. El procedimiento mostrar visualiza el árbol, girado 90 grados a la izquierda respecto a la forma habitual que se tiene de verlo en papel.

```

unit U_Avl;
interface
type
  Tipoinfo= integer;
  Ptrae = ^Nodoae;
  Nodoae = record
    Info : Tipoinfo;
    Fe   : -1..1;

```

```

Izqdo,
Drcho : Ptrae
end;

procedure insertar_balanceado (var R: Ptrae; var hh: boolean;
                               X: Tipoinfo);
procedure borrar_balanceado (var R:Ptrae; var hh:boolean;
                               X: Tipoinfo);
function Crear (X: Tipoinfo): Ptrae;
procedure Rotacion_DD(var N:Ptrae; Nl:Ptrae);
procedure Rotacion_DI(var N:Ptrae; Nl:Ptrae);
procedure Rotacion_II(var N:Ptrae; Nl:Ptrae);
procedure Rotacion_ID(var N:Ptrae; Nl:Ptrae);
procedure Equilibrar1(var N: Ptrae; var hh: boolean);
procedure Equilibrar2(var N: Ptrae; var hh: boolean);
function Arbolvacio(R: Ptrae): boolean;
function Buscdir(R: Ptrae; P: Tipoinfo): Ptrae;
procedure mostrar(R: Ptrae; h: integer);
implementation

.....


function Buscdir(R: Ptrae; P: Tipoinfo): Ptrae;
var
  Sw : boolean;
begin
  Sw:= false;
  while not Sw and (R <> nil) do
    if R^.Info= P then
      Sw:= true
    else if R^.Info> P then
      R:= R^.Izqdo
    else
      R:= R^.Drcho;
  Buscdir:= R
end;

procedure mostrar(R: Ptrae; h: integer);
var
  J: integer;
begin
  if Not Arbolvacio(R) then
    with R^ do
      begin
        mostrar(Drcho, h+1);
        for J:= 1 to h do
          write('   ');
        writeln(Info);
        mostrar(Izqdo, h+1);
      end
    end;
begin
end.

```

El programa que a continuación exponemos presenta un menú de opciones para así poder manejar las diversas operaciones del TAD árbol equilibrado.

```

program Arbol_equilibrado(input, output);
uses
  crt, U_avl;
var
  R: Ptrae;
  Op: integer;
procedure menu;
begin
  clrscr;
  gotoxy(15,1);
  writeln ('Operaciones con Árboles Equilibrados');
  gotoxy(15,2);
  writeln('-----');
  gotoxy(16,4);
  writeln('1. Insertar claves en el árbol.');
  gotoxy(16,7);
  writeln('2. Eliminar nodos del árbol.');
  gotoxy(16,9);
  writeln('6. Terminar ejecución.');
  gotoxy(17,12);
  write('¿ Elija una opción ?: ')
end;

procedure Annadir(var R: Ptrae);
const
  Clv= 100;
  Mx= 11;
var
  X: Tipoinfo;
  Hh: boolean;
  N,I: integer;
begin
  clrscr;
  if Arbolvacio(R) then
  begin
    N:= Mx; {Número de claves para árbol inicial}
    write('Claves que son insertadas: ');
    for I:= 1 to N do
    begin
      X:= random(Clv);
      write(X, '-');
      insertar_balanceado(R, Hh, X)
    end;
    gotoxy(WhereX-1, WhereY);
    writeln(' ')
  end
  else begin
    mostrar(R, 0);
    write('Nueva clave: ');
    readln(X);
    insertar_balanceado(R, Hh, X)
  end
end;

```

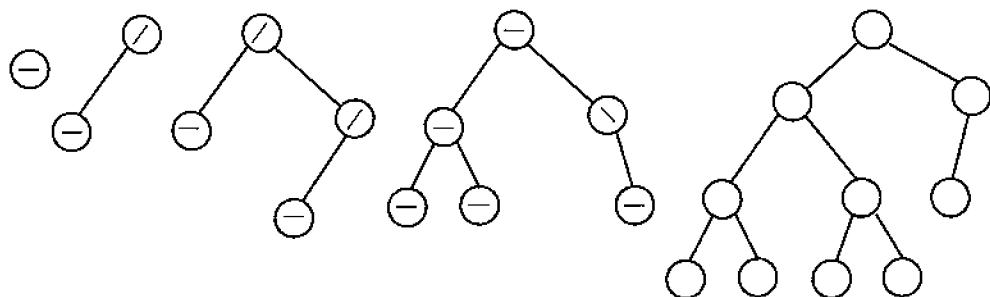
```

end;
procedure Eliminar(var R: Ptrae);
var
  X: Tipoinfo;
  Hh: boolean;
begin
  mostrar(R,0);
  write('Clave a eliminar: ');
  readln(X);
  borrar_balanceado(R, Hh, X)
end;
function Opcion: integer;
var
  Y: integer;
  Ax,Ay: integer;
begin
  repeat
    Ax:= WhereX;
    Ay:= WhereY;
    readln(Y);
    gotoxy(Ax,Ay);
  until Y in [1..2,6];
  Opcion:= Y
end;
begin
  randomize;
  R:= nil;
  repeat
    menu;
    Op:= Opcion;
    clrscr;
    Case Op of
      1: Annadir(R);
      2: Eliminar(R);
    end;
    mostrar(R, 0);
    repeat until readkey in [#0..#255]
  until Op= 6
end.

```

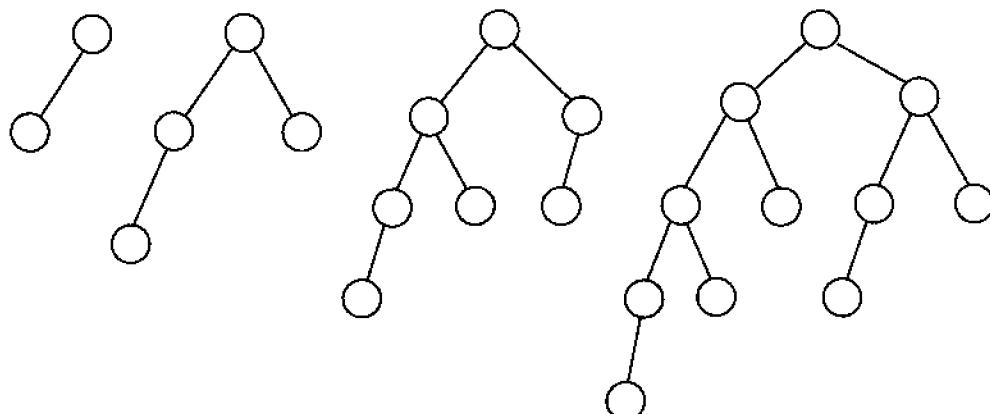
RESUMEN

Un árbol AVL o equilibrado es un árbol binario de búsqueda en el que las alturas de los subárboles izquierdo y derecho del raíz difieren a lo sumo en uno y en los que los subárboles izquierdo y derecho son, a su vez, árboles equilibrados.



Árboles equilibrados

- Inserción de un nodo. Se puede insertar un nuevo nodo en un árbol equilibrado utilizando en primer lugar el algoritmo de inserción de árbol binario, comparando la clave del nuevo nodo con el raíz e insertando el nuevo nodo en el subárbol izquierdo o derecho apropiado.
- Supresión de un nodo. La supresión de un nodo x de un árbol equilibrado requiere las mismas ideas básicas, incluyendo rotaciones simples y dobles, que se utilizan por inserción.
- Árbol de Fibonacci. Es un árbol equilibrado que se define así:
 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. A_{h-1} y A_{h-2} son árboles de Fibonacci de alturas $h-1$ y $h-2$, entonces, $A_h = \langle A_{h-1}, x, A_{h-2} \rangle$ es un árbol de Fibonacci de altura h .



EJERCICIOS

- 11.1. Dibuje el árbol AVL T que resulta de insertar las claves 14, 6, 24, 35, 59, 17, 21, 32, 4, 7, 15, 22 partiendo de un árbol vacío.
- 11.2. 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 desequilibrio cuya restauración sea del tipo «rotación simple», «rotación compuesta».
- 11.3. En el árbol T resultante del ejercicio 11.1 realiza eliminaciones consecutivas de claves y reequilibrios, si fueran necesarios, hasta encontrar una rotación compuesta.
- 11.4. Tenemos que el árbol equilibrado B está vacío. Encontrar una secuencia de n claves que al ser insertadas siguiendo el procedimiento descrito en el capítulo, se realicen al menos una vez las distintas rotaciones: II, DD, ID, DI. ¿Cuál es el valor mínimo de n para una secuencia del tipo descrito?
- 11.5. Encontrar un árbol equilibrado con n claves de tal forma que después al eliminarlas se realicen las cuatro rotaciones para restablecer el equilibrio.
- 11.6. a) ¿Cuál es el número mínimo de nodos en un árbol equilibrado de altura 10? b) Generalizando, ¿cuál es el número mínimo de nodos de un árbol equilibrado de altura n ?
- 11.7. El procedimiento de insertar una clave en un árbol equilibrado está escrito de manera recursiva. Escribir el procedimiento de insertar en un árbol equilibrado de forma iterativa.
- 11.8. Escribir el procedimiento de borrar una clave en un árbol equilibrado de forma iterativa.

PROBLEMAS

- 11.1. Se quiere leer un archivo de texto y almacenar en memoria todas las palabras de dicho texto y su frecuencia. La estructura de datos va a ser tal que permita realizar una búsqueda en un tiempo $O(\log n)$, sin que dependa de la entrada de datos, por lo que se requiere utilizar un árbol avl. El archivo de texto se llama carta.dat, se pide almacenar dicho texto en memoria, utilizando la estructura de árbol indicada anteriormente. Posteriormente se ha de realizar un procedimiento, que dada una palabra nos indique el número de veces que aparece en el texto.
- 11.2. En un archivo de texto se encuentran los nombres completos de los alumnos del taller de teatro de la universidad. Escribir un programa que lea el archivo y forme un árbol binario de búsqueda con respecto a la clave apellido. Una vez formado el árbol binario de búsqueda, formar con sus nodos un árbol de fibonacci T_n .
- 11.3. Los n pueblos del distrito judicial Lupianense están dispuestos en un archivo. Cada registro del archivo tiene el nombre del pueblo y el número de sus habitantes. Escribir un programa para disponer en memoria los pueblos de la comarca de la siguiente forma: en un vector almacenamos el número de pueblos, cada elemento del vector tiene el nombre del pueblo y la raíz de un árbol AVL con los nombres de los habitantes de cada pueblo. Al no estar los habitantes almacenados en el archivo se tienen que pedir por entrada directa al usuario.

Nota: Como es conocido el número de nodos de cada árbol AVL, aplicar la construcción de árbol de fibonacci.

- 11.4. Al problema 11.3 se desea añadir la posibilidad de manejar la estructura. Así, añadir la opción de cambiar el nombre de una persona de un determinado pueblo; habrá que tener en cuenta que esto puede suponer que ya no sea un árbol de búsqueda, por ello será nece-

sario eliminar el nodo y después crear otro nuevo con la modificación introducida. También queremos que tenga la opción de que dado el pueblo A y el pueblo B, se elimine el pueblo A añadiendo sus habitantes al pueblo B; en cuyo caso hay que liberar la memoria ocupada por los habitantes de A.

- 11.5. Un archivo F contiene los nombres que formaban un árbol binario de búsqueda equilibrado R, y que fueron grabados en F en el transcurso de un recorrido en anchura de R. Escribir un programa que realice las siguientes tareas: a) Leer el archivo F para reconstruir el árbol R. b) Buscar un nombre determinado en R, en caso de encuentro mostrar la secuencia de nombres contenidos entre la raíz de R y el nodo donde figura el nombre buscado.
- 11.6. Una empresa de servicios tiene tres departamentos, comercial (1), explotación (2) y comunicaciones (3). Cada empleado está adscrito a uno de ellos. Se ha realizado un redistribución del personal entre ambos departamentos. El archivo EMPRESA contiene en cada registro los campos Número-Idt, Origen, Destino. El campo Origen toma los valores 1, 2, 3 dependiendo del departamento inicial al que pertenece el empleado. El campo Destino toma los mismos valores, dependiendo del nuevo departamento asignado al empleado. El archivo no está ordenado. Escribir un programa que almacene los registros del archivo EMPRESA en tres árboles AVL, uno por cada departamento origen, y realice el intercambio de registros en los árboles según el campo destino.

Árboles B

CONTENIDO

- 12.1. Definición de un árbol B.
- 12.2. Representación de un árbol B de orden m .
- 12.3. Proceso de creación en un árbol B.
- 12.4. Búsqueda de una clave en un árbol B.
- 12.5. Algoritmo de inserción en un árbol B de orden m .
- 12.6. Recorrido en un árbol B de orden m .
- 12.7. Eliminación de una clave en un árbol B de orden m .
- 12.8. TAD árbol B de orden m .
- 12.9. Realización de un árbol B en memoria externa.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Los árboles B se utilizan para la creación de bases de datos. Así, una forma de implementar los índices de una base de datos relacional es a través de un árbol B.

Otra aplicación dada a los árboles B es en la gestión del sistema de archivos del sistema operativo OS/2, con el fin de aumentar la eficacia en la búsqueda de archivos por los subdirectorios.

También se conocen aplicaciones de los árboles B en sistemas de compresión de datos. Bastantes algoritmos de compresión utilizan árboles B para la búsqueda por clave de datos comprimidos.

12.1. DEFINICIÓN DE UN ÁRBOL B

Es costumbre denominar a los nodos de un *árbol B*, página. Cada nodo, cada página, es una unidad a la que se accede en bloque.

Las estructuras de datos que representan un árbol B de orden m tienen las siguientes características:

- Todas las páginas hoja están en el mismo nivel.
- Todos las páginas internas, menos la raíz, tienen a lo sumo m ramas (no vacías) y como mínimo $m/2$ (redondeando al máximo entero) ramas.
- El número de claves en cada página interna es uno menos que el número de sus ramas, y estas claves dividen las de las ramas a manera de un árbol de búsqueda.
- La raíz tiene como máximo m ramas, puede llegar a tener hasta 2 y ninguna si el árbol consta de la raíz solamente.

Los árboles B más utilizados cuando se manejan datos en memoria principal son los de orden 5; un orden mayor aumenta considerablemente la complejidad de los algoritmos y un orden menor disminuye la eficacia de la localización de claves.

En la Figura 12.1 se muestra un árbol B de orden 5, donde las claves son las letras del alfabeto.

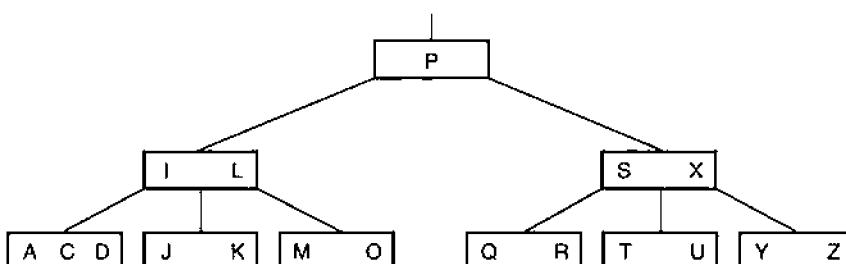


Figura 12.1. Árbol de orden 5.

En el árbol B de la figura hay 3 niveles. Todas las páginas contienen 2, 3 o 4 elementos. La raíz es la excepción, tiene un solo elemento. Todas las páginas que son hojas están en el nivel más bajo del árbol, en la figura en el nivel 3.

Las claves mantienen una ordenación de izquierda a derecha dentro de cada página. Estas claves dividen a los nodos descendientes a la manera de un árbol de búsqueda, claves de nodo izquierdo menores, claves de nodo derecho mayores. Esta organización supone una extensión natural de los árboles binarios de búsqueda. El método a seguir para localizar una clave en el árbol B va a seguir un camino de búsqueda.

12.2. REPRESENTACIÓN DE UN ÁRBOL B DE ORDEN m

Para representar una página o nodo del árbol B tenemos que pensar en almacenar las claves y almacenar las direcciones de las ramas que cuelgan de los nodos. Para ello se utilizan dos vectores, y un campo adicional que en todo momento contenga el número de claves de la página.

Las declaraciones siguientes se refieren a un árbol B de orden 5.

```

const
  Max= 4; {Número máximo de claves de una página. max = m-1, siendo m el
            orden del árbol}
  Min= 2; {Número mínimo de claves en un nodo distinto de la raíz;
            min = 1/2m-1}
  
```

```

type
  Tipoclave= ... ;
  Ptrb= ^Pagina;
  Posicion= 0..Max;
  Pagina= record
    Cuenta: 0 .. max;
    Claves: array [1 .. max] of Tipoclave;
    Ramas: array [Posicion] of Ptrb;
  end;

```

La razón de indexar Ramas desde 0 está en que cada página tiene un número de ramas igual al de claves más 1, y que es la forma más fácil de acceder a la rama más a la izquierda.

12.3. PROCESO DE CREACIÓN EN UN ÁRBOL B

La condición de que todas las hojas de un árbol B se encuentren en el mismo nivel impone el comportamiento característico de los árboles B: crecen «hacia arriba», crecen en la raíz.

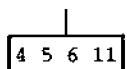
El método que se sigue para añadir una nueva clave en un árbol B es el siguiente:

- Primero se busca si la clave a insertar está ya en el árbol, para lo cual se sigue el camino de búsqueda.
- En el caso de que la clave no esté en el árbol, la búsqueda termina en un nodo hoja. Entonces la nueva clave se inserta en el nodo hoja. Más bien, se intenta insertar en el nodo hoja como a continuación estudiamos.
- De no estar lleno el nodo hoja, la inserción es posible en dicho nodo y termina la inserción de la clave.
- El comportamiento característico de los árboles B se pone de manifiesto ahora. De estar la hoja llena, la inserción no es posible en dicho nodo, entonces se divide el nodo (incluyendo virtualmente la clave nueva) en dos nodos 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 insertarla en el nodo antecedente. Es por esto por lo que se dice que el **árbol crece hacia arriba**. En esta ascensión de claves medianas puede ocurrir que llegue al nodo raíz, entonces ésta se divide en dos nodos y la clave enviada hacia arriba se convierte en una nueva raíz. Esta es la forma de que el árbol B crezca en altura.

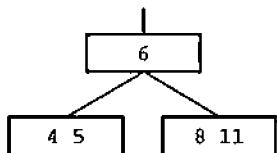
Un seguimiento de creación de un árbol B de orden 5 con las claves enteros sin signo. Al ser de orden 5 el número máximo de claves en cada nodo será 4. Las claves que se van a insertar:

6 11 5 4 8 9 12 21 14 10 19 28 3 17 32 15 16 26 27

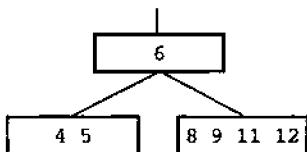
- Con las cuatro primeras se completa el primer nodo (recordar que también es llamado página), eso sí, ordenadas crecientemente a la manera de árbol de búsqueda.



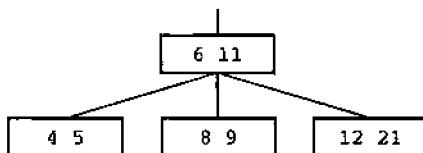
- La clave siguiente, 8, encuentra el nodo ya lleno. La clave mediana de las cinco claves es 6. El nodo lleno se divide en dos, excepto la clave mediana que «sube» y se convierte en nueva raíz:



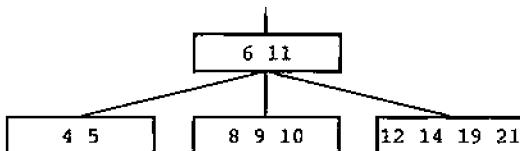
- Las siguientes claves, 9, 12, se insertan siguiendo el criterio de búsqueda, en el nodo rama derecha de la raíz por ser ambas claves mayores que 6:



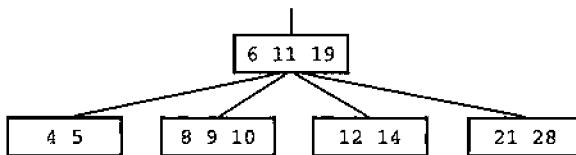
- La clave 21 sigue por el camino de búsqueda, derecha del nodo raíz. El nodo donde debe de insertarse está lleno, se parte en dos nodos, y la clave mediana 11 asciende por el camino de búsqueda para ser insertada en el nodo raíz:



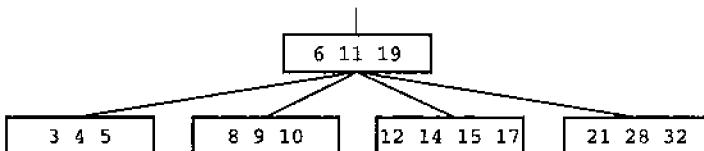
- Las siguientes claves, 14, 10, 19, son insertadas en los nodos de las ramas que se corresponden con el camino de búsqueda:



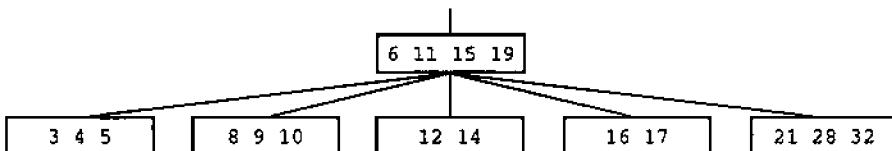
- Al insertar la clave 28 de nuevo se va a producir el proceso de división del nodo derecho del último nivel y ascensión por el camino de búsqueda de la clave mediana 19:



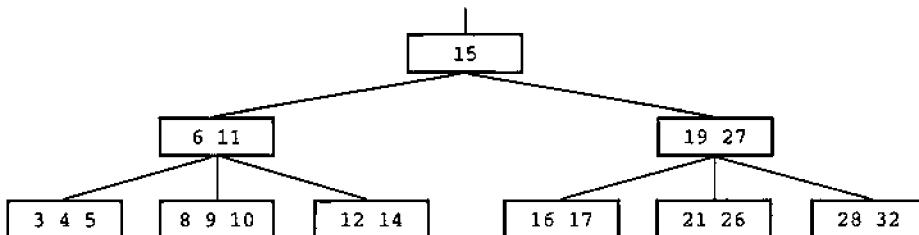
- Continuando con la inserción, las claves 3 17 32 15 son insertadas en los nodos hoja siguiendo el camino de búsqueda:



- Al llegar la clave 16, «baja» por el camino de búsqueda (rama derecha de clave 11 en nodo raíz). El nodo donde va a ser insertado está lleno. Se produce la división en dos del nodo y la ascensión de la clave mediana 15:



- Por último, al insertar las claves 26 y 27, primero se llena el nodo más a la derecha del último nivel con la clave 26. Con la clave 27, que sigue el mismo camino de búsqueda, ocurre que provoca que se divida en dos el nodo y ascienda al nodo padre la clave mediana 27. Ahora bien, ocurre que el nodo padre también está lleno por lo que también se divide en dos, sube la clave mediana 15, pero como el nodo dividido es la raíz, con esta clave se forma un nuevo nodo raíz: el árbol ha crecido en altura. El árbol crece hacia arriba, hacia la raíz.



En esta formación de árbol B que hemos ido realizando podemos observar estos dos hechos relevantes:

- Una división de un nodo, prepara a la estructura para inserciones simples de nuevas claves.

- Siempre es la clave mediana la que «sube» al nodo antecedente. La mediana no tiene por qué coincidir con la clave que se está insertando. Por lo que podemos afirmar que no importa el orden en que lleguen las claves en el balanceo del árbol.

12.4. BÚSQUEDA DE UNA CLAVE EN UN ÁRBOL B

En los algoritmos de inserción se hace uso de la operación de búsqueda; por ello se expone a continuación el algoritmo de búsqueda de una clave en un árbol B.

El algoritmo sigue la misma estrategia que la búsqueda en un árbol binario de búsqueda. Salvo que en los árboles B cuando estamos posicionados en una página (nodo) hay que inspeccionar las claves de que consta. La inspección da como resultado la posición de la clave, o bien el camino a seguir en la búsqueda.

La operación es realizada por un procedimiento que tiene como entrada la raíz del árbol B y la clave a localizar. Devuelve un valor lógico (true si encuentra la clave), la dirección de la página que contiene a la clave y la posición dentro del nodo. El examen de cada página se realiza en el procedimiento auxiliar Buscarnodo.

```
procedure Buscarnodo (Clave: Tipoclave ; P: Ptrb;
                      var Encontrado: boolean; var K:Posicion);
{Examina la página referenciada por P. De no encontrarse, K será el
 índice de la rama por donde ''bajar''}
begin
  if Clave < P^.claves[1] then
    begin
      Encontrado:= false;
      K:=0 {Rama por donde descender}
    end
  else begin
    {Examina claves de la página}
    K:= P^.Cuenta; {Número de claves de la página}
    while (Clave< P^.claves[k]) and (K> 1) do
      K:= K-1;
    Encontrado := (Clave = P^.Claves[K])
  end
end;
```

El procedimiento Buscar controla el proceso de búsqueda:

```
procedure Buscar (Clave: Tipoclave; Raiz: Ptrb;
                  var Encontrado:boolean;var N:Ptrb; var Pos:Posicion);
begin
  if Arbolvacio(Raiz) then
    Encontrado:= false
  else begin
    Buscarnodo(Clave, Raiz, Encontrado, Pos);
    if Encontrado then
      N:= Raiz {Dirección de la página}
    else {Pos es el índice de la rama}
      Buscar(Clave,Raiz^.Ramas[Pos], Encontrado, N, Pos)
  end
end;
```

12.5. ALGORITMO DE INSERCIÓN EN UN ÁRBOL B DE ORDEN m

La inserción de una nueva clave en un árbol B es relativamente simple, como ocurre en la búsqueda. Si hay que insertar una clave en una página (nodo) que no está llena (número de claves $< m-1$), el proceso de inserción involucra a esa página únicamente. Sólo si la página está llena la inserción afecta a la estructura del árbol. La inserción de la clave mediana en la página antecesora puede a su vez causar el desbordamiento de la misma, con el resultado de propagarse el proceso de partición, pudiendo llegar a la raíz. Esta es la única forma de aumentar la altura del árbol B.

Una formulación recursiva va a ser la adecuada para reflejar el proceso de propagación en la división de las páginas, debido a que se realiza en el sentido inverso por el camino de búsqueda.

El diagrama de bloques del proceso de inserción se muestra en la Figura 12.2

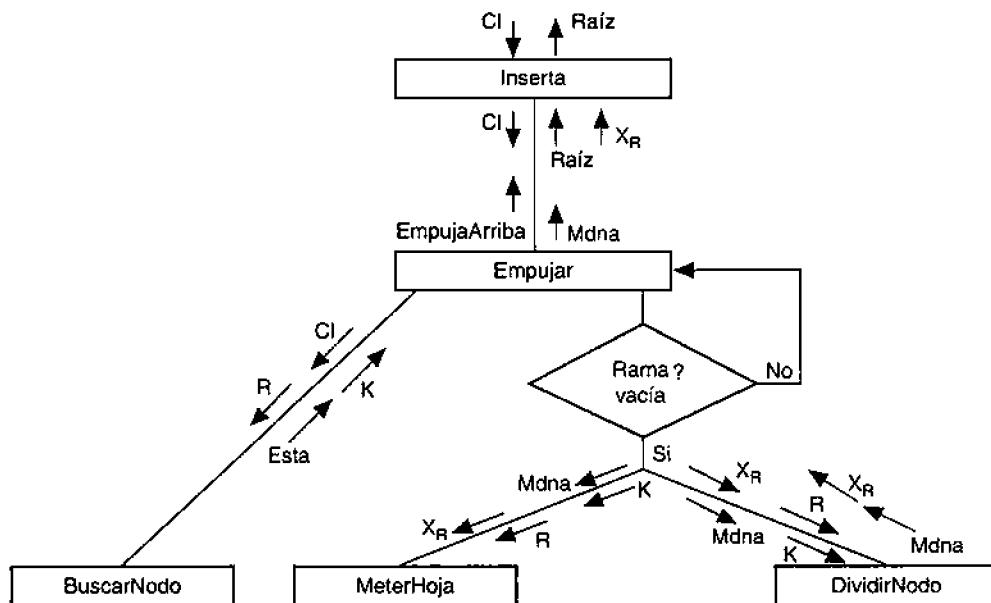
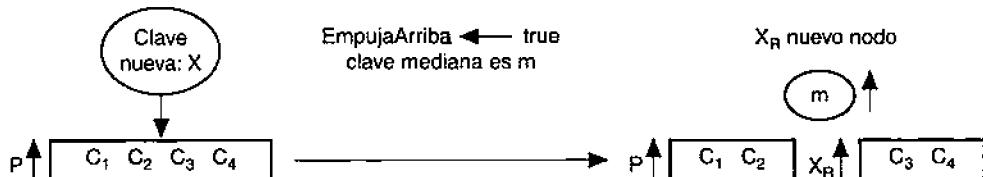


Figura 12.2. Diagrama de bloques del proceso de inserción.

El procedimiento de inserción recibe la clave nueva y la raíz del árbol B. Este procedimiento pasa control al procedimiento empujar que «baja» por el camino de búsqueda determinado por Buscarnodo, hasta llegar a un nodo hoja. El procedimiento MeterHoja realiza la acción de añadir la clave en el nodo. Si el nodo tiene ya el máximo de claves, es llamado el procedimiento dividirnodo para que a partir del nodo lleno se formen dos nodos y ascienda la clave mediana.



La propagación hacia arriba del proceso de división puede llegar hasta la raíz, por tanto aumentar la altura del árbol. La variable EmpujaArriba detecta este hecho en el procedimiento Insertar.

Procedimiento Inserta

```
procedure Inserta (Cl: Tipoclave; var Raiz: Ptrb);
var
  EmpujaArriba: boolean;
  X : Tipoclave;
  Xr, P : Ptrb;
begin
  Empujar(Cl, Raiz, EmpujaArriba, X, Xr);
  if EmpujaArriba then {La propagación del proceso de división llegó a la
                        raíz, es creado nuevo nodo}
    begin
      new(P);
      P^.Cuenta:= 1;
      P^.Claves[1]:= X;{Clave mediana que ha ascendido}
      P^.Ramas[0]:= Raiz;
      P^.Ramas[1]:= Xr;
      Raiz:= P {Nueva raíz}
    end
  end;
```

Procedimiento Empujar

Este procedimiento controla la realización de las tareas más importantes en el proceso de inserción de una clave. Lo primero que hace es «bajar» por el camino de búsqueda hasta llegar a una rama vacía. Cuando esto ocurre se devuelve la clave y se activa el interruptor que indica que la clave "sube". Al subir por el camino de búsqueda, primero se encuentra con el nodo hoja. Si hay hueco, se inserta la clave y el proceso termina. De estar lleno, contador de claves cuenta tiene el máximo de ellas, el procedimiento Dividirnodo crea un nuevo nodo (Xr tiene su dirección) que junto al nodo original se reparten las claves, y la clave mediana sigue «subiendo» por el camino de búsqueda. La forma más fácil de codificar el hecho de «bajar» y luego «subir» por el camino de búsqueda es mediante llamadas recursivas a Empujar, de tal forma que cuando las llamadas retornan control se está volviendo («subiendo») por el camino de búsqueda, con el switch EmpujaArriba se detecta si ha habido división de nodo y por tanto hay una clave mediana.

```

procedure Empujar(Cl:Tipoclave;R:Ptrb;var EmpujaArriba: boolean;
                  var Mdna: Tipoclave; var Xr: Ptrb);
var
  K : Posicion;
  Esta: boolean;{Detecta que la clave ya esté en el nodo}
begin
  if Arbolvacio(R) then
    begin {Termina la recursión, estamos en rama vacía}
      EmpujaArriba:= true;
      Mdna:= Cl;
      Xr:= nil
    end
  else begin
    Buscarnodo(Cl, R, Esta, K);{K: rama por donde seguir}
    if Esta then
      begin
        writeln (Cl, 'No permitido claves repetidas');
        halt(1)
      end;
    Empujar(Cl,R^.Ramas[K],EmpujaArriba, Mdna, Xr);
    if EmpujaArriba then
      if R^.Cuenta< Max then {No está lleno el nodo}
        begin
          EmpujaArriba:= false; {Proceso termina}
          MeterHoja(Mdna, Xr, R, K);{Insertará la clave Mdna en el nodo
                                      R, posición K+1}
        end
      else begin {Nodo lleno}
        EmpujaArriba:= true;
        Dividirnodo(Mdna, Xr, R, K, Mdna, Xr)
      end
    end
  end
end;

```

Procedimiento MeterHoja

A este procedimiento se llama cuando ya se ha determinado que hay hueco para añadir a la página una nueva clave. Se le pasa como argumentos la clave, la dirección de la página (nodo), la dirección de la rama con el nodo sucesor, y la posición a partir de la cual se inserta.

```

procedure MeterHoja(X:Tipoclave;Xder, P:Ptrb; K:Posicion);
var
  I: Posicion;
begin
  with P^do
  begin
    for I:= Cuenta downto K+1 do
      begin {Son desplazadas claves/ramas para insertar X}
        Claves[I+1]:= Claves[I];
        Ramas[I+1]:= Ramas[I]
      end;
  end;

```

```

Claves[K+1]:= X;
Ramas[K+1]:= Xder;
Cuenta:= Cuenta+1
end
end;

```

Procedimiento de división de un nodo lleno

El nodo donde hay que insertar la clave X y la rama correspondiente está lleno. A nivel lógico es dividido el nodo en dos nodos y la clave mediana enviada hacia arriba para una reinscripción posterior. Para ello lo que se hace es crear un nuevo nodo donde se llevan las claves mayores de la clave mediana, dejando en el nodo original las claves menores. Primero se determina la posición que ocupa la clave mediana, teniendo en cuenta la posición central del nodo y la posición donde debería de insertarse la nueva clave. A continuación se divide el nodo, se inserta la nueva clave en el nodo que le corresponde y por último se «extrae» la clave mediana.

```

procedure DividirNodo(X:Tipoclave;Xder:P:Ptrb;K:Posicion;
                      var Mda: Tipoclave; var Mder:Ptrb);
var
  I, Posmda: Posicion;
begin
  if K <= Min then {Min es la posición central)
                     {La clave se situa a la izquierda}
    Posmda:= Min
  else
    Posmda:= Min+1;
  new(Mder);           { Nuevo nodo }
  with P^do
    begin
      for I:= Posmda+1 to Max do
      begin
        Mder^.Claves[I-Posmda]:= Claves[I];{Es desplazada la mitad
                                                derecha al nuevo nodo, la clave mediana se queda en nodo
                                                izquierdo}
        Mder^.Ramas[I- Posmda]:= Ramas[I]
      end;

      Mder^.Cuenta:= Max- Posmda; {Claves en el nuevo nodo}
      Cuenta:=Posmda;{Claves que quedan en nodo izquierdo}
      {Inserción de clave X y su rama derecha}
      if K<= Min then
        Meterhoja(X,Xder,P,K)      {inserta en nodo izquierda}
      else
        Meterhoja(X, Xder, Mder, K-Posmda);
        {extrae mediana. del nodo izquierdo }
      Mda:= Claves[Cuenta];
      Mder^.Ramas[0]:= Ramas[Cuenta]; {Rama, del nuevo nodo es la rama de
                                       la mediana}
      Cuenta:= Cuenta-1 {Disminuye ya que se quita la mediana}
    end
  end;

```

12.6. RECORRIDO EN UN ÁRBOL B DE ORDEN m

El recorrido en un árbol B consiste en visitar todos los nodos del árbol, y para cada nodo todas las claves de que consta. Con una pequeña variación de los recorridos de un árbol binario, se puede aplicar al de un árbol B.

Se presenta a continuación el recorrido que visita las claves en orden creciente, es un recorrido en *inorden*. En primer lugar, la versión recursiva:

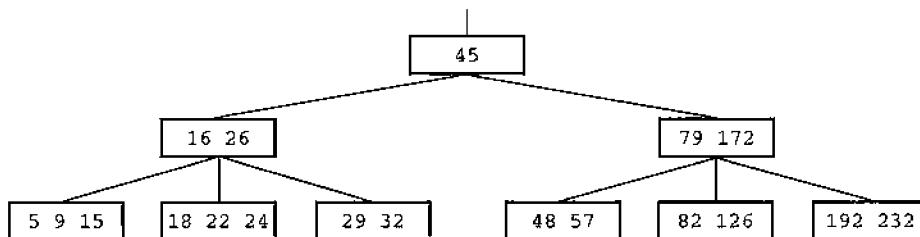
```
procedure InordenB(R: Ptrb);
var
  I: integer;
begin
  if not Arbolvacio(R) then
    begin
      InordenB(R^.Ramas[0]);
      for I := 1 to R^.Cuenta do
        begin
          writeln(R^.Claves[I]);
          InordenB(R^.Ramas[I])
        end
      end
    end;
end;
```

La versión no recursiva utiliza una pila para almacenar la dirección del nodo y el índice de la rama.

```
procedure InordenB (R: Ptrb);
var
  I: integer;
  P: Ptrb;
  Pila: Ptrpla;
begin
  Pcrear(Pila);
  P := R;
  repeat
    I := 0;
    while not Arbolvacio(P) do
      begin
        Pmeter(P,I, Pila);
        P := P^.Ramas[I]
      end;
    if not Pvacia(pila) then
      begin
        Psacar(P,I, Pila);
        I := I+1;
        if I <= P^.Cuenta then
          begin
            writeln(P^.Claves[I]); {Proceso de la clave}
            if I < P^.Cuenta then
              Pmeter(P,I, Pila);
            P := P^.Ramas[I]
          end
      end
    until Pvacia(Pila) and Arbolvacio(P)
end;
```

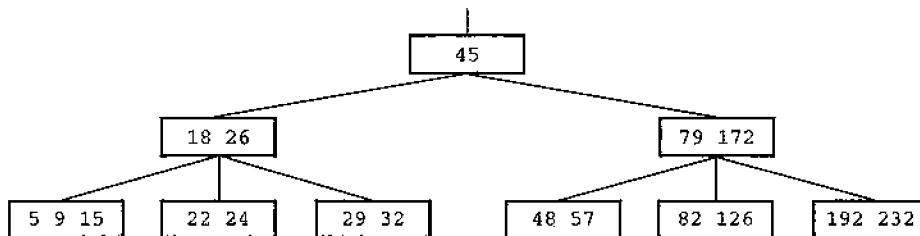
12.7. ELIMINACIÓN DE UNA CLAVE EN UN ÁRBOL B DE ORDEN m

Al igual que las inserciones de una clave siempre se hacen en una página hoja, las eliminaciones siempre se harán en una página hoja. Es evidente que una clave no tiene por qué estar en un nodo hoja, para ello se aplica la siguiente característica de los árboles B: si una clave no está en una hoja, su clave predecesora o sucesora (predecesora o sucesora en el orden natural) estará en un nodo hoja.



Una primera consecuencia es que si la clave a eliminar no está en una hoja, se pondrá a la clave predecesora en la posición ocupada por la clave eliminada y se suprimirá la clave en la hoja. Así se observa la figura con el árbol B de orden 5.

Se quiere eliminar la clave 16, esta clave no está en una hoja; el sucesor inmediato de 16, que es 18, se pone en la posición de 16 y luego 18 se suprime en el nodo hoja en que se encuentra:

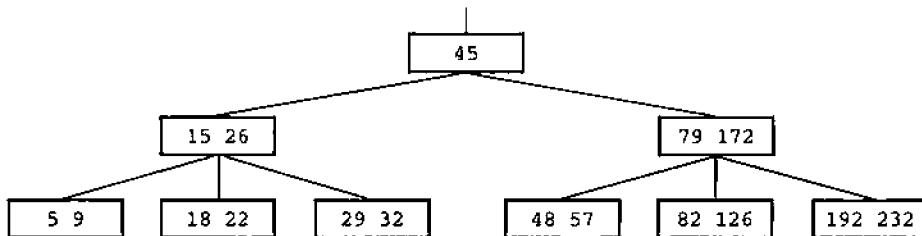


Esta supresión que acaba de realizarse es el más fácil, es el caso en el que la hoja donde se va a eliminar contiene un número de claves mayor que el mínimo de claves que puede haber en un nodo, una vez eliminada no hay que hacer una acción posterior.

En el caso de que la hoja donde se vaya a eliminar la clave tenga el número mínimo de claves, al eliminar una clave se va a quedar con menos claves que el mínimo exigido por nodo. Claro está que hay que realizar una operación de movimiento de claves para que el árbol siga siendo un árbol B: se procede a examinar las hojas inmediatamente adyacentes al nodo con el mismo antecedente. Si una de estas hojas tiene más del número mínimo de claves, puede moverse una de ellas para así restablecer el mínimo de clave. 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

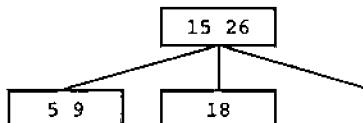
nodo padre para que a su vez descienda de éste otra clave al nodo que se quiere restaurar. Así queda restaurado el nodo y sigue siendo un árbol B.

En el árbol resultante de la eliminación anterior se elimina ahora la clave 24. El nodo que la contiene tiene dos claves, entonces se elimina la clave y se analiza los nodos contiguos del mismo antecesor. Resulta que en el nodo izquierdo tiene tres claves, por lo que asciende la clave 15 al nodo padre, y a su vez de éste desciende la clave 18 para ser insertada en el nodo a restaurar. El árbol resultante:

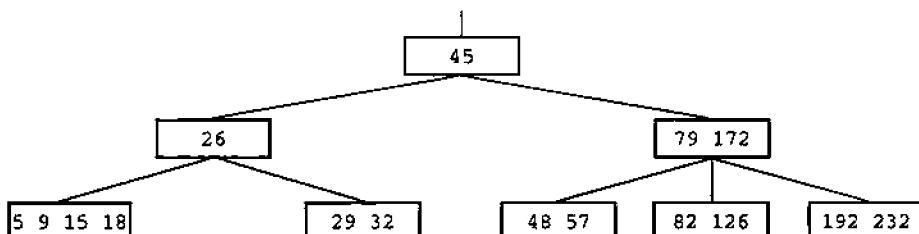


Aún queda otro caso más complejo que los dos anteriores. Al eliminar la clave 22 del nodo hoja, éste se queda con una menos que el mínimo y las dos hojas contiguas tienen únicamente el mínimo de claves. La estrategia a seguir en este caso: se toma la hoja a eliminar, su contigua y la clave mediana de ambas, procedente del nodo antecesor, y se combinan como un nuevo nodo hoja. Claro está que esta combinación puede dejar al nodo antecesor con un número de claves menor que el mínimo, entonces el proceso se propaga hacia arriba, hacia la raíz. En el caso límite, la última clave será extraída de la raíz y luego disminuirá la altura del árbol B.

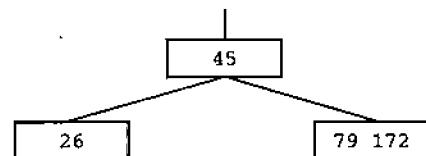
Partiendo del último árbol B, ahora queremos eliminar la clave 22. El nodo donde se encuentra se queda con una sola clave, y los nodos adyacentes tienen justamente el mínimo de claves. La siguiente combinación restaura al nodo:



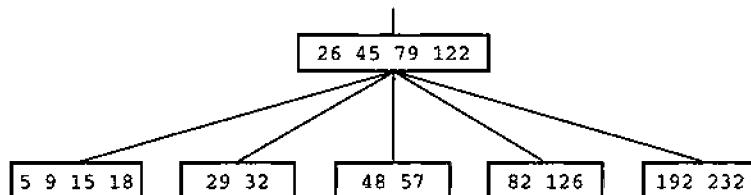
El árbol que resulta inmediatamente después del proceso:



Este proceso ha dejado al nodo antecesor con una sola clave, menos que el mínimo necesario. El proceso se propaga al nivel anterior, produciéndose esta combinación:



El árbol B que resulta



ha disminuido la altura del árbol, el proceso de combinación de dos nodos ha alcanzado a la raíz.

Diagrama de bloques del proceso de eliminación

El proceso empieza con la llamada al procedimiento Eliminar, éste pasa control a EliminarRegistro que es el encargado de controlar todo el proceso.

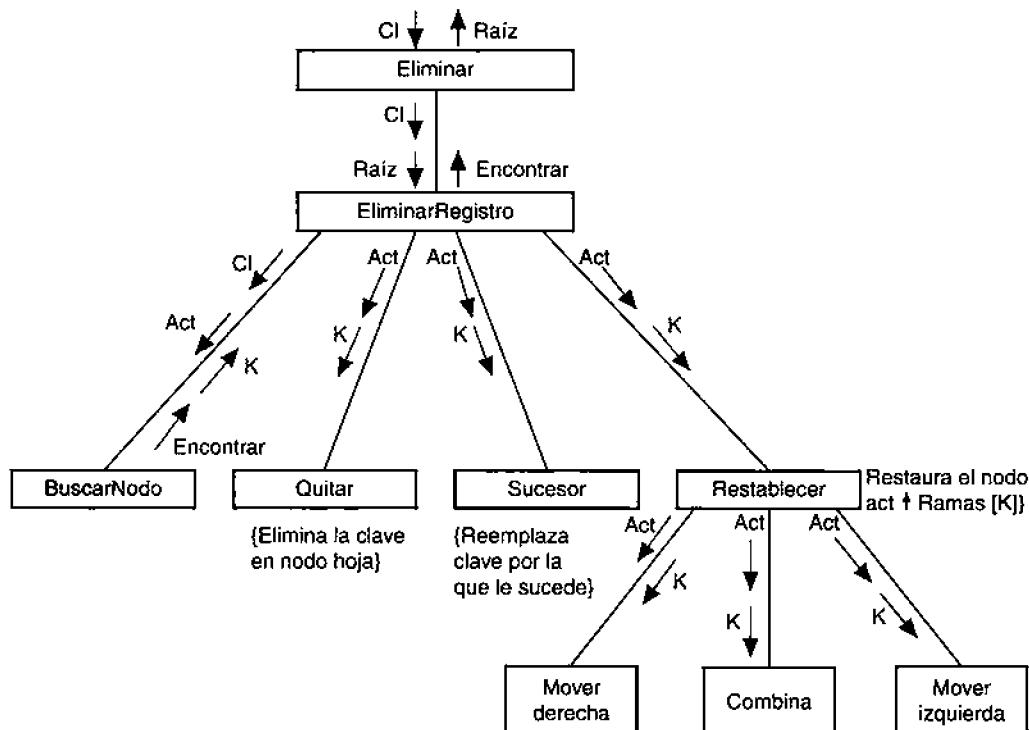


Figura 12.3. Diagramas de bloques del proceso de eliminación.

Procedimiento Eliminar

El procedimiento Eliminar tiene la misma estructura que Insertar. Tiene como misión principal devolver la nueva raíz si la raíz inicial se ha quedado sin claves.

```
procedure Eliminar (Cl: Tipoclave; var Raiz: Ptrb);
var
  Encontrado: boolean;
  P: Ptrb;
begin
  EliminarRegistro(Cl, Raiz, Encontrado);
  if not Encontrado then
    writeln('Error:la clave no está en el árbol')
  else if Raiz^.Cuenta= 0 then {La raíz se ha quedado vacía}
  begin {Se libera el nodo y se obtiene la nueva raíz}
    P:= Raiz;
    Raiz:= Raiz^.Ramas[0];
    dispose(P)
  end
end;
end;
```

Procedimiento EliminarRegistro

Este procedimiento controla el proceso de borrado de una clave. Primero busca el nodo donde se encuentra la clave a eliminar. Si el nodo es una hoja, llama a la rutina que elimina. De no ser hoja, es hallado el inmediato sucesor de la clave, se coloca en el nodo donde se encuentra la clave; después se suprime la clave sucesor en el nodo hoja.

La forma más fácil de expresar este proceso es la recursiva; al retornar la llamada recursiva debe de comprobar si hay un número de claves suficientes, mayor o igual que el mínimo necesario (2 en el caso de árbol B de orden 5), de no ser así hay que mover claves al nodo para alcanzar el número necesario de claves.

```
procedure EliminarRegistro(Cl:Tipoclave; R: Ptrb;
                           var Encontrado: boolean);
var
  K: Posicion;
begin
  if Arbolvacio(R) then
    Encontrado:= false {Se ha llegado "debajo" de una hoja, la clave no
                        está en el árbol}
  else with R^ do
  begin
    Buscarnodo(Cl, R, Encontrado, K);
    if Encontrado then
      if Ramas[K-1]= nil then {Las ramas están indexadas desde índice 0
                                a Max, por lo que este nodo es hoja}
        Quitar(R, K)
      else begin {No es hoja}
        Sucesor(R, K); {Reemplaza Claves[K]por su sucesor}
        EliminarRegistro(Claves[K],Ramas[K],Encontrado); {Elimina la
                                                          clave sucesora en su nodo}
      end;
    end;
  end;
```

```

        if not Encontrado then {Es una inconsistencia, la clave debe de
                                estar en el nodo}
        begin
            writeln('Error en el proceso ');
            Halt (1)
        end
    end
else {No ha sido localizada la clave}
    EliminarRegistro(C1, Ramas[K], Encontrado);
{Las llamadas recursivas devuelven control a este punto del
procedimiento: se comprueba que el nodo hijo mantenga un número de
claves igual o mayor que el mínimo necesario}
    if Ramas[K]<> nil then {Condición de que no sea hoja}
        if Ramas[K]^ .Cuenta < Min then
            Restablecer(R, K)
    end
end;

```

Procedimiento Quitar

Esta rutina recibe la dirección del nodo y la posición de la clave a eliminar. Elimina la clave junto a la rama que le corresponde.

```

procedure Quitar(P: Ptrb; K: Posicion);
var
    J: Posicion;
begin
    with P^ do
    begin
        with P^ do
        begin
            for J:= K+1 to Cuenta do
            begin
                Claves[J-1]:= Claves[J]; {Desplaza una posición a la izquierda, con
                                         ello es eliminada la clave}
                Ramas[J-1]:= Ramas[J]
            end;
            Cuenta:= Cuenta - 1
        end
    end;

```

Procedimiento Sucesor

En esta rutina se busca la clave inmediatamente sucesor de la clave K, que por la propiedad de los árboles B está en una hoja, y ésta reemplaza a la clave K.

```

procedure Sucesor(P: Ptrb; K: Posicion);
var
    Q: Ptrb;
begin
    Q:= P^.Ramas[K];
    while Q^.Ramas[0]<> nil do
        Q:= Q^.Ramas[0];
        {Q referencia a un nodo hoja}
    P^.Claves[K]:= Q^.Claves[1]
end;

```

Procedimiento Restablecer

En este procedimiento se realizan las acciones más complejas del proceso. Restaura el nodo $P^.$ Ramas[K] el cual se ha quedado con un número de claves menor que el mínimo.

El procedimiento siempre toma una clave a partir del hermano de la izquierda (claves menores), sólo utiliza al hermano derecho cuando no hay hermano izquierdo. Puede ocurrir que no pueda tomar claves y entonces llamar a Combina para formar un nodo con dos nodos.

```
procedure Restablecer(P: Ptrb; K: Posicion);
  {P tiene la dirección del nodo antecesor del nodo P^.Ramas[K] que se
   ha quedado con menos claves que el mínimo}
begin
  if K > 0 then
    {Tiene "hermano" izquierdo}
    if P^.Ramas[K-1]^ .Cuenta > Min then
      {Tiene más claves que el mínimo y por tanto puede
       desplazarse una clave}
      MoverDrcha(P, K)
    else
      Combina(P, K)
    else
      {Sólo tiene "hermano" derecho}
      if P^.Ramas[1]^ .Cuenta > Min then
        {Tiene más claves que el mínimo}
        MoverIzqda(P, 1)
      else
        Combina(P, 1)
  end;
```

Procedimiento MoverDerecha

Este procedimiento mueve una clave del nodo antecesor (P) al nodo que se está restaurando. A su vez, asciende la clave mayor del hermano izquierdo al nodo antecesor.

```
procedure MoverDrcha(P: Ptrb; K: Posicion);
  {En este procedimiento se deja "hueco" en el nodo P^.Ramas[K]^ que es
   el nodo que tiene menos claves que el mínimo necesario, inserta la
   clave k del nodo antecesor y a su vez asciende la clave mayor(la
   más a la derecha) del hermano izquierdo}
var
  J: Posicion;
begin
  with P^.Ramas[K]^ do {es el nodo con menos claves que el mínimo}
  begin
    for J := Cuenta downto 1 do
    begin
      claves [J+1] := claves [J];
      Ramas [J+1] := Ramas [J];
    end;
```

```

Cuenta:= Cuenta+1;
Ramas[1]:= Ramas[0];
Claves[1]:= P^.Claves[K] {"baja" la clave del nodo padre}
end;
{Ahora "sube" la clave desde el hermano izquierdo al nodo padre, para
reemplazar la que antes bajó}

with P^.Ramas[K-1]^ do {Hermano izquierdo}
begin
  P^.Claves[K]:= Claves[Cuenta];
  P^.Ramas[K]^ .Ramas[0]:= Ramas[Cuenta];
  Cuenta:= Cuenta-1
end
end;

```

Procedimiento MoverIzquierda

Realiza la misma acción que el procedimiento Moverderecha, salvo que ahora la clave que asciende al nodo antecedente es la clave menor (izquierda) del nodo a restaurar.

```

procedure MoverIzqda(P: Ptrb; K: Posicion);
{Desciende la clave K(1) del nodo padre P al hijo izquierda y la
inserta en la posición más alta, de esta forma se restablece el mínimo
de claves. Después sube la clave 1 del hermano derecho.}
var
  J: Posicion;
begin
  with P^.Ramas[K-1]^ do {es el nodo con menos claves que el mínimo}
  begin
    Cuenta:= Cuenta+1;
    Claves[Cuenta]:= P^.Claves[K];
    Ramas[Cuenta]:= P^.Ramas[K]^ .Ramas[0]
  end;
  with P^.Ramas[K]^ do {Hermano derecho}
  begin
    P^.Claves[K]:= Claves[1]; {Sube al nodo padre la clave 1 del
                               hermano derecho, sustituye a la que bajó}
    Ramas[0]:= Ramas[1];
    Cuenta:= Cuenta-1;
    for J:= 1 to Cuenta do
    begin
      Claves[J]:= Claves[J+1];
      Ramas[J]:= Ramas[J+1]
    end
  end
end;

```

Procedimiento Combina

En este procedimiento se forman un solo nodo con dos nodos. 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. Una vez finalizado el proceso, libera el nodo obsoleto.

```

procedure Combina(P: Ptrb; K: Posicion);
{Forma un nuevo nodo con el hermano izquierdo, la mediana entre el nodo
 problema y su hermano izquierdo situada en el nodo padre, y las claves
 del nodo problema. Es liberado el nodo problema}
var
  J: Posicion;
  Q: Ptrb;
begin
  Q:= P^.Ramas[K];
  with P^.Ramas[K-1]^ do {Hermano izquierdo}
  begin
    Cuenta:= Cuenta+1;
    Claves[Cuenta]:= P^.Claves[K]; {"baja" clave mediana desde el nodo
      padre}
    Ramas[Cuenta]:= Q^.Ramas[0];
    for J:= 1 to Q^.Cuenta do
    begin
      Cuenta:= Cuenta+1;
      Claves[Cuenta]:= Q^.Claves[J];
      Ramas[Cuenta]:= Q^.Ramas[J];
    end
  end;
  end;

  {Son reajustadas las claves y ramas del nodo padre debido a que
  antes descendió la clave k}
  with P^ do
  begin
    for J:= K to Cuenta-1 do
    begin
      Claves[J]:= Claves[J+1];
      Ramas[J]:= Ramas[J+1]
    end;
    Cuenta:= Cuenta-1
  end;
  dispose(Q)
end;

```

12.8. TAD ÁRBOL B DE ORDEN m

Hasta ahora se ha descrito un árbol B, cómo representar al árbol B de orden m y qué operaciones son definidas sobre la estructura; es decir, se ha definido el tipo abstracto de datos árbol B. A continuación se escribe la unidad árbol B, aunque sólo la sección de interface; la sección implementation ha sido escrita paso a paso en los apartados anteriores.

```

unit ArbolB;
interface
  const
    Orden = 5;
    Max = Orden-1; {Número máximo de claves en un nodo}
    Min = (Orden-1) div 2; {Número mínimo de claves en un nodo distinto
      de la raíz}

```

```

type
  Tipoclave = integer;
  Ptrb = ^Página;
  Posicion= 0..Max;
  Página= record
    Cuenta: 0 .. max;
    Claves: array [1 .. max] of Tipoclave;
    Ramas: array [Posicion] of Ptrb;
  end;

procedure Buscarnodo (Clave: Tipoclave ; P: Ptrb;
                      var Encontrado: boolean; var K: Posicion);
procedure Buscar (Clave: Tipoclave; Raiz: Ptrb;
                  var Encontrado: boolean;
                  var N: Ptrb; var Pos: Posicion);
procedure Inserta (Cl: Tipoclave; var Raiz: Ptrb);
procedure Empujar (Cl: Tipoclave; R: Ptrb;
                   var EmpujaArriba: boolean;
                   var Mdna: Tipoclave; var Xr: Ptrb);
procedure MeterHoja(X: Tipoclave; Xder, P: Ptrb; K: Posicion);
procedure DividirNodo(X: Tipoclave; Xder, P: Ptrb;
                      K: Posicion; var Mda: Tipoclave;
                      var Mder: Ptrb);
procedure MostrarB(R: Ptrb); {Es el recorrido en inorden}
function Arbolvacio(R: Ptrb):boolean;

{Operaciones para el proceso de borrado de una clave}
procedure Eliminar (Cl: Tipoclave; var Raiz: Ptrb);
procedure EliminarRegistro (Cl: Tipoclave; R: Ptrb;
                           var Encontrado: boolean);
procedure Restablecer(P: Ptrb; K: Posicion);
procedure Quitar(P: Ptrb; K: Posicion);
procedure Sucesor(P: Ptrb; K: Posicion);
procedure MoverIzqda(P: Ptrb; K: Posicion);
procedure MoverDrcha(P: Ptrb; K: Posicion);
procedure Combina(P: Ptrb; K: Posicion);
implementation
  .....
begin
end.

```

Esta unidad se emplea a título de ejemplo para crear un árbol B de orden 5, considerando el tipo de las claves entero.

Las claves son generadas aleatoriamente para evitar problemas, antes de insertar la clave se examina si ya está en el árbol. También se realizan operaciones de eliminación de claves en el árbol creado.

CODIFICACIÓN

```

program Procesa_arbolB(input,output);
uses crt, Arbolb;
const
  Mx=9999;

```

```

var
  Rb: Ptrb;
  Cl: integer;
  Op,N: integer;

procedure Menu(var Opc: integer);
begin
  clrscr;
  gotoxy(10,2); writeln(' 1. Insertar n claves');
  gotoxy(10,4); writeln(' 2. Insertar 1 clave');
  gotoxy(10,6); writeln('3. Eliminar clave');
  gotoxy(10,8); writeln('4. Listar claves');
  gotoxy(10,10);writeln('5. Fin proceso');
  repeat
    gotoxy(10,13);writeln('Opción ?: ');
    readln(Opc)
  until Opc in {1..5};
end;

procedure Annade_Claves(var Rb:Ptrb;N:integer);
var
  K:0..100;
  X:integer;
  P:Posicion;
  Esta:boolean;
  Nd:Ptrb;
begin
  K:=0;
  write('Claves insertadas: ');
  repeat
    X:=random(Mx);
    Buscar(X,Rb,Esta,Nd,P);
    if not Esta then
      begin
        Inserta(X,Rb);
        write(X,' ');
        K:=K+1;
      end
  until K=N;
  writeln;
end;
begin  {Bloque principal}
  Rb:=nil;
  Randomize;
  repeat
    Menu(Op);
    case Op of
      1: begin
        clrscr;gotoxy(4,5);
        write('¿Cuántas claves?: ');readln(N);
        Annade_Claves(Rb,N)
      end;
      2: Annade_Claves(Rb,1);
      3: begin
        write('Clave que se desea eliminar?:');readln(Cl);
        Eliminar(Cl,Rb)
      end;
    end;
  end;

```

```

        4: MostrarB(Rb)
    end;
    repeat until readkey in [#0..#255]
    until Op=5
end.

```

12.9. REALIZACIÓN DE UN ÁRBOL B EN MEMORIA EXTERNA

En los apartados anteriores se ha definido la estructura árbol B, su representación en memoria y las operaciones para insertar claves y eliminar. Ahora bien, esta estructura para que realmente sea útil debe de estar almacenada en un archivo externo y ser manejada accediendo al archivo.

Los procesos de búsqueda, inserción, eliminación y recorrido van a ser los mismos, salvo un cambio muy importante en la representación y es que ya no se utilizan variables dinámicas, los enlaces no serán punteros sino número de registros en los que se almacena el nodo o página descendiente.

Para crear un nuevo nodo (página) hay que buscar un «hueco» en el archivo, representado por un número de registro, posicionar el pointer del archivo en ese «hueco» y escribir el nodo. De igual forma, cada vez que se modifica un nodo por un cambio en las claves hay que escribir el registro en la posición del archivo correspondiente.

En esta realización de árboles B las ramas van a ser número de registro. La función Huecos concede el número de registro donde almacenar un nodo. Para ello se sigue la siguiente estrategia:

- De la pila de registros liberados, se saca el elemento cima.
- Si la pila está vacía, se obtiene el siguiente registro libre de la memoria externa.

El proceso de liberación de nodos supone almacenar en la pila de registros libres el nodo que se libera. Los procedimientos de inserción y eliminación son los mismos que en la realización con estructuras dinámicas, salvo las modificaciones que supone no utilizar punteros sino números de registro. Son utilizados dos archivos, el que almacena el árbol propiamente dicho (llamado Fichab) y otro para almacenar el número de registro del nodo raíz y la pila de registros libres (llamado FHuecos). Es inmediato pensar que hay que grabar el número de registro de la raíz para que a partir de él pueda accederse a toda la estructura.

Los tipos de datos para realizar a un árbol B de orden 5 en memoria externa:

```

const
    Orden= 5;
    Max=Orden-1;           {Número máximo de claves en un nodo}
    Min=(Orden-1) div 2; {Número mínimo de claves en un nodo distinto de la
                         raíz}

type
    Tipoclave= ...;
    Apuntador= integer;

```

```

Posicion= 0..Max;
Pagina= record
  Cuenta: 0 .. Max;
  Claves: array [1 .. Max] of Tipoclave;
  Ramas: array [Posicion] of Apuntador
end;
Fichero = file of Pagina;
Flibre = file of integer;

```

Para adaptarse a la nueva manera de direccionar los nodos y para grabar o cargar la estructura son necesarias estas nuevas rutinas.

Función Hueco

Devuelve la posición del siguiente registro libre. Éste puede venir de la pila de registros liberados, o bien el registro siguiente al último concedido.

```

function Hueco (Pila: Tpila; var F: Archivo):integer;
begin
  if not Pvacia (Pila) then
    Hueco:= Psacar(Pila)
  else
    Hueco:= filesize(F)
end;

```

Procedimiento Inicial

Será el primer procedimiento en ser llamado. Asigna archivos y vuelca en la pila los número de registros liberados. Devuelve la raíz del árbol B.

```

procedure Inicializa (var F: Archivo; var Fh: Flibre;
                      var R:integer; var Pila: Tpila);
var
  I: Apuntador;
begin
  Pcrear (Pila);
  assign(F, 'datosab.dat');
  assign(Fh, 'huecos.dat');
  {$i-}
  reset(Fh);
  if ioreult < > 0 then
    begin
      rewrite(Fh);
      rewrite(F);
      R:= Vacio {constante predefinida, parametriza árbol vacío}
    end
  else begin
    reset(F);
    if eof(Fh) then
      R:= Vacio
  end
end;

```

```

else
  read(Fh,R) {El primer registro es la raíz del árbol B}
end;
{$i+}
while not eof(Fh) do
begin
  read(Fh,I);
  Pmeter(Pila,I)
end
end;

```

Procedimiento de Finalizar

Este procedimiento será el último en llamarse. En él se escribe como primer registro el número de registro de la raíz del árbol. A continuación, los registros liberados que están en la pila.

```

procedure fin (var Fh: Flibre; var F: Archivo;
               R : Apuntador; var Pila: Tpila);
var
  H: Apuntador;
begin
  reset(Fh);
  write(Fh,R);
  while not Pvacia(Pila) do
  begin
    write (Fh,Cima(Pila));
    Pborrar(Pila)
  end;
  close(Fh);
  close(F)
end;

```

Procedimiento de liberar un nodo

Simplemente mete en la pila de registros libres el número de registro donde está el nodo a ser liberado.

```

procedure Liberar (P: Apuntador; var Pila: Tpila);
begin
  Pmeter(Pila, P);
end;

```

Operación de recorrido del árbol B

Recorre el árbol B que se encuentra en el disco. Pasa por cada una de las claves y son escritas en pantalla.

```

procedure recorrer (R: Apuntador; var F: Archivo);
var
  I: Posicion;
  Pg: Pagina;

```

```

begin
  if R <> vacio then
    begin
      seek(F,R);
      read(F,Pg);
      recorrer(Pg.Ramas[0],F);
      for I:=1 to Pg.Cuenta do
        begin
          write(Pg.Claves[I],' ');
          recorrer(Pg.Ramas[I],F)
        end
      end
    end;
end;

```

Unidad con las operaciones de árbol B en archivo

A continuación se muestra la unidad para realizar el tipo abstracto de datos árbol B en archivo.

A las rutinas anteriores, se añaden las ya escritas para el proceso de inserción y el proceso de borrado de claves. Presentan pequeñas diferencias debido fundamentalmente a la no utilización de punteros para acceder a un nodo o página.

Ahora este acceso se realiza con:

```

seek(F,R);
read(F,Pg);

```

Siendo F el archivo, R el número de registro y Pg el nodo.

Unidad para realizar el TAD árbol B de orden 5 en un archivo

```

unit ArbolB;
interface
uses PilaPun;
const
  Vacio = 29999; {Marca que indica registro vacío}
  Orden = 5;
  Max= Orden-1; {Número máximo de claves en un nodo}
  Min= (Orden-1) div 2; {Número mínimo de claves en un nodo distinto de
                        la raíz}
type
  Tipoclave= integer; {Por comodidad en las pruebas adoptamos este tipo
                       de clave}
  Posicion= 0..Max;
  Pagina= record
    Cuenta: 0 .. Max;
    Claves: array [1 .. Max] of Tipoclave;
    Ramas: array [Posicion] of Apuntador
  end;
  Archivo= file of Pagina;
  Flibre= file of Apuntador;
var
  Pila: Tpila;
function Hueco (var Pla:Tpila;var F:Archivo):Apuntador;

```

```

procedure Inicializa (var F: Archivo; var Fh: Libre;
                      var R:integer);
procedure Fin (var Fh: Libre; var F: Archivo; R: Apuntador);
procedure Liberar (P: Apuntador; var Pila: Tpila);
procedure Recorrer (R: Apuntador; var F: archivo);
procedure Buscarnodo (Clave: Tipoclave; P: Pagina;
                      var Encontrado: boolean; var K: Posicion);
procedure Buscar (Clave: Tipoclave; R: Apuntador;
                  var Encontrado: boolean; var Pos: Posicion;
                  var P: Apuntador; var F: Archivo);
function Arbolvacio(R: Apuntador): boolean;
{Operaciones específicas para el proceso de inserción}
procedure Inserta (Cl: Tipoclave; var R: Apuntador;
                   var F: Archivo);
procedure Empujar(Cl: Tipoclave; R: Apuntador;
                  var EmpujaArriba: boolean;
                  var Mdn: Tipoclave; var Xr: Apuntador;
                  var F: Archivo );
procedure DividirNodo(X: Tipoclave; Xder: Apuntador;
                      var P: Pagina; K: Posicion;
                      var Mda: Tipoclave; var Xd: Apuntador;
                      var F: Archivo );
procedure MeterHoja(X: Tipoclave; Xder: Apuntador;
                     var P: Pagina; K: Posicion);
{Operaciones para el proceso de borrado de una clave}
procedure Eliminar (Cl: Tipoclave; var Raiz: Apuntador;
                     var F: Archivo);
procedure EliminaRegistro (Cl: Tipoclave; var P: Apuntador;
                           var Encontrado: boolean; var F: Archivo);
procedure Restablecer (P: Apuntador; K: Posicion; var F: Archivo);
procedure Quitar(P: Apuntador; K: Posicion; var F: Archivo);
procedure Sucesor(P: Apuntador; K: Posicion; var F: Archivo);
procedure Moverderecha(P: Apuntador; K: Posicion; var F: Archivo);
procedure Moverizquierda(P: Apuntador; K: Posicion; var F: Archivo);
procedure Combina(P: Apuntador; K: Posicion; var F: Archivo);
implementation

function Arbolvacio(R: Apuntador): boolean;
begin
  Arbolvacio := R=vacio
end;

function Hueco (var Pla: Tpila; var F: Archivo): Apuntador;
var
  H: Apuntador;
begin
  if not Pvacia(Pla) then
    Psacar(H, Pla)
  else
    H := filesize(F);
  Hueco := H
end;

procedure Inicializa (var F: Archivo; var Fh: Libre;
                      var R: integer);
var

```

```

I: Apuntador;
begin
  Pcrear(Pila);
  assign(F,'datosab.dat');
  assign(Fh,'huecos.dat');
  {$i-}
  reset(Fh);
  if ioreasult < > 0 then
  begin
    rewrite(Fh);
    rewrite(F);
    R:= Vacio {constante predefinida, parametriza árbol vacío}
  end
  else begin
    reset(F);
    if eof(Fh) then
      R:= Vacio
    else
      read(Fh,R) {El primer registro es la raíz del árbol B}
  end;
  {$i+}
  while not eof(Fh) do
  begin
    read(Fh,I);
    Pmeter(I,Pila)
  end
end;

procedure Fin (var Fh: Flibre; var F: Archivo;R:Apuntador);
var
  H: Apuntador;
begin
  reset(Fh);
  write(Fh, R);
  while not Pvacia(Pila) do
  begin
    H:= Pcima(Pila);
    write(Fh,H);
    Pborrar(Pila)
  end;
  close(Fh);
  close(F)
end;

procedure Liberar (P: Apuntador; var Pila: Tpila);
begin
  Pmeter(P,Pila);
end;

procedure recorrer (R: Apuntador; var F: archivo);
var
  I: Posicion;
  Pg: Pagina;
begin
  if R < > vacio then
  begin

```

```
seek(F, R);
read(F, Pg);
recorrer(Pg.Ramas[0], F);
for I:=1 to Pg.Cuenta do
begin
  write(Pg.Claves[I], ' ');
  recorrer(Pg.Ramas[I], F)
end
end;
end;

procedure buscarnodo (Clave:Tipoclave; P: Pagina;
                      var Encontrado: boolean; var K: Posicion);
begin
  if Clave < P.Claves[1] then
    begin
      Encontrado:=false;
      K:=0
    end
  else begin
    K:= P.Cuenta;
    while (Clave < P.Claves[k]) and (k>1) do
      k:=k-1;
    Encontrado:= (Clave = P.Claves[k])
  end
end;
end;

procedure buscar (Clave: Tipoclave; R: Apuntador;
                  var Encontrado:boolean;
                  var Pos: Posicion; var P: Apuntador;
                  var F: Archivo );
var
  Pg:Pagina;
begin
  Encontrado:=false;
  while not Encontrado and (R <> vacio ) do
  begin
    seek(F,R);
    read(F,Pg);
    buscarnodo(Clave,Pg,Encontrado,Pos);
    if Encontrado then
      P:= R
    else
      R:= Pg.Ramas[Pos]
  end
end;
end;

procedure Inserta (Cl:Tipoclave; var R:Apuntador;var F: Archivo);
var
  P: Pagina;
  Sube: boolean;
  Mdna: Tipoclave;
  N, Xr: Apuntador;
begin
  Empujar(Cl,R,Sube,Mdna,Xr,F);
  if Sube then
```

```

begin
  P.Cuenta:=1;
  P.Claves[1]:= Mdna;
  P.Ramas[0]:= R;
  P.Ramas[1]:= Xr;
  N:= Hueco(Pila, F);
  seek(F, N);
  write(F, P);
  R:= N
end
end;

procedure Empujar(Cl: Tipoclave; R: Apuntador;
                  var EmpujaArriba: boolean;
                  var Mdna: Tipoclave; var Xr:Apuntador;
                  var F: Archivo);
var
  K: Posicion;
  Pg: Pagina;
  Esta: boolean;
begin
  if R=vacio then
  begin
    EmpujaArriba:=true;
    Mdna:=Cl;
    Xr:=vacio
  end
  else begin
    seek(F,R);
    read(F,Pg);
    buscarnodo(Cl,Pg,Esta,K);
    if Esta then
    begin
      writeln ('Clave ya existe. Revisar código');
      halt(1)
    end
    else begin
      Empujar (Cl,Pg.ramas[K],EmpujaArriba,Mdna,Xr,F);
      if EmpujaArriba then
        if Pg.Cuenta < Max then
        begin
          EmpujaArriba:=false;
          Meterhoja(Mdna,Xr,Pg,K);
          seek(F,R);
          write(F,Pg)
        end
        else begin
          EmpujaArriba:=true;DividirNodo(Mdna,Xr,Pg,K,Mdna,Xr,F);
          seek(F,R);
          write(F,Pg)
        end
      end
    end
  end
end;
end;

```

```

procedure DividirNodo(X: Tipoclave; Xder: Apuntador;
                      var P: Pagina; K: Posicion;
                      var Mda:Tipoclave;var Xd:Apuntador;
                      var F: Archivo );
var
  Mde: Pagina;
  j,i,pos:integer;
begin
  if K<= Min then
    pos:= Min
  else
    pos:= Min+1;
  Xd:= Hueco(Pila,F);
  for i:= pos+1 to Max do
  begin
    Mde.Claves[i-pos]:=P.Claves[i];
    Mde.Ramas[i-pos]:=P.Ramas[i]
  end;
  Mde.Cuenta:= Max-pos;
  P.Cuenta:=pos;
  if K<= Min then
    Meterhoja(X,Xder,P,K)
  else
    Meterhoja(X,Xder,Mde,K-pos);
  Mda:= P.Claves[P.Cuenta];
  Mde.Ramas[0]:= P.Ramas[P.Cuenta];
  P.Cuenta:= P.Cuenta -1 ;
  seek(F,Xd);
  write(F,Mde);
end;

procedure MeterHoja(X: Tipoclave; Xder: Apuntador;
                     var P: Pagina;K: Posicion);
var
  i:integer;
begin
  for i:= P.Cuenta downto K+1 do
  begin
    P.Claves[i+1]:=P.Claves[i];
    P.Ramas[i+1]:= P.Ramas[i]
  end;
  P.Claves[k+1]:= X;
  P.Ramas[k+1]:= Xder;
  P.Cuenta:= P.Cuenta+1
end;

procedure Eliminar (Cl: Tipoclave; var Raiz: Apuntador;
                    var F: Archivo);
var
  Aux: Apuntador;
  Encontrado: boolean;
  Pg: Pagina;

```

```

begin
  EliminaRegistro(C1,Raiz,Encontrado,F);
  if not Encontrado then
    writeln('No Encontrada la clave a eliminar')
  else begin
    seek(F,Raiz);
    read(F,Pg);
    if Pg.Cuenta = 0 then
      begin
        Aux:= Raiz;
        Raiz:= Pg.Ramas[0];
        Liberar(Aux,Pila)
      end
    end
  end;
end;

procedure EliminaRegistro (Cl:Tipoclave;var P:Apuntador;
                           var Encontrado:boolean;var F: Archivo);
var
  K: Posicion;
  Aux,Pg: Pagina;
begin
  if P= vacio then
    Encontrado:=false
  else begin
    seek(F,P);
    read(F,Pg);
    with Pg do
      begin
        buscarnodo(Cl,Pg,Encontrado,K);
        if not Encontrado then
          EliminaRegistro (Cl,Ramas[K],Encontrado,F)
        else if Ramas[0]= vacio then
          Quitar(F,K,F)
        else begin
          Sucesor(P,K,F);
          seek(F,P); {De nuevo se lee porque ha cambiado}
          read(F,Pg);
          EliminaRegistro(claves[K],Ramas[K], Encontrado,F)
        end;
        if Ramas[K]< > vacio then
          begin
            seek(F,Ramas[K]);
            read(F,Pg);
            if Cuenta< Min then
              Restablecer(P,K,F)
            end
          end
        end
      end;
  end;
end;

procedure Restablecer (P: Apuntador; K: Posicion; var F:Archivo);
var
  J: Apuntador;
  Aux: Pagina;

```

```

begin
  seek(F,P);
  read(F,Aux);
  if K > 0 then
  begin
    J:=Aux.Ramas[K-1];
    seek(F,J);
    read(F,Aux);
    if Aux.Cuenta > Min then
      Moverderecha(P,K,F)
    else
      Combina(P,K,F)
  end
  else begin
    seek(F,P);
    read(F,Aux);
    j:= Aux.Ramas[1];
    seek(F,J);
    read(F,Aux);
    if Aux.Cuenta > Min then
      Moverizquierda(P,1,F)
    else
      Combina(P,1,F)
  end
end;

procedure Quitar(P:Apuntador; K: Posicion; var F: Archivo);
var
  i: Apuntador;
  Pg: Pagina;
begin
  seek(F,P);
  read(F,Pg);
  for i:=k+1 to Pg.Cuenta do
  begin
    Pg.Claves[i-1]:= Pg.Claves[i];
    Pg.Ramas[i-1]:= Pg.Ramas[i]
  end;
  Pg.Cuenta:= Pg.Cuenta-1;
  seek(F,P);
  write(F,Pg)
end;

procedure Sucesor(P:Apuntador; K: Posicion; var F: Archivo);
var
  D: Tipoclave;
  Q: Apuntador;
  Aux,Pg:Pagina;
begin
  seek(F,P);
  read(F,Pg);
  Q:= Pg.Ramas[k];
  seek(F,Q);
  read(F,Aux);

```

```

while Aux.Ramas[0] <> vacio do
begin
  seek(F,Aux.Ramas[0]);
  read(F,Aux)
end;
D:=Aux.Claves[1];
Pg.Claves[K]:=D;
seek(F,P);
write(F,Pg)
end;

procedure Moverderecha(P:Apuntador; K: Posicion; var F:Archivo);
var
  j:Apuntador;
  Aux,Auxiz,Auxdr:Pagina;
begin
  seek(F,P);
  read(F,Aux);
  seek(F,Aux.Ramas[k-1]);
  read(F,Auxiz);
  seek(F,Aux.Ramas[k]);
  read(F,Auxdr);
  for J:=Auxdr.Cuenta downto 1 do
  begin
    Auxdr.Claves[j+1]:=Auxdr.Claves[j];
    Auxdr.Ramas[j+1]:=Auxdr.Ramas[j]
  end;
  Auxdr.Ramas[1]:=Auxdr.Ramas[0];
  Auxdr.Cuenta:=Auxdr.Cuenta+1;
  Auxdr.Claves[1]:=Aux.Claves[K];
  Aux.Claves[k]:=Auxiz.Claves[Auxiz.Cuenta];
  Auxdr.Ramas[0]:=Auxiz.Ramas[Auxiz.Cuenta];
  Auxiz.Cuenta:=Auxiz.Cuenta-1;
  seek(F,P);
  write(F,Aux);
  seek(F,Aux.Ramas[k-1]);
  write(F,Auxiz);
  seek(F,Aux.Ramas[k]);
  write(F,Auxdr)
end;

procedure Moverizquierda(P:Apuntador; K: Posicion; var F: Archivo);
var
  Aux,Auxiz,Auxdr: Pagina;
  J: Posicion;
begin
  seek(F,P);
  read(F,Aux);
  seek(F,Aux.Ramas[K-1]);
  read(F,Auxiz);
  seek(F,Aux.Ramas[K]);
  read(F,Auxdr);
  Auxiz.Cuenta:=Auxiz.Cuenta+1;
  Auxiz.Claves[Auxiz.Cuenta]:=Aux.Claves[K];

```

```

Auxiz.Ramas[Auxiz.Cuenta]:=Auxdr.Ramas[0];
Auxdr.Ramas[0]:=Auxdr.Ramas[1];
Aux.Claves[K]:= Auxdr.Claves[1];
Auxdr.Cuenta:=Auxdr.Cuenta-1;
for j:=1 to Auxdr.Cuenta do
begin
  Auxdr.Claves[j]:=Auxdr.Claves[j+1];
  Auxdr.Ramas[j]:=Auxdr.Ramas[j+1]
end;
seek(F,P);
write(F,Aux);
seek(F,Aux.Ramas[k-1]);
write(F,Auxiz);
seek(F,Aux.Ramas[k]);
write(F,Auxdr)
end;

procedure Combina(P:Apuntador; K: Posicion; var F:Archivo);
var
  Dcho,Izqdo:Apuntador;
  j: Posicion;
  Aux,Auxiz,Auxdo:Pagina;
begin
  seek(F,P);
  read(F,Aux);
  Dcho:=Aux.Ramas[k];
  Izqdo:=Aux.Ramas[k-1];
  seek(F,Dcho);
  read(F,Auxdo);
  seek(F,Izqdo);
  read(F,Auxiz);
  Auxiz.Cuenta:=Auxiz.Cuenta+1;
  Auxiz.Claves[Auxiz.Cuenta]:=Aux.Claves[k];
  Auxiz.Ramas[Auxiz.Cuenta]:=Auxdo.Ramas[0];
  for j:=1 to Auxdo.Cuenta do
  begin
    Auxiz.Cuenta:= Auxiz.Cuenta+1;
    Auxiz.Claves[Auxiz.Cuenta]:=Auxdo.Claves[j];
    Auxiz.Ramas[Auxiz.Cuenta]:=Auxdo.Ramas[j]
  end;
  for j:= K to Aux.Cuenta-1 do
  begin
    Aux.Claves[j]:=Aux.Claves[j+1];
    Aux.Ramas[j]:=Aux.Ramas[j+1]
  end;
  Aux.Cuenta:=Aux.Cuenta-1 ;
  seek(F,P);
  write(F,Aux);
  seek(F,Izqdo);
  write(F,Auxiz);
  liberar(Dcho,Pila)
end;
begin
end.

```

Programa de gestión de un árbol B en memoria externa

```

program Arbol_b;
uses crt, ArbolB,Pilapun;
const
  B= 999;
var
  F:Archivo;
  Fh:Flibre;
  Apun,R,D:Apuntador;
  Re, L, Cl: integer;
  Pos:Posicion;
  Encontrado:boolean;
  Res:char;
begin
  R:=vacio;
  clrscr;
  Inicializa(F,Fh,R);
  randomize;
  writeln ('Elementos presentes');
  Recorrer(R,F);
  repeat
    writeln;
    writeln('1. Añadir claves');
    writeln('2. Eliminar Clave');
    writeln('3. SALIR');
    repeat
      Readln(Re)
    until Re in [1..3];
    if Re= 1 then
    begin
      for L:=1 to 2 do
      begin
        C1:= random(B);
        Buscar(C1,R,Encontrado,Pos,D,F);
        if not Encontrado then
          Inserta(C1,R,F)
      end;
      writeln;
      Recorrer(R,F);
    end;
    if Re= 2 then
    begin
      writeln('Clave a eliminar');
      readln(Cl);
      Eliminar(Cl,R,F);
      Recorrer(R,F)
    end
  until (Re = 3);
  Fin(Fh,F,R);
  repeat until keypressed
end.

```

RESUMEN

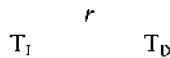
Un *árbol-B* es una estructura de datos que se puede utilizar para implementar una tabla que reside en memoria externa, tal como un disco. Los *árboles B* se denominan también *árboles 2-3* y son una clase de árboles no binarios sino ternarios, que obligan a que todas las hojas se encuentren al mismo nivel.

Un *árbol 2-3* permite que el número de hijos de un nodo interno varíen entre dos y tres. Un *árbol 2-3* es un *árbol* en el que cada nodo interno (no hoja) tiene dos o tres hijos y todas las hojas están al mismo nivel. Una definición recursiva de un *árbol 2-3* es: T es un *árbol 2-3 de altura h* si:

1. T está vacío (un *árbol 2-3* de altura 0).

O alternativamente:

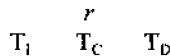
2. T es de la forma



donde r es un nodo y T_1 y T_D son los dos *árboles 2-3*, cada uno de altura $h-1$. En este caso, T_1 se llama *subárbol izquierdo* y T_D se llama *subárbol derecho*.

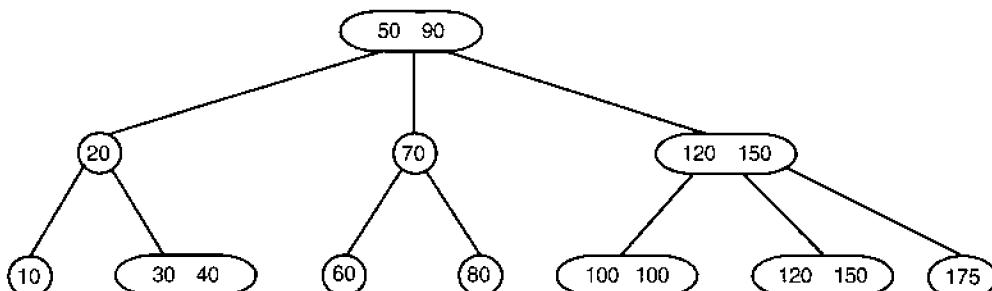
O bien,

3. T es de la forma



donde r es un nodo y T_1 , T_C , T_D son *árboles 2-3*, cada uno de altura $h-1$. En este caso, T_1 se llama *subárbol izquierdo*, T_C se llama *subárbol central* y T_D se llama *subárbol derecho*.

Un *árbol 2-3* no es necesariamente binario, pero está siempre equilibrado:



En un *árbol 2-3*, los nodos internos pueden tener dos o tres hijos, permitiendo que el número de hijos varíe; los algoritmos de inserción y supresión pueden mantener fácilmente el equilibrio del árbol.

EJERCICIOS

- 12.1. Dada la secuencia de claves enteras: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80, dibuja el árbol B de orden 5 cuya raíz es R, que se corresponde con dichas claves.
- 12.2. En el árbol R del problema 1, elimina la clave 91 y dibuja el árbol resultante. Vuelve a eliminar, ahora, la clave 48. Dibuja el árbol resultante, ¿ha habido reducción en el número de nodos?
- 12.3. Modifique la rutina de inserción de un árbol B de orden m para que si se intenta añadir una clave a un nodo que ya está lleno, se efectúe una búsqueda de un hermano no lleno antes de partir el nodo.
- 12.4. En un árbol B de orden 5 se insertan las claves 1, 2, 3, ..., n . ¿Qué claves originan la división de un nodo? ¿Qué claves hacen que la altura del árbol crezca?
- 12.5. En el árbol B de orden 5 del ejercicio 12.4, las claves se eliminan en el mismo orden de creación. ¿Qué claves hacen que los nodos se queden con un número de claves menor que el mínimo y den lugar a la unión de dos nodos? ¿Qué claves hacen que la altura del árbol disminuya?
- 12.6. Prevemos tener un archivo de un millón de registros, cada registro ocupará 50 bytes de memoria. Los bloques de memoria son de 1.000 bytes de longitud y además hay un puntero por cada bloque que ocupa 4 bytes de memoria. Diseñar una organización con árboles B para este archivo.
- 12.7. El procedimiento de búsqueda de una clave se ha realizado con una llamada recursiva al final del procedimiento. Volver a escribir el procedimiento eliminando la llamada recursiva.
- 12.8. Un árbol B^* es un árbol B en el que cada nodo está, al menos, lleno en las dos terceras partes (en vez de la mitad), menos quizás el nodo raíz. La inserción de nuevas claves en el árbol B^* supone que si el nodo que le corresponde está lleno mueve las claves a los nodos hermanos (de manera similar a como se mueven en la eliminación cuando hay que restaurar el número de claves de un nodo). Con lo cual se pospone la división del nodo hasta que los dos nodos hermanos estén completamente llenos. Entonces, éstos pueden dividirse en tres nodos, cada uno de los cuales estará lleno en sus dos terceras partes. Especifique los cambios que necesita el algoritmo de inserción de una clave en un árbol B para aplicarlo a un árbol B^* .
- 12.9. Dado un árbol B^* según está definido en 12.8, especificar los cambios necesarios que necesita el algoritmo de eliminación de una clave en un árbol B para aplicarlo a un árbol B^* .
- 12.10. Dada la secuencia de claves enteras del ejercicio 12.1: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80, dibuja el árbol B^* de orden 5 cuya raíz es R, que se corresponde con dichas claves.
- 12.11. Con las claves del árbol B de orden 5 del ejercicio 12.4 dibuja un árbol B^* de orden 5.
- 12.12. En el árbol B^* de orden 5 del ejercicio 12.11, las claves se eliminan en el mismo orden de creación: 1, 2, 3, 4, ..., n . ¿Qué claves hacen que los nodos se queden con un número de claves menor que el mínimo y den lugar a la unión de dos nodos? ¿Qué claves hacen que la altura del árbol disminuya?

PROBLEMAS

- 12.1. Dada la secuencia de claves enteras: 190, 57, 89, 90, 121, 170, 35, 48, 91, 22, 126, 132 y 80, dibuja el árbol B de orden 5 cuya raíz es R, que se corresponde con dichas claves.

- 12.2. En el árbol R del problema 12.1 elimina la clave 01 y dibuja el árbol resultante. Vuelve a eliminar ahora la clave 48. Dibuja el árbol resultante, ¿ha habido reducción en el número de nodos?
- 12.3. 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 y en el rango de las unidades de 1.000. (Así, a Madrid le corresponde del 1 al 1.000; a Toledo, del 1.001 al 2.000...)
- Escribir un programa que permita gestionar la información indicada, formando una estructura en memoria de árbol B con un máximo de 6 claves por página. La clave de búsqueda del árbol B es el número de orden del centro, además tiene asociado el nombre del centro. El programa debe permitir añadir centros, eliminar, buscar la existencia de un centro por la clave y listar los centros existentes.
- 12.4. En el problema 12.3 cuando se termina la ejecución se pierde toda la información. Pues bien, modificar el programa 1 para que al terminar la información se grabe la estructura en un archivo de nombre centros.txt.
- Escribir un programa que permita leer el archivo centros.txt para generar a partir de él la estructura de árbol B. La estructura puede experimentar modificaciones —nuevos centros, eliminación de alguno existente—, por lo que al terminar la ejecución debe de escribirse de nuevo el árbol en el archivo.
- 12.5. Se quiere dar más contenido a la información tratada en el problema 12.3. 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.
- 12.6. A la estructura ya creada del problema 12.3, añadir las operaciones que responden a estos requerimientos:
- Dada una provincia cuyo rango de centros es conocido, por ejemplo de 3.001 a 3.780, eliminar en todos sus centros escolares los libros que estén repetidos en cada centro, y que informe del total de libros liberados.
 - Dado un centro n , en su biblioteca se desea que de ciertos libros haya m ejemplares (por ejemplo, 5.)

Grafos. Representación y operaciones

CONTENIDO

- 13.1. Grafos y aplicaciones.
- 13.2. Conceptos y definiciones.
- 13.3. Representación de los grafos.
- 13.4. TAD grafo.
- 13.5. Recorrido de un grafo.
- 13.6. Componentes conexas de un grafo.
- 13.7. Componentes fuertemente conexas de un grafo.
- 13.8. Matriz de caminos.
- 13.9. Puntos de articulación de un grafo.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Este capítulo introduce al lector a conceptos matemáticos importantes denominados *grafos* que tienen aplicaciones en campos tan diversos como sociología, química, geografía, ingeniería eléctrica e industrial, etc. Los grafos se estudian como estructuras de datos o tipos abstractos de datos. Este capítulo estudia los algoritmos más importantes para procesar grafos. También representa operaciones importantes y algoritmos de grafos que son significativos en informática.

13.1. GRAFOS Y APLICACIONES

Con los árboles binarios se han representado relaciones entre objetos en las que existe una jerarquía. Con frecuencia, es necesario representar relaciones arbitrarias entre obje-

tos de datos. Los grafos se clasifican en *dirigidos* y no *dirigidos* y son modelos naturales de tales relaciones. Así, los grafos se usan para representar redes de alcantarillado, redes de comunicaciones, circuitos eléctricos, etc. Una vez modelado el problema mediante un grafo se pueden hacer estudios sobre diversas propiedades. Para ello se utilizan algoritmos concretos que resuelvan ciertos problemas.

La teoría de grafos ha sido aplicada en el estudio de problemas que surgen en áreas diversas de las ciencias, como la química, la ingeniería eléctrica o la investigación operativa. El primer paso siempre será representar el problema como un grafo. En esta representación cada elemento, cada objeto del problema, forma un nodo. La relación, comunicación o conexión entre los nodos da lugar a una arista, que puede ser dirigida o bidireccional (no dirigida). En la Figura 13.1 aparece una red de comunicaciones.

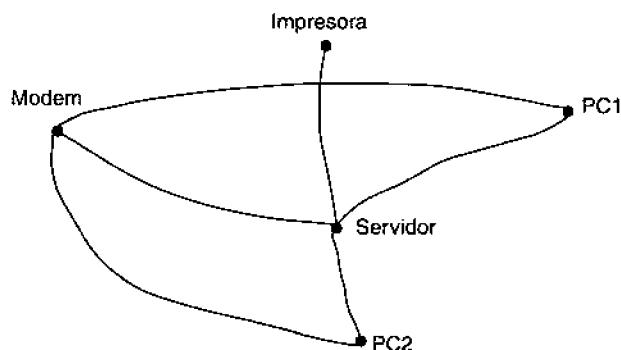
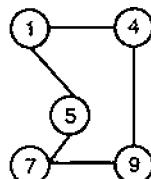


Figura 13.1. Un grafo como red de comunicaciones.

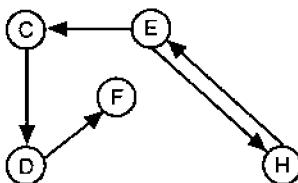
13.2. CONCEPTOS Y DEFINICIONES

Un grafo consiste en un conjunto de vértices o nodos V y un conjunto de arcos A . Se representa con el par $G = (V, A)$.

EJEMPLO 13.1



El conjunto de vértices $V = \{1, 4, 5, 7, 9\}$ y el conjunto de arcos $A = \{(1,4), (5,1), (7,9), (7,5), (4,9), (4,1), (1,5), (9,7), (5,7), (9,4)\}$ forman el grafo no dirigido $G = (V, A)$.

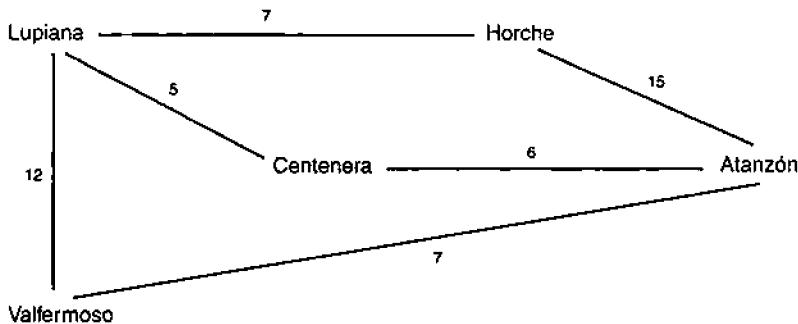
EJEMPLO 13.2

El conjunto de vértices $V = \{C, D, E, F, H\}$ y el conjunto de arcos $A = \{(C,D), (D,F), (E,H), (H,E), (E,C)\}$ forman el grafo dirigido $G = \{V, A\}$.

Un arco o arista está formado por un par de nodos y se escribe (u,v) siendo u,v el par de nodos. Un *grafo es dirigido (digrafo)* si los pares de nodos que forman los arcos son ordenados, se representan $u \rightarrow v$. El segundo ejemplo es un grafo dirigido.

Un *grafo no dirigido* es aquel que los arcos están formados por pares de nodos no ordenados, no apuntados, se representa $u - v$. El grafo del primer ejemplo no es dirigido.

Dado el arco (u,v) es un grafo, se dice que los vértices u y w son adyacentes. Si el grafo es dirigido, el vértice u es adyacente a v , y v es adyacente de u . Un arco tiene, a veces, asociado un factor de peso, en cuyo caso se dice que es un grafo valorado. Pensemos, por ejemplo, en un grafo formado por los pueblos que forman una comarca; un par de pueblos está unido o no por un camino vecinal y un factor de peso que es la distancia en kilómetros:



forman un grafo valorado no dirigido.

13.2.1. Grado de entrada, grado de salida

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 , $\text{grado}(v)$, es el número de aristas 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 , $\text{gradent}(v)$, es el número de arcos que llegan a v , grado de salida de v , $\text{gradsal}(v)$, es el número de arcos que salen de v . A veces no se sabe distinguir entre arco y arista, la diferencia está en qué aristas son arcos hacia los dos sentidos.

Por ejemplo, $\text{grado}(Lupiana) = 3$. Es el grafo dirigido del ejemplo 13.2, $\text{gradent}(D) = 1$ y el $\text{gradsal}(D) = 1$.

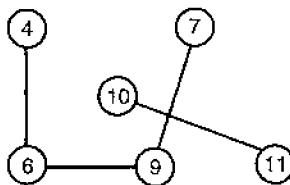
13.2.2. Camino

Un camino P en grafo G de longitud n desde un vértice v_0 a v_n es la secuencia de $n + 1$ vértices:

$$P = (v_0, v_1, v_2, \dots, v_n)$$

tal que $(v_i, v_{i+1}) \in A(\text{arcos})$ para $0 \leq i \leq n$. La longitud del camino es el número de arcos que lo forma.

EJEMPLO 13.3



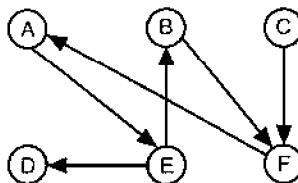
$P_1 = (4, 6, 9, 7)$ es un camino de longitud 3.

En algunos grafos se dan arcos desde un vértice a sí mismo (v, v), el camino $v - v$ se denomina bucle. No es frecuente encontrarse con grafos que tengan bucles.

Un camino $P = (v_0, v_1, v_2, \dots, v_n)$ es simple si todos los nodos que forman el camino son distintos, pudiendo ser iguales v_0, v_n (los extremos del camino). En el ejemplo anterior P_1 es un camino simple.

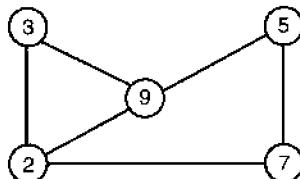
Un ciclo es un camino simple cerrado, $v_0 = v_n$ de longitud ≥ 2 , en definitiva compuesto al menos por 3 nodos.

EJEMPLO 13.4

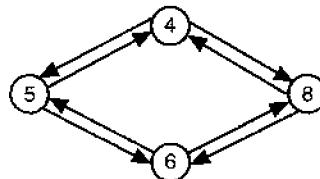


Los vértices (A, E, B, F, A) en este grafo dirigido forman un ciclo de longitud 4. Un ciclo de longitud k se denomina k -ciclo. En el ejemplo 13.4 tenemos un 4-ciclo.

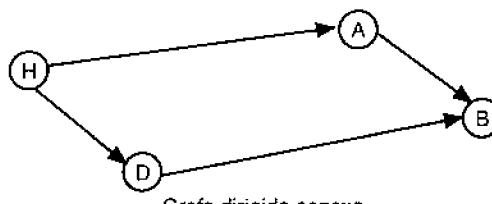
Un grafo no dirigido G es conexo si existe un camino entre cualquier par de nodos que forman el grafo. En el caso de ser un grafo dirigido podemos distinguir entre fuertemente conexo y conexo. Un grafo dirigido es fuertemente conexo si existe un camino entre cualquier par de nodos que forman el grafo; un grafo dirigido es conexo si existe una cadena que une cualquier par de vértices. Un grafo completo es aquel que tiene un arco para cualquier par de vértices.

EJEMPLO 13.5

Grafo conexo



Grafo fuertemente conexo



Grafo dirigido conexo

13.3. REPRESENTACIÓN DE LOS GRAFOS

Al pensar en los tipos de datos para representar un grafo en memoria, se debe tener en cuenta que se debe representar un número (finito) de vértices y de arcos que unen dos vértices. Se puede elegir una representación secuencial, mediante arrays; o bien una representación dinámica, mediante una estructura multienlazada. La representación mediante arrays se conoce como matriz de adyacencia, la representación dinámica se denomina lista de adyacencia. La elección de una manera u otra dependerá de las operaciones que se apliquen sobre los vértices y arcos.

13.3.1. Matriz de adyacencia

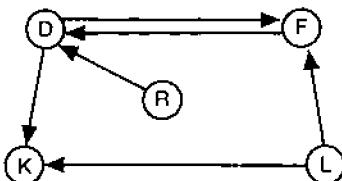
Sea $G = (V, A)$ un grafo de n nodos, suponemos que los nodos $V = \{v_1, v_2, \dots, v_n\}$ están ordenados y podemos representarlos por sus ordinales $\{1, 2, \dots, n\}$. La representación de los arcos se hace con una matriz A de $n \times n$ elementos a_{ij} definida:

$$a_{ij} \begin{cases} 1 & \text{si hay un arco } (v_i, v_j) \\ 0 & \text{si no hay arco } (v_i, v_j) \end{cases}$$

la matriz se denomina *matriz de adyacencia*. En ocasiones la matriz de adyacencia es una matriz booleana en la que el elemento a_{ij} es verdadero (*true*) si existe arco (v_i, v_j) y falso (*false*) en caso contrario.

EJEMPLO 13.6

Sea el grafo dirigido de la figura siguiente

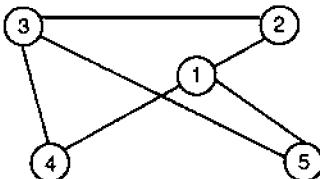


Supongamos que el orden de los vértices es el siguiente: {D, F, K, L, R}, y por tanto la matriz de adyacencia:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

EJEMPLO 13.7

Ahora el grafo de la figura es no dirigido.



La matriz de adyacencia:

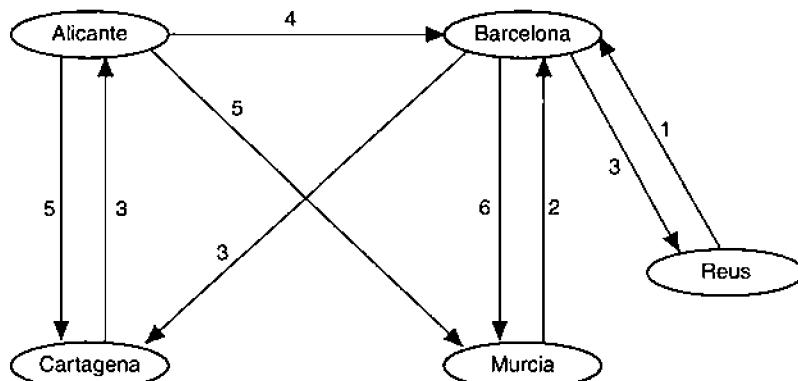
$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Podemos observar qué es una matriz simétrica. En los grafos no dirigidos la matriz de adyacencia siempre será simétrica, ya que cada arco no dirigido (v_i, v_j) se corresponde con los arcos dirigidos $(v_i, v_j), (v_j, v_i)$.

Los grafos con factor de peso, grafos valorados, pueden representarse de tal forma que si existe arco, el elemento a_{ij} es el factor de peso; la no existencia de arco supone que a_{ij} es 0 o ∞ (esto sólo si el factor de peso no puede ser 0). A esta matriz se la denomina matriz valorada.

EJEMPLO 13.8

El grafo valorado de la figura es un grafo dirigido con factor de peso



Si suponemos los vértices en el orden $V = \{\text{Alicante}, \text{Barcelona}, \text{Cartagena}, \text{Murcia}, \text{Reus}\}$ la matriz de pesos P:

$$P = \begin{pmatrix} 0 & 4 & 5 & 5 & 0 \\ 0 & 0 & 3 & 6 & 3 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Representación en Pascal

Al ser una representación estática debe de estar previsto el máximo de nodos al que puede llegar el grafo. Los nodos o vértices tienen un identificador, los cuales se guardan en un vector de cadenas. El tipo arco podríamos representarlo con un registro con un campo de tipo boolean, tomará el valor verdadero (*true*) si (v_i, v_j) es una arista, y en caso necesario, falso (*false*) otro campo de tipo numérico, que represente el factor de peso. La matriz de adyacencia que se representa es de tipo entero, tomando los valores de 0 a 1, según haya arco o no entre un par de vértices.

```

const
  Maxvert=20;
type
  Indicevert = 1 .. Maxvert;
  
```

```

Vertice= string;
Vertices=array[Indicevert,Indicevert] of 0..1
Grafo = record
  N: Indicevert;
  V: Vertices;
  A: MatAdcia
end;
var
  G: Grafo

```

En el tipo *Grafo* está definido el campo *N* (opcional) para tener contabilizados el número de vértices, no pudiendo superar el máximo establecido. La matriz *G* es una variable de tipo *Grafo*. *A* es la matriz de adyacencia. En el caso de que el grafo sea con factor de peso y éste sea de tipo numérico, puede asociarse directamente a la matriz, o bien, definir el tipo arco:

```

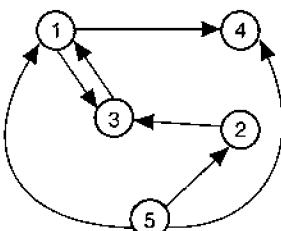
type
  ...
Arco = record
  Adte : boolean;
  Peso : real
end;
MatAdcia = array [Indicevert,Indicevert] of Arco;

```

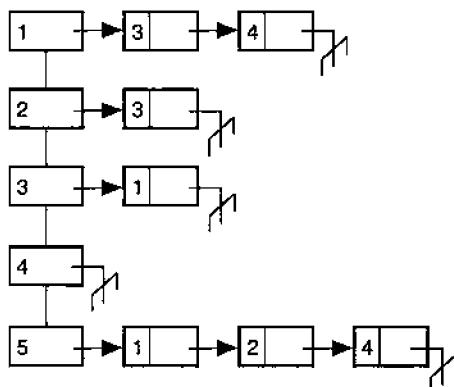
13.3.2. Listas de adyacencia

La representación de un grafo con matriz de adyacencia puede resultar poco eficiente en algunos casos. Así, cuando el número de vértices varía a lo largo del proceso, dándose el caso de que el número de ellos sea mayor que el previsto. También cuando el grafo es disperso, es decir, tiene pocos arcos, y por tanto la matriz de adyacencia tiene muchos ceros (matriz sparce), resulta poco eficiente la matriz de adyacencia ya que el espacio que ocupa es el mismo que si el grafo tuviera muchos arcos. Cuando se dan estas ineficiencias se representa un grafo mediante listas enlazadas que se denominan listas de adyacencia.

Las listas de adyacencia son una estructura multienlazada formada por una lista directorio cuyos nodos representan *los vértices* del grafo y del que además emergen una lista enlazada cuyos nodos, a su vez, representan los arcos con vértice origen el del nodo de la lista directorio. Con la figura siguiente nos acercamos a esta representación de un grafo.



La representación de este grafo dirigido mediante listas de adyacencia:



Representación en Pascal

Cada nodo de la lista directorio tiene que guardar el vértice que representa, la dirección de acceso a la lista de adyacencia (arcos que «salen» de dicho vértice) y además la dirección del nodo siguiente de la lista directorio. Gráficamente:

Vértice	Lista Adcia	Sgte
---------	-------------	------

Cada nodo de la lista de adyacencia de un vértice del grafo almacena la dirección del vértice (en la lista directorio) con el que forma un arco, en caso de ser un grafo valorado también el factor de peso, y como en todas las listas la dirección del nodo siguiente. Gráficamente:

Dir-Vértice	<Peso>	Sgte
-------------	--------	------

Los tipos de datos:

```

type
  PtrDir = ^NodoDir;
  PtrAdy = ^NodoLy;
  NodoDir = record
    Vert : string;  {identificador del vértice}
    Lady : PtrAdy;
    Sgte : PtrDir
  end;
  
```

```

NodoLy = record
  PtrV: PtrDir;
  Peso : real; {campo factor de peso, es opcional}
  Sgte : PtrAdy
end;
var
  G : PtrDir;

```

Con la variable puntero G se accede a la lista directorio, a partir de la cual se puede acceder a las listas de adyacencia.

13.4. TAD GRAFO

Los grafos, al igual que las pilas, colas... son tipos abstractos de datos. Ya hemos establecido los datos y su representación, ahora definimos las operaciones básicas sobre esos datos y su realización que será dependiente de la representación de los datos.

La operación *unión* (X, Y) añade el arco (X, Y) al grafo. En caso de ser un grafo valorado se define la operación *unión_peso* (X, Y, W) que añade el arco (X, Y) cuyo factor de peso es W.

Operación *borr_arco* (X, Y) elimina del grafo el arco (X, Y).

Operación *adyacente* (X, Y), función lógica que debe devolver true si forman un arco los vértices (X, Y).

Operación *nuevo_vértice* (X) añade el vértice X al grafo G.

Operación *borra_vértice* (X) elimina del grafo G el vértice X.

Estas operaciones son el núcleo a partir del cual se construye el grafo. En la representación con listas encadenadas se hace uso de las operaciones propias de una lista enlazada.

13.4.1. Realización con matriz de adyacencia

En esta realización los vértices están representados por el ordinal (1..n) dentro del conjunto de vértices.

Operación *unión*

```

procedure union (var G : Grafo; V1, V2 : Indicevert);
begin
  G.A[V1,V2] := 1;
end;

```

Para un grafo valorado

```

procedure union_peso (var G: Grafo; V1, V2:Indicevert, W: real);
begin

```

```

G.A[V1, V2].Adye := true;
G.A[V1,V2].Peso := W
end;

```

La operación que elimina un arco

```

procedure borr_arco (var G:Grafo; V1, V2: Indicevert);
begin
  G.A[V1,V2] := 0;
end;

```

La operación *adyacente* (*X, Y*)

```

function adyacente (G:Grafo; V1,V2:Indicevert): boolean;
begin
  adyacente := G.A[V1,V2] = 1;
end;

```

La operación *nuevo-vértice* (*X*)

```

procedure Nuevo_vertice (var G:Grafo; X:string);
begin
  with G do
    begin
      if N<=Maxvert then
        begin
          N := N+1;
          V[N] := X;
        end
      else {Error: no es posible nuevos nodos}
        end
    end;
end;

```

Operación *borra_vértice* (*X*) elimina del grafo *G* el vértice *X*.

La operación de eliminar un vértice del grafo no sólo supone suprimirlo de la lista de vértices, sino que además se debe ajustar la matriz de adyacencia, ya que el índice de los vértices posteriores al eliminado debe quedar decrementado en 1.

```

procedure borra_vertice(var G:Grafo; X:string);
var
  I:Indicevert;
  P:integer;
begin
  P := Ordinal(G,X);
  with G do
    begin
      if P in [1..N] then
        begin
          {Eliminación de vértice X}
          for I := P to N-1 do
            V[I] := V[I+1];
          {Ajuste de matriz de adyacencia}
          {Desplaza columnas}
        end;
    end;
end;

```

```

for C := P to N-1 do
  for I := 1 to N do
    A[I,C] := A[I,C+1];
    {Desplaza filas}
  for I := P to N-1 do
    for C := 1 to N do
      A[I,C] := A[I+1,C]
    {Se reduce el número de vértices}
  N := N-1
end
else {Vértice no existe en el grafo}
end
end;

```

La operación Ordinal (es una operación auxiliar).

```

function Ordinal(G:Grafo; X:string): integer;
var
  I : Indicevert
  Rd : integer;
begin
  Rd:=-1; I:=0;
  while (I<G.N) and (Rd = -1) do
  begin
    I := I+1
    if G.V[I] = X then
      Rd := I
    else
      I := I+1
  end;
  Ordinal := Rd
end;

```

13.4.2. Realización con listas de adyacencia

Las operaciones básicas con listas se implementan con variables puntero y variables dinámicas. Algunas operaciones son similares a las realizadas con el TDA lista, a pesar de lo cual son realizadas. Así, la operación auxiliar para crear un vértice.

```

function Crearvt (Vt:string) : PtrDir;
var
  V:PtrDir;
begin
  new(V);
  V^.Vert := Vt;
  V^.Lady := nil; V^.Sgte := nil;
  Crearvt := V
end;

```

Operación unión.

Dado el grafo y dos vértices que forman un arco se añade el arco en la correspondiente lista de adyacencia. La realización de `unión_peso` sólo presenta la diferencia de asignar el factor de peso al crear el nodo de la lista de adyacencia.

```

procedure union (var G:PtrDir; V1,V2:string);
var
  P,Q: PtrDir;
begin
  P := Direccion(G,V1);
  Q := Direccion(G,V2);
  if (P<>nil) and (Q<>nil) then
    with P^ do
      if Lady = nil then {lista de adyacencia está vacía}
        Lady := CrearLy(Q)
      else
        Ultimo(Lady)^.Sgte := CrearLy(Q)
  end;

```

Han surgido operaciones auxiliares, como Dirección y Último. La primera operación devuelve un puntero al nodo de la lista directorio donde se encuentra un vértice; la segunda operación devuelve un puntero al último nodo de la lista. Los códigos de ambas funciones son:

```

function Direccion(G: PtrDir; V: string): PtrDir;
var
  P,D: PtrDir;
begin
  P:= nil; D := G;
  while (P=nil) and (D<>nil) do
    if D^.Vert = V then
      P := D
    else D := D^.Sgte;
  Direccion := P
end;

function Ultimo(L: PtrAdy): PtrAdy;
begin
  if L<>nil then
    while L^.Sgte<>nil do
      L := L^.Sgte;
    Ultimo := L
end;

```

Operación elimina arco: Una vez encontrada la lista de adyacencia se procede a eliminar el nodo que representa el arco.

```

procedure borr_arco(var G:PtrDir; V1,V2:string);
var
  P,Q: PtrDir;
  R,W: PtrAdy;
  Sw: boolean;
begin
  P := Direccion(G,V1);
  Q := Direccion(G, V2);
  if(P<>nil) and (Q<>nil) then
    begin
      R := P^.Lady; W := nil; Sw := false;
      while (R<>nil) and not Sw do
        if R^.PtrV = Q then

```

```

begin
  if W = nil then
    P^.Lady := R^.Sgte
  else
    W^.Sgte := R^.Sgte;
  dispose(R);
  Sw := true
end
else begin
  W := R;
  R := R^.Sgte
end
end;
end;

```

Operación Adyacente.

```

function Adyacente(G: PtrDir; V1,V2: string):boolean;
var
  P,Q: PtrDir;
  R,W: Ptradv;
  Sw: boolean;
begin
  P := Direccion(G,V1),
  Q := Direccion(G,V2);
  if(P<>nil)and(Q<>nil) then
  begin
    (Proceso de búsqueda)
    R := P^.Lady; Sw := false;
    while(R<>nil) and not Sw do
    begin
      Sw := R^.PtrV = Q;
      if not SW then R := R^.Sgte
    end;
    Adyacente := Sw
  end
  else Adyacente := false
end;

```

Operación Nuevo_vertice (X).

Añade un vértice X del grafo a la lista directorio. Siempre se añade como último nodo.

```

procedure Nuevo_vertice(var G: PtrDir; X: string);
begin
  if G<>nil then
    Ultimo(G)^.Sgte := CrearVt(X)
  else
    G := CrearVt(X)
end;

```

Operación Borra_vertice (X) elimina del grafo G el vértice X.

La operación de eliminar un vértice del grafo supone eliminar los arcos que van a él y después suprimirlo de la lista directorio. Se utilizan operaciones auxiliares nuevas:

Anterior (G, P) devuelve un puntero a la dirección del nodo anterior y Libera que libera la memoria de cada nodo de la lista de adyacencia.

```

procedure Borra_vertice(var G: PtrDir; X: string);
var
  P,Q: PtrDir,
  R,W: Ptrady;
  Sw: boolean;
begin
  P := Direccion(G,X);
  if (P<>nil) then
  begin
    Q := G;
    while Q<>nil do {suprime todos los posibles arcos Q^.Vert-X}
    begin
      borrar_arco(G, Q^.Vert,X);
      Q := Q^.Sgte
    end;
    {Ahora elimina nodo de la lista directorio}
    if G = P then
      G := G^.Sgte
    else
      Anterior(G,P)^.Sgte := P^.Sgte;
    Libera(P^.Lady);
    dispose(P)
  end
end;

```

13.5. RECORRIDO DE UN GRAFO

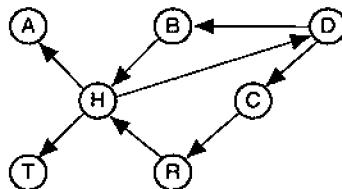
La operación de recorrer una estructura consiste en visitar (procesar) cada uno de los nodos a partir de uno dado. Así, para recorrer un árbol se parte del nodo raíz y según el orden se visitan todos los nodos. De igual forma, recorrer un grafo consiste en visitar todos los vértices alcanzables a partir de uno dado. Hay dos formas: recorrido en profundidad y recorrido en anchura.

13.5.1. Recorrido en anchura

Este método comienza visitando el vértice de partida A, para a continuación visitar los adyacentes que no estuvieran ya visitados. Así sucesivamente con los adyacentes. Este método utiliza una cola como estructura auxiliar en la que se mantienen los vértices que se vayan a procesar posteriormente. La estrategia la expresamos de forma más concisa en estos pasos:

1. Visitar el vértice de partida A.
2. Meter en la cola el vértice de partida y marcarle como procesado.
3. Repetir los pasos 4 y 5 hasta que la cola esté vacía.
4. Retirar el nodo frente (W) de la cola, visitar W.
5. Meter en la cola todos los vértices adyacentes a W que no estén procesados y marcarlos como procesados.

En el grafo dirigido de la figura queremos hacer recorrido a partir del vértice D.



El seguimiento de la estrategia indicada: inicialmente añadir D a la cola, retirar el elemento frente de la cola D, meter en cola los vértices adyacentes a D no procesados. La siguiente figura muestra el estado de la cola en cada paso, así como la sucesión de vértices recorridos.

Vértices del recorrido desde D

{D}

{D, B}

{D, B, C}

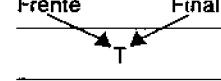
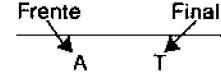
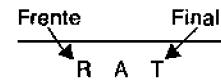
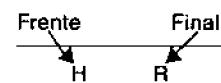
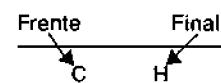
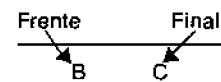
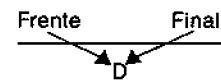
{D, B, C, H}

{D, B, C, H, R}

{D, B, C, H, R, A}

{D, B, C, H, R, A, T}

Cola



Cola vacía

En este ejemplo el recorrido del grafo en anchura a partir de D es el conjunto de todos los nodos. En definitiva, todos los vértices del grafo son alcanzables desde el vértice D.

13.5.2. Realización del recorrido en anchura

Para codificar el recorrido en anchura tenemos que utilizar las operaciones del TAD cola. Para marcar los vértices ya visitados se pueden seguir varias alternativas, elegimos la de definir una lista con todos los vértices del grafo y un campo que indique si está o no procesado.

Esta realización supone que el grafo está representado mediante listas de adyacencia. Los tipos de datos para la cola y la lista de visitados.

```
type
  Tipoelemen = Vertice;
  Ptrnodoq = ^Nodoq;
  Nodoq = record
    Info: Tipoelemen;
    Sgte: Ptrnodoq
  end;
  Cola = record
    Frente,
    Final: Ptrnodoq
  end;
  Ptr_Lv = ^Nodo_Lv;
  Nodo_Lv = record
    V: PtrDir;
    Vsdo: boolean;
    Sgte: Ptr_lv
  end;
```

CODIFICACIÓN

```
procedure Recorrido_Anchura(G,W: PtrDir);
var
  Lv,Av: Ptr_lv;
  Qe: Cola;
  N: PtrDir;
  L: PtrAdy;
procedure Lista_Visitados(var Lv: Ptr_Lv; G: PtrDir);
var P: PtrDir;
function CreaV(Q: PtrDir): Ptr_Lv;
var A: Ptr_Lv;
begin
  new(A);
  A^.V := Q; A^.Vsdo := false;
  CrearV := A
end;
begin
  if G<>nil then
  begin
    Lv := CreaV(G); P := Lv; G := G^.Sgte;
```

```

while G<> nil do
begin
  P^.Sgte := CreaV(G);
  P := P^.Sgte; G := G^.Sgte
end
end;
function Direccion(L: Ptr_Lv; W: PtrDir): Ptr_Lv;
var
  P,D: Ptr_Lv;
begin
  P := nil; D := L;
  while(P=nil)and(D<>nil) do
  if D^.V=W then
    P := D
  else D:= D^.Sgte;
  Direccion := P
end;
begin {Recorrido en anchura}
  Lista_Visitados(Lv,G);
  Qcrear(Qe);
  {Recorrido a partir del vértice W}
  Qponer(W,Qe); Direccion(Lv,W)^.Vsdo := true;
  repeat
    Quitar(W,Qe);
    {Vértice W visitado}
    L := W^.Lady;
    {Mete en cola los adyacentes}
    while L<>nil do
    begin
      N := L^.V; Av := Direccion(Lv,N);
      if not Av^.Vsdo then
      begin
        Qponer(N,Qe);
        Av^.Vsdo := true
      end;
      L := L^.Sgte
    end
    until Qvacia(Qe)
end;

```

En el caso de querer visitar todos los vértices del grafo, una vez que ha terminado el recorrido a partir de uno dado (W) hay que buscar si queda algún vértice sin visitar en cuyo caso se vuelve a recorrer a partir de él, así sucesivamente hasta que todos los vértices estén procesados.

13.5.3. Recorrido en profundidad

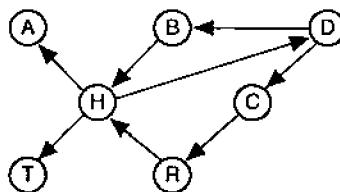
El recorrido en profundidad persigue el mismo objetivo que el recorrido en anchura: visitar todos los vértices del grafo alcanzables desde un vértice dado. La búsqueda en profundidad empieza por un vértice V del grafo G; V se marca como visitado. Después se recorre en profundidad cada vértice adyacente a V no visitado; así hasta que no haya más vértices adyacentes no visitados. Esta técnica 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.

La definición recursiva del recorrido en profundidad ya nos indica que tenemos que utilizar una pila como estructura para guardar los vértices adyacentes no visitados a uno dado. De igual forma que en el recorrido en anchura, hacemos uso de una lista de vértices para controlar los ya visitados.

En la siguiente figura desarrollamos el recorrido en profundidad para el mismo grafo y mismo vértice de partida (D) que en el recorrido en anchura.

13.5.4. Recorrido en profundidad de un grafo



Vértices del recorrido desde D

- D
- {D}
- {D, C}
- {D, C, R}
- {D, C, R, H}
- {D, C, H, R, T}
- {D, C, R, H, T, A}
- {D, C, R, H, T, A, B}

Pila

[D]	← Cima
[B C]	← Cima
[B R]	← Cima
[B H]	← Cima
[B A T]	← Cima
[B A]	← Cima
[B]	← Cima
[]	Pila Vacía

13.5.5. Realización del recorrido en profundidad

La codificación del recorrido en profundidad exige la utilización de operaciones del TAD pila. Para marcar los vértices ya visitados utilizamos la misma lista de vértices del grafo que en el recorrido de anchura.

Suponemos el grafo representado mediante listas de adyacencia. Los tipos de datos para la pila y la lista de visitados.

```

type
  Tipoelemen = Vertice;
  Ptrnодоп = ^Nодоп;
  Nодоп = record
    Info: Tipoelemen;
    Sгte: Ptrnодоq
  end;
  Ptr_Lv = ^Nodo_Lv; {Mismo tipo que en el recorrido en anchura}

```

CODIFICACIÓN

```

procedure Recorrido_Profundidad(G,W: PtrDir);
var
  Lv, Av: Ptr_Lv;
  Pila: Ptrnодоп;
  N: PtrDir;
  L: PtrAdy;
  {Código de procedimientos y funciones auxiliares, igual que
  recorrido en anchura}
begin
  Lista_Visitados(Lv,G);
  Pilavacia(Pila);
  {Recorrido en profundidad a partir del vértice W}
  Meter(W,Pila); Direccion(Lv,W)^.Vsdo := true;
  repeat
    Sacar(W,Pila);
    {Vertice W visitado}
    L := W^.Lady;
    {Mete en pila los vértices adyacentes}
    while L<>nil do
      begin
        N := L^.V; Av := Direccion(Lv,N);
        if not Av^.Vsdo then
          begin
            Meter(N,Pila);
            Av^.Vsdo := true
          end;
        L := L^.Sгte
      end
    until Esvacia(Pila)
end;

```

Realización recursiva

El recorrido en profundidad se implementa de manera más sencilla utilizando la técnica recursiva, se adapta íntegramente a la definición recursiva del recorrido. En este caso la lista de vértices visitados se inicializa a false «fuera» del procedimiento, antes de la llamada a Profundidad.

```

procedure Profundidad(W: PtrDir; var Lv: Ptr_Lv);
var
  L: PtrAdy;
begin

```

```

Direccion(Lv,W)^.Vsdo := true; {Lv es la lista de nodos visitados}
L := W^.Lady;
while L<>nil do
begin
  if not Direccion(L^.V)^.Vsdo then {si no visitado}
    Profundidad(L^.V,Lv);
  L := L^.Sgte
end
end;

```

13.6. 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 se puede determinar todas las componentes conexas del mismo.

En un grafo dirigido podemos distinguir entre grafo dirigido conexo y grafo fuertemente conexo. Un grafo dirigido es conexo si para cada par de vértices existe una cadena que los une. Y un grafo dirigido es fuertemente conexo si para cada par de vértices existe un camino que los une. El concepto de cadena se utiliza más adelante en el estudio del flujo máximo en una red.

Un algoritmo para determinar las componentes conexas de un grafo G no dirigido.

1. Realizar un recorrido del grafo a partir de cualquier vértice w . Los vértices visitados son guardados en el conjunto W .
2. Si el conjunto W es el conjunto de todos los vértices del grafo, entonces el grafo es conexo.
3. Si el grafo no es conexo, W es una componente conexa.
4. Se toma un vértice no visitado, z , y se realiza de nuevo el recorrido del grafo a partir de z . Los vértices visitados W forman otra componente conexa.
5. El algoritmo termina cuando todos los vértices han sido visitados.

CODIFICACIÓN

Se parte de un grafo G representado mediante listas de adyacencia, los vértices según son visitados son almacenados en un vector para así comparar los vértices visitados con el total de vértices y determinar si el grafo es conexo. Es necesario seguir utilizando la lista de vértices visitados.

Los tipos de datos que se incorporan a los ya definidos para el recorrido:

```

const
  M = 100; {Máximo de vértices}
type
  Conjunto = record
    Vc: array[1..M] of Vertice;
    N: 0..M
  end;

```

El recorrido del grafo no dirigido para obtener una componente conexa:

```

procedure Conexa(G,Z: PtrDir; var Lv: Ptr_Lv; var W: Conjunto);
var
  Av: Ptr_Lv;
  Pila: Ptrnодоп;
  N: PtrDir;
  L: PtrAdy;
  {Código de procedimientos y funciones auxiliares, igual que
  recorrido en anchura}
begin
  Pilavacia(Pila);
  Meter(Z,Pila); Direccion(Lv,Z)^.Vsdo := true;
  repeat
    Sacar(Z,Pila);
    with W do
    begin
      N := N+1;
      Vc[N] := Z^.Vert
    end;
    L := Z^.Lady;
    while L<> nil do
    begin
      N := L^.V; Av := Direccion(Lv,N);
      if not Av^.Vsdo then
      begin
        Meter(N,Pila);
        Av^.Vsdo := true
      end;
      L := L^.Sgte
    end
    until Esvacia(Pila)
  end;

procedure Componentes_Conexas(G: Grafo);
var
  T,W: Conjunto;
  Z: PtrDir;
  J: integer;
function Dir_Novis(L: Ptr_Lv): Ptr_Lv;
var
  D: Ptr_Lv;
  Sw: boolean;
begin
  Sw := false; D := L;
  while (D<>nil) and not Sw do
  begin
    Sw := not D^.Vsdo;
    if not Sw then D := D^.Sgte
  end;
  Dir_Novis := D
end;
begin
  {En T guardamos el conjunto de todos los vértices}
  with T do
  begin

```

```

N := 0; Z := G;
while Z<>nil do
begin
  N := N+1;
  Vc[N] := Z^.Vert
  Z := Z^.Sgte
end;
end;
Lista_Visitados(Lv,G); {Todos los vértices se ponen a false su
                        campo visitado}
{Comienza a partir de primer vértice no visitado}
Z := Dir_Novis(Lv);
while Z<>nil do
begin
  W.N := 0;
  Conexa(G,Z,Lv,W);
  if W.N = T.N then
    writeln('Tenemos un grafo no dirigido Conexo')
  else begin
    write('Componente Conexa:');
    for J := 1 to W.N do
      write(W.Vc[J], '');
    writeln
  end;
  Z := Dir_Novis(Lv)
end
end;

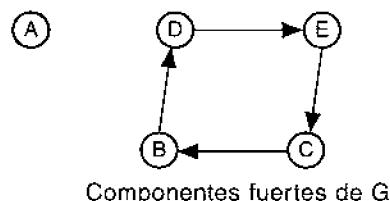
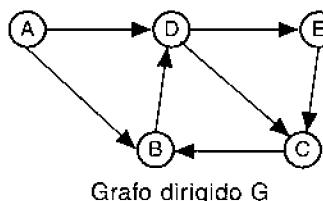
```

13.7. COMPONENTES FUERTEMENTE CONEXAS DE UN GRAFO DIRIGIDO

Un grafo dirigido fuertemente conexo es aquel en el cual existe un camino entre cualquier pareja de vértices del grafo. De no ser fuertemente conexo se pueden determinar componentes fuertemente conexas del grafo.

EJEMPLO

La figura siguiente muestra un grafo dirigido con sus componentes fuertes.



Pueden seguirse diversas estrategias para encontrar si un grafo G es fuertemente conexo o en su caso determinar las componentes conexas. El siguiente algoritmo utiliza el recorrido en profundidad.

1. Obtener el conjunto de descendientes de un vértice de partida v , $D(v)$, incluido el propio vértice v . Así tenemos todos los vértices desde los que hay un camino que comenzando en v llega a él.
2. Obtener el conjunto de ascendientes de v , $A(v)$, incluido el propio vértice v . Estos vértices son aquellos en los que comienza un camino que llega a v .
3. Los vértices comunes que tiene D y A , es decir, $D(v) \cap A(v)$, es el conjunto de vértices de la componente fuertemente conexa a la que pertenece v . Si este conjunto es igual al conjunto de vértices de G , entonces el grafo es fuertemente conexo.
4. Si no es un grafo fuertemente conexo se selecciona un vértice cualquiera w que no esté en ninguna componente fuerte de las encontradas [$w \notin D(v) \cap A(v)$] y se procede de la misma manera, es decir, se repite a partir de 1 hasta obtener todas las componentes fuertes del grafo.

Para el paso 1 se realiza un recorrido en profundidad del grafo G a partir del vértice v . Los vértices que son visitados se guardan en el conjunto D .

Para el paso 2 hay que proceder en primer lugar a construir otro grafo dirigido G_i que sea el resultante de invertir las direcciones de todos los arcos de G . Y a continuación proceder como en el paso 1.

En cuanto a los tipos de datos, el tipo vértice se supone entero de tal forma que el ordinal del vértice coincide con el vértice. Además se supone que el máximo de vértices es 100. El grafo está representado mediante listas de adyacencia. Se define el vector *boolean* visitado cada elemento del vector tiene verdadero (*true*) si en el recorrido el vértice con el que se corresponde ha sido visitado. El recorrido está codificado según su definición, de manera recursiva, de tal forma que los vértices alcanzados desde un vértice de partida se guardan como sus descendientes; la lista de vértices visitados se implementa con un vector global. Repitiendo el recorrido a partir del mismo vértice, pero con el grafo inverso (cambiando el sentido de los arcos), los vértices alcanzados a partir del vértice de partida son sus ascendientes.

CODIFICACIÓN

El programa que se presenta tiene dos partes diferenciadas. La primera lee los arcos del grafo, crea las listas de adyacencia del grafo G y a la vez las listas de adyacencia del grafo inverso $InvG$. La segunda encuentra las componentes conexas y las escribe.

```
program ComponentesFuertes (input, output);
const n = 100;
type
  Vertice = 1..n;
  PtrDir = ^NodoDir;
  PtrAdy = ^NodoLy;
  NodoDir = record
    Vert: Vertice; {Identificador del vértice}
    Lady: PtrAdy;
    Sgte: PtrDir
```

```

end;
NodoLy = record
  PtrV: PtrDir;
  Sgte: PtrAdy
end;
Visitados = array[Vertice] of boolean;
var
  G, InvG: PtrDir;
  Vstdos,Fuertes,Descendentes,Ascendentes,Proc,Nulo: Visitados;
  NumVerts, I,J: Vertice;

procedure ListaDirectorio(var G: PtrDir; NumVerts: integer);
var J: integer;
begin
  G := nil;
  for J := 1 to NumVerts do
    Nuevo_vertice(G,J); {Operación definida en TAD grafo}
end;

procedure Arcos(var G,Gi: PtrDir; NumVerts: vertice);
type
  Arco = record
    x,y: Vertice;
  end;
var
  U: Arco;
begin
  write ('Introduce arcos como par ordenado de vertices(fin
          eof):');
  while not eof do
  begin
    read(U.x,U.y); {Unión operación del TDA grafo}
    Union(G,U.x,U.y);Union(Gi,U.y,U.x);
  end
end;

procedure Profundidad(G: PtrDir; V: Vertice; var Ac: Visitados);
var
  L: PtrAdy;
begin
  Vstdos[V] := true; Ac[V] := true;
  L := G^.Lady;
  while L <>nil do
  begin
    if not Vstdos[L^.PtrV^.Vert] then
      Profundidad(L^.PtrV,L^.PtrV^.Vert,Ac);
    L := L^.Sgte
  end
end;
function Esconexo(Vs: Visitados; Nv: Vertice): boolean;
var
  J: integer; S: boolean;
begin
  S := true;
  for J := 1 to Nv do
    S := S and Vs[J];

```

```

Esconexo := S
end;
procedure Interseccion(D,A: Visitados; Nv: Vertice;
                        var F: Visitados);
var J: Vertice;
begin
  for J := 1 to Nv do
    F[J] := F[J] or (D[J] and A[J])
end;

begin {programa principal}
  write ('Numero de Vertices:'); readln(NumVerts);
  ListaDirectorio(G,NumVerts);ListaDirectorio(InvG,NumVerts);
  Arcos(G,InvG,NumVerts);
  for I := 1 to NumVerts do
    Nulo[I] := false;
  Fuertes := Nulo; Proc := Nulo;
  for I := 1 to NumVerts do
  begin
    if not Proc[I] then {Vértice de partida cualquiera que no
                          está en ninguna componente ya obtenida.}
    begin
      Vstdos := Nulo; Descendentes := Nulo; Ascendentes := Nulo;
      Profundidad(G,I,Descendentes);
      Vstdos := Nulo;
      Profundidad(InvG,I,Ascendentes);
      Interseccion(Descendentes,Ascendentes,NumVerts,Fuertes);
      if Esconexo (Fuertes, NumVerts) then
      begin
        writeln ('El grafo es fuertemente conexo');
        Proc := Fuertes
      end
      else begin
        write ('Componente fuerte:');
        for J := 1 to NumVerts do
          if Fuertes[J] then write(J,' ');
        writeln;
        for J := 1 to NumVerts do
          Proc[J]:= Proc[J] or Fuertes[J];
        Fuertes := Nulo; {Prepara para buscar otra componente fuerte}
      end;
    end;
    G := G^.Sgte; InvG := InvG^.Sgte
  end
end.

```

13.8. MATRIZ DE CAMINOS. CIERRE TRANSITIVO

Conceptos importantes a considerar en la construcción de grafos son: *Camino entre par de vértices, la matriz de caminos y el cierre transitivo.*

Camino entre par de vértices

Sea G un grafo de n vértices y A su matriz de adyacencia de tipo lógico. Obsérvese la expresión lógica: $A[i, k] \text{ and } A[k, j]$. Esta expresión será cierta si y sólo si los va-

lores de ambos operandos lo son, lo cual implica que hay un arco desde el vértice i al vértice k y otro desde el vértice k al j . También podemos decir que la expresión sea cierta implica la existencia de un camino de longitud 2 desde el vértice i al j .

Ahora consideremos la expresión:

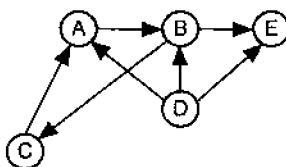
$(A[i,1] \text{ and } A[1,j]) \text{ or } (A[i,2] \text{ and } A[2,j]) \text{ or } \dots \text{ or } (A[i,\text{Numverts}] \text{ and } A[\text{Numverts},j])$

Si esta expresión es cierta implica que hay al menos un camino de longitud 2 desde el vértice i al vértice j que pase a través del vértice 1, o a través del 2, o a través del vértice Numverts .

Recordando el producto matricial $A \times A$ observamos que la expresión anterior si cambiamos and por producto y or por suma representa el elemento A_{ij} de la matriz A^2 . Según esto los elementos (A_{ij}) de la matriz A^2 son verdaderos si existe un camino de longitud 2 desde el vértice i al vértice $j \forall i, j = 1..n$.

De igual forma el producto matricial $A^2 \times A = A^3$ nos permite determinar si existe un camino de longitud 3 entre cualquier par de vértices del grafo. En general, para determinar la existencia de camino de longitud m entre cualquier par de vértices se forma el producto boolean de los caminos de la matriz A^{m-1} con la matriz adyacente A .

En la figura siguiente tenemos un grafo dirigido.



La matriz de adyacencia

$$A = \begin{pmatrix} F & T & F & F & F \\ F & F & T & F & T \\ T & F & F & F & F \\ T & T & F & F & T \\ F & F & F & F & F \end{pmatrix}$$

El producto boolean $A \times A$

$$A^2 = \begin{pmatrix} F & F & T & F & T \\ T & F & F & F & F \\ F & T & F & F & F \\ F & T & T & F & T \\ F & F & F & F & F \end{pmatrix}$$

Así, A_{15} es verdadero (*true*) ya que hay un camino de longitud 2 desde $A-E$ ($A \rightarrow B \rightarrow E$).

El producto boolean $A^2 \times A$

$$A^3 = \begin{pmatrix} T & F & F & F & F \\ F & T & F & F & F \\ F & F & T & F & T \\ T & F & T & F & T \\ F & F & F & F & F \end{pmatrix}$$

Así, A_{41} es verdadero (*true*) ya que hay un camino de longitud 3 desde $D \rightarrow A$ ($D \rightarrow B \rightarrow C \rightarrow A$).

Con el producto boolean se ha obtenido si hay camino de longitud m entre un par de vértices. En caso de que la matriz de adyacencia está representada mediante 0, 1 podemos obtener no sólo si hay camino de longitud m sino además el número de caminos.

Sea $G = (V, A)$ un grafo representado por la matriz de adyacencia A tal que el elemento A_{ij} es 1 si hay un arco desde el vértice i al j , también nos da el número de caminos del longitud 1 desde i a j . Haciendo un razonamiento similar, la expresión:

$$(A[i,j] * A[1,j]) + (A[i,2] * A[2,j]) + \dots + (A[i,NumVerts] * A[NumVerts,j])$$

nos da todos los posibles caminos de longitud 2 desde el vértice i al j , además, recordando el producto matricial $A \times A$, representa el elemento A_{ij} de la matriz A^2 . Podemos generalizar, la forma de obtener el número de caminos de longitud k entre cualquier par de vértices del grafo es obtener el producto matricial $A^2, A^3 \dots A^k$, entonces el elemento $A_k(i,j)$ nos da el número de caminos de longitud k desde el vértice i hasta el vértice j .

EJEMPLO 13.9

Consideremos el grafo anterior cuya matriz de adyacencia es

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

El producto $A \times A$

$$A^2 = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Así, hay un camino de longitud 2 desde el vértice D al vértice E.

El producto $A^2 \times A$

$$A^3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Existe un camino de longitud 3 desde el vértice C al vértice D.

Procedimiento para obtener el número de caminos

A continuación se escribe el procedimiento para obtener la matriz A^k que nos permite determinar el número de caminos de longitud k entre dos vértices. El procedimiento tiene como entrada la matriz de adyacencia, la longitud del camino K y el número de vértices Nvs . La salida del procedimiento es la matriz A^k .

```

procedure Producto_A(A: MatAdcia; K: integer; Nvs: Vertice;
                     var Ak: MatAdcia);
var
  I: integer;
procedure Prod(A,B: MatAdcia; var N: MatAdcia);
var f,c,k,S: integer;
begin
  for f := 1 to Nvs do
    for c := 1 to Nvs do
      begin
        S := 0;
        for K := 1 to Nvs do
          S := S+A[f,k]*B[c,k];
        N[f,c] := S
      end
    end;
  begin
    Ak := A;
    if K>=2 then
      begin
        Prod(A,A,Ak);
        for I := 3 to K do
          Prod(Ak,A,Ak)
      end
    end;
  end;

```

Matriz de caminos

Sea G un grafo con n vértices. La matriz de caminos de G es la matriz $n \times n P$ definida:

$$P_{ij} = \begin{cases} 1 & \text{si hay un camino desde } V_i \text{ a } V_j, \\ 0 & \text{si no hay camino desde } V_i \text{ a } V_j \end{cases}$$

El camino de V_i a V_j será simple si $V_i \neq V_j$, o bien un ciclo cuando los extremos sean el mismo vértice $V_i = V_j$. Al ser el grafo G de n vértices, un camino simple ha de tener longitud $n-1$ o menor, y un ciclo ha de tener longitud n o menor. Según esto podemos encontrar la siguiente relación entre la matriz de caminos P , la matriz de adyacencia A y las sucesivas potencias de A .

Dada la matriz $B_n = A + A_2 + A_3 + \dots + A_n$, la matriz de caminos $P = (p_{ij})$ es tal que un elemento $p_{ij} = 1$ si y sólo si $B_n(i,j) \geq 1$ y en otro caso $p_{ij} = 0$.

Una vez que tenemos la matriz de camino de un grafo G dirigido podemos determinar de manera más fácil si el grafo es fuertemente conexo. Recordar que para que G sea fuertemente conexo se ha de cumplir que para todo par de vértices V_i, V_j ha de existir un camino de V_i a V_j y un camino de V_j a V_i . Por tanto, para que G sea fuertemente conexo la matriz de caminos P ha de tener todos los elementos a 1.

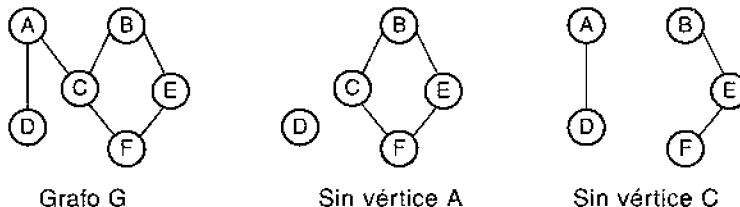
Cierre transitivo

El cierre 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 . Según esta definición, un grafo es fuertemente conexo si y sólo si su cierre transitivo es un grafo completo.

13.9. 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, la componente conexa en que está el vértice se divide en dos o más componentes. Por ejemplo, en la figura tenemos un grafo que tiene dos puntos de articulación: el vértice A y el vértice C . Al eliminar el vértice C el grafo que es conexo se convierte en dos componentes conexos: $\{B,E,F\}$ y $\{A,D\}$; si se elimina el vértice A , el grafo se divide en estos dos componentes conexos: $\{C,B,E,F\}$ y $\{D\}$. Sin embargo, al suprimir cualquier otro vértice del grafo, el componente conexo no se divide.

EJEMPLO 13.10



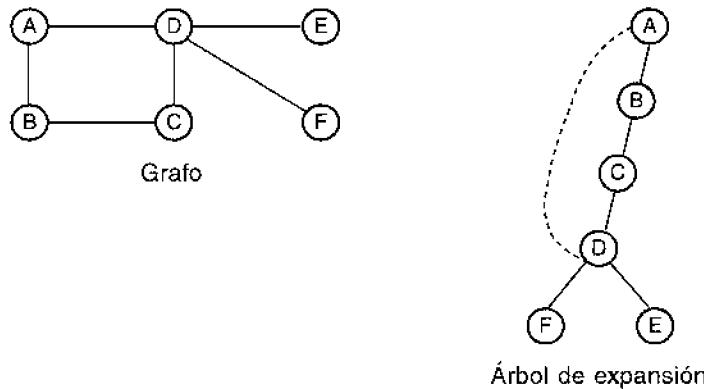
Los grafos tienen propiedades relativas a los puntos de articulación. Un grafo sin puntos de articulación se dice que es un *grafo biconexo*. De no ser el grafo biconexo, es interesante encontrar componentes biconexos. Un grafo tiene *conectividad k* si la eliminación

de $k-1$ vértices cualesquiera del grafo no lo divide en componentes conexas (no lo desconecta). Cuanto mayor sea la conectividad de un grafo (una red, por ejemplo) tanto mayor probabilidad tendrá de mantener la estructura ante el fallo (eliminación) de alguno de sus vértices.

Algoritmo de búsqueda de puntos de articulación

El algoritmo utiliza los recorridos en profundidad de un grafo para encontrar todos los puntos de articulación.

El recorrido recursivo en profundidad del grafo a partir de un vértice A puede representarse mediante un *árbol de expansión*. La raíz del árbol es el vértice de partida A. Cada arco del grafo estará como una arista en el árbol. Si en el proceso recursivo del recorrido tenemos que al pasar por los vértices adyacentes de v , arcos (v, u) , el vértice u no está visitado, entonces (v, u) es una arista del árbol; si el vértice u se ha visitado ya, entonces (v, u) se dice que es una arista hacia atrás (realmente no es una arista e incluso se dibuja con línea discontinua). La figura nos muestra un grafo y el árbol del recorrido en profundidad.



Numerando los nodos del árbol en un recorrido en preorden obtenemos 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 la llamamos $Num(v)$.
 2. Para cada vértice v del árbol del recorrido en profundidad determinamos el vértice de numeración más baja [en este caso llamado $Bajo(v)$] que es alcanzable desde v a través de 0 o más aristas del árbol y como mucho una arista hacia atrás (de retroceso).
- La definición de $Bajo(v)$ se expresa matemáticamente como el mínimo de los siguientes tres valores:

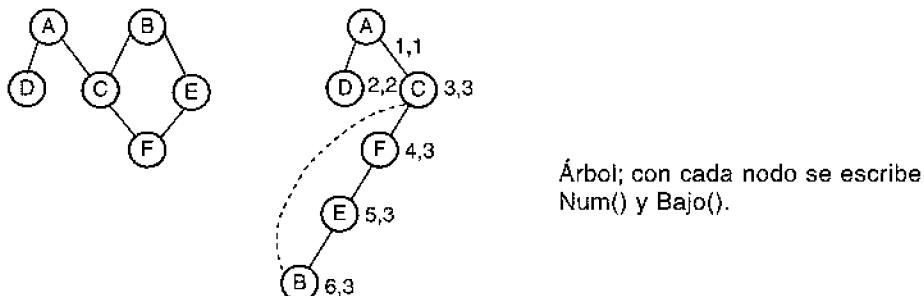
- a) $Num(v)$.
- b) El menor valor de $Num(w)$ para los vértices w de las aristas hacia atrás (v,w) del árbol.
- c) El menor valor de $Bajo(w)$ para los vértices w de las aristas (v,w) del árbol.

En la figura del ejemplo 13.11 se encuentran los valores $Num(v)$, $Bajo(v)$ de todos los vértices.

3. Una vez que tenemos los valores $Num(v)$, $Bajo(v)$, se determinan los puntos de articulación.
 - 3.1. La raíz del árbol (vértice de partida) es punto de articulación si y sólo si tiene dos o más hijos.
 - 3.2. Cualquier otro vértice w es punto de articulación si y sólo si w tiene al menos un hijo u tal que $Bajo(u) \geq Num(w)$.

EJEMPLO 13.11

En la figura siguiente mostramos un grafo conectado (conexo) y el árbol de expansión.



Tomando el vértice B, el valor $Bajo(B) = 3$ porque el mínimo [$Num(B)$, $Num(C)$] es 3; observar que B tiene un arco de retroceso B-C.

El procedimiento para asignar los valores $Num(v)$ a cada vértice del grafo consiste en recorrer en profundidad el grafo, cada llamada recursiva incrementa el contador de llamadas que es el valor de $Num(v)$ para el vértice actual. A la vez en el array Arista se guarda el vértice w con el que el vértice actual forma una arista del árbol de expansión. Para facilitar la comprensión, el tipo vértice es del rango de $1..n$, siendo n el número de vértices, y para marcar los vértices visitados se utiliza un array lineal.

```

const N = 8;
type
  Vertice = 1..n;
  Visitados = array[Vertice] of boolean;
  Numcion = array[Vertice] of Vertice;

procedure Val_Num(G:PtrDir, V:Vertice; var Vstdos:Visitados;
                  var C:integer; var Num:Numcion; var Arista:Numcion);
var
  L: PtrAdy;
  W: PtrDir;

```

```

begin
  Vstdos[V] := true;
  C := C+1; Num[V] := C;
  L := G^.Lady; {G tiene la dirección del vértice V}
  while L <> nil do
    begin
      W := L^.PtrV;
      if not Vstdos[W^.Vert] then
        begin
          Arista[W^.Vert] := V;
          Val_Num(W,W^.Vert,Vstdos,C,Num,Arista)
        end;
      L := L^.Sgte
    end
end;

```

El procedimiento para calcular los valores llamados *Bajo(v)* se realiza con un recorrido en postorden de los vértices. Son necesarios los valores calculados en el procedimiento anterior de *Num* y *Arista*. Además se puede determinar los puntos de articulación de vértices que no son la raíz (vértice de partida) al conocerse el valor de *Bajo()*.

```

procedure Val_Bajo(G:PtrDir; V:Vertice; var Num:Numcion;
  var Vstdos:Visitados;var Arista:Numcion;var Bajo:Numcion);
var
  L: Ptrady;
  W: PtrDir;
begin
  Vstdos[V] := true;
  Bajo[V] := Num[V]; {valor inicial para cálculo del mismo}
  L := G^.Lady;
  while L <> nil do
    begin
      W := L^.PtrV;
      if not Vstdos[W^.Vert] then
        begin
          if Num[W^.Vert] > Num[V] then {arco del árbol}
            begin
              Val_Bajo(W,W^.Vert,Num,Vstdos,Arista,Bajo);
              {Calcula Bajo(w) tal que w hijo de V}
              if Bajo[W^.Vert] >= Num[V] then
                writeln (V,' es un punto de articulación.');
              Bajo[V] := Minimo(Bajo[V],Bajo[W^.Vert])
              {menor valor de Bajo(w)...regla c}
            end
          else if Arista[V] <> W^.Vert then {arista hacia atrás}
            begin
              Bajo[V]:= Minimo(Bajo[V],Num[W^.Vert]); {menor de Num(w)}
              writeln (V,' es un punto de articulación.');
            end
          end
        end
      L := L^.Sgte
    end
end;

```

Los procedimientos *Val_Num* y *Val_Bajo* se pueden combinar en un solo procedimiento que calcule a la vez *Num(v)* y *Bajo(v)*; y además nos muestre los puntos de articulación (excepto si lo es el vértice de partida).

```

procedure Ptos_Artc(G:PtrDir; V:Vertice; var Num:Numcion;
                     var C:integer;var Vstdos:Visitados;
                     var Arista:Numcion; var Bajo:Numcion);
var
  L: Ptrady;
  W: PtrDir;
begin
  Vstdos[V] := true; C := C+1; Num[V] := C;
  Bajo[V] := Num[V]; {Valor inicial para el cálculo del mínimo}
  L := G^.Lady;
  while L <> nil do
  begin
    W := L^.PtrV;
    if not Vstdos[W^.Vert] then
    begin
      Arista[W^.Vert] := V;
      Ptos_Artc(W,W^.Vert.Num,C,Vstdos,Arista,Bajo);
      if Bajo(W^.Vert) >= Num(V) then
        writeln(V,' es un punto de articulación.');
      Bajo[V] := Minimo(Bajo[V],Bajo[W^.Vert]); {menor valor de
                                                    Bajo(v)...regla c}
    end
    else if Arista[V]<> W^.Vert then {arista hacia atrás}
      Bajo[V] := Minimo(Bajo[V],Num[W^.Vert]); {menor de
                                                    Num(w)...regla b}
    L := L^.Sgte
  end
end;

```

RESUMEN

Un grafo G consta de dos conjuntos ($G = \{V, E\}$): un conjunto V de vértices o nodos y un conjunto E de aristas (parejas de vértices distintos) que conectan los vértices. Si las parejas 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 lista de adyacencia. Cada una tiene sus ventajas y desventajas relativas. La elección depende de las necesidades de la aplicación dada.

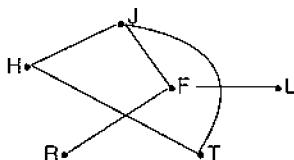
Existen diversos tipos de *grafos no dirigidos*. Dos vértices de un *grafo no dirigido* se llaman *adyacentes* si existe una arista desde el primero al segundo. Un *camino* es una secuencia de vértices distintos, cada uno adyacente al siguiente. Un *ciclo* es un camino que contenga al menos tres vértices, 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 puede ser en analogía con los árboles, *recorrido en profundidad* y *recorrido en anchura*. El *recorrido en profundidad* es aplicable a los *grafos dirigidos* y a los *no dirigidos* y es una generalización del recorrido preorden de un árbol. El *recorrido en anchura* es también aplicable a *grafos dirigidos* y *no dirigidos*, que generaliza el concepto de recorrido por niveles de un árbol.

EJERCICIOS

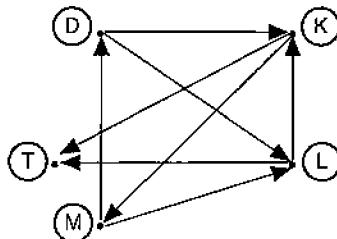
13.1. Sea el grafo no dirigido G de la figura



- Describir G formalmente en términos de su conjunto V de nodos y de su conjunto A de aristas.

- Encontrar el grado de cada nodo.

13.2. Sea el grafo dirigido de la figura

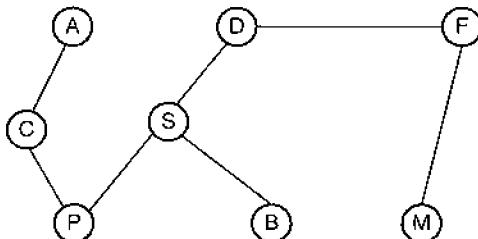


- Describir el grafo formalmente en términos de su conjunto V de nodos y de su conjunto A de aristas.

- Encontrar el grado de entrada y el grado de salida de cada vértice.

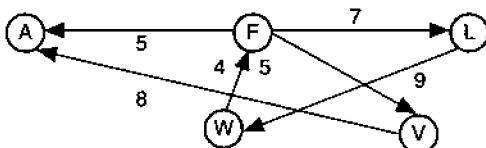
- Encontrar los caminos simples del vértice M al vértice T.

13.3. Sea el grafo G de la figura



- Encontrar todos los caminos simples del nodo A al nodo F.
- Encontrar el camino más corto de C a D.
- ¿Es un grafo conexo?

13.4. Dado el grafo valorado de la figura



- Encontrar la matriz de pesos del grafo.
- Representar el grafo mediante listas de adyacencia.

13.5. Un grafo G consta de los siguientes nodos $V = \{A, B, C, D, E\}$ y la matriz de adyacencia

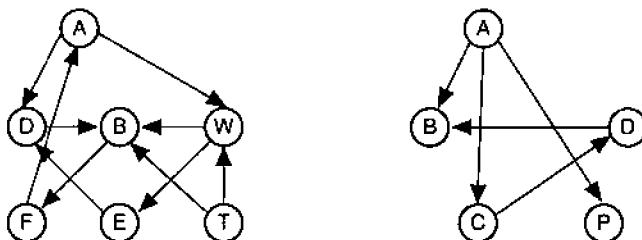
$$M = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

- Dibujar el grafo correspondiente.
- Representar el grafo mediante listas de adyacencia.

13.6. Dado el grafo G del ejercicio 13.5 realice el recorrido del grafo en profundidad partiendo del nodo C.

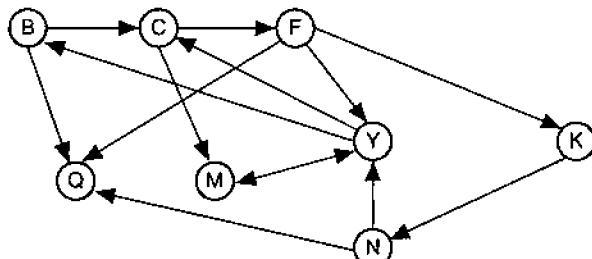
13.7. Dado el grafo G del ejercicio 13.5 realice el recorrido del grafo en anchura partiendo del nodo C.

13.8. Un grafo dirigido aciclico (gda) es un grafo dirigido sin ciclos. Dados los siguientes grafos:



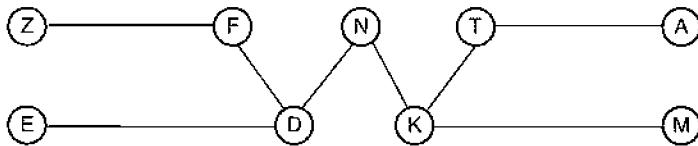
indicar si son gda,s. En caso de no serlo escribir los ciclos.

13.9. Dado el grafo de la figura



encontrar las componentes fuertemente conexas.

13.10. Dado el grafo G de la figura



- a) Escribe la matriz de adyacencia de G.
 b) Escribe la matriz de caminos de G.
- 13.11. Dibujar un grafo dirigido cuyos vértices son números enteros desde 3 hasta 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

- 13.1. Escribir un programa para dar de entrada los vértices y las aristas de un grafo dirigido. La representación del grafo será mediante matriz de adyacencia. El programa pedirá un vértice para realizar el recorrido en profundidad del grafo a partir de dicho vértice.
- 13.2. Un grafo valorado está formado por los vértices 4, 7, 14, 19, 21, 25. Las aristas siempre van de un vértice de mayor valor numérico a otro de menor valor, y el peso es el módulo del vértice origen y el vértice destino. Escribir un programa que represente el grafo en listas de adyacencia. Además, realizar un recorrido en anchura desde un vértice dado.
- 13.3. Queremos formar un grafo de manera aleatoria con los siguientes requisitos: consta de 10 vértices que son números enteros de 11 a 99. Dos vértices x,y están relacionados si $x+y$ es múltiplo de 3. Escribir un programa para representar el grafo descrito mediante una matriz de adyacencia. Determinar la matriz de caminos utilizando las potencias de la matriz de adyacencia.
- 13.4. Una región está formada por 12 comunidades. Se establece la relación de desplazamiento de personas en las primeras horas del día. Así, la comunidad A está relacionada con la comunidad B si desde A se desplazan n personas a B, de igual forma puede haber relación entre B y A si se desplazan m personas de B hasta A. Escribir un programa que represente el grafo descrito mediante listas de adyacencia. ¿Tiene fuentes y sumideros?
- 13.5. Dado el grafo descrito en el problema 13.4, escribir un programa para representarlo mediante listas enlazadas de tal forma que cada nodo de la lista directorio contenga dos listas: una que contiene los arcos que salen del nodo, y la otra que contiene los arcos que terminan en el nodo.
- 13.6. Dado un grafo dirigido en el que los vértices son números enteros positivos y el par (x,y) es un arco si $x-y$ es múltiplo de 3, escribir un programa para representar el grafo mediante listas de adyacencia de tal forma que cada lista sea circular. Una vez el grafo en memoria determinar el grado de entrada y el grado de salida de cada nodo.
- 13.7. Un grafo no dirigido está representado en memoria mediante listas de adyacencia y se desea saber si el grafo es cíclico o acíclico. Escribir un programa que represente el grafo y las funciones o procedimientos necesarios que determinen si el grafo es cíclico o acíclico. Además, escribir un procedimiento para listar los nodos que forman un ciclo (en caso de que lo haya).

13.8. Un algoritmo para detectar ciclos en un grafo dirigido consta de los siguientes pasos:

1. Obtener los sucesores de cada uno de los vértices.
2. Buscar un vértice sin sucesores y eliminarlo de los conjuntos de sucesores.
3. Repetir paso 2 siempre que haya algún vértice sin sucesores.
4. Si todos los vértices del grafo han sido eliminados, el grafo no tiene ciclos.

Realizar un programa en el que se represente un grafo dirigido mediante listas de adyacencia y detecte si el grafo dirigido tiene ciclos siguiendo el algoritmo descrito.

13.9. Se tiene un grafo representado mediante una matriz de adyacencia, escribir los procedimientos necesarios para representar dicho grafo mediante listas de adyacencia.

13.10. Un grafo en el que los vértices son regiones y los arcos tienen factor de peso está representado mediante una lista directorio que contiene a cada uno de los vértices y de las que sale una lista circular con los vértices adyacentes. Ahora se quiere representar el grafo mediante una matriz de pesos, de tal forma que si entre dos vértices no hay arco su posición en la matriz tiene 0, y si entre dos vértices hay arco su posición contiene el factor de peso que le corresponde. Escribir los procedimientos necesarios para que partiendo de la representación mediante listas se obtenga la representación mediante la matriz de pesos.

Algoritmos fundamentales con grafos

CONTENIDO

- 14.1. Ordenación topológica.
- 14.2. Matriz de caminos: algoritmo de Warshall.
- 14.3. Problema de los caminos más cortos con un solo origen: algoritmo de Dijkstra.
- 14.4. Problema de los caminos más cortos entre todos los pares de nodos: algoritmo de Floyd.
- 14.5. Problema del flujo de fluidos.
- 14.6. Árbol de expansión de coste mínimo.
- 14.7. Problema del árbol de expansión de coste mínimo: algoritmos de Prim y Kruskal.
- 14.8. Codificación del árbol de expansión de coste mínimo.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Existen numerosos problemas que se pueden formular en términos de grafos. La resolución de estos problemas requiere examinar todos los nodos o todas las aristas de un grafo; sin embargo, existen ocasiones en que la estructura del problema es tal que sólo se necesitan visitar algunos de los nodos o bien algunas de las aristas. Los algoritmos imponen implícitamente un orden en estos recorridos: visitar el nodo más próximo o las aristas más cortas, y así sucesivamente; otros algoritmos no requieren ningún orden concreto en el recorrido.

En base a 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. De igual modo se consideran algoritmos muy eficientes probado en situaciones críticas y de todo orden, tales como los algoritmos de Dijkstra, Warshall, Prim y Kruskal, entre otros.

14.1. ORDENACIÓN TOPOLOGICA

Un grafo G dirigido y sin ciclos se denomina un **gda** (grafo dirigido acíclico) o grafo acíclico. Los **gda** son útiles para la representación de estructuras sintácticas de expresiones aritméticas. Los grafos dirigidos acíclicos también son útiles para la representación de ordenaciones parciales. Una ordenación parcial R en un conjunto C es una relación binaria de precedencia tal que:

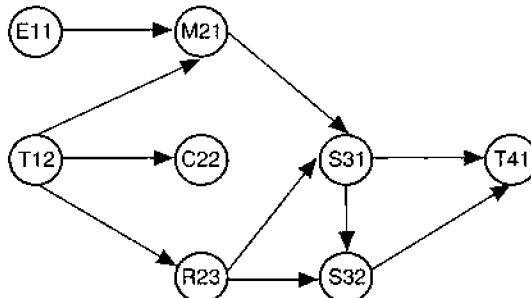
1. Para todo u perteneciente a C, $u \sim u$ es falso, por tanto, la relación R es no reflexiva.
2. Para todo $u, v, w \in C$, si $u R v$ y $v R w$ entonces $u R w$. La relación R es transitiva.

Tal relación R sobre C se llama ordenación parcial de C. Un ejemplo inmediato es la relación de inclusión en conjuntos.

Un grafo G sin ciclos se puede considerar un conjunto parcialmente ordenado. Para probar que un grafo es acíclico puede utilizarse la 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.

Una ordenación topológica T de un grafo acíclico es una ordenación lineal de los vértices, tal que si hay un camino del vértice v_i al vértice v_j , entonces v_j aparece después de v_i en la ordenación T.

EJEMPLO 14.1



La figura muestra un grafo acíclico que representa la estructura de prerequisitos de ocho cursos. Un arco cualquiera (r, s) significa que el curso r debe de terminarse antes de empezar el curso s . Por ejemplo, el curso M21 se puede empezar sólo cuando terminen los cursos E11 y T12; se dice que E11 y T12 son prerequisitos 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 obtenemos estas ordenaciones topológicas:

E11 - T12 - M21 - C22 - R23 - S31 - S32 - T41
 T12 - E11 - R23 - C22 - M21 - S31 - S32 - T41

Está claro que una ordenación topológica no es posible en un grafo con ciclos. ¿Qué implica el que haya un ciclo? Pues que si v, w pertenecen al ciclo, entonces v precede a w , y que w precede a v lo cual es evidentemente imposible.

Algoritmo para la obtención de una ordenación topológica T

Para que resulte más familiar la exposición del algoritmo, los vértices del grafo van a representar tareas. En primer lugar hay que buscar alguna tarea que no tenga predecesores, que no tenga prerequisitos, 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 (el prerequisito v ya se ha satisfecho). La estrategia se repite, se coge otro vértice w sin arcos incidentes, y así sucesivamente.

Precisando un poco más la estrategia, el que un vértice v no tenga arcos incidentes lo expresamos como que el *gradoentrada* (v) = 0. Eliminar los arcos que salen de v implica que el *gradoentrada* (w) de todos los vértices w adyacentes de v disminuyen en 1.

En una estructura de tipo cola se guardan los vértices con *gradoentrada* 0, el elemento frente de la cola v pasa a formar parte de la ordenación T . Se disminuye el *gradoentrada* de los adyacentes de v , y aquellos vértices cuyo *gradoentrada* se haga 0 se meten en la cola; así sucesivamente hasta que la cola esté vacía.

CODIFICACIÓN

La codificación se hace pensando en que el grafo tiene relativamente pocos arcos, lo que da origen a una matriz de adyacencia sparce (muchos ceros); por lo que la representación es con listas de adyacencia. Se utilizan las operaciones del TAD grafo.

```

const
  N = {número de vértices}
type
  (tipos para representar un grafo por listas de adyacencia)
  Vertice = 1..N;
  Ptrnodoq = ^Nodoq;
  Nodoq = record
    Info: Vertice;
    Sgte: Ptrnodoq
  end;
  Cola = record
    Frente,
    Final: Ptrnodoq
  end;

  (Función que devuelve el grado de entrada de un vértice)
  function GradoEntrada(V: Vertice; G: PtrDir): integer;
    (G es la dirección de lista directorio del grafo)
  var
    K: integer; W: Vertice;
  begin
    K:=0;
    while G <> nil do
      if W = V then K:=K+1;
      G:=G^.Sgte;
    end;
    result := K;
  end;

```

```

begin
  W:=G^.Ver;
  if Adyacente(G,W,V) then { hay arco W → V }
    K:=K+1;
  G:=G^.Sgte
end;
  GradoEntrada:=K
end;
procedure OrdenTopologica(G:PtrDir);
type
  Vector = array[1..N] of integer;
var
  Grd: Vector;
  V: Vertice;
  Pv,Pw:PtrDir;
  Lad:Ptrady;
  Q: Cola;
begin
  {Determina grado de entrada de cada vértice}
  for V:= 1 to N do
    Grd[V]:= GradoEntrada(V,G);
  Qcrear( Q); write('Ordenación topológica: ');
  {Mete en cola vértices con grado de entrada 0}
  for V:= 1 to N do
    if Grd[V] = 0 then
      Qponer(V,Q);
  while not Qvacia(Q) do
  begin
    Qsacar(V,Q);
    write(V:2,' ');
    Pv:= Direccion(G,V); Lad:=Pv^.Lady;
    {Decrementa grado entrada de V. adytes y si se pone a 0 es llevado
     a cola}
    while Lad <> nil do
    begin
      Pw:=Lad^.Pvert;
      V:=Pw^.Ver;
      Grd[V]:= Grd[V]-1;
      if Grd[V] = 0 then
        Qponer(V,Q)
    end
  end;
end;
end;

```

En cuanto al tiempo de ejecución de algoritmo es de $O(a + V)$ siendo a el número de arcos y V el de vértices. En caso de que la representación del grafo sea con la matriz de adyacencia el bucle mientras se sustituye por:

```

for K:= 1 to N do { Número de vértices }
begin
  Qsacar(V,Q);
  write(V:2,' ');
  for W:=1 to N do { busca adyacentes }
    if G.A[V,W]<> 0 then { W es adyacente de V}

```

```

begin
    Grd[W]:= Grd[W]-1;
    if Grd[W]=0 then
        Qponer(W,Q)
    end
end;

```

y en este caso el tiempo de ejecución es de $O(n^2)$.

14.2. MATRIZ DE CAMINOS: ALGORITMO DE WARSHALL

El método de calcular las sucesivas potencias de la matriz de adyacencia para así determinar la matriz de caminos es poco eficiente. Warshall propuso un algoritmo más eficiente para calcular la matriz de caminos (también llamado cierre transitivo).

Tenemos un grafo G de n vértices representado por su matriz de adyacencia A . Queremos encontrar la matriz de caminos P del grafo G . Para explicar el algoritmo se define una secuencia de matrices n -cuadradas de 0 y 1 $P_0, P_1, P_2, P_3 \dots, P_n$; 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 diferencia entre P_k y P_{k-1} radica en la incorporación del vértice k para poder formar el camino del vértice i y al j . Lo describimos con más detalle:

$$P_0[i,j] = \begin{cases} 1 & \text{si hay un arco de } i \text{ a } j. \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_1[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por ningún otro vértice} \\ & \text{a no ser por el vértice 1.} \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_2[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice a no} \\ & \text{ser por los que están comprendidos entre los vértices 1 .. 2.} \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_3[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice a no} \\ & \text{ser por los que están comprendidos entre los vértices 1 .. 3.} \\ 0 & \text{en otro caso.} \end{cases}$$

Observamos que en cada paso se incorpora un nuevo vértice, el que coincide con el índice de P , a los anteriores para poder formar camino.

$$P_k[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice a no} \\ & \text{ser por los que están comprendidos entre los vértices 1 .. } k. \\ 0 & \text{en otro caso.} \end{cases}$$

$$P_n[i,j] = \begin{cases} 1 & \text{si hay un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice a no} \\ & \text{ser por los que están comprendidos entre los vértices 1 .. } n; \text{ en definitiva en el camino puede estar cualquier vértice.} \\ 0 & \text{en otro caso.} \end{cases}$$

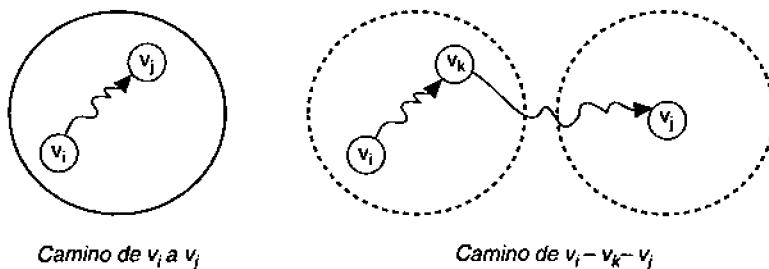
Según estas definiciones, P_0 es la matriz de adyacencia A del grafo. Y al ser un grafo de n vértices la matriz P_n es la matriz de caminos.

Warshall encontró la siguiente relación 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. Ya existe un camino simple de v_i a v_j del que pueden formar parte los vértices de índice 1 a $k-1$ (v_i a v_{k-1}); por tanto, el elemento de la matriz $P_{k-1}[i,j] = 1$.
2. Hay 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 a $k-1$; por tanto, esto equivale a cumplirse

$$(P_{k-1}[i,k] = 1) \text{ y } (P_{k-1}[k,j] = 1)$$

Estos dos casos pueden representarse en la figura siguiente:



En definitiva, Warshall encuentra una relación entre la matriz P_{k-1} y P_k que nos permite, partiendo de P_0 que es la matriz de adyacencia, encontrar P_n (matriz de caminos) por sucesivas iteraciones.

La relación para encontrar los elementos de P_k la podemos expresar como si fuera una operación lógica.

$$P_k[i,j] = P_{k-1}[i,j] \vee (P_{k-1}[i,k] \wedge P_{k-1}[k,j])$$

Como nuestra matriz es de 0..1, esta relación la debemos expresar

$$P_k[i,j] = \text{Mínimo} [1, P_{k-1}[i,j] + (P_{k-1}[i,k] * P_{k-1}[k,j])]$$

Se tiene un grafo G de n vértices, representado por su matriz de adyacencia. El algoritmo expresado en pseudocódigo encuentra la matriz de caminos P.

```
{Iniciarizar P}
desde i <- 1 hasta n hacer
    desde j <- 1 hasta n hacer
        P(i,j) <- A(i,j)
    fin desde
fin desde
```

{A continuación se obtienen las sucesivas matrices P_0 , P_1 , P_2 , P_3 , ..., P_n 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] <- Minimo(1, P[i,j]+(P[i,k] *P[k,j]))
        fin desde
    fin desde
fin desde
fin algoritmo

```

Se puede observar que la eficiencia del algoritmo es $O(n^3)$.

CODIFICACIÓN

El tipo de datos para representar la matriz de adyacencia es el ya utilizado

```

const
  Maxvert= 20;
type
  Indicevert = 1 .. Maxvert;
  MatAdcia= array[Indicevert,Indicevert] of 0..1
procedure Warshall(A:MatAdcia;n: Indicevert,var P:MatAdcia);
var i,j,k: Indicevert;
function Minimo(x,y:integer):integer;
begin
  if y < x then
    Minimo:= y
  else
    Minimo:= x
end;
begin
  for i:= 1 to n do
    for j:= 1 to n do
      P[i,j] := A[i,j];
  {
  }
  for k = 1 to n do
    for i:= 1 to n do
      for j:= 1 to n do
        P[i,j] := Minimo(1, P[i,j]+(P[i,k] *P[k,j]))
end;

```

14.3. PROBLEMA DE LOS CAMINOS MÁS CORTOS CON UN SOLO ORIGEN: ALGORITMO DE DIJKSTRA

Otro problema que se plantea con relativa frecuencia es determinar el camino más corto entre un par de vértices. Se parte de un grafo dirigido y valorado (*factor de peso*), por lo que cada arco (v_i, v_j) tiene asociado un coste $c_{ij} \geq 0$. De tal forma que el coste de un camino v_1, v_2, \dots, v_k es $\sum_{i=1}^{k-1} c_{i,i+1}$ y lo que se pretende es encontrar el camino de v_1 a v_k de coste mínimo.

Otro problema que se plantea es determinar el camino de menor longitud de v_1 a v_k , entendiendo como tal el que tenga menor número de arcos para ir de v_1 a v_k .

Un ejemplo del problema de los caminos más cortos es el determinar la ruta que en menor tiempo nos lleva desde un punto (nuestra casa, por ejemplo) a un conjunto de centros de la ciudad. Los vértices intermedios son paradas de Bus o Metro.

14.3.1. Algoritmo de la longitud del camino más corto

La Figura 14.1 nos muestra un grafo dirigido sin factor de peso, ya que se desea encontrar la longitud del camino más corto (menos arcos).

Se elige el vértice v_1 como el vértice desde el que se desea determinar la longitud de camino más corto al resto de los vértices. El camino más corto de v_1 a v_1 es obvio que es 0. A continuación buscamos todos los vértices cuya distancia de v_1 sea 1, éstos son los vértices adyacentes de v_1 : v_2 , v_4 ; en la Figura 14.1 están marcados con 1. Los vértices cuya longitud de camino es 2 son aquellos adyacentes de v_2 y de v_4 , éstos son v_3 , v_6 , v_5 ; en la figura están marcados con 2. Examinando los vértices adyacentes a los anteriores, v_3 , v_6 , v_5 que son v_5 , v_7 , v_8 , tenemos caminos de longitud 3; en la Figura 14.1 están marcados con 3.

La estrategia que se sigue es la de búsqueda en anchura. Para codificar este algoritmo se utiliza una tabla en la que cada elemento representa un vértice y tiene tres informaciones:

- *Alcanzado*: estará a verdadero (*true*) si se ha pasado en el recorrido por el vértice.
- *Distancia*: número de arcos desde el vértice inicial al vértice representado.
- *Último*: guarda el último vértice desde el que se alcanza el vértice representado, para así poder reconstruir el camino.

Inicialmente los valores de estos tres campos estarán a falso, ∞ (excepto vértice de partida, que estará a cero) y 0.

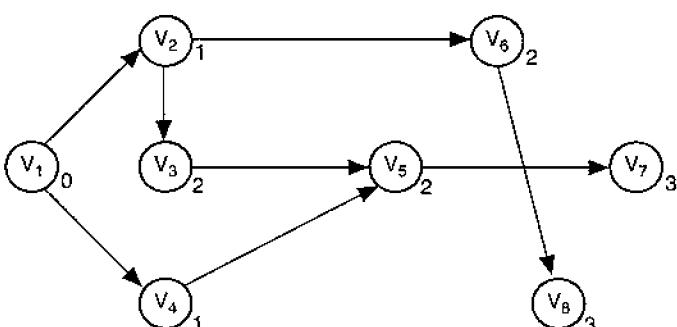


Figura 14.1. Grafo dirigido sin factor de peso.

También se hace uso de una cola. Como primer paso, la cola contiene el vértice de partida (v_1), que es el vértice de longitud de camino 0. A continuación se saca el elemento frente y añaden los vértices adyacentes a él, que son los de longitud de camino 1, y así sucesivamente. La cola (primero en entrar primero en salir) nos garantiza que no se procesan vértices de longitud de camino $m + 1$ hasta que no han sido procesados todos que tienen longitud m .

CODIFICACIÓN

```

type
  (tipos para representar los vértices y la matriz de adyacencia)
  Estado = record
    Alcanzado: boolean;
    Distancia: integer;
    Ultimo: Vertice;
  end;
  Tabla = array[1..NumVert] of Estado;

procedure Long_camino(var T: Tabla; S: Vertice; A: Matadcia);
var
  Dist : integer;
  V,W: Vertice;
  Q : Cola;
procedure Inicial(var T:Tabla;S:Vertice);
var v: integer;
begin
  for v:= 1 to NumVert do
    with T[v] do
  begin
    Alcanzado:= false;
    Distancia:= maxint; { representa el  $\infty$  }
    if v = S then Distancia:=0;
    Ultimo:=0
  end
end;
begin
  Inicial(T,S);
  Qcrear( Q);Qponer(S,Q);
  repeat
    Quitar(V,Q); { retira el vértice frente }
    T[V].Alcanzado:=true;
    { mete en la cola todos los vértices adyacentes de V no procesados }
    for W:=1 to NumVert do
      if A[V,W]<> 0 then { W es adyacente de V }
        if T[W].Distancia = maxint then {vértice no alcanzado}
          begin
            T[W].Distancia:= T[V].Distancia+1;
            T[W].Ultimo:=V;
            Qponer(W,Q)
          end
    until Qvacia(Q)
end;

```

En el procedimiento podemos observar que el campo Alcanzado realmente no es necesario. Si algunos vértices no son alcanzables desde el vértice de partida el campo Distancia queda con el valor maxint; además, una vez que un vértice es procesado no vuelve a entrar en la cola. Por consiguiente, podríamos suprimir el campo Alcanzado aunque su mantenimiento nos permite un más fácil seguimiento del algoritmo y posterior proceso de la tabla T.

Examinando el algoritmo podemos estimar que el tiempo de ejecución es $O(n)$ del bucle para meter en cola los vértices adyacentes, como el bucle se ejecuta un máximo de n veces, por lo que podemos estimar que el tiempo total es $O(n^2)$. Si el número de arcos a fuera mucho menor que n^2 y representando los arcos mediante listas de adyacencia el tiempo de realización del procedimiento sería $O(a + n)$, siendo a el número de arcos.

Algoritmo del camino más corto: algoritmo de Dijkstra

Tenemos 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. Este algoritmo es un típico ejemplo de algoritmo ávido, que resuelven los problemas en sucesivos pasos, seleccionando en cada paso la solución más óptima en aras de resolver el problema.

Recordamos que al ser un grafo valorado cada arco (v_i, v_j) tiene asociado un coste $c_{ij} \geq 0$. De tal forma que el coste de un camino v_1, v_2, \dots, v_k es $\sum_{i=1}^{k-1} c_{i,i+1}$.

El algoritmo 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 como ya conocido. Así, sucesivamente va marcando vértices hasta que de todos los vértices es conocida la distancia mínima al origen s .

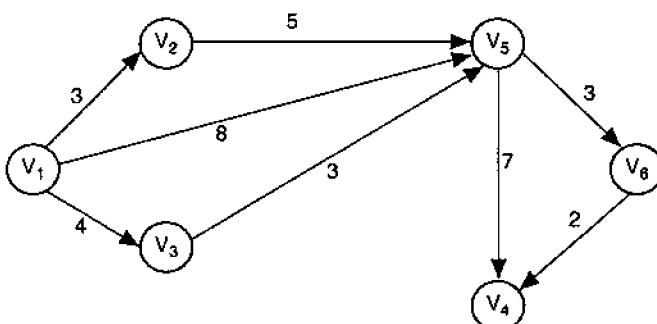
Veamos con más detalle cómo realizar esta estrategia. En un conjunto F tenemos los vértices cuya distancia más corta respecto al origen es ya conocida. En el vector D se almacena la distancia más corta (coste mínimo) desde el origen a cada vértice del grafo. Inicialmente, F contiene únicamente el origen s y los elementos de D , D_i , el coste de los arcos (v_s, v_i) ; si no hay arco de s a i suponemos el coste ∞ . En cada paso se agrega algún vértice v de los que todavía no están en F , es decir, de $V - F$, que tenga el menor valor $D(v)$; además se actualizan los valores $D(w)$ para los vértices w que todavía no están en F : $D(w) = \min(D(w), D(v) + c_{vw})$. De esta manera ya conocemos que hay un camino de s a v cuyo coste mínimo es $D(v)$.

Para reconstruir el camino de coste mínimo que nos lleva de s a cada vértice v del grafo se almacena para cada vértice el último vértice que hizo el coste mínimo. Entonces asociado con cada vértice tenemos dos campos, la distancia o coste mínimo y el último vértice que hizo el camino más corto; esto nos lleva a definir un registro con esos dos estados y una tabla (como ya hicimos en el problema de longitud de camino).

En la figura del ejemplo 14.2 tenemos un grafo dirigido valorado y queremos calcular el coste mínimo desde el vértice 1 al resto de los vértices siguiendo el algoritmo de Dijkstra.

EJEMPLO 14.2

Se tiene el grafo valorado dirigido:



La matriz de adyacencia con los pesos de los arcos y considerando el peso como ∞ cuando no hay arco:

$$C = \begin{pmatrix} \infty & 3 & 4 & \infty & 8 & \infty \\ \infty & \infty & \infty & \infty & 5 & \infty \\ \infty & \infty & \infty & \infty & 3 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 7 & \infty & 3 \\ \infty & \infty & \infty & 2 & \infty & \infty \end{pmatrix}$$

Los valores que se van obteniendo en los sucesivos pasos están representados en forma tabular:

Paso	F	W	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

Así por ejemplo, el camino mínimo de V_1 a V_6 es 10 y la secuencia de vértices que hacen el camino mínimo: $V_1 - V_3 - V_5 - V_6$.

CODIFICACIÓN

En la codificación se supone que el vértice origen es el de índice 1. Además se tiene como entrada la matriz de pesos la cual contiene el coste de cada arco, de no haber arco contiene maxint; {representa el ∞ }.

```

const
  N = 15; { número de vértices }
type
  {tipos para representar los vértices y matriz de adyacencia = matriz de pesos}
  MatAdcia= array[Indicevert,Indicevert] of integer;
  Estado=record
    Distancia: integer;{Suponemos que el factor de peso es entero}
    Ultimo: Vertice;
  end;
  Tabla= array[1..N] of Estado;

procedure Dijkstra(var T:Tabla; C:MatAdcia);
  {C la matriz de pesos = A matriz de adyacencia}
type
  ConjVert= set of 1..N;
var
  I,V,W: IndiceVert;{1..N}
  Todos,F: ConjVert;

procedure Inicial(var T:Tabla);
var v: integer;
begin
  for v:= 2 to N do
  with T[v] do
  begin
    Distancia:= C[1,v];
    Ultimo:= 1
  end
end;
begin
  function Minimo(T:Tabla;R:ConjVert):IndiceVert;
  var
    J,V: IndiceVert;
    Mx: integer;
  begin
    Mx:= maxint;
    for J:= 2 to N do
    if (J in R) and (Mx > T[J].Distancia) then
    begin
      V:= J;
      Mx:= T[J].Distancia
    end;
    Minimo:=V
  end;
  begin
    F:= [1]; {Vertice origen}
    Todos:=[1..N];
    Inicial(T);

```

```

{N-1 pasos para seleccionar los N-1 vértices}
for I:= 1 to N-1 do
begin
  {Búsqueda del vértice de menor distancia}
  W:= Minimo(T,Todos-F);
  F:= F+[W];
  {Se actualizan las Distancias para los vértices restantes}
  for V:=2 to N do
    if V in (Todos-F) then
      with T[V] do
        if (T[W].Distancia+ C[W,V])< Distancia) then
          begin
            Distancia:= T[W].Distancia+ C[W,V];
            Ultimo:= W
          end
    end
  end;
end;

```

En cuanto al tiempo de ejecución del algoritmo, enseguida deducimos que lleva un tiempo de $O(n^2)$ debido a los dos bucles anidados de orden n . 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 integrados en F (Todos-F) con una cola (como en el algoritmo de Longitud de camino), entonces puede obtenerse un tiempo de $O(a \log n)$.

14.4. PROBLEMA DE LOS CAMINOS MÁS CORTOS ENTRE TODOS LOS PARES DE VÉRTICES: ALGORITMO DE FLOYD

En algunas aplicaciones puede resultar interesante determinar el camino mínimo entre todos los pares de vértices de un grafo dirigido valorado. El problema podría resolverse por medio del algoritmo de Dijkstra, aplicándolo a cada uno de los vértices, pero hay otra alternativa que es más directa y es mediante el algoritmo de Floyd.

Sea G un grafo dirigido valorado, $G = (V, A)$. Suponemos que los vértices están numerados de 1 a N ; la matriz de adyacencia A en este caso es la matriz de pesos, de tal forma que todo arco (v_i, v_j) tiene asociado un peso c_{ij} ; si no existe arco (v_i, v_j) suponemos que $c_{ij} = \infty$. Cada elemento de la diagonal se hace 0.

Ahora se quiere encontrar la matriz D de $N \times N$ elementos tal que cada elemento D_{ij} sea el coste mínimo de los caminos de (v_i, v_j) .

El proceso que sigue el algoritmo de Floyd tiene los mismos pasos que el algoritmo de Warshall para encontrar la matriz de caminos. Se genera iterativamente la secuencia de matrices $D_0, D_1, D_2, \dots, D_k, \dots, D_n$ cuyos elementos tienen el significado:

$$D_0[i,j] = \begin{cases} c_{ij} & \text{coste (peso) del arco de } i \text{ a } j \\ \infty & \text{si no hay arco.} \end{cases}$$

$D_1[i,j] = \min(D_0[i,j], D_0[i,1] + D_0[1,j])$. Es decir, el menor de los costes entre el anterior camino de $i \rightarrow j$ y la suma de los costes de caminos de $i \rightarrow 1$, $1 \rightarrow j$.

$D_2[i,j] = \min(D_1[i,j], D_1[i,2] + D_1[2,j])$. Es decir, el menor de los costes entre el anterior camino de $i \rightarrow j$ y la suma de los costes de caminos de $i \rightarrow 2$, $2 \rightarrow j$.

En cada paso se incorpora un nuevo vértice para ver si hace el camino mínimo entre un par de vértices.

$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$. Es decir, el menor de los costes entre el anterior camino de $i \rightarrow j$ y la suma de los costes de caminos de $i \rightarrow k$, $k \rightarrow j$.

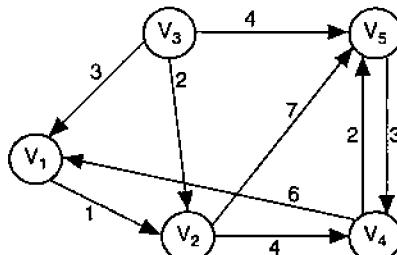
De esta forma hasta llegar a la matriz D_n que será la matriz de caminos mínimos del grafo.

EJEMPLO 14.3

En la figura siguiente se tiene un grafo dirigido con factor de peso; aplicando los sucesivos pasos del algoritmo de Floyd se llega a la matriz de caminos mínimos.

La matriz de pesos:

$$C = \begin{pmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & \infty & 4 \\ 6 & \infty & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{pmatrix}$$



Las sucesivas matrices D_1, D_2, \dots, D_5 que es la matriz de caminos mínimos:

$$D_1 = \begin{pmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & \infty & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{pmatrix}$$

Al incorporar el vértice 1 ha cambiado $D_1(4,2)$ ya que $C(4,1) + C(1,2) < C(4,2)$.

$$D_2 = \begin{pmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{pmatrix}$$

Al incorporar el vértice 2 ha habido varios cambios. Así ha ocurrido con $D_1(1,4)$:

$$D_1(1,2) + D_1(2,4) < D_1(1,4).$$

$$D_3 = \begin{pmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{pmatrix}$$

Al incorporar el vértice 3 no ha habido cambios; al vértice 3 no llega ningún arco.

Así se seguiría para determinar D_4 y D_5 .

Al igual que se ha hecho en el algoritmo de Dijkstra por cada vértice se desea guardar el índice del último vértice que ha conseguido que el camino sea mínimo del v_i a v_j , en caso de que el camino sea directo tiene un cero. Para ello se utiliza una matriz de vértices.

CODIFICACIÓN

```

const
  N = 15; (número de vértices)
type
  (tipos para representar los vértices y matriz de adyacencia = matriz de
  pesos)
  MatAdcia = array[Indicevert,Indicevert] of integer;
  MatCmo= array[Indicevert,Indicevert] of 0..N;
procedure Floyd(C: MatAdcia; var D: MatAdcia; var Tr: MatCmo);
var
  i,j,k: integer;
begin
  for i:= 1 to n do
    for j:= 1 to n do
      begin
        D[i,j] := C[i,j];
        Tr[i,j] := 0
      end;
  (El camino mínimo de un vértice a sí mismo se considera cero)
  for j:= 1 to n do
    D[j,j] := 0;
  for k = 1 to n do
    for i = 1 to n do
      for j:= 1 to n do
        if (D[i,k]+D[k,j])<D[i,j] then
          begin
            D[i,j] := D[i,k] + D[k,j];
            Tr[i,j] := k
          end
      end
    end;
  end;

```

14.4.1. Recuperación de caminos

En la matriz Tr se ha guardado el último vértice que ha hecho el camino de coste mínimo del vértice i al j . Para obtener la sucesión de vértices que determinan el camino hay que «volver hacia atrás», y qué mejor forma de hacerlo que con llamadas recursivas.

El procedimiento Camino recibe el par de vértices y utiliza la matriz Tr como variable global para no cargar el tiempo de ejecución.

```
procedure Camino(Vi,Vj:IndiceVert);
var Vk: integer;
begin
  Vk:= Tr[Vi,Vj];
  if Vk <> 0 then
    begin
      Camino(Vi,Vk);
      write(Vk, ' ');
      Camino(Vk,Vj)
    end
  end;
end;
```

14.5. CONCEPTO DEL FLUJO

La primera idea de flujo se obtiene del significado popular: forma de enviar objetos de un lugar a otro.

Ejemplos de la vida diaria: 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.

Esta es la idea intuitiva de flujo, ahora exponemos la formulación matemática de flujo:
Se llama flujo a una función F_{ij} definida en A (arcos) que verifica las propiedades:

1. $F_{ij} \geq 0 \quad \forall (i, j) \in A$
2. $\sum_{j \in \alpha_i} F_{ij} - \sum_{j \in \beta_i} F_{ji} = 0 \quad i \in V, i \neq S, i \neq T$

α_i ≡ conjunto vértices conectados con el vértice i mediante arcos que salen de i .

β_i ≡ conjunto vértices conectados con el vértice i mediante arcos que entran en i .

V es el conjunto de nodos.

S es el nodo inicial del flujo.

T es el nodo destino del flujo.

3. $F_{ij} \leq U_{ij} \quad \forall (i, j) \in A$

Siendo U_{ij} la capacidad del arco i,j .

La segunda condición impone la conservación de la cantidad total de flujo (ley de Kirchoff). Las condiciones primera y tercera imponen la cota superior e inferior de los valores del flujo.

14.5.1. Planteamiento del problema

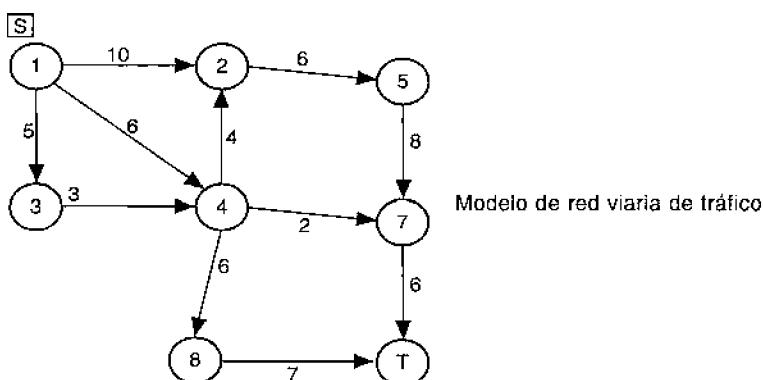
Sobre una red de flujos pueden plantearse diversos problemas. Así se puede desear maximizar la cantidad transportada desde un vértice origen a un vértice destino. Otro problema típico es conocer el modo más económico de enviar una determinada cantidad de objetos desde un origen a un destino en un sistema de distribución.

Por consiguiente, un grafo con factor de peso, una red de flujo es una estructura ideal para modelar estas situaciones.

EJEMPLO 14.4

Planificación de rutas alternativas para circulación en horas punta.

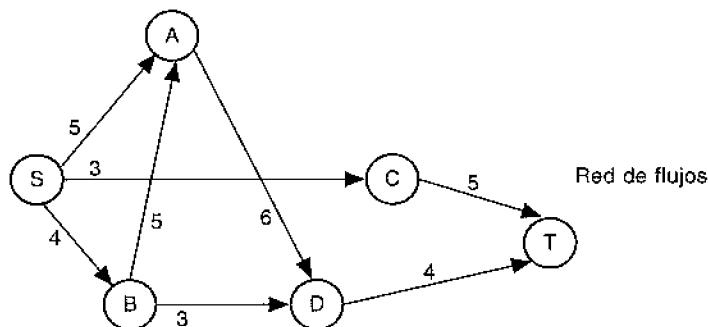
Supóngase un punto de origen con fuerte tráfico, por ejemplo una ciudad dormitorio, y otro punto que representa el centro, zona comercial, de la gran ciudad. Entre ambos puntos se pueden establecer diversas rutas de entrada, cada una de ellas con una capacidad de tráfico máxima, dato que es conocido y expresado en miles de vehículos por



hora. 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. Esto nos conduce a un problema de optimización.

EJEMPLO 14.5

Consideremos un sistema de tuberías de transporte de agua; cada arco representa un tubo y el factor de peso es la capacidad en litros/minuto. Los vértices representan puntos en los cuales las tuberías están unidas y el agua es transportada de un tubo a otro. Dos nodos, S y T, representan la fuente de emisión del agua y el punto de consumo, respectivamente. Se desea maximizar la cantidad de agua que fluye desde el vértice fuente hasta su punto de consumo.



Los pesos representan la capacidad de cada tubería en litros/minutos.

14.5.2. Formulación matemática

Sea la red $G = (V, A)$ con un único vértice S fuente y un único sumidero T , donde asociado a cada arco (i, j) hay un número U_{ij} que representa la capacidad máxima del arco.

$$\sum_{v \in V} F(S, X) = Vf = \sum_{v \in V} F(X, T)$$

la cantidad de «agua», «tráfico» ... 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 vértice sumidero T .

Matemáticamente, maximizar $\sum_{i \neq t} f_{it}$

sujeto a las condiciones (flujo que entra = flujo que sale): $\sum_j f_{ij} - \sum_i f_{ji} = 0; \quad i \neq S, i \neq T$

$$0 \leq f_{ij} \leq U_{ij} \quad (i, j) \in A$$

La función objetivo es la suma de los flujos que llegan al vértice sumidero. Es precisamente la cantidad que se desea hacer máxima. Esta función de flujo se denomina *el óptimo*.

14.5.3. Algoritmo del aumento del flujo: algoritmo de Ford y Fulkerson

Este es uno de los algoritmos 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 partir de una función de flujo, flujo cero, e iterativamente ir mejorando el flujo. La mejora se da en la medida que el flujo de S hasta T aumenta, teniendo en cuenta las condiciones que ha de cumplir la función de flujo en lenguaje natural:

- Flujo que entra a un nodo ha de ser igual al flujo que sale.
- En todo momento el flujo no puede superar la capacidad del arco.

Los arcos de la red pueden clasificarse en tres categorías:

- *No modificable*: arcos cuyo flujo no puede aumentarse ni disminuirse, por tener capacidad cero o tener un coste prohibitivo.
- *Incrementable*: arcos cuyo flujo puede aumentarse, transportan un flujo inferior a su capacidad.
- *Reducible*: arcos cuyo flujo puede ser reducido.

Con estas categorías podemos establecer las siguientes mejoras de la función de flujo desde S a T:

- 1.^a *Forma de mejora*: encontrar un camino P del vértice fuente S al sumidero T tal que el flujo a través de cada arco del camino (todos los arcos incrementables) no supera a la capacidad:

$$F(V_i, V_j) \leq U_{i,j} \quad \forall (i, j) \in P$$

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 sumidero T a la fuente S formado por arcos reducibles, entonces es posible reducir el flujo de T a S y por tanto aumentar en las mismas unidades de flujo de S a T en la cantidad:

$$\text{Mínimo } \{F_{i,j}, \forall (i, j) \in P'\}$$

- 3.^a *Forma de mejora*: existe un camino P1 desde S hasta algún nodo V con los arcos incrementables, un camino P2 desde un nodo W hasta el sumidero T con arcos incrementables, y un camino P3 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:

$$\begin{aligned} &\text{Mínimo } \{(U_{i,j} - F_{i,j}), \quad \forall (i, j) \in P1, \text{ o bien, } \in P2\} \\ &\text{Mínimo } \{F_{i,j}, \quad \forall (i, j) \in P3\} \end{aligned}$$

Este mínimo se llama aumento de flujo máximo de la cadena. Acaba de aparecer en este contexto de redes el concepto de cadena, conviene que establezcamos una definición.

Una cadena que une los vértices v_i, v_j es una sucesión de arcos 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.

De manera más escueta, una cadena, también llamada *semicamino*, desde V a W es la secuencia de nodos $V = X_1, X_2, \dots, X_n = W$ tal que para todo i , $1 \leq i \leq n$ se cumple que (X_{i-1}, X_i) o (X_i, X_{i-1}) es un arco.

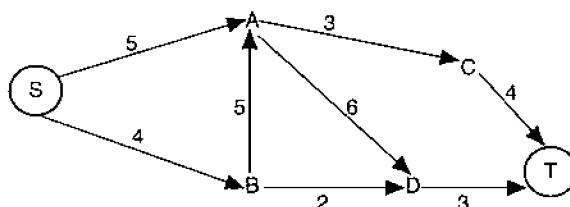
De inmediato podemos afirmar que todo camino P de V a W es una cadena, aunque no toda cadena es un camino. En el grafo de la figura siguiente encontramos que una cadena de S a T es:

$$S - A - B - D - T$$

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. El objetivo que persigue el algoritmo de aumento de flujo es encontrar cadenas de aumento de flujo en la red y aumentar el flujo en la cantidad determinada en los tres modos expuestos. La solución óptima se encuentra cuando ya no hay más cadenas de aumento de flujo.

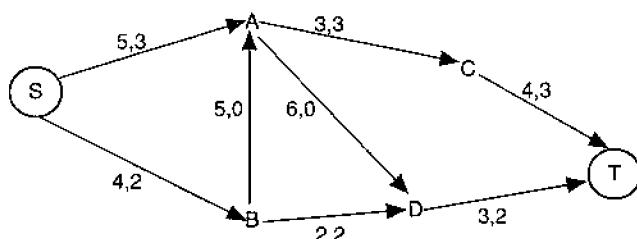
14.5.4. Ejemplo de mejora de flujo

Considérese de nuevo la red de tuberías

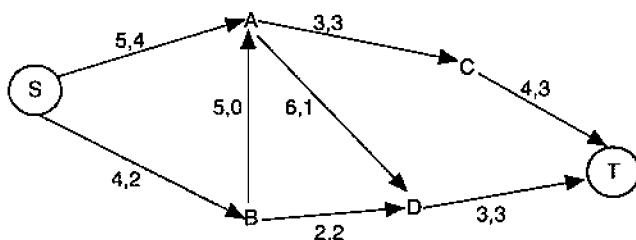


los pesos de los arcos representan las capacidades en litros/minuto.

Existen los caminos P_1 $S-A-C-T$, P_2 $S-B-D-T$ y P_3 $S-A-D-T$ con arcos incrementables. Por el camino P_1 el flujo puede ser mejorado en 3 unidades, por el camino P_2 el flujo es mejorable en dos unidades. Una vez realizadas estas mejoras la función de flujo queda:

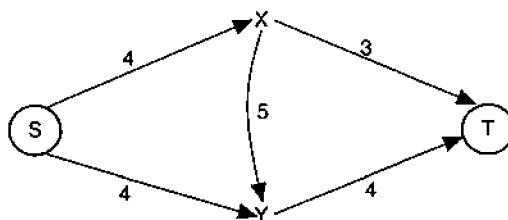


Ahora en los arcos se ha representado la capacidad y el flujo actual. Aún puede hacerse otra mejora por el camino P3 en 1 unidad. La función flujo quedará:

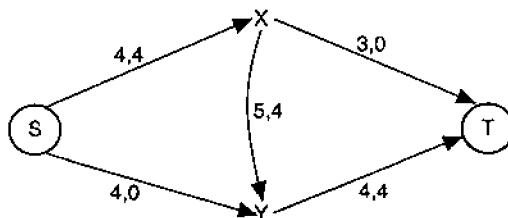


Como se observa de la fuente S salen 6 unidades de flujo que son las que llegan al sumidero T.

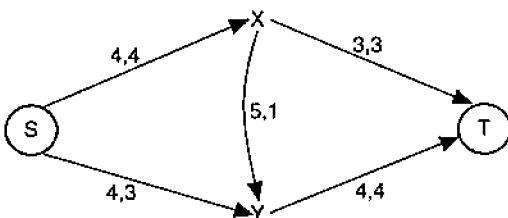
Para que sirva de ejemplo de mejora de flujo por aumento de cadena, sea el siguiente grafo en el que representamos la capacidad de cada arco:



La función de flujo queda con la mejora por caminos con arcos incrementables:



Ahora si el flujo de Y a X es reducido, aumentando en las mismas unidades el de S a Y, el flujo de entrada de X a T aumenta en dicha cantidad. Ha habido un aumento neto del flujo de S a T. Expresándolo en términos de cadena: hay un camino de S a Y con arcos incrementables (4,0), un camino de X a Y con arcos reducibles (4) y un camino de X a T con arcos incrementables (3,0); en definitiva, la cadena S - Y - X - T. El aumento de flujo por la cadena es el valor mínimo entre arcos incrementables y reducibles, que en este caso es 3 unidades. La función de flujo:



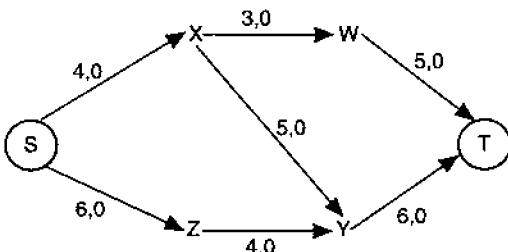
14.5.5. Esquema del algoritmo de aumento de flujo

Los pasos que sigue el algoritmo de aumento de flujo:

1. Parte de un flujo inicial, $F_{ij} = 0$. Determina arcos que son incrementables y los que son reducibles. Marca el vértice fuente S .
2. Repetir hasta que sea marcado el vértice sumidero T , o bien no sea posible marcar más vértices:
 - 2.1. Si un vértice i está marcado, existe el arco (i, j) y es tal que (i, j) es arco incrementable, entonces marcar vértice j .
 - 2.2. Si un vértice i está marcado, existe el arco (j, i) y es tal que (j, i) es un arco reducible, entonces marcar el vértice j .
3. Si ha sido marcado el vértice sumidero T , hemos obtenido una cadena de aumento de flujo. Aumentar el flujo al máximo aumento de flujo permitido por la cadena. Actualizar los flujos de cada arco con las unidades de aumento. Borrar todas las marcas, salvo la del vértice S , y repetir a partir del paso 2.
4. Si no ha sido marcado el vértice T , finalizar la aplicación del algoritmo; no es posible enviar más flujo desde S hasta T .

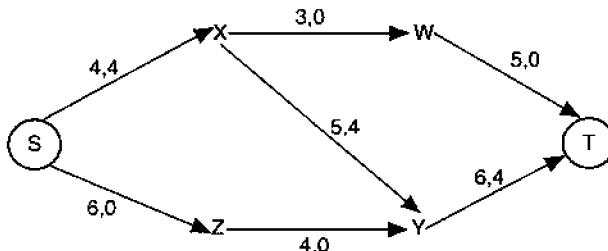
EJEMPLO 14.6

Dada la siguiente red, en la que los pesos de los arcos representan la capacidad y el flujo actual, se aplican los pasos que propone el algoritmo de Ford-Fulkerson para encontrar el flujo máximo.

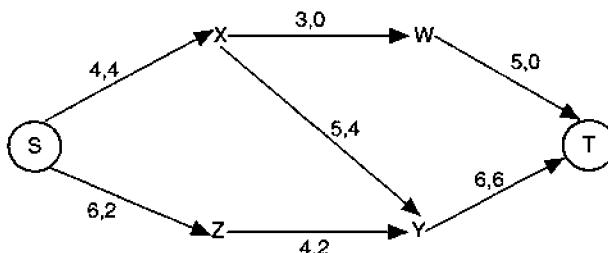


En una primera pasada se marcan los vértices S, X, Y, T ; todos los arcos son incrementables, se tiene una cadena de aumento de flujo. El incremento de flu-

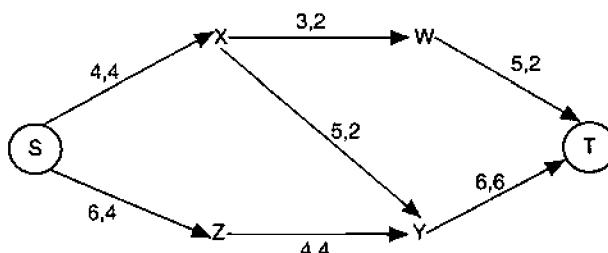
$\Delta_j = \min\{C_{ij} - F_{ij}\}$ para todo arco de la cadena}, en esta cadena es 4. Ahora se actualizan los flujos de cada arco con el valor 4.



En una segunda pasada se marcan los vértices S, Z, Y, T; todos los arcos son incrementables, se tiene una segunda cadena de aumento de flujo. El incremento de flujo = $\min\{C_{ij} - F_{ij}\}$ para todo arco de la cadena}, en esta cadena es 2. Se actualiza los flujos:



En una tercera pasada se marcan los vértices S, Z, Y, X, W, T; los arcos (S, Z), (Z, Y) son incrementables, el arco (X, Y) es reducible y los arcos (X, W), (W, T) son incrementables; se tiene una tercera cadena de aumento de flujo. El incremento de flujo de la cadena es 2. Se actualizan los flujos:



En esta red ya no es posible encontrar más cadenas de aumento de flujo. Por tanto, el flujo máximo que llega a T es $4 + 4 = 8$; su solución está representada en la figura anterior.

El análisis de la solución proporciona información muy valiosa para valorar en qué arcos conviene aumentar la capacidad para que aumente más el flujo que llega a T.

14.5.6. Tipos de datos y pseudocódigo

Los procesos indicados anteriormente para encontrar el flujo máximo se detallan a continuación:

```

inicio
  Inicializar la función de flujo a cero en cada arco
  Nocadena ← false
  repetir
    <Encontrar una cadena de S a T que incremente flujo hacia T en x>
    si <No se puede encontrar cadena> entonces
      Nocadena ← true
    sino
      <Aumentar flujo a cada nodo en la cadena en x>
    fin_si
  hasta Nocadena
fin

```

Los tipos de datos son aquellos necesarios para representar las acciones indicadas en el algoritmo.

Para «marcar» un nodo, una vez que éste ha sido colocado en una cadena, utilizamos el vector boolean Encadena, de tal forma que Encadena[nodo] indica si el nodo está en la cadena que se está formando o no. En el vector *boolean* Finvia un elemento Finvia[nodo] indica si el nodo está al final de la cadena que se está formando.

Para obtener correctamente la secuencia de nodos que forman la cadena, el vector Precede es tal que Precede[nodo] referencia al nodo precedente del nodo índice en la cadena de la que forma parte. El vector *boolean* Adelante es tal que Adelante[nodo] es true si el arco es hacia adelante, hacia T; en definitiva, el arco (precede[nodo], nodo).

Las unidades de flujo acumuladas hasta un nodo se guardan en la posición correspondiente del vector Flujos.

La capacidad de un arco y su flujo actual se almacenan respectivamente en las posiciones correspondientes de las matrices C, F: C[i, j] y F[i, j].

Ahora se pueden expresar con un poco mas de detalle el algoritmo de aumento de flujo:

```

Inicio
  Para todos los nodos:
    Finvia[nodo]← false
    Encadena[nodo]← false
  {S, T son los nodos origen y sumidero}
    Finvia[S]← true
    Encadena[S]← true
  Fin_Para
  mientras (no Encadena[T]) y (exista nd.,Finvia[nd]) hacer
    Finvia[nd] ← false

```

```

mientras existe nodo i,,(no Encadena[i]) y
    (F[nd,i]< C[nd,i])y Adyacente(nd,i)hacer
        {arco{nd,i} es incrementalable}
        Encadena[i], Finvia[i], Adelante[i]← true
        Precede[i]← nd
        x ← C[nd,i]- F[nd,i]
        {Almacena el minimo de los arcos ...}
        si Flujos[nd]< x entonces
            Flujos[i]← Flujos[nd]
        sino
            Flujos[i]← x
        fin_si
    fin_mientras
    {Ahora busca arcos hacia S reducibles }
    mientras existe nodo i,,(no Encadena[i]) y
        F[i,nd]>0 y Adyacente(i,nd) hacer
        {arco (i,nd) es reducible}
        Encadena[i], Finvia[i]← true
        Adelante[i]← false
        Precede[i]← nd
        si Flujos[nd]< F[i,nd] entonces
            Flujos[i]← Flujos[nd]
        sino
            Flujos[i]← F[i,nd]
        fin_si
    fin_mientras
fin_mientras {termina bucle de búsqueda de cadena}
si Encadena[T] entonces
    <Aumentar el flujo en los arcos de la cadena>
sino
    <El flujo ya es máximo>
fin_si
fin

```

14.5.7. Codificación del algoritmo de flujo máximo: Ford-Fulkerson

En la unidad Flujo encapsulamos los tipos de datos y los procedimientos que implementan el algoritmo de Ford-Fulkerson.

```

unit UnitFlux;
interface
  const
    Maxnodos = 10;
  type
    Indexnodo = 1..Maxnodos;
    Vnodos = array[Indexnodo] of Indexnodo;
    Matriz = array [Indexnodo,Indexnodo] of integer;
    Vectorlogico =array [Indexnodo] of boolean;
  procedure Imprime (Fj: matriz);
  procedure Maxflujo (Cap:Matriz; S, T:Indexnodo;
                      var Fj: Matriz; var Fjototal:integer);
implementation
  uses printer;
  procedure Cadena(var Pred: Vnodos; S,A: Indexnodo);

```

```

var
  Nd: Indexnodo;
begin
  if A= S then
    write(lst,' Cadena de aumento de flujo: ':40, S)
  else begin
    Cadena(Pred, S, Pred[A]);
    write(lst,' -- ', A)
  end
end;
end;

procedure Imprime (Fj: Matriz);
var
  Vi,Vj: Indexnodo;
begin
  writeln (lst,'Flujo de cada arco de la red':50);
  writeln (lst,'Origen Arco, Final Arco, Flujo':55);
  for Vi:= 1 to Maxnodos do
    for Vj:= 1 to Maxnodos do
      if Fj[Vi,Vj] <>0 then
        writeln (lst, Vi:25, ' :14, Vj, ' :11,Fj[Vi,Vj]);
  writeln(lst);
  writeln(lst)
end;

procedure Maxflujo;
var
  pred,nd,i: Indexnodo;
  x: integer;
  Precede: Vnodos;
  Flujo: array[Indexnodo] of integer;
  Finvia,Adelante,Encadena: Vectorlogico;

  function Alguno (Finvia :Vectorlogico): boolean;
    {Busca si existe un nodo i que sea fin de vía}
  var
    i: Indexnodo;
    Encontrado: boolean;
  begin
    i:= 1;
    Encontrado:= false;
    while (i<= Maxnodos) and not Encontrado do
      if Finvia[i] then
        Encontrado:=true
      else
        i:=i+1;
    Alguno:=Encontrado
  end;

begin { inicio del proced. Maxflujo }
  for nd:=1 to Maxnodos do
    for i:=1 to Maxnodos do
      Fj[nd,i]:=0;
  Fjototal:=0;
  repeat
    { tratamos de encontrar una vía desde S a T }
    for nd:=1 to Maxnodos do

```

```

begin
  Finvia[nd]:=false;
  Encadena[nd]:=false
end;
Finvia[S]:= true;
Encadena[S]:= true;
Flujo[S]:= maxint ; {máximo valor posible}
while (not Encadena[T]) and (Alguno(Finvia)) do
begin
  nd:=1;
  while not Finvia[nd] do {búsqueda del último nodo en la
                           semicadena actual}
    nd:=nd+1;
  Finvia[nd]:= false;
  for i:=1 to Maxnodos do
  begin
    {La condición de adyacentes es cierta en cuanto
     Cap[nd,i]>0}
    {Inspecciona si es arco incrementable}
    if (Fj[nd,i]<Cap[nd,i]) and (not Encadena[i]) then
    begin
      Encadena[i]:=true;
      Finvia[i]:=true;
      Precede[i]:=nd;
      Adelante[nd]:=true;
      x:=Cap[nd,i]-Fj[nd,i];
      if Flujo[nd]<x then
        Flujo[i]:=Flujo[nd]
      else
        Flujo[i]:=x
    end;
    {Ahora se trata la posibilidad de que sea un arco
     hacia S, en vez de hacia T. En cuyo caso el arco
     podría ser reducible}
    {La condición de adyacentes es cierta en cuanto
     Fj[i,nd]>0}
    if (Fj[i,nd]>0)and (not Encadena[i]) then
    begin
      Encadena[i]:=true;
      Finvia[i]:=true;
      Precede[i]:=nd;
      Adelante[nd]:=false;
      if Flujo[nd] <Fj[i,nd] then
        Flujo[i]:=Flujo[nd]
      else
        Flujo[i]:=Fj[i,nd]
    end
  end {última sentencia del bucle for}
end ; {última sentencia del bucle while}

if Encadena[T] then
  {Ha sido encontrada una cadena, puede aumentar el flujo en la
   semivía(cadena) al sumidero T)
begin
  x:= Flujo[T];
  Fjototal:= Fjototal+x;

```

```

nd:=T;
{Regresa por los nodos de la cadena, hasta S, para actualizar
los arcos}
while nd <> S do
begin
  pred:= Precede[nd];
  if Adelante[pred] then {aumenta el flujo desde el predecesor}
    Fj[pred,nd]:= Fj[pred,nd]+x
  else {es reducible}
    Fj[nd,pred]:=Fj[nd,pred]-x;
  nd:= pred
end;
Cadena(Precede, S, T); {Escribe la cadena }
writeln(lst)
end
until not Encadena[T];
Imprime(Fj)
end; {fin del procedimiento}
begin
end.

```

Obsérvese que en el procedimiento que implementa el algoritmo, cada vez que se encuentra una cadena de aumento de flujo se escribe la misma. Para poder hacer un mejor seguimiento, la salida se dirige a impresora.

Por último, el programa que se muestra a continuación requiere primero que se introduzca la capacidad de los arcos de la red. A continuación llama al procedimiento de flujo máximo.

```

program flujo_maximo;
uses UnitFlux, printer;
var
  Cap, Fj: Matriz;
  Fjtotal: integer;
  N1, N2, S, T: Indexnodo;
begin {programa principal}
  {La capacidad es inicializada a 0}
  for N1:= 1 to Maxnodos do
    for N2:= 1 to Maxnodos do
      Cap[N1,N2]:=0;
  {Son requeridas las capacidades de los arcos de la red}
  writeln ('La secuencia de datos es: ');
  writeln (' Nodo inicial, final del arco y su capacidad');
  writeln (' Termina la entrada con ^Z');
  while not eof do
    readln (N1,N2,Cap[N1,N2]);
    writeln;
    write ('Vértice fuente y sumidero: ');
    readln (S,T);
    Maxflujo(Cap,S,T,Fj,Fjtotal);
    writeln (lst);
    writeln (lst,' ':14,'Flujo total: ',Fjtotal)
  end.

```

14.6. PROBLEMA DEL ÁRBOL DE EXPANSIÓN DE COSTE MÍNIMO

Este problema se aplica sobre grafos no dirigidos. Un grafo no dirigido $G = (V, A)$ es un conjunto finito de vértices V y de arcos A . Cada arco está formado por un par no ordenado de vértices.

Los grafos no dirigidos se emplean para modelar relaciones simétricas entre entes. Los entes están representados por los vértices del grafo.

14.6.1. Definiciones

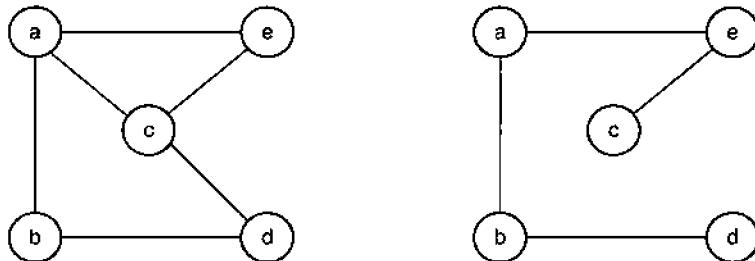
Red conectada: es una red tal, que cualquier par de vértices pueden ser unidos mediante un camino.

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 $n-1$ arcos.
2. Si se añade un arco a un árbol, entonces resulta un ciclo.

Árbol de expansión: es un árbol que contiene a todos los vértices de una red.

De esta última definición concluimos que buscar un árbol de expansión de una red es una forma de averiguar si la red es conectada.



La figura anterior muestra un grafo no dirigido y su árbol de expansión. Según el árbol de expansión el grafo es una red conectada.

14.6.2. Árboles de expansión de coste mínimo

Una de las aplicaciones típicas de los árboles de expansión está en el diseño de redes de comunicación en todas sus vertientes.

Pensemos en los pueblos que forman parte del Ayuntamiento de Sigüenza¹; cada par de pueblos está conectado por un camino vecinal, el peso de un camino directo entre dos

¹ Pueblo de la provincia de Guadalajara (España).

pueblos, un arco, viene dado por la distancia en kilómetros. Se quiere construir una red de carriles-bicicleta de coste mínimo de tal forma que cada par de pueblos esté comunicado a un coste mínimo.

El planteamiento, 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 y sin ciclos, cuya suma de los costes de los arcos sea mínima. Hay que encontrar el árbol de expansión de coste mínimo.

14.7. ALGORITMO DE PRIM Y KRUSKAL

El algoritmo de Prim para encontrar el árbol de expansión mínimo sigue la metodología de hacer en cada iteración «lo mejor que se puede hacer», esta metodología se llama «*algoritmo voraz*».

El punto de partida es un grafo $G = (V, A)$ que es una red, siendo $c(i, j)$ el peso o coste asociado al arco (i, j) .

Supóngase $V = \{1, 2, 3, \dots, n\}$, el algoritmo arranca asignando un vértice inicial al conjunto W , por ejemplo el vértice 1

$$W = \{1\}$$

A partir del vértice inicial el árbol de expansión crece, añadiendo en cada iteración otro vértice z de $V-W$ tal que si u es un vértice de W , el arco (u, z) es el más corto. El proceso termina cuando $V = W$.

Observamos que en todo momento el conjunto de nodos que forma W forman una componente conexa sin ciclos.

Otra forma de expresar el algoritmo de Prim es que, partiendo de un vértice inicial u , tomar la arista menor (u, v) que no forme un ciclo, y así iterativamente ir tomando nuevos arcos de menor peso (u, v) sin formar ciclos y formando en todo momento una componente conexa.

El conjunto de vértices de la Figura 14.2 $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ parte de $W=\{1\}$, el arco mínimo es $(1, 2)$, por lo que $W = \{1, 2\}$.

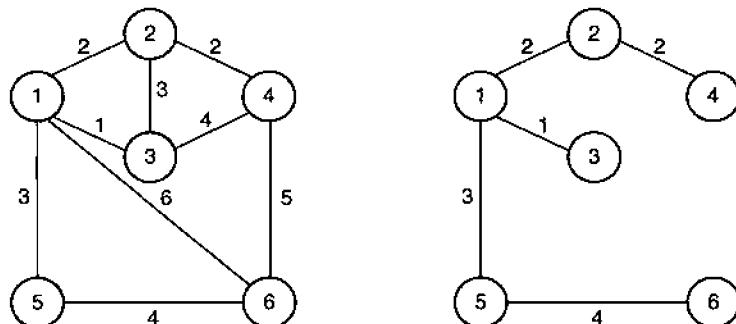


Figura 14.1. Grafo valorado y su árbol de expansión mínimo.

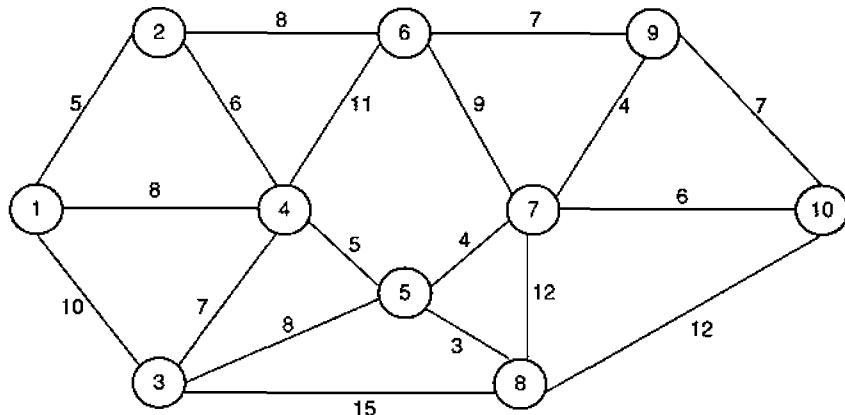


Figura 14.2. Grafo valorado conexo; representa una red telefónica.

El siguiente arco mínimo $(2, 4)$, $W=\{1, 2, 4\}$. El algoritmo puede expresarse:

```

procedimiento Prim(G, T)
  { G : grafo
    T : conjunto de arcos del árbol de coste mínimo}
  var
    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 , (u,v) sea mínimo, siendo u ∈ W>
      W <- W+{v}
      T <- T+{(u,v)}
    fin_mientras
  fin_procedimiento

```

Para encontrar el arco de menor coste entre W y $V-W$ en cada iteración se utilizan dos arrays:

Mas_Cerca, tal que $\text{Mas_Cerca}[i]$ contiene el vértice de W de menor coste respecto el vértice i de $V-W$.
Coste, tal que $\text{Coste}[i]$ contiene el coste del arco $(i, \text{Mas_Cerca}[i])$.

En cada paso se revisa Coste para encontrar algún vértice z de $V-W$ que sea el más cercano a W . A continuación se actualizan los arrays **Mas_Cerca** y **Coste**, teniendo en cuenta que z ha sido añadido a W .

La implementación del algoritmo en Pascal:

```

procedure Prim (C: Matriz; var Ct: real);
{C: Matriz de adyacencia. Si no existe arco (i,j), Cij tiene el
 valor infinito.
 Ct: Coste del árbol de expansión mínimo}
var
  Coste: array[1..n] of real;
  Mas_Cerca: array[1..n] of integer;
  i, j: integer;
  z : Indexnodo;
  Min : real;
  W : ConjVert;
begin
  Ct:= 0;
  {Vértice inicial es el 1}
  W:= [1];
  for i:= 2 to n do
  begin
    Coste[i]:= C[1,i]; {Inicialmente el coste de cada vértice de
                          V-[1] es el arco (1,i)}
    Mas_Cerca[i]:= 1 {debido a que W=[1]}
  end;
  for i:= 2 to n do
  begin
    {Encuentra el vértice z de V-W más cercano (menor arco a algún
     vértice de W)}
    Min:= Coste[2];
    z:= 2;
    for j:= 3 to n do
      if Coste[j]< Min then
      begin
        Min:= Coste[j];
        z:= j
      end;
    Ct:= Ct+ Min;
    {Escribe arco del árbol de expansión}
    writeln(z, ' -- ',Mas_cerca[z]);
    {El vértice z es añadido a W}
    Coste[z]:= Infinito;
    W:= W+[z];
    {Ajusta los costes del resto de vértices}
    for j:=2 to n do
      if (C[z,j]< Coste[j]) and not (j in W) then
      begin
        Coste[j]:= C[z,j];
        Mas_Cerca[j]:= z
      end;
    end;
  end;
end;

```

Algoritmo de Kruskal

De nuevo tenemos un grafo conexo no dirigido valorado $G = (V, A)$ y una función de coste c_{ij} definida en los arcos de A . Kruskal propone otra estrategia para encontrar el árbol de expansión de coste mínimo.

El algoritmo comienza con un grafo T con los mismos vértices V pero sin arcos. Se puede decir que cada vértice es una componente conexa en sí mismo.

Para construir componentes conexas cada vez mayores, se examinan los arcos de A, en orden creciente del coste. Si el arco conecta dos vértices que se encuentran en dos componentes conexas distintos, entonces se añade el arco a T. Se descartarán los arcos si conectan dos vértices contenidos en el mismo componente, ya que pueden provocar un ciclo si se le añadiera al árbol de expansión para ese componente conexo. Cuando todos los vértices están en un solo componente, T es un árbol de expansión de coste mínimo del grafo G.

En el algoritmo siguiente se definen estos tipos de datos:

```

IndexNodo= 1..n;
ConjVert= set of IndexNodo;
Arco= record
  u, v: IndexNodo;
  Coste: real
end;
ListArcos= array[1..n*n] of Arco;
ListConjs= array[1..n] of ConjVert;

procedure Kruskal (C: matriz; var T: ListArcos; var Ct:real);
var
  Arcos: ListArcos;
  CompConx: ListConjs;
  u, v,
  C_u, C_v,
  Comp_n : IndexNodo;
  Kn, Ka, K: integer;
  Min_Ar: Arco;

begin
  Kn:= 0; {Número de arcos en T}
  Ka:= 0; {Número de arcos de la matriz de costes C}
  Ct:= 0; {Coste del árbol de expansión}
  Inicial(C,Arcos,Ka); {obtiene los Ka arcos en orden creciente de costes}

  for v:= 1 to n do
    CompConx[v]:= [v]; {Obtiene los n componentes conexos iniciales,
                         con cada vértice}
  Comp_n:= n; {Número de componentes conexos}
  K:= 0;
  while Comp_n> 1 do
    begin
      K:= K+1;
      Min_Ar:= Arcos[K]; {Arco mínimo actual}
      C_u:= NumCompon(Min_Ar.u, CompConx); {Componente donde se
                                                encuentra el vértice u del arco mínimo}
      C_v:= NumCompon(Min_Ar.v, CompConx);
      if C_u<>C_v then {Conecta dos componentes distintos}
        begin
          Ct:= Ct+ Min_Ar.Coste;
          writeln('Arco ', Min_Ar.u, ' -> ', Min_Ar.v);
          Combina(C_u,C_v,CompConx);{Une componentes u y v.Nueva componente
                                    queda en u}
        end;
    end;
end;

```

```

Comp_n:= Comp_n-1;
Kn:= Kn+1;
T[Kn]:= Min_Ar
end
end;
end;

```

14.8. CODIFICACIÓN. ÁRBOL DE EXPANSIÓN DE COSTE MÍNIMO

En la siguiente unidad encapsulamos los tipos de datos necesarios para realizar los algoritmos que determinan el árbol de expansión mínimo. En esta unidad también quedan incorporados los tipos para representar la matriz de adyacencia.

```

unit UniCamno;
interface
  const
    Maxnodos= 10;
    n= Maxnodos; {Para mantener identificadores}
  type
    Indexnodo= 1..Maxnodos;
    ConjVert= set of Indexnodo;
    Arco= record
      u, v: Indexnodo;
      Coste: real
    end;
    ListArcos= array{1..n*n} of Arco;
    ListConjs= array{1..n} of ConjVert;
    Matriz= array [Indexnodo,Indexnodo] of real;
  procedure Prim (C: Matriz; var Ct: real);
  procedure Kruskal(C: Matriz; var T: ListArcos; var Ct: real);
  procedure Inicial(C: Matriz; var T: ListArcos; var Ka: integer);
  procedure Combina(Cu,Cv: Indexnodo; var Cx:ListConjs);
  function NumCompon(v: Indexnodo; Cx:listConjs): integer;

implementation
  procedure Ordena_Rapido (var Lars: ListArcos; K: integer);
    procedure Sort (iz,de: integer);
    var
      i, p: integer;
      X, W: Arco;
    begin
      i:=iz;
      p:=de;
      X:=Lars[{(iz+de) div 2}];
      repeat
        while Lars[i].Coste < X.Coste do
          i:=i+1;
        while x.Coste < Lars[p].Coste do
          p:=p-1;
        if i <= p then

```

```

begin
  W:= Lars[i];
  Lars[i]:= Lars[p];
  Lars[p]:= W;
  i:= i+1;
  p:= p-1
end
until i > p;
if iz < p then Sort (iz,p);
if i < de then Sort (i,de);
end;
begin
  Sort (1,K)
end;

procedure Inicial(C: Matriz; var T: ListArcos; var Ka: integer);
{De la matriz de costes, obtiene los arcos en orden ascendente de peso}
var
  J,I: Indexnodo;
begin
  {Primero son almacenados los arcos.
  Para después ordenarlos según el coste}
  Ka:= 0;
  for I:= 1 to n do
    for J:= 1 to n do
      if C[I,J]<> 0 then { arco <i,j> }
      begin
        Ka:= Ka+1;
        with T[Ka] do
        begin
          U:= I;
          V:= J;
          Coste:= C[I,J]
        end
      end;
  Ordena_Rapido(T, Ka)
end;

procedure Combina(Cu,Cv: Indexnodo; var Cx: ListConjs);
{Une las componentes conexas Cu, Cv. La unión queda en la posición
Cu; la componente de la posición Cv queda vacía}
begin
  Cx[Cu]:= Cx[Cu]+Cx[Cv];
  Cx[Cv]:= []
end;

function NumCompon(v: Indexnodo; Cx: ListConjs): integer;
{Obtiene el número de componente donde se encuentra el vértice v}
var
  Nc: integer;
  Comx: boolean;
begin
  {El número de componentes es n, aunque puede haber componentes vacías}
  Nc:= 0;
  Comx:= false;

```

```

while (Nc < N) and Not Comx do
begin
  NC:= NC+1;
  Comx:= v in Cx[Nc]
end;
if Comx then
  NumCompon:= NC
else
  NumCompon:= 0
end;

procedure Prim (C: Matriz; var Ct: real);
  {C: Matriz de adyacencia. Si no existe arco (i,j), Cij tiene
  el valor infinito.
  Ct: Coste del árbol de expansión mínimo}
var
  Coste: array[1..n] of real;
  Mas_Cerca: array[1..n] of integer;
  i, j: integer;
  z : Indexnodo;
  Min : real;
  W : ConjVert;
begin
  Ct:= 0;
  {Vértice inicial es el 1}
  W:= [1];
  for i:= 2 to n do
  begin
    Coste[i]:= C[1,i]; {Inicialmente coste de cada vértice de V-[1]
    es el arco (1,i)}
    Mas_Cerca[i]:= 1 {debido a que W=[1]}
  end;
  for i:= 2 to n do
  begin
    {Encuentra el vértice z de V-W más cercano{menor arco a algún
    vértice de W}}
    Min:= Coste[2];
    z:= 2;
    for j:= 3 to n do
      if Coste[j]< Min then
      begin
        Min:= Coste[j];
        z:= j
      end;
    Ct:= Ct+ Min;
    {Escribe arco del árbol de expansión}
    writeln(z, ' -- ',Mas_cerca[z]);
    { El vértice z es añadido a W }
    Coste[z]:= Infinito;
    W:= W+[z];
    {Ajusta los costes del resto de vértices}
    for j:=2 to n do
      if (C[z,j]< Coste[j]) and not (j in W) then
      begin
        Coste[j]:= C[z,j];
        Mas_Cerca[j]:= z
      end;
  end;
end;

```

```

    end;
end;
end;

procedure Kruskal(C: matriz; var T: ListArcos; var Ct:real);
var
  Arcos: ListArcos;
  CompConx: ListConjs;
  u,v,
  C_u, C_v,
  Comp_n : IndexNodo;
  Kn,Ka,K: integer;
  Min_Ar: Arco;
begin
  Kn:= 0; { Número de arcos en T }
  Ka:= 0; {Número de arcos de la matriz de costes C}
  Ct:= 0; { Coste del árbol de expansión }
  Inicial(C,Arcos,Ka); {obtiene los Ka arcos en orden creciente de costes}
  for v:= 1 to n do
    CompConx[v]:= [v]; {Obtiene los n componentes conexos iniciales,
                         con cada vértice}
  Comp_n:= n; {Número de componentes conexos}
  K:= 0;
  while Comp_n> 1 do
    begin
      K:= K+1;
      Min_Ar:= Arcos[K]; { Arco mínimo actual }
      C_u:=NumCompon(Min_Ar.u,CompConx);
      {Componente donde se encuentra el vértice u del arco mínimo}
      C_v:= NumCompon(Min_Ar.v, CompConx);
      if C_u<>C_v then {Conecta dos componentes distintos}
      begin
        Ct:= Ct+ Min_Ar.Coste;
        writeln('Arco',Min_Ar.u,'→',Min_Ar.v);
        Combina(C_u,C_v,CompConx);{Une componentes u y v.
                                  Nueva componente queda en u)
        Comp_n:= Comp_n-1;
        Kn:= Kn+1;
        T[Kn]:= Min_Ar
      end
    end;
end;
begin
end.

```

Por último, el siguiente programa permite introducir los costes de los arcos de un grafo y con los algoritmos de Prim o de Kruskal obtener el árbol de expansión de coste mínimo.

```

program Costeo_minimo;
uses UniCamno, crt;
const
  Infinito = 1E+38
var

```

```

Cos: Matriz;
T: ListArcos;
N1, N2: Indexnodo;
CosteTot: real;

procedure grafo(var M: Matriz);
var
  N1, N2: Indexnodo;
  Fchero:string;
  F: text;
begin
  clrscr;
  writeln('Archivo con datos de grafo con ', n, 'Nodos');
  readln(Fchero);
  assign(F,Fchero);
  {$I-}
  reset(F);
  {$I+}
  if Ioreadlt<> 0 then
  begin
    writeln('Archivo no existe, o está en otra unidad');
    Halt
  end;
  while not eof(F) do
  begin
    readln(F,N1,N2,M[N1,N2]);
    M[N2,N1]:= M[N1,N2]
  end;
end;

begin
  {Los costes son inicializados a infinito}
  for N1:= 1 to Maxnodos do
    for N2:= 1 to Maxnodos do
      Cos[N1,N2]:=Infinito;

{Los datos del grafo se encuentran en un archivo.
En el procedimiento grafo se carga la matriz de costes}
  Grafo(Cos);
  write ('Árbol de expansión de coste mínimo, ');
  writeln('(Algoritmo de Prim) ');
  writeln;
  CosteTot:= 0;
  Prim(Cos, CosteTot);
  write('Coste del árbol de expansión mínimo: ');
  writeln(CosteTot:12:1);
  repeat until keypressed;
  writeln;
  write ('Árbol de expansión de coste mínimo, ');
  writeln'(Algoritmo de Kruskal) ';
  writeln;
  CosteTot:= 0;
  Kruskal(Cos, T, CosteTot);
  write('Coste del árbol de expansión mínimo: ');
  writeln(CosteTot:12:1)
end.

```

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 , tales que todos los vértices $V, W \in G$; 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*.

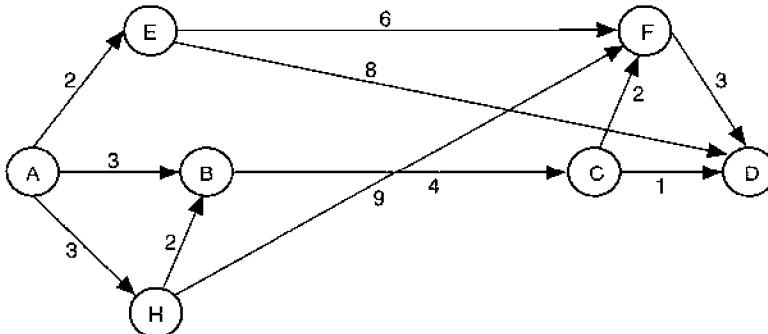
Dado un grafo G , una de las operaciones clave es la búsqueda de caminos mínimos que sean lo más cortos posible, donde la longitud de un camino, también conocida como *coste*, se define como la suma de las etiquetas de las aristas que lo componen. Una característica fundamental en un algoritmo es encontrar la distancia mínima de un vértice al resto y otro para encontrar la distancia mínima posible entre todo par de vértices.

Uno de los problemas más importantes es calcular el coste del camino mínimo desde un vértice al resto. Una de las soluciones más eficientes a este problema es el denominado *algoritmo de Dijkstra*. Asimismo, se trata de determinar el coste del camino más corto entre todo par de vértices de un grafo etiquetado y de esta manera se puede generalizar la situación estudiada. Además de *Dijkstra*, se dispone del llamado *algoritmo de Floyd*, que proporciona una solución más compacta y elegante.

Un *árbol de expansión* es un árbol que contiene a todos los vértices de una red y es una forma de averiguar si la red es conectada. Los algoritmos que resuelven los árboles de expansión mínima más usuales son: *algoritmo de Prim* y *algoritmo de Kruskal*.

EJERCICIOS

14.1. Dada la siguiente red:

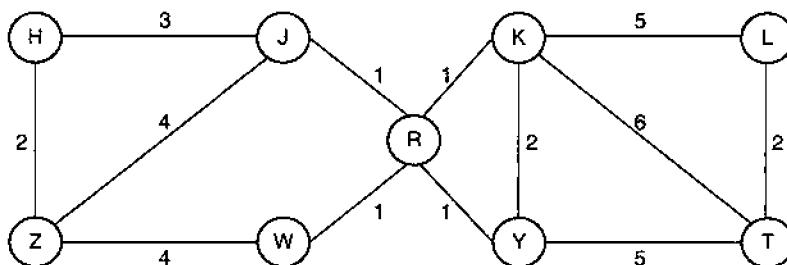


encontrar una ordenación topológica.

14.2. En la red del ejercicio 14.1 los arcos representan actividades y el factor de peso representa el tiempo necesario para realizar dicha actividad (un Pert). Cada vértice v de la red representa el tiempo que tardan todas las actividades representadas por los arcos que terminan en v . El ejercicio consiste en asignar a cada vértice v de la red 14.1 el tiempo necesario para que todas las actividades que terminan en v se puedan realizar, éste lo llamamos $tn(v)$. Una forma de hacerlo: asignar tiempo 0 a los vértices sin predecesores; si a todos los

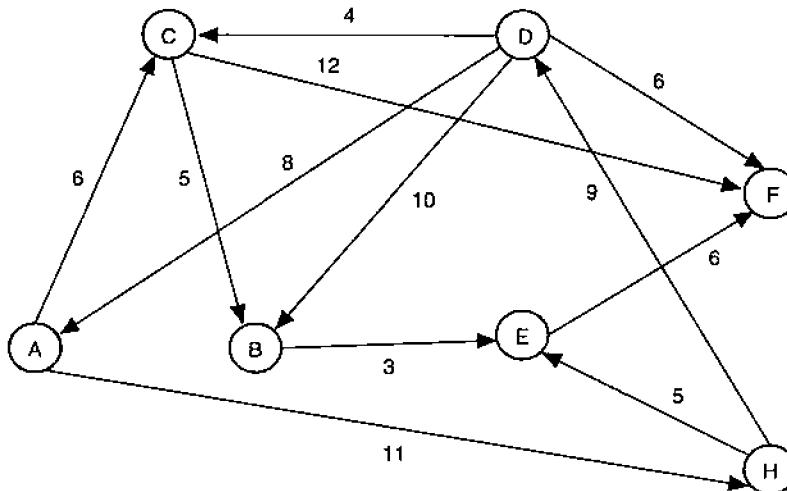
predecesores de un vértice v se les ha asignado tiempo, entonces $tn(v)$ es el máximo para cada predecesor de la suma del tiempo del predecesor con el factor de peso del arco desde ese predecesor hasta v .

- 14.3. Tomando de nuevo la red del ejercicio 14.1 y teniendo en cuenta el tiempo de cada vértice $tn(v)$ calculado en 14.2, ahora queremos calcular el tiempo límite en que todas las actividades que terminan en el vértice v pueden ser completadas sin atrasar la terminación de todas las actividades, a este tiempo lo llamamos $tl(v)$. Para lo cual podemos proceder: asignar $tn(v)$ a todos los vértices v sin sucesores. Si todos los sucesores de un vértice v tienen tiempo asignado, entonces $tl(v)$ es el mínimo de entre todos los sucesores de la diferencia entre el tiempo asignado al sucesor, $tl(v')$, y el factor de peso desde v hasta el sucesor v' .
- 14.4. Una ruta crítica de una red es un camino desde un vértice que no tiene predecesores hasta un vértice que no tiene sucesores, tal que para todo vértice v del camino se cumple que $tn(v) = tl(v)$. Encontrar las rutas críticas de la red del ejercicio 14.1.
- 14.5. Dado el grafo de la figura:



encontrar un árbol de expansión de coste mínimo con el algoritmo de Prim.

- 14.6. Dado el grafo del ejercicio 14.5, encontrar un árbol de expansión de coste mínimo con el algoritmo de Kruskal.
- 14.7. En el grafo dirigido con factor de peso de la figura:



encontrar el camino más corto desde el vértice A a todos los demás vértices del grafo.

- 14.8. En el grafo del ejercicio 14.7 encontrar el camino más corto desde el vértice D a todos los demás vértices del grafo siguiendo el algoritmo de Dijkstra. Incluir la ruta que forman los caminos.
- 14.9. En el grafo del ejercicio 14.7 encontrar los caminos más cortos entre todos los pares de vértices. Para ello, seguir paso a paso el algoritmo de Floyd.
- 14.10. Dibujar un grafo dirigido con factor de peso en el que algún arco tenga factor de peso negativo, de tal forma que al aplicar el algoritmo de Dijkstra para determinar los caminos más cortos desde un origen se obtenga un resultado erróneo.
- 14.11. Con el algoritmo de Dijkstra se calculan los caminos mínimos desde un vértice inicial a los demás vértices del grafo. Escribe las modificaciones necesarias para que teniendo como base el algoritmo de Dijkstra se calculen los caminos mínimos entre todos los pares de vértices. ¿Cuál es la eficiencia de este algoritmo?
- 14.12. El algoritmo de Prim o el algoritmo de Kruskal resuelven el problema de encontrar el árbol de expansión de coste mínimo. La cuestión es: ¿funcionan correctamente dichos algoritmos si el factor de peso es negativo?
- 14.13. El algoritmo de Dijkstra no se puede aplicar a un grafo valorado en el que alguna arista tiene un factor de peso negativo. El algoritmo de Bellman-Ford resuelve el problema cuando hay aristas negativas, aunque no lo resuelve cuando hay ciclos en el grafo con peso negativo. En caso de haber un ciclo con peso negativo el algoritmo lo indica y termina.

Sea $G = (V, A)$ un grafo valorado de n vértices, consideramos el vértice origen el 1 y P la matriz de pesos, el algoritmo de Bellman-Ford para hallar los caminos mínimos desde un origen lo expresamos en pseudocódigo:

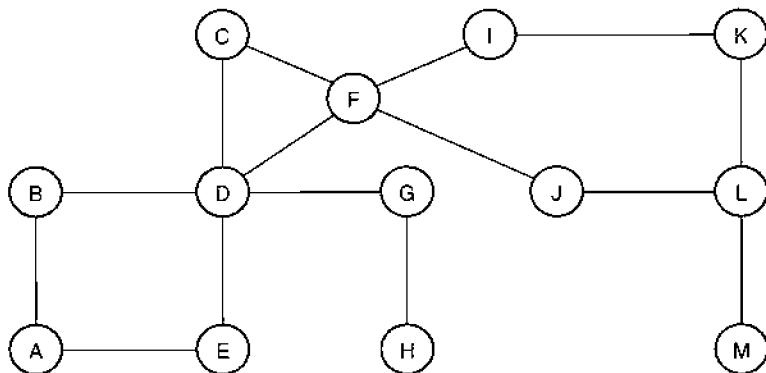
```

Inicio
  desde v ← 2 hasta n { n es el número de nodos } hacer
    d(v) ← ∞
  fin_desde
  s ← 1
  d(s) ← 0 { marca el origen del camino }
  desde i ← 1 hasta n - 1 { n es el número de nodos } hacer
    para cada arista (u,v) ∈ A 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 peso negativo }
    para cada arista (u,v) ∈ A hacer
      si d(v) > d(u) + P(u,v), entonces
        Error ← true
      fin_si
    { en el array d están los caminos mínimos desde s a los demás
      vértices si Error es false }
  Fin

```

Demostrar el funcionamiento del algoritmo de Bellman-Ford dibujando dos grafos con factor de peso, y alguna arista negativa. En uno de ellos ha de haber un ciclo con peso negativo.

- 14.14. Determinar los vértices que son puntos de articulación en el grafo de la figura:



- 14.15. 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 todo vértice tiene iguales su grado de entrada y de salida.

Dibujar un grafo dirigido en el que se pueda encontrar un circuito de Euler.

PROBLEMAS

- 14.1. Escribe un programa en el cual sea representado en memoria una red (un Pert: grafo dirigido sin ciclos y factor de peso) mediante la matriz de adyacencia (matriz de pesos) y calcule:

- El tiempo $tn(v)$.
- El tiempo $tl(v)$.

Además encuentre las rutas críticas de la red.

(Nota: véanse los ejercicios 14.2-14.4.)

- 14.2. Escribir un programa en el que dada una red (un Pert), representada en memoria mediante listas de adyacencia, encuentre las rutas críticas.
- 14.3. Tenemos una red (un Pert) representada con su matriz de adyacencia. Escribir un programa que calcule el mínimo tiempo en el que todo trabajo se puede terminar si tantas actividades como sea posible son realizadas en paralelo. El programa debe de escribir el tiempo en el que se inicia y se termina toda actividad en la red.
- 14.4. Escribir un programa para que dada una red (Pert) representada por su matriz de adyacencia, determine el tiempo mínimo de realización del trabajo si como máximo se pueden realizar n actividades (de las posibles) en paralelo. El programa debe mostrar el tiempo de inicio y el de finalización de cada actividad.
- 14.5. Escribir un procedimiento para implementar el algoritmo de Dijkstra con esta modificación: en la búsqueda del camino mínimo desde el origen a cualquier vértice puede haber más de un camino del mismo camino mínimo, entonces seleccionar aquel con el menor número de arcos.

- 14.6. El algoritmo de Dijkstra resuelve el problema de hallar los caminos mínimos desde un único vértice origen a los demás vértices. Escribir un procedimiento para resolver el problema de que dado un grafo G representado por su matriz de pesos encuentre los caminos mínimos desde todo vértice V a un mismo vértice destino D.
- 14.7. Escribir un programa en el que dado un grafo valorado, teniendo ciertos factores de peso negativos, representado en memoria mediante la matriz de pesos, determine los caminos más cortos desde el vértice origen a los demás vértices. Utilizar el algoritmo de Bellman-Ford del ejercicio 14.13.
- 14.8. Dado un grafo no dirigido con factor de peso escribir un programa que tenga como entrada dicho grafo, lo represente en memoria y determine el árbol de expansión de coste máximo.
- 14.9. 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 todo vértice tiene iguales sus grados de entrada y de salida. Escribir un programa en el que se represente mediante listas de adyacencia un grafo dirigido. Implemente un algoritmo para encontrar, si existe, un circuito de Euler.
- 14.10. Un grafo está representado en memoria mediante listas de adyacencia. Escribir las rutinas necesarias para determinar la matriz de caminos.
(Nota: Seguir la estrategia expuesta en el algoritmo de Warshall pero sin utilizar la matriz de adyacencia.)
- 14.11. Se quiere escolarizar una zona rural compuesta de 4 poblaciones: Lupiana, Centenera, Atanzón y Pinilla. Para ello se va a construir un centro escolar en la población que mejor coste de desplazamiento educativo tenga (mínimo de la función Z_i).

$Z_i = \sum p_j d_{ij}$; donde p_j es la población escolar de la población j y d_{ij} es la distancia mínima del pueblo j al pueblo i .

Las distancias entre los pueblos en kilómetros:

Lup	Cen	Atn	Pin
Lup	7	11	4
Cen		5	12
Atn			8

La población escolar de cada pueblo: 28, 12, 24, 8, respectivamente de Lupiana, Centenera, Atanzón y Pinilla.

Codificar un programa que tenga como entrada los datos expuestos y determine la población donde conviene situar el centro escolar.

- 14.12. Un grafo G representa una red de centros de distribución. Cada centro dispone de una serie de artículos y un stock de ellos, representado mediante una estructura lineal ordenada respecto al código del artículo. Los centros están conectados aunque no necesariamente bidireccionalmente. Escribir un programa en el que se represente el grafo como un grafo dirigido ponderado (el factor de peso que represente la distancia en kilómetros entre dos centros). En el programa debe de estar la opción de que un centro H no tenga el artículo z y lo requiera, entonces el centro más cercano que disponga de z se lo suministra.
- 14.13. Europa consta de n capitales de cada uno de sus estados, cada par de ciudades está conectada o no por vía aérea. En caso de estar conectadas (se entiende por vuelo directo) se sabe las millas de la conexión y el precio del vuelo. Las conexiones no tienen por qué ser bidireccionales, así puede haber vuelo Viena-Roma y no Roma-Viena.

- Escribir un programa que represente la estructura expuesta y resuelva el problema: disponemos de una cantidad de dinero D , deseamos realizar un viaje entre dos capitales, $C1-C2$, y queremos información sobre la ruta más corta que se ajuste a nuestro bolsillo.
- 14.14.** La ciudad dormitorio de Martufa está conectada a través de una red de carreteras, que pasa por poblaciones intermedias, con el centro de la gran ciudad. Cada conexión entre dos nodos soporta un número máximo de vehículos a la hora. Escribir un programa para simular la salida de vehículos de la ciudad dormitorio y llegada por las diversas conexiones al centro de la ciudad. La entrada de datos ha de ser los nodos de que consta la red (incluyendo ciudad-dormitorio y centro-ciudad) y la capacidad de cada conexión entre dos nodos. El programa de calcular el máximo de vehículos/hora que pueden llegar al centro y cómo se distribuyen por las distintas calzadas.

PARTE **IV**

Archivos y ordenación

Ordenación, búsqueda y mezcla

CONTENIDO

- 15.1. Introducción.
- 15.2. Ordenación.
- 15.3. Ordenación por burbuja.
- 15.4. Ordenación por selección.
- 15.5. Ordenación por inserción.
- 15.6. Ordenación *Shell*.
- 15.7. Ordenación rápida (*quicksort*).
- 15.8. Ordenación por mezcla (*mergesort*).
- 15.9. Ordenación *Heapsort*.
- 15.10. Ordenación *Binsort*.
- 15.11. Ordenación Radix Sort.
- 15.12. Búsqueda lineal.
- 15.13. Búsqueda binaria.
- 15.14. Búsqueda binaria recursiva.
- 15.15. Mezcla.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Las computadoras emplean gran parte de su tiempo en operaciones de ordenación, de búsqueda y de mezcla. Los arrays (vectores y tablas) se utilizan con mucha frecuencia para almacenar datos, por ello los algoritmos para el diseño de estas operaciones son fundamentales y se denominan a las operaciones internas, debido a que los arrays guardan sus datos de modo temporal en memoria interna y desaparecen cuando se apaga la computadora.

Los métodos de ordenación, búsqueda y mezcla son numerosos; en este capítulo se consideran algunos de los más eficientes. Su importancia reside en el hecho de que su análisis y algoritmos servirán también, en gran medida, para realizar las mismas operaciones con registros, archivos y estructuras dinámicas de datos.

15.1. INTRODUCCIÓN

Tres operaciones muy importantes en programación de computadoras son: *ordenación*, *búsqueda* y *mezcla*; son esenciales para un gran número de programas de proceso de datos y se estima que en estas operaciones las computadoras por término medio gastan gran parte de su tiempo. La búsqueda, ordenación y mezcla son también procesos que las personas se encuentran normalmente en sus vidas diarias. Considérese, por ejemplo, el proceso de encontrar una palabra en un diccionario o un nombre en una guía o listado de teléfonos. La *búsqueda* de un elemento específico se simplifica considerablemente por el hecho de que las palabras en el diccionario y los nombres en la guía telefónica están *ordenados* o *clasificados* en orden alfabético. Asimismo, la operación de mezclar datos de dos listas o conjuntos de datos en una sola lista suele ser una operación frecuente.

En este capítulo se estudian los métodos más usuales de ordenación, búsqueda y mezcla, relativos a listas o vectores (arrays), ya que si bien estas operaciones se aplican sobre otras estructuras de datos como registros o archivos, su mayor aplicación está casi siempre asociada a los vectores, y, por otra parte, la comprensión de los algoritmos en vectores —extrapolables a otras estructuras— es más fácil.

15.2. 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* 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:

$$i < j \text{ implica que } K[i] \leq K[j]$$

y se dice que está en *orden descendente* si:

$$i > j \text{ implica que } K[i] \geq K[j]$$

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 de Turbo Pascal—, 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*

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.

15.3. ORDENACIÓN POR BURBUJA

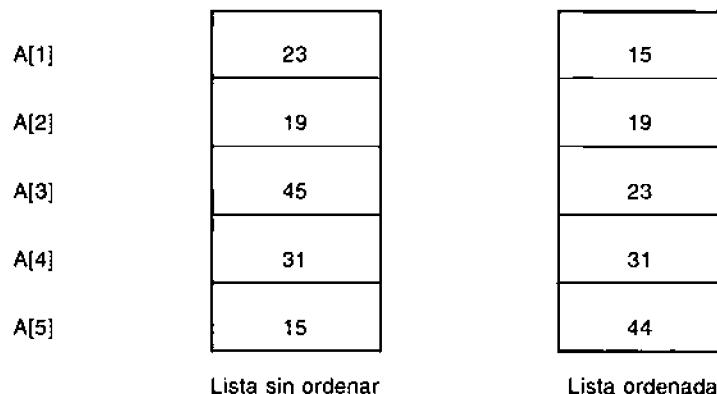
Este método es clásico y muy sencillo, aunque por desgracia poco eficiente. La **ordenación por burbuja** («*bubble sort*») se basa en *comparar elementos* adyacentes de la lista (vector) e intercambiar sus valores si están desordenados. De este modo se dice que los valores más pequeños *burbujean* hacia la parte superior de la lista (hacia el primer elemento), mientras que los valores más grandes se *hunden* hacia el fondo de la lista.

Consideremos, como ya se ha expuesto, clasificaciones en orden creciente.

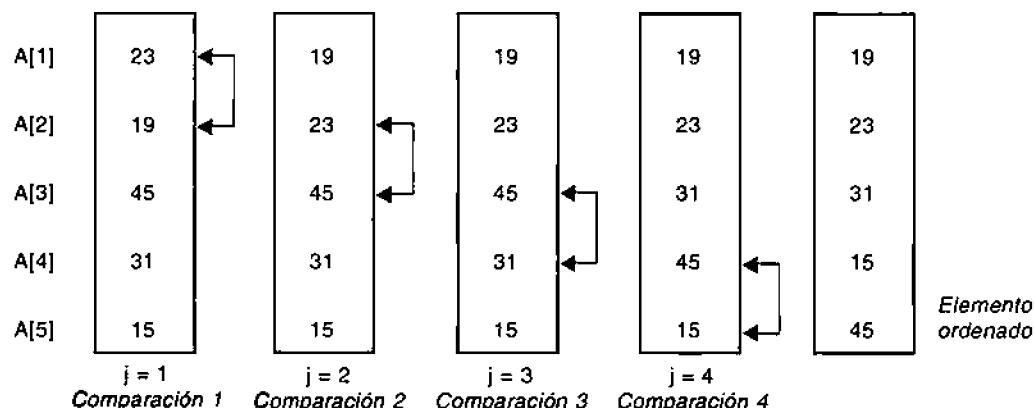
15.3.1. Análisis

Supongamos un vector $A[1], A[2], \dots, A[n]$. Se comienza el seguimiento del vector de izquierda a derecha, comparando $A[1]$ con $A[2]$; si están desordenados, se intercambian

entre sí. A continuación se compara $A[2]$ con $A[3]$, intercambiándolos si están desordenados. Este proceso de comparaciones e intercambios continúa a lo largo de toda la lista. Estas operaciones constituyen una *pasada* a través de la lista. Al terminar esta pasada el elemento mayor está en la parte inferior de la lista y alguno de los elementos más pequeños ha burbujeado hacia arriba de la lista. Se vuelve a explorar de nuevo la lista, comparando elementos consecutivos e intercambiándolos cuando estén desordenados, pero esta vez el elemento mayor no se compara, ya que se encuentra en su posición correcta. Se siguen las comparaciones hasta que toda la lista está ordenada, cosa que sucederá cuando se hayan realizado $(n - 1)$ pasadas. Para su mejor comprensión, veamos gráficamente el proceso anterior con un vector (lista) de cinco elementos: $A[1], A[2], A[3], A[4], A[5]$.

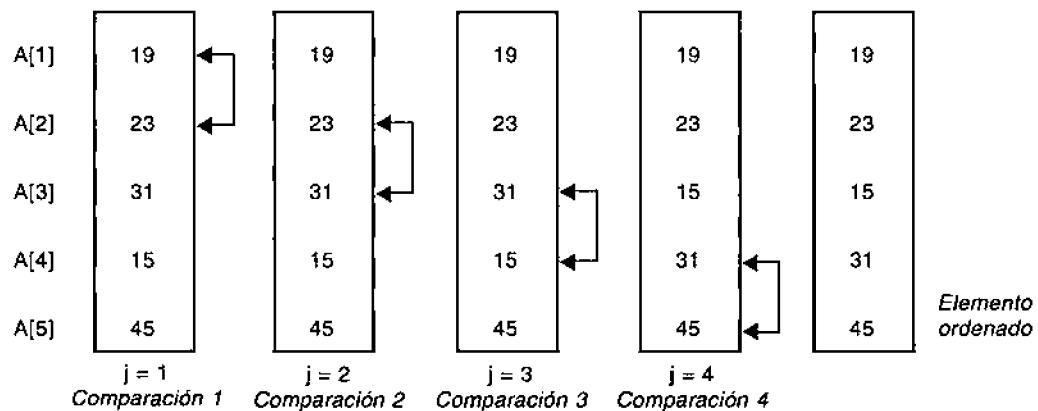


En la lista A , i será el número de la pasada y j indica el orden del elemento de la lista. Se comenzará en el elemento j -ésimo y el $(j + 1)$ -ésimo.

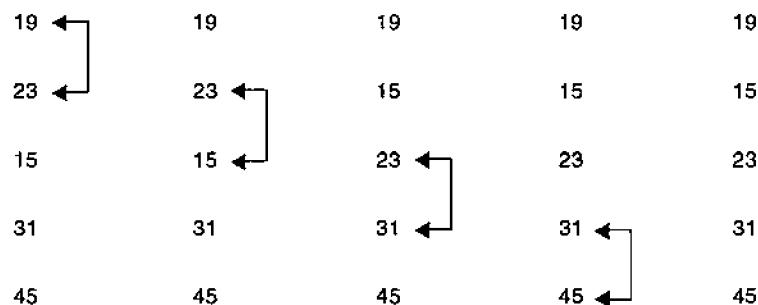
Pasada 1: $i = 1$ 

Se han realizado cuatro comparaciones ($5 - 1$ o bien $n - 1$, en el caso de n elementos) y tres intercambios (rotulados por el símbolo ↪).

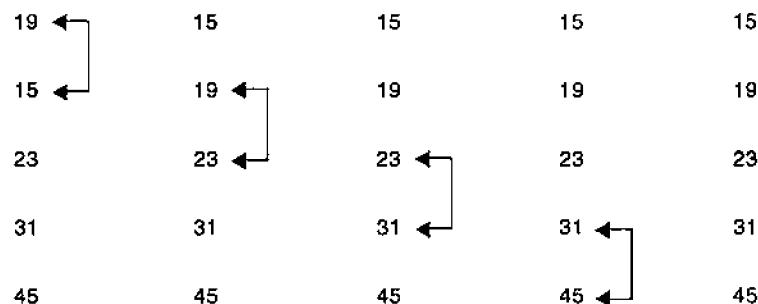
Pasada 2: $i = 2$



Pasada 3: $i = 3$



Pasada 4: $i = 4$



Se observa que se necesitan cuatro pasadas para ordenar una lista de números de cinco elementos, por lo que una lista de n elementos necesitará $n-1$ pasadas. El proceso se describe así:

1. Realizar cuatro pasadas por la lista: $i = 1, 2, 3, 4$.
2. Para la pasada 1 ($i = 1$) se realizan comparaciones ($j = 1, 2, 3, 4$). $A[1]$ con $A[2]$, $A[2]$ con $A[3]$, etc.
3. Para la pasada 2 ($i = 2$) se realizan 3 comparaciones ($j = 1, 2, 3$).
4. Para la pasada 3 ($i = 3$) se realizan 2 comparaciones ($j = 1, 2$).
5. Para la pasada 4 ($i = 4$) se realizan 1(5-4) comparaciones ($j = 1$):

El número de pasadas (4 o bien $n - 1$) se puede controlar con un bucle **for**, y cada secuencia de comparaciones se puede controlar con un bucle **for** anidado al bucle de pasadas, en el que j varía desde 1 hasta 5 menos el valor específico de i

<i>bucle externo i</i>	$i=1$	$j=1, 2, 3, 4$	$5-i=4$	<i>bucle interno (n-i)</i>
	$i=2$	$j=1, 2, 3$	$5-i=3$	
	$i=3$	$j=1, 2$	$5-i=2$	
	$i=4$	$j=1$	$5-i=1$	

Por consiguiente, el bucle **for** que controla cada pasada será:

```
for j = 1 to 5-i
```

Algoritmo (pseudocódigo)

```
desde i ← 1 hasta n-1 hacer
  desde j ← i hasta n-i hacer
    si A[j] > A[j+1]
      entonces Intercambio (A[j], A[j+1])
    fin-si
  fin_desde {bucle j}
fin_desde {bucle i}
```

La operación de intercambio se realiza con las instrucciones

```
Aux ← A[j]
A[j] ← A[j+1]
A[j+1] ← Aux
```

o mejor modularizando esta operación con un procedimiento Intercambio que recibe como parámetro de entrada los dos valores a intercambiar y devuelve al procedimiento llamador los dos valores ya intercambiados como parámetros de salida.

```
procedure Intercambio (var A, B ; integer);
var
  Aux : integer;
begin
  Aux := A;
  A := B;
  B := Aux
end;
```

15.3.2. Procedimientos de ordenación

Método 1

```

procedure burbuja1 (var A: Lista; N : integer);
{ordenar A[1],...,A[N] en orden ascendente
Lista en un array unidimensional definido en el programa principal de N
elementos de tipo entero}
var
  I, J, Aux : integer;
begin
  for I := 1 to N -1 do
    for J := 1 to N - I do
      if A[J] > A [J+1] then
        begin {intercambio dentro del programa}
          Aux      := A [J];
          A [J]     := A [J+1];
          A[J+1]   := Aux;
        end; {fin de los bucles for}
  end; {Burbuja1}

```

Método 2

```

procedure Burbuja1 (var A : Lista; N : integer);
var
  I, J : integer;

procedure Intercambio (var X, Y : integer);
var
  Aux : integer;
begin
  Aux := X;
  X   := Y;
  Y   := Aux
end;

begin
  for I := 1 to N-1 do
    for J := 1 to N-I do
      if A[J] > A [J+1] then
        Intercambio (A[J], A[J+1]);
  end; {Burbuja1}

```

15.3.3. Algoritmo de burbuja mejorado (refinamiento)

La técnica de ordenación por burbuja compara elementos consecutivos de la lista, de modo que si en una pasada no ocurrieran intercambios, significaría que la lista está ordenada. El algoritmo burbuja se puede mejorar si disponemos de algún tipo de indicador que registre si se han producido intercambios en la pasada. Cuando se explore la lista y el indicador no refleje intercambios, la lista estará ya ordenada y se terminarán las comparaciones.

El indicador será una variable lógica `NoIntercambio` (o bien ordenado) que se inicializa a «verdadero» (`true`) (significa que la lista a priori está ordenada). Si dos elementos se intercambian en una pasada, `NoIntercambio` se pone a `false`. Al principio de cada pasada `NoIntercambio` se fija a `true` y se pone a `false` si se producen intercambios. El bucle externo `for` se sustituye por un bucle `repeat-until` o bien `while-do` y un contador `i` se necesitará para contar el número de pasadas.

Pseudocódigo BurbujaMejorado

```
i ← 1
repetir
    NoIntercambio ← true
    desde j ← 1 hasta n-i hacer
        si A[j] > A [j+1]
            entonces Intercambio (A[j], A[j+1])
            NoIntercambio ← false
        fin_si
    fin_desde
    i ← i+1
hasta_que NoIntercambio = true
```

15.3.4. Programación completa de ordenación por burbuja

Se genera aleatoriamente una lista de 100 números enteros (o bien se leen de un archivo de entrada: teclado o disco) y se desea escribir un programa que realice las siguientes tareas:

1. Leer lista de números aleatorios.
2. Visualizar lista.
3. Ordenar lista por burbuja.
4. Visualizar lista ordenada.

El procedimiento `Leer` sirve para introducir la lista de 100 números; el procedimiento `Visualizar` permite imprimir cualquier lista, en este caso tanto la lista ordenada como desordenada; y el procedimiento `Ordenar` clasifica la lista `A`.

```
program OrdenarBurbuja;
{Ordenación ascendente por el método de la burbuja}
const
  Limite = 100;
type
  Item = integer;
  rango = 0..Limite;
  Lista = array [Rango] of Item;
var
  ListaItem : Lista;
  Numitems : integer;
procedure Leer (var A: Lista; N: integer);
var
  I : integer;
begin
  for I := 1 to N do
    A [I] := Random (1000)
end;
```

```

procedure Escribir (var A: Lista; N : integer);
var
  I : integer;
begin
  for I := 1 to N do
    Write (A[I] : 4);
  Writeln
end;

procedure Burbuja (var A : Lista; N : integer);
var
  I, J : integer;
  {procedimiento de intercambio}
  procedure Intercambio (var A,B : item);
  var
    Aux : item;
  begin
    Aux := A;
    A := B;
    B := Aux
  end;
begin
  for I:= 1 to N-1 do
    for J:= I +1 to N do
      if A[J-1] > A[J] then
        Intercambio (A[J-1],A[J])
end;

{programa principal}
begin
  Leer(ListaItem, Limite);
  Writeln('La lista original es');
  Escribir(ListaItem, Limite);
  Burbuja(ListaItem, Limite);
  Writeln('La lista ordenada es');
  Escribir(ListaItem, Limite)
end.

```

15.3.5. Análisis de la ordenación por burbuja

Este algoritmo proporciona buen rendimiento en cuanto a su sencillez, pero por el contrario su eficiencia es pobre. Para una lista de n elementos, el proceso de ordenación requiere $n - 1$ pasadas y el número de comparaciones se refleja en la tabla siguiente:

<i>Pasada</i>	<i>Comparaciones</i>
1	$n - 1$
2	$n - 2$
3	$n - 3$
.	.
.	.
$n - 1$	1

El número total de comparaciones es

$$1 + 2 + 3 + \dots + (n - 3) + (n - 2) + (n - 1) = \frac{n^2 - n}{2} = \frac{1}{2}(n^2 - n)$$

La función de eficiencia de rendimiento de un algoritmo se representa con la función $O(n)$, también llamada *Notación de O-grande*. En el caso de la burbuja, en el peor de los casos, el número de comparaciones es $O(n^2)$.

El algoritmo de burbuja es una ordenación cuadrática, lo que significa elevado número de comparaciones y, por consiguiente, excesivo tiempo de ejecución, o dicho de otro modo: es un *algoritmo lento*.

15.4. ORDENACIÓN POR SELECCIÓN

El algoritmo de ordenación por selección de una lista (vector) de n elementos tiene los siguientes pasos:

1. Encontrar el elemento mayor de la lista.
2. Intercambiar el elemento mayor con el elemento de subíndice n (o bien si es el elemento menor con el subíndice 1).
3. A continuación se busca el elemento mayor en la sublista de subíndices $1..n-1$, y se intercambia con el elemento de subíndice $n-1$; por consiguiente, se sitúa el segundo elemento mayor en la posición $n-1$.
4. A continuación se busca el elemento mayor en la sublista $1..n-2$, y así sucesivamente.

Algoritmo

Desde $j \leftarrow n$ hasta 2 [decremento - 1] hacer:

- Encontrar el elemento mayor en el array $1..j$.
- Si el elemento mayor no está en el subíndice j , entonces intercambiar elemento mayor con el de subíndice i .

El algoritmo de PosMayor debe guardar j como la posición del elemento mayor y luego poder intercambiar.

```
program OrdenarSeleccion;
{leer 100 enteros. Ordenar. Visualizar}
const
  Limite = 100;
type
  Lista = array [1..Limite] of integer;
```

```

var
  I, Num : 1..Limite;
  A : Lista;
function PosMayor (Ultimo:integer; var Tabla: Lista):integer;
{encuentra el indice del elemento mayor en la Tabla [1..Ultimo]}
var
  Indice_Max, Indice : 1..Limite;
begin
  Indice_max := 1;
  for Indice := 2 to Ultimo do
    if Tabla [Indice] > Tabla [Indice_Max]
      then Indice_Max := Indice;
  PosMayor := Indice_Max
end;

procedure Selección (Limi : integer; var Tabla : Lista);
var
  Aux, J, Mayor : integer;
begin
  for J := Limi downto 2 do
    begin
      {encontrar el Elemento mayor de 1..J}
      Mayor := PosMayor (J, Tabla);
      {intercambio con el Elemento Tabla [J]}
      Aux := Tabla [Mayor];
      Tabla [Mayor] := Tabla [J];
      Tabla [J] := Aux
    end
  end;
{programa principal}
begin
  for I := 1 to Limite do
    begin
      A [I] := Random (100);
      Write (A[I] : 4)
    end;
  WriteLn;
  Selección (Limite, A);
  for I := 1 to Limite do
    Write (A[I] : 4);
  WriteLn
end.

```

Análisis de la ordenación por selección

Número de comparaciones por cada una de las pasadas.

<i>Pasada</i>	<i>Número de comparaciones</i>
1	$n - 1$
2	$n - 2$
3	$n - 3$
.	.
.	.
$n - 1$	1

El número total de comparaciones es

$$1 + 2 + 3 + \dots + n - 2 + n - 1 = \frac{n \cdot (n - 1)}{2} = \frac{1}{2}(n^2 - n)$$

La eficiencia, como se observa, es similar al método de la burbuja.

15.5. ORDENACIÓN POR INSERCIÓN

Este método está basado en la técnica utilizada por los jugadores de cartas para clasificar sus cartas. El jugador va colocando (insertando) cada carta en su posición correcta.



El método se basa en considerar una parte de la lista ya ordenada y situar cada uno de los elementos restantes insertándolo en el lugar que le corresponde por su valor, todos los valores a la derecha se desplazan una posición para dejar espacio.

Algoritmo

```
{para cada elemento de la lista después del primero}
desde k ← 2 hasta n hacer
```

- Guardar el valor de ese elemento A[k] en una variable Aux.
- Hacer espacio para Aux desplazando todos los valores mayores que dicho valor A[k] una posición.
- Insertar el valor de Aux en el lugar del último valor desplazado.

```
fin_desde
```

La operación de desplazamiento se realiza con un procedimiento Desplazar, que mueve todos los elementos de la lista mayores que Aux, comenzando con el elemento de la lista de posición Aux-1. Si Aux es el valor más pequeño, la operación de desplazamiento termina cuando un valor menor o igual a Aux se alcanza. Aux se inserta en la posición que ocupaba el último valor que se desplazó.

Algoritmo de desplazamiento

```

mientras el primer elemento no se desplaza
y valor del elemento > Aux hacer
  • Desplazar elemento una posición.
  • Comprobar valor del siguiente elemento.
  • Definir NuevaPos como posición original del último elemento
    desplazado
fin_mientras

```

Codificación del procedimiento OrdenarInsercion

```

procedure OrdenacionInversa (var Tabla: Lista; N : integer);
{Tabla (entrada/salida) , N (entrada)}
{Lista = array de N elementos enteros}
var
  K : integer; {subíndice del siguiente elemento al que se inserta}
  NuevaPos : integer; {subíndice de este elemento después de la inserción}
  Aux : integer;

begin
  for K := 2 to N do
    begin
      Aux := Tabla [K]; {obtener siguiente elemento a insertar}
      {desplazar todos los valores > Aux un elemento}
      Desplazar (Tabla, K, Aux, NuevaPos);
      {insertar Aux en posición NuevaPos}
      Tabla [NuevaPos] := Aux
    end
  end;

```

El procedimiento Desplazar es

```

procedure Desplazar (var Tabla : Lista; Aux, K : integer;
                      var NuevaPos : integer);
var
  Encontrado : boolean; {indicador}
begin
  {desplazar valores > Aux . Comenzar con el elemento K-1}
  Encontrado := false;
  while (K >1) and not Encontrado do
    if (Tabla [K-1] > Aux) then
      begin
        Tabla [K] := Tabla [K-1];
        K           := K-1
      end
    else
      Encontrado := true;
  NuevaPos := K
end; {Desplazar}

```

15.5.1. Algoritmo de inserción binaria

El análisis de la ordenación por inserción es un poco más complicado. El número de comparaciones en i -ésimo paso es como máximo $k - 1$ y como mínimo 1. Por consiguiente, la media proporciona el número de comparaciones.

$$\lceil (k - 1) / 2 \rceil = k/2$$

El número de comparaciones (C) es

$$C_{\max} = \sum_{i=2}^n (i - 1) = 1 + 2 + \dots + (n - 1) = \frac{n^2 - n}{2} = \frac{1}{2} (n^2 - n)$$

$$C_{\min} = \sum_{i=2}^n (1) = 1 + 1 + \dots + 1 = (n - 1)$$

$$C_{\text{media}} = (C_{\max} + C_{\min})/2 = \left(\frac{1}{2} (n^2 - n) + (n - 1) \right)/2 = \frac{1}{4}(n^2 + n - 2)$$

Como se observa, la eficiencia es $O(n^2)$.

15.5.2. Comparación de ordenaciones cuadráticas

	<i>Caso favorable</i>	<i>Caso desfavorable</i>
Selección	$O(n^2)$	$O(n^2)$
Burbuja	$O(n)$	$O(n^2)$
Inserción	$O(n)$	$O(n^2)$

Como el tiempo requerido para ordenar un array (vector) de n elementos es proporcional a n^2 , ninguno de estos algoritmos es particularmente bueno para arrays grandes ($n \geq 100$). Para listas de mayor número de elementos, los métodos avanzados son los más idóneos ya que su eficiencia en lugar de depender de n^2 depende de $n \times \log_2 n$, lo que reduce considerablemente el tiempo de ejecución.

15.6. ORDENACIÓN SHELL

La ordenación Shell debe el nombre a su inventor, D. L. Shell [CACM 2 (julio, 1959), 30-32]. Se suele denominar también *ordenación por disminución de incremento (gap)*. La idea general del método (algoritmo) es la siguiente:

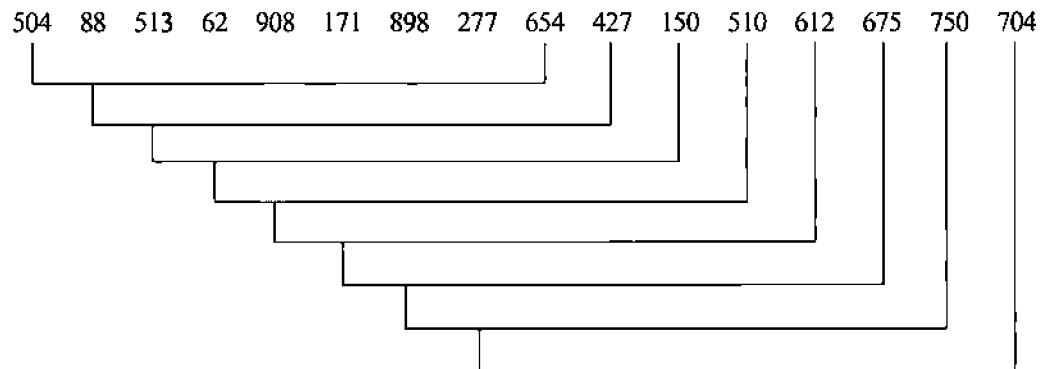
Lista original

504 88 513 62 908 171 898 277 654 427 150 510 612 675 750 704

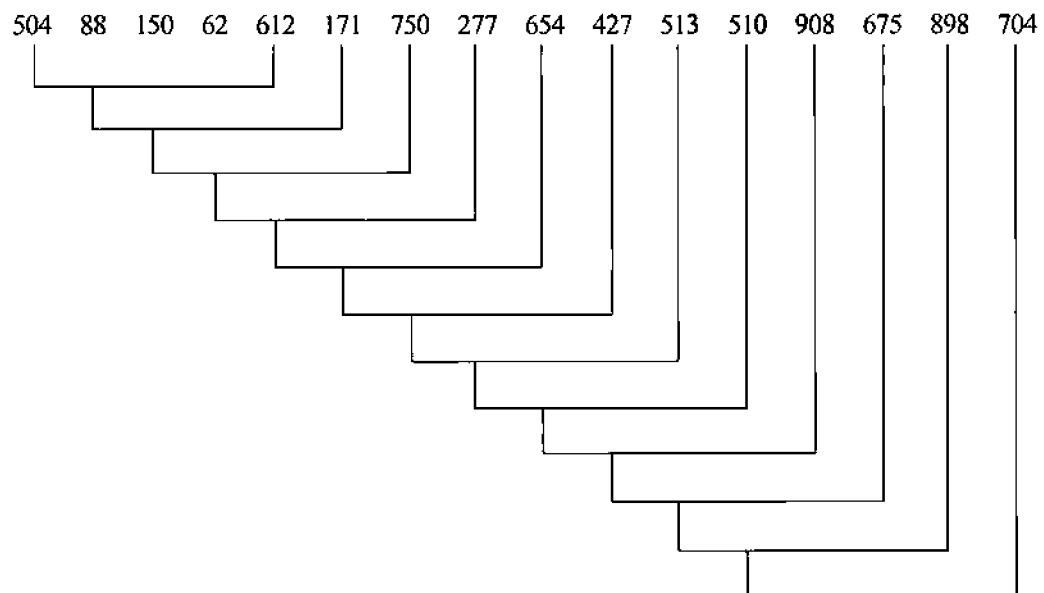
1. Se divide la lista original (16 elementos, en este ejemplo) en ocho grupos de dos (considerando un incremento o intervalo de $16/2 = 8$).

2. Se clasifica cada grupo por separado (se comparan las parejas de elementos y si no están ordenados se intercambian entre sí de posiciones).
3. Se divide ahora la lista en cuatro grupos de cuatro (intervalo o salto de $8/2 = 4$) y nuevamente se clasifica cada grupo por separado.
4. Un tercer paso clasifica dos grupos de ocho registros y luego un cuarto paso completa el trabajo clasificando los 16 registros.

Primer paso (división/ordenación por 8)



Segundo paso (división/ordenación por 4)



Tercer paso (división/ordenación por 2)

504 88 150 62 612 171 513 277 654 427 750 510 908 675 898 704

Cuarto paso (división/ordenación por 1)

150 62 504 88 513 171 612 277 654 427 750 510 898 675 908 704
62 88 154 171 277 427 504 510 513 612 654 675 704 760 898 908

El algoritmo de Shell tiene diferentes modelos; recogemos en este libro uno de los más populares y citados en numerosas obras de programación.

Algoritmo

```

intervalo ← n div 2
mientras (intervalo > 0) hacer
    desde i ← (intervalo + 1) hasta n hacer
        j ← i-intervalo
        mientras (j>0) hacer
            k ← j + intervalo
            si a[j] <= a[k]
                entonces
                    j ← 0
                sino
                    Intercambio (a[j], a[k]);
            fin_si
            j ← j-intervalo
        fin_mientras
    fin_desde
    intervalo ← intervalo div 2
fin_mientras

program OrdenarShell;
{modelo de ordenación de 500 enteros aleatorios}
const
    NumElementos = 500;
type
    Rango = 1..NumElementos;
    Lista = array [Rango] of integer;
var
    L : Lista;

procedure GenerarAleatorios (var A:Lista; Elementos: integer);
var
    I : integer;
begin
    Randomize;
    for I := 1 to Elementos do
        A [I] := Random (1000)
end;

```

```

procedure Visualizar (var A:Lista; Elementos:integer);
var
  I : integer;
begin
  for I := 1 to Elementos do
    Write (A[I] :6, '');
  WriteLn
end;

procedure Intercambio (var X, Y : integer);
var
  Aux : integer;
begin
  Aux := X;
  X := Y;
  Y := Aux
end;

procedure Shell (var A :Lista; N : integer);
var
  Intervalo, I, J, K : integer;
begin
  Intervalo := N div 2;
  while Intervalo > 0 do
    begin
      for I := (Intervalo + 1) to N do
        begin
          J := I - Intervalo;
          while (J > 0) do
            begin
              K := J + Intervalo;
              if A [J] <= A [K]
                then
                  J := 0
                else
                  Intercambio (A[J], A[K]);
              J := J - Intervalo
            end {while}
        end;
      Intervalo := Intervalo div 2
    end
end;

begin
  {programa principal}
  WriteLn ('Comienza la ordenación');
  GenerarAleatorios (L, NumElementos);
  Shell (L, NumElementos);
  Visualizar (L, NumElementos)
end.

```

15.7. ORDENACIÓN RÁPIDA (QUICKSORT)

Uno de los métodos más rápidos y más frecuentemente utilizado en ordenación de arrays es el conocido como ordenación rápida (*Quicksort*). Fue inventado por C. H.

Hoare, y la cantidad de código necesario es sorprendentemente pequeño comparado con la excelente velocidad que proporciona.

La idea básica de la ordenación rápida de un array (lista) es:

- Elegir un elemento del array denominado *pivote*.
- Dividir o partir el array original en dos subarrays o mitades (sublistas), de modo que en una de ellas estén todos los elementos menores que el pivote y en la otra sublista todos los elementos mayores que el pivote.
- Las sublistas deben ser ordenadas, independientemente, del mismo modo, lo que conduce a un algoritmo recursivo.

La elección del pivote es arbitraria, aunque por comodidad es usual utilizar el término central de la lista original, o bien el primero o último elemento de la misma.

Como ejemplo ilustrativo de la división de una lista en dos sublistas, consideremos la siguiente línea de enteros:

9 23 31 17 21 19 13 15 26

1. Elijamos el elemento pivote; supongamos el término central, 21.

pivote

9	23	31	17	21	19	13	15	26
---	----	----	----	----	----	----	----	----

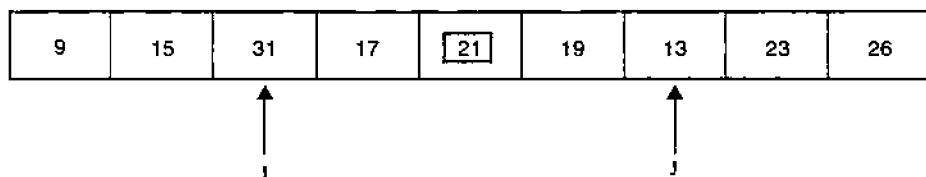
2. A continuación se establecen dos punteros en el array I o J. El primer puntero apunta al primer elemento. Por consiguiente, $I = 1$. El segundo puntero apunta al último elemento y, por tanto, $J = 9$ (noveno elemento):

9	23	31	17	21	19	13	15	26
---	----	----	----	----	----	----	----	----

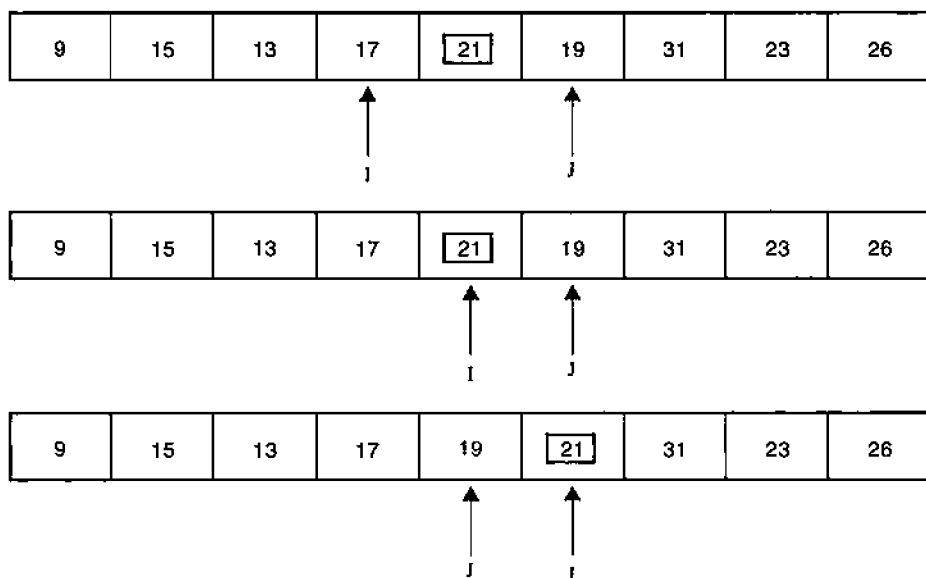
3. Mientras I apunte a un elemento que sea *menor* que 20, se incrementa el valor de I en 1, hasta que se encuentre un elemento mayor que el pivote. A continuación se realiza la misma tarea con el puntero J, buscando un elemento menor que 21, y mientras no lo encuentra se decrementa J en 1.

9	23	31	17	21	19	13	15	26
---	----	----	----	----	----	----	----	----

4. Se intercambian los elementos apuntados por I y J y a continuación se incrementan en uno los contadores I, J.



5. El proceso se repite



6. En el momento en que $J > I$, se ha terminado la partición. Se han generado dos sublistas, que tienen las propiedades citadas: la primera sublista, todos los elementos menores o igual a 20, y en la segunda, todos los elementos mayores que 20.

Sublista izquierda

9 15 13 17 19

Sublista derecha

21 31 23 26

Sintácticamente hablando, si el array original es a

$a[k] \leq 20$

for $k = 1..I - 1$
 $(k = 1..5)$

$a[k] > 20$

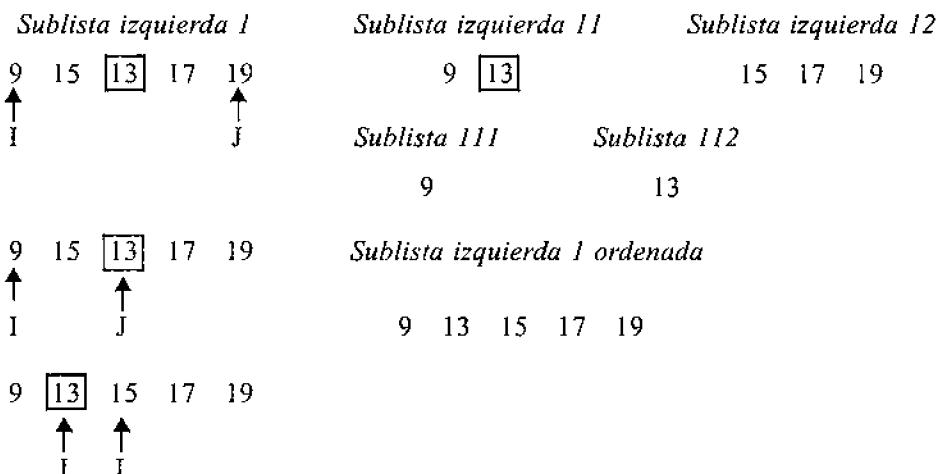
for $k = j + 1..N$
 $(k = 6..9)$

Si se utiliza J como índice final de la primera sublistas, se tiene:

Sublista izquierda 1..J (9 15 13 17 19)

Sublista derecha I..N (21 31 23 26)

7. La ordenación de las sublistas implica el mismo proceso que antes, excepto que los índices en el caso de la sublista son (1..5) y (6..9). Los pasos en el caso de la sublista izquierda son:



Algoritmo de ordenación rápida

1. Inicializar I a Primero (primer índice del array)
2. Inicializar J a Ultimo (último índice del array)
3. Seleccionar el elemento pivote (término Central)
Central $\leftarrow A[(\text{Primero} + \text{Ultimo}) \text{ div } 2]$
4. repetir
 - 4.1. mientras $A[I] < \text{Central}$ hacer


```
I  $\leftarrow I + 1$ 
fin-mientras
```
 - 4.2. mientras $A[J] > \text{Central}$ hacer


```
J  $\leftarrow J - 1$ 
fin-mientras
```
 - 4.3. si $I \leq J$ entonces Intercambiar ($A[I], A[J]$)


```
hasta que I > J
```
5. si $J > \text{Primero}$, llamar al procedimiento Partir, para dividir la sublista izquierda [$\text{Primero}..J$]
6. si $I < \text{Ultimo}$, llamar al procedimiento Partir, para dividir la sublista derecha [$I..Ultimo$]

Programa ordenación rápida

```

program TestRapido;
type
  Enteros = array [1..100] of integer;
var
  Lista : Enteros;
  K     : integer;

procedure Rapido (var A: Enteros; N : integer);

procedure Partir (Primero, Ultimo : integer);
var
  I, J      : integer;
  Central  : integer;

procedure Intercambiar (var M,N : integer);
var
  Aux : integer;
begin
  Aux := M;
  M   := N;
  N   := Aux
end; {Intercambiar}

begin {Partir}
  I := Primero;
  J := Ultimo;
  {encontrar Elemento pivote (central) }
  Central := A [(Primero + Ultimo) div 2];
repeat
  while A[I] < Central do
    I := I + 1;
  while A[J] > Central do
    J := J - 1;
  if I <= J then
    begin
      Intercambiar (A[I], A[J]);
      I := I + 1;
      J := J - 1
    end {if}
  until I > J;
  if Primero < J
    then Partir (Primero, J);
  if I < Ultimo
    then Partir (I, Ultimo)
end; {Partir}

begin {Rápido}
  Partir (1, N)
end; {Rápido}
{programa principal}
begin
  {lectura de 100 elementos aleatorios}
  for K := 1 to 100 do

```

```

begin
  Lista [K] := Random (1000);
  Write (Lista [K] : 8)
end;
WriteLn;
{llamada al procedimiento Rapido}
Rapido (Lista, 100);
{escritura de la lista ordenada}
for K := 1 to 100 do
  Write (Lista [K] : 8);
WriteLn
end.

```

15.7.1. Análisis de la ordenación rápida

El método de ordenación rápida es el más veloz de los conocidos. El único inconveniente de este método es la cantidad de memoria que se requiere en la pila. Caso de tener problemas de memoria, deberá realizar pruebas para evitar errores en ejecución. En este caso le recomendamos utilizar el método de Shell.

Si se supone que la lista se divide siempre en dos partes iguales, entonces, después de la d -ésima división de la lista, se tendrán 2^d partes. El número de iteraciones del procedimiento *partición* (partir) es $\Theta(n)$ para todas las partes. Como había $\log_2 n$ divisiones, el algoritmo requerirá $\Theta(n * \log_2 n)$.

15.8. ORDENACIÓN POR MEZCLA (MERGESORT)

Como su nombre sugiere, la idea básica de la ordenación es la *mezcla* de listas ya ordenadas. La filosofía de la mezcla ya la conoce el lector, la diferencia reside en que en este caso las listas estarán ordenados por un campo clave determinado.

Algoritmo

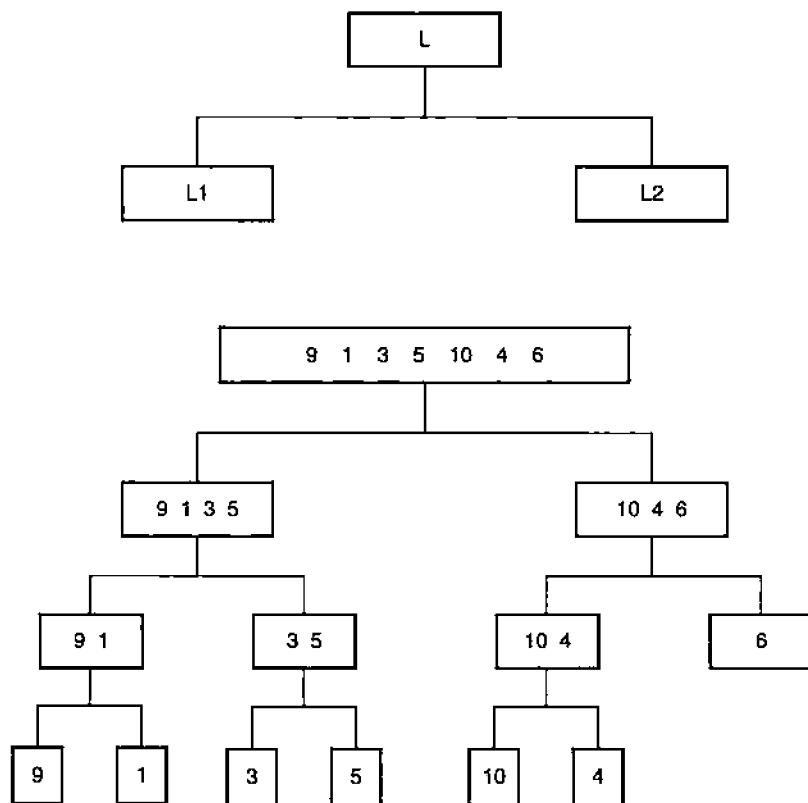
1. Dividir la lista en dos mitades.
2. Ordenar la sublista izquierda.
3. Ordenar la sublista derecha.
4. Mezclar las dos sublistas juntas.

EJEMPLO 15.1

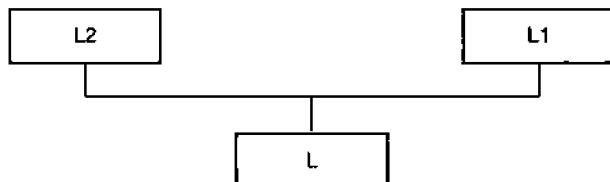
Ordenar por mezcla la lista.

9 1 3 5 10 4 6

El proceso consiste en dividir la lista en dos mitades y cada una de las mitades en otras mitades. Este proceso se repite hasta que cada sublista contiene, cada una, una entrada, según se aprecia en el gráfico.

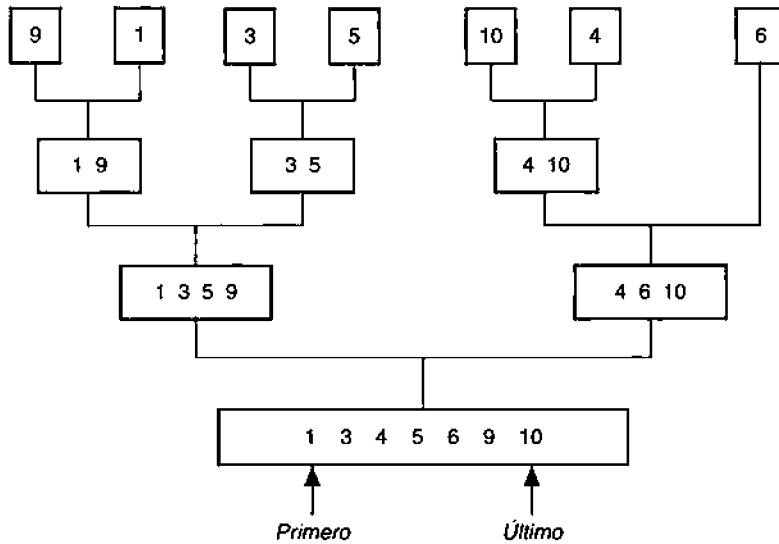


La mezcla comienza con las sublistas de un solo elemento, que se mezclan en sublistas más grandes cuyos elementos están ya ordenados, y el proceso continúa hasta que se construye una única lista ordenada.



El procedimiento de **ordenar por mezcla** se diseña con ayuda de la recursividad para dividir las listas y ordenar las sublistas; posteriormente se llama a un procedimiento Mezcla similar al estudiado ya en otro capítulo.

El diagrama de mezcla se ilustra así:

*Algoritmo OrdMezcla*

1. si Primero < Ultimo, entonces (índices de la lista original)
 - 1.1. Central \leftarrow (Primero+Ultimo) div 2 (punto de división para la partición)
 - 1.2. Llamar a OrdMezcla a [Primero..Central]
 - 1.3. Llamar a OrdMezcla a [Central+1..Ultimo]
 - 1.4. Mezclar a [Primero..Central] con a [Central+1..Ultimo]

El procedimiento OrdMezcla (*mergesort*) es un proceso recursivo que se llama a sí mismo para ordenar la sublista izquierda y a continuación la sublista derecha, y una vez ordenadas las dos sublistas se llama al procedimiento *Mezcla*.

```

procedure OrdMezcla (var A:ListaEnteros; Primero, Ultimo:integer);
{procedimiento recursivo ordena la sublista A[Primero..Ultimo]}
var
  Central : integer; {índice del último elemento de la sublista derecha}
procedure Mezcla (var Lista:ListaEnteros; Ida, Dcha,PuntoCen: integer);
{mezcla la sublista ordenada Lista [Ida] a Lista [PuntoCen] con la
 sublista ordenada Lista [PuntoCen + 1] a Lista [Dcha]}
var
  Aux : ListaEnteros;
  X, Y, Z : integer;
begin {Mezcla}
  X := Ida
  Y := PuntoCen + 1;
  Z := X;
  {bucle para mezclar las sublistas}
  while ( X <= PuntoCen) and ( Y<= Dcha) do
    
```

```

begin
  if Lista [X].Clave <= Lista[Y].Clave
    then
      begin
        Aux [Z] := Lista [X];
        X       := X + 1
      end
    else
      begin
        Aux [Z] := Lista [Y];
        Y       := Y + 1
      end;
    Z := Z + 1
  end;

{bucle para copiar elementos restantes, si existen}
while X <= PuntoCen do
  begin
    Aux [Z] := Lista [X];
    X       := X + 1;
    Z       := Z + 1
  end;
while I <= Dcha do
  begin
    Aux [Z] := Lista [Y];
    Y       := Y + 1;
    Z       := Z + 1
  end;
{copiar Aux en Lista}
for X := Ida to Dcha do
  Lista [X] := Aux [X]
end; {Mezcla}

begin {OrdMezcla}
  if Primero < Ultimo
    then Central := (Primero + Ultimo) div 2;
  OrdMezcla (Lista, Primero, Central);
  OrdMezcla (Lista, Central + 1, Ultimo);
  Mezcla (Lista, Primero, Ultimo, Central)
end; {OrdMezcla}

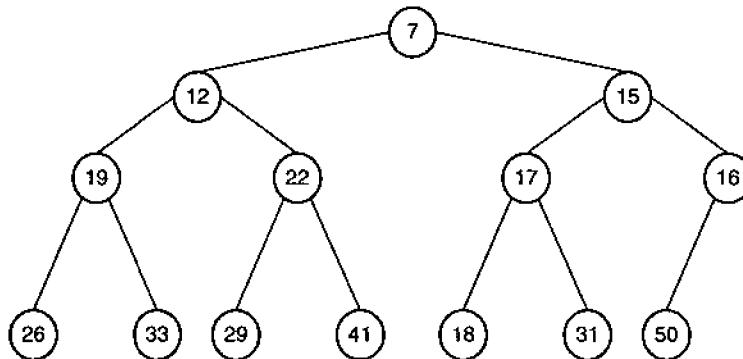
```

15.9. MÉTODO DE ORDENACIÓN POR MONTÍCULOS (HEAPSORT)

Este algoritmo de ordenación está basado en la estructura de montículo. Por lo que vamos a estudiar en primer lugar el concepto de montículo y después describimos el algoritmo.

15.9.1. Montículo

Se define un montículo de tamaño n como un árbol binario completo de n nodos, tal que el contenido de cada nodo es mayor o igual al contenido de su padre.



Utilizamos un array para representar el árbol binario, de tal forma que si el índice i hace referencia a un nodo, entonces el nodo hijo izquierdo está referenciado por el índice $2*i$, el nodo hijo derecho por el índice $2*i + 1$, y el nodo padre por $i \text{ div } 2$.

Así el nodo raíz ocupará la posición 1 en el array, y en la posición 2 y 3 estarán sus nodos hijo izquierdo y derecho, respectivamente.

La representación del árbol del ejemplo en un array

7	12	15	19	22	17	16	26	33	29	41	18	31	50
---	----	----	----	----	----	----	----	----	----	----	----	----	----

Utilizando esta representación tendremos que para que se cumpla que todo nodo del árbol ha de ser mayor o igual que el nodo padre, ha de cumplirse:

$$\begin{aligned} V[i] &\leq V[2*i] \\ &\forall i = 1 .. n/2 \\ V[i] &\leq V[2*i+1] \end{aligned}$$

Es claro que a partir de esta definición de montículo la raíz del árbol (o primer elemento del array) es el elemento más pequeño; en definitiva, $V[1]$ siempre tendrá el elemento más pequeño.

Según esto podemos descomponer el método de ordenación heapsort en los siguientes pasos:

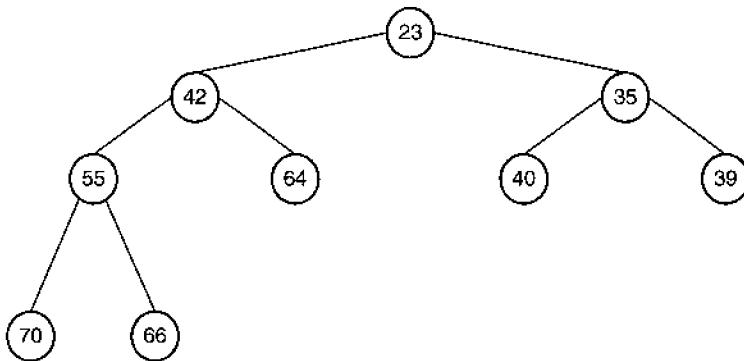
1. Construir un montículo inicial con todos los elementos del vector: $V[1], V[2], \dots, V[n]$
2. Intercambiar los valores de $V[1]$ y $V[n]$ (siempre se queda el máximo en el extremo).
3. Reconstruir el montículo con los elementos $V[1], V[2], \dots, V[n - 1]$.
4. Intercambiar los valores de $V[1]$ y $V[n - 1]$.
5. Reconstruir el montículo con los elementos $V[1], V[2], \dots, V[n - 2]$.

Está claro que estamos en 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. Lo expresamos en forma algorítmica:

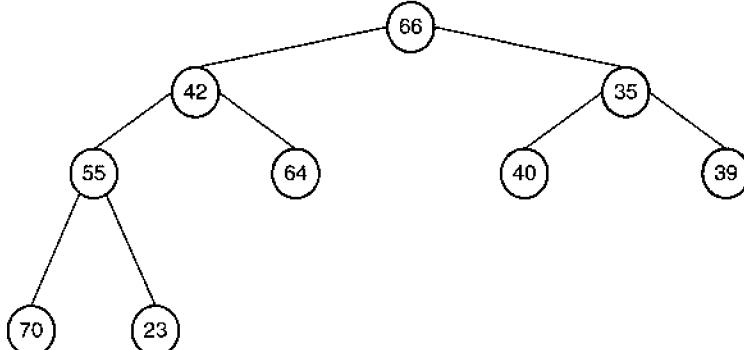
```
procedimiento Ordenacion_Heapsort (Vector, N)
  inicio
    <Construir montículo inicial (Vector, 1, N)>
    desde K ← N hasta 2 hacer
      intercambiar (Vector[1], Vector[K])
      construir montículo (Vector, 1, K-1)
    fin_desde
  fin Ordenacion_Heapsort
```

Según el algoritmo debemos considerar dos problemas: construir el montículo inicial y cómo restablecer los montículos intermedios. Como ahora veremos, la solución a ambos problemas va a ser mediante una misma rutina. Consideraremos que ya está construido el montículo inicial y se ha realizado el intercambio. La figura nos muestra esta situación.

Montículo construido



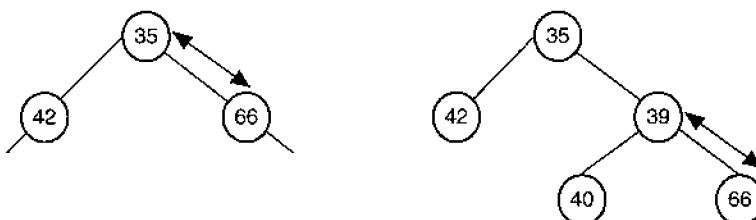
Debido al intercambio queda



Para restablecer el montículo hay dos posibilidades:

- $V[1]$ es menor o igual que los valores de sus hijos, entonces la propiedad del montículo no se ha roto.
- En otro caso, el elemento mínimo que necesitamos poner en $V[1]$ es o su hijo izquierdo o su hijo derecho ($V[2]$, $V[3]$, respectivamente). Por lo que se determina el menor de sus hijos y éste se intercambia con $V[1]$. El proceso continúa repitiendo las mismas comparaciones entre el nodo intercambiado y sus nodos hijos; así hasta llegar a un nodo en el que no se viole la propiedad de montículo, o estemos en un nodo hoja.

La figura nos muestra el proceso de restablecer la condición de montículo en el árbol resultante del intercambio; este proceso se expresa diciendo que el nodo situado en la cúspide del árbol se deja «hundir» por el camino de claves mínimas.



15.9.2. Procedimiento empuja

En el procedimiento Criba escribimos la codificación del algoritmo, restablece el montículo dejando hundir la clave por el camino de claves mínimas.

```

procedure Criba (var V:Vector; Primero, Ultimo:integer);
  {Primero; representa el nodo raíz}
var
  EsMtclo: boolean;
  Hijo:integer;
begin
  EsMtclo := false;
  while (Primero <= Ultimo div 2) and not EsMtclo do
    {primera condición quiere expresar que no sea una hoja de árbol}
  begin
    if 2*Primero = Ultimo then
      Hijo := 2*Primero {tiene un único descendiente}
    else {selecciona el mayor de los dos hijos}
      if V[2*Primero] > V[2*Primero+1] then
        Hijo := 2*Primero
      else
        Hijo := 2*Primero+1;
      {compara nodo raíz con el mayor de sus hijos}
      if V[Primero] < V[Hijo] then

```

```

begin
  Intercambia (V[Primero], V[Hijo]);
  Primero := Hijo {para continuar por la rama de claves mínimas}
end
else EsMtclo := true
end;
end;

```

15.9.3. Montículo inicial

Para construir el montículo inicial se llama a Criba (V, j, n) para todo $j = n/2, n/2 - 1, \dots, 1$. En definitiva se construye el montículo de «abajo» a «arriba», desde el penúltimo nivel del árbol hasta la raíz.

```

for j := n div 2 downto 1 do
  Criba (V, j, n);

```

De esta forma el procedimiento Criba es el núcleo de la realización del método de ordenación Heapsort.

Codificación de Heapsort

Antes de escribir la codificación cabe hacer una observación. El método ordena descentemente ya que siempre intercambia el elemento menor, $V[1]$, con el último del montículo actual. Para que la ordenación sea en orden ascendente simplemente debemos de invertir el vector. Pero si se desea que directamente termine en orden ascendente, se cambia la condición de montículo de modo que un nodo tenga la clave mayor (en vez de menor) que las claves de sus hijos.

La codificación del procedimiento de ordenación Heapsort:

```

procedure Ordenacion_Heapsort (var V: Vector; N: integer);
var
  J: integer;
begin
  for J := N div 2 downto 1 do
    Criba (V, J, N);
  for J := N downto 2 do
    begin
      Intercambia (V[1], V[J]);
      Criba (V, 1, J-1)
    end
  end;

```

15.10. MÉTODO DE ORDENACIÓN BINSORT

Este método, también llamado clasificación por urnas, plantea conseguir tiempos de ejecución menores de $O(n \log n)$ para ordenar n elementos siempre que se conozca algo acerca del tipo de las claves por las que se están ordenando.

Supongamos que tenemos un array de registros V, que 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 i := 1 to n do
    T[V[i].Clave] := V[i];
```

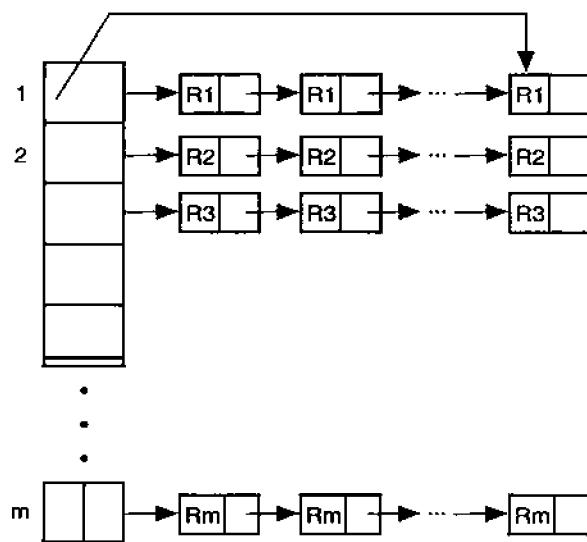
Sencillamente determina la posición que le corresponde según el valor del campo clave. El bucle lleva un tiempo de ejecución $O(n)$.

Esta ordenación tan sencilla que hemos 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 1, 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 se tiene un array de 1 a m urnas.

Urnas

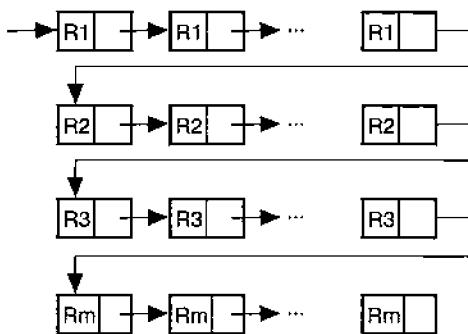


Realización del método de ordenación *binsort*

Para esta implementación consideramos que el campo clave de los registros está en el rango entero 1 .. m. Son necesarias m urnas por lo que vamos a definir un array de m urnas. Las urnas van a 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.

Una vez que hayamos distribuido los registros en las diversas urnas es necesario concatenar las listas. En la figura siguiente se muestra cómo realizar la concatenación.



Los tipos de datos para esta realización

```

const
  Limite = 1000;
  M = 100; {Máximo valor de la clave}
type
  TipoClave = 1..M;
  Registro = record
    Clave: TipoClave;
  end;
  Vector = array[1..Limite] of Registro;
  
```

La codificación del procedimiento de ordenación y las operaciones auxiliares que utiliza:

```

procedure Binsort(var V: Vector; N: integer);
{Tipos para manejo de las urnas. Éstas son representadas por listas}
type
  Puntero = ^Nodo;
  Nodo = record
    R: Registro;
    Sgte: Puntero
  end;
  Lista = record
    Frente, Final: puntero
  end;
  T_Urnas = array[1..M] of Lista;
var
  Urnas: T_Urnas;
  J, I: integer;
  L: Puntero;
  {procedimientos locales para el algoritmo}
  
```

```

procedure CrearUrnas(var U: T_Urnas);
var K: integer;
begin
  for K:= 1 to M do
  begin
    U[K].Frente:= nil;
    U[K].Final:= nil
  end
end;

function EstaVacia(Urna:Lista):boolean;
begin
  EstaVacia:= Urna.Frente=nil,
end;

procedure AñadirEnUrna(var UnaUrna: Lista; R: Registro);
  {Inserta el registro como último de la urna}
var
  T: Puntero;
begin
  new(T);
  T^.R:= R; T^.Sgte:= nil;
  with UnaUrna do
  begin
    if EstaVacia(UnaUrna) then
      Frente:= T
    else
      Final^.Sgte:= T;
    Final:= T
  end
end;

procedure EnlazarUrna(var Una: Lista; U: Lista);
begin
  if not EstaVacia(U) then
  begin
    Una.Final^.Sgte:= U.Frente;
    Una.Final:= U.Final
  end
end;{Sentencias de binsort}

begin
  CrearUrnas(Urnas);
  {Distribución de los registros en sus correspondientes urnas}
  for J:= 1 to N do
    AnadirEnUrna(Urnas[V[J].Clave], V[J]);
    {Concatena las listas que representan a las urnas desde Urnai hasta
     Urnan}
  I:= 1; {búsqueda de primera urna no vacía}
  while EstaVacia(Urnas[I]) do {la lógica del problema nos dice}
    I := I+1; {ha de haber alguna urna vacía}
  for J:= I+1 to M do
    EnlazarUrna(Urnas[I], Urnas[J]);
    {Se recorre la lista-urna resultante de la concatenación}
  J:= 1;
  L:= Urnas[I].Frente;
  while L <> nil do

```

```

begin
  V[J] := L^.R;
  J := J+1;
  L := L^.Sgte
end
end;

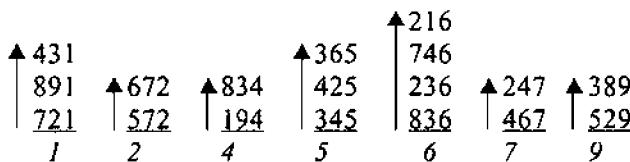
```

15.11. MÉTODO DE ORDENACIÓN RADIX-SORT

Este método se puede considerar como una generalización de la clasificación por urnas¹. 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. Para centrarnos en lo que estamos diciendo, supóngase que tenemos que 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

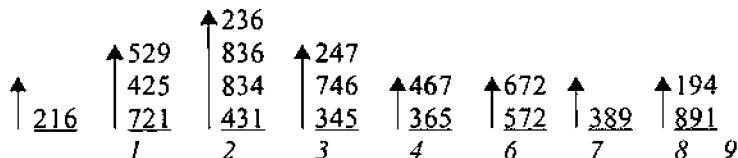
Atendiendo al dígito de menor peso (unidades) se reparten las fichas en montones del 0 al 9 (por ejemplo, el elemento 345 se coloca en el montón 5, el elemento 721 en el montón 1, etc.; las urnas de números cuyo primer dígito es 0, 3 y 8 no existen).



Tomando 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. Pues bien, ahora de nuevo distribuimos la secuencia de fichas en montones respecto al segundo dígito:

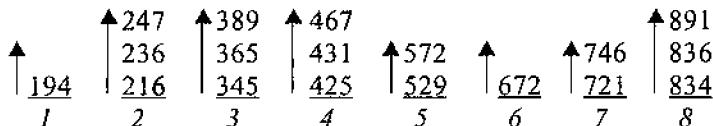


Tomando de nuevo los montones en orden, la secuencia de fichas queda así:

216 721 425 529 431 834 836 236 345 746 247 365 467 572 672 389 891 194

¹ Una urna se considera una lista enlazada en la que se almacenan los elementos leídos con un determinado criterio, el valor del dígito en la secuencia creciente de pesos (unidades, decenas, etc.).

Esta secuencia de fichas ya la tenemos ordenada respecto a los dos últimos dígitos, es decir, respecto a las decenas. Por último se distribuye de nuevo las fichas respecto al tercer dígito:



Tomando de nuevo los montones en orden, la secuencia de fichas queda ya ordenada:

194 216 236 247 345 365 389 425 431 467 529 572 672 721 746 834 836 891

La idea clave de la ordenación *radix-sort* (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_0 . En ese momento la secuencia estará ordenada.

Tipos de datos

Al igual que en el método de *binsort* las urnas estarán representadas por un vector de listas. En el caso de que la clave respecto a la que se ordena sea un entero, tendremos 10 urnas numeradas de 0 a 9. Las listas tienen una realización dinámica, cada lista se mantiene con dos punteros, uno al frente y otro al final de la lista, así el añadir un nuevo registro es inmediato ya que se enlaza por el final, de igual forma, concatenar las urnas consistirá en enlazar al final de una con el frente de la siguiente. Estas acciones ya están hechas en el método *binsort*.

A continuación se presentan los tipos de datos:

```

const
  M = 9;      {numeración de las urnas 0,1,2,...,9}
  Limite = 1000;
type
  TipoClave = 0.. maxint;
  Registro = record
    Clave: TipoClave;
  end;
  Vector = array[1..Limite] of Registro;
  {tipos para definir las urnas}
  Puntero = ^Nodo;
  Nodo = record
    R: Registro;
    Sgte: Puntero
  end;
  Lista = record
    Frente, Final: puntero
  end;
  T_Urnas = array[0..M] of Lista;

```

CODIFICACIÓN

La codificación supone que se está ordenando respecto a una clave entera y positiva. En primer lugar se determina el número de dígitos que tiene el campo clave, mediante divisiones sucesivas por 10; para ello, se toma el entero máximo que admite la computadora (en nuestro caso, 32.767). A continuación, se introducen de modo sucesivo cada elemento en su correspondiente urna mediante la unidad ? y se realizan iteraciones del bucle **for I**, hasta que I es el número máximo de dígitos del entero más grande de la lista a ordenar.

A continuación se presenta la codificación del procedimiento de ordenación Radix Sort.

```

procedure RadixSort(var V:vector; N: integer);
var
  Urnas: T_Urnas;
  I, J, R: integer;
  Aux, D, Peso, Ndig: integer;
  L, A: Puntero;
begin
  {Se calcula el número de dígitos}
  Aux := MaxInt; {MaxInt=32.767, en computadoras de 16 bit}
  Ndig := 0;
  while Aux >= 1 do
  begin
    Aux := Aux div 10;
    Ndig := Ndig + 1 {número de dígitos de la clave}
  end;

  Peso := 1; {Nos permite obtener los dígitos de menor a mayor peso}
  for I := 1 to Ndig do
  begin
    CrearUrnas(Urnas);{Crea las urnas con un bucle de 0 a M}
    for J := 1 to N do
    begin
      D :=(V[J].clave div Peso) mod 10;
      AnadirEnUrna(Urnas[D], V[J]);
    end;
    J := 0; {búsqueda de primera urna no vacía}
    while EstaVacia(Urnas[J]) do
      J := J+1;
    for R := J+1 to M do
      EnlazarUrna(Urnas[J], Urnas[R]);
      {Se recorre la lista-urna resultante de la concatenación}
    R := 1;
    L := Urnas[J].Frente;
    while L <> nil do
    begin
      V[R] := L^.r;
      R := R + 1;
      A := L;
      L := L^.Sgte;
      dispose(A)
    end;
    Peso := Peso * 10;
  end;
end;

```

15.12. BÚSQUEDA LINEAL

Otro problema importante en proceso de datos, como ya se ha comentado, es la *búsqueda* en un conjunto de datos de un elemento específico y la recuperación de alguna información asociada al mismo.

Existen diferentes métodos de búsqueda:

búsqueda lineal o secuencial

búsqueda binaria o dicotómica

búsqueda hash o por conversión de claves

Las dos primeras se pueden aplicar a listas implementadas con arrays, y la tercera es más propia de estructuras tipo registros o archivos. En esta sección y en la siguiente trataremos de la búsqueda lineal y de la búsqueda binaria dejando para el capítulo de archivos el tercer método.

15.12.1. Análisis

La búsqueda lineal o secuencial es la técnica más simple para buscar un elemento en un array (vector). Consiste el método en el recorrido de todo el vector, desde el primer elemento hasta el último, y de uno en uno. Si el vector contiene el elemento, el proceso devolverá la posición del elemento buscado dentro del vector y, en caso contrario, un mensaje que indique la falta de éxito en la búsqueda.

Mediante un bucle **desde** se compara el elemento *t* buscado con *a[i]*. En caso de encontrarlo, se almacena la posición (el índice del array) del mismo y finalmente se devolverá al programa principal. Dado que los algoritmos de búsqueda normalmente sólo devuelven la posición, es muy frecuente que la implementación del algoritmo se haga con una función.

EJEMPLO

Supongamos una lista de números de la Seguridad Social incluidos en un array *a* y se desea buscar a ver si existe el número 453714.

Pseudocódigo 1

```
Posicion ← 0
{lista = vector a[i] de n elementos}
desde i ← 1 hasta n hacer
    si a[i] = t
        entonces Posicion ← i
    fin_si
fin_desde
```

Este algoritmo tiene un grave inconveniente: sea cual sea el resultado (existe/no existe el elemento) se recorre el vector completo. El algoritmo tiene una mejora: detectar el momento de localizar el elemento y terminar el bucle. Así, el algoritmo mejorado se

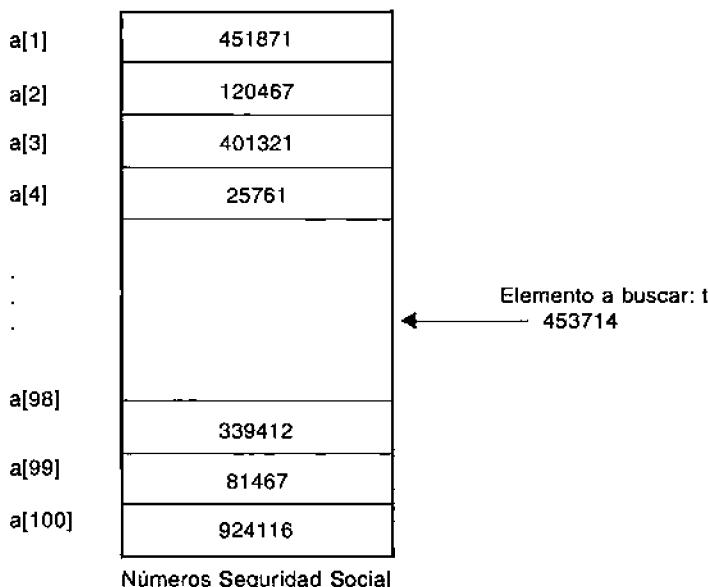


Figura 15.1. Búsqueda lineal de un elemento.

puede realizar con un bucle **while** o **repeat**, y utilizando unas banderas (interruptor) que detecten cuándo se encuentra el elemento. El bucle se terminará por dos causas:

- La bandera o indicador (**Encontrado**, por ejemplo) toma el valor esperado (por ejemplo, *true* —verdadero—) si la búsqueda ha tenido éxito.
- El valor del índice *i* es mayor que el número de términos de la lista, lo que significa que se ha terminado de recorrer la misma y el elemento buscado no ha aparecido.

Pseudocódigo 2

```

Encontrado ← falso
Posicion ← 0
    i ← 1
mientras (i <= n) y (No Encontrado) hacer
    si a[i] = t
        entonces Posicion ← i
                Encontrado ← verdadero
    fin_si
    i ← i+1
fin_mientras

```

o bien con la estructura **repetir**

```

repetir
    .
    .
hasta_que (Encontrado) o (i > n)

```

Función búsqueda lineal

```

function BusquedaLineal (A:Lista; {entrada, vector búsqueda}
N : integer; {entrada, número de elementos}
T : integer; {elemento a buscar}) : integer;
var
  Encontrado : boolean;
  I : integer;
begin
  BusquedaLineal := 0; {posición del elemento caso de no existir}
  Encontrado := false;
  I := 1;
  while (I <= N) and not Encontrado do
    if A[I] = T then
      begin
        BusquedaLineal := I;
        Encontrado := true;
      end {fin del if }
    else I := I + 1
  end; {posición del elemento en la lista}

```

Si se desea saber en el programa principal si existe el elemento, bastará preguntar con una sentencia **if** cuál es el valor de la función **BusquedaLineal**; si es cero no existe el elemento, y en caso contrario existe y su valor es la posición en la lista o vector.

Nota

Como Turbo Pascal tiene la sentencia **exit** para terminación anormal de un bucle, la búsqueda lineal se podría terminar nada más encontrarse el elemento ejecutando a continuación de la sentencia **exit** (en general, el uso de **exit** está orientado a la programación en tiempo real).

```

function BusquedaLinealDos (A: Lista; N : integer;
                           T: integer) : integer;
var
  I : integer;
begin
  for I := 1 to N do
    if T = A[I] then
      begin
        BusquedaLineal := I;
        Exit
      end;
  BusquedaLineal := 0
end;

```

Programa (se muestra una variante del algoritmo de la función)

```

program LinealBusqueda;
const
  Total = 100;

```

```

type
  Lista = array [1..Total] of integer;
var
  L : Lista;
  P, J, Num : integer;

function BusquedaLineal (t: integer; A: Lista; Maximo: integer): integer;
var
  I : integer;
  Encontrado : boolean;
begin
  Encontrado := false;
  I := 0;
  while (I < Maximo) and not Encontrado do
    begin
      I := I+1;
      Encontrado := A [I] = t
    end;
  if Encontrado
    then BusquedaLineal := I
    else BusquedaLineal := 0
end;

begin {programa principal}
  for J := 1 to Total do {lectura de 100 enteros aleatorios}
    L [J] := Random (100);
  repeat
    Write ('introduzca número a buscar');
    ReadLn (Num);
    P := BusquedaLineal (Num, L, Total);
    if P = 0
      then
        WriteLn ('no existe el número en la lista')
      else
        Writeln ('encontrado en la posición ',P:1)
    until Num = 0 {marca fin de datos de entrada}
end.

```

15.12.2. Eficiencia de la búsqueda lineal

El método de búsqueda secuencial, en el peor de los casos (el elemento buscado está al final de la lista o no existe), requiere consultar los n elementos de la lista para encontrar el elemento deseado o determinar que el elemento no existe en la lista. Entonces el tiempo de búsqueda es directamente proporcional al número de elementos de la lista, por lo que utilizando la notación O se tiene para el tiempo t la fórmula:

$$t = O[f(n)]$$

o simplificando

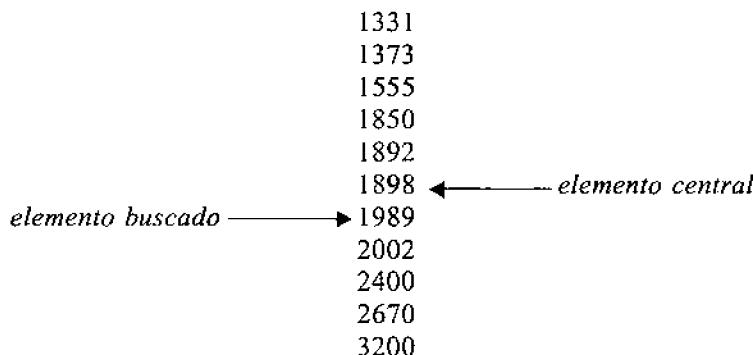
$$t = O(n)$$

15.13. BÚSQUEDA BINARIA

La búsqueda lineal, por su simplicidad, es buena para listas de datos pequeñas, para listas grandes es ineficiente; la búsqueda binaria es el método idóneo. Se basa en el conocido método de *divide y vencerás*.

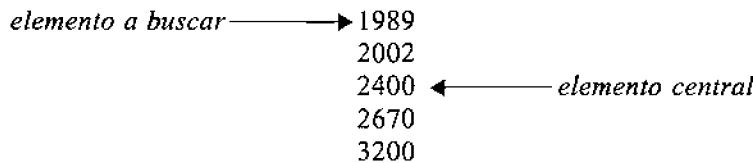
Este método tiene una clara expresión en la búsqueda de una palabra en un diccionario. Cuando se busca una palabra no se comienza la búsqueda por la página 1 y se sigue secuencialmente, sino que se abre el diccionario por una página donde aproximadamente se piensa puede estar la palabra, es decir, se divide el diccionario en dos partes; al abrir la página se ve si se ha acertado o en qué parte (la primera o la segunda) se encuentra la palabra buscada. Se repite este proceso hasta que por divisiones o aproximaciones sucesivas se encuentra la palabra.

Supongamos que la lista donde se busca es

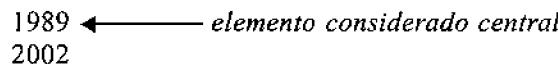


y que se busca el número 1989.

Se examina en primer lugar el elemento central de la lista (las divisiones se toman iguales), 1898. Dado que 1989 es mayor que 1898, el elemento a buscar estará en la segunda mitad:



El elemento central en esta sublistas es 2400, y como 1989 es menor, la nueva sublistas donde buscar es



Como ya no hay elemento central, se toma el número inmediatamente anterior a la posición central, que en este caso es 1989. En este caso se ha encontrado el elemento

deseado en tres comparaciones, mientras que en la búsqueda lineal hubiese necesitado al menos seis comparaciones (la mitad de los elementos, redondeada a un entero).

Este método es muy eficiente, con el único inconveniente, como habrá deducido, de requerir la lista ordenada.

Algoritmo

1. Establecer *Primero* $\leftarrow 1$ y *Ultimo* $\leftarrow n$ (*n*, número de elementos). Estas variables representan la primera y última posición de la lista o sublista donde se está buscando y permite el cálculo de la posición del elemento central.
2. *Encontrado* \leftarrow falso (variable lógica).
3. **mientras** *Primero* \leq *Ultimo* y no *Encontrado* **hacer**
 - {Encontrar posición central}
 - Central* \leftarrow (*Primero*+*Ultimo*) div 2
 - {Comparar elemento buscado *t* con a [*Central*]}
 - si *t* = *a*[*Central*]
 - entonces *Encontrado*=verdadero
 - sino si *t* > *a*[*Central*]
 - entonces *Primero* \leftarrow *Central* + 1
 - sino *Ultimo* \leftarrow *Central* - 1
- fin_mientras**
4. **si** *Encontrado*
 - entonces *Posicion* \leftarrow *Central* {existe el elemento}
 - sino *Posicion* \leftarrow 0 {no se ha encontrado}
- fin_si**

Programa

La búsqueda binaria requiere una ordenación previa del vector o lista en el que se va a efectuar la búsqueda. Por consiguiente, las acciones típicas (módulos) en un algoritmo de búsqueda binaria son:

1. Lectura del vector.
2. Ordenación del vector.
3. Búsqueda binaria.
4. Visualizar resultados.

Aprovechando que en Turbo Pascal disponemos de la directiva de compilación \$I, compile el procedimiento de ordenación Shell, para archivos de inclusión y déle el nombre Shell, se grabará con Shell.inc.

```
Program BusquedaBinaria;
const
  Limite = 100;
type
  Lista = array [1..Limite] of integer;
var
  A : Lista;
  I,J,t : integer;
```

```

{directiva de inclusión, aquí se inserta el procedimiento Shell}
{$I Shell.inc}
function Binaria (T: integer; var L: Lista; N: integer):integer;
var
  Primero, Ultimo, Central : integer;
  Encontrado              : boolean;
begin
  Primero   := 1;
  Ultimo    := N;
  Encontrado:= false;
  while (Primero <= Ultimo) and not Encontrado do
  begin
    Central := (Primero + Ultimo) div 2;
    if T = L[Central]
      then
        Encontrado := true
    else
      if T > L[Central]
        then Primero := Central + 1
        else Ultimo := Central - 1
  end; {while}
  if not Encontrado
    then Binaria := 0
    else Binaria := Central
end;

begin {programa principal}
  {lectura de 100 enteros aleatorios}
  for I := 1 to 100 do
    A[I] := Random (100);
  Shell (A,100);
  Write ('introduzca número a buscar');
  ReadLn (T);
  J := Binaria (T,A,100);
  if J = 0
    then WriteLn ('el número no figura en lista');
    else WriteLn ('el número ocupa la posición', J:1);
  WriteLn
  {si desea repetir la búsqueda, añada a este programa una estructura
  repetitiva}
end.

```

15.13.1. Eficiencia de la búsqueda binaria

En el algoritmo de búsqueda binaria, con cada comparación se divide en dos mitades el tamaño de la lista en estudio. Si n es el tamaño de la lista, los tamaños sucesivos de las sublistas serán

$$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \dots$$

$$\frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3}, \dots$$

El proceso terminará cuando el tamaño se hace igual o menor que 1. Por consiguiente, si k es el número mayor de comparaciones

$$\frac{n}{2^k} \leq 1$$

o bien

$$n \leq 2^k$$

Si se toman logaritmos en base 2, en ambos lados, se tiene

$$\begin{aligned}\log_2 n &\leq k, \quad \log_2 2 = k \\ \log_2 n &\leq k\end{aligned}$$

Por consiguiente, la eficiencia de la búsqueda binaria se puede escribir como

$\mathcal{O}(\log_2 n)$

que es bastante más rápido que la búsqueda lineal, como se puede ver en la Figura 15.3, que representa las dos funciones \mathcal{O} .

El tiempo que se ahorra utilizando el algoritmo de búsqueda binaria es muy considerable. Para una lista de 50.000 elementos, la búsqueda lineal en el peor de los casos requiere 50.000 comparaciones y 25.000 por término medio, mientras que la búsqueda binaria *nunca* requerirá más de $\log_2 50.000$. A fin de ser sinceros, al tiempo de la búsqueda binaria habría que sumarle el tiempo empleado en ordenar la lista.

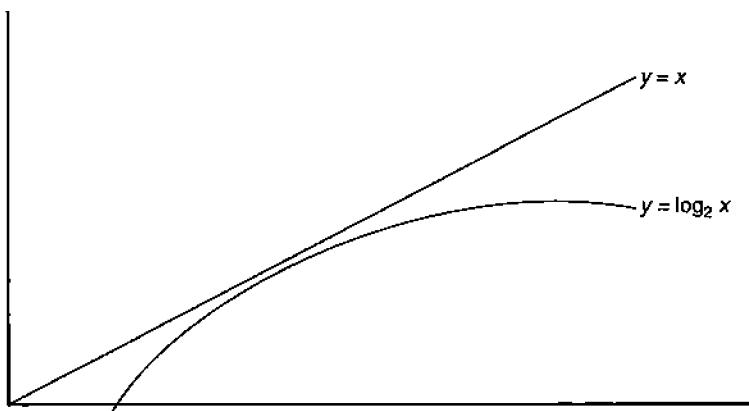


Figura 15.3. Funciones \mathcal{O} de búsqueda.

Recuerde

La búsqueda binaria sólo funcionará correctamente si la lista está ordenada. Sin embargo, la búsqueda lineal funciona tanto si la lista está ordenada como si no está ordenada.

15.14. BÚSQUEDA BINARIA RECURSIVA

Ya se examinó en otro capítulo el diseño de un subprograma para realizar la búsqueda en una lista (*array*) ordenado por el método de búsqueda binaria . Reconsideremos este ejemplo y reescribamos el subprograma de búsqueda utilizando recursividad, y recurriendo a dos estructuras muy frecuentes: a) *array de enteros*; b) *array de registros*, y como generalización del método en un caso utilizaremos una función y en otro un procedimiento.

15.14.1. Función búsqueda

Datos de entrada: Array ordenado A
 Primero, Ultimo (índices extremos de A)
 Clave (elemento buscado)

Datos de salida: verdadero,
 falso (existe/no existe la clave buscada)

Programa

```
function Busqueda (var A:Lista; Max, Min:integer; Clave:integer):boolean;
var
  Central : integer; {elemento central del array}
begin
  if Min > Max
  then Busqueda := false
  else
    begin
      Central := (Max + Min) div 2;
      if A[Central] = Clave
        then
          Busqueda := true
        else
          begin
            if Clave > A [Central]
            then
              Min := Central + 1
            else
              Max := Central - 1;
            Busqueda := Busqueda (A, Max, Min, Clave)
          end
    end
  end
{búsqueda}
end;
```

15.14.2. Procedimiento BusquedaBin

Considérese ahora un array de registros.

```

type
  InfoEmp = record
    Nombre : string [30];
    NumSS : integer; {número de la Seguridad Social}
    Salario: real;
    Edad   : integer;
  end
  Array_Emp = array [1..100] of InfoEmp;
var
  Lista : Array_Emp;

```

Parámetros de entrada

Lista	
Clave	<i>(número de la Seguridad Social buscado)</i>
Primero, Ultimo	<i>(límite del array —índices—)</i>

Parámetros de salida

Indice (posición que ocupa el número de la SS buscado, NumSS)

Programa

```

procedure BusquedaBin (var Lista:Array_Emp; var Indice:integer;
                      Primero,Ultimo:integer;clave:integer);
var
  Central : integer;
begin
  if Primero > Ultimo then
    Indice := 0
  else begin
    Central := (Primero + Ultimo) div 2;
    if Clave = Lista [Central].NumSS then
      Indice := Central
    else
      if Clave < Lista [Central].NumSS then
        begin {búsqueda en la primera mitad}
          Ultimo := Central-1;
          BusquedaBin (Lista, Indice, Primero, Ultimo, Clave)
        end
      else
        begin {búsqueda en la segunda mitad}
          Primero := Central + 1;
          BusquedaBin (Lista, Indice, Primero, Ultimo, Clave)
        end
    end {else}
  end; {BusquedaBin}

```

15.15. MEZCLA

El proceso de **mezcla**, **fusión** o **intercalación** (*merge* en inglés) consiste en tomar dos vectores ordenados (*a*, *b*) y obtener un nuevo vector (*c*) también ordenado.

El algoritmo más sencillo para resolver el problema es:

1. Situar todos los elementos del vector *a* en el nuevo vector *c*.
2. Situar todos los elementos del vector *b* en el nuevo vector *c*.
3. Ordenar todo el vector *c*.

Esta solución tiene un inconveniente: no se tiene en cuenta que los vectores *a* y *b* ya están ordenados; ello supone una ralentización del proceso. El algoritmo que tiene en cuenta la ordenación es el siguiente:

1. Seleccionar el elemento de valor o clave más pequeño con cualquiera de los dos vectores y situarlo en el nuevo vector *c*.
2. Comparar *a*(*i*) y *b*(*j*) y poner el elemento de vector más pequeño en *c*(*k*) (*i*, *j*, *k* son los índices de los elementos correspondientes en los vectores).
3. Seguir esta secuencia de comparaciones hasta que los elementos de un vector se hayan agotado, en cuyo momento se copia el resto del otro vector en *c*.

EJEMPLO 15.2

*Mezclar las dos listas de números *a* y *b*.*

2	4	78	97				<i>Lista A</i>
-15	0	13	15	78	90	96	<i>Lista B</i>

El proceso gráfico se muestra en la Figura 15.4.

Procedimiento mezcla de los vectores *A* y *B*

```

procedure Mezcla (var A,B,C : Lista; M,N : integer);
{A y B: entrada. Vectores ya ordenados}
{M y N: número de elementos de A y B respectivamente}
{C : salida. Vector mezcla ordenado}
{El tipo Lista tendrá una longitud mínima de M + N elementos}
var
  I, J, K : integer;
begin
  I := 1;
  J := 1;
  K := 1;
  while (I <= M) and (J <= N) do
    begin
      if A[I] <= B[J]
        then
          C[K] := A[I];
          I := I + 1;
        else
          C[K] := B[J];
          J := J + 1;
        end;
      K := K + 1;
    end;
  end;

```

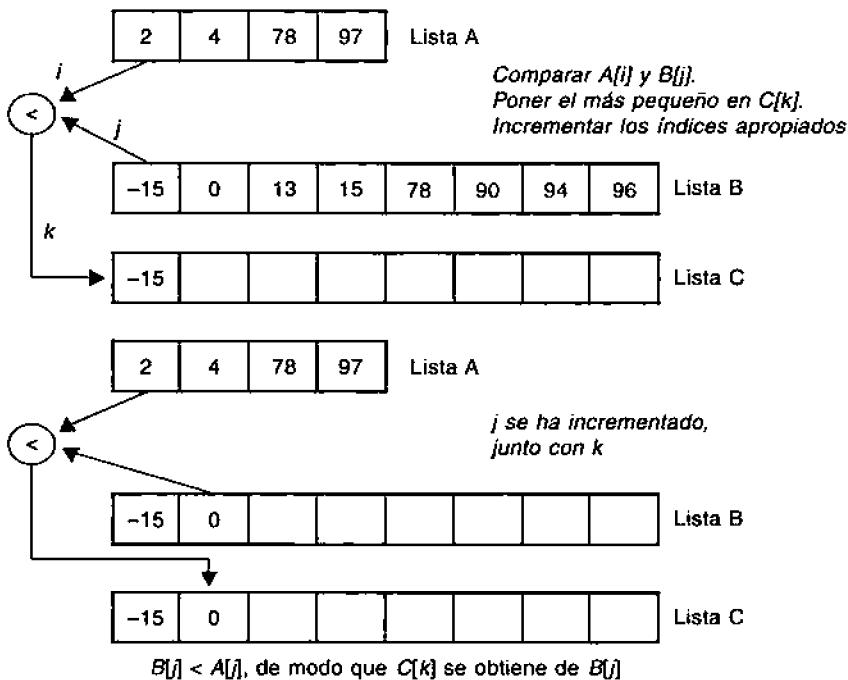


Figura 15.4. Mezcla de dos listas.

```

begin
  C [K] := A[I];
  I      := I + 1
end
else
begin
  C [K] := B[J];
  J      := J + 1
end;
K := K + 1
end;

{copiar el resto del vector no agotado}
if I > M
then
  for P := J to N do
    begin
      C [K] := B [P];
      K      := K + 1
    end
else (J > N)
  for P := I to M do
    begin
      C [K] := A [P];
      K      := K + 1
    end
end;

```

RESUMEN

Los procesos de programación más usuales son: búsqueda y ordenación. Son partes esenciales de un gran número de programas de procesamiento de datos. La búsqueda y la ordenación son también procesos que se encuentran normalmente en la vida diaria. Es necesario constatar el hecho de que en muchas ocasiones se consigue una eficiencia considerable cuando se trata de buscar palabras en un diccionario o un nombre en una dirección telefónica, que vienen dispuestas en orden alfabético.

Los procedimientos de búsqueda básicos son: lineal y binaria. Una *búsqueda lineal* es sólo adecuada para listas de datos pequeñas, mientras que para una lista de datos larga es muy ineficiente. Uno de los métodos de búsqueda más eficiente es la *búsqueda binaria*.

Los métodos de ordenación simple presentan muy poca diferencia en la eficiencia y así, para listas o vectores pequeños, se pueden considerar los algoritmos de ordenación, ordenación mixtos, etc., mientras que para listas o vectores más grandes, los algoritmos más eficientes son las *ordenaciones de selección* y *de Shell*. Según un estudio de Hale y Easton¹ de 1987, los tiempos de ordenación requeridos para ordenar arrays de 512 y 2.500 elementos son:

	512 elementos	2.500 elementos
Burbuja	5.654 segundos	131.618
Burbuja con indicador	5.713 segundos	135.469
Ordenación por selección	5.967 segundos	137.526
Ordenación por selección modificada	3.389 segundos	77.480
Ordenación por inserción	1.453 segundos	30.361

Para el caso de arrays grandes, se consideran los métodos de ordenación avanzados, que son mucho más rápidos y eficientes que los métodos de ordenación elementales ya citados. Sin embargo, excepto en el caso de ordenación rápida (*quicksort*), los métodos avanzados son mucho más grandes —en código— que las ordenaciones simples.

Los tiempos requeridos para los métodos de ordenación avanzados en el caso de arrays de 512 y 2.500 elementos se resumen en la siguiente tabla (Hale y Easton, 1987):

	512 elementos	2.500 elementos
Ordenación Shell	.370 segundos	2.540 segundos
Ordenación por mezcla	.487 segundos	3.171 segundos
Ordenación por montículo	.342 segundos	2.092 segundos
Quicksort recursiva	.195 segundos	1.160 segundos
Quicksort no recursiva	.308 segundos	1.674 segundos

EJERCICIOS

- 15.1. Escribir un programa que lea una serie de números enteros, los ordene en orden descendente y a continuación visualice la lista ordenada.

¹ Hale, Guy J., y Easton, Richard J.: *Applied Data Structures Using Pascal*, Lexington, Massachusetts, Heath and Company, 1987, págs. 158 y 419.

- 15.2. Un método de ordenación muy simple, pero no muy eficiente, de elementos $x_1, x_2, x_3, \dots, x_n$ 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_3 .
 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.
- 15.3. Escribir un programa que lea 42 números enteros en un array 7×6 y realizar las siguientes operaciones:
- Imprimir el array.
 - Encontrar el elemento mayor del array.
 - Indicar dónde se encuentra el elemento mayor del array.
 - Si el elemento mayor está repetido, indicar cuántas veces y la posición de cada elemento repetido.
- 15.4. Igual que el ejercicio 15.1, pero con el método Shell.
- 15.5. Se lee una lista de números desde teclado y se desea saber si entre dichos números se encuentra el 333. En caso afirmativo, visualizar su posición en la lista. Resolver el problema por:
- Búsqueda secuencial.
 - Búsqueda binaria.

PROBLEMAS

- 15.1. 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.
- 15.2. 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.
- 15.3. Se dispone de dos vectores, *Maestro* y *Esclavo*, del mismo tipo y número de elementos. Se deben imprimir en dos columnas adyacentes. Se ordena el vector *Maestro*, pero siempre que un elemento de *Maestro* se mueva, el elemento correspondiente de *Esclavo* debe moverse también; es decir, cualquier cosa que se haga a *Maestro*[*i*] debe hacerse a *Esclavo*[*i*]. Después de realizar la ordenación se imprimen de nuevo los vectores. Escribir un programa que realice esta tarea.
- 15.4. 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 la visualice. La información debe ser ordenada por ventas de mayor a menor y visualizada de nuevo.

- 15.5. Se desea realizar un programa principal que realice las siguientes tareas con procedimientos o funciones:

- a) Leer una lista de números desde el teclado.
- b) Visualizar dichos números.
- c) Ordenar en modo creciente.
- d) Visualizar lista ordenada.
- e) Buscar si existe el número 444 en la lista.

Ampliar el programa anterior de modo que pueda obtener y visualizar en el programa principal los siguientes tiempos:

- t1. Tiempo empleado en ordenar la lista de coordenadas.
- t2. Tiempo que se emplearía en *ordenar* la lista ya ordenada.
- t3. Tiempo empleado en ordenar la lista ordenada en orden inverso.

- 15.6. 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 cada una de las listas A y B.
- b) Crear una lista C por intercalación o mezcla de las listas A y B.
- c) Localizar si existe en la lista C el número 255.

Se desea visualizar también en el programa principal las siguientes tareas:

- a) Escribir un mensaje «*existe*»/«*no existe*» el número 255.
- b) Visualizar la lista C ordenada.

- 15.7. Escribir un programa que genere un vector de 10.000 números aleatorios de 1 a 500. Realice la ordenación del vector por dos métodos:

- *Binsort*
- *Radixsort*

Escriba el tiempo empleado en la ordenación de cada método.

- 15.8. Escribir un programa que lea una serie de números enteros, los ordene en orden descendente y a continuación visualice la lista ordenada.

- 15.9. Un método de inserción muy simple, pero no muy eficiente, de elementos X_1, X_2, \dots, X_n en orden ascendente es el siguiente:

Paso 1: Localizar el elemento más pequeño entre X_1 y X_n y cambiarlo con X_1 .

Paso 2: Localizar el elemento más pequeño entre X_2 y X_n y cambiarlo con X_2 .

...

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

- 15.10. La fecha de nacimiento de una persona está representada por el registro
Fecha = record dia: 1..31; mes: 1..12; anno: 1900..2010 end. Escribir un programa que tenga como entrada el nombre y la fecha de nacimiento de los alumnos de un colegio. La

salida ha de ser un listado en orden de nacimiento. Utilizar como método de ordenación RadixSort (sin convertir fecha).

- 15.11. En un archivo se ha guardado la lista de pasajeros de un vuelo con salida en Roma. Se sabe que el número de pasajeros no sobrepasa los 350 y que cada pasajero está identificado con un número de control en el rango de 1 a 999. Por un error informático hay números de control repetidos. Escribir un programa que realice las siguientes operaciones.
- Ordenar en memoria interna la lista de pasajeros por el número de control.
 - Los pasajeros que tienen un número de control repetido, asignarles el número de control no existente más bajo.
 - Escribir en el archivo la lista de pasajeros en orden creciente del número de control.
- 15.12. Dado un vector x de n elementos reales, donde n es impar, diseñar una función que calcule y devuelva la mediana de este 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.
- 15.13. Se implementan cadenas de caracteres mediante arrays de caracteres de una dimensión. Diseñar un programa que ordene cadenas. Aplicar el método de RadixSort para la ordenación.
- 15.14. Se quiere construir una agenda telefónica con la siguiente información: nombre, domicilio y número de teléfono. Diseñar un programa para mantener la agenda, que como mucho almacenará información sobre 100 personas, de tal forma que la búsqueda se realice por el nombre de la misma y que se mantenga durante su procesamiento ordenada alfabéticamente de forma ascendente.

Análisis de algoritmos

CONTENIDO

- 16.1. La medida de la eficiencia de algoritmos.
- 16.2. Notación *O-grande*.
- 16.3. La eficiencia de los algoritmos de búsqueda.
- 16.4. Análisis de algoritmos de ordenación.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

REFERENCIAS BIBLIOGRÁFICAS.

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

16.1. MEDIDA DE LA EFICIENCIA DE ALGORITMOS

El coste de un programa de computadora está constituido por diversos componentes. Algunos de estos componentes implican el coste del tiempo humano —tiempo de las personas que desarrollan, mantienen y usan el programa—. Los 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 tratará de 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 cuando se selecciona un algoritmo y su 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 dificultades 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 medir la eficiencia de los algoritmos se usará el principio de invarianza que establece que la eficiencia de dos implementaciones diferentes de un mismo algoritmo difieren tan solo en una constante multiplicativa.

16.1.1. Análisis del orden de magnitud

Las técnicas matemáticas analizan algoritmos con independencia de implementaciones específicas, de computadoras o de datos. El enfoque matemático considera un enfoque de tiempo del algoritmo como una función del tamaño. Se define el tamaño de un ejemplar de un determinado problema como el número de dígitos binarios para representarlo en la computadora, codificado de un modo razonablemente compacto. El medio para medir el tamaño de un ejemplar de un problema depende de la aplicación (ejemplos típicos son el tamaño de un array, el número de nodos de una lista enlazada, el número de registros de un archivo o el número de elementos de una lista ordenada). Es decir, normalmente identificamos el tamaño con el número de elementos lógicos contenidos en un ejemplar de entrada.

A medida que crece el tamaño de un ejemplar 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 tamaño del ejemplar de 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 de un determinado

problema. Para poder medir este tiempo supondremos una computadora hipotética pero semejante en resultados a la realidad.

¿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 ».

16.1.2. Consideraciones de eficiencia

Las consideraciones de eficiencia fundamentales son: *el tiempo y el espacio*. El programador normalmente debe optimizar ambos conceptos, aunque normalmente 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 la sentencia:

El algoritmo Alg 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 sentencia es válida también si Alg requiere n^2 segundos, $5*n^2$ segundos, o $n^2/10$ segundos para resolver un problema de tamaño n .

16.1.3. 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 (*terminación*).

16.1.4. 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 $T(n) = cn$, donde $c > 0$ es una constante positiva. 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 Pascal ejecutada por el programa o como la longitud de tiempo tomada para ejecutar el programa en alguna 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 $T_{media}(n)$, el tiempo medio de ejecución del programa sobre todas las entradas de tamaño n . El tiempo de ejecución medio es a veces una medida más realista de lo que el rendimiento verá 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.

16.2. NOTACIÓN O-GRANDE

Supongamos que se ha escrito y depurado un programa en Pascal 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. Al principio de la década de los noventa la relación de las velocidades de las supercomputadoras más rápidas y las computadoras personales más lentas era aproximadamente 1.000 a 1.
2. El compilador Pascal específico utilizado para generar instrucciones máquinas para esta computadora. El número de instrucciones máquinas utilizadas para ejecutar una sentencia dada variará de un compilador a otro compilador.

En consecuencia no se puede considerar a un programa en Pascal 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 «*O-grande*» que 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 $4n-1$ en un array de longitud n , se dirá que emplea un tiempo $O(n)$ que se lee «*O grande de n*» o bien «*O de n*» y que informalmente significa «*algunos tiempos constantes n*».

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

16.2.1. Descripción de tiempos de ejecución

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

Así, sea $T(n)$ 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 .

Así, sea $f(n)$ una función definida sobre enteros no negativos. Se dice « $T(n)$ es $O(f(n))$ », si $T(n)$ 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 $n \geq n_0$ se tendrá que $T(n) \leq cf(n)$.

Se puede aplicar la definición de «*O grande*» para probar que $T(n)$ es $O(f(n))$ para unas funciones específicas T y f . En consecuencia, se elige una, y a continuación se prueba que $T(n) \leq cf(n)$, suponiendo sólo que n es un entero no negativo y que n es al menos tan grande como nuestra opción n_0 .

16.2.2. Definición conceptual

Especificamente, la notación $f(x) = O(g(x))$ significa que existen constantes C y x_0 tales que $-Cg(x) \leq f(x) \leq Cg(x)$ para todo $x \geq x_0$. En otras palabras, $f(x) = O(g(x))$ significa que $|f(x)| \leq C|g(x)|$ para $x \geq x_0$.

Por consiguiente $|g(x)|$ es un límite superior¹ para $|f(x)|$, como se ilustra en la Figura 16.1.

Supóngase que $f(x) = 2x^2 + 3x - 1$. Entonces

$$\begin{aligned} f(x) &\leq 3.5x^2 && \text{para } x \geq 1.6 \\ f(x) &\leq 2x^3 && \text{para } x \geq 1.8 \\ f(x) &\leq x^4 && \text{para } x \geq 1.9 \end{aligned}$$

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 pueden actuar hasta límites superiores en $f(x)$ si x es suficientemente grande.

¹ Salmon, William I., *Structures and Abstractions*, Boston, MA, Irwin, 1991, págs. 733-737.

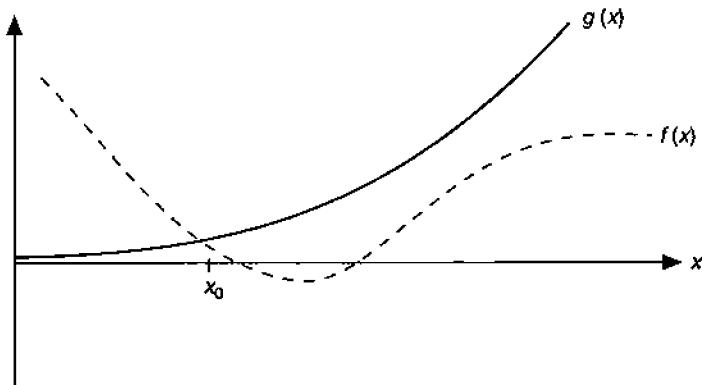


Figura 16.1. $f(x) = O(g(x))$ si existen números C y x_0 tal que: $|f(x)| \leq Cg(x)$ para todo $x \geq x_0$.

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

16.2.3. Funciones tipo monomio

Supóngase que $1 < x$. Multiplicando repetidamente en ambos lados de la desigualdad por el positivo x , quedará $x < x^2, x^2 < x^3, x^3 < x^4, \dots$

Por inducción, se deduce que $1 = x^0 < x^1 < x^2 < x^3 < \dots$

En otras palabras: $x^n < x^{n+1}$ para todo $x > 1, n \geq 0$

Por consiguiente: $x^n = O(x^n) = O(x^{n+1}) = O(x^{n+2}) = \dots$

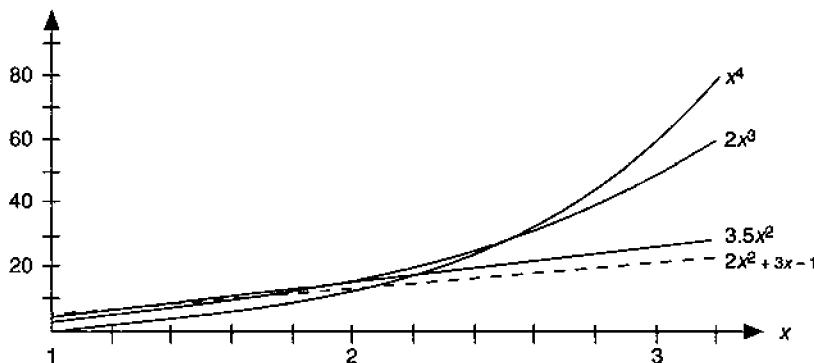


Figura 16.2. Las funciones individuales $3.5x^2, 2x^3$ y x^4 residen por encima de $f(x) = 2x^2 + 3x - 1$ para x bastante grande.

² Salmon, William I., *op. cit.*, pág. 734.

En general casi siempre se considera el límite superior más bajo para una función dada, para cualquier constante k , se tendrá:

$$\begin{aligned} k &= O(1) \\ kx &= O(x) \\ kx^2 &= O(x^2), \dots \end{aligned}$$

16.2.4. 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

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

de modo que

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

Por consiguiente, una función polinómica de grado n de x es $O(x^n)$ para $x > 1$, $n \geq 0$.

16.2.5. Órdenes exponenciales y logarítmicas

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

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

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 nominal positivo.

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

La Figura 16.3³ muestra el gráfico de e^x , $\ln x$ y x^n .

³ Salmon, William L., *op. cit.*, pág. 735.

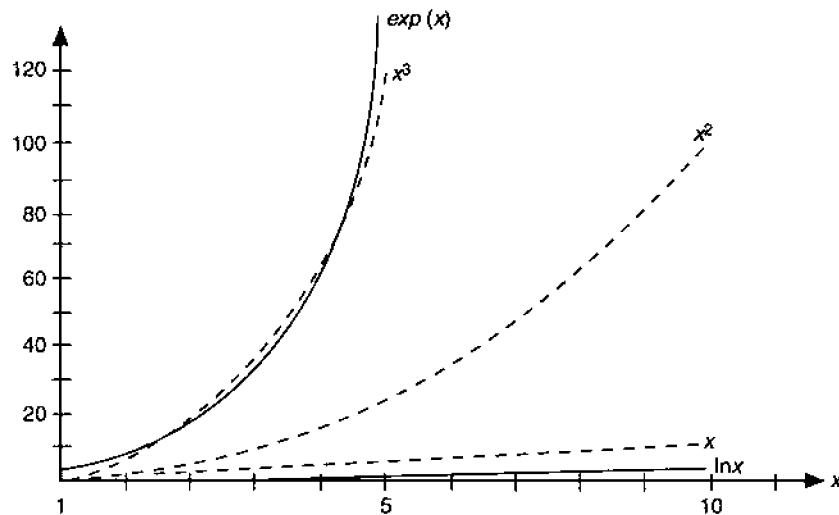


Figura 16.3. Para x grandes, el grafo de e^x se sitúa por encima del grafo de cualquier potencia de x y el grafo de $\ln x$ se sitúa por debajo del grafo de cualquier potencia positiva de x^3 .

16.2.6. Bases de logaritmos 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 se 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

$$\begin{aligned} \log_B x &= \log_B (b^{\log_b x}) = (\log_b x) (\log_B b) \\ \log_B x &= \log_B (B^{\log_B x}) = (\log_B x) (\log_B B) = \log_B x \end{aligned}$$

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:

$$\log_2 x = \frac{\log_{10} x}{\log_{10} 2} = 3.3219281 \log_{10} x$$

y

$$\log_2 x = \frac{\ln x}{\ln 2} = 1.4426950 \ln x$$

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

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

EJEMPLOS

La medida de *O-grande* de una función depende de su variable independiente y es el orden del término que crece más rápidamente.

$$2n^3 - 8n + 5 = O(n^3)$$

$$(r^2 - 3r + 5) / (r - 1) = O(r)$$

$$15 = O(1)$$

$$\log_{10} P + 5P = O(P)$$

$$\log_2 x + (5 / x) = O(\log x)$$

EJEMPLO 16.1

Sea $f(n) = n^*(n + 1) + 4$ para $n = 1, 2, \dots$

Demostrar que f es $O(n^2)$.

Demostración

Se necesita encontrar constantes c y N tales que

$$f(n) \leq c * n^2 \quad \text{para todo } n \geq N$$

dado que $n * (n + 1) + 4 = n^2 + n + 4$ para todos los enteros positivos n , conocemos que $f(n) \leq n^2 + n^2 + 4 + n^2$ siempre que $n \geq 1$. Y así sucesivamente para $c = 6$ y $N = 1$, $|f(n)| \leq c * n^2$ para todo $n \geq N$. Por consiguiente, f es $O(n^2)$.

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

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

EJEMPLO 16.2

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 tales que para todo $n \geq N$

$$|f(n)| \leq c * \log_a(n)$$

Recordemos una prioridad fundamental de los logaritmos

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

Sea

$$c_1 = c * \log_a(b)$$

Entonces para todo $n \geq N$ se tiene que

$$\begin{aligned} |f(n)| &\leq c * \log_a(n) \\ &= c * \log_a(b) * \log_b(n) \\ &= c_1 * \log_b(n) \end{aligned}$$

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

16.2.7. 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:

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

Ejemplo

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

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

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

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

De modo similar para el algoritmo *All2*, 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 , cosa que no es cierta para $n \leq 200$ 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.

16.3. LA EFICIENCIA DE LOS ALGORITMOS DE BÚSQUEDA

Como ejemplo de aplicación del análisis de algoritmos, considérese en primer lugar la eficiencia de los dos algoritmos de búsqueda: *búsqueda secuencial* y *búsqueda binaria* en una lista o vector (*array*).

16.3.1. Búsqueda secuencial

El algoritmo de búsqueda secuencial en una lista (*array*) de n elementos, busca cada elemento por turno, comenzando con el primero hasta que, o bien se encuentra el elemento deseado o se alcanza el final de la colección de datos. En el mejor caso, el elemento deseado es el primero que se examina, de modo que sólo se necesita una comparación. Por consiguiente, en el 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. Por consiguiente, en el 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; por consiguiente, el algoritmo es $O(n)$ en el caso medio.

Búsqueda secuencial

- Caso mejor $O(1)$
- Caso medio $O(n)$
- Caso peor $O(n)$

16.3.2. 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 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 la 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 el elemento deseado. Sin embargo, se puede calcular el número máximo de comparaciones que requiere una búsqueda binaria, esto es, el peor caso. El número de comparaciones es igual al número de veces que el algoritmo divide el array por la mitad. Supongamos que $n = 2^k$ para 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 $n/2$.
3. Inspeccionar el elemento central de un array de tamaño $n/2^2$.
4. Etcétera.

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 .

$$k = \log_2 n$$

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 $n = 2^k$.

¿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 $2^4 = 16 < 30$ y $2^5 = 32 \geq 32$.)

El algoritmo requerirá como máximo k divisiones para obtener un subarray con un elemento. Es decir,

$$\begin{aligned} k - 1 &< \log_2 n < k \\ k &< \lfloor \log_2 n \rfloor < k + 1 \\ k &= \lfloor \log_2 n \rfloor \quad \text{redondeado por defecto} \end{aligned}$$

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

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 $\log_2 1.000.000 + 1 \approx 20$. Para 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.

16.4. 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.

16.4.1. Análisis de la ordenación por selección

Un procedimiento Pascal que realiza una ordenación por selección de un array (lista) A de n elementos puede ser éste:

```
procedure OrdenarSeleccion (var A: TipoArray; n: integer);
var
  i, j, jmax : integer;
begin
  for i := n downto 2 do
    begin
      jmax := 1;
      for j := 2 to i do {encontrar clave mayor}
        if A[j].clave > A[jmax].clave then
          jmax := j;
      intercambio(A[i], A[jmax]);
    end;
end;
```

Como primera etapa en el análisis de algoritmos se debe contar el número de comparaciones e intercambios que requiere la ordenación de n elementos. El procedimiento

Ordenar Selección del Capítulo 15 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, el procedimiento siempre realiza $n - 1$ intercambios (n , número de elementos del array). Existen $n - 1$ llamadas a cada uno de los restantes procedimientos. Cada llamada hace que su bucle se ejecute 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:

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

La suma es cuadrática por lo que el número de comparaciones claves es $\frac{n * (n - 1)}{2}$ o la función $O(n^2)$.

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

16.4.2. Análisis de la ordenación por burbuja

Como se observó al describir el método de ordenación por burbuja se requiere como máximo $n - 1$ pasadas a través de la lista o el array. Durante la pasada 1, se hacen $n - 1$ comparaciones y como máximo $n - 1$ intercambios; durante la pasada 2, se hacen $n - 2$ comparaciones y como máximo $n - 2$ intercambios. En general, durante la pasada i , se hacen $n - i$ comparaciones y a lo más $n - i$ intercambios. Por consiguiente, en el peor caso, habrá un total de

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

comparaciones y el mismo número de intercambios. Recuerde que cada intercambio requiere tres movimientos de datos. Por consiguiente, existen al menos

$$\frac{3 * n * (n - 1)}{2} = \frac{3}{2} n^2 - \frac{3}{2} n$$

operaciones importantes en el peor de los casos. Por consiguiente, la eficiencia del algoritmo de burbuja en el peor de los casos es $O(n^2)$.

El mejor caso sucede cuando los datos originales están ya ordenados. Por consiguiente, el número de movimientos de claves es cero. A pesar de ello, el número de comparaciones sigue siendo de orden n^2 , $O(n^2)$, ya que los dos bucles anidados deben iterarse.

16.4.3. Análisis de la ordenación por inserción

Un procedimiento Pascal que realiza una ordenación por inserción en un array de n elementos puede ser:

```
procedure OrdenarInsercion (var A: TipoArray; n: integer);
...
begin
  for NoOrdenado := 2 to n do
  begin
    ...
    while (Pos > 1) and (A[Pos-1] > ItemSig) do
    begin
      A[Pos] := A[Pos-1];
      Pos := pred(Pos)
    end;
    ...
  end
  ...
end;
```

El bucle **for** del procedimiento **OrdenarInsercion** se ejecuta $n - 1$ veces. Dentro de este bucle existe un bucle **while** que se ejecuta a lo más el valor de **NoOrdenado**-1 veces para valores de **NoOrdenado** que están en el rango 2 a n . Por consiguiente, en el peor de los casos, las comparaciones del algoritmo vienen dadas por

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

$$\text{es decir, } 1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2}$$

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

$$n * (n - 1) = n^2 - n$$

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 inefficiencia.

16.4.4. Ordenación por mezcla

El algoritmo de ordenación por mezcla emplea la técnica de *divide y vencerás* para ordenar los n elementos de una lista a . Recordemos que el algoritmo consistía en:

1. Dividir la lista en dos mitades.
2. Ordenar cada mitad.
3. Mezclar las dos mitades.

El algoritmo de mezcla ordena las dos mitades por llamadas recursivas al algoritmo y a continuación mezcla las dos mitades para obtener un nuevo array ordenado de elementos. Un procedimiento *OrdenarMezcla* es

```
procedure Ordenar (var A: TipoArray; Primero, Central, Ultimo: Indice);
...
procedure OrdenarMezcla (var A: TipoArray; Primero, Ultimo: Indice);
...
var
  Central : Indice;
begin {OrdenarMezcla}
  if Primero < Ultimo then
    begin
      Central := (Primero + Ultimo) div 2;
      {Ordenar cada mitad}
      OrdenarMezcla (A, Primero, Central);
      OrdenarMezcla (A, Central + 1, Ultimo);
      {Mezclar las dos mitades ordenadas}
      Mezclar (A, Primero, Central, Ultimo);
    end;
end;
```

El algoritmo *OrdenarMezcla* es un recursivo que requiere determinar el tiempo empleado por cada una de las tres fases del algoritmo *divide y vencerás*. El algoritmo se ejecuta de modo similar para todas las entradas; de modo que las consideraciones se aplicarán a los tiempos de ejecución en el caso medio y en el caso peor.

Así, consideremos el tiempo empleado por el procedimiento *Mezclar*. Cuando se llama a *Mezclar* se deben mezclar las dos listas más pequeñas en una nueva lista con los n elementos. El procedimiento *Mezclar* hace una pasada a cada una de las sublistas. Por consiguiente, el número de operaciones realizadas por *Mezclar* aparece a lo máximo como el producto de una constante multiplicada por n . Si se consideran las llamadas recursivas se tendrá entonces que el procedimiento *OrdenarMezcla* implicará:

$$\text{constante} * n * (\text{profundidad de llamadas recursivas})$$

El tamaño de la lista a ordenar se divide por dos en cada llamada recursiva, de modo que la profundidad de las llamadas recursivas es aproximadamente igual al nú-

mero de veces que n se puede dividir por dos, parándose cuando el resultado es menor o igual a uno. El tiempo de ejecución viene expresado por la relación de recurrencia siguiente:

$$T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ 2T(n/2) + O(n) & \text{si } n > 1 \end{cases}$$

Estas funciones $T(n)$ dependen de $\log_2 n$, de modo que la profundidad de llamadas recursivas es aproximadamente $\log_2 n$. Por consiguiente, la ordenación por mezcla ejecutará aproximadamente el siguiente número de operaciones: alguna constante multiplicada por n y después multiplicada por $\log_2 n$. En otras palabras, la ordenación por mezcla tiene un tiempo de ejecución de $O(n \log n)$ operaciones, en donde $n \log n$ es la notación de n multiplicada por $\log n$.

Ordenación por mezcla tiene un tiempo de ejecución en el caso peor de $O(n \log n)$ operaciones, que es más rápida que los tiempos de ejecución $O(n^2)$ de los algoritmos de ordenación e inserción. La tabla muestra la comparación de tiempos de ejecución de funciones n^2 y $n \log_2 n$, donde se aprecia que el tiempo de ejecución $n \log_2 n$ es sustancialmente más rápido que el tiempo de ejecución n^2 , especialmente si la lista (el array) es grande.

16.4.5. Ordenación rápida

El algoritmo de ordenación rápida divide el array a en dos subarrays (sublistas) que se pueden ordenar de modo independiente. Se selecciona un elemento específico del array $a[p]$ llamado *pivot* y se divide el array original en dos subarrays que se ordenarán de modo independiente mediante llamadas recursivas del algoritmo. En esencia, el algoritmo *OrdenarRapido* se realiza con un procedimiento similar a:

Tabla 16.1. 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

```

procedure OrdenarRapido (var A: TipoArray; i, j: integer);
...
  if i < j then
  begin
    Particion (A,i,j,k)
    OrdenarRapido (A,i,k-1)
    OrdenarRapido (A, k+1,j)
  end;

```

donde i, j son los índices inicial y final, respectivamente, de la porción del array que se desea ordenar, y k es el índice del elemento pivote.

Cada llamada recursiva al procedimiento `OrdenarRapido` incluye una llamada al procedimiento `Particion` más un número constante de otras operaciones. El procedimiento `Particion` requiere una pasada a través del array y esto es un algoritmo $O(n)$. Sin embargo, $O(n)$ más una constante constituyen operaciones $O(n)$. Eso significa que el número total de operaciones para cada nivel de recursión son operaciones $O(n)$. Por consiguiente, la ordenación rápida tiene un tiempo de ejecución que es $O(n)$ veces la profundidad de recursión. En otras palabras, la ordenación rápida tiene un tiempo de ejecución que es

$$(constante) * n * (\text{profundidad de recursión})$$

La profundidad de recursión será el número de veces que se puede dividir n por la mitad. El número de veces que se puede dividir n por 2 es aproximadamente $\log_2 n$. El algoritmo ejecuta $O(n)$ operaciones en cada nivel de recursión, y entonces el algoritmo de ordenación rápida se ejecutará en el tiempo $O(n \log n)$ y se comporta como un algoritmo $O(n \log n)$ en la mayoría de los casos. Ordenación rápida tiene un tiempo de ejecución en el caso medio que es $O(n \log n)$.

En el peor de los casos y si las particiones están muy desequilibradas, habrá n niveles de llamadas recursivas; dado que la ordenación rápida ejecuta $O(n)$ operaciones en cada nivel de recursión, es, por consiguiente, un algoritmo $O(n^2)$ en el peor de los casos.

Tiempo de ejecución de ordenación rápida

El tiempo medio de ejecución es $O(n \log n)$ y en el peor de los casos es $O(n^2)$.

16.4.6. Análisis del método *Radix Sort*

Suponemos que el vector de registros V tiene n elementos. Al ser el campo clave entero el número de urnas es $d = 10$. Además, el número de dígitos de que consta el campo clave va a ser K . Con estas premisas y teniendo en cuenta los dos bucles anidados de que consta el algoritmo principal, tenemos que el tiempo de ejecución es $O(K * n + K * d)$.

Si las claves se consideran como cadenas binarias de longitud $\log(n)$ entonces $K = \log(n)$ y el método *Radix Sort* tomará un tiempo de ejecución $O(n \log n)$.

16.5. TABLAS COMPARATIVAS DE TIEMPOS

Tabla 16.2. Búsqueda binaria y secuencial (casos peores). (Búsqueda de una clave mayor que cualquiera del array.)

Tamaño del array n	Número de comparaciones clave	
	Búsqueda binaria ($2 \cdot \text{Trunc}(\log_2 n)$)	Búsqueda secuencial $n + 1$
100	12	101
1.000	18	1.001
10.000	26	10.001
100.000	32	100.001

Tabla 16.3. Búsqueda binaria y secuencial (tiempos de ejecución en el peor de los casos)

Tamaño del array n	Búsqueda binaria		Búsqueda secuencial
	Recursiva	Iterativa	Iterativa
100	1,6	0,5	2,4
1.000	2,2	0,8	24
10.000	2,9	1,0	240

Tabla 16.4. Intercambios y comparaciones clave en ordenación por selección

n	Tiempo de intercambio proporcional a n	Tiempo de comparación proporcional a n^2
100	100	10.000
1.000	1.000	1.000.000
10.000	10.000	100.000.000
100.000	100.000	10.000.000.000

Tabla 16.5. Número de comparaciones clave requeridas en el mejor de los casos

n	Quicksort, $O(n \log_2 n)$	Selección Sort, $O(n^2)$
100	$O(664)$	$O(10.000)$
1.000	$O(9.966)$	$O(1.000.000)$
10.000	$O(132.877)$	$O(100.000.000)$
100.000	$O(1.660.964)$	$O(10.000.000.000)$

Tabla 16.6. Tiempos de ejecución de ordenación de arrays de enteros (en segundos)

<i>n</i>	<i>Quicksort</i>	<i>Ordenación por selección</i>
250	0,3	0,6
500	0,8	2,2
1.000	1,6	8,2
2.000	3,4	32,0
4.000	8,0	126,1

RESUMEN

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

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

El análisis de algoritmos debe ser independiente de implementaciones específicas, computadoras y datos. Mediante el análisis de algoritmos es posible 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 por personal que no sea el diseñador.
3. No existe otro algoritmo que resuelva el problema utilizando menos recursos.

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

Hoy día la eficiencia en lo relativo al aspecto práctico del binomio *tiempo × memoria* no es tan relevante como hace años, debido esencialmente a que en la actualidad, y cada día más, la potencia y la memoria de las computadoras crece y crece sin límite aparente. Sin embargo, este crecimiento incita a la resolución de problemas cada vez más complejos, lo que obliga a considerar la cuidadosa administración de los recursos de la computadora, lo que conlleva la vigencia del estudio de la eficiencia.

La evaluación de la memoria no suele ser muy compleja, aunque es preciso considerar la memoria estática y la memoria dinámica. La memoria estática se medirá mediante el cálculo de las variables de tipos no estructurados, campos de los registros y los componentes de los vectores. Si se utiliza memoria dinámica dependerá de la cantidad de datos y del funcionamiento del programa. Se analiza de forma similar a como se hace con el tiempo.

Para la evaluación de tiempo se suele suponer una computadora ideal. La cantidad de tiempo, llamada complejidad, será una función $T(n)$, donde n hace referencia a una medida de los datos. Es decir, $T_A(n)$ 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 $T(n^2)$ (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 .

Tasas de crecimiento aproximadas con respecto a diferentes métodos de ordenación

	Caso peor	Caso medio
Ordenación por selección	n^2	n^2
Ordenación por burbuja	n^2	n^2
Ordenación por inserción	n^2	n^2
Ordenación por mezcla	$n \cdot \log n$	$n \cdot \log n$
Ordenación rápida (quicksort)	n^2	$n \cdot \log n$
Ordenación radix	$n \cdot \log n$	$n \cdot \log n$
Ordenación por montículo	$n \cdot \log n$	$n \cdot \log n$

EJERCICIOS

16.1. Probar las sentencias 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 $O(r \log r)$

16.2. Probar que:

$$\begin{aligned} O(f(x) + k) &= O(f(x)) \text{ donde } k \text{ es una constante} \\ O(kf(x)) &= O(f(x)) \text{ donde } k \text{ es una constante} \\ O(f(x)) + O(g(x)) &= O(g(x)) + O(f(x)) \\ O(f(x) + O(g(x))) &= O(g(x)) \text{ si } O(f(x)) \leq O(g(x)) \end{aligned}$$

16.3. 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) $n \ln x + e^x - 6e^{-x}$
- g) $n^2 \ln n + e^{-n} - 6n^3$
- h) $x^2 \ln x + e^x - 6e^x$

16.4. ¿Qué método de ordenación elegirá para ordenar cada una de las siguientes situaciones?

- a) Una lista de 20 enteros.
- b) Una lista de 20 registros, suponiendo que las claves están en memoria y los registros en disco.

- c) Una lista de 2.000 registros en orden aleatorio.
- d) Una lista de 20.000 registros, el 95 por 100 de los cuales ya están ordenados.
- e) Una lista enlazada de 20 registros.
- f) Una lista enlazada de 20.000 registros en orden aleatorio.

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

$$O(x^k), O(x^{k+1}), O(x^{k+2}) \text{ y } (2^x)$$

16.6. Analice el tiempo de ejecución de las siguientes funciones recursivas:

- a) function f1 (n: integer): integer;


```

begin
  if n<= 1 then
    f1:= 1
  else
    f1:= n*f1(n-1)
end;
```
- b) function f2 (n:integer): integer;


```

begin
  if n<= 1 then
    f2:= 1
  else
    f2:= n*f2(n div 2)
end;
```
- c) function f3 (n:integer): integer;


```

var
  x, i:integer;
begin
  if n<= 1 then
    f3:= 1
  else begin
    x:= f3(n div 2);
    for i:= 1 to n do
      x:= x+1
    end
  end;
```
- d) function f4 (n:integer): integer;


```

var
  x, i:integer;
begin
  if n<= 1 then
    f4:= 1
  else begin
    x:= f4(n - 1);
    for i:= 1 to n do
      x:= x+1
    end
  end;
```

```

e) function f5 (n: integer): integer;
begin
  if n<= 1 then
    f5:= 1
  else
    f5:= f5(n-1)+f5(n-1)
end;

f) function f6 (n:integer): integer;
var
  x, i:integer;
begin
  if n<= 1 then
    f6:= 1
  else begin
    x:= f6(n div 2)+f6(n div 2);
    for i:= 1 to n do
      x:= x+1
    end
  end;

g) function f7 (n:integer): integer;
var
  x, i, j:integer;
begin
  if n<= 1 then
    f7:= 1
  else begin
    x:= f7(n div 2)+f7(n div 2);
    for i:= 1 to n do
      for j:= 1 to n do
        x:= x+j
    end
  end;

```

16.7. Analice el tiempo de ejecución de los siguientes fragmentos de programa:

```

a) z:= 0;
   for i:= 1 to n do
     z:= z+1;

b) i:= 1; x:= 0;
   while i<= n do
   begin
     x:= x+1;
     i:= i+3
   end;

c) i:= 1; x:= 0;
   while i<= n do
   begin
     x:= x+3;
     i:= i*3
   end;

```

- d) `i:= 1; x:= 0;
repeat
 j:= 1;
 while j <= n do
 begin
 x:= x+1;
 j:= j*2
 end;
 i:= i+1
until i > n;`
- e) `i:= 2; x:= 0;
repeat
 j:= 1;
 while j <= i do
 begin
 x:= x+3;
 j:= j*2
 end;
 i:= i+1
until i > n;`
- f) `x:= 0;
for i:= 1 to n do
for j:= 1 to n do
 x:= x+2;`
- g) `x:= 0;
for i:= 1 to n do
for j:= 1 to n do
for k:= 1 to j do
 x:= x+2;`
- h) `x:= 0;
for i:= 1 to n do
for j:= 1 to n do
for k:= 1 to j do
 x:= x+2;`
- i) `x:= 0;
for i:= 1 to n do
for j:= 1 to i do
 for k:= 1 to j do
 x:= x+2;`

16.8. Escriba procedimientos y funciones para resolver los siguientes supuestos y calcule su tiempo de ejecución:

- Recorrer un árbol en inorder, preorden y postorden.
- Calcular el máximo común divisor de dos números enteros positivos.
- Calcular el número combinatorio

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

- d) Encontrar el menor elemento de un vector de n elementos.
- e) Encontrar el k -ésimo elemento de un vector de n datos. El elemento buscado es el que ocuparía la posición k en orden ascendente si el vector estuviera ordenado.
- f) Evaluar un polinomio de grado n en el punto x_0 .
- g) Calcular la suma de dos matrices cuadradas de orden $n \times n$.
- h) Calcular el producto de dos matrices cuadradas de orden $n \times n$.
- i) Resolver un sistema de n ecuaciones con n incógnitas por el método de Gauss (triangularización de matrices).

PROBLEMAS

- 16.1.** Suponga un algoritmo que tarda 5 segundos para resolver un ejemplar de un determinado problema de tamaño $n = 10.000$. ¿Cuánto tardará en resolver un ejemplar de tamaño $n = 30.00$ para los casos:
- a) $O(n^2)$.
 - b) $O(n^3)$.
 - c) $O(n \log(n))$.
 - d) $O(2^n)$.
 - e) $O(5^n)$.
- 16.2.** Un algoritmo A resuelve un problema de tamaño n en tiempo n^2 horas. Otro algoritmo B resuelve el mismo problema de tamaño n en n^3 milisegundos. Determine el umbral n_0 a partir del cual el algoritmo A es mejor que el B.
- 16.3.** Sean X, Y dos números enteros positivos grandes de tamaño n (n dígitos):
- a) Escriba un algoritmo que calcule $Z = X + Y$. Calcule su complejidad.
 - b) Escriba un algoritmo que calcule $Z = X * Y$. Calcule su complejidad.
 - c) Escriba un algoritmo que calcule $Z = X * Y$ en tiempo $O(n \log(n))$.
- 16.4.** Responder a los siguientes supuestos:
- a) Escriba un algoritmo que calcule el elemento mayor de un vector de tamaño n en tiempo n .
 - b) Escriba un algoritmo que calcule el elemento menor de un vector de tamaño n en tiempo n .
 - c) Escriba un algoritmo que calcule el elemento mayor y menor de un vector de tamaño n en tiempo $3/2 * n + c$, siendo c una constante.
- 16.5.** Escriba un algoritmo que calcule el elemento que ocupa la posición k de un vector desordenado de tamaño n :
- a) En tiempo $O(n * \log(n))$.
 - b) En tiempo esperado $O(n)$, aunque en el peor caso pueda ser $O(n^2)$.
- 16.6.** Resolver los siguientes supuestos:
- a) Escriba un algoritmo recursivo que calcule el determinante de una matriz cuadrada de orden n . Calcule su complejidad.
 - b) Aplique el método de Gauss de resolución de sistemas de ecuaciones para obtener un algoritmo que calcule el determinante en tiempo $O(n^3)$.

- 16.7. En los siguientes problemas de cálculo numérico determinar su complejidad:
- Escriba un algoritmo recursivo que calcule la inversa de una matriz cuadrada de orden n . Calcule su complejidad.
 - Use el método de resolución de sistemas de ecuaciones de Gauss-Jordan para escribir un algoritmo que calcule la inversa en tiempo $O(n^3)$.
- 16.8. Calcule la complejidad de los siguientes algoritmos sobre listas enlazadas:
- Inserción de un elemento en una lista enlazada ordenada.
 - Borrado de un elemento en una lista enlazada ordenada.
 - Búsqueda de un elemento en una lista enlazada ordenada.
- 16.9. Suponga una lista enlazada ordenada implementada en dos *arrays* paralelos con los campos de información y siguiente. Escriba un algoritmo que encuentre la posición que ocupa el elemento x en tiempo $O(n^{1/2})$. *Nota:* debe usar un algoritmo probabilista.
- 16.10. Calcule la complejidad de los algoritmos que resuelvan los siguientes problemas recursivos.
- Problema de buscar la salida en laberinto representado por un tablero de $n \times n$.
 - Problema del salto de caballo en un tablero de $n \times n$.
 - Problema de situar n reinas sin «comerse» en un tablero de $n \times n$.
 - Problema de la selección óptima de n pesos que no superen un volumen V.
- 16.11. Determinar la complejidad de las siguientes operaciones sobre árboles implementados con punteros:
- Calcule la complejidad de los algoritmos de inserción y borrado de claves de un árbol binario de búsqueda, en los casos peor, mejor y medio.
 - Calcule la complejidad de los algoritmos de inserción y borrado en árboles AVL.
 - Calcule la complejidad de los algoritmos de inserción y borrado en árboles B.
- 16.12. Calcule la complejidad de los algoritmos de las siguientes operaciones de tratamiento de grafos:
- Recorrido de la anchura de un grafo.
 - Recorrido en profundidad. Tanto recursivo como iterativo.
- 16.13. Calcule la complejidad de los siguientes algoritmos fundamentales sobre grafos:
- Encontrar una ordenación topológica de un grafo sin ciclos.
 - Determinar los caminos mínimos de un grafo valorado, aplicando el algoritmo de Dijkstra.
 - Determinar la matriz de caminos, aplicando el algoritmo de Warshall.
 - Determinar el árbol de expansión de coste mínimo, aplicando el algoritmo de Kruskal.
 - Determinar el árbol de expansión de coste mínimo, aplicando el algoritmo de Prim.
 - Determinar la función de flujo máximo, aplicando el algoritmo de Ford-Fulkerton.

REFERENCIAS BIBLIOGRÁFICAS

- Brassard, G., y Bratley, P.: *Fundamentos de algoritmia*, Madrid, Prentice-Hall, 1997.
- Heileman, Gregory L.: *Estructuras de datos, algoritmos y programación orientada a objetos*, Madrid, McGraw-Hill, 1997.
- Carrano, Y. M.; Helman, P., y Veroff, R.: *Data structures and problem solving with Turbo Pascal*, California, Benjamin/Cummings, 1993.
- Kruse, Robert L.: *Data structures and program design*, Third edition. Englewood Cliffs, New Jersey, Prentice-Hall, 1994.
- Koffman, Elliot B., y Maxim, Bruce R.: *Software Design and Data Structures in Turbo Pascal*, Reading Massachusetts, Addison-Wesley, 1994.
- Salmon, William J.: *Structures and abstractions*, Boston, M. A. Irwinm, 1991.
- Shaffer, Clifford A.: *A Practical Introduction to Data Structures and Algorithm Analysis*, Upper Saddle River, Prentice-Hall, 1992.
- Parson, Thomas W.: *Introduction to algorithms in Pascal*, New York, Wiley, 1995.
- Main, M., y Savitch, W.: *Data Structures and Other Objects*, Turbo Pascal edition, Redwood City, California, Benjamin/Cummings, 1995.

Archivos (ficheros). Fundamentos teóricos

CONTENIDO

- 17.1. Noción de archivo. Estructura jerárquica.
- 17.2. Conceptos, definiciones y terminología.
- 17.3. Organización de archivos.
- 17.4. Operaciones sobre archivos.
- 17.5. Gestión de archivos.
- 17.6. Mantenimiento de archivos.
- 17.7. Procesamiento de archivos. Problemas resueltos.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Los archivos son estructuras de datos muy importantes. Los programas utilizan archivos para comunicarse con el mundo exterior y con otros. Los valores de los datos de entrada se leen desde archivos de datos y los resultados se escriben en un archivo que puede ser visualizado.

Los archivos pueden clasificarse en *archivos externos* y *archivos internos*. Los archivos externos son aquellos que se utilizan para comunicarse con las personas o programas y se almacenan externamente al programa. Los archivos internos son aquellos que se crean pero que no se guardan.

En el capítulo se estudian los conceptos fundamentales de archivos, junto a los métodos y procedimientos para su manipulación.

17.1. NOCIÓN DE ARCHIVO. ESTRUCTURA JERÁRQUICA

Las estructuras de datos enunciadas en los capítulos anteriores se encuentran almacenadas en la memoria central o principal. Este tipo de almacenamiento, conocido por *almacenamiento principal o primario*, tiene la ventaja de su pequeño tiempo de acceso y,

además, que este tiempo necesario para acceder a los datos almacenados en una posición es el mismo que el tiempo necesario para acceder a los datos almacenados en otra posición del dispositivo —memoria principal—. Sin embargo, no siempre es posible almacenar los datos en la memoria central o principal de la computadora, debido a las limitaciones que su uso plantea:

- La cantidad de datos que puede manipular un programa no puede ser muy grande debido a la limitación de la memoria central de la computadora.
- La existencia de los datos en la memoria principal está supeditada al tiempo que la computadora está encendida y el programa ejecutándose (tiempo de vida efímero). Esto supone que los datos desaparecen de la memoria principal cuando la computadora se apaga o se deja de ejecutar el programa.

Estas limitaciones dificultan:

- La manipulación de gran número de datos, ya que —en ocasiones— pueden no caber en la memoria principal.
- La transmisión de salida de resultados de un programa pueda ser tratada como entrada a otro programa.

Para poder superar estas dificultades se necesitan dispositivos de almacenamiento secundario (memorias externas o auxiliares) como cintas, discos magnéticos, etc., donde se almacenará la información o datos que podrá ser recuperada para su tratamiento posterior. Las estructuras de datos aplicadas a colecciones de datos en almacenamientos secundarios se llaman *organización de archivos*.

La noción de *archivo o fichero* está relacionada con los conceptos de:

- Almacenamiento permanente de datos.
- Fraccionamiento o partición de grandes volúmenes de información en unidades más pequeñas que puedan ser almacenadas en memoria central y procesadas por un programa.

Un *archivo o fichero* es un conjunto de datos estructurados en una colección de entidades elementales o básicas denominadas *registros o artículos*, que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajo denominadas *campos*.

17.1.1. Campos

Los caracteres se agrupan en *campos* de datos. Un *campo* es un *item o elemento de datos elementales*, tales como un nombre, número de empleados, ciudad, número de identificación, etc.

Un campo está caracterizado por su tamaño o longitud y su tipo de datos (cadena de caracteres, entero, lógico, etc.). Los campos pueden incluso variar en longitud. En la mayoría de los lenguajes de programación los campos de longitud variable no están soportados y se suponen de longitud fija.

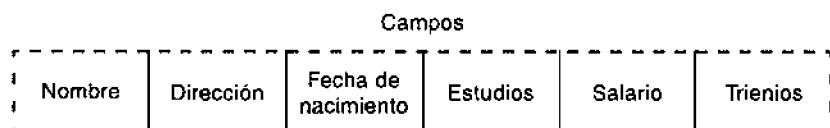


Figura 17.1. Campos de un registro.

Un campo es la unidad mínima de información de un registro.

Los datos contenidos en un campo se dividen con frecuencia en *subcampos*; por ejemplo, el campo fecha se divide en los subcampos día, mes, año.

<i>campo</i>	0	7	0	7	1	9	9	5
<i>subcampo</i>	día		mes			año		

Los rangos numéricos de variación de los subcampos anteriores son:

$$1 \leq \text{día} \leq 31$$

$$1 \leq \text{mes} \leq 12$$

$$1 \leq \text{año} \leq 1987$$

17.1.2. Registros

Un *registro* es una colección de información, normalmente relativa a una entidad particular. Un registro es una colección de campos lógicamente relacionados, que pueden ser tratados como una unidad por algún programa. Un ejemplo de un registro puede ser la información de un determinado empleado que contiene los campos de nombre, dirección, fecha de nacimiento, estudios, salario, trienios, etc.

Los registros pueden ser todos de *longitud fija*; por ejemplo, los registros de empleados pueden contener el mismo número de campos, cada uno de la misma longitud para nombre, dirección, fecha, etc. También pueden ser de *longitud variable*.

Los registros organizados en campos se denominan *registros lógicos*.



Figura 17.2. Registro.

Archivos (ficheros)

Un *fichero (archivo)* de datos —o simplemente un *archivo*— es una colección de registros relacionados entre si 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 fichero de nóminas de una empresa, inventarios, stocks, etc.

La Figura 17.3 recoge la estructura de un archivo correspondiente a los suscriptores de una revista de informática.

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.

Bases de datos

Una colección de archivos a los que puede accederse por un conjunto de programas y que contienen todos ellos datos relacionados constituye una base de datos. Así, una base de datos de una universidad puede contener archivos de estudiantes, archivos de nóminas, inventarios de equipos, etc.

Estructura jerárquica

Los conceptos carácter, campo, registro, archivo y base de datos son *conceptos lógicos* que se refieren al medio en que el usuario de computadoras ve los datos y cómo se organizan éstos. Las estructuras de datos se organizan de un modo jerárquico, de modo que el nivel más alto lo constituye la base de datos y el nivel más bajo el carácter.

17.2. CONCEPTOS, DEFINICIONES Y TERMINOLOGÍA

Aunque en el apartado anterior ya se han comentado algunos términos relativos a la teoría de archivos, en este apartado se enunciarán todos los términos más utilizados en la gestión y diseño de archivos.

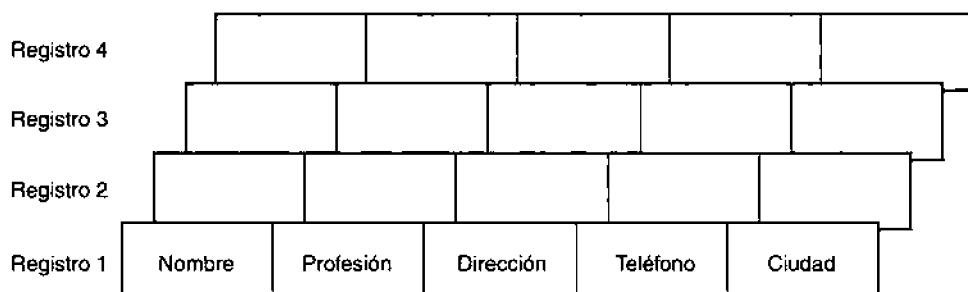


Figura 17.3. Estructura de un archivo «suscriptores».

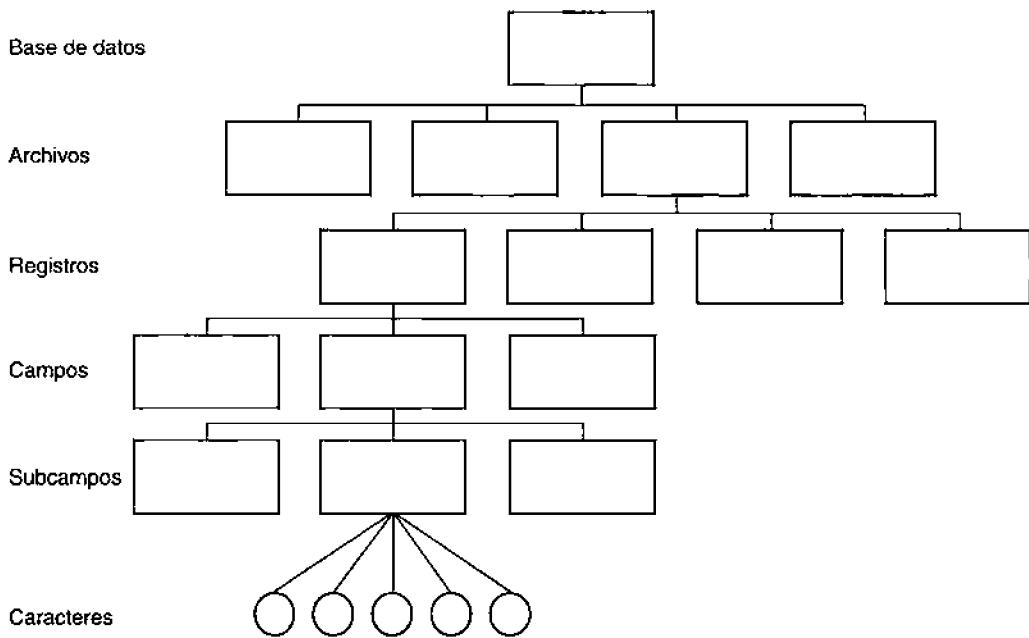


Figura 17.4. Estructuras jerárquicas de datos.

Clave (indicativo)

Una *clave (key) o indicativo* es un campo de datos que identifica únicamente al registro y lo diferencia de otros registros. Esta clave debe ser diferente para cada registro. Claves típicas son nombres o números de identificación.

Registro físico o bloque

Un *registro físico o bloque* es la cantidad más pequeña de datos que pueden transferirse en una operación de entrada/salida entre la memoria central y los dispositivos periféricos o viceversa. Ejemplos de registros físicos son: una línea de impresión, un sector de un disco magnético, etc.

Un bloque puede contener uno o más registros lógicos.

Un registro lógico puede ocupar menos de un registro físico, exactamente un registro físico o más de un registro físico.

Factor de bloqueo

Otra característica que es importante en relación con los archivos es el concepto de *factor de bloqueo o blocage*. El número de registros lógicos que puede contener un registro físico se denomina factor de bloqueo.

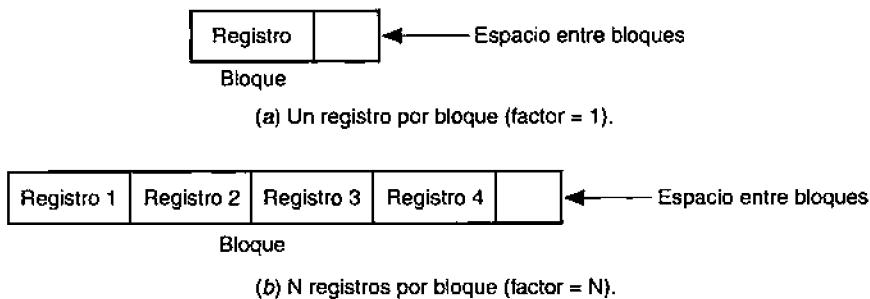


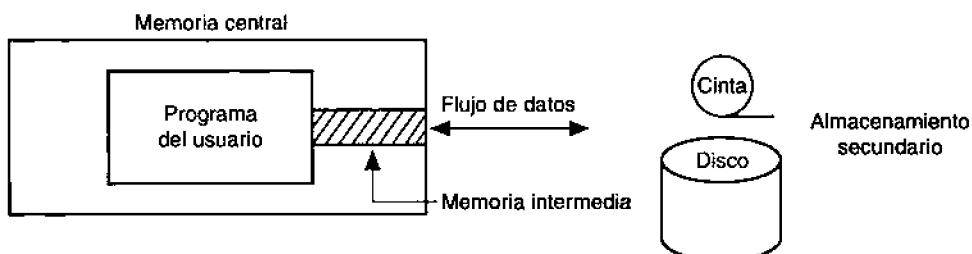
Figura 17.5. Factor de bloqueo.

Se pueden dar las siguientes situaciones:

- *Registro lógico > Registro físico*. Un registro lógico se extiende más allá de un bloque, se denominan *registros expandidos*.
- *Registro lógico = Registro físico*. El factor de bloqueo es 1 y se dice que los registros *no están bloqueados*.
- *Registro lógico < Registro físico*. El factor de bloqueo es mayor que 1 y los registros *están bloqueados*.

La importancia del factor de bloqueo se puede apreciar mejor con un ejemplo. Supongamos que se tienen dos archivos. Uno de ellos tiene un factor de bloqueo de 1 (un registro en cada bloque). El otro archivo tiene un factor de bloqueo de 10 (10 registros/bloque). Si cada archivo contiene un millón de registros, el segundo archivo requerirá 900.000 operaciones de entrada/salida menos para leer todos los registros. En el caso de las computadoras personales con un tiempo medio de acceso de 90 milisegundos, el primer archivo emplearía alrededor de 24 horas más para leer todos los registros del archivo.

Un factor de bloqueo mayor que 1 siempre mejora el rendimiento; entonces ¿por qué no incluir todos los registros en un solo bloque? La razón reside en que las operaciones de entrada/salida que se realizan por bloques se hacen a través de un área de la memoria central denominada *memoria intermedia (buffer)* y entonces el aumento del bloque implicará aumento de la memoria intermedia y, por consiguiente, se reducirá el tamaño de la memoria central.



El tamaño de la memoria intermedia de un archivo es el mismo que el del bloque. Como la memoria central es más cara que la memoria secundaria, no conviene aumentar el tamaño del bloque alegremente, sino más bien conseguir un equilibrio entre ambos criterios.

En el caso de las computadoras personales, el registro físico suele ser un sector del disco (512 bytes).

La Tabla 17.1 resume los conceptos lógicos y físicos de un registro.

17.3. ORGANIZACIÓN DE ARCHIVOS

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.

Tabla 17.1. Unidades de datos lógicos y físicos

Organización lógica	Organización física	Descripción
Carácter	Bit Byte (octeto, 8 bits)	Un dígito binario. En la mayoría de los códigos un carácter se representa aproximadamente por un byte.
Campo	Palabra	Un campo es un conjunto relacionado de caracteres. Una palabra de computadora es un número fijo de bytes.
Registro	Bloque (1 página = bloques de longitud)	Los registros pueden estar bloqueados.
Archivo	Área	Varios archivos se pueden almacenar en un área de almacenamiento.
Base de datos	Áreas	Colección de archivos de datos relacionados se pueden organizar en una base de datos.

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»).*

17.3.1. Organización secuencial

Un archivo con organización secuencial es una sucesión de registros almacenados consecutivamente sobre el soporte externo, de tal modo que para acceder a un registro n dado es obligatorio pasar por todos los $n - 1$ artículos que le preceden.

Los registros se graban consecutivamente cuando el archivo se crea y se debe acceder consecutivamente cuando se lean dichos registros.

- El orden físico en que fueron grabados (escritos) los registros es el orden de lectura de los mismos.
- Todos los tipos de dispositivos de memoria auxiliar soportan la organización secuencial.

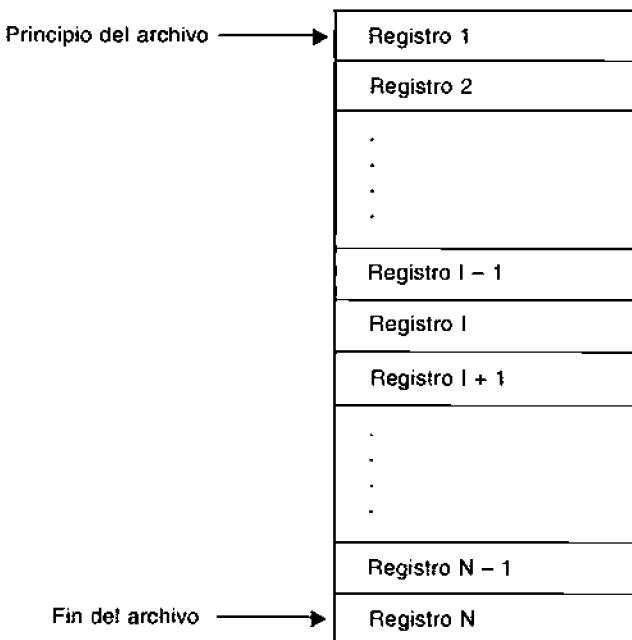


Figura 17.6. Organización secuencial.

Los archivos organizados secuencialmente contienen un registro particular —el último— que contiene una marca fin de archivo (detectable mediante las funciones EOF o bien FDA/FDF). Esta marca fin de archivo suele ser un carácter especial como '*'.

17.3.2. Organización directa

Un archivo está organizado en modo directo cuando el orden físico no se corresponde con el orden lógico. Los datos se sitúan en el archivo y se accede a ellos directamente —aleatoriamente— mediante su posición, es decir, el lugar relativo que ocupan.

Esta organización tiene la *ventaja* de que se pueden leer y escribir registros en cualquier orden y posición. Son muy rápidos de acceso a la información que contienen.

La organización directa tiene el *inconveniente* de que necesita programar la relación existente entre el contenido de un registro y la posición que ocupa. El acceso a los registros en modo directo implica la posible existencia de huecos libres dentro del soporte y, por consecuencia, pueden existir huecos libres entre registros.

La correspondencia entre clave y dirección debe poder ser programada y la determinación de la relación entre el registro y su posición física se obtiene mediante una fórmula.

Las condiciones para que un archivo sea de organización directa son:

- Almacenado en un soporte direccionable.
- Los registros deben contener un campo específico denominado *clave* que identifica cada registro de modo único, es decir, dos registros distintos no pueden tener un mismo valor de clave.
- Existencia de una correspondencia entre los posibles valores de la clave y las direcciones disponibles sobre el soporte.

En la práctica el programador no gestiona directamente direcciones absolutas, sino *direcciones relativas* respecto al principio del archivo. La manipulación de dirección relativa permite diseñar el programa con independencia de la posición absoluta del archivo en el soporte.

El programador crea una relación perfectamente definida entre la clave indicativa de cada registro y su posición física dentro del dispositivo de almacenamiento.

17.4. OPERACIONES SOBRE ARCHIVOS

Tras la decisión del tipo de organización que ha de tener el archivo y los métodos de acceso que se van a aplicar para su manipulación, es preciso considerar todas las posibles operaciones que conciernen a los registros de un archivo. Las distintas operaciones que se pueden realizar son:

- *Creación.*
- *Consulta.*
- *Actualización* (altas, bajas, modificación).
- *Clasificación.*

- *Reorganización.*
- *Destrucción (borrado).*
- *Reunión, fusión.*
- *Rotura, estallido.*

17.4.1. Creación de un archivo

Es la primera operación que sufrirá el archivo de datos. Implica la elección de un entorno descriptivo que permita un ágil, rápido y eficaz tratamiento del archivo.

Para utilizar un archivo, éste tiene que existir, es decir, las informaciones de este archivo tienen que haber sido almacenadas sobre un soporte y ser utilizables. La *creación* exige organización, estructura, localización o reserva de espacio en el soporte de almacenamiento, transferencia del archivo del soporte antiguo al nuevo.

Un archivo puede ser creado por primera vez en un soporte, proceder de otro previamente existente en el mismo o diferente soporte, ser el resultado de un cálculo o ambas cosas a la vez.

La Figura 17.7 muestra un organigrama de la creación de un archivo ordenado de empleados de una empresa por el campo clave (número o código de empleado).

17.4.2. Consulta de un archivo

Es la operación que permite al usuario acceder al archivo de datos para conocer el contenido de uno, varios o todos los registros.

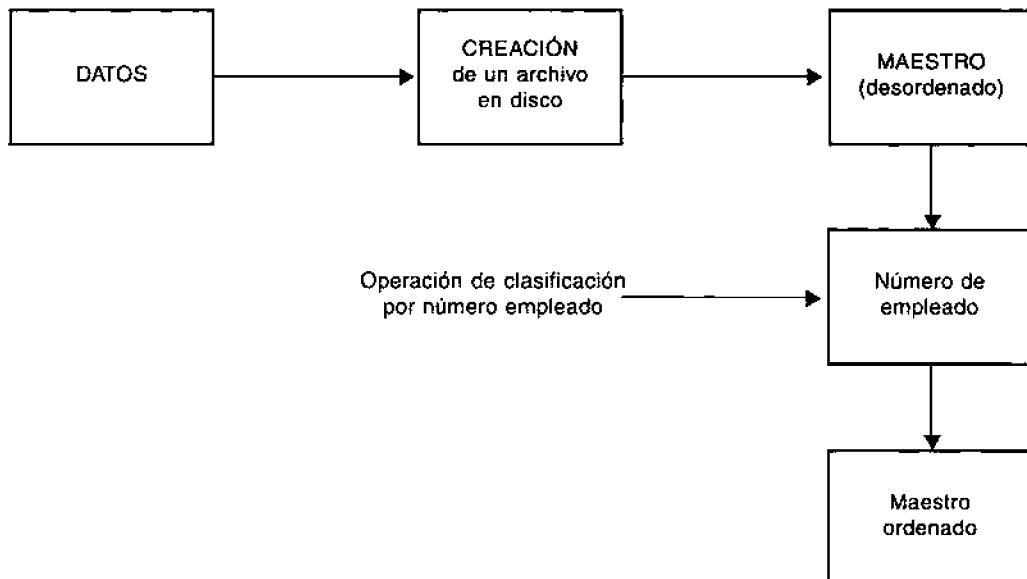


Figura 17.7. Creación de un archivo ordenado de empleados.

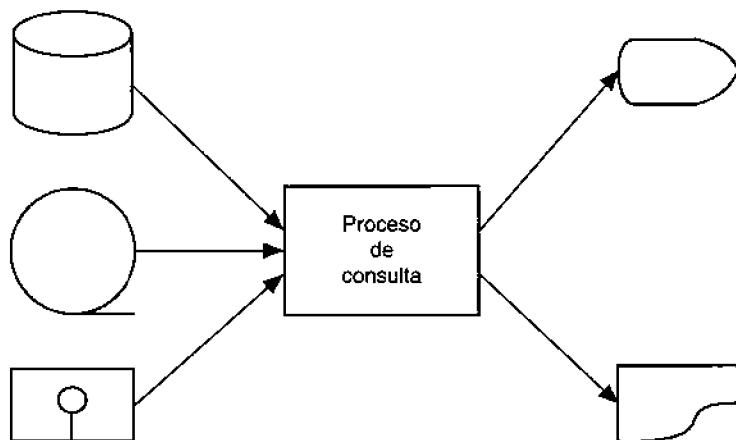


Figura 17.8. Consulta de un archivo.

17.4.3. Actualización de un archivo

Es la operación que permite tener actualizado (puesto al día) el archivo, de tal modo que sea posible realizar las siguientes operaciones con sus registros:

- Consulta del contenido de un registro.
- Inserción de un registro nuevo en el archivo.
- Supresión de un registro existente.
- Modificación de un registro.

Un ejemplo de actualización es el de un archivo de un almacén, cuyos registros contienen las existencias de cada artículo, precios, proveedores, etc. Las existencias, pre-

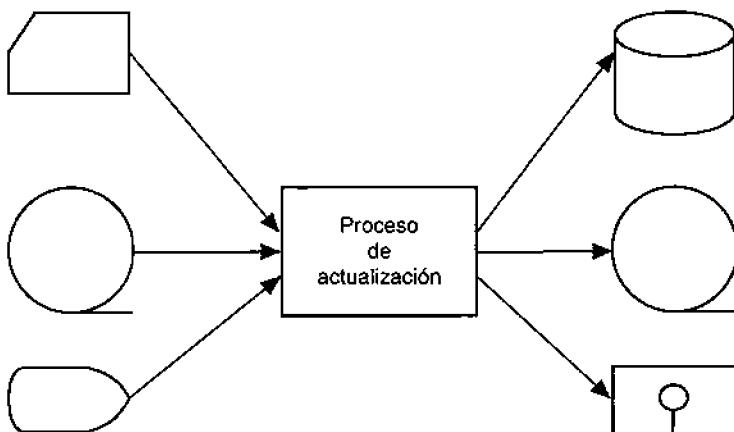


Figura 17.9. Actualización de un archivo.

cios, etc., varían continuamente y exigen una actualización simultánea del archivo con cada operación de consulta.

17.4.4. Clasificación de un archivo

Una operación muy importante en un archivo es la *clasificación u ordenación (sort, en inglés)*. Esta clasificación se realizará de acuerdo con el valor de un campo específico, pudiendo ser *ascendente* (creciente) o *descendente* (decreciente): alfabética o numérica.

17.4.5. Reorganización de un archivo

Las operaciones sobre archivos modifican la estructura inicial o la óptima de un archivo. Los índices, enlaces (punteros), zonas de sinónimos, zonas de desbordamiento, etc., se modifican con el paso del tiempo, lo que hace a la operación de acceso al registro cada vez más lenta.

La reorganización suele consistir en la copia de un nuevo archivo a partir del archivo modificado, a fin de obtener una nueva estructura lo más óptima posible.

17.4.6. Destrucción de un archivo

Es la operación inversa a la creación de un archivo (*kill, en inglés*). Cuando se destruye (anula o borra) un archivo, éste ya no se puede utilizar y, por consiguiente, no se podrá acceder a ninguno de sus registros.

17.4.7. Reunión, fusión de un archivo

Reunión. Esta operación permite obtener un archivo a partir de otros varios.

Fusión. Se realiza una fusión cuando se reúnen varios archivos en uno solo, intercalándose unos en otros, siguiendo unos criterios determinados.

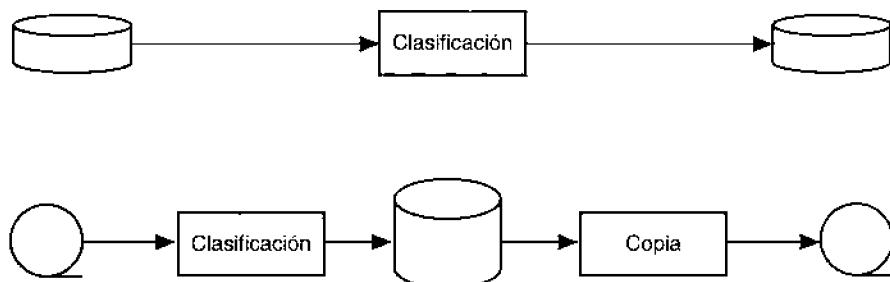


Figura 17.10. Clasificación de un archivo.

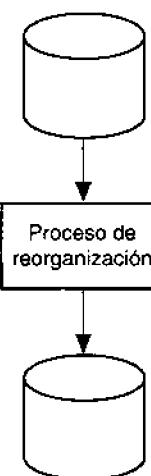


Figura 17.11. Reorganización de un archivo.

17.4.8. Rotura/estallido de un archivo

Es la operación de obtener varios archivos a partir de un mismo archivo inicial.

17.5. GESTIÓN DE ARCHIVOS

Las operaciones más usuales en los registros son:

- *Consulta*: lectura del contenido de un registro.
- *Modificación*: alterar la información contenida en un registro.
- *Inserción*: añadir un nuevo registro al archivo.
- *Borrado*: suprimir un registro del archivo.

Las operaciones sobre archivos se realizan mediante programas, de modo tal que los archivos se identifican por un nombre externo al que están asociados. Pueden existir

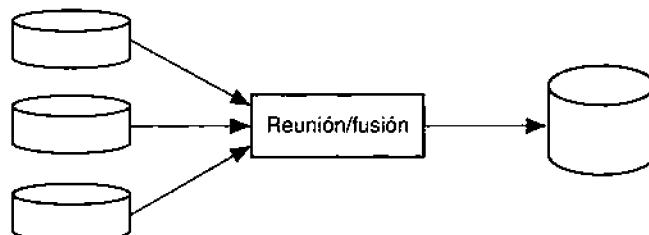


Figura 17.12. Fusión de archivos.

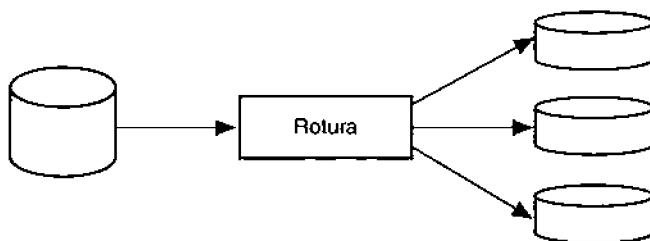


Figura 17.13. Rotura de un archivo.

programas que procesen el mismo archivo de datos. La mayoría de los programas ejecutarán las siguientes clases de funciones:

- **Crear archivos (*create*).**
- **Abrir o arrancar (*open*)** un archivo que fue creado con anterioridad a la ejecución de este programa.
- **Incrementar o ampliar el tamaño del archivo (*append, extend*).**
- **Cerrar el archivo** después que el programa ha terminado de utilizarlo (*close*).
- **Borrar (*delete*)** un archivo que ya existe.
- **Transferir datos desde (*leer*) o a (*escribir*)** el dispositivo diseñado por el programa.

Antes de que cualquier dato pueda ser escrito o leído de un archivo, el archivo debe ser creado en el dispositivo correspondiente. *Las operaciones sobre archivos tratan con la propia estructura del archivo, no con los registros de datos dentro del archivo.*

Con anterioridad a la creación de un archivo se requiere diseñar la estructura del mismo mediante los campos del registro, longitud y tipo de los mismos.

Para poder gestionar un archivo mediante un programa es preciso declarar el archivo, su nombre y la estructura de sus registros. La declaración se realizará con las siguientes instrucciones:

```

archivo Nombre
registro Nombre
campo 1=tipo
campo 2=tipo
campo 3=tipo
fin.Registro

```

17.6. MANTENIMIENTO DE ARCHIVOS

La operación de mantenimiento de un archivo incluye todas las operaciones que sufre un archivo durante su vida y desde su creación hasta su eliminación o borrado.

El mantenimiento de un archivo consta de dos operaciones diferentes:

- *Actualización.*
- *Consulta.*

La **actualización** es la operación de eliminar o modificar los datos ya existentes, o bien introducir nuevos datos. En esencia, es la puesta al dia de los datos del archivo.

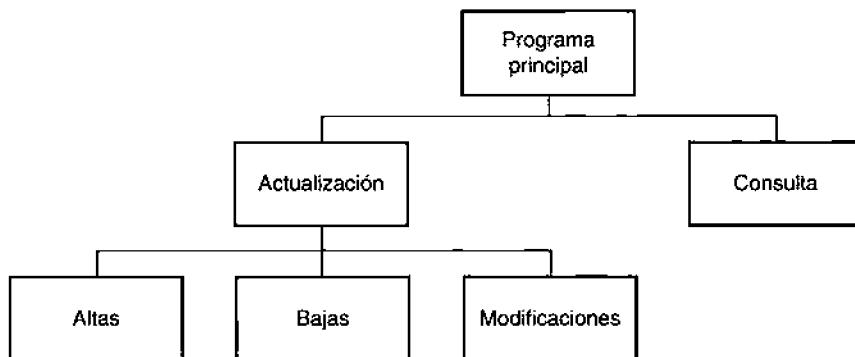
Las operaciones de actualización son:

- *Altas,*
- *Bajas,*
- *Modificaciones.*

Las operaciones de consulta tienen como finalidad obtener información total o parcial de los datos almacenados en un archivo y presentados en dispositivos de salida: pantalla o impresora, bien como resultados o como listados.

Todas las operaciones de mantenimiento de archivos suelen constituir módulos independientes del programa principal y su diseño se realiza con subprogramas (*subrutinas o procedimientos específicos*).

Así, los subprogramas de mantenimiento de un archivo constarán de:



17.6.1. Altas

Una operación de *alta* en un archivo consiste en la adición de un nuevo registro. En un archivo de empleados, un alta consistirá en introducir los datos de un nuevo empleado. Para situar correctamente un alta, se deberá conocer la posición donde se desea almacenar el registro correspondiente: al principio, en el interior o al final de un archivo.

El algoritmo del subprograma ALTAS debe contemplar la comprobación de que el registro a dar de ALTA no existe previamente.

17.6.2. Bajas

Una *baja* es la acción de eliminar un registro de un archivo. La baja de un registro se puede presentar de dos formas distintas: indicación del registro específico que se desea dar de baja o bien visualizar los registros del archivo para que el usuario elija el registro a borrar.

La baja de un registro puede ser *lógica* o *física*. Una *baja lógica* supone el no borrado del registro en el archivo. Esta baja lógica se manifiesta en un determinado campo del registro con una *bandera, indicador o «flag»* —carácter *, \$, etc.—, o bien con la escritura o rellenado de espacios en blanco en el registro específico.

Una *baja física* implica el borrado y desaparición del registro, de modo que se crea un nuevo archivo que no incluye el registro dado de baja.

17.6.3. Modificaciones

Una *modificación* en un archivo consiste en la operación de cambiar total o parcialmente el contenido de uno de sus registros.

Esta fase es típica cuando cambia el contenido de un determinado campo de un archivo; por ejemplo, la dirección o la edad de un empleado.

La forma práctica de modificar un registro es la visualización del contenido de sus campos; para ello se debe elegir el registro o registros a modificar. El proceso consiste en la lectura del registro, modificación de su contenido y escritura, total o parcial del mismo.

17.6.4. Consulta

La operación de *consulta* tiene como fin visualizar la información contenida en el archivo, bien de un modo completo, bien de modo parcial.

Las operaciones de consulta de archivo deben contemplar diversos aspectos que faciliten la posibilidad de conservación de datos. Los aspectos más interesantes a tener en cuenta son:

- Opción de visualización en pantalla o listado en impresora.
- Detención de la consulta a voluntad del usuario.
- Listado por registros o campos individuales o bien listado total del archivo (en este caso deberá existir la posibilidad de impresión de listados, con opciones de saltos de página correctos).

17.6.5. Operaciones sobre registros

Las operaciones de transferencia de datos a/o desde un dispositivo a la memoria central se realizan mediante las instrucciones:

```
leer (<var_tipo_archivo>, lista de entrada de datos)
escribir (<var_tipo_archivo>, lista de salida de datos)
```

Las operaciones de acceso a un registro y de paso de un registro a otro se realiza con las acciones *leer* y *escribir*.

17.7. PROCESAMIENTO DE ARCHIVOS. PROBLEMAS RESUELTOS

Los archivos secuenciales terminan con una marca final de archivo (**FDA** o **EOF**). Cuando se tengan que añadir registros a un archivo secuencial se añadirán en las marcas fin de archivo.

Las operaciones básicas que se permiten en un archivo secuencial son: *escribir su contenido, añadir un registro al final del archivo y consultar sus registros*. Las demás operaciones exigen una programación específica.

Los archivos secuenciales son los que ocupan menos memoria y son útiles cuando se desconoce a priori el tamaño de los datos. También son muy empleados para el almacenamiento de información cuyo contenido sufra pocas modificaciones en el transcurso de su vida útil.

PROBLEMA 17.1

Se desea realizar una aplicación que nos sirva de ejemplo sobre las operaciones básicas de archivos creación y consulta. El supuesto es el siguiente: Informatizar la gestión de una librería. El elemento básico es el libro, del que queremos la información, título, autor, código, precio y número de unidades. Las operaciones básicas que se realizarán sobre el archivo librería son las de creación y consulta.

ANÁLISIS

Para representar el elemento libro se utiliza el tipo registro con los campos autodescritos en el enunciado. El tipo registro y archivo, al igual que las rutinas para manejo del archivo, creación y consulta quedan definidas en la unidad que a continuación se muestra.

CODIFICACIÓN DE UNIDAD Archivolibros

```
unit ArchivoLibros;
interface
type
  cadena30 = string[30];
  cadena6 = string[6];
  Reglibro = record
    Titulo, Autor:cadena30;
    Código:cadena6;
    Precio:word;
    Unidades:byte
  end;
  Libreria=file of Reglibro;
  {Operaciones}
  procedure Apertura (var F:Libreria);
  procedure Libro (var L:Reglibro);
  procedure Mostrarlibro (var L:Reglibro);
  procedure Generararchivo (var F:Libreria);
  procedure Consultalibro (var F:Libreria);

implementation
  uses crt;
  procedure Apertura (var F:Libreria);
  const
    Archivo='Libros.dat'; [Nombre del fichero]
```

```

var
  Unidad:string[12];
  Nombre:string[24];
  Status:integer;
begin
  clsCr;
  write ('Unidad/Directorio Archivo Libreria:');
  readln (unidad)
  Nombre := Unidad+Archivo;
  {concatenación de unidad con nombre}
  {$I-}
  assign (F, Nombre);
  reset(F);
  Status := IoResult;
  if (Status <> 0) then
    begin {Archivo no existe va a ser creado}
      rewrite (F);
      Status := IoResult;
      if (Status <> 0) then
        writeln ('hay un error en el archivo, revisa el path.')
      else
        writeln ('Archivo de libreria va a ser creado')
    end
  else
    writeln ('Archivo',Nombre,';dispuesto para consultas.')
  {$I+}
end;

procedure Libro (var L:Registro);
{Asigna interactivamente los campos de un libro}
begin
  with L do
  begin
    write ('Introduzca titulo:'); readln (Titulo);
    write ('Introduzca autor:'); readln (Autor);
    write ('Introduzca codigo:'); readln (Codigo);
    write ('Introduzca precio:'); readln (Precio);
    write ('Introduzca unidades:'); readln (Unidades);
  end
end;

procedure Generarchivo (var F:Libreria);
{Crea el archivo libreria}
var
  Masdatos:char;
  L:Reglibro;
begin
  repeat
    Libro(L); Write(F, L);
    gotoxy(2,23); ClnEol;
    Write ('Mas libros(s-n):');readln(Masdatos)
  until upcase (Masdatos)<>'S';
  close(F)
end;

```

```

procedure Mostrarlibro (var L:Reglibro);
  {Muestra los datos de un libro}
begin
  ClrScr;
  with L do
  begin
    WriteLn ('Titulo:', Titulo);
    WriteLn ('Autor:', Autor);
    WriteLn ('Codigo:', Codigo);
    WriteLn ('Precio:', Precio);
    Writeln ('Unidades:', Unidades)
  end
end;

procedure Consultalibro (var F:Libreria);
  {Busca un libro conociendo el título del mismo}
var
  Hallado:boolean;
  Libro:Reglibro;
  Titulo:Cadena30;
begin
  ClrScr;
  Write ('Introduzca titulo a consultar:'); readln (Titulo);
  reset(F);
  Hallado := false;
  while not eof(F) and not Hallado do
  begin
    read (F, Libro);
    Hallado := Titulo = Libro.Titulo
  end;
  if Hallado then
    Mostrarlibro(Libro)
  else
    WriteLn ('No existe el libro', titulo, 'en la librería');
  close(F);
end;

begin
end.

```

PROBLEMA 17.2. Creación de una librería

Con la unidad Archivo se puede escribir un programa que permita crear el archivo de la librería y optativamente consultar libros, o bien un listado de toda la librería. En este programa no se contempla la posibilidad de dar nuevas altas, ni de modificar registros.

```

program Biblioteca;
uses
  Crt, ArchivoLibros;
var
  F:Libreria;
  procedure Pausa;

```

```

const
  Caracteres= [#0..#255];
var
  Letra:char;
begin
  GotoXY (2,23); ClrEol;
  Write ('Pulse una tecla para continuar');
  repeat
    Letra := ReadKey
    until Letra in Caracteres;
end;

procedure Listadototal (var F:Libreria);
  {Muestra los datos de toda la librería}
var
  Libro:Reglibro;
begin
  reset(F);
  while not eof(F) do
  begin
    read(F, Libro);
    Mostrarlibro(Libro);
    Pausa
  end
end;

procedure Consultar (var F:Libreria);
  {Permite consulta un libro o toda la librería}
const
  Opciones = ['1',L];
var
  Opcion:char;
begin
  ClrScr;
  repeat
    Write ('Consultar libro(1) o listar libreria(L); readln (opcion)');
  until Opcion in Opciones;
  if Opcion = '1' then
    ConsultaLibro(F)
  else
    ListadoTotal(F)
end;

begin
  ClrScr;
  Apertura(F);
  if eof(F) then
    Generararchivo(F)
  else
    Consultar(F)
end.

```

PROBLEMA 17.3

Sobre la librería creada en el anterior ejercicio se desea incorporar las siguientes operaciones, que se refieren a consultas específicas sobre los libros almacenados.

- ¿Cuántos libros de un cierto autor existen?
- ¿Cuántos libros tienen un precio entre P1 y P2 pesetas?
- ¿Cuál es el número de código de cierta obra literaria?

ANÁLISIS

La unidad en la que se encuentran los tipos de datos y las operaciones necesarias para el proceso del ejercicio anterior va a ser ampliada para contener las operaciones de búsqueda que ahora se requieren. Todos los procesos de búsqueda son secuenciales.

Unidad ArchivoLibros contiene los tipos de datos necesarios para archivo de libros. Incorpora todas las operaciones de creación y consulta.

```
unit ArchivoLibros;
interface
type
  cadena30:string[30];
  cadena6:string[6];
  Reglibro=record
    Titulo, Autor:cadena30;
    Codigo:cadena6;
    Precio:word;
    Unidades:byte
  end;
  Libreria=file of Reglibro;
  {Operaciones}
  procedure Apertura (var F:Libreria);
  procedure Libro (var L:Reglibro);
  procedure Mostrarlibro (var L:Reglibro);
  procedure Generarchivo (var F:Libreria);
  procedure Consultalibro (var F:Libreria);
  function LibrosAutor(varF:Libreria;Autor:cadena30):byte;
  function LibrosPrecio(var:Libreria;Maximo,Minimo:word):word;
  function CodTitulo(var F:Libreria;Titulo:Cadena30):Cadena6;
implementation
  uses crt;

procedure Apertura (var F:Libreria);
const
  Archivo='Libros.dat'; {Nombre del fichero}
var
  Unidad:string[12];
  Nombre:string[24];
  Status:integer;
begin
  clrscr;
  write ('Unidad/Directorio Archivo Libreria:');
  readln (Unidad);
  Nombre := Unidad + Archivo;
  {concatenación de unidad con nombre}
  {SI-}
  assign (F, Nombre);
  reset(F);
  Status := IoResult;
```

```

if (Status <> 0) then
begin {Archivo no existe va a ser creado}
  rewrite (F);
  Status := IoResult;
  if (Status <> 0 then
    writeln ('Hay un error en el Archivo, revisa el path.')
  else
    writeln ('Archivo de libreria va a ser creado')
end
else
  writeln ('Archivo', Nombre,';dispuesto para consultas.')
{$I+}
end;

procedure Libro (var L: Reglibro);
{Asigna interactivamente los campos de un libro}
begin
  with L do
  begin
    write ('Introduzca título:');readln (Titulo);
    write ('Introduzca autor:');readln (Autor);
    write ('Introduzca código:');readln (Codigo);
    write ('Introduzca precio:');readln (Precio);
    write ('Introduzca unidades:');readln (Unidades)
  end
end;

procedure Generararchivo (var F:Libreria);
{Crea el archivo libreria}
var
  Masdatos:char;
  L:Reglibro;
begin
  repeat
    Libro(L); Write(F,L);
    gotoxy(2,23); clreol;
    Write('Más libros (s-n): ');
    readln (Masdatos)
  until upcase (Masdatos) <>'S';
  close(F)
end;

procedure Mostrarlibro (var L: Reglibro);
{Muestra los datos de un libro}
begin
  ClrScr;
  with L do
  begin
    WriteLn ('Título:',Titulo);
    WriteLn ('Autor:', Autor);
    WriteLn ('Código:', Codigo);
    WriteLn ('Precio:', Precio);
    WriteLn ('Unidades:', Unidades)
  end
end;

```

```

procedure Consultalibro (var F: Libreria);
  {Busca un libro conociendo el título del mismo}
var
  Hallado:boolean;
  Libro:Registro;
  Titulo:Cadena30;
begin
  ClrScr;
  Write ('Introduzca título a consultar:');
  readln(Titulo);
  reset(F);
  Hallado := false;
  while not eof(F) and not Hallado do
  begin
    read(F, Libro);
    Hallado := Titulo = Libro.Titulo
  end;
  if Hallado then
    Mostrarlibro(Libro)
  else
    writeln ('No existe el libro', titulo, 'en la librería');
  close(F);
end;

function LibrosAutor (var F:Libreria;Autor:Cadena30):byte;
  {Indica los títulos que existen de un autor determinado}
var
  Total:byte;
  Libro:Reglibro;
begin
  Total := 0;
  reset(F);
  while not eof(F) do
  begin
    read(F, Libro);
    if Libro.Autor = Autor then
      Total := Total + 1
  end;
  close (f);
  LibrosAutor := Total
end;

function LibrosPrecio(var F:Libreria;Maximo,Minimo:word):word;
{Libros con precio comprendido entre máximo y mínimo}
var
  Total:word;
  Libro:Reglibro;
begin
  Total := 0;
  reset(F);
  while not eof(F) do
  begin
    read (F,Libro);
    with Libro do
      if (Precio<=Maximo) and (Precio>=Minimo) then
        Total := Total + 1
  end;
end;

```

```

    end;
  close(F);
  Librosprecio := Total
end;

function CodTitulo(var F:Libreria;Titulo:Cadena30):Cadena6;
{Proporciona el código de un título específico}
var
  Libro:Reglibro;
begin
  CodTitulo := 'No hay';
  reset(F);
  while not eof(F) do
  begin
    read(F, Libro);
    if Libro.Titulo = Titulo then
      CodTitulo := Libro.Codigo
  end;
  close(F)
end;
begin
end.

```

PROBLEMA 17.4

Programa de consulta de libros. Es una continuación del programa de creación del Archivo; utiliza la misma unidad: archivolibros.

```

program ConsultasLibreria;
{Realiza consultas por títulos, autores y precios de los libros de
una librería}
uses
  Archivolibros, Crt;
var
  Libros:Libreria;
  Autor, Titulo:Cadena30;
  Maximo,Minimo,Mayor:word;
  Ejemplares, Opc:byte;

procedure TipoConsulta (var Opc:byte);
begin
  gotoxy(10,3);writeln('1.Consulta por autor.');
  gotoxy(10,5);writeln('2.Consulta por título.');
  gotoxy(10,7);writeln('3.Consulta por precio.');
  gotoxy(10,9);writeln('6.Fin de consultas.');
  repeat
    gotoxy(12,11);
    write('Opcion?'); readln(Opc)
  until Opc in[1..3,6];
end;

procedure Pausa;
begin
  repeat
    gotoxy(15,22);write('Pulsa una tecla para continuar.')
  until keypressed;
end;

```

```

until ord(readkey) in [0..255]
end;
begin
  clrscr;
  Apertura(Libros);
  repeat
    clrscr;
    TipoConsulta(Opc);
    case Opc of
      1:begin
        gotoxy(5,15);write('Introduzca autor:');
        readln(Autor);
        gotoxy(5,16);
        writeln(Autor,'tiene',LibrosAutor (Libros, Autor), 'títulos')
        end;
      2:begin
        gotoxy(5,15);write('Introduzca título:'); readln(Titulo);
        gotoxy(5,16);
        writeln('Código de ', titulo.:',Codtitulo (Libros,Titulo));
        end;
      3:begin
        gotoxy(5,15);
        write('Introduzca precio mínimo:'); readln(Minimo);
        gotoxy(5,16);
        write('Introduzca precio máximo:'); readln(Maximo);
        gotoxy(5,17);
        writeln('El número de títulos con precio entre ',maximo.'y', minimo,
               LibrosPrecio(Libros, maximo, minimo));
        end;
      end
    Pausa;
  until Opc=6
end.

```

PROBLEMA 17.5

Se dispone de un archivo de Usuarios de un centro de cálculo. Cada registro tiene la información del nombre, número de identificación, clave de acceso, límite de recursos y recursos utilizados hasta la fecha; los registros están dispuestos en orden creciente del número de identificación. El archivo se debe actualizar diariamente con la Actividad del centro recogida en un archivo que tiene en cada registro los campos número de identificación y recursos utilizados por la tarea realizada; este archivo también está ordenado en orden ascendente respecto al número de identificación.

ANÁLISIS

El programa que resuelve el problema maneja dos archivos externos, Usuarios.Dat y Actividad.Dat. Estos dos archivos ya están creados y por tanto hay que abrirlos para lectura. Además, los registros ya vienen ordenados en orden creciente respecto al campo número de identificación. En la unidad CentroCal se define los tipos de datos para manejo de ambos archivos, y la operación de apertura de ambos archivos.

La modificación de los registros del archivo maestro (*Usuarios.Dat*) se realiza a la manera de archivo de acceso directo, aunque los registros ocupen posiciones consecutivas. El proceso de actualización se realiza con un bucle mientras no sea fin de archivo de Actividad. Se lee un registro del archivo Actividad, a continuación se lee del archivo Usuarios hasta encontrar un registro con un número de identificación igual, o bien que el número de identificación sea mayor. En el caso de no encontrarse el número de identificación, será visualizado el registro correspondiente del archivo Actividad en forma de un listado de errores. La actualización consiste en sumar a los recursos utilizados los recursos de la tarea realizada y volver a escribir el registro en el archivo de Usuarios.

```

unit CentrCal;
interface
type
  Tipuser=record
    Nombre:string[15];
    Numidt:integer;
    Clave:string[10];
    Limrcos, Recsos:integer
  end;
  Tipact=record
    Numidt,
    Recsos:integer
  end;
  Factivid=file of Tipact;
  Fusuario=file of Tipuser;
  procedure Apertura(var Users:Fusuario;var Activ:Factivid);

implementation
  uses crt;
  procedure Apertura(var Users:Fusuario;var Activ:Factivid);
  var
    Nombre:string[25];
    Status:integer;
    Unidad:string[20];
  begin
    clrscr;
    repeat
      write('Unidad/Camino del archivo Actividad:');
      readln(Unidad);
      Nombre := Unidad ''Activida.dat';
      {$I-}
      assign(Activ,Nombre); reset(Activ);
      status := IoResult;
      if (status<>0) then
        begin
          write('Error al abrir archivo ACTIVIDAD. ¿Nuevo intento(S/N)?:');
          {$I+}
          if upcase(ReadKey)<> 'S' then
            (aborta la ejecucion)
            reset(Activ)
        end
    until (status=0);
  end;

```

```

until status = 0;
repeat
  write('Unidad/camino del archivo Usuarios:');
  readln(Unidad);
  Nombre := Unidad + 'Usuarios.dat';
  {$I-}
  assign(Users, Nombre); reset(Users);
  status := IoResult;
  if (status<> 0) then
  begin
    write('Error al abrir archivo USUARIOS.¿Nuevo intento(S/N)?:');
    {$I+}
    if upcase(ReadKey)<> 'S' then
      (aborta la ejecución)
    reset(Users)
  end
  until status = 0
end;
begin
end.

```

PROBLEMA 17.6

A esta unidad se le puede añadir las operaciones para crear los respectivos archivos, ordenarlos, hacer consultas, etc. A continuación se presenta el programa para realizar el cometido pedido.

```

program Actualiza(input,output,usuarios,Actividad);
uses Centrcal, Crt;
var
  Regus:Tipuser;
  Regact:Tipact;
  Actividad:Factivid;
  Usuarios:Fusuario;
  Menor,Leerusr:boolean;
  I:integer;

begin
  Apertura(Usuarios,Actividad);
  {cabecera para listado de errores}
  clrscr;
  writeln('REGISTROS NO ENCONTRADOS':30);
  writeln('-----':30);
  Leerusr := true;
  while not eof(Actividad) do
  begin
    read(Actividad, Regact);
    Menor := false;
    while not Menor and not eof(Usuarios) do
    begin
      if Leerusr then
        read(Usuarios, Regus);
      if Regus.Numidt > Regact.Numidt then
      begin

```

```

Menor := true;
LeerUs := false;
writeln(Regact.Numidt:10,Regact.Recsos:10)
end
else
if Regus.numidt = Regact.Numidt then
begin
  LeerUs := true;
  Regus.Recsos := Regus.Recsos+Regact.Recsos;
  i := filepos(Usuarios) - 1;
  seek(Usuarios,i);
  {Queda apuntado al registro a modificar}
  write(Usuarios,Regus);
  menor := true
  end;
  end {fin de while not menor...}
end; {fin de while de lectura de Actividad}
close(Actividad);
close(Usuarios)
end.

```

PROBLEMA 17.7

Se dispone de un archivo de inventario cuyo nombre es Stock.dat. Cada registro tiene los campos nombre, código, cantidad, precio y fabricante. Se desea buscar un artículo por el número de código, visualizarlo en caso de encontrarse y en caso contrario, un mensaje a pantalla.

ANÁLISIS

Todo consiste en realizar una búsqueda secuencial en un archivo de datos desordenado. La búsqueda se terminará cuando se encuentre el dato buscado o cuando se llegue a final del archivo.

CODIFICACIÓN

```

program Almacen(Stock,output);
uses
  Dos, Crt;
type
  Cadena30 = string[30];
  Cadena6 = string[6];
  Regarticulo = record
    Nombre,Fabricante:Cadena30;
    Codigo:Cadena6;
    Precio:integer;
    Cantidad:byte
  end;
  Fichero = file of Regarticulo;
var
  Stock:fichero;
  Masconsultas:char;

```

```

procedure Pausa;
const
  Caracteres = [#0..#255];
var
  Letra:char;
begin
  GotoXY(2,23);
  ClrEol; Write('Pulse una tecla para continuar');
repeat
  Letra := ReadKey
  until Letra in Caracteres;
end;

procedure Creararticulo(var Articulo:regarticulo);
  {Obtiene los datos de un articulo}
begin
  ClrScr;
  with Articulo do
  begin
    Write('Introduzca nombre:'); readln(Nombre);
    Write('Introduzca fabricante:'); readln(Fabricante);
    Write('Introduzca código:'); readln(Codigo);
    Write('Introduzca precio:'); readln(Precio);
    Write('Introduzca cantidad:'); readln(Cantidad)
    end
  end;
  {Muestra los datos de un articulo}
begin
  ClrScr;
  with Articulo do
begin
  Writeln('nombre:', nombre);
  Writeln('fabricante:', fabricante);
  Writeln('código:', código);
  Writeln('precio:', precio:1);
  Writeln('cantidad:', cantidad)
end
end;

procedure Generarfichero(var F:fichero);
  {Crea el fichero de artículos}
var
  Masdatos:char;
  Articulo:regarticulo;
begin
  rewrite(F);
  repeat
    Creararticulo(Articulo);
    Write(f, Articulo);
    gotoxy(2,23);clrEol;
    Write('Más artículos(S-N);readln(Masdatos)
until upcase(Masdatos) <> 'S';
  close(F);
end;

```

```

procedure Consultar(var F:fichero);
var
  Hallado:boolean;
  Articulo:Regarticulo;
  Codigo:cadena30;
begin
  ClrScr;
  Write('Introduzca código articulo:');readln(Codigo);
  reset(f);
  Hallado := false;
  while not eof(F) and not Hallado do
  begin
    read(F,articulo);
    Hallado := Codigo=Articulo.Codigo
  end;
  if Hallado then
    Mostrararticulo(Articulo)
  else
    writeln('No existe el código',Codigo,'en almacén');
    Pausa
  end;

begin
  clrscr;
  assign(Stock, 'C:\stock.dat');
  Generarfichero(stock);
  repeat
    Consultar(stock);
    gotoxy(2,23);
    clreol;
    Write('más consultas:(s-n)');readln(Masconsultas)
  until upcase(Masconsultas)<>'S'
end.

```

PROBLEMA 17.8

Se tiene dos archivos cuyos registros mantienen un orden ascendente respecto de un campo clave. Se desea mezclar los dos archivos para formar un único archivo, también ordenado respecto del mismo campo clave.

ANÁLISIS

Por sencillez a la hora de probar la mezcla de archivos los registros son valores enteros. El procedimiento de mezclar lee los dos archivos de enteros ordenados en orden ascendente y crea el archivo mezcla cuyo contenido es la unión de los dos archivos, y debe también quedar ordenado. Si hay números repetidos se escriben también.

El procedimiento de mezcla realiza el proceso con un bucle mientras no sea fin de fichero de los dos archivos. En el bucle se lee de un archivo y/o del otro dependiendo de que se haya escrito en el archivo mezcla desde uno u otro o de los dos a la vez en la anterior iteración. Para llevar este control se utilizan dos variables de tipo boolean que

indican desde qué archivo (s) se ha escrito y por tanto de qué archivo (s) se va a leer. También se desarrolla un procedimiento para escribir el contenido de un archivo.

CODIFICACIÓN

```

program Mezcla_f(input,output,F1,F2,F3);
uses crt;
type
  TipoFich = file of integer;
var
  F1,F2,F3:TipoFich;

procedure Visualizar(var F:TipoFich);
var
  x:integer;
begin
  {Síntesis del puntero del fichero al primer registro}
  seek(F,0);
  while not eof(F) do
  begin
    read(f,x); write(x,'')
  end;
  gotoxy(1,WhereY+2)
end;

procedure Apertura;
var
  Nombre:string[25];
  Status:integer;
begin
  clrscr;
  repeat
    write('Nombre de archivoF1:');readln(Nombre);
    {$I-}
    assign(F1,Nombre);
    reset(F1);
    Status := IoResult;
    if(Status<>0) then
      begin
        write('Error en la apertura del fichero F1.', '¿Quieres intentarlo
              de nuevo?');
        {$I+}
        if upcase(ReadKey)<>'S'then
          {Aborta la ejecución}
        reset(F1)
      end
    until Status = 0;
  repeat
    write('Nombre de archivoF2:');readln(Nombre);
    {$I-}
    assign(F2,Nombre);
    reset(F2);
    Status := IoResult;
    if(Status<>0) then

```

```

begin
  write('Error en la apertura del fichero F2.', '¿Quieres intentarlo
        de nuevo?:');
  {$I+}
  if upcase(ReadKey)<>'S' then
    (Aborta la ejecución)
    reset(F2)
end
until Status = 0;
repeat
  write('Nombre del archivo F3:'), readln(Nombre);
  {$I-}
  assign (F3,NOMBRE);
  rewrite(F3);
  Status := IoResult;
  if(Status<>0) then
  begin
    write('Error en la apertura del fichero F3.', '¿Quieres intentarlo
          de nuevo?:');
    {$I+}
    if upcase(ReadKey)<>'S' then
      (Aborta la ejecución)
      rewrite(F3)
    end
    until Status = 0
  end;
  (procedimiento de mezcla)

procedure mezclar(var F1,F2,F3:TipoFich);
var
  leer_F1,leer_F2:boolean;
  x1,x2:integer;
begin
  {puntero de fichero lo situamos al comienzo}
  seek(F1,0);seek(F2,0);
  leer_F1 := true; leer_F2 := true;
  while not(eof(F1) or eof(F2)) do
  begin
    if leer_F1 then read(F1,x1);
    if leer_F2 then read(F2,x2);
    {Ahora se comparan para escribir x1 o x2, o los dos}
    if x1 < x2 then
    begin
      write(F3,x1);
      leer_F1 := true;
      leer_F2 := false
    end
    else
    if x1 > x2 then
    begin
      write(F3,x2);
      leer_F1 := false;
      leer_F2 := true
    end
    else begin {son iguales}
      write(F3,x1,x2);
    end
  end
end;

```

```

leer_F1 := true;
leer_F2 := true
end
end; {fin de lectura de los dos ficheros a la vez}
{son escritos los elementos restantes}

while not eof(F1) do {en caso de no haber terminado F1}
begin
  read(F1,x1);
  if not leer_F2 then {queda por procesar x2 de F2}
    if x1 < x2 then
      write(F3,x1)
    else begin
      write(F3,x2,x1);
      leer_F2 := false
    end
  else
    write(F3,x1)
  end;
  while not eof(F2) do {en caso de no haber terminado F2}
  begin
    read(F2,x2);
    if not leer_F1 then {queda por procesar x1 de F1}
      if x2 < x1 then
        write(F3,x2)
      else begin
        write(F3,x1,x2);
        leer_F1 := false
      end
    else
      write(F3,x2)
    end;
    if not leer_F1 then write(F3,x1);
    {Quedaba por escribir x1}
    if not leer_F2 then write(F3,x2)
    {Quedaba por escribir x2}
  end;
begin {bloque principal}
Apertura;
clrscr; gotoxy(25,1);
writeln('***Registros del archivo F1***');writeln;
visualizar(F1);
gotoxy(25,WhereY+1);
writeln('***Registros del archivo F2***');writeln;
visualizar(F2);
Mezclar(F1,F2,F3);
gotoxy(25,WhereY+2);
writeln('***Registros del archivo F3***');
visualizar(F3)
end.

```

RESUMEN

Como ya conoce el lector, las variables, los arrays, conjuntos y registros se utilizan todos para almacenar datos durante la ejecución de un programa. Existen otras estructuras de datos que sir-

ven para almacenar datos, tales como listas enlazadas y árboles. Sin embargo, ninguna de estas estructuras se pueden utilizar para almacenar y guardar datos de una ejecución de un programa a la siguiente. Todas estas variables y estructuras, existen sólo mientras el programa en que están definidas se está ejecutando; el espacio de memoria asignado a las mismas por el programa se libera cuando se termina el programa. Con el objeto de almacenar datos de modo que se puedan utilizar para diferentes ejecuciones de un mismo programa o incluso ejecuciones de diferentes programas, la mayoría de los lenguajes de programación tienen una estructura denominada *acíclico* (fichero), en el que se almacenan los datos. Un acíclico es una estructura de datos que consta de una secuencia de componentes, todos ellos del mismo tipo. Con el propósito de que los datos se guarden de modo que puedan ser utilizados de nuevo, los datos deben escribirse en un archivo, y éste se ha de almacenar en un dispositivo de almacenamiento masivo, tal como una cinta, un disquete, un disco, CD-ROM, etc.

El archivo es una estructura lineal que se compone de una serie de registros y no tiene longitud fija, dado que se almacenan externamente a la memoria del programa. La manipulación de archivos se caracteriza por su organización y tipo de acceso a los datos del archivo.

Los modos de organización de archivos son: secuenciales, relativos, indexados o secuenciales/indexados. De igual forma, las claves de acceso pueden ser por posición o por contenido, y los modos de acceso: secuencial y/o directo. En *acceso secuencial*, se puede acceder a los valores sólo en el orden en que están almacenados en el archivo. En *acceso aleatorio*, se puede acceder a los valores en cualquier orden realizado por el programador.

Se puede establecer una correspondencia entre archivos y otras estructuras de datos:

Archivo secuencial:	Lista simple
Archivo aleatorio (relativo):	Vector
Archivo indexado:	Tabla
Archivo secuencial indexado:	Tabla + lista ordenada

Las operaciones generales con archivos son:

- Creación.
- Recorrido.
- Consulta.
- Actualización.

EJERCICIOS

- 17.1. Escribir un procedimiento en el que se defina un tipo archivo de arrays de 10 elementos de tipo real. Escribir en el archivo tantos arrays como el usuario desee.
- 17.2. Dado el archivo del ejercicio 1, escribir un procedimiento que lea los datos de un archivo de arrays que se pasa como parámetro al procedimiento. Procesar hasta el final del archivo de tal forma que se escriba la media aritmética de cada array.
- 17.3. Definir un tipo archivo de palabras (cadenas). Escribir un procedimiento que tenga como parámetro un archivo de cadenas y lo cree con palabras leídas del dispositivo estándar de entrada.
- 17.4. El archivo de palabras lo tenemos ya creado. Escribir un algoritmo para que dado el archivo de palabras KRUSTER se pueda añadir nuevas palabras (a voluntad del usuario) al final del archivo.

- 17.5. Escribir el procedimiento de añadir palabras para así codificar el algoritmo del ejercicio 4.
- 17.6. Escribir las especificaciones del tipo abstracto (TAD) archivo secuencial.
- 17.7. Escribir las especificaciones del tipo abstracto de datos archivo de acceso directo.
- 17.8. Se tiene dos archivos de registros ordenados respecto un determinado campo de tipo ordinal. Escribir un algoritmo que forme un tercer archivo que resulte de mezclar de manera ordenada ambos archivos.
- 17.9. De un archivo secuencial de registros sin ningún tipo de orden se sabe que tiene registros repetidos respecto el campo Tlfno. Escribir un algoritmo para crear otro archivo que tenga los registros con claves repetidas del archivo original.

PROBLEMAS

- 17.1. Un centro académico dispone de un archivo con sus alumnos. La información de cada alumno está compuesta de Nombre, Apellido, Número de expediente, Fecha de entrada, curso actual y número de Tlf. Escribir un programa que:
 - Muestre la relación de alumnos de entrada en el centro en el mismo año A.
 - Muestre la relación de alumnos que se encuentran en el curso C.
- 17.2. Diseñar un archivo para mantener la relación de matrículas de vehículos a motor para una provincia. Escribir un programa que genere el archivo MATRICULA.
- 17.3. Diseñar un archivo para almacenar las multas de tráfico puestas a los conductores, o vehículos de una cierta provincia. Escribir un programa que genere el archivo MULTAS.
- 17.4. Se quiere enviar una notificación a todos los conductores que tienen multas no pagadas con una antigüedad mayor de tres meses. La información necesaria se encuentra en el archivo MULTAS y MATRICULA.
- 17.5. Una agencia matrimonial dispone del archivo MUJERES en el que se encuentran todas las personas de sexo femenino de la población de Guadalajara; cada registro consta de nombre completo, fecha de nacimiento y característica. De igual forma dispone del archivo HOMBRES con los mismos datos pero referidos al sexo masculino.
Escribir un programa que realice las siguientes acciones:
 - Ordene (en memoria interna) los registros de los respectivos archivos por orden de edad.
 - Dado el nombre de una señora del archivo MUJERES, muestre todas las posibles parejas que puede formar con una diferencia de edad menor de 3 años.
- 17.6. Escribir un programa que permita crear un archivo inventario de los libros de nuestra librería. Los campos asociados con cada libro son: título, autor, código, precio y unidades. Una vez que dispongamos del archivo generado, leerlo para calcular e imprimir el valor total del inventario.
- 17.7. Se dispone del archivo HABITANTES que contiene los datos correspondientes a todos los pobladores del pueblo Felisando. Cada registro tiene los campos nombre (completo), dirección, edad, fecha de nacimiento, sexo y estado. Se quiere hacer las siguientes operaciones:

- Generar el archivo NOMBRES que sólo contenga los nombres de las personas y su edad.
 - Escribir en pantalla los nombres que empiecen por las letras A, B, C, junto a su dirección, que hayan nacido en el mes de la flores (mayo).
- 17.8. Se dispone de dos archivos secuenciales, A1 y A2, con los mismos campos y ordenados crecientemente respecto una misma clave K. Escribir un programa para obtener un archivo único que contenga los registros de los dos archivos y ordenado decrecientemente respecto de K.
- 17.9. Escribir un programa que compare dos archivos con los mismos campos, F1 y F2, registro tras registro, para determinar si son iguales.
- 17.10. En el archivo de registros GRAFO, tenemos que cada registro consta de un arco del grafo G y el factor de coste del arco. Escribir un programa para representar en memoria el grafo con listas de adyacencia.
- 17.11. El archivo SOTOMA contiene por cada registro datos personales y número de socio (DNI) del club deportivo Sotomayor. El archivo BALDECOA contiene el mismo tipo de registro relativo al club excursionista del mismo nombre. Se desea crear el archivo COMUN con aquellos socios que tengan el mismo número de socio.
- 17.12. Escribir un programa con la finalidad de mezclar dos archivos de enteros cuyos elementos están en orden ascendente. Los enteros del archivo resultante deben de estar en orden ascendente.

Tratamiento de archivos de datos

CONTENIDO

- 18.1. Archivos de texto.
- 18.2. Cómo aumentar la velocidad de acceso a archivos.
- 18.3. Archivos indexados.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

Los archivos de texto son un tipo especial de archivo que consiste en un archivo de carácter dividido en líneas. Los componentes de los archivos de texto son caracteres.

En el capítulo se analizan los conceptos básicos junto con los algoritmos de manipulación de archivos de texto, así como diferentes aplicaciones para entender su estructura interna y su lectura.

Otros tipos de archivos, tratados en el capítulo, y utilizados con gran frecuencia en el procesamiento de grandes volúmenes de información son los archivos indexados que son aquellos cuya organización se basa en los valores que toman un determinado campo clave y cuya manifestación se basa en el uso de una tabla de índices.

18.1. ARCHIVOS DE TEXTO

Los archivos de texto son un caso particular de archivos de organización secuencial. Un archivo de texto es una serie continua de caracteres que se pueden leer uno tras otro.

Un archivo de texto es un archivo en el que cada registro es del tipo cadena de caracteres.

El tratamiento de archivos de texto es elemental y en el caso de lenguajes como Pascal es posible detectar lectura de caracteres especiales como *fin de linea* o *fin de archivo*.

PROBLEMA 18.1

Se dispone de un archivo de texto, de nombre Fuente.txt y se desea hacer una copia de este archivo en otro archivo cuyo nombre será SegurCopi.Txt.

ANÁLISIS

Es bastante simple copiar un archivo en otro. Una vez abiertos los dos archivos, uno para lectura y el otro para escritura, se lee carácter a carácter del archivo Fuente y se escribe carácter a carácter en el archivo Destino.

CODIFICACIÓN

```
program copia(input,output,Fuente,Destino);
uses crt;
var
  Fuente,Destino:text;
  Ch:Char;
procedure apertura;
var
  Nombre:string[25];
  Status:integer;
begin
  clrscr;
  repeat
    write('Nombre del archivo fuente');
    readln(Nombre);
    {$I-*}
    assign(Fuente,Nombre);
    reset(Fuente);
    Status := IoResult;
    if(Status<>0 then
      begin
        write('Error en la apertura del fichero
              Fuente. ','¿Quieres intentarlo de nuevo?:');
        {$I+}
        if upcase(ReadKey)<>'S'then {aborta la ejecución}
          reset(Fuente)
      end
    until Status = 0;

  repeat
    write('Nombre del archivo destino:');readln(Nombre);
    {$I-*}
    assign(Destino,Nombre);
    rewrite(Destino);
    Status := IoResult;
    if(Status<>0) then
      begin
        write('Error en la apertura del fichero
              Destino. ','¿Quieres intentarlo de nuevo?:');
        {$I+}
        if upcase(ReadKey)<>'S'then {aborta la ejecución}
      end
  until Status = 0;
```

```

        rewrite(Destino)
      end
    until Status = 0
end;
begin {bloque principal}
Apertura;
(Lectura del archivo carácter a carácter)
while not eof(Fuente) do
begin
  while not eoln(Fuente) do
  begin
    read(Fuente,Ch);
    write(Destino,Ch);
  end;
  readln(Fuente);
  writeln(Destino)
end;
close(Fuente);
close(Destino)
end.

```

PROBLEMA 18.2

En un archivo de texto, se tiene nombres de clientes y sus correspondientes direcciones. En otro archivo de texto Carta .Txt, se tiene escrita una carta de presentación de un producto. Se desea escribir un programa para enviar cartas personalizadas, según las personas del archivo Dirección .Txt.

ANÁLISIS

El problema se resuelve tomando como archivo maestro o guía el que contiene las direcciones de los clientes. Por cada dirección de un cliente hay que leer el archivo carta para así formar una carta con su encabezado que es su dirección. Todo termina cuando se acabe el archivo de direcciones.

```
program Cartas;
```

{Este programa genera cartas personalizadas a partir de un archivo modelo de carta y de un archivo que contiene la relación de todos los destinatarios de la misma.

El archivo de carta (modelo) representa con el carácter ASCII 200, el nombre del destinatario, y con el carácter 201 la dirección del mismo. Se generan tantos archivos de cartas como registros tiene el archivo de destinatarios. El archivo modelo se supone previamente creado. El programa genera el archivo de destinatarios.}

```

uses
  Dos, Crt;
type
  cadena08 = string[8];
  cadena30 = string[30];
  Item = record
    ApeNom,
    direccion:cadena30

```

```
end;
personas = file of Item;
var
  ModeloCarta, (*contiene el modelo de carta*)
  Personal:text; (*fichero de carta personalizada*)
  Destinatarios:personas;(*fichero de destinatarios*)
  persona:Item; (*registro de destinatario*)
  CartaDos:Cadena30;(*nombre fichero carta personal*)
  error:byte;
  letra:char;

procedure Pausa;
const
  Caracteres=[#0..#255];
var
  letra:char;
begin
  GotoXY(2,23); ClrEol;
  Write('Pulse una tecla para continuar');
  repeat
    letra := ReadKey
  until letra in Caracteres;
end;

procedure Mostrar(var carta:text);
{Muestra en pantalla la carta modelo y la personalizada}
var
  letra:char;
begin
  ClrScr;
  reset(carta);
  while not eof(carta) do
    begin
      while not eoln(carta) do
        begin
          read(carta,letra); Write(letra)
        end;
        readln(carta); WriteLn
      end;
      close(carta)
    end;

procedure Crear(var f:personas);
{Genera el fichero de destinatarios}
var
  registro:Item;
  I:byte;
  Masdatos:char;
begin
  rewrite(f);
  repeat
    ClrScr;
    WriteLn('Introduzca datos de destinatario':55);
    with registro do
      begin
        Write('introduzca nombre:');readln(ApeNom);
        ApeNom:=uppercase(ApeNom);
        Apellido:=copy(ApeNom,1,3);
        Nombre:=copy(ApeNom,4,8);
        Sexo:=copy(ApeNom,13,1);
        if Sexo='M' then Sexo:=1 else Sexo:=0;
        DNI:=copy(ApeNom,14,8);
        Fecha:=copy(ApeNom,23,8);
        Telefono:=copy(ApeNom,32,10);
        Direccion:=copy(ApeNom,42,20);
        Localidad:=copy(ApeNom,62,15);
        Provincia:=copy(ApeNom,77,2);
        Postal:=copy(ApeNom,80,5);
        Sexo:=chr(Sexo+48);
        Apellido:=chr(Apellido+48);
        Nombre:=chr(Nombre+48);
        Telefono:=chr(Telefono+48);
        Postal:=chr(Postal+48);
        Fecha:=chr(Fecha+48);
        Direccion:=chr(Direccion+48);
        Localidad:=chr(Localidad+48);
        Provincia:=chr(Provincia+48);
      end;
      write(f);
    end;
  until I=0;
  close(f);
end;
```

```

for I:=Length(ApeNom)+1 to 30 do
  ApeNom := ApeNom+'';
  Write(' introduzca direccion:');
  readln(Direccion);
  for I:=Length(Direccion)+1 to 30 do
    Direccion := Direccion+''
  end;
  Write(f,registro);
  GotoXY(2,23); ClrEol;
  Write('otro destinatario?');readln(Masdatos)
until Uppcase(Masdatos)<>'S';
close(f)
end;

procedure Escribir(var f:personas);
{Muestra el fichero de destinatarios}
var
  registro:Item;
  I:byte;
  Masdatos:char;
begin
  reset(f);
  while not eof(f) do
  begin
    read(f,registro);
    with registro do
      WriteLn(ApeNom,'',Direccion)
  end;
  close(f)
end;

begin
  ClrScr;
  assign(Modelocarta,'c:\datos\carta.txt');
  mostrar(ModeloCarta);
  Pausa;
  assign(Destinatarios,'c:\datos\destinos.dat');
  {$I-}
  reset(ModeloCarta);
  error := IoResult;
  {$I+}
  if error<>0 then
    WriteLn('Fichero modelo de carta erroneo')
  else begin
    Crear(Destinatarios);
    WriteLn('Fichero de destinatarios':52);
    Escribir(destinatarios);
    Pausa; ClrScr;
    reset(Destinatarios);
    while not eof(Destinatarios) do
    begin
      ClrScr;
      read(Destinatarios,persona);
      WriteLn(Persona.ApeNom);
      Write('fichero DOS para',Persona.ApeNom,':');
      readln(CartaDos);
    end;
  end;
end.

```

```

assign(Personal,CartaDos);
rewrite(Personal);
reset(modelocarta);

while not eof(modelocarta) do
begin
  while not eoln(modelocarta) do
  begin
    read(modelocarta,letra);
    case letra of
      #200:Write(Personal,Persona.Apenom);
      01:Write(Personal,Persona.Direccion)
    else
      Write(Personal,letra)
    end
  end;
  writeln(Personal);
  readln(modelocarta)
end;
close(modelocarta);close(Personal);
Mostrar(Personal); Pausa
end;
close(Destinatarios)
end
end.

```

18.2. CÓMO AUMENTAR LA VELOCIDAD DE ACCESO A ARCHIVOS

La desventaja que presentan los archivos secuenciales frente a los archivos aleatorios es que las distintas operaciones se ralentizan. Cada operación requiere la lectura del archivo completo, además algunos registros pueden ser modificados y por tanto necesitar ser escritos.

Existen organizaciones de archivos que permiten acceder a un registro, leyendo en la memoria interna una pequeña fracción del archivo completo. En estas organizaciones cada uno de los registros ha de tener una clave que identifica de manera única al registro. Por ejemplo, el campo número de matrícula del archivo de alumnos puede considerarse una clave. Puede suponerse que no existen simultáneamente dos registros en el archivo con el mismo número de matrícula.

Para lograr mayor rapidez en las operaciones con archivos secuenciales ciertos lenguajes de programación permiten acceder directamente a bloques, en vez de recorrer en secuencia los bloques de que consta el archivo. Para ello se utilizan apuntadores a los propios bloques que son direcciones físicas de los mismos.

18.2.1. Archivos con función de direccionamiento hash

El tiempo de búsqueda de los algoritmos de manejo de ficheros tratados anteriormente depende del número n de registros. La técnica del direccionamiento *hash* o dispersión es muy utilizada para tener un acceso rápido a información almacenada en archivos.

El objetivo que persigue esta técnica de direccionamiento la mostramos en este ejemplo.

Supongamos un colegio de primaria con 240 alumnos. Cada alumno tiene asignado un número de identificación de 6 dígitos, que será usado como campo clave en el archivo de alumnos del colegio. Se podría pensar en una organización tal que usase el número de identificación como dirección del registro en memoria. Con esta organización, la búsqueda no requerirá ninguna comparación, directamente se accede al registro. Por desgracia, esta organización requerirá un espacio de 1.000.000 posiciones de memoria, mientras que el espacio realmente necesario son 240 posiciones. Es un gasto de memoria desproporcionado, a costa de tiempo, que no merece la pena.

Para el desarrollo de esta técnica se supone que se dispone de un archivo F de n registros. Cada registro tiene un campo clave k que determinará biunívocamente los registros de F . También se supone que existe una tabla en memoria de m posiciones y que l es el conjunto de direcciones de las posiciones de la tabla. La claves k pueden ser enteros, cadenas de caracteres...; por facilidad en la notación se supone que tanto las claves k como las direcciones l son enteros.

La idea básica es utilizar la clave para determinar la dirección del registro, pero para no desperdiciar tanto espacio, hay que realizar una transformación mediante una función hash del conjunto k de claves sobre el conjunto l de direcciones de memoria.

$$h(x): k \rightarrow l$$

Esta es la función de direccionamiento *hash* o función de dispersión.

Hay que contemplar el hecho de que la función $h(x)$ no de valores distintos: es posible (según la función elegida) que dos claves diferentes k_1 y k_2 den la misma dirección. 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.

Resolución de colisiones.

18.2.2. Funciones hash

Existe un número considerable de funciones *hash*. Dos criterios nos deben guiar al seleccionar una función. En primer lugar, la función $h(x)$ debe de calcularse fácilmente, dependerá de la clave k . En segundo lugar, la función $h(x)$ debe de distribuir uniformemente las direcciones sobre el conjunto l de forma que se minimice el número de colisiones. Nunca existirá una garantía plena de que no haya colisiones, y más sin conocer de antemano las claves y las direcciones. La experiencia nos enseña de que siempre habrá que estar preparado para cuando se produzca alguna colisión.

Algunas de las funciones *hash* de cálculo más fácil y rápido las exponemos a continuación.

Aritmética modular

Se elige m un número primo, o con pocos divisores, mayor que el número n de registros. La función *hash* se define:

$h(k) = k \bmod m$ para que las direcciones vayan de 0 a $m-1$,
o bien

$$h(k) = (k \bmod m) + 1, \text{ así las direcciones irán de } 1 \text{ a } m.$$

En esta fórmula m ha de ser primo para minimizar el número de colisiones.

EJEMPLO 18.1

En el archivo de alumnos se supone que el número de direcciones es 996. La elección de m en este caso será 997, que es el número primo más próximo.

Se aplica esta función hash a los alumnos cuyo número es:

245643 245981 257135

y se obtienen estas direcciones

$$\begin{aligned} h(245643) &= 245643 \bmod 997 = 381 \\ h(245981) &= 245981 \bmod 997 = 719 \\ h(257135) &= 257135 \bmod 997 = 906 \end{aligned}$$

Plegamiento

La técnica del plegamiento consiste en partir la clave k en varias partes $k_1, k_2, k_3 \dots k_n$, y la combinación de las partes de un modo conveniente (con frecuencia sumando las partes) da como resultado la dirección del registro.

Cada parte k_i , con a lo sumo la excepción de la última, tiene el mismo número de dígitos que la dirección especificada.

La función hash se define

$$h(k) = k_1 + k_2 + \dots + k_r$$

En esta operación se desprecian los dígitos más significativos que se obtengan de acarreo.

EJEMPLO 18.2

En el archivo de alumnos con el campo clave de 6 dígitos. Un entero de 6 dígitos se puede dividir en grupos de tres y tres dígitos.

Aplicamos esta función hash a los alumnos cuyo número:

245643 245981 257135

se obtienen estas direcciones

$$\begin{aligned} h(245643) &= 245+643 = 888 \\ h(245981) &= 245+981 = 1226 = 226 \text{ (se ignora el acarreo 1)} \\ h(257135) &= 257+135 = 392 \end{aligned}$$

A veces se hace la inversa a las partes pares, $k_2, k_4 \dots$ antes de sumarlas, con el fin de afinar más. Así obtendríamos estas direcciones:

$$\begin{aligned} h(245643) &= 245+346 = 591 \\ h(245981) &= 245+189 = 434 \\ h(257135) &= 257+531 = 788 \end{aligned}$$

Mitad del cuadrado

Este método consiste en calcular el cuadrado de la clave k y la dirección del registro viene representada por los dígitos de k^2 que ocupan cierta posición.

La función se define

$$h(k) = c$$

Siendo C el número formado por los dígitos de k^2 que se encuentran en las posiciones P_1, P_2, \dots, P_r . Es importante utilizar siempre las mismas posiciones de k^2 para todas las claves.

EJEMPLO 18.3

En el archivo de alumnos con el campo clave de 6 dígitos.

Aplicamos esta función *hash* a los alumnos cuyo número:

245643 245981 257135

se obtienen estas direcciones

245643 → 60340483449 escogiendo los dígitos cuarto, quinto y sexto por la derecha
 $h(245643) = 483$

245981 → 60506652361
 $h(245981) = 652$

257135 → 66118408225
 $h(257135) = 408$

18.2.3. Resolución de colisiones

La función *hash* $h(k)$ no siempre proporciona direcciones distintas; puede ocurrir que para dos claves diferentes k_1, k_2 $h(k_1)=h(k_2)$, se obtenga la misma dirección. Este hecho es conocido como colisión. Es evidente que se deben proporcionar métodos de resolución de colisiones.

Considérese el ejemplo de un taller de 18 empleados, con el campo clave del DNI y 100 posibles direcciones. Con la función *hash* del módulo y tomando como m el primo 101, las claves

123445678 123445880

proporcionan las direcciones

$$\begin{aligned} h(123445678) &= 123445678 \bmod 101 = 44 \\ h(123445880) &= 123445880 \bmod 101 = 44 \end{aligned}$$

Se tienen dos claves a las que aplicada la función *hash* se transforman en la misma dirección, se dice que las claves han colisionado.

A continuación se muestran dos formas de resolver las colisiones, direccionamiento abierto y encadenamiento; el procedimiento que se elija dependerá fundamentalmente de la relación entre el número *n* de registros (también, número de claves) y el número de direcciones posibles *m*. Esta relación se expresa $\lambda = n/m$, llamado factor de carga.

La eficiencia de una función *hash* junto con un método de resolución de colisiones se mide por el número medio de comparaciones de claves necesarias para encontrar la dirección que ocupa un registro con clave *k*. La eficiencia depende principalmente del factor de carga λ .

Las siguientes dos cantidades se discuten con cada método de resolución de colisiones.

$S(\lambda)$ = número medio de comparaciones para una búsqueda con éxito.

$U(\lambda)$ = número medio de comparaciones para una búsqueda sin éxito.

18.2.3.1. Direccionamiento abierto

Es la forma más primaria de resolver una colisión entre claves. Supóngase que se va a añadir el registro *R* de clave *k*, pero la posición que nos da la función $h(k) = p$ está ya ocupada por otro registro. La forma de resolver esta colisión es buscar la primera posición disponible que siga a *p* y asignar a esa posición *R*. Hay que asumir que si tenemos *m* posiciones disponibles, la posición que sigue a *m* es la posición 1, a la manera de array circular.

EJEMPLO 18.4

Supongamos que se tienen los registros de claves, K1, K2, K3, K4, K5, K6, K7, K8 y K9. Para cada uno la función hash:

Registro: K1 K2 K3 K4 K5 K6 K7 K8 K9

$h(k) :$ 5 8 11 9 5 7 8 6 14

Entonces los registros en el archivo aparecerán grabados en las siguientes posiciones:

Registro: K1 K2 K3 K4 K5 K6 K7 K8 K9

Posición: 5 8 11 9 6 7 10 12 14

El registro K8 es el único que tiene la dirección *hash* 6, sin embargo, esa posición ya está ocupada por K5 debido a una colisión previa, la siguiente posición libre es la 12.

El número medio de comparaciones para una búsqueda con éxito *S*:

$$S = \frac{1+1+1+1+2+1+3+7+1}{9} = 2.00$$

El número medio de comparaciones para una búsqueda sin éxito, suponiendo que hay 15 posiciones disponibles de almacenamiento:

$$U = \frac{1+1+1+1+9+8+7+6+5+4+3+2+1+2+1}{15} = 3.47$$

La suma del segundo parámetro U acumula las comparaciones que deberían realizarse para encontrar una posición vacía para cada una de las 15 posiciones. Así, si fuera necesario añadir el registro R de clave k , siendo $h(k) = 10$. Como la posición 10 está ya ocupada, y lo mismo ocurre con la 11, 12 y 13 es la primera libre, se han realizado cuatro comparaciones.

Esta forma de resolver colisiones se denomina *prueba lineal*. Los valores medios de S y U pueden expresarse en función del factor de carga λ :

$$S(\lambda) = \frac{1}{2} \left(1 + \frac{1}{1-\lambda} \right) \quad \text{y } U(\lambda) = \frac{1}{2} \left(1 + \frac{1}{(1-\lambda)^2} \right)$$

El direccionamiento abierto no es recomendable cuando el archivo vaya a experimentar muchos borrados de registros. Y esto es debido a que si suponemos que en la posición p está el registro R y se elimina, en una posterior búsqueda de una posición libre, la posición p figura como que está ocupada. Entonces, habrá que indicar mediante un campo que en la posición p se encuentra un registro que a nivel lógico está eliminado y, por tanto, p está disponible para almacenar otro registro.

El método de direccionamiento abierto o prueba lineal tiene como principal inconveniente el que agrupa los registros en posiciones contiguas cuando el factor de carga supera el 50 por 100. El situar los registros en posiciones contiguas aumenta el tiempo medio de búsqueda para un registro. Se emplean dos técnicas para aminorar el problema del agrupamiento:

Prueba cuadrática

Suponiendo que a un registro con clave k le corresponde la dirección p ; el método de prueba lineal busca en las direcciones $p, p+1, p+2, \dots$. El método de prueba cuadrática busca en las direcciones $p, p+1, p+4, p+9, \dots p+i^2, \dots$

Doble direccionamiento hash

Este método utiliza una segunda función *hash* para resolver una colisión. Supongamos que al registro de clave k le corresponde la dirección $h(k) = p$, pero resulta que está ocupada por otro registro. Entonces con la segunda función $h'(k) = p'$, la nueva dirección es $p + p'$; si también hubiera colisión, la siguiente dirección sería $p + 2p'$, y así sucesivamente.

Resumiendo, la secuencia posible de direcciones:

$p, p+p', p+2p', p+3p' \dots$

18.2.3.2. Direccionamiento por encadenamiento

Este método para resolver colisiones de claves con la misma dirección *hash* se basa en la formación de una lista enlazada con todos los registros que tienen la misma dirección.

Sea R un registro con la clave k , $h(k)$ es el número de lista enlazada que contiene el registro si R ya está en el archivo. Los registros tendrán un campo adicional *Sgt e* que se usará para mantener el encadenamiento de los registros con la misma dirección *hash*.

En la figura se puede ver la estructura de datos básica para este método. La idea fundamental es que si se ha elegido una función hash con un rango de m valores, 0 a $m - 1$, se tiene una tabla de m elementos indexada de 0 a $m - 1$ que contendrá los punteros de m listas enlazadas.

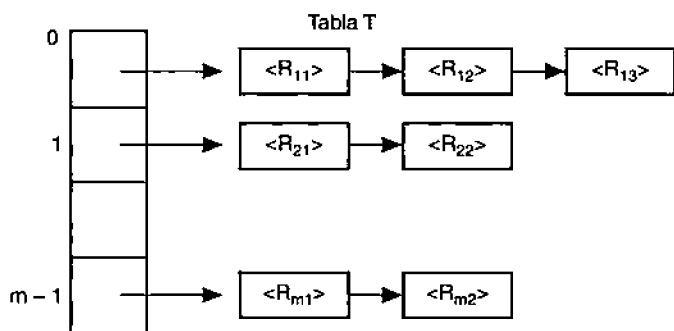


Figura 18.1. Direccionamiento por encadenamiento.

Para añadir un registro con clave k , este se almacena en primera posición libre del conjunto posiciones de memoria reservadas para el archivo. A continuación insertamos el registro en la lista de índice $h(k)$. Puede ocurrir que la lista esté vacía, por no haber habido colisión, o bien que al producirse colisión haya que insertar en la lista. La inserción en la lista puede hacerse como primer elemento, último o una inserción ordenada. El campo *Sgt e* de cada registro hay que actualizarlo según el tipo de inserción.

EJEMPLO 18.5

Supongamos que se tienen los registros del ejemplo anterior:

Registro: K1 K2 K3 K4 K5 K6 K7 K8 K9
 $h(k) :$ 5 8 11 9 5 7 8 6 14

Tabla T
 Disposición en archivo

Registro 1 : K1
Registro 2 : K2
Registro 3 : K3
Registro 9 : K9
DSP = 10

El rango de $h(x)$ es 0 .. 14, la tabla T de listas enlazadas estará indexada de 0 a 14. En la Figura 18.1 aparece la disposición de los registros en memoria. Se puede observar que con este método la posición que ocupa un registro R no está directamente relacionado con $h(x)$. Un registro R se sitúa en la primera posición libre del fichero (la variable DSP tiene la primera posición libre).

La búsqueda de un registro R no es más que la búsqueda de un nodo en una lista enlazada. Con la función *hash* $h(x)$ obtenemos el índice de lista que corresponde, después se busca en la lista. Lo mismo ocurre con la eliminación de un registro, primero se busca en la lista enlazada que le corresponde y a continuación se elimina de la lista.

El número medio de comparaciones para una búsqueda con éxito y para una búsqueda sin éxito:

$$S(\lambda) \approx 1 + \frac{1}{2} \lambda \approx U(\lambda); e^{-\lambda} + \lambda$$

Con el método de encadenamiento el factor de carga $\lambda = n/m$ ha de ser mayor que 1.

PROBLEMA 18.3

Resolución de un archivo de atletas

La finalidad de este ejercicio es realizar las operaciones básicas con un archivo en el que los registros están en direcciones dadas por una función hash. Se incluye la resolución de posibles colisiones.

El enunciado del problema es el siguiente:

Se desea almacenar en un archivo los atletas participantes en un cross popular. El máximo de participantes se tiene establecido en 250. Los datos de cada atleta: Nombre, Edad, Sexo, Fecha de nacimiento y Categoría (Junior, Promesa, Senior, Veterano).

ANÁLISIS

Supóngase que el conjunto de direcciones del que se dispone es de 400. Estas direcciones están en el rango de 0 a 399. Se elige como campo clave el Apellido del atleta, una cadena de 25 caracteres como máximo.

La función *hash* utilizada es la de aritmética modular, para lo cual nos es necesario transformar en valor numérico a la cadena. Para ello, CalCula simplemente suma el ordinal ASCII de los caracteres alfabéticos.

Para resolver colisiones el método que se desarrolla es el de prueba lineal (direcccionamiento abierto).

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. Se limita a estas operaciones ya que se pretende un sencillo ejemplo, el lector puede añadir otras operaciones como son listar los tres primeros clasificados, listar los participantes según orden de llegada... La unidad *hash1* tiene los tipos de datos necesarios para resolver el problema, las operaciones con el archivo y la realización de función *hash*.

```

unit Hash1;
interface
  const
    M= 399;
  type
    Rango= 0..M;
    Tsexo= (Hombre,Mujer);
    TCategoría=(Junior, Promesa, Senior, Veterano);
    Tnom= String[25];
    Tfecha= record
      D: 1..31;
      M: 1..12;
      A: 0..99
    end;
    Estado= (Libre,Ocupado);
    Atleta= record
      St: Estado;
      Apellido: Tnom;
      Nombre: Tnom;
      Edad: 16..99;
      Sexo: Tsexo;
      Cat: TCategoría;
      FechaN: Tfecha
    end;
    FichAtl= file of Atleta;

    function Hash(C: Tnom):Rango;
    function PruebaLineal(var F:FichAtl; p:integer):Rango;
    function Status(E:Integer): Estado;
    procedure Crear(var F:FichAtl;var S:boolean);
    procedure Insertar(var F:FichAtl; R:Atleta);
    procedure Borrar(var F: Fichat1; R:Atleta);
    procedure BuscNom(var F:Fichat1; C:Tnom;
                      var Esta:boolean; var R:Atleta);
    procedure BuscaRg(var F:Fichat1;R:Atleta
                      var Esta:boolean;var H:Rango);
implementation
  function Status(E:Integer): Estado;
  begin
    if E=1 then
      Status:= Ocupado
    else
      Status:= Libre
  end;
  function Hash(C: Tnom):Rango;
  var
    I, Suma: integer;
  begin
    Suma:= 0;
    for I:= 1 to length(C) do
      Suma:= Suma+ord(C[I]) ;
    Hash:= Suma mod M
  end;

```

```

function PruebaLineal(var F:FichAtl; p:Rango):Rango;
{Método de prueba lineal para resolver colisiones, devuelve la
 primera dirección libre a partir de la dirección hash p.}
var
  R: Atleta;
begin
  filepos(F,p);{posiciona el pointer del fichero}
  read(F,R);
  while Status(R.St)=Ocupado do {registro ocupado}
  begin
    p:= (p+1) mod (M+1);{Siguiente dirección, considerando una
    secuencia circular}
    read(F,R)
  end;
  Pruebalineal:= p
end;

procedure Crear(var F:FichAtl;var S:boolean);
{Reserva espacio de memoria de disco para M+1 registros}
var
  N: string;
  R: Atleta;
  J: integer;
  procedure Inic(var R:Atleta);
  begin
    with T do
    begin
      St:= Libre;
      Apellido:=' ';
      Nombre:=' ';
      Edad:=0;
      Sexo:=Hombre;
      Cat:=Junior;
      Fechan.D:=1;
      Fechan.M:=1;
      Fechan.A:=0;
    end
  end;
begin
  write('Camino/Nombre del archivo: ');
  readln(N);
  assign(N,F);
  {$I-}
  rewrite(F);
  if Ioresult<> 0 then
    begin
      writeln('Error al crear el archivo');
      S:= false
    end
  else
    begin
      Inic(R);
      for J:=0 to M do
        write(F,R);
      S:= true
    end
end;

```

```

    end
    ($I+)
end;

procedure Insertar(var F:Fichatl; R:Atleta);
var
  H, Nr: integer;
begin
  H:= Hash(R.Apellido);
  Nr:= Pruebalineal(F,H);
  { El registro ocupará la posición Nr }
  filepos(F,Nr);
  R.St:= Ocupado;
  write(F,R)
end;

procedure Borrar(var F: Fichatl; R:Atleta);
var
  Encontrado:boolean;
  H: Rango;
begin
  {Inspecciona si está el registro}
  BuscaRg(F,R,Encontrado,H);
  if Encontrado then
  begin
    filepos(F,H);
    R.St:= libre;
    write(F,R)
  end
  else
    writeln('Registro correspondiente a ',
           R.Apell, 'No está en el archivo.')
end;

procedure BuscNom(var F:Fichatl; C:Tnom;
                  var Esta:boolean; var R:Atleta);
{A partir de la dirección hash correspondiente a C, busca el apellido C}
var
  H,P: Rango;
  R: Atleta;
begin
  H:= Hash(C);
  P:= H;
  repeat
    filepos(F,P);{posiciona el pointer del fichero}
    read(F,R);
    Esta:=(Status(R.St)=Ocupado) and (R.Apellido=C);
    p:= (p+1) mod (M+1) {Siguiente dirección}
  until Esta or (P=H)
end;

procedure BuscaRg(var F:Fichatl;R:Atleta
                  var Esta:boolean;var H:Rango);
{Busca la dirección en la que se encuentra R}
var
  P: Rango;
  W: Atleta;

```

```

begin
  H:= Hash(R.Apellido);
  P:= H;
  repeat
    filepos(F,P);(posiciona el pointer del fichero)
    read(F,W);
    Esta:=(Status(W.St)=Ocupado) and
      (W.Apellido=R.Apellido) and
      (W.Nombre=R.Nombre) and
      (W.Edad=R.Edad) and
      (W.Sexo=R.Sexo);
    p:= (p+1) mod (M+1) {Siguiente dirección}
  until Esta or (P=H);
  if Esta then
    H:= P-1
  end;
end.

```

En este sencillo programa se pone en funcionamiento la unidad Hash.

```

program Atletas(Fatletas);
uses
  crt,Hash1;
const
  L= 4;
type
  Opc= 0..L;
var
  Op:Opc;
  F:FichAtl;
  R:Atleta;
  Flag:boolean;

procedure Menu(var Q:Opc);
begin
  writeln (' 1. Crear '); writeln;
  writeln (' 2. Añadir '); writeln;
  writeln (' 3. Borrar '); writeln;
  writeln (' 4. Buscar '); writeln;
  writeln (' 0. Salir '); writeln;
  repeat
    write('Opción elegida?: ');
    readln(Q)
  until Q in [0..L];
end;

procedure Abrir(var F:Fichatl, var Oper:boolean);
var
  N: string;
begin
  if not Oper then
  begin
    write('Camino/Nombre del archivo:');readln(N);
    assign(N,F);
    {$I-}
  end;
end;

```

```

Oper:= true;
reset(F);
if Ioresult<> 0 then
begin
  writeln('Error archivo; no puede ser leido.');
  Oper:= false
end;
{$I+}
end
end;

procedure Eatleta(var E:Atleta);
var
  Ap1l: Tnom;
  C: char;
begin
  write('Apellido: '); readln(Ap1l);
  if Ap1l<>'' then
  begin
    E.Apellido:= Ap1l;
    write('Nombre: '); readln(E.Nombre);
    write('Edad: '); readln(E.Edad);
    write('¿Junior,Promesa,Senior,Veterano
          (J/P/S/V)?:');
    repeat
      readln(C)
    until C in ['J','j','P','p','S','s','V','v'];
    case C of
      'J','j': E.Categoría:= Junior;
      'P','p': E.Categoría:= Promesa;
      'S','s': E.Categoría:= Senior;
      'V','v': E.Categoría:= Veterano;
    end;
    write('¿Mujer,Hombre(M/H)?: ');
    repeat
      readln(C)
    until C in ['M','m','H','h'];
    if C in ['M','m'] then
      E.Sexo:= Mujer
    else
      E.Sexo:= Hombre
    end
  end;
end;

procedure Annadir(var F:FichAtl;var Oper:boolean);
var
  R: Atleta;
begin
  if not Oper then
    Abrir(F,Oper);
  if Oper then
  begin
    writeln('Termina la entrada pulsando ENTER' );
    repeat
      Eatleta(R);
      if R.Apellido<>'' then

```

```
    Insertar(F,R)
    until E.Apellido= ''
  end
end;

procedure Eliminar(var F:FichAtl;var Oper:boolean);
var
  R: Atleta;
  A: Tnom;
  S: boolean;
  {Pide el apellido del atleta que se desea borrar del archivo}
begin
  if not Oper then
    Abrir(F,Oper);
  if Oper then
  begin
    write('Apellido del atleta: ');
    readln(A);
    BuscNom(F,C,S,R);
    if S then
      Borrar(F,R)
    end
  end;
end;

procedure Mostrar(R:Atleta);
begin
  with R do
  begin
    writeln(Apellido, ' ',Nombre);
    case Categoría of
      Junior : write('Junior ');
      Promesa : write('Promesa ');
      Senior : write('Senior ');
      Veterano: write('Veterano ');
    end;
    case Sexo of
      Mujer : write('Femenino, ');
      Hombre: write('Masculino, ')
    end;
    writeln(Edad,' años')
  end
end;

procedure Presentar(var F:FichAtl);
var
  R: Atleta;
  A: Tnom;
  Esta: boolean;
  {Pide el apellido del atleta que se desea mostrar}
begin
  if not Oper then
    Abrir(F,Oper);
  if Oper then
  begin
    write('Apellido del atleta: ');
    readln(A);
```

```

BuscNom(F,C,Esta,R);
  if Esta then
    Mostrar(R)
  else
    Writeln('No hay atleta con ese apellido.')
  end
end;

begin
  ClrsCr;
  Flag:= false;
  repeat
    Menu(Op);
    case OP of
      1: Crear(F,Flag);
      2: Annadir(F,Flag);
      3: Eliminar(F,Flag);
      4: Presentar(F,Flag)
    end
  until (Op=0) or Flag
end.

```

18.2.4. Realización con encadenamiento

Para mejorar la efectividad en las operaciones con registros cuando se prevé muchas colisiones se utiliza el direccionamiento cerrado, formando listas enlazadas con los registros con la misma dirección *hash* (colisiones). Un registro *R* se coloca en la primera dirección disponible del archivo y además se inserta en la lista enlazada indexada por $h(k)$, siendo k la clave de *R*.

Por un lado se tiene que manejar el archivo con los registros, y por otro lado el vector de listas. Los índices de este vector se corresponden con el rango de la función *hash*. Los registros a los que le corresponde la misma $h(k)$ forman parte de la lista $\mathcal{V}[h(k)]$. De esta forma la operación de búsqueda o de borrado se convierte en una operación en una lista enlazada.

Ahora bien, se plantea el problema de qué hacer con el vector de listas. No puede permanecer en memoria principal indefinidamente, cuando se quiera dejar de procesar la información del archivo hay que almacenar también el vector, para lo cual se dispone de otro archivo, ENLACES. La disposición del vector en ENLACES ha de ser tal que nos facilite un posterior proceso del archivo.

Al volver a procesar el archivo de registros, la primera acción es reconstruir el vector de listas partiendo del archivo ENLACES.

Los datos se disponen secuencialmente y todos se consideran de tipo entero. El primer registro es el número de direcciones *hash*, *M*. En el segundo registro se graba la siguiente posición libre del archivo de datos, para así empezar a grabar a partir de ésta. El resto responden a esta secuencia: índice, número de registros de la lista y posición de cada registro en el archivo. Repitiéndose esta secuencia por cada índice del vector, que es lo mismo que decir por cada valor de $h(k)$.

EJEMPLO 18.5

En el ejemplo ya expuesto anteriormente:

Registro: K1 K2 K3 K4 K5 K6 K7 K8 K9

h(k) : 5 8 11 9 5 7 8 6 14

El archivo ENLACES tiene los siguientes datos:

14	9	0	0	1	0	2	0	3	0	4	0	5	2	4	0	6	1	7	7	1	5
8	2	6	1	9	1	3	10	0	11	0	12	0	13	0	14	1	8				

De este contenido de ENLACES podemos decir que los registros ocupan las posiciones de 0 a 14, hay 9 registros grabados (direcciones de 0 a 8), y que el siguiente estará en la dirección 9. Obsérvese que registros con $h(k) = 5$ hay dos que ocupan las posiciones 4 y 0 respectivamente (5 2 4 0); y sucesivamente se explica la secuencia de datos del archivo ENLACES. También que las inserciones en la lista siempre se hacen por el comienzo de la lista.

Se plantea el siguiente supuesto para plasmar estas ideas.

PROBLEMA 18.4

Una tienda es capaz de ofrecer un máximo de 100 artículos diferentes. El registro que identifica a cada artículo tiene los campos: unidades, precio, identificación y fecha de caducidad. El programa debe grabar registros, borrar y buscar.

ANÁLISIS

Para mejorar el tiempo de acceso a los registros se construye un archivo aplicando las técnicas hash; en particular, direccionamiento por encadenamiento para resolver colisiones. El campo clave es el identificador del artículo, se supone de 5 dígitos. La función hash que se aplica responde a la técnica de «mitad del cuadrado»: una vez elevado al cuadrado el campo clave, la dirección hash se corresponde con los dos dígitos centrales.

El archivo ENLACES tiene todos sus campos de tipo entero y representa al vector de listas. Siempre que el proceso del archivo termina se vuelve a escribir las listas. Y siempre que el proceso se inicie hay que leer ENLACES para reconstruir la estructura multienlazada. La unidad LISTA tiene los tipos de datos y operaciones para manejo de listas. En la unidad HASH2 escribimos los tipos, la función hash y las operaciones de los archivos.

Interfaz de LISTA

```
unit LISTA;
interface
type
  PtrLst= ^NodoL;
  NodoL= record
```

```

Nreg: integer;
Sgte: PtrLst
end;
procedure Listavacia(var L:PtrLst);
function Esvacia(L:PtrLst):boolean;
procedure Inserprim(X:integer;var L:PtrLst);
function Localiza(X:integer;L:PtrLst):PtrLst;
procedure Suprime(X:integer;var L:PtrLst);
function Anterior(P:PtrLst;L:PtrLst):PtrLst;
function Cuantos(L:PtrLst):integer;
...
...

```

Estas son las operaciones que son utilizadas en la unidad *hash2* para tratar las listas enlazadas de registros con la misma función *hash*.

Unidad para resolución de colisiones mediante listas

```

unit Hash2;
interface
uses PtrLst;
const
  M= 99;
type
  Rango= 0..M;
  Vector= array[Rango] of PtrLst;
  Tfecha= record
    D: 1..31;
    M: 1..12;
    A: 0..99
  end;
  Articulo= record
    Unidades: integer;
    Precio: real;
    Idt:string[5];
    Caducidad: Tfecha
  end;
  Ftienda= file of Articulo;
  Enlaces= file of integer;
function Hash(Id:String[5]):Rango;
procedure Crear(var F:Ftienda;var E:Enlaces;
               var V:Vector;var S:boolean);
procedure Insertar(var F:Ftiendas;var V:Vector;
                  var Sg:integer; R:Articulo);
procedure Borrar(var F:Ftiendas;var V:Vector;R:Articulo);
procedure BuscIdt(var F:Ftiendas; C:String[5];
                  V: Vector;var Esta:boolean; var R:Articulo);
procedure BuscaRg(var F:Ftiendas;R:Articulo;
                  V:vector;var Esta:boolean;var P:integer);
procedure InicProces(var F:Ftiendas;var E:Enlaces; var V:Vector;
                     var Sg:integer;var S:boolean);
procedure FinProces(var E:Enlaces;V:Vector;Sg:integer);
implementation

```

```

function Hash(Id:String[5]):Rango;
{Método de mitad de cuadrado: dos dígitos centrales del cuadrado del
campo clave}
const
  Digs= 2; {dígitos centrales}
  Pot2= 100;
var
  Cl: longint; { Entero largo }
  I, K: integer;
function NumDgs(X:longint):integer;
var
  D: integer;
begin
  D:=0;
  while X>0 do
  begin
    X:= X div 10;
    D:= D+1
  end;
  NumDgs:= D
end;
begin
  Cl:=0; { Clave en binario }
  for I:=1 to length(Id) do
    Cl:= Cl*10+(ord(Id[I])-ord('0'));
  Cl:= Cl*Cl;
  Pos:=(Numdgs(Cl)+1) div 2;{Posición central del cuadrado}
  for I:=1 to Pos-(Digs-1) do {Son eliminados dígitos de menor peso}
    Cl:=Cl div 10;
  Hash:= Cl mod Pot2;
end;

procedure Crear(var F:Ftienda;var E:Enlaces;
               var V:Vector;var S:boolean);
  {Son abiertos los archivos e inicializado el vector de listas}
var
  I: integer;
  N: string;
begin
  write('Camino/Nombre del archivo de datos: ');
  readln(N);
  assign(N,F);
  {$I-}
  rewrite(F);
  if Iorestart<> 0 then
  begin
    writeln('Error al crear el archivo de datos');
    S:= false
  end
  else begin
    write('Camino del archivo ENLACES: ');
    readln(N);
    assign(N+'ENLACES',E);
    rewrite(E);
    if Iorestart<> 0 then

```

```

begin
  writeln('Error al crear el archivo de listas');
  S:= false
end
else begin
  {Los dos archivos están preparados. Son grabadas las direcciones
   hash y primera dirección libre}
  write(E,M);
  write(E,0);
  for I:=0 to M do
    V[I]:= nil;
  S:= true
end
end;
end;

procedure Insertar(var F:Ftiendas;var V:Vector;
                    var Sg:integer;R:Articulo);
{Se añade el registro en la posición Sg del archivo.}
var
  H: Rango;
begin
  filepos(F,Sg);
  write(F,R);
  {Según la dirección hash, es insertado el número de registro en
   la lista de colisiones}
  H:= Hash(R.Idt);
  Inserprim(Sg,V[H]);
  Sg:= Sg+1;
end;

procedure Borrar(var F:Ftiendas;var V:Vector;R:Articulo);
{Elimina R de la lista que le corresponde según si clave, aunque
 físicamente queda en el archivo.}
var
  H: Rango;
  Esta: boolean;
  D: integer;
begin
  H:= Hash(R.Idt);
  BuscaReg(V,R,Esta,D);
  if Esta then
    Suprime(D,V[H]) { Elimina de la lista }
  else
    writeln('El articulo no se encuentra en el archivo')
end;

procedure BuscIdt(var F:Ftiendas; C:String[5];
                  V:Vector;var Esta:boolean; var R:Articulo);
{Busca en la tabla de listas el registro con la clave Idt}
{Devuelve el registro R}
var
  H: Rango;
  Nr: integer;
  Lc: PtrLst;

```

```

begin
  Esta:= false;
  H:= Hash(C);
  Lc:= V[H];
  while not Esta and not Esvacia(Lc) do
    begin
      Nr:= Lc^.Nreg;
      filepos(F,Nr);
      read(F,R);
      Esta:= R^.Idt=C;
      if not Esta then
        Lc:= Lc^.Sgte
    end
  end;

procedure BuscaRg(var F:Ftiendas;R:Articulo;
                  V:vector;var Esta:boolean;var P:integer);
{Busca el registro R en la estructura V, devuelve el número de registro R}
var
  H: Rango;
  RR: Articulo;
  Lc: PtrLst;
begin
  Esta:= false;
  H:= Hash(R.Idt);
  Lc:= V[H];
  repeat
    if not Esvacia(Lc) then
      begin
        P:= Lc^.Nreg;
        filepos(F,P);
        read(F,RR);
        Esta:= RR=R;
        if not Esta then
          Lc:= Lc^.Sgte
      end
    until Esta or Esvacia(Lc);
end;

procedure InicProces(var F:Ftiendas;var E:Enlaces; var V:Vector;
                     var Sg:integer;var S:boolean);
{Prepara la estructura de listas enlazadas a partir del archivo de
enlaces. Devuelve la siguiente dirección libre del archivo de datos}
var
  I: Rango;
  N: string;
  K,Y,D: integer;
begin
  write('Camino del archivo ENLACES: ');
  readln(N);
  assign(N+'ENLACES',E);
  reset(E);
  if Iorresult<> 0 then
    begin
      writeln('Error al abrir el archivo de listas');
      S:= false
    end
end;

```

```

end
else begin
  read(E,K);
  read(E,Sg);
  {Inicializa cada lista}
  for I:=0 to K do
    V[I]:= nil;
    {Lectura de cada dirección hash y de las posiciones de los
     registros en colisión}
  while not eof(E) do
begin
  read(E,K,Y); {Dir. hash y número de reg,s}
  for I:=1 to Y do
  begin
    read(E,D);
    Inserprim(D,V[K])
  end;
  S:= true
end
end;
write('Camino/Nombre del archivo de datos: ');
readln(N);
assign(N,F);
{$I-}
reset(F);
if Iorestart<> 0 then
begin
  writeln('Error al abrir el archivo de datos');
  S:= false
end
end;

procedure FinProces (var E:Enlaces;V:Vector;Sg:integer);
{Almacena la estructura de listas enlazadas en el archivo de enlaces}
var
  I: Rango;
  K,Y,D: integer;
  L: Ptrlst;
begin
  rewrite(E);
  write(E,M);
  write(E,Sg);
  for I:=0 to M do
  begin
    write(E,I); {dirección hash}
    L:=V[I];
    K:=Cuantos(L);
    write(E,K);
    for D:=1 to K do
    begin
      begin
        write(E,L^.Nreg);
        L:=L^.Sgte
      end
    end
  end;
end;
end. { fin de la unidad }

```

A continuación escribimos la unidad de programa que utiliza a la unidad *hash2*. En el programa creamos el archivo, añadimos nuevos registros, borramos registros...

```

program Tienda(Ftienda,Enlaces);
uses
  crt,Hash2;
const
  L= 4;
type
  Opc= 0..L;
var
  Op: Opc;
  F: Ftienda;
  E: Enlaces;
  V: Vector;
  A: Articulo;
  Flag: boolean;

procedure Menu(var Q:Opc);
begin
  writeln (' 1. Crear '); writeln;
  writeln (' 2. Añadir '); writeln;
  writeln (' 3. Borrar '); writeln;
  writeln (' 4. Buscar '); writeln;
  writeln (' 0. Salir '); writeln;
  repeat
    write('¿Opción elegida?: ');
    readln(Q)
  until Q in [0..L];
end;

procedure LeeArtclo(var R:Arteta);
var
  Id: string[5];
  C: char;
begin
  write('Código de Identificación:'); readln(Id);
  if Id<>'' then
  with R do
  begin
    Idt:= Id;
    write ('Unidades: '); readln(Unidades);
    write ('Precio: '); readln(Precio);
    write ('Fecha de Caducidad(D/M/A): ');
    with Caducidad do
      readln(D,M,A)
  end
  end;
end;

procedure Annadir(var F:Ftienda;var V:Vector;Sg:integer);
var
  R: Articulo;
begin
  writeln('Termina la entrada pulsando ENTER' );

```

```

repeat
  LeeArtculo(R);
  if R.Idt<>'' then
    Insertar(F,V,Sg,R)
  until R.Idt= ''
end;

procedure Eliminar(var F:Ftienda;var V:Vector);
var
  R: Atleta;
  Id: string[5];
  S: boolean;
  {Pide el apellido del atleta que se desea borrar del archivo}
begin
  write('Identificación del artículo: ');
  readln(Id);
  BuscIdt(F,Id,V,S,R);
  if S then
    Borrar(F,V,R)
end;

procedure Mostrar(R:Articulo);
begin
  with R do
  begin
    writeln ('Identificación: ',Idt);
    writeln ('Unidades : ',Unidades);
    writeln ('Precio/unidad : ',Precio);
    with Caducidad do
      writeln ('Caducidad : ',D,'-',M,'-'A)
  end
end;

procedure Presentar(var F:Ftienda);
var
  R: Articulo;
  Id: string[5];
  Esta: boolean;
  {Pide la identificación del artículo}
begin
  write('Identificación del artículo: ');
  readln(Id);
  BuscIdt(F,Id,V,Esta,R);
  if Esta then
    Mostrar(R)
  else
    writeln('Código no está en el archivo')
end;

procedure Actualiza(var F:Ftienda;var E:Enlaces;
                     var V:Vector;var Oper:boolean;Op:integer);
begin
  if not Oper then
    InicProces(F,E,V,Sg,Oper);
  if Oper then
    case OP of

```

```

2: Annadir(F,V,Sg);
3: Eliminar(F,V);
4: Presentar(F)
end;
begin
  Clrscr;
  Flag:= false;
repeat
  Menu(Op);
  if Op=1 then
    Crear(F,E,V,Flag)
  else
    Actualiza(F,E,V,Flag,Op)
until (Op=0) or Flag
end.

```

18.3. ARCHIVOS INDEXADOS

En un libro, en un listín telefónico..., las búsquedas se realizan situándose primero en el índice en el que están en orden alfabético las palabras, o bien los nombres, asociados con el número de página en que se encuentra. Ese es también el fundamento de la organización de los archivos indexados.

La organización de un archivo de registros clasificados de acuerdo con los valores de un campo clave se conoce como archivos indexados.

Para facilitar las búsquedas se utiliza una tabla de índices, que a su vez estará grabada en un segundo archivo, llamado índice disperso o simplemente archivo de índices.

El archivo de índices consta de pares (Cl, r) donde Cl es un valor del campo clave elegido y r es la dirección física del registro que tiene dicha clave.

La tabla de índices se mantiene ordenada respecto al campo clave. De tal forma que las búsquedas serán mucho más eficientes al aplicar el algoritmo de búsqueda binaria.

18.3.1. Archivo indexado de una fonoteca

Planteamos un supuesto en el que plasmar la representación y las operaciones de un archivo indexado.

Una fonoteca está formada por un número indeterminado de documentos sonoros. Cada documento queda identificado por los siguientes campos: *Referencia*, *Descripción* y *Número de copias*.

La Referencia es un campo alfanumérico de 12 caracteres, lo utilizamos como campo clave.

ANÁLISIS

La organización del archivo fonoteca va a ser indexada, el campo clave es la *Referencia* del documento sonoro (disco, cinta...), se supone una correspondencia biunívoca entre Referencia y documento sonoro.

Una tabla en memoria contiene pares de elementos (*Referencia*, *R*) ordenados respecto a *Referencia*.

La operación de añadir un registro al archivo supone reservar un registro del archivo de datos y añadir a la tabla ordenada el campo clave *Referencia* y el número de registro.

La operación de eliminar un registro supone buscar en la tabla el campo clave y así obtener la localización del registro. El registro a eliminar se marca como borrado y se empaqueta la tabla.

Todo tipo de búsqueda en la tabla de índices se hará utilizando la técnica de búsqueda binaria y así aprovechar el hecho de que la tabla está ordenada.

Siempre que se acabe el proceso con el archivo hay que almacenar la tabla de índices en el archivo de índices (índice disperso). De igual forma, siempre que se empiece de nuevo el proceso del archivo habrá que leer el archivo índice para formar la tabla de índices.

Las operaciones que se pueden realizar sobre un archivo indexado son las operaciones generales sobre archivos: añadir un registro, recorrer el archivo, consulta de un registro, eliminar. Además hay que incorporar las operaciones sobre tablas para el manejo de la tabla de índices.

En la unidad Índices se encapsula la representación de los datos y las operaciones.

Unidad archivo indexado

```

unit Indices;
interface
const
  M= 1000;
type
  Tclave:string[12];
  Elemento= record
    Copias :integer;
    Ref :Tclave;
    Dscrpcon:string[80];
    Actis :0..1;
  end;
  Indice= record
    Rf:Tclave;
    Nr:integer
  end;
  Tindex= record
    Tabla:array[1..M] of Indice;
    N :0..M
  end;
  Arfono= file of Elemento;
  Arindx= file of Indice;

procedure Annade(var F:Arfono;R:Elemento;var T:Tindex);
procedure Elimina(var F:Arfono;Rf:Tclave;var T:Tindex);
procedure Recorre(var F:Arfono;var T:Tindex);
procedure Consulta(var F:Arfono;var T:Tindex;X:Tclave);
  { Operaciones auxiliares }
procedure FormaTabla(var Fx:Arindx;var T:Tindex);

```

```

procedure Finproceso(var Fc:Arindx;var T:Tindex);
  { Operaciones sobre la tabla de índices }
procedure InsertT(var T:Tindex;Rf:Indice);
function Locat(T:Tindex;Rf:Tclave):0..M;
procedure Empaquar(var T:Tindex;Nt:integer);
implementation
procedure Annade(var F:Arfono;R:Elemento;var T:Tindex);
{Añade el registro R en el archivo F. Además inserta entrada con la clave
y el número de registro en la tabla de índices }
var
  NN: integer;
  I: integer;
  Rc: Indice;
begin
  {El nuevo registro se añade como siguiente al número de registro más
  alto en la tabla }
  NN:=0;
  for I:=1 to T.N do
    if T.Tabla[I].Nr>NN then
      NN:=T.Tabla[I].Nr;
  NN:=NN+1;
  seek(F,NN);
  write(F,R);
  {Ahora se inserta en la tabla de indices}
  Rc.Rf:= R.Ref;
  Rc.Nr:= NN;
  InsertT(T,Rc);
end;

procedure Elimina(var F:Arfono;Rf:Tclave;var T:Tindex);
{Elimina R del archivo. Busca en la tabla de índices la clave de R y así
obtiene el número de registro. De la tabla desaparece la entrada, en el
archivo se marca R como borrado.}
var
  Nt: integer;
  Nr: integer;
begin
  Nt:=Locat(T,R.Ref);
  if (Nt>0) then
  begin
    Nr:=T.Tabla[Nt].Nr
    seek(F,Nr);
    read(F,R);
    R.Actis:=0;
    seek(F,Ng)
    write(F,R);
    Empaquar(T,Nt) {Empaque la tabla, borrando entrada Nr}
  end
end;

procedure Recorre(var F:Arfono;var T:Tindex);
{Procesa cada registro del archivo. El proceso consiste en visualizar
cada uno de los registros. El orden de proceso es el que marca la tabla
de índices.}

```

```

var
  I: integer;
  R: elemento;
begin
  for I:=1 to T.N do
  begin
    seek(T.Tabla[I].Nr);
    read(F,R);
    Escribe(R)
  end
end;

procedure Consulta(var F:Arfono;var T:Tindex;X:Tclave);
{Busca una clave en la tabla para visualizar el registro que le
corresponde}
var
  P: Integer;
  R: elemento;
begin
  P:= Locat(T,X);
  if P<>0 then
  begin
    seek(F,T.Tabla[P].Nr);
    read(F,R);
    Escribe(R)
  end
  else
    writeln('Clave ',X,' No se encuentra')
end;

procedure Escribe(R:elemento);
begin
  writeln;
  writeln(R.Dscrpcion);
  writeln('Copias: ',R.Copias,' ':8, 'Referencia: ',R.Ref);
end;

function Locat(T:Tindex;Rf:Tclave):0..M;
{Es aplicada la búsqueda dicotómica para así aprovechar la ordenación
de la tabla}
var
  Mt,I,J: integer;
  Encontrado: boolean;
begin
  Encontrado:= false;
  I:= 1;
  J:= T.N;
  Mt:= (I+J) div 2;
  while (I<=J) and not Encontrado do
    if Rf< T.Tabla[Mt].Rf then
      J:= Mt-1
    else if Rf> T.Tabla[Mt].Rf then
      I:= Mt+1
    else
      Encontrado:= true;
end;

```

```

if Encontrado then
  Locat:= Mt
else
  Locat:= 0
end;

procedure InsertT(var T:Tindex;Rc:Indice);
var
  I,J: integer;
  Sw: boolean;
begin
  if Locat(T,Rc.Rf)<>0 then
    writeln('Atención la clave ',Rf,' ya existe.')
  else begin
    I:= T.N; Sw:= false;
    while (I<=1) and not Sw do
      if T.Tabla[I].Rf>Rc.Rf then
        begin
          T[I+1]:= T[I];
          I:=I-1
        end
      else {Clave menor, I+1 posición de inserción}
        Sw:= true;
    T.Tabla[I+1]:= Rc;
    T.N:=T.N+1
  end
end;

procedure Empaquear(var T:Tindex;Nt:integer);
{En la tabla de índices ha habido una eliminación en la posición Nt. Hay
que comprimir la tabla}
var
  I: integer;
begin
  for I:= Nt+1 do T.N do
    T.Tabla[I-1]:= T.Tabla[I];
  T.N:= T.N-1
end;

procedure FormaTabla(var Fx:Arindx;var T:Tindex);
{En esta rutina se transfieren los índices desde el archivo a la tabla de
índices}
var
  I: integer;
begin
  I:=0;
  T.N:=0;
  while not eof(Fx) do
    with T do
      begin
        N:=N+1;
        read(Fx,Tabla[N]);
      end
    {N es actualizada según son leídos registros y guardados en
     la tabla}
  end;
end;

```

```

procedure Finproceso(var Fc:Arindx;var T:Tindex);
var
  I: integer;
begin
  rewrite(Fc);
  for I:= 1 to T.N do
    write(Fc,T.Tabla[I])
  end;
end.

```

18.3.2. Programa de manipulación de archivos indexados

En el siguiente programa manejamos la unidad Indices. En el programa se añaden registros, puede eliminarse registros sonoros, recorrer todo el archivo... En la primera acción se da de alta al archivo y se añade un número determinado de registros.

```

program Sonoro(Fono, Index);
uses
  crt,Indices;
const
  L= 4;
type
  Opc= 0..L;
var
  Op: Opc;
  Fono: Arfono;
  Index: Aindex;
  Inic: boolean;
  T: Tindex;

procedure Menu(var Q:Opc);
begin
  writeln (' 1. Añadir '); writeln;
  writeln (' 2. Eliminar'); writeln;
  writeln (' 3. Recorrer '); writeln;
  writeln (' 0. Salir ') ; writeln;
  repeat
    write('Opción elegida:?' );
    readln(Q)
  until Q in [0..L];
end;

procedure Leefono(var R:Elemento);
var
  Id: Tclave;
begin
  write('Referencia unidad sonora: '); readln(Id);
  if Id<>'' then
    with R do
    begin
      Ref:= Id;
      write('Copias: '); readln(Copias);
    end;
  end;
end;

```

```
    write('Descripción: '); readln(Dscrpcon);
    Actis:= 1
  end
end;

procedure Alta(var F:Arfono;var T:Tindex);
var
  R: Elemento;
begin
  writeln('Proceso termina pulsando Enter al primer campo');
  Leefono(R);
  while R.Id<>'' do
    begin
      Annade(F,R,T);
      Leefono(R)
    end
  end;
end;

procedure Nuevoreg(var F:Arfono;var T:Tindex);
var
  R:Elemento;
begin
  Leefono(R);
  if R.Id<>'' then
    Annade(F,R,T)
end;

procedure Borrar(var F:Arfono;var T:Tindex);
var
  Rf:Tclave;
begin
  write('Clave del registro a dar de baja: ');
  readln(Rf);
  Elimina(F,Rf,T)
end;

procedure Abrir(var F:Arfono; var Ix:Aindex
                           var Daralta: boolean);
var
  S: string;
begin
  Daralta:= false;
  write('Camino del archivo Fono:');readln(S); assign(S+'Arfono',F);
  {$I-}
  reset(F);
  {$I+}
  if Ioreult<> 0 then
    begin
      rewrite(F);
      Daralta:= true
    end;
  write('Camino archivo de indices:');readln(S);
  assign(S+'ArIndexo',Ix);
  if not Daralta then reset(Ix);
end;
```

```

begin
  Clrscr;
  Abrir(Fono, Index, Inic);
  {En caso de proceso inicial son dados de alta registros}
  if Inic then
    Alta(Fono, T);
  repeat
    Menu(Op);
    case Op do
      1: Nuevoreg(Fono, T);
      2: Eliminar(Fono, T);
      3: Recorre(Fono, T);
    end
  until Op=0;
  {Por último, se almacena en archivo de índices la tabla}
  Finproceso(Index, T);
end.

```

RESUMEN

Los archivos de texto son archivos de caracteres que se dividen en líneas, separados por caracteres de fin de línea. Los archivos se declaran en Pascal mediante las sentencias:

```

type registro = record
  ... campos
end;
archivo = file of registro;

```

Los archivos de texto se declaran del siguiente modo:

```

type
  texto = text;

```

Los *archivos relativos* o *aleatorios* se pueden manejar accediendo en forma directa a cualquiera de los registros, sin necesidad de haber leído otros anteriores. El acceso es por posición, especificando el número de orden del registro a usar. En Pascal estándar no existen archivos relativos.

Los *archivos indexados* se pueden manejar accediendo en forma directa a cualquiera de los registros, sin necesidad de haber leído otros anteriores. El acceso es por contenido, especificando el valor de uno de los campos del registro, que se denomina *clave de acceso*. Las operaciones primitivas de manipulación de un archivo indexado simple son: *Iniciar*, *Eliminar*, *Leer-Indexado*, *EscribirIndexado*, *BorrarClave* y *ExisteClave*. Un archivo secuencial indexado posee las mismas operaciones primitivas que el archivo indexado simple y, además, dispone de otros primitivos adicionales para realizar el recorrido del archivo completo en el orden de las claves.

Como regla general, el espacio ocupado por el archivo indexado se reparte en dos zonas: *a*) en una zona se almacenan los datos en forma no ordenada; *b*) en otra zona se construyen unos índices que permiten acceder rápidamente al registro apropiado, bien en forma directa o secuencial, según convenga.

EJERCICIOS

- 18.1. Escribir un procedimiento que tenga como entrada dos archivos de texto y forme un tercer archivo que sea la concatenación de ellos.
- 18.2. Se desea escribir un procedimiento para escribir en un archivo de texto HEXADEC números en hexadecimal, de tal forma que los datos de entrada son valores enteros leídos del dispositivo estándar de entrada, son transformados a hexadecimal y escritos dos por línea en el archivo.
- 18.3. Escribir un procedimiento para leer los números en hexadecimal del archivo de texto HEXADEC creado en el ejercicio anterior y escribir los números en decimal por pantalla.
- 18.4. Los registros correspondientes a las claves 34, 67, 92, 40, 103, 21, 128, 98 se van a almacenar en una tabla *hash* de tamaño 7.
Mostrar el contenido de la tabla al usar el método de la división. Utilizar el método secuencial para resolver las colisiones.
- 18.5. Considerando los mismos registros que en el ejercicio anterior, mostrar la tabla al usar el método de encadenamiento en la resolución de colisiones.
- 18.6. Se desean grabar 5.000 registros en un archivo, sus claves ya están convertidas a números de registro (dirección lógica), toman valores relativos entre 1 y 7.500, en cubetas. Cada cubeta puede contener 10 registros, aunque sólo 8 posiciones de la cubeta se llenan de registros con diferentes claves. En caso de que existan colisiones, algunos sinónimos pueden situarse en las cubetas.
- a) ¿Cuáles cubetas se necesitarán para el archivo?
 - b) ¿En qué cubeta se situarán los registros con los números relativos: 0006 7114 4262 4618?
 - c) ¿Qué sucederá si existe un segundo registro cuya dirección sea 7114 (sinónimo)?
 - d) ¿Qué sucederá si son cuatro los registros con la dirección lógica 7114?
 - e) ¿Cuál es el factor de carga del archivo?
- 18.7. Escribir la función PLEGADO que tenga como entrada el campo clave de un registro de coches de alquiler y devuelva la dirección *hash* de dos dígitos utilizando el método de la mitad del cuadrado. Tener en cuenta que la clave siempre consta de 4 dígitos.
- 18.8. Tenemos las siguientes claves (4 dígitos) correspondientes a los alumnos de un colegio mayor: 2345, 4567, 8091, 3255, 6023, 7805, 1250, 4321.
Encontrar las direcciones *hash* de cada clave utilizando:
- a) El método de división, tomando $m = 31$.
 - b) El método del plegado.

PROBLEMAS

- 18.1. Se quiere compactar un archivo de texto con el siguiente criterio: eliminar cualquier sucesión de dos o más caracteres en blanco y sustituirlos por uno solo. Escribir un procedimiento que tenga como entrada un archivo de texto y lo compacte según el criterio expuesto.
- 18.2. Se tiene un archivo secuencial de cuentas corrientes, CORRIENTES, los campos de cada registro: titular, domicilio, teléfono, saldo, número de cuenta y fecha de apertura. Escribir

- un programa para generar a partir de CORRIENTES un archivo de CUENTAS, con la misma información. Eliminar los blancos innecesarios aplicando el problema 18.1.
- 18.3. En un archivo de texto hay líneas de la forma `#include NombreArchivo`. Escribirse un programa que lea el archivo mencionado y nos genere otro archivo de texto en el que se ha reemplazado la línea por el contenido del archivo mencionado.
- 18.4. Escribir un programa que tenga como entrada una palabra (cadena) y el nombre de un archivo. La salida por impresora (o por pantalla) ha de ser todas las líneas del archivo que contengan la palabra exactamente.
- 18.5. Una agencia de alquiler de coches ha comprado y matriculado 100 unidades. Las matrículas tienen como identificación M-0455-TR a M-0554-TR. Diseñar un archivo aleatorio cuyos registros son los datos de cada coche. Los registros serán de longitud fija, organizados por claves. Escribir un programa que cree el archivo.
- 18.6. Una compañía prevé llegar a tener 500 empleados, va a almacenar los datos personales en registros cuyo campo clave son los números de seguridad social (9 dígitos). Obtener una función de conversión para almacenar sus registros con un factor de carga del 75 por 100.
- 18.7. Escribir una función que tenga como entrada un registro y devuelva la dirección relativa correspondiente a un archivo de 1 a 1.000 direcciones. El campo clave es el número de seguridad social de 9 dígitos; el método a utilizar que sea el de plegamiento.
- 18.8. Un archivo de 675 registros tiene un factor de carga de 0,9. Convertir las claves 2345 y 6976 a direcciones relativas utilizando el método de conversión de base y suponiendo que las claves se escriben en base 16 y en base 8.
- 18.9. El archivo de texto EMPLEADOS de una empresa con 125 empleados contiene por línea un entero positivo correspondiente al número de nómina, el apellido, el nombre y el sueldo bruto anual. Realizar un programa que lea el archivo y añada los datos de cada empleado a una tabla *hash* con encadenamiento, tomando como clave el número de empleado. El archivo puede tener claves repetidas, en cuyo caso se considerará la que tenga mayor cantidad de ingresos.
- 18.10. Escribir un subprograma que tenga como entrada un registro y nos devuelva la dirección *hash* utilizando el método de plegado con inversión para encontrar la dirección, ésta ha de ser de dos dígitos. El campo clave es el número de alumno de 4 dígitos.

Ordenación externa

CONTENIDO

19.1. Ordenación externa. Métodos de ordenación.

19.2. Métodos de mezcla directa.

19.3. Métodos de mezcla equilibrada múltiple.

19.4. Método polifásico de ordenación externa.

RESUMEN.

EJERCICIOS.

PROBLEMAS.

REFERENCIAS BIBLIOGRÁFICAS.

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 analizan métodos de ordenación más populares y eficaces.

19.1. ORDENACIÓN EXTERNA. MÉTODOS 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:

$$\forall i < j \Rightarrow K(i) < K(j)$$

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 por contra el número de registros de los que puede constar un archivo elevado, tanto de 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.

19.2. MÉTODO DE MEZCLA DIRECTA (MEZCLA SIMPLE)

Este es el método más simple que utiliza el esquema iterativo de separar secuencias y mezclarlas. Se trabaja con el archivo original y dos archivos auxiliares. El proceso consiste:

1. Separar los registros del archivo original O en dos mitades F1, F2.
2. Mezclar 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 mitades F1, 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 O está ordenado.

EJEMPLO 19.1

Considérese un archivo que está formado por los registros de las claves enteras siguientes: 34 23 12 59 73 44 8 19 28 51 en el archivo O.

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 lo obtenemos tomando logaritmos, $i \geq \log_2 n$, $\lceil \log n \rceil$ 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 lo que no resulta interesante analizar el número de comparaciones.

19.2.1. Codificación

A continuación es codificado un programa que implementa el método de ordenación mezcla directa. Para facilitar la comprensión, los registros son de tipo entero. Además se

facilitan diversos procedimientos auxiliares, como el que genera un archivo de números aleatorios y el de listar por pantalla el archivo.

```

program MzclaDIREC;
{N: Número de registros, máxima secuencia. L: Longitud de cada
secuencia parcial}
uses crt;
type
  Tclave = integer;
  Treg = record
    Clave: Tclave;
  end;
  Tfichero = file of Treg;
var
  F,F1,F2:Tfichero;
  N,L:integer;

procedure Listar (var F : Tfichero; Nf : integer);
var
  k: integer;
  R: Treg;
begin
  reset(F);
  k := 0;
  writeln('Fichero ':20,Nf,' : ');
  while not eof(F) do
  begin
    k := k+1;
    read(F, R);
    write(R.Clave:4, ' ');
    if (k mod 15 = 0) then writeln
  end;
  writeln;
end;

procedure Asign_generar;
const
  nm = 45; {Número de registros para probar}
var
  I: integer;
  R: Treg;
  Nom: string;
begin
  {Asignar nombre externo a los ficheros}
  assign(F, 'c:\tp\bin\foriginal.dat');
  Nom:= 'c:\tp\bin\auxila01.dat';
  assign(F1, Nom);
  Nom:= 'c:\tp\bin\auxila02.dat';
  assign(F2, Nom);
  rewrite(F);
  {Generación de números aleatorios}
  randomize;
  for I:= 1 to nm do

```

```

begin
  R.Clave:= random(999);
  write(F, R)
end
end;

procedure Cuenta(var F:Tfichero;var Nrec:integer);
var
  R:Treg;
begin
  Nrec:=0;
  reset(F);
  while not eof(F) do
  begin
    Nrec:=Nrec+1;
    read(F,R)
  end
end;

procedure Distribuir(var F,F1,F2:Tfichero;L,Nrec:integer);
var
  Nsec,Resto,I:integer;
  procedure Subsecuencia(var F,T:Tfichero;L:integer);
  var
    R:Treg;
    J:integer;
  begin
    for J:=1 to L do
    begin
      read(F,R);
      write(T,R)
    end
  end;
begin
  Nsec:= Nrec div (2*L);{Subsecuencias en cada archivo}
  Resto:= Nrec mod (2*L);
  reset(F);
  rewrite(F1); rewrite(F2);
  for I := 1 to Nsec do
  begin
    Subsecuencia(F,F1,L);
    Subsecuencia(F,F2,L)
  end;
  {Se procesa el resto de registros}
  if Resto > L then
    Resto := Resto-L
  else begin
    L := Resto;
    Resto := 0
  end;
  Subsecuencia(F,F1,L);
  Subsecuencia(F,F2,Resto)
end;

procedure Mezclar(var F1,F2,F:Tfichero;
  var L:integer;Nrec:integer);

```

```

var
  Nsec,Resto,S,
  I,J,K,L1,L2 :integer;
  R1,R2: Treg;

procedure Proceso(var F,T:Tfichero;var R:Treg;
                  var Cuenta:integer);
begin
  write(F,R);
  Cuenta := Cuenta+1;
  if not eof(T) then read(T,R)
end;
begin
  Nsec := Nrec div (2*L);{Subsecuencias resultantes}
  Resto := Nrec mod (2*L);
  rewrite(F);
  reset(F1); reset(F2);
  read(F1,R1);read(F2,R2);
  for S := 1 to Nsec+1 do
begin
  L1 := L;L2 := L;
  if S = Nsec+1 then
    {Es procesado el resto en la última iteración}
    if Resto > L then
      L2 := Resto-L
    else begin
      L1 := Resto;
      L2 := 0
    end;
  I := 1;J := 1;
  while (I <= L1) and (J <= L2) do
    if R1.Clave < R2.Clave then
      Proceso(F,F1,R1,I)
    else
      Proceso(F,F2,R2,J);
    {Los registros no tratados en la subsecuencia se escriben
     directamente}
  for K := I to L1 do
    Proceso(F,F1,R1,I);
  for K := J to L2 do
    Proceso(F,F2,R2,J);
end;
  L := 2*L
end;

begin {Bloque principal)
  Clrscr;
  Asign_generar;
  Listar(F,0);
  Cuenta(F,N);
  L := 1;
  while L < N do
begin
  Distribuir(F,F1,F2,L,N);
  Mezclar(F1,F2,F,L,N);

```

```

end;
Listar(F,0)
end.

```

19.3. MÉTODO DE 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, esto puede conseguirse incrementando el número de archivos auxiliares.

Supongamos que se tienen w tramos distribuidos equitativamente en m archivos, la mezcla de los w tramos produce 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 mediante mezcla de m -uplas tramos es $\lceil \log_m n \rceil$, como cada pasada realiza n operaciones de movimiento con los registros la eficiencia es $O(n \log_m n)$. La mejora obtenida en cuanto al número de pasadas es $\log_2 m$ veces menos que se transfiere cada registro.

La mezcla equilibrada múltiple utiliza m archivo 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. El método puede ser expresado:

1. Distribuir registros del archivo original por tramos en los $m/2$ primeros archivos auxiliares. A continuación, estos 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 de 2 hasta que quede un único tramo, entonces la secuencia está ordenada.

19.3.1. Tipos de datos

La principal variación en cuanto a tipos datos es que para representar los archivos auxiliares se define el tipo array de archivos. Por facilidad en la comprensión los registros son de tipo del de la clave (entero).

```

const
  n= { Número de archivos auxiliares };
  n2= n/2;
type
  Tfichero= file of Tclave;
  Nfichero= 1..n;
var
  F: array[Nfichero] of Tfichero;

```

La variable F representa a los archivos auxiliares, alternativamente la primera mitad y la segunda irán cambiando su cometido, entrada o salida.

19.3.2. Cambio de finalidad de archivo

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.

`C: array[Nfichero] of Nfichero; Tabla de indices de archivo.`

Inicialmente $C[i]:=i \quad \forall i \in Nfichero$. Por lo que los archivos de entrada serán $F[C[1]], F[C[2]], \dots, F[C[n_2]]$; los ficheros de salida son la otra mitad $F[C[n_2+1]], F[C[n_2+2]], \dots, F[C[n]]$.

Para realizar el cambio de archivo de entrada por el de salida hay que intercambiar los valores de las dos mitades de la tabla de correspondencia:

```

C[1] ↔ C[n2+1]
C[2] ↔ C[n2+2]
.
.
.
C[n2] ↔ C[n]

```

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 otra tabla de correspondencia para archivos de entrada Cd para su acceso. La tabla Cd tiene en todo momento los índices de los archivos de entrada activos (en los que no se ha alcanzado el fin de tramo).

19.3.3. 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 $F[C[1]]$.

El número de tramos a mezclar llegará un momento que sea menor que el número de archivos de entrada. La variable K1 contiene el número de archivos de entrada, cuando el número de tramos L sea mayor o igual que n2, el valor de K1 será n2; en caso de ser L menor que n2, K1 será L, y cuando L sea 1, K1 será 1 y el archivo $F[C[1]]$ estará ordenado.

19.3.4. Codificación

El programa presenta unos procedimientos de ayuda para la prueba del método de ordenación. Así dispone de una rutina que genera el archivo original, con claves aleatorias, y de una rutina para asignar nombres a los archivos auxiliares.

```

program Mezcla_equilibrada;
uses
  crt;
const
  n = 6;
  n2 = n div 2;
type
  Tclave = integer;
  Tfichero = file of Tclave;
  Nfichero = 1..n;
  T_item = array [1..n2] of Tclave;
  T_nfic = array [Nfichero] of Tfichero;
  F_tramo = array [1..n2] of boolean;
var
  f0: Tfichero;
  f: array [Nfichero] of Tfichero;

procedure Distribucion (var L: integer);
var
  j: Nfichero;
  x, Ant: Tclave;
begin
  reset(f0);
  Ant := -maxint;
  x := Ant+1; { inicialización }
  j := 1;
  L := 0;
  repeat
    {Copiar un tramo de f0 al fich j}
    while (Ant <= x) and not eof(f0) do
      begin
        read(f0,x);
        if (Ant <= x) then
          begin
            write(f[j],x);
            Ant:= x
          end
        end;
    end;
    L:= L+1; {Ha sido formado un nuevo tramo}
    if j < n2 then
      j:= j+1
    else
      j:= 1;
    if not eof(f0) then {Empieza nuevo tramo}
      begin
        write(f[j], x);
        Ant:= x
      end
  end;
end;

```

```

else if eof(f0)and (Ant > x) then {tramo con último elemento}
begin
  write(f[j],x);
  L:= L+1
end;
until eof(f0);
end;

procedure Minimo(It:T_item; At:F_tramo; n:integer;
                  var Ix: integer);
var
  i: integer;
  m: Tclave;
begin
  i:= 0;
  Ix:= 0;
  m:= maxint;
  while i < n do
begin
  i:= i+1;
  { si activo }
  if At[i] and (It[i] < m) then
  begin
    m:= It[i];
    Ix:= i
  end
end
end;

procedure Leeritems (var It: T_item; nfe:integer;cd: T_nfic);
var
  j: integer;
begin
  for j:= 1 to nfe do
    read(f[cd[j]], It[j]); {lee un item de cada fichero de entrada}
end;

function Fintramos (At: F_tramo; nfe: integer): boolean;
var
  i: integer;
begin
  Fintramos:= true;
  for i:= 1 to nfe do
  if At[i] then
    Fintramos:= false;
end;

procedure Asign_generar;
const
  nm = 100;
var
  i: integer;
  x: Tclave;
  Nom: string;

```

```

begin
    { Asignar nombre externo a los ficheros }
    assign(f0, 'c:\tp\forignal.dat');
    Nom:= 'auxila00.dat';
    for i:=1 to n do
    begin
        if i < 10 then
            Nom[8]:= chr(ord('0')+i)
        else begin {suponemos máximo dos dígitos}
            Nom[7]:= chr(ord('0')+i div 10);
            Nom[8]:= chr(ord('0')+i mod 10)
        end;
        assign(f[i], Nom)
    end;
    rewrite(f0);
    { Generación de números aleatorios }
    randomize;
    for i:= 1 to nm do
    begin
        x:= random(9999);
        write(F0, x)
    end
end;

procedure Listar (var f:Tfichero; Nf: Nfichero);
var
    k: integer;
    z: Tclave;
begin
    reset(f);
    k:= 0;
    writeln('Fichero ',Nf);
    while not eof(f) do
    begin
        k:= k+1;
        read(f, z);
        write(z:4,' ');
        if (k mod 15 = 0) then writeln
    end;
    writeln;
end;

procedure Mezclacinta;
var
    i, j, Cx      : Nfichero;
    k1, k2, L, t, Mx: integer;
    x, Tx, Anter   : Tclave;
    Itms          : T_item;
    c, Cd          : T_nfic;
    Fdc           : boolean;
    Actv          : F_tramo;
    ch             : char;
begin
    {Distribuir tramos iniciales en fich. 1 .. n2}
    for i:= 1 to n2 do
        rewrite(f[i]);

```

```

Distribucion(L);
for i:= 1 to n do {Inicializa indices de cinta}
  C[i]:= i;
  {Mezclar ficheros C[1]..C[n2] a C[n2+1]..C[n]}
repeat
  if L < n2 then k1:= L
  else k1:= n2;
  for i:= 1 to k1 do
begin
  Listar(f[C[i]],C[i]);
  reset(f[C[i]]);
  Cd[i]:= C[i]
end;
repeat read (ch) until ch in['A'...'z'];

L:= 0; { Número de tramos mezclados }
j:= n2+1; { índice de fichero de salida }
for t:= j to n do
  rewrite(f[C[t]]);
Leeritems(Itms,k1,cd); {Lectura del primer item de
                        cada archivo de entrada}
                        {Mezclar los tramos}
repeat
  k2:= k1; {k2 es el número de archivos activos}
  L:= L+1; {Mezcla de un nuevo tramo}
  for t :=1 to k2 do {Inic. de marca de fin tramo}
    Actv[t]:= true;
  repeat {seleccionar el elemento mínimo}
    Minimo(Itms, Actv, k2, Mx);
    {Mx tiene el índice del mínimo, es escrito en el archivo salida }
    write(f[C[j]], Itms[Mx]);
    Fdc:= eof(f[Cd[Mx]]);{fin archivo:de tramo}
    if Fdc then
    begin
      rewrite(f[Cd[Mx]]);{Eliminar archivo}
      Cd[Mx]:= Cd[k2];
      Cd[k2]:= Cd[k1];
      Itms[Mx]:= Itms[k2];
      Actv[Mx]:= Actv[k2];
      k1:= k1-1;
      k2:= k2-1;
    end
    else begin
      {Se lee el siguiente item del archivo Cd[Mx], para
       saber si es fin de tramo}
      Anter:= Itms[Mx];
      read(f[Cd[Mx]], Itms[Mx]);
      if (Anter > Itms[Mx])then {Fin de tramo}
        Actv[Mx]:=false
    end
  until Fintramos(Actv, k2);
  if j< n then {escribe en siguiente archivo}
    j:= j+1
  else
    j:= n2+1
until k1 = 0;

```

```

{Cambio en la finalidad de archivos:
 los de entrada pasan a ser los de salida}
for i:= 1 to n2 do
begin
  Cx:= C[i];
  C[i]:= C[i+n2];
  C[i+n2]:= Cx
end
until L = 1;
Listar(f[C[1]],C[1])
end;
begin
  Asign_generar;
  clrscr;
  reset(f0);
  Listar(f0, 1);
  Mezclacinta;
end.

```

19.4. MÉTODO POLIFÁSICO DE ORDENACIÓN EXTERNA

La estrategia seguida en el método de mezcla equilibrada múltiple emplea $2m$ archivos para m tramos. Puede mejorarse consiguiendo ordenar m tramos con solo $m + 1$ archivos, para ello hay que abandonar la idea rígida de pasada en la que la finalidad de los archivos de entrada no cambia hasta que se lean todos ellos.

El método polifásico utiliza m archivos para ordenar n registros de un archivo. 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. El método tiene la dificultad de que el número de tramos ha de ser conocido y responder a una secuencia que estará en función del número m de archivos.

Cabe recordar la propiedad base de todos los métodos de mezcla, y es que: «la mezcla de n tramos de los archivos de entrada se transforma en n tramos en el archivo de salida».

19.4.1. Ejemplo con tres archivos

A continuación mostramos un ejemplo en el que suponemos un archivo original con claves enteras. El número de archivos auxiliares va a ser $N = 3$ y el archivo origen consta de estos registros:

```

4 31 860' 202 272 670' 318' 161 371 425' 81 474' 70 840' 59 293 916' 367
773' 327 696 843' 717' 306' 162 329 465' 246 824'

```

Los tramos están separados por el carácter ', en total hay 13 tramos. Los archivos auxiliares para ordenación que sean $F1$, $F2$, $F3$ y la distribución inicial es tal que $F1$ contiene 8 tramos y $F2$ contiene 5 tramos:

F1: 4 31 860'318'81 474'59 293 916'327 696 843'306'162 329 465' 246 824'
 F2: 202 272 670'161 371 425'70 840'367 773' 717'

En la primera pasada se mezclan 5 tramos de F1 con 5 tramos de F2 y se forman 5 tramos en F3:

F1: 306'162 329 465' 246 824'
 F2:
 F3: 4 31 202 272 670 860'161 318 371 425'70 81 474 840'59 293 367 773
 916' 327 696 717 843'

El archivo F2 se ha leído, quedan 3 tramos en F1 y 5 tramos en F3. El proceso continúa mezclando 3 tramos de F1 y F3 en F2:

F1:
 F2: 4 31 202 272 306 670 860'161 162 318 329 371 425 465'70 81 246 474
 824 840'
 F3: 59 293 367 773 916'327 696 717 843'

Ahora ha sido el archivo F1 el primero en ser leído, quedan 3 tramos en F2 y 2 tramos en F3. Se mezclan 2 tramos desde F2, F3 a F1:

F1: 4 31 59 202 272 293 306 367 670 773 860 916'161 162 318 327 329 371
 425 465 696 717 843'
 F2: 70 81 246 474 824 840'
 F3:

A continuación se mezcla 1 tramo desde F1, F2 a F3:

F1: 161 162 318 327 329 371 425 465 696 717 843'
 F2:
 F3: 4 31 59 70 81 202 246 272 293 306 367 474 670 773 824 840 860 916

Por último se mezcla 1 tramo de F1, F3 a F2 y el archivo queda ordenado:

F2: 4 31 59 70 81 161 162 202 246 272 293 306 318 327 329 367 371 425 465
 474 670 696 717 773 824 840 843 860 916

Observamos que la sucesión de tramos que se han manejado —13, 8, 5, 3, 2, 1— es precisamente la de los números de Fibonacci:

$$\begin{aligned} f_{i+1} &= f_i + f_{i-1} \quad \forall i \geq 1 \\ f_1 &= 1 \\ f_0 &= 0 \end{aligned}$$

Se pone en forma tabular la distribución perfecta de los tramos en los archivos para cada pasada, numerando éstas en forma inversa:

L	$t_1^{(L)}$	$t_2^{(L)}$
0	1	0
1	1	1
2	2	1
3	3	2
4	5	3
5	8	5

(Total 13 tramos)

De la tabla anterior puede deducirse ciertas relaciones entre t_1, t_2 en un nivel:

$$\begin{aligned}t_2^{(L+1)} &= t_1^{(L)} \\t_1^{(L+1)} &= t_1^{(L)} + t_2^{(L)} \\ \forall L > 0 \text{ y } t_1^{(0)} &= 1, t_2^{(0)} = 0\end{aligned}$$

19.4.2. Ejemplo con cuatro archivos

En este otro ejemplo el archivo original consta de 31 tramos y disponemos cuatro archivos auxiliares para realizar la ordenación. Nos fijamos únicamente en el número de tramos que se mezclan en cada pasada partiendo de 31 tramos y la diferencia con los actuales son los que quedan:

F1	F2	F3	F4	
13	11	7	0	<i>Tramos en la distribución inicial</i>
6	4	0	7	<i>Mezcla de F1,F2,F3 en F4</i>
2	0	4	3	<i>Mezcla de F1,F2,F4 en F3</i>
0	2	2	1	<i>Mezcla de F1,F3,F4 en F2</i>
1	1	1	0	<i>Mezcla de F2,F3,F4 en F1</i>
0	0	0	1	<i>Mezcla de F1,F2,F3 en F4 y fin del proceso: F4 archivo ordenado.</i>

Ponemos de nuevo en forma tabular la distribución perfecta de los tramos en los archivos para cada pasada, numerando estas en forma inversa:

L	$t_1^{(L)}$	$t_2^{(L)}$	$t_3^{(L)}$
0	1	0	0
1	1	1	1
2	2	2	1
3	4	3	2
4	7	6	4
5	13	11	7

(total 31 tramos)

De la tabla pueden deducirse ciertas relaciones entre t_1, t_2, t_3 en un nivel:

$$\begin{aligned}t_3^{(L+1)} &= t_1^{(L)} \\t_2^{(L+1)} &= t_1^{(L)} + t_3^{(L)} \\t_1^{(L+1)} &= t_1^{(L)} + t_2^{(L)} \\ \forall L > 0 \text{ y } t_1^{(0)} &= 1, t_2^{(0)} = 0, t_3^{(0)} = 0\end{aligned}$$

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:

$$\begin{aligned}f_{i+1} &= f_i + f_{i-1} + f_{i-2} \quad \forall i \geq 2 \\f_2 &= 1 \\f_1 &= 0 \\f_0 &= 0\end{aligned}$$

como puede observarse en la columna $t_i^{(L)}$: $13 = 7 + 4 + 2$; $7 = 4 + 2 + 1$.

19.4.3. 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 y de la correspondiente sucesión de fibonacci. Puede generalizarse la distribución perfecta para n archivos, el número de tramos para cada pasada L se obtiene:

$$\begin{aligned}t_{n-1}^{(L+1)} &= t_1^{(L)} \\t_{n-2}^{(L+1)} &= t_1^{(L)} + t_{n-1}^{(L)} \\&\vdots \\t_2^{(L+1)} &= t_1^{(L)} + t_3^{(L)} \\t_1^{(L+1)} &= t_1^{(L)} + t_2^{(L)} \\&\forall L > 0 \text{ y } t_1^{(0)} = 1, t_i^{(0)} = 0 \quad \forall i \leq n-1\end{aligned}$$

Con estas relaciones puede conocerse de antemano el número de tramos necesarios para aplicar el método polifásico con n archivos. Es evidente que no siempre el número de tramos del archivo a ordenar va a coincidir con ese número perfecto, ¿qué hacer en ese caso? La respuesta es sencilla: los tramos necesarios para completar la distribución perfecta se simulan su existencia, como tramos vacíos. ¿Cómo tratar los tramos ficticios? La selección de un tramo ficticio de un archivo i es simplemente ignorar el archivo, y por consiguiente hay que desechar dicho archivo de la mezcla del tramo correspondiente. En el caso de que el tramo sea ficticio en los $n - 1$ archivos de entrada, 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 $n - 1$ archivo.

Se definen dos arrays, A y D. El array A contiene los números de tramos que ha de tener cada archivo de entrada en un nivel dado (nivel equivale a decir fila en las tablas de números de tramos). El array D contiene el número de tramos ficticios que tiene cada archivo. El proceso se inicia asignando a A la primera fila de la tabla de números de tramos, siempre $(1, 1, \dots, 1)$; a la vez a D también $(1, 1, \dots, 1)$. Se copia un tramo en el archivo i , se decremente $D[i]$, así con cada uno de los $n - 1$ archivos. Si no se ha terminado el archivo original, es determinado el número de tramos del siguiente nivel y los que hay que añadir a cada archivo para alcanzar el 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 según se vayan añadiendo tramos al archivo se vaya decrementando D_i .

Por ejemplo, para $n = 4$ archivos el primer nivel A(1,1,1), el segundo nivel A(2,2,1), por lo que los tramos a añadir $(2,2,1) - (1,1,1) = (1,1,0)$ y D será (1,1,0). Una vez completado los tramos para este nivel, D habrá quedado (0,0,0), se pasa al siguiente nivel. Los tramos para el tercer nivel 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).

Así, sucesivamente, cuanto más tramos haya más niveles se alcanzan, en cualquier caso una vez terminado el proceso de distribución del archivo original tenemos en A el número de tramos para ese nivel y en D el número de tramos ficticios que tiene cada archivo, necesario para después el proceso de mezcla.

19.4.4. Mezcla polifásica versus mezcla múltiple

Las principales diferencias de la ordenación polifásica respecto a la mezcla equilibrada múltiple:

1. En cada pasada hay un solo archivo destino (salida), en vez de los $m/2$ que necesita la mezcla equilibrada múltiple.
2. La finalidad de los archivos (entrada, salida) cambia en cada pasada rotativamente. 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 de 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 , $i = 1..m - 1$, esto nos dice que hay que mezclar $m - 1$ tramos ficticios, dando lugar a otro tramo ficticio en el archivo destino, se refleja incrementando D_n del archivo destino (de salida). No siempre será así, sino que habrá que mezclar un tramo de cada archivo que verifique $D_i = 0$, y disminuir en uno D_i en los demás archivos, así indicamos que hay 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 fin de fichero 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.

19.4.5. Algoritmo de mezcla

En el algoritmo de mezcla que expresamos a continuación C es la tabla de correspondencia de los $N - 1$ archivos, Actv es un vector lógico indexado por C[i] que está a verdadero (*true*) si el archivo que le corresponde está activo.

```

desde I <-1 hasta N-1 hacer
  C[I] <-I
fin_desde

repetir
  {Mezcla de los ficheros C[1]...C[N-1] hasta C[N])
  Z <- A[N-1] {número de tramos a mezclar en esta pasada}
  d[N] <- 0
  <Preparar para escribir F[C[N]]>
  repetir {Mezcla de un tramo desde los ficheros}
    k <- 0 {k representa el número de ficheros activos}
    desde I <- 1 hasta N-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 ficheros}
    sino {Todos los tramos son ficticios}
      d[N] <- d[N]+1 {tramo ficticio en el fichero destino}
    fin_si
    Z <- Z-1 {tramo ya mezclado}
  hasta Z = 0
  <Preparar para lectura F[C[N]]>
  <Rotar los ficheros en la tabla de correspondencia C>
  <Calcular los números de fibonacci A[I] del sgte nivel>
  <Preparar para escritura F[C[N]]>
  Nivel <- Nivel-1
hasta Nivel = 0

```

La mezcla de tramos se realiza de igual forma que en la mezcla múltiple: con un procedimiento de lectura de ítems de todos los archivos y la selección del mínimo.

19.4.6. Codificación

A continuación se muestra la codificación del programa de ordenación de ficheros por el método polifásico para $N = 4$ y siendo los registros claves enteras generadas aleatoriamente.

```

program Mezcla_Polifasica;
uses
  crt;
const
  N = 4;
type
  Tclave = integer;
  Tfichero = file of Tclave;
  Nfichero = 1..N;
  Tnum = array [Nfichero] of integer;
  Titem = array [1..N] of Tclave;

```

```

Tnfic = array [Nfichero] of Nfichero;
Ftramo = array [1..N] of boolean;
var
  F0 : Tfichero;
  F : array [Nfichero] of Tfichero;
  A,D,Ultimo : Tnum;
  Nivel : integer;
procedure Listar (var F : Tfichero;Nf : integer);
var
  K : integer;
  Z : Tclave;
begin
  reset(F);
  K := 0;
  writeln('Fichero ',Nf);
  while not eof(F) do
  begin
    K := K+1;
    read(F, Z);
    write(Z:4,' ');
    if (K mod 15 = 0) then writeln;
  end;
  writeln;
end;

procedure Distribucion (var A,D,Ultimo:Tnum; var Nivel :integer);
  {A : números de fibonacci para un nivel.
  D : número de tramos ficticios por cada fichero.}
var
  I,J : Nfichero;
  Nreg,Clave: Tclave;
  Proc: boolean; { Indica si procesado último registro}
procedure SelecFichero(var J: Nfichero);
  {Este procedimiento determina el índice de siguiente archivo.
  En el caso de haberse completado el nivel en curso, calcula
  los tramos del siguiente nivel A, y la diferencia con el anterior: D}
var
  I: Nfichero;
  Z,T: integer;
begin
  if D[J] < D[J mod(N-1)+1] then
    J := J mod(N-1)+1
  else if D[J] = 0 then {paso al sgte Nivel}
  begin
    Nivel := Nivel+1;
    Z := A[1];
    {Cálculo de números de fibonacci}
    for I := 1 to N-1 do
    begin
      T := A[I];
      A[I] := Z+A[I+1];
      D[I] := A[I]-T
    end;
    J := 1
  end
end;

```

```

procedure Copiartramo(J: Nfichero;var Anter,Nr: Tclave);
begin
  repeat
    write(F[J],Nr);
    Anter:=Nr;
    read(F0,Nr)
  until (Nr<Anter) or eof(F0)
  end;
begin { Distribución de tramos }
  {Primer nivel de números de Fibonacci}
  for I := 1 to N-1 do
  begin
    A[I] := 1;
    D[I] := 1;
    rewrite(F[I])
  end;
  reset(F0);
  Nivel := 1;
  J := 1; {Número de fichero}
  Proc := false;
  A[N] := 0;D[N]:=0; {Se corresponde con el fichero destino}
  read(F0,Nreg); {Lectura adelantada}
  repeat
    SelecFichero(J);
    CopiarTramo(J,Clave,Nreg);
    Ultimo[J]:= Clave;
    D[J] := D[J]-1 {Tramo ficticio sustituido por uno real}
  until (J = N-1) or eof(F0);

  {Tratamiento especial de fin de fichero}
  if eof(F0) then
  begin
    Proc := true;
    if (Clave < Nreg) then
      {Ha habido fin de fichero y continúa el tramo}
      write(F[J],Nreg)
    else {Nuevo tramo y por tanto nuevo fichero}
      if (J = N - 1) then
        begin
          if (Ultimo[1] > Nreg) then
            {No continúa el tramo en fichero 1}
            SelecFichero(J);
            write(F[1],Nreg)
          end
        else begin
          SelecFichero(J);
          write(F[J],Nreg);
          D[J] := D[J]-1
        end;
      end;
    end;
    {
    }
  while not eof(F0) do
  begin
    SelecFichero(J);
    if (Ultimo[J] <= Nreg) then

```

```

begin
  Copiartramo(J,Clave,Nreg); {Continúa el tramo}
  Ultimo[J] := Clave;
  if eof(F0) then
    begin
      Proc := true;
      if (Clave<=Nreg) then {Fin de fichero y mismo tramo}
        D[J]:=D[J]+1; {tramo ficticio no se sustituye}
        write(F[J],Nreg)
      end
    else begin {Nuevo tramo}
      SelecFichero(J);
      CopiarTramo(J,Clave,Nreg);
    end
  end
  else CopiarTramo(J,Clave,Nreg);
  Ultimo[J] := Clave;
  D[J] :=D[J]-1
end;
{Proceso de la última clave no seleccionada}
if not Proc then
  if (Clave<Nreg) then
    {Ha habido fin de fichero y continúa el tramo}
    write(F[J],Nreg)
  else begin {Nuevo tramo y nuevo fichero}
    SelecFichero(J);
    write(F[J],Nreg);
    if (Ultimo[J] > Nreg) then
      {No continúa el tramo en fichero J}
      D[J] := D[J]-1
    end
  end;
{Fin de proceso de distribución}

procedure MezclaMultiple(A,D:Tnum;Nivel:integer);
var
  I, J, Cx      : Nfichero;
  K,Cn,Dn,Z,Mx : integer;
  T, Anter      : Tclave;
  Itms         : Titem;
  C, Cd         : Tnfic;
  Fdc          : boolean;
  Actv         : Ftramo;

procedure Minimo(It:Titem;At:Ftramo;K:integer;var Ix:integer);
var
  I: integer;
  M: Tclave;
begin
  I :=0; Ix := 0;
  M := maxint;
  while I < K do
    begin
      I := I+1;
      if At[C[I]] and (It[I]<M) then {si activo}

```

```

begin
  M := It[I];
  Ix := I
end
end;
end;

procedure Leeritems (var It:Titem;Nfe:integer;C:Tnfic);
var
  J: integer;
begin
  for J := 1 to Nfe do
    read(F[C[J]],It[J]); {lee un item de cada fichero }
end;

function Fintramos (At:Ftramo;Nfe:integer):boolean;
var
  I: integer;
begin
  Fintramos:= true;
  for I := 1 to Nfe do
    if At[C[I]] then Fintramos := false;
end;

begin { De mezcla múltiple }
  for I := 1 to N do
    C[I] := I;
  for I := 1 to N-1 do
    reset(F[I]);
  {Mezclar ficheros C[1]..C[N-1] a C[N]}
  Leeritems(Items,N-1,C); {Lectura del primer item de cada
                           fichero de entrada}
repeat
  Z:= A[N-1]; D[N]:=0;
  rewrite(F[C[N]]);
  repeat
    K := 0; {Cuenta ficheros con tramo real}
    for I := 1 to N-1 do
      if D[I] = 0 then
        begin
          K:=K+1;
          Actv[C[I]] := true;
        end
      else begin {Tramo ficticio}
        D[I]:= D[I]-1;
        Actv[C[I]]:=false
      end;
    if K > 0 then {Mezcla de ficheros activos C[1]...C[K]}
    begin
      repeat {seleccionar el elemento mínimo}
        Minimo(Items,Actv,N-1,Mx);
        {Mx tiene el índice del mínimo}
        write(F[C[N]],Items[Mx]);
        Fdc := eof(F[C[Mx]]); {fin fichero:de tramo }
      if Fdc then
        Actv[C[Mx]]:= false
    end;
  end;
end;

```

```

    else begin
      {Es leído el siguiente item del fichero C[Mx], para
       saber si es fin de tramo}
      Anter := Itms[Mx];
      read(F[C[Mx]], Itms[Mx]);
      if (Anter>Itms[Mx]) then {Fin de tramo}
        Actv[C[Mx]] := false
      end
      until Fintramos(Actv,N-1);
    end
    else D[N] := D[N]+1; {mezcla de N-1 tramos ficticios}
    Z := Z-1
  until Z=0; {Fin de nivel, de "pasada")
{Rotación de archivos: nivel anterior de números de Fibonacci}
  Listar(F[C[N]],C[N]);
  reset(F[C[N]]); {En nueva pasada será fichero de entrada}
  read(F[C[N]],Itms[N]);
  Cn := C[N];Dn := D[N];
  T := Itms[N];Z := A[N-1];
  for I:= N downto 2 do
  begin
    C[I]:= C[I-1];D[I] := D[I-1];
    Itms[I] := Itms[I-1];
    A[I]:=A[I-1]-Z
  end;
  C[1] := Cn;D[1]:= Dn;
  Itms[1] := T;A[1] := Z;
  Nivel := Nivel-1
until Nivel = 0;
  Listar(F[C[1]],C[1]); {Fichero ordenado)
end;

procedure Asign_generar;
const
  nm = 300;
var
  i : integer;
  x : Tclave;
  Nom : string;
begin
  {Asignar nombre externo a los ficheros}
  assign(F0, 'c:\tp\forignal.dat');
  Nom:= 'auxila00.dat';
  for i :=1 to N do
  begin
    if i < 10 then
      Nom[8]:= chr(ord('0')+i)
    else begin { suponemos máximo dos dígitos }
      Nom[7]:= chr(ord('0')+i div 10);
      Nom[8]:= chr(ord('0')+i mod 10)
    end;
    assign(F[i], Nom)
  end;
  rewrite(F0);
  {Generación de números aleatorios}
  randomize;

```

```

for i := 1 to nm do
begin
  x := random(9999);
  write(F0, x)
end .
end;

begin
  Asign_generar;
  clrscr;
  reset(F0);
  Listar(F0,0);
  Distribucion(A,D,Ultimo,Nivel);
  MezclaMultiple(A,D,Nivel);
end.

```

RESUMEN

La ordenación de archivos se denomina, en general, *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, bien relativo o secuencial indexado, usando como clave el campo por el que se desea ordenar.

Si se desea realizar la ordenación de archivos usando solamente como estructuras de almacenamiento de archivos secuenciales con un formato similar al que se desea ordenar, hay que trabajar usando el esquema de *separación y mezcla*.

En el caso 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 los sucesivos registros en uno de los archivos auxiliares, mientras que dichos registros formen una secuencia ordenada.

Existen diferentes métodos avanzados de ordenación de archivos: *mezcla equilibrada*, *mezcla múltiple*, *mezcla equilibrada múltiple* y *mezcla polifásica*.

EJERCICIOS

- 19.1. El método de ordenación externa mezcla natural se diferencia de la mezcla directa en que en vez de mezclar tramos de longitud fija, mezcla tramos máximos. Entendiendo por tramos máximo como aquellos que están parcialmente ordenados. Por ejemplo, en la secuencia 34 51 12 17 28 36 15 19 los tramos son 34 51; 12 17 28 36; 15 19. La estrategia de la ordenación por mezcla natural sigue los mismos pasos que la mezcla directa con la diferencia indicada. Dada la secuencia de claves que se encuentran en el archivo original F₀ —14 27 33 5 8 11 23 44 22 31 46 7 19 3 7 8 11 1 99 23 40 6 11 14 17—, y utilizando dos ficheros auxiliares F₁, F₂, realiza un seguimiento del método hasta que quede una única secuencia ordenada.
- 19.2. Realice la ordenación del archivo del ejercicio 1 por el método de mezcla directa. ¿En qué método se realizan menos pasadas?
- 19.3. Escribe una secuencia de 50 registros, representados por claves enteras de 1 a 99. Realiza la ordenación con dos archivos auxiliares ficheros F₁, F₂ y utilizando los métodos de mezcla directa y mezcla natural. Compara el número de pasadas realizadas en cada método.

- 19.4. Escribir una secuencia de claves enteras de tantos tramos como sea necesario para una distribución perfecta, según los números de fibonacci en dos archivos F_1 , F_2 . A continuación haz un seguimiento del método de ordenación polifásico con tres archivos. Contabiliza el número de pasos realizados.
- 19.5. La secuencia de claves obtenida en el ejercicio 19.5 utilízala para hacer un seguimiento del método mezcla equilibrada múltiple con cuatro archivos. ¿Con qué método se ha realizado más rápida la ordenación?
- 19.6. Suponer dos archivos que están ordenados alfabéticamente. Escribe un algoritmo para mezclar ordenadamente los dos archivos en un tercero.

PROBLEMAS

- 19.1. Se tiene un archivo con los datos de los alumnos de un centro escolar *Pineda*. Cada alumno está representado por un registro con el campo clave primer apellido. A igualdad de apellido se toma como segundo campo clave el nombre del alumno. Escribir un programa para que dado tal archivo sea ordenado por el método de mezcla directa.
- 19.2. Algunos alumnos del centro escolar referido en el problema 19.1 han experimentado modificaciones, cambios de domicilio ... En el archivo *Cambios* están registradas las modificaciones, de tal forma que un registro contiene apellido, nombre y los campos modificados. El archivo *Cambios* está ya ordenado respecto la misma clave que *Pineda*. Escribir un programa que modifique el archivo *Pineda* confrontándolo con el archivo *Cambios*. El proceso se ha de hacer en memoria externa y aprovechando el hecho de estar ordenados.
- 19.3. Escribir los procedimientos en Pascal necesarios para codificar el método de ordenación mezcla natural. Previamente revise el primer ejercicio propuesto.
- 19.4. Escribir un procedimiento para mezclar dos archivos ordenados respecto de un campo entero. El procedimiento tiene como parámetros los archivos a mezclar, F_a y F_b , y debe de formar un tercer archivo, mezcla, que resulta de escribir en el ordenadamente los registros de los archivos F_a y F_b .
- 19.5. Para acelerar la ordenación de un archivo externo se dispone de un vector de registros...
- 19.6. En el archivo Arcos se tiene los arcos (x,y) que forman un grafo dirigido sin ciclos. Suponemos que no hay suficiente memoria interna para contener el conjunto completo de vértices o de arcos al mismo tiempo. Escribir un programa de clasificación topológica externa que escriba un ordenamiento lineal de los vértices, de modo que si $x \rightarrow y$ el vértice x aparezca antes que el y .

REFERENCIAS BIBLIOGRÁFICAS

Wirth, Niklaus: *Algoritmos y estructuras de datos*, México, Prentice-Hall, 1987.

PARTE **V**

Programación orientada a objetos

Objetos: Conceptos fundamentales y programación orientada a objetos

CONTENIDO

- 20.1. La estructura de los objetos: sintaxis.
 - 20.2. Secciones pública y privada.
 - 20.3. Definición de objetos mediante unidades.
 - 20.4. La herencia.
 - 20.5. Los métodos.
 - 20.6. Objetos dinámicos.
 - 20.7. Polimorfismo.
 - 20.8. Constructores y destructores.
 - 20.9. Procedimientos new y dispose.
 - 20.10. Mejoras en Programación Orientada a Objetos.
- RESUMEN.
EJERCICIOS.
PROBLEMAS.

El concepto de *programación orientada a objetos*, POO (Object Oriented programming, OOP), no es nuevo; lenguajes clásicos como Smalltalk se basan en ella y muchos otros con filosofía estructurada han añadido propiedades de objetos, tales como Turbo Borland Pascal, Delphi, C++ o Ada-95. La programación orientada a objetos se basa en la idea natural de la existencia de un mundo lleno de objetos, de modo que la resolución del problema se realiza en términos de objetos. El objeto es la extensión natural del tipo abstracto de datos, y la programación orientada a objetos, aquella que se basa en el objeto como componente fundamental.

POO entraña una metodología de programas, que comienza a tener una gran utilidad en la ingeniería del software. El diseño y la programación POO ayuda considerablemente a los ingenieros de los programas.

En este capítulo se tratan los conceptos básicos de la programación orientada al objeto (o a objetos) y cómo se pueden *implementar* los mismos en las versiones de Turbo Pascal. Como síntesis indicar que Turbo Pascal 5.5 sólo incluye cuatro palabras reservadas nuevas: *object*, *constructor*, *destructor* y *virtual*. Turbo Pascal 6.0 incluyó *private* y Turbo Pascal 7.0 ha incluido *public* e *inherited*.

20.1. LA ESTRUCTURA DE LOS OBJETOS: SINTAXIS

Los objetos se representan en la mayoría de los lenguajes de programación como estructuras de datos. A nivel de programa, los objetos se representan como registros en Pascal.

En Pascal estándar o versiones de Turbo Pascal anteriores a la 5.5 los datos se definen en estructuras y se manipulan con procedimientos y funciones. Por el contrario, en Turbo Pascal 5.5 los datos y los procedimientos se combinan en *objetos*. Un objeto contiene tanto las características de una entidad (sus datos) como su comportamiento o funcionamiento (sus procedimientos). Así como la ecuación fundamental de la programación estructurada en Pascal es el conocido título de una obra de Niklaus Wirth:

Algoritmos + estructuras de datos = programas

la ecuación básica en POO es

código + datos = objeto

En Turbo Pascal las características o atributos de un objeto se definen de un modo similar a un registro. La diferencia principal es que un registro consta totalmente de campos, mientras que un objeto tiene campos y procedimientos o funciones (*métodos*) asociados con ellos.

El primer paso al trabajar con objetos es declarar un tipo objeto. La sintaxis de una declaración de un tipo objeto es similar a la sección de interfaz de una unidad y la declaración de un tipo registro. El formato general para declarar un tipo objeto es:

```
identificador-tipo = object
  declaración de campos
```

```
  . . .
  declaración de métodos
```

```
end;
```

donde *identificador-tipo* es un identificador del nombre del objeto; *declaración de campos* tiene el mismo formato que la declaración usual de variables y *declaración de métodos* es una cabecera de las funciones o procedimientos (*métodos*).

Los objetos son tipos de datos definidos por el usuario. Estos tipos se llaman **clases**, sus campos dato se llaman **variables de instancia** y sus procedimientos o funciones se llaman **métodos**. Un método está asociado con un tipo de objeto específico y no se puede asociar con otros tipos de datos. Una declaración de la clase tiene el formato:

```
type
  NombreClase = object
    Campo1 : TipoVariable1;
    Campo2 : TipoVariable2;
    procedure NombreProc1;
    procedure NombreProc2;
    ...
    function NombreFunc1 : TipoFunc1;
    function NombreFunc2 : TipoFunc2;
    ...
  end;
```

Un **objeto** es una variable que tiene una clase como su tipo y se especifica con una declaración de la forma

```
var
  NombreObjeto : NombreClase;
```

Las clases se denominan **tipos objeto**. Cada objeto declarado de ese tipo se refiere como una **instancia** del tipo objeto. Un tipo objeto es una plantilla para crear objetos, al igual que un tipo registro es una plantilla para crear registros. Se pueden declarar dos variables del tipo *NombreClase*.

```
var
  MiObjeto1, MiObjeto2 : NombreClase;
```

MiObjeto1 y *MiObjeto2* tienen los mismos atributos (los mismos campos) y tienen acceso a los mismos métodos (*NombreProc1*, *NombreProc2*, *NombreFunc1*, etc.). La declaración de las dos variables tales como *MiObjeto1* y *MiObjeto2* crean dos áreas de memoria diferente que almacenan los valores de cada uno de los campos.

La declaración de una variable con un tipo objeto maneja la tarea de crear instancias de la clase de objetos. ¿Cómo se comunica el programa con los objetos que ha creado? Se hace invocando los métodos asociados con el tipo objeto. La llamada a los métodos de los objetos se llaman **mensajes** y tiene el formato

NombreObjeto.NombreMetodo

Por ejemplo, para declarar un tipo objeto llamado *MiTipoObjeto* que tenga dos campos (*Campo1* y *Campo2*), un procedimiento (*MiProcedimiento*) y una función (*MiFuncion*), se recurre a la sintaxis de la declaración del tipo:

```

type
  MiTipoObjeto = object
    Campo1 : Integer;
    Campo2 : Real;
    procedure MiProcedimiento;
    function MiFuncion (ArgumentoFuncion:Integer): Integer;
  end;

```

La declaración real de los métodos de objetos es a continuación de la sección de declaración de procedimientos del programa. La diferencia entre la declaración de un procedimiento y de una función, es que el identificador de tipo debe preceder al identificador de función o procedimiento en la cabecera. Por ejemplo, la declaración de MiProcedimiento y MiFuncion es:

```

procedure MiTipoObjeto.MiProcedimiento;
  .
  .
end;
function MiTipoObjeto.MiFuncion (ArgumentoFuncion: Integer): Integer;
  .
  .
end;

```

La notación punto se utiliza aquí en el mismo sentido que cuando se utiliza para especificar un campo en un registro o procedimiento, función, constante o variable en una unidad.

Regla

Los métodos se definen en dos partes del programa: (1) Las cabeceras se especifican en la declaración del tipo de objeto; (2) La implementación —cabecera más cuerpo— fuera de la definición del tipo.

La comunicación entre objetos se realiza vía mensajes. Por ejemplo, se pueden declarar dos variables del tipo MiTipoObjeto:

```

var
  MiObjeto1, MiObjeto2 : MiTipoObjeto;

```

Para invocar MiProcedimiento con MiObjeto1, se escribe

```

MiObjeto1.Miprocedimiento;

```

También invocaremos a MiProcedimiento con la otra variable de objeto

```

MiObjeto2.Miprocedimiento;

```

las variables de un tipo objeto son similares a otras variables Pascal en muchos aspectos. Por ejemplo, si se permite utilizar una variable de tipo objeto en una sentencia de asignación tal como la siguiente sentencia:

```
MiObjeto1 := MiObjeto2;
```

Esta sentencia copiará los valores de todos los campos de MiObjeto2 en los campos correspondientes de MiObjeto1. Un programa en Turbo Pascal puede acceder a los campos dentro de un registro; para ello se utiliza la notación punto para cualificar el nombre de la variable. Por ejemplo, se permite escribir una sentencia tal como

```
MiObjeto1.Campo1 := 1;
```

Esta sentencia producirá que el contenido de Campo1 en MiObjeto1 se reemplace con el valor 1.

EJEMPLO 20.1

Declarar un objeto Estudiante que contiene los campos Nombre, Nota1 y Nota2 de dos asignaturas y una función que calcule la media de ambas notas, así como un procedimiento que informe de dicha media.

```
type
  Estudiante = Object
    Nombre : string;
    Nota1 : integer;
    Nota2 : integer;
    procedure Iniciar (nm:string; S1, S2:integer);
    function Media : real;
    procedure Informar;
  end;
```

Una variable objeto MiEstudiante1, que es del tipo Estudiante, se declara como

```
var MiEstudiante1, MiEstudiante2 : Estudiante;
```

EJERCICIO 20.1

Un programa que hace uso completo de la clase Estudiante:

```
program CalcularNotas;
type
  {objeto declarado}
  Estudiante = Object
    Nombre : string;
    Nota1 : integer;
    Nota2 : integer;
    procedure Iniciar (nm:string, S1, S2:integer);
```

```

function Media : real;
procedure Informar;
end;

procedure Estudiante.Iniciar (nm:string; S1, S2:integer);
begin
  Nombre := nm;
  Notal := S1;
  Nota2 := S2;
end;

function Estudiante.Media : real; {método definido}
begin
  Media := (Notal + Nota2)/2
end;

procedure Estudiante.Informar;
begin
  WriteLn (nombre, 'calculo de la media' ,Media:5:1)
end;

var Estd1, Estd2 : Estudiante; {variables declaradas}
begin {cuerpo del programa}
  Estd1.Iniciar ('Mortimer', 8, 7);
  Estd1.Informar;
  Estd2.Iniciar ('Mackoy', 9, 5);
  Estd2.Informar
end.

```

Al ejecutar el programa la salida será

```

Mortimer calculo de la media 7.5
Mackoy calculo de la media 7.0

```

Una variable de un tipo objeto se conoce como una *instancia*. Las instancias son los objetos Estd1 y Estd2. La llamada al procedimiento Estd1.Informar pasa el mensaje Informar al objeto Estd1.

EJEMPLO 20.2

Consideremos un registro denominado *NaveEspacial* que se desea definir para diseñar un programa de juegos.

```

type
  NaveEspacial = record
    Longitud,
    Misiles,
    Velocidad,
    Altitud : Integer;
    Activar : Boolean;
  end;
var
  Icaro, Neptuno : NaveEspacial;

```

El funcionamiento o comportamiento del registro se define con procedimientos y funciones que realizan las tareas típicas de una NaveEspacial: despegar, acelerar, etc.

```
procedure Despegar;
begin
  ...
end;

procedure Acelerar;
begin
  ...
end;

procedure Frenar;
begin
  ...
end;

procedure Aterrizar;
begin
  ...
end;
```

El objeto NaveEspacial correspondiente al registro de igual nombre se representa con la sintaxis siguiente:

```
type
  NaveEspacial = object
    Longitud,
    Misiles,
    Velocidad,
    Altitud : Integer;
    Activar : Boolean;
    procedure Despegar;
    procedure Aterrizar;
    procedure Acelerar;
    procedure Frenar;
  end;
```

¿Cómo se declaran los procedimientos y funciones en POO? La declaración de los subprogramas se realiza con *métodos* que son rutinas asociadas con un tipo objeto particular y se consideran como una parte de la definición de un tipo objeto. En el apartado 20.5 se estudiarán con más detalle los métodos.

Para poder utilizar los objetos es preciso asignar un nombre al objeto en la sección de declaración como variables estáticas o como punteros asignados a las pilas dinámicas (se verá más adelante).

```
var
  Nave : NaveEspacial;
```

Acceso a los campos

Se puede acceder a los campos de datos del objeto tal como se accede a los campos de un registro ordinario, bien mediante la sentencia **with** o mediante el signo punto.

```

Nave.Longitud := 425;
with Nave do
begin
  Misiles := 12;
  Velocidad := 30;
  .
  .
end;

```

EJEMPLO 20.3

El objeto NaveEspacial con las variables y procedimientos definidos se declaran en la sección Type y en la sección en la que se definen las rutinas ordinarias.

```

type
  naveEspacial = object
    Longitud,
    Misiles,
    Velocidad,
    Altitud : Integer;
    Activar : Boolean;
    procedure Iniciar;
    procedure Despegar;
    procedure Acelerar;
    procedure Frenar;
    procedure Aterrizar
  end;

```

La declaración de los procedimientos en el cuerpo del programa se realiza con el nombre del objeto y una extensión separada con un punto con el nombre del procedimiento

```

procedure NaveEspacial.Iniciar;
begin
  Activar := false;
  Longitud := 254;
  Misiles := 7
end;

procedure Naveespacial.Despegar;
begin
  Activar := true
end;

```

y así sucesivamente. Una vez que un objeto ha sido definido se pueden declarar variables utilizando el nombre del objeto.

```

var
  A, B : NaveEspacial;

```

En los programas se puede escribir el segmento de código

```

with B do
begin
  Iniciar;
  Despegar;
  Acelerar;
  Aterrizar
end;

```

o bien acceder a determinados campos con

```
B.Activar := true;
```

20.1.1. Encapsulamiento

El *encapsulamiento* consiste en la creación de objetos que funcionan como unidades completas. La combinación de características y métodos (código y datos) dentro del mismo tipo objeto se denomina encapsulamiento. En el siguiente ejemplo los cuatro procedimientos están encapsulados dentro de *Libro*.

```

type
  Cadl = string [80];
  Libro = object
    Autor, Titulo, Codigo : Cadl;
    Precio : Integer;
    {métodos asociados a Libro}
    procedure Iniciar (A, T, C : Cadl; Pr : Integer);
    procedure Escribir;
    procedure Leer;
    procedure Listar
  end;  {Libro}

```

El programador nunca necesita acceder directamente a los campos dato de un objeto. En el ejemplo anterior, los objetos listados contienen sólo cuatro campos de datos: *Autor*, *Titulo*, *Codigo* y *Precio*. Para facilitar el acceso a estos campos, el objeto define cuatro procedimientos, cada uno de los cuales puede informar o alterar el valor de un campo.

20.2. SECCIONES PÚBLICA Y PRIVADA

Las versiones 5 y 6.0 permitían declarar secciones privadas que servían para restringir el acceso a cualquiera de los campos de datos y métodos; esto se conseguía definiendo una sección privada (mediante la cláusula *private*). Posteriormente, la versión 7.0 incorporó la posibilidad de declarar explícitamente la sección pública con la palabra reservada *public*. Cuando no se indica expresamente la sección *private*, toda la sección correspondiente es pública, que son todas las tratadas anteriormente.

20.2.1. Declaración de una base

```
type NombreClase = object
  {sección pública}
  public
    campos datos públicos
    métodos públicos
  {sección privada}
  private
    campos dato privados
    métodos privados
end;  {objeto}
```

Los campos y métodos privados son accesibles sólo dentro de la unidad o programa que contiene la definición del tipo de objeto. Normalmente y con el objeto de conseguir la ocultación de datos, se debe recurrir a la sección privada. Después de la primera sección pública (sin etiquetar) y la sección privada (opcional) introducida con el identificador `private`, se puede incluir otra sección de componentes públicos. Para hacer esta operación, es preciso insertar la palabra reservada o cláusula `public` y a continuación los componentes públicos. Después de eso, se puede insertar el identificador `private` y más componentes privados, y a continuación insertar el identificador `public` y más componentes públicos. Dentro de cada sección pública o privada, todos los campos dato deben preceder a todos los campos método, tal como se describe en las secciones públicas y privadas. Por ejemplo, el siguiente tipo objeto tiene tres secciones públicas y dos secciones privadas:

```
type
  Muestra = object
    procedure darNombre (NuevoNombre: string);
    function ObtenerNombre : string;
  private:
    Nombre : string;
  public:
    Direccion : string;
    procedure VerDireccion;
  private:
    Altura : integer;
    peso : integer;
  public:
    function CalcularPeso (valor:integer):boolean;
  end;
```

Regla

Normalmente un tipo objeto tiene sus campos datos privados y todos o casi todos sus métodos públicos, constituyendo la interfaz de objeto.

20.2.2. Declaración de un objeto

<i>Sección pública</i>	Contiene los campos dato y los métodos disponibles al resto de los módulos.
<i>Sección privada</i>	Contiene campos y métodos que sólo están disponibles para los subprogramas que contienen la definición del tipo de objetos. Se recomienda que los campos sean privados.

EJEMPLO 20.3.

Declaración de algunos tipos objetos

```

1. type Edad = object
            procedure Iniciar;
            procedure Avanzar (Cantidad:word);
            function CalcularAnyos:word;
            procedure Imprimir;
            private
            Meses:word;
            end;
2. type TipoCirculo = object
            procedure Iniciar (R:real);           {Inicia círculo y radio}
            function Radio : real;                {Devuelve el radio}
            function Diametro:real;              {Devuelve el diámetro}
            function Area:real;                 {Devuelve el área}
            function Circunferencia:real;        {Devuelve la circunferencia}
            private
            ElRadio:real;
            end; {objeto}

```

Una vez declarado el tipo de objeto `TipoCirculo` se puede declarar una variable objeto

```
var MiCirculo : TipoCirculo;
```

que se llama una *instancia* del tipo `TipoCirculo`. Recuerde que las solicitudes a un objeto se llaman *mensajes* y son simplemente llamadas a procedimientos o funciones. Por consiguiente, un objeto responde a un mensaje actuando sobre sus datos. Asimismo recuerde también que los métodos se han de implementar posteriormente en el programa o, si el objeto es parte de una unidad, en la sección de implementación de la unidad.

20.2.3. Implementación de los métodos

Cuando se implementa un método su nombre se cualifica con el tipo de dato del objeto. Así, en el caso del objeto `TipoCirculo` el método `Diametro` se implementa de la forma siguiente:

```
function TipoCirculo.Diametro : real ;
begin
  Diametro := 2.0 * ElRadio
end;
```

Observe que el nombre de la función es `TipoCirculo.Diametro` y no simplemente `Diametro`. Dentro del cuerpo del subprograma que define un método del objeto, se pueden referenciar los campos dato del objeto y otros métodos sin tal cualificación. Sin embargo, cuando se invoca un método desde el exterior de la definición del objeto, esto es, de otras partes del programa, se debe cualificar el nombre del método con una variable objeto. Por ejemplo, se puede escribir

```
MiCirculo.Iniciar (4.25);
WriteLn (MiCirculo.Diametro);
```

para iniciar un objeto de radio 4.25 y visualizar su diámetro; se puede utilizar, al igual que con registros, la sentencia `with`, tal como

```
with MiCirculo do
begin
  Iniciar (4.225);
  WriteLn (Diametro)
end;
```

20.3. DEFINICIÓN DE OBJETOS MEDIANTE UNIDADES

La definición de un objeto dentro de una o más unidades proporciona normalmente un medio adecuado para utilizar objetos en otros programas. Cuando se define un objeto dentro de una unidad, la sección privada del objeto está oculta al programa que utiliza la unidad.

EJEMPLO 20.4.

La siguiente unidad contiene la definición del objeto `TipoCirculo` examinado anteriormente.

```
unit UnCirculo;
interface
type  TipoCirculo = object
    procedure Iniciar (R: real);
    function Radio   : real;
    function Diametro : real;
    function Area    : real;
    function Circunferencia : real;
    procedure VerEstadisticas; {estadísticas del círculo}
    private
      Elradio : real;
end;
```

```

implementation

procedure TipoCirculo.Iniciar (R : real);
begin
  ElRadio := R
end;  {TipoCirculo.Iniciar}

function TipoCirculo.Radio : real;
begin
  Radio := ElRadio
end;  {TipoCirculo.Radio}

function TipoCirculo.Diametro : real;
begin
  Diametro := 2.0 * ElRadio
end;  {TipoCirculo.Diametro}

function TipopCirculo.Area : real;
begin
  Area := Pi * sqr (Elradio)
end;  {TipopCirculo.Area}

function TipoCirculo.Circunferencia : real;
begin
  Circuferencia := Pi * Diametro
end;  {TipoCirculo.Circunferencia}

procedure TipoCirculo.VerEstadisticas;
begin
  WriteLn;
  WriteLn ('Radio =', Radio:6:1);
  WriteLn ('Diámetro = ', Diametro:6:1);
  WriteLn ('Circunferencia = ', Circunferencia:12:1);
  WriteLn ('Área = ', Area:12:1);
end;  {TipoCirculo.VerEstadisticas}

end.  {unidad}

```

20.3.1. Uso de un tipo objeto

El siguiente programa muestra cómo utilizar una unidad

```

program DemostrarUsoUnidad;

uses Circulo;

var MiCirculo : TipoCirculo;

begin  {programa principal}
{establecer radio}
  MiCirculo.Iniciar (6.25);
{visualizar radio, diámetro, circunferencia, área}
  MiCirculo.VerEstadisticas;

end.

```

Dado que la definición del tipo objeto está dentro de la unidad que utiliza este programa, se puede invocar cualquier método (todos son públicos) del objeto dentro del programa, pero no se puede acceder al campo dato privado del objeto ElRadio.

Reglas de diseño de un objeto

Aunque no son rigurosas las reglas de diseño, sí que es recomendable seguir algunos criterios a la hora de construir sus tipo objetos. Estas reglas son:

1. Los campos públicos preceden a los campos privados y dentro de cada uno de estos dos grupos, los campos dato preceden a los métodos.
2. El orden de los campos debe ser:
 - a) Todos los campos datos públicos.
 - b) Todas las cabeceras de métodos públicos.
 - c) El identificador (cláusula) **private**.
 - d) Todos los campos dato privados.
 - e) Todos los métodos privados.

No es usual que un tipo objeto tenga todas estas partes, pero está permitido. Normalmente un objeto tendrá todos o la mayoría de sus campos dato privados y todos o la mayoría de sus métodos públicos.

20.4. LA HERENCIA

La *herencia* es una propiedad (mecanismo) que permite a un objeto heredar propiedades de otra clase de objetos. La herencia permite a un objeto contener sus propios procedimientos o funciones y heredar los mismos de otros objetos.

Regla

Sólo tipos objetos pueden heredar características de otros tipos objeto. La herencia no es posible para otras estructuras de datos (tales como registros).

EJEMPLO 20.5.

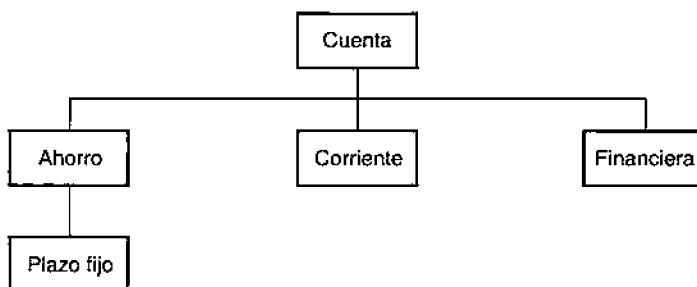
Se desea diseñar una aplicación bancaria teniendo presente los diferentes tipos de cuentas: Ahorro, Corriente, Financiera, Plazo Fijo. Describir las posibles relaciones de herencia.

La cuenta de ahorro a *Plazo Fijo* son casos especiales de cuentas de ahorro. Cada tipo de cuenta contiene un nombre de cliente, un número de cuenta, un saldo y una tasa

de interés. Las cuentas corrientes y financieras permiten extender cheques mientras que las cuentas de ahorro no lo permiten. Las cuentas de ahorro y corriente son subclases de Cuenta y las cuentas de Plazo Fijo son una subclase de las cuentas de ahorro.

Definición de tipos ascendientes/descendientes

La propiedad *herencia* permitirá definir tipos objetos *descendientes* o *ascendientes* (antepasados) de un tipo objeto dado. La herencia hace las tareas de programación más fáciles, ya que se pueden crear sus objetos de modo creciente. Es decir, se puede definir un tipo general de objeto y se utiliza como una parte de objetos específicos sin necesidad de tener que declarar todos los campos individuales nuevamente.



Para definir un tipo objeto descendiente, basta con incluir el nombre del tipo ascendiente dentro de un paréntesis después de la palabra reservada `object`.

La herencia en POO es similar a la técnica de registros anidados en Turbo Pascal. Permite que un nuevo objeto herede las características de un antepasado.

La propiedad de la herencia hace las tareas de programación mucho más fáciles y más flexibles. No se piensa escribir cada una de las características explícitamente para cada elemento, ya que los elementos pueden heredar características de otros.

Por ejemplo, si se trata de representar información sobre animales, se puede diferenciar entre mamíferos, pájaros, insectos, etc., que compartirán características comunes de animales. Los pájaros son descendientes del objeto Animales, y, por el contrario, Animales es ascendiente o antepasado de pájaros. De igual modo, Águilas es un descendiente de Pájaros y en consecuencia de Animales.

Para definir un tipo objeto que hereda de un ascendiente se debe incluir el nombre del tipo ascendiente dentro de un paréntesis y después la palabra reservada `object`.

`nombre tipo descendiente = object (nombre ascendiente)`

Cuando una clase se deriva de otra clase se le denomina *subclase* y a la clase de la que se deriva se la denomina *clase base* o *superclase*.

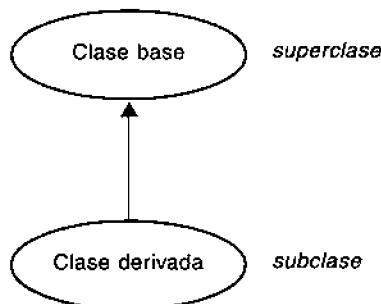


Figura 20.1. Herencia simple.

Una subclase hereda todas las variables de instancias y métodos de la superclase y a continuación se definen los nuevos. La declaración de una subclase difiere de la de una clase sólo en una cosa, el nombre de la clase base aparece entre paréntesis después de la palabra reservada *object*.

Declaración subclase

```
type NombreClase2 = object (NombreClase1)

    nuevaVarInst1 : TipoNuevaVar1;
    nuevaVarInst2 : TipoNuevavar2;
    ...
    procedure NuevoNombrePro1;
    procedure NuevoNombreProc2;
    ...
    function NuevoNombreFunc1;
    function NuevoNombreFunc2;
    ...
end;
```

*Nuevas variables
de instancia*

Nuevos métodos

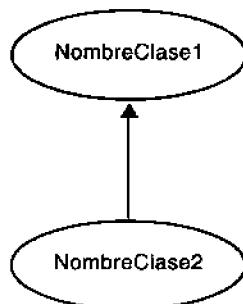


Figura 20.2. Duración de NombreClase1.

EJEMPLO 20.6

Declarar la jerarquía de las clases siguientes:

```

type
  TipoClase = string [20];
  CadenaNombre = string [30];
  Persona = object
    Nombre : CadenaNombre;
    procedure CambiarNombre (NuevoNombre : CadenaNombre);
  end; {persona}
  Estudiante = object (persona)
    Clase : TipoClase;
    Media : real;
    procedure ImprimirRegistro;
  end;

procedure Persona.CambiarNombre (NuevoNombre:CadenaNombre);
begin {Persona.CambiarNombre}
  Nombre := NuevoNombre;
end; {Persona.CambiarNombre}

procedure Estudiante.ImprimirRegistro;
begin
  {acciones}
end;

```

Una posible declaración de variables de las clases Persona y Estudiante

```

var
  P1, P2 : Persona;
  E1, E2 : Estudiante;

```

Así, una instancia E1 declarada del tipo Estudiante se puede manipular del modo siguiente:

```

E1.Nombre := 'Pepe Mackoy';
WriteLn (E1.Nombre);
E1.CambiarNombre ('Luis Carrigan');
WriteLn (E1.Nombre);

```

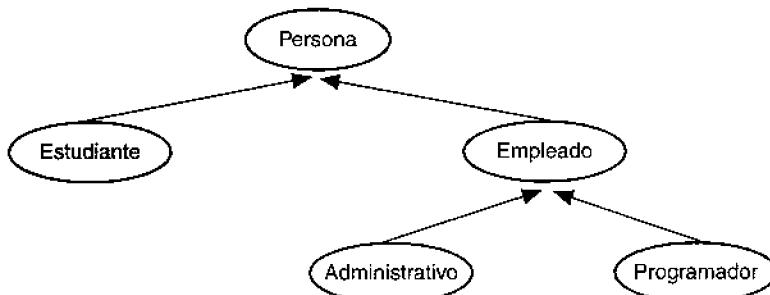


Figura 20.3. Jerarquía de clases.

El tipo estudiante tiene más campos dato que el tipo Persona. Las instancias E1 y E2 tienen los campos de datos llamados Clase y Media, así como el campo dato llamado Nombre. El tipo hereda el método CambiarNombre de la clase Persona, pero puede también tener otros métodos que se aplican sólo al tipo Estudiante; tendrá el método ImprimirRegistro además del método CambiarNombre que se hereda del tipo Persona.

EJEMPLO 20.7

Tipos objetos relacionados entre sí por la propiedad de la herencia.

```
type Coordenadas = object {Coordenadas ascendiente}
    X, Y : integer
    end;
Pixel = object (Coordenadas) {Pixel, descendente}
    Visible : Boolean
    end;
```

En este ejemplo se pueden definir descendientes del tipo Pixel, y a su vez descendientes del tipo descendente de Pixel, y así sucesivamente. Pixel es descendiente de Coordenadas. A la inversa Coordenadas es antepasado inmediato de Pixel.

Un tipo objeto puede tener cualquier número de descendientes inmediatos, pero sólo un antepasado inmediato.

EJEMPLO 20.8

Definición de un tipo objeto Escrito con los campos Autor, Titulo, Código, Año de publicación, Número de páginas y de los objetos descendientes Libro, Artículo (de periódico) y Folleto, cuya estructura se muestra en la Figura 20.4.

```
program Demo;
type
  Cad80   = string [80];
  Cad10   = string [10];
  Escrito = object
    Autor, Titulo, Código : Cad80;
    Anno, Páginas        : Integer; {Escrito}
    end;
  Libro   = object (Escrito)
    Editorial : Cad80;
    Precio    : real
    end;
  Artículo = object (Escrito)
    Periódico, Sección : Cad80;
    Fecha            : Cad10
    end; {Artículo}
```

```

Folleto = object (Escrito)
    Tema : Cad80
    end; {Folleto}
var
    MiEsc      : Escrito;
    MiLibro    : Libro;
    MiArticulo : Articulo;
    MiFolleto  : Folleto;
begin
    {codigo}
    ...
end.

```

En el objeto **Articulo**, además de todos los campos del objeto **Escrito**, se encuentran definidos los campos **Periodico**, **Seccion** y **Fecha**. Aunque los campos nunca se mencionan explícitamente en la definición de **Articulo**, este tipo tendrá campos de cadena **Cad80** y **Cad10**, además de los propios del tipo **Escrito**.

Los campos heredados se representan todos al mismo nivel. Por ejemplo, si **MiLibro** es de tipo **Libro**, los siguientes campos son válidos:

MiLibro.Editorial
MiLibro.Precio

MiLibro.Autor
MiLibro.Titulo

MiLibro.Codigo
MiLibro.Paginas

Reglas

Cuando se especifica que un tipo objeto es un descendiente de otro tipo, la clase descendiente hereda todas las características de la clase ascendiente, así como cualquier clase heredada por el tipo ascendiente o sus antecesores.

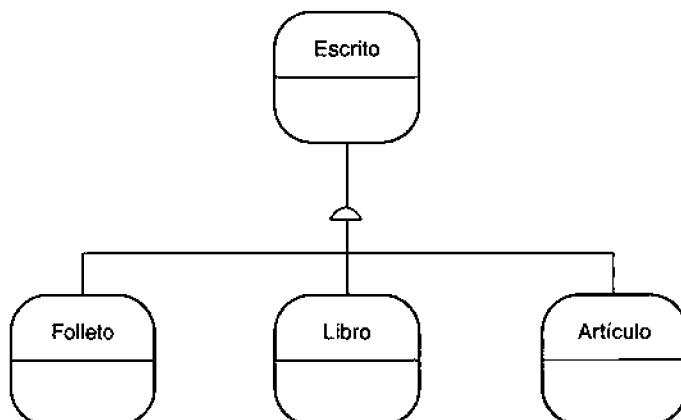


Figura 20.4. Descendiente del objeto Escrito.

Se puede transportar la herencia a otros niveles de descendientes. Por ejemplo, se puede definir **Tecnico** como un objeto derivado de **Libro** (por ejemplo, libro técnico).

{Definir un tipo objeto que herede las características de Libro que a su vez hereda características de Escrito. Una variable Técnico tendrá ocho campos, dos campos de Libro, cinco de Escrito y un campo más llamado especialidad}

```
Type Tecnico = object (Libro)
    especialidad : Cad80;
  end;
var
  LiTec : Tecnico;
```

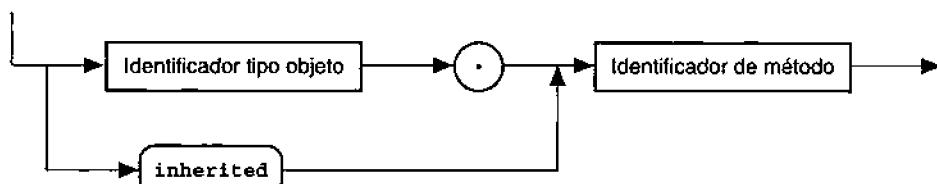
Los siguientes campos son válidos:

```
LiTec.Autor;
LiTec.Titulo;
LiTec.Paginas;
```

20.4.1. Herencia con inherited

Esta nueva palabra reservada se puede utilizar para representar al ascendiente. **Inherited**, se utiliza en la llamada de un método heredado de un objeto ascendiente (*padre*) a partir de un objeto descendiente (*hijo*). En las versiones anteriores era necesario especificar el nombre preciso del padre para activar uno de sus métodos. Con la nueva versión 7.0, basta situar la palabra clave **inherited** delante de la llamada al método. Esta característica evitará las modificaciones ulteriores del código de una aplicación si la jerarquía de una biblioteca de objetos se modifica.

En las versiones 5.X/6.0/7.0 se realizará la activación de un método especificado mediante el designador de método cualificado. Este tipo de llamada se conoce como una *activación de método cualificado*. Su diagrama de sintaxis es:



Así, por ejemplo, imaginemos que se tiene un tipo **Tdibujo**, del cual se deriva **TDibujo3D**. Normalmente para acceder al método **Dibujar** del ascendiente se debe utilizar el método cualificado y en consecuencia el formato **TDibujo.Dibujar**. Sin embargo, ahora con la versión 7.0 como **inherited** es opcional se puede utilizar también **inherited Dibujar**. ¿Cuál es la ventaja? Si en una etapa posterior la jerarquía de objetos se modifica, no será necesario modificar el código de la aplicación. Así, si el método **Dibujar** se mueve a un objeto llamado **TGraficos** en una etapa posterior, no tiene que cambiar las funciones del descendiente que cuenta con ellos.

Cláusula inherited

Cuando se *inicializa* un objeto derivado o descendiente (*subclase*), el primer paso a dar es la llamada al método *Iniciar* de su ascendiente inmediato (*superclase*). Esta operación ayuda a evitar código duplicado ahorrando al método *Iniciar* del objeto descendiente la *inicialización* de los campos de datos definidos en el objeto ascendiente. La palabra reservada *inherited* permite *llamar* al método *Iniciar* del ascendiente sin conocer el nombre del ascendiente. Es decir, suprimiendo la jerarquía de tipos del objeto persona-Estudiante.

```
type
  Edades = 0..120;
  Persona = object
    Apellido : string [25];
    Nombre   : string [20];
    Edad     : Edades;
    procedure Iniciar;
    procedure PonerNombre (NombreNuevo, ApellidoNuevo:string);
  end;
  Estudiante = object (Persona)
    Grupo   : string [10]
    Profesor : string[30];
    Curso   : integer;
    procedure Iniciar;
  end;
```

Los procedimientos *Iniciar* pueden ser éstos:

```
procedure Persona.Iniciar;
begin
  Apellido := '';
  Nombre   := '';
  Edad     := 0;
end;
```

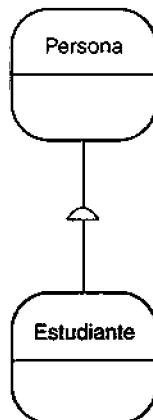


Figura 20.5. Jerarquía Persona-Estudiante.

```

procedure Estudiante.Iniciar;
begin
  inherited Iniciar;
  Grupo    := '';
  Profesor := '';
  Curso    := 0;
end;

```

En general, `inherited` permite referirse a métodos del ascendiente (*su superclass*) sin utilizar el nombre del tipo ascendiente como un cualificador. Otro ejemplo puede ser la jerarquía **Figura-Círculo**.

El método `Circulo.Visualizar` contiene la llamada al método `Figura.Visualizar` para referenciar al método `Visualizar` definido en el tipo objeto `Figura` ascendiente de `Círculo`. Utilizando `inherited` la llamada comparable al método correspondiente será:

```
inherited Visualizar
```

La palabra reservada `inherited` no se puede utilizar en métodos de objetos que no tengan un tipo ascendiente o superclass.

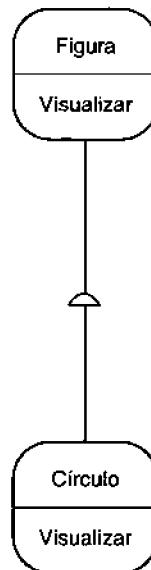


Figura 20.6. Jerarquía de tipos de objetos **Figura-Círculo**.

20.5. LOS MÉTODOS

Una característica importante en programación orientada a objetos es la capacidad para especificar métodos cuando se define un objeto. Un *método* es una rutina asociada con un tipo objeto especificado. Esta rutina está disponible para cualquier objeto y objetos descendientes. En esencia es una parte de una definición del tipo objetos.

Un método es un procedimiento o función que está asociado a un tipo de objeto. No se pueden asociar métodos con otros tipos de datos.

Los procedimientos y funciones declaradas dentro de un objeto se conocen como métodos.

Los métodos se definen en dos partes en el programa: las cabeceras de los procedimientos y funciones se especifican en la declaración del tipo objeto, y la implementación (esto es, cabecera más cuerpo) de estos subprogramas aparece fuera de la definición de tipo.

20.6. OBJETOS DINÁMICOS

Un programa puede tener variables dinámicas de un tipo objeto y punteros a tales instancias de objetos dinámicos. Turbo Pascal proporciona diversas características especiales que se aplican a objetos dinámicos. Se pueden asignar dinámicamente campos de un objeto, de hecho se pueden asignar objetos completos dinámicamente.

20.6.1. Asignación de objetos dinámicos

Cuando a un objeto se le asigna memoria en tiempo de compilación y permanece asignada durante la ejecución del programa es un ejemplo de *objeto asignado estáticamente*. En contraste, la memoria de un *objeto asignado dinámicamente* se asigna durante la ejecución del programa y sólo permanece asignada mientras se desee.

Los objetos dinámicos se asignan mediante el uso del identificador estándar `new`.

Cuando se crea una variable dinámica con una llamada a `new`, vendrá seguida por una llamada a un constructor. Para facilitar el proceso, la operación `new` ha sido ampliada de modo que pueda ser llamada con dos parámetros: una variable puntero como primer parámetro y una llamada al constructor como segundo parámetro.

Formato

```

type
  tipo_objeto = object
    nombre:string;
    constructor nombre_constructor;
  end;
  ptr_tipo_objeto = ^tipo_objeto;
  constructor tipo_objeto.nombre_constructor;
  begin
    {acciones}
  end;
var
  varp : ptr_tipo_objeto;
begin
  new (varp);
  varp^.nombre_constructor;
  {o bien}
  new (varp, nombre_constructor);
  {o bien}
  varp := new (ptr_tipo_objeto, nombre_constructor);

```

Suponga que P es una variable puntero, Iniciar es un constructor de variables dinámicas de su tipo de dominio. Una nueva variable dinámica se puede crear e inicializar con las siguientes dos llamadas:

```

new(P);
P^.Iniciar ('Pepe Mackoy');

```

que son equivalentes a la siguiente forma ampliada de new:

```

new (P, Iniciar ('PepeMackoy'));

```

EJEMPLO 20.9

Asignar instancias dinámicamente de TipoCirculo mediante sentencias

```

type PtrTipoCirculo = ^TipoCirculo;
var PtrMiCirculo : PtrTipoCirculo;

```

Crear a continuación una instancia del objeto dinámico utilizando el procedimiento new.

```

new (PtrMiCirculo);

```

o la función new

```

PtrMiCirculo := new (PtrTipoCirculo);

```

Si el tipo objeto contiene métodos virtuales, como es el caso de `TipoCirculo`, se debe llamar a su constructor:

```
PtrMiCirculo^.Iniciar(5.5);
```

o bien

```
PtrMiCirculo := new (PtrTipoCirculo, Iniciar(5.5));
```

20.6.2. Liberación de memoria y destructores

Cuando se desea liberar la memoria asignada a objetos dinámicos se ha de utilizar el procedimiento `dispose`.

Formato

```
dispose (varp);
dispose (varp, destructor);
```

EJEMPLO 20.10

```
dispose(x);
dispose(x, Done);
```

Los objetos asignados dinámicamente que contienen métodos virtuales deben definir un método especial —llamado *destructor*— dentro del objeto. El destructor, que por convenio se suele nombrar por `Done` (*Terminado*), realiza todas las tareas necesarias para liberar el objeto.

Para liberar un objeto dinámico que tiene métodos virtuales, se utiliza el formato ampliado del procedimiento `dispose`:

```
dispose (PtrMiCirculo, Done);
```

Si el objeto contiene campos asignados dinámicamente, el destructor del objeto debe llamar también a `dispose`.

Un objeto puede heredar su destructor desde un objeto ascendiente. Un objeto puede tener varios destructores para tratar diversas situaciones, pero se llama sólo a un destructor por instancia del objeto. Los destructores pueden ser o bien estáticos o virtuales, aunque son recomendables los destructores virtuales para asegurar que futuros descendientes del objeto puedan liberarse por sí mismos correctamente. Los destructores son necesarios sólo para objetos asignados dinámicamente u objetos con campos asignados dinámicamente.

PROBLEMA 20.1

La unidad PilaPrueba es una pila implementada utilizando objetos dinámicos. La pila se implementa como una lista enlazada. El puntero Cima apunta a la cabeza de la lista enlazada que sirve también como elemento superior de la pila. Las operaciones meter y sacar se realizan insertando y borrando nodos en la cabeza (cima) de la lista.

```
{$R+}
unit PilaPr; {PilaPrueba}
{pila genérica}
interface
  type IItemPtr = ^Item;
  Item = object
    constructor Iniciar;
    destructor Terminado; virtual;
    procedure Imprimir; virtual;
  private
    Enlace : ItemPtr;
  end;

  Pila = object
    procedure Inicializar;
    procedure Meter (Elem : ItemPtr);
    procedure Sacar (var ElemSup : ItemPtr);
    function vacia : boolean;
    procedure LiberarPila;
  private
    Cima : ItemPtr;
  end;
implementation
  constructor Item.Iniciar;
  begin
    Enlace := nil;
  end;

  destructor Item.Terminado;
  begin
  end;

  procedure Item.Imprimir;
  begin
  end;

  procedure Pila.Inicializar;
  begin
    Cima := nil;
  end;

  procedure Pila.Meter (Elem : ItemPtr);
  begin
    Elem ^.Enlace := Cima;
    Cima := Elem;
  end;
```

```

procedure Pila.Sacar (var ElemSup: ItemPtr);
begin
  ElemSup := Cima;
  Cima := Cima^.Enlace;
end;

function Pila.Vacia : boolean;
begin
  Vacia := (Cima = nil);
end;

procedure Pila.LiberarPila;
var
  Aux : ItemPtr;
begin
  while Cima < > nil do
  begin
    Aux := Cima;
    Cima := Cima^.Enlace;
    dispose (Aux, Terminado);
  end;
end;
end.

```

20.6.3. La cláusula Self

En la definición/declaración de un método existe siempre un parámetro implícito con el identificador **Self**, que corresponde a un parámetro formal variable que posee el tipo objeto. En el bloque del método, **Self** representa la instancia cuyo componente método fue designada para activar el método. Por consiguiente, cualquier cambio realizado a los valores de los campos de **Self** se reflejan en la instancia.

Esta propiedad de **Self** se manifiesta de modo práctico del modo siguiente: los identificadores de campos de un objeto no se pueden redefinir como parámetros formales o identificadores locales en ninguno de los métodos del objeto. Es posible, sin embargo, para los identificadores que no son parte del objeto, tener el mismo nombre que los identificadores de campo del objeto. Para referir a un campo de un objeto que tenga el mismo nombre que otro identificador del mismo ámbito, el identificador de campo del objeto puede ser cualificado por el identificador **Self**. Dentro de un método, **Self** siempre se refiere a la instancia del objeto que hace la llamada del método. Por ejemplo, en el tipo objeto **Figura**

```

type
  Figura = object
    Forma           : string;
    Area, Perimetro : Real;
    constructor ObtenerFigura;
    procedure CalculoArea; virtual;
    procedure CalculoPerimetro; virtual;
    procedure Visualizar; virtual;
  end;

```

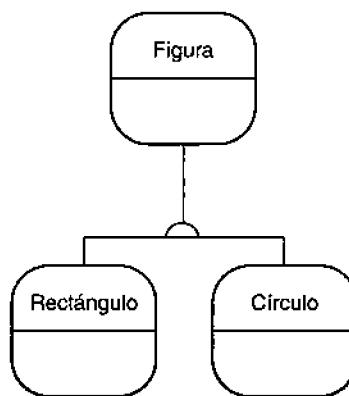
dentro del método Figura.CalcularArea una asignación al campo Area del objeto se puede escribir como:

```
Self.Area := 25.0;
```

PROBLEMA 20.2

Escribir un programa que represente la jerarquía de objetos: Figura, Círculo y Rectángulo.

La jerarquía de estos tres objetos es



objeto Figura	CamposData:	Area Perímetro Forma	tipo real tipo real tipo cadena
	Métodos	CalculoArea CalculoPerímetro Visualizar	superficie figura perímetro figura ver la figura
objeto Círculo	CamposData	Radio	tipo real
	Métodos	CalculoArea CalculoPerímetro Visualizar	
objeto Rectángulo	CamposData	Altura Anchura	tipo real tipo real
	Métodos	CalcularArea CalcularPerímetro Visualizar	
unit FigGeo;			
interface			
type			
Figura = object			
Tipo : String {tipo de figura}			
Area : Real;			
Perímetro : Real;			
constructor ObtenerFigura;			
procedure CalculoArea; virtual;			
procedure CalculoPerímetro; virtual;			
procedure Visualizar; virtual;			
end; {Figura}			

```

Rectangulo = object (Figura)
    Altura, Anchura : Real;
    constructor Iniciar (Alt, Anch: Real);
    procedure CalculoArea; virtual;
    procedure CalculoPerimetro; virtual;
    procedure Visualizar; virtual;
end; {Rectángulo}

Circulo = object (Figura)
    Radio : Real;
    constructor Iniciar (R : Real);
    procedure CalculoArea; virtual;
    procedure CalculoPerimetro; virtual;
    procedure Visualizar; virtual;
end; {Círculo}

implementation
    constructor Figura.ObtenerFigura;
    {Inicializa objetos del tipo figura}
begin
    Write ('Introduzca nombre de la figura:');
    ReadLn (Tipo)
end;

procedure Figura.CalculoArea; {asigna valor cero a Area}
begin
    Area := 0.0
end; {CalculoArea}

procedure Figura.CalculoPerimetro; {asigna valor cero a Perimetro}
begin
    Perimetro := 0.0
end; {CalculoPerimetro}

procedure Figura.Visualizar;
begin
    WriteLn ('El tipo de figura es: ',Tipo);
    WriteLn ('El área es ', Area:4:2);
    WriteLn ('El perímetro es ', Perimetro:4:2);
end;
{implementaciones del tipo objeto Rectangulo}
constructor Rectangulo.Iniciar (Alt, Anch:Real);
{inicializa objetos de tipo rectangulo}
begin
    Tipo := 'Rectángulo';
    Altura := Alt;
    Anchura := Anch
end;
procedure Rectangulo.CalculoArea;
{cálculo del área de objetos rectángulo}
begin
    Area := Altura * Anchura
end;

```

```

procedure Rectangulo.CalcularPerimetro;
{cálculo del perímetro del rectángulo}
begin
    Perimetro := 2 * (Anchura + Altura)
end;

procedure Rectangulo.Visualizar;
{Visualiza los datos del objeto rectángulo}
begin
    Figura.Visualizar;
    WriteLn ('La anchura es:', Anchura:5:2);
    WriteLn ('La altura es:', Altura:6:2);
end;
{implementaciones del tipo objeto Círculo}
constructor Circulo.Iniciar (R:Real);
{inicializa objetos de tipo Circulo}
begin
    Tipo := 'Círculo';
    Radio := R
end; {Iniciar}

procedure Circulo.CalcularArea;
{Cálculo de la superficie del círculo}
begin
    Area := 3.1416 * Radio * Radio
end; {CalculoArea}

procedure Circulo.CalcularPerimetro;
{cálculo de la longitud de la circunferencia}
begin
    Perimetro := 2 * 3.1416 * Radio
end; {CalculoArea}

procedure Circulo.Visualizar;
begin
    Figura.Visualizar;
    WriteLn ('El radio es:', Radio:6:2)
end; {Visualizar}
end. {Final de Unidad FigurasGeom}

```

El método `Visualizar` podría implementarse de forma que comenzara por llamar a los métodos `CalcularArea` y `CalcularPerimetro` y, a continuación, presentara los resultados obtenidos. Aunque `CalcularArea` y `CalcularPerimetro` se han redefinido y son distintos en cada objeto, es posible definir `Visualizar` una sola vez, como se muestra en el siguiente programa, constituyendo un ejemplo de polimorfismo:

```

unit FigGeo;

interface
type
    figura = object
        tipo      : string;
        area     : real;
        perimetro : real;

```

```

constructor ObtenerFigura;
procedure CalculoArea; virtual;
procedure CalculoPerimetro; virtual;
procedure visualizar;
end;
rectangulo = object (Figura)
  altura, anchura : real;
  constructor iniciar (alt, anch : real);
  procedure CalculoArea; virtual;
  procedure CalculoPerimetro; virtual;
end;
circulo = object (Figura)
  radio : real;
  constructor iniciar (r : real);
  procedure CalculoArea; virtual;
  procedure CalculoPerimetro; virtual;
end;

implementation
constructor Figura.ObtenerFigura;
begin
  Write ('Introduzca el nombre de la figura:');
  ReadLn(tipo)
end;
procedure Figura.CalculoArea;
begin
  area := 0.0
end;
procedure Figura.CalculoPerimetro;
begin
  perimetro := 0.0
end;
procedure Figura.Visualizar;
begin
  CalculoArea;
  CalculoPerimetro;
  WriteLn ('El tipo de figura es:',tipo);
  WriteLn ('El Área es:           ',area:4:2);
  WriteLn ('El perimetro es:      ',perimetro:4:2);
end;
constructor Rectangulo.iniciar(alt, anch : real);
begin
  tipo := 'Rectángulo';
  altura := alt;
  anchura := anch
end;
procedure Rectangulo.CalculoArea;
begin
  area := altura * anchura
end;
procedure Rectangulo.CalculoPerimetro;
begin
  perimetro := 2 * (altura + anchura)
end;
constructor Circulo.iniciar (r : real);

```

```

begin
  tipo := 'Circulo';
  radio := r
end;
procedure Circulo.CalcularArea;
begin
  area := 3.141516 * radio * radio
end;
procedure Circulo.CalcularPerimetro;
begin
  perimetro := 2 * 3.141516 * radio
end;
end.

Program llamaobj;
uses FigGeo;
var
  lado1, lado2 : real;
  r             : rectangulo;
  radio         : real;
  c             : circulo;
begin
  ReadLn (lado1, lado2);
  r.iniciar (lado1, lado2);
  r.visualizar;
  ReadLn (radio);
  c.iniciar (radio);
  c.visualizar
end.

```

20.7. POLIMORFISMO

El polimorfismo y la ligadura de tipos dinámica (en tiempo de ejecución) son dos características específicas de la programación orientada a objetos. *Polimorfismo* significa que el mismo operador se puede utilizar con diferentes tipos de objetos que responderán del modo apropiado a ese operador. Por ejemplo, el operador + es polimórfico (se dice también que está *sobrecargado*) en Pascal ya que indica concatenación, suma o unión, en función de los operandos.

El polimorfismo en su acepción más usual se refiere a un método de un objeto que tendrá el mismo nombre que un método de su objeto ascendente. Así, en la jerarquía de tipos objetos de la Figura 20.8, el método Dibujar se define el de la clase Figura pero se *declara* (implementa) en las clases u objetos derivados; dado que el procedimiento Dibujar será distinto según se trata de una figura Circulo, triangulo, Cuadrado o Elipse.

La herencia y la *ligadura tardía o postergada o también dinámica* significa que la decisión de la elección del método adecuado (correspondiente a un objeto) se determina en tiempo de ejecución. La *ligadura dinámica* es la característica que permite posponer la ligadura de un tipo objeto a un identificador de instancia hasta el momento de la ejecución. Esta característica es la que posibilita el hecho del polimorfismo y que ciertos métodos se comporten de modo polimórfico.

El hecho práctico más sobresaliente del polimorfismo reside en el hecho de que con ligadura dinámica se ejecute sólo una versión del método, aunque luego las implementaciones serán diferentes. Para permitir que se pueda dar el polimorfismo es necesario hacer el método correspondiente *virtual* en lugar de un método estático que es el sistema tradicional.

Consideremos el ejemplo anterior de la jerarquía de tipos de objeto Figura. Cada figura puede ser un objeto de un tipo de datos diferente. Por ejemplo, un rectángulo puede constar de altura, anchura y centro, mientras que un círculo puede constar de un centro y un radio. En un programa bien diseñado todos los objetos serán descendientes de un tipo ascendente llamado Figura. Supongamos ahora que se desea dibujar una figura en la pantalla. Cada método de dibujo de una figura de la jerarquía será diferente, pero pueden tener todos ellos el mismo nombre Dibujar. Es decir, los métodos se llamarán en todas las figuras Dibujar. Ahora bien, si R es una instancia de un objeto rectángulo y C es una instancia de un objeto círculo, entonces R.Dibujar y C.Dibujar serán métodos implementados con códigos diferentes. Ahora bien, el tipo ascendente Figura puede tener métodos que se apliquen a todas las figuras. Por ejemplo, puede tener un método denominado Centrar que mueva una figura al centro de la pantalla. De este modo Figura.Centrar puede utilizar el método Dibujar para volver a dibujar la figura en el centro de la pantalla. Con cualquier otra figura el método Dibujar deberá ser implementado en cada una de ellas y el sistema práctico es declarar los métodos virtuales.

Métodos virtuales son los métodos que utiliza Turbo Pascal para proporcionar ligadura dinámica.

Advertencia

En esencia cuando se construye un método virtual, se está indicando al compilador *Yo no conozco cómo está implementado ese método. Espere hasta que se utilice en un programa y a continuación obtenga la implementación de la instancia del objeto.* Esta técnica de esperar hasta el momento de la ejecución para determinar la implementación de un procedimiento se denomina *ligadura dinámica*.

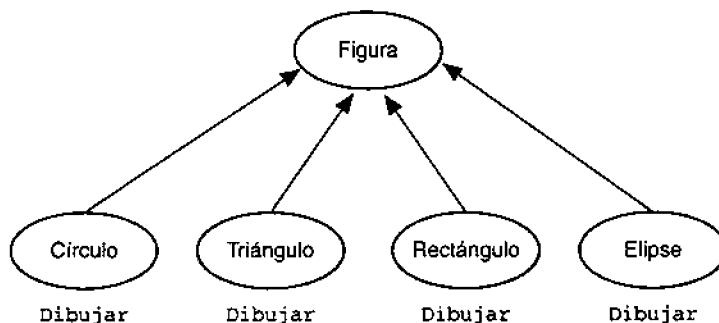


Figura 20.7. Jerarquía de objetos.

Reglas

1. Para indicar que un método es virtual, escriba simplemente la palabra reservada **virtual** después de la cabecera del subprograma en la definición del tipo objeto.
2. Cualquier método de un tipo objeto descendiente que tenga el mismo nombre que un método virtual del ascendiente debe ser también virtual y debe tener los mismos parámetros con el objeto de evitar errores de compilación. De hecho, las cabeceras de todas las implementaciones de un método virtual deben ser idénticas.

La Figura 20.8 ilustra el efecto general de un método virtual.

Ejemplo

*Considerar los tipos objetos **TipoCirculo** y **TipoEsfera** como una extensión o derivación de **TipoCirculo**. Las declaraciones respectivas, considerando el método **Area** virtual son:*

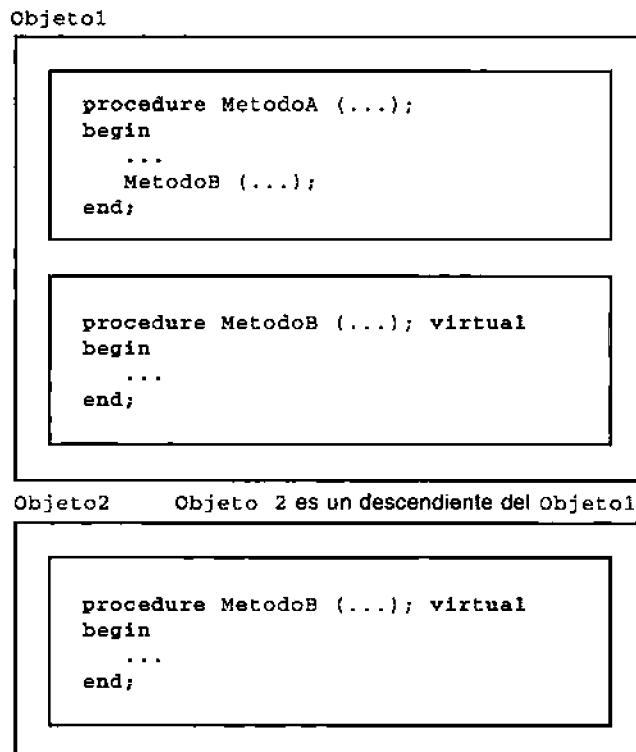


Figura 20.8. Llamada a métodos virtuales.

```

type    TipoCirculo = object
        constructor Iniciar (R : real);
        function radio : real;
        function Diametro : real;
        function Area : real; virtual;
        function Circunferencia : real;
        procedure VerEstadistica;
private
        ElRadio : real;
end;  {objeto}

TipoEsfera = object (TipoCirculo)
{El constructor se hereda}
        function Area : real; virtual;
        function Volumen : real;
end;  {objeto}

```

20.7.1. Constructores y destructores en objetos dinámicos

Los métodos virtuales han de cumplir un requisito. El tipo objeto que contiene un método virtual debe tener un *constructor*, que es un método de iniciación que se debe llamar antes de llamar a un método virtual. Los constructores son simplemente un tipo especial de procedimiento que se representa con la palabra reservada **constructor** en lugar de **procedure**. Cada instancia de un objeto se inicia por una llamada a un constructor antes de que la instancia del objeto se utilice con un método virtual.

Los tipos objetos que contienen métodos virtuales deben definir un constructor.

Obsérvese que un objeto puede heredar un constructor en lugar de definirlo; de igual forma los constructores por sí mismos no pueden ser virtuales. Se debe llamar al constructor por cada instancia de un objeto. Por consiguiente, si **MiCirculo** y **TuCirculo** son instancias de **TipoCirculo** se deberá incluir la sentencia

```

MiCirculo.Iniciar;
TuCirculo.Iniciar;

```

en su programa.

Un objeto que sólo tiene métodos estáticos puede tener un constructor, y cualquier objeto puede tener varios constructores.

Regla

1. El constructor ha de ser llamado antes que se invoque a cualquier método virtual. La llamada a un método virtual sin llamar previamente al constructor puede producir el bloqueo del sistema y el compilador no tendrá forma de compilar el orden en que los métodos son llamados.

2. Cada instancia individual de un objeto debe ser iniciada por una llamada independiente de constructor. No es suficiente iniciar una instancia de un objeto y a continuación asignar esa instancia a instancias adicionales.

¿Qué construyen los constructores? Cada tipo de objeto que define un método virtual tiene una tabla de métodos virtuales **TMV** (*Virtual Method Table, VMT*). Por cada método virtual en el objeto, la TMV contiene un puntero (apuntador) a las instrucciones reales que implementan el método. Una llamada al constructor establece este puntero. Por consiguiente, cuando una instancia de un objeto llama a un método virtual se ejecutan las instrucciones del método por medio de los punteros TMV a las versiones de los métodos virtuales que son apropiados a la instancia del objeto. Cada tipo de objeto sólo tiene un TMV.

Para comprobar el estado de inicialización de los métodos virtuales se ha de utilizar la directiva `{ $R+ }`. Cuando esta directiva está activada, una llamada a un método virtual no iniciado produce un error de comprobación de rango durante la ejecución. Si no se utiliza `$R+`, una llamada a un método no inicializado puede hacer que Turbo Pascal se comporte de modo impredecible.

20.7.2. Anulación de métodos heredados

A veces, un objeto descendiente necesita realizar algún método de un modo diferente a como está definido en el objeto ascendiente. Esta acción es posible mediante la redefinición o anulación del método heredado de su ascendiente. Si una declaración de un método en un descendiente, especifica el mismo identificador del método como una declaración de un método en el ascendiente, entonces la declaración en el descendiente *anula* la declaración del ascendiente. El ámbito de un método anulado se extiende sobre el dominio del descendiente en el que se ha introducido o hasta que el identificador del método es nuevamente anulado.

La anulación de un método estático puede cambiar la cabecera del método cuando lo deseé. En contraste, en una anulación de un método virtual debe corresponderse exactamente el orden, tipos y nombres de los parámetros, así como el tipo del resultado de la función, en su caso. La anulación debe incluir de nuevo una directiva `virtual`.

PROBLEMA 20.3

Diseñar e implementar un tipo objeto Reloj, que contenga al menos los siguientes métodos públicos: poner la hora, recuperar la hora, imprimir el valor de la hora, adelantar el reloj un número determinado de minutos. Una vez diseñado el tipo objeto Reloj definir y declarar otro tipo objeto Relojradio en el que la función Imprimir se redifina (se anule) con una implementación diferente de la clase ascendiente Reloj.

```
type
  TipoHora = 0..23;
  TipoMinuto = 0..59;
  Reloj = object
    procedure Asignar (H:TipoHora; M:TipoMinuto);
    function LeerHora : TipoHora;
```

```

function LeerMinutos : TipoMinuto;
procedure Imprimir;
procedure Adelantar (M : word);
private
  Hora    : TipoHora;
  Minutos : TipoMinuto;
end;

```

El tipo objeto derivado (descendiente) RelojRadio

```

type RelojRadio = object (Reloj)
  function radio : boolean;
end;

```

Se desea anular el procedimiento `Imprimir` que simplemente imprime la hora en la superclase `Reloj`, para que en la clase derivada además de imprimir la hora, active la radio y sintonice una determinada emisora. La anulación del método heredado `Imprimir` se hará con la sintaxis siguiente:

```

type RelojRadio=
  object (Reloj)
    procedure Imprimir; {Anula el método Reloj.Imprimir}
    function SintonizarRadio: boolean; {true si radio encendida}
  end;

{implementación de Imprimir}
procedure RelojRadio.Imprimir;
var
  Numero : integer {minuto real de la lectura del reloj}
const
  DOSPUNTOS = ':';
begin
  {nueva implementación del método}
end;
function RelojRadio. SintonizarRadio;
begin
  ...
end;

```

Otro sistema mejor para anular el método `Reloj.Imprimir` es

```

procedure RelojRadio.Imprimir
begin
  Reloj.Imprimir;
  {restantes sentencias}
end;

```

PROBLEMA 20.4

Escribir un programa que permita hacer uso de las propiedades de herencia y anulación de métodos en una aplicación que visualice información sobre diferentes tipos de animales.

Sea el caso de la jerarquía de animales que contempla la clasificación: vertebrados, mamíferos y artrópodos:

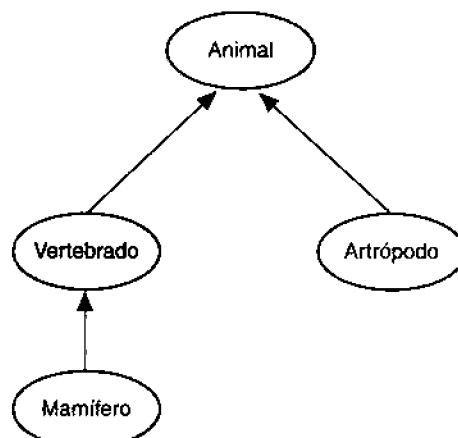
```
program HerenciaAnimal;
{proporciona características y clasificaciones de tipo de animales}
uses Crt;

type Animales = object
    procedure DarMensaje;
    end;
Vertebrado = object (Animales)
    procedure DarMensaje;
    end;
Mamifero = object (Vertebrado)
    procedure DarMensaje;
    end;
Artrropodo = object (Animales)
    procedure DarMensaje;
    end;

var MiAnimal : Animales;
    MiVertebrado : Vertebrado;
    MiMamifero : Mamifero;
    MiArtrropodo : Artrropodo;

procedure Animales.DarMensaje;
begin
    WriteLn ('Animales:');
    WriteLn ('Propiedad:los animales se pueden mover')
end;

procedure Vertebrado.DarMensaje
begin
    WriteLn ('Vertebrados:');
    WriteLn ('Propiedad:tienen vértebras (columna vertebral y cráneo)');
    Animal.DarMensaje
end;
```



```

procedure Mamifero.DarMensaje;
begin
  WriteLn ('Mamíferos:');
  WriteLn ('Propiedad: Animal vertebrado de temperatura constante');
  Vertebrado.DarMensaje
end;

procedure Artropodo.DarMensaje;
begin
  WriteLn ('Artrópodos:');
  WriteLn ('Propiedad: animal invertebrado (insectos y arañas)');
  Animal.DarMensaje
end;

begin
  ClrScr;
  MiAnimal.DarMensaje;
  WriteLn;
  MiVertebrado.DarMensaje;
  WriteLn;
  MiMamifero.DarMensaje;
  WriteLn;
  MiArtropodo.Darmensaje
end.

```

Al ejecutar el programa se visualizaría:

```

Animales:
Propiedad: Los animales se pueden mover

Vertebrados:
Propiedad. Tienen vértebras: (columna vertebral y cráneo)
Animales:
Propiedad: Los animales se pueden mover

Mamíferos:
Propiedad: Animal vertebrado de temperatura constante
Vertebrados:
Propiedad: Tienen vértebras (columna vertebral y cráneo)
Animales:
Propiedad: Los animales se pueden mover

Artrópodos:
Propiedad: Animal invertebrado (insectos y arañas)
Animales:
Propiedad: Los animales se pueden mover

```

20.8. CONSTRUCTORES Y DESTRUCTORES

Los *constructores* y *destructores* son rutinas, o métodos, que se asocian con un objeto particular. Desde el punto de vista sintáctico, un método, tal como un constructor, es similar a un procedimiento (ejecuta alguna tarea relativa a un objeto particular).

Un constructor puede inicializar los campos de una variable objeto específico. Este

constructor puede ser utilizado por otros objetos que se definan como casos especiales del objeto con el cual se asoció el constructor.

El *constructor* advierte al compilador que un método particular se utilizará para establecer una estructura de datos manipulada por un método virtual.

EJEMPLO 20.11

```
type
  Buque = object
    X, Y, Blindaje : integer;
    Flanco        : char;
    constructor IniciarX (NuevaX : integer)
  end;
```

En el objeto Buque, la definición del constructor virtual objeto puede ser similar a ésta, considerando el sufijo virtual en la cabecera del procedimiento.

```
type
  Buque = object
    X, Y, Blindaje : integer;
    Flanco        : char;
    constructor IniciarX (NuevaX : integer);virtual;
  end;
```

Turbo Pascal 5.5 proporciona un tipo especial de método llamado *destructor*, denominado así porque limpia y dispone de objetos asignados dinámicamente. Como con cualquier otro método, se pueden definir destructores múltiples para un solo tipo de objeto.

Un destructor se define con todos los restantes métodos de objetos en la definición de tipos objetos.

```
Punto = object (Coordenadas)
  Visible   : Boolean;
  Siguiente : PuntoPtr;
  constructor Init (IniciarX, IniciarY : integer);
  destructor Terminado; virtual;
  procedure Visualizar; virtual;
  procedure Ocultar; virtual;
  function EsVisible : Boolean
end;
```

Los destructores pueden ser heredados y estáticos o virtuales, aunque es recomendable que sean virtuales de modo que en cada caso el destructor correcto se ejecutará para su tipo objeto.

Otros ejemplos

1. Un objeto Dibujo se declara así:

```
Dibujo = object
  private
    X, Y, Long : integer;
    Nombre     : string;
```

```

public
  constructor Copiar (var F:Dibujo);
  constructor Iniciar (FX, FY, Flong : integer; nombreF:string);
  destructor Done; virtual;
  procedure Visualizar; virtual;
  procedure Editar; virtual;
  ...
private
  procedure VisualizarPuntos(x,y:integer; S:string);
end;

```

2. El objeto Demo, que contiene métodos, datos constructores y destructores, tiene todos los campos con carácter público:

```

Demo = object
  Campo1 : Integer;
  constructor Iniciar;
  destructor Done;
  procedure Metodo;
end;

```

y una variable de tipo demo

```

var
  Instancia:Demo;

```

con esta definición las sentencias siguientes son legales.

```

Instancia.Iniciar;
Instancia.Metodo;
WriteLn (Instancia.Campo1);
Instancia.Done;

```

En muchas ocasiones se desea evitar que los campos dato de un objeto no se puedan modificar con sentencias tales como:

```
Instancia.Campo1 := 4567;
```

En estos casos es preciso declarar privados dichos campos en el objeto Demo.

```

Demo = object
  constructor Iniciar;
  destructor Done;
  private
    Campo1: Integer;
    procedure Metodo;
end;

```

Las declaraciones privadas (Campo1 y Metodo) deben venir al final del objeto. Todas las restantes declaraciones encima de private son públicas. Con esta nueva definición del objeto Demo las cuatro sentencias anteriores siguen siendo válidas, pero las dos sentencias siguientes:

```
WriteLn (Instancia.Campo1);
Instancia.Metodo;
```

se compilarán sólo si las sentencias aparecen en el mismo programa o modelo unidad que declare Demo.

EJEMPLO 20.12

Se puede asignar almacenamiento para una variable objeto e iniciar ese almacenamiento con una simple llamada.

```
type
  CadReg  = string [80];
  PtrLocal = ^Local;
  Local   = object
    Nombre    : CadReg;
    Longitud : integer;
    N         : Real;
    Constructor Iniciar (Cad : CadReg;
                           P     : integer;
                           Kg    : Real);
    Destructor Vacio;
  end;
var
  Test : PtrLocal;
```

Esta declaración define tres tipos: una cadena de 80 caracteres, un puntero a *Local* y un objeto denominado *Local*. Este objeto tiene tres campos y dos métodos asociados con ella.

Los constructores se utilizan para inicializar objetos. Tipicamente la inicialización se basa en valores pasados como parámetros al constructor. Los destructores son opuestos a los constructores y se utilizan para *limpiar* objetos después de su uso. Los constructores son vitales para utilizar métodos virtuales y los destructores son cruciales para el uso de la asignación dinámica.

20.9. LOS PROCEDIMIENTOS *NEW* Y *DISPOSE* EN POO

La sintaxis de los procedimientos *New* y *Dispose* se modificó en Turbo Pascal 5.5 para facilitar la inicialización y asignación de almacenamiento dinámico de objetos. Se puede llamar a estos procedimientos tal como ya conoce, pero también puede hacerlo con un segundo parámetro —bien un constructor o bien un destructor.

Los procedimientos anteriores toman dos parámetros: una variable dinámica —nombre del puntero— y un nombre de procedimiento llamada al constructor.

Formato

<pre>New (P, Construir) Dispose (P, Destruir)</pre>

<i>P</i>	variable puntero que apunta a un tipo objeto
<i>Construir</i>	llamadas a constructores de tipo objeto
<i>Destruir</i>	llamada a destructores de tipo objeto

La sentencia New indica a Turbo Pascal que asigne la variable objeto *P* y ejecute su método constructor *Construir*.

La sintaxis ampliada de New y Dispose tiene el mismo efecto (son equivalentes) que la ejecución de las sentencias siguientes:

New	New (P)	Dispose	P^.Destruir;
	P^.Construir		Dispose (P);

No obstante, la sintaxis ampliada mejora la fiabilidad y genera también código más corto y eficiente, por lo que es más recomendable su uso.

Teniendo las declaraciones.

```
tipe
  Item = object
    Cad : string (25);
    N.D : integer;
    Constructor Iniciar (C : string(25); NN,DD : (integer));
  end;
Var Test : ^Item;
```

Se puede realizar una llamada a New con la sentencia

```
New (Test, Iniciar ('Feliz Dia', 5,320));
```

La llamada precedente realiza las siguientes tareas:

- Asigna espacio suficiente en la pila dinámica para almacenar una variable de tipo local.
- Asigna la posición de este segmento de memoria a Test.
- Asigna los argumentos pasados a Iniciar a los campos adecuados de Test^.

Regla

- | |
|---|
| <ul style="list-style-type: none"> • La asignación dinámica de una variable se realiza con new. • La designación dinámica es una imagen espejo de la asignación y se realiza con dispose. |
|---|

20.10. MEJORAS EN PROGRAMACIÓN ORIENTADA A OBJETOS

Turbo Pascal 6.0 introdujo una nueva palabra private que señala la sección privada de un objeto. Turbo Pascal 7.0 ha añadido dos nuevas palabras reservadas: public,

que marca la sección pública del objeto, e *inherited*, que se utiliza para la llamada de un método heredado de un objeto ascendiente a partir de un objeto descendiente.

20.10.1. La ocultación mediante *public* y *private*

Todas las declaraciones de un objeto son públicas, por defecto. No obstante, en numerosas ocasiones se requiere que métodos y datos sean privados. Turbo Pascal 6.0 y 7.0 contiene la palabra reservada *private*, de modo que todas las declaraciones a continuación de la palabra *private* son accesibles sólo a las sentencias en el interior del módulo objeto.

Los campos y métodos privados son accesibles sólo dentro de la unidad en que se declara el objeto; se declara inmediatamente después de los campos y métodos ordinarios, a continuación de la palabra reservada opcional *private*. La sintaxis completa de una declaración objeto es:

```
type
  ObjetoNuevo = object (ascendiente)
    campos;          {públicos}
    métodos;         {públicos}
    private
      campos;        {privados}
      métodos;        {privados}
  end;
```

Así por ejemplo, el objeto Demo, que contiene métodos, datos, constructores y destructores, tiene todos los campos con carácter público:

```
Demo = object
  Campo1 : Integer
  constructor Iniciar;
  destructor Done;
  procedure Método
end;
```

y una variable de tipo demo

```
var
  Instancia:Demo;
```

con esta definición las sentencias siguientes son legales.

```
Instancia.Iniciar;
Instancia.Método;
WriteLn (Instancia.Campo1);
Instancia.Done;
```

En muchas ocasiones se desea evitar que los campos dato de un objeto no se puedan modificar con sentencias tales como:

```
Instancia.Campo1 := 4567;
```

En estos casos es preciso declarar privados dichos campos en el objeto Demo.

```
Demo = Object
  constructor Iniciar;
  destructor Done;
  private
    Campo1: Integer;
    procedure Metodo;
end;
```

Las declaraciones privadas (Campo1 y Metodo) deben venir al final del objeto. Todas las restantes declaraciones encima de private son públicas. Con esta nueva definición del objeto Demo las cuatro sentencias anteriores siguen siendo válidas, pero las dos sentencias siguientes:

```
WriteLn (Instancia.Campo1)
Instancia.Metodo;
```

se compilarán sólo si las sentencias aparecen en el mismo programa o modelo de unidad que declare Demo.

El ámbito de los identificadores declarados en la sección private actúan como identificadores de componentes públicos dentro del módulo que contiene la declaración del tipo objeto, pero fuera del módulo cualquier componente privado es desconocido e inaccesible.

Turbo Pascal 7.0 añade la palabra reservada public que significa que los métodos y campos se pueden agrupar de modo más lógico, por su función, dentro de la definición del objeto. Un objeto Dibujo se declara así:

```
Dibujo = object
  private
    x,y, long : Integer;
    Nombre : String;
  public
    constructor Copiar (var F:Dibujo);
    constructor Iniciar (FX,FY,Flong : Integer;
                           nombreF:String);
    destructor Done; virtual;
    procedure Visualizar; virtual;
    procedure Editar; virtual;
    ...
  private
    procedure VisualizarPuntos (x,y:Integer; S:String);
end;
```

RESUMEN

En este capítulo se han examinado las características orientadas a objetos que soporta Turbo Pascal. Las propiedades fundamentales de la programación orientada a objetos son: *abstracción, encapsulamiento, herencia y polimorfismo*.

- Un *tipo objeto o clase* (en C++, Smalltalk, Java, etc.) es una estructura de datos, encapsula datos y operaciones (*métodos*: procedimientos y funciones) en una misma estructura. Un programa orientado a objetos es una colección de objetos que interactúan entre sí.
- Una definición de un tipo objeto puede tener campos dato privados y métodos privados, que son accesibles sólo dentro de la unidad o programa que contiene la definición del tipo objeto. Asimismo, un tipo objeto tiene una sección pública donde suele ir los métodos públicos que conforman la interfaz del tipo de objeto. Las palabras reservadas *private* y *public* delimitan ambas secciones.
- Los tipos objetos pueden tener relaciones de ascendientes y descendientes. Los tipos objetos descendientes heredan campos datos y métodos de los tipos objetos definidos anteriormente como ascendientes.
- La herencia permite definir una familia de tipos objetos con una superclase y subclases derivadas.
- El polimorfismo permite utilizar el mismo nombre de un procedimiento/función que realiza una operación sobre tipos de datos diferentes.
- Las subclases (tipos objetos descendientes) pueden anular un método declarado en una superclase duplicando el nombre del método definido anteriormente.
- Los métodos polimórficos han de ser declarados virtuales. Los tipos objetos que contienen métodos virtuales deben incluir un método de iniciación (Iniciar) llamado constructor.
- Los objetos dinámicos que se asignan en tiempo de ejecución, utilizan variables puntero que son similares a las variables puntero ordinarias. Los tipos objeto que contienen métodos virtuales deben incluir un destructor, que es un método de limpieza que libera el almacenamiento asignado al objeto.

Sintaxis

Declaración de un tipo objeto

```
type NombreObjeto = object
    dato_publico_1 : tipo1;
    dato_publico_2 : tipo2;
    ...
    metodo_publico1;
    metodo_publico2;
    ...
private
    dato_privado_1 : tipon+1;
    dato_privado_2 : tipon+2;
    ...
    metodo_privado1;
    metodo_privado2;
end;
```

Tipo objeto derivado

```
type Empleado = object (Persona);
    Nombre : String;
    Salario : Real;
end;
```

Declaración de instancias de objetos

```
var                               var
  Nombre_objeto : tipo_objeto;      P1, P2 : Persona;
```

Declaración de un método

```
procedure Persona.CalcularSalario (Factor : real);
begin
  ...
end;
```

Métodos virtuales

```
type Ventas = object
  NumeroStock : integer;
  Precio : real;
  constructor Iniciar (Num:integer;ElPrecio:real);
  function Factura: real; virtual;
  function Mayor (var Otros:Ventas):boolean;
end;
```

Constructores/destructores

```
constructor Iniciar;
destructor Terminado; virtual;
new (sintaxis ampliada);
dispose (sintaxis extendida);
```

ERRORES TÍPICOS DE PROGRAMACIÓN

- Cuando se implementa un método de un objeto, asegúrese cualificar el nombre del método con el identificador del tipo objeto mediante el operador punto. En caso contrario aparecerán mensajes del compilador similar a Unknown identifier o bien Undefined forward reference.
- Un error frecuente en el trabajo con objetos procede del uso incorrecto de variables puntero o fallos al utilizar constructores para iniciar instancias de tipo objeto.
- Cuando se trabaja con tipos objetos que tienen métodos virtuales, se debe utilizar constructores para iniciar instancias de estos tipos o cualquiera de sus descendientes.
- Se deben definir siempre métodos para manipular campos de objetos y evitar manipulación de campos directamente en un programa cliente.
- Si se utiliza procedure en lugar de constructor para un tipo objeto que contiene un método virtual no se recibirá un mensaje de error de sintaxis, pero la ejecución probablemente se abortará.
- Se debe llamar al constructor de cada instancia de un tipo objeto que tenga métodos virtuales. La llamada a un método virtual no inicializado producirá el mal funcionamiento del programa. Para evitar este error se ha de utilizar la directiva {\$R+} al menos durante las etapas de depuración del desarrollo de programas.

EJERCICIOS

- 20.1. Escribir una unidad que represente un tipo objeto cola genérica.
- 20.2. Escribir una unidad que represente un tipo objeto lista enlazada.
- 20.3. Escribir un tipo objeto Figura y tipo de objetos descendientes Cuadrado, Triangulo, Circulo, Elipse, Rectangulo, Triangulo Isosceles, Triangulo equilátero y Hexagono.
- 20.4. Construir jerarquías de clases de personas, empleados, gerentes, informáticos, programadores, ingenieros de software, administradores, mecánicos, fontaneros, electricistas.
- 20.5. Construir jerarquía de clases de animales: *a)* de la actualidad; *b)* prehistóricos.
- 20.6. Escribir una clase Punto que contenga las coordenadas cartesianas x, y , de un punto en un plano. Deducir a partir de ella la subclase Circulo que permita calcular el radio y el área de dicho círculo. Una vez declaradas dichas clases, declarar objetos MiPunto y MiCirculo.
- 20.7. Escribir un programa que haga uso de las clases Circulo y Punto.
- 20.8. Describir y justificar los objetos que se obtienen en cada uno de los casos siguientes:
 - a)* Los habitantes de Europa y sus direcciones de correo.
 - b)* Los clientes de un banco que tienen una caja fuerte alquilada.
 - c)* Las direcciones de correo electrónico de una universidad.
 - d)* Los empleados de una empresa y sus claves de acceso a sistemas de seguridad.
- 20.9. ¿Cuáles serán los objetos que han de considerarse en los siguientes temas?
 - a)* Un programa para maquetar una revista.
 - b)* Un contestador telefónico.
 - c)* Un sistema de control de ascensores.
 - d)* Un sistema de suscripciones a una revista.
- 20.10. Deducir los objetos necesarios para diseñar un programa de computadora que permita practicar diferentes juegos de cartas.

PROBLEMAS

- 20.1. Consideraremos una pila como un tipo abstracto de datos. Se trata de definir una clase que implemente una pila de 100 caracteres mediante un array. Las funciones miembro de la clase deben ser:
meter, sacar, pilavacia y pilallena.
- 20.2. Reescribir la clase pila que utilice una lista enlazada en lugar de un array (sugerencia: utilice otra clase para representar los modos de la lista).

REFERENCIAS BIBLIOGRÁFICAS

Booch, Grady: *Análisis y Diseño Orientado a objetos*, Madrid, Addison-Wesley, Díaz de Santos, 1995.

- Joyanes Aguilar, L.: *Programación orientada a objetos*, Madrid, McGraw-Hill, 1996.
- *Programación orientada a objetos*, 2.^a edición, Madrid, McGraw-Hill, 1998.
- *Programación orientada a objetos*, 3.^a edición, Madrid, McGraw-Hill, 1998.
- *C++ a su alcance*, Madrid, McGraw-Hill, 1995.
- O'Brien, S. K., y Nameroff, S.: *Turbo Pascal 7. Manual de referencia*, Madrid, McGraw-Hill, 1993.
- Pressman, Roger: *Ingeniería del Software*, Madrid, McGraw-Hill, 1998.
- Rumbaugh, James, et al: *Modelo y diseño orientado a objetos (metodología OMT)*, Madrid, Prentice Hall, 1996.
- Yourdon, Ed., y Coad, Peter: *Object-Oriented Analysis*, Prentice Hall, 1990.
- *Object-Oriented Analysis*, 2.^a edición, Prentice Hall, 1991.

*Vademécum de Matemáticas
para la resolución de algoritmos
numéricos*

1. Suma de enteros de 1 a n

$$S = 1 + 2 + 3 + \dots + n$$

$$S = \left(\frac{n+1}{2} \right) n$$

2. Suma de una progresión aritmética

n = número de términos

a = primer término

d = diferencia común

u = último término, $u = a + (n - 1)d$

$$S = a + (a + d) + (a + 2d) + \dots + [a + (n - 1)d]$$

$$S = n \left(\frac{a + u}{2} \right) = \frac{n}{2} \{2a + (n - 1)d\}$$

3. Suma de una progresión geométrica

n = número de términos

a = primer término

r = ratio común, $r \neq 1$

l = último término, $l = ar^{(n-1)}$

$$S = a + ar + ar^2 + \dots + ar^{(n-1)}$$

$$S = a \left(\frac{r^n - 1}{r - 1} \right) = a \left(\frac{1 - r^n}{1 - r} \right)$$

4. Suma de los cuadrados de n números enteros

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

5. Suma de los cubos

$$\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$$

6. Suma de los números impares

$$S = 1 + 3 + 5 + \dots + (2n-1) = n^2$$

7. Suma de las potencias de $1/2^i$

$$S = \sum_{k=i}^j \frac{1}{2^k} = \frac{1}{2^i} + \frac{1}{2^{i+1}} + \dots + \frac{1}{2^j} \quad (\text{dos veces el primero menos el último})$$

$$S = \frac{2}{2^i} - \frac{1}{2^j}$$

8. Suma de id^i

$$\sum_{i=1}^n id^i = \left(\frac{d}{(d-1)^2} \right) [(nd-n-1)d^n + 1]$$

9. Suma de los fondos (floors) de logaritmos en base dos

$$L_n = \sum_{i=1}^n \lfloor \lg i \rfloor = (n+1)q - 2^{(q+1)} + 2 \quad \text{donde} \quad q = \lfloor \lg(n+1) \rfloor$$

10. Serie armónica, H_n

$$H_n = \sum_{i=1}^n \frac{1}{i} = \ln n + \gamma + O(1/n) \quad \text{donde } \gamma = 0,57721566$$

11. Leyes de potencias y exponentes

$$x^a x^b = x^{a+b}$$

$$x^a / x^b = x^{a-b}$$

$$x^0 = 1, \text{ dado } x \neq 0$$

$$x^{-b} = 1/x^b$$

$$x^{1/n} = \sqrt[n]{x}$$

$$1^a = 1, \text{ y } x^l = x \quad \text{para cualquier } a \text{ y } x$$

$$(xy)^a = x^a y^a$$

$$(x^a)^b = x^{ab}$$

$$(x/y)^a = x^a / y^a$$

12. Leyes de los logaritmos

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(x/y) = \log_b x - \log_b y$$

$$\log_b(1/x) = -\log_b x$$

$$\log_b(x^n) = n \log_b x$$

$$\log_b 1 = 0$$

$$\log_b b = 1$$

$$x \mathfrak{R} y \Leftrightarrow \log_b x \mathfrak{R} \log_b y, \text{ para } \mathfrak{R} \in \{=, \leq, <, >, \geq\} \text{ y } b > 1$$

$$\log_b x = \frac{\log_c x}{\log_c b}$$

$$\log_b c = \frac{l}{\log_c b}$$

13. Definición del operador mod

Sean a y b dos números reales. Definimos un mod $0 = a$, y si $b \neq 0$, se define un $a \text{ mod } b = a - b \lfloor a/b \rfloor$.

14. Leyes de Floor y Ceiling

Sea a un número entero, y sean x e y números reales.

$$\lfloor a \rfloor = \lceil a \rceil = a$$

$$\lfloor x \pm a \rfloor = \lfloor x \rfloor \pm a$$

$$\lceil x \pm a \rceil = \lceil x \rceil \pm a$$

$$\lfloor x \pm y \rfloor \geq \lfloor x \rfloor \pm \lceil y \rceil$$

$$\lceil x \pm y \rceil \leq \lceil x \rceil \pm \lceil y \rceil$$

$$-\lfloor x \rfloor = \lceil -x \rceil$$

$$-\lceil x \rceil = \lfloor -x \rfloor$$

$$\text{Si } x \leq y, \text{ entonces } \lfloor x \rfloor \leq \lfloor y \rfloor \quad y \quad \lceil x \rceil \leq \lceil y \rceil$$

15. Leyes de congruencia

(A) Si $a \equiv b$ (módulo m) y $r \equiv s$ (módulo m), entonces $(a \pm r) \equiv (b \pm s)$ (módulo m) y $(ar) \equiv (bs)$ (módulo m).

(B) Dados c y m primos entre sí, si $ac \equiv bc$ (módulo m), entonces $a \equiv b$ (módulo m).

16. Solución a relaciones de recurrencia

$$T(1) = a \text{ y } T(n) = b + c T(n-1)$$

Condición \Rightarrow Solución

$$c = 1 \Rightarrow T(n) = bn + (a - b)$$

$$c \neq 1 \Rightarrow T(n) = \left(\frac{a-b}{c} + \frac{b}{c-1} \right) c^n - \left(\frac{b}{c-1} \right)$$

17. Solución a relaciones de recurrencia

$$T(1) = a \text{ y } T(n) = bn + c + dT(n-1)$$

Condición \Rightarrow Solución

$$d = 1 \Rightarrow$$

$$T(n) = \left(\frac{b}{2} \right) n^2 + \left(\frac{b}{2} + c \right) n + (a - b - c)$$

$d \neq 1 \Rightarrow$

$$T(n) = \left(\frac{a - b - c}{d} + \frac{b + c}{d - 1} + \frac{b}{(d - 1)^2} \right) d^n - \left(\frac{b}{d - 1} \right) n + \left(\frac{c - (b + c) d}{(d - 1)^2} \right)$$

18. Solución a relaciones de recurrencia

$$T(1) = a$$

$T(n) = bn + c + d$ ($T(n/p)$, donde $p > 1$, $d > 0$)

Condición \Rightarrow Solución

$d = p \Rightarrow$

$$T(n) = b n \log_p n + \left(a + \frac{c}{d - 1} \right) n - \left(\frac{c}{d - 1} \right)$$

$d = 1 \Rightarrow$

$$T(n) = \left(\frac{bp}{p - 1} \right) n + c \log_p n + a + \left(\frac{bp}{p - 1} \right)$$

$$d = 1 \text{ y } b = 0 \Rightarrow T(n) = c \log_p n + a$$

$d \neq 1 \text{ y } d \neq p \Rightarrow$

$$T(n) = \left(a + \frac{bp}{d - p} + \frac{c}{d - 1} \right) n^{\log_p d} - \left(\frac{bp}{d - p} \right) n - \left(\frac{c}{d - 1} \right)$$

Unidades estándar de Turbo Borland Pascal 7

CONTENIDO

- B.1. Las unidades estándar.
- B.2. La unidad System.
- B.3. La unidad Printer.
- B.4. La unidad Dos.
- B.5. Procedimientos y funciones de la unidad Dos.
- B.6. La unidad Crt.
- B.7. La unidad Strings: funciones.

RESUMEN.

En este apéndice se examinan en detalle las siete unidades estándar proporcionadas con Turbo Pascal. Los objetos (constantes, tipos, variables, procedimientos y funciones) declarados en ellas en la parte del interface pueden ser utilizados por cualquier programa, con tal de que la unidad sea declarada en la cláusula uses.

Para utilizar una de las unidades estándar, basta con incluir su nombre en su cláusula uses, por ejemplo:

```
uses Crt, Graph, Printer;
```

Las unidades estándar residen en la biblioteca TURBO.TPL, que se carga automáticamente cuando se arranca Turbo Pascal. Si desea ahorrar memoria, puede eliminar las unidades raramente utilizadas como Turbo3 y Graph, utilizando el programa de utilidad TPUMOVER.

A lo largo del capítulo se describen todos los objetos accesibles por el programador y que deberá conocer a fin de sacar el máximo rendimiento a Turbo/Borland 7.0, perteneciente a todas las unidades estándar con la excepción de las unidades Graph y Graph3 que no son objeto de estudio en esta obra.

B.1. LAS UNIDADES ESTÁNDAR

Las unidades estándar son independientes en su uso, excepto las unidades que facilitan la compatibilidad con la versión 3.0: *Turbo3* y *Graph3*, que requieren la unidad *Crt*. Esta dependencia se muestra en la Tabla B.1.

La unidad *System* está siempre utilizada implicitamente y nunca necesita ser especificada en una cláusula *uses*.

B.2. LA UNIDAD SYSTEM

La unidad *System* no exporta ninguna función ni procedimiento. Es de hecho la librería en tiempo de ejecución. Esta unidad se utiliza automáticamente por cualquier unidad o programa y no necesita nunca incluirse en una cláusula *uses*. Citaremos aquí las diferentes variables predeclaradas disponibles.

```
var
  Input      : Text;
  Output     : Text;
  PrefixSeg  : Word;
  HeapOrg    : Pointer;
  HeapPtr    : Pointer;
  FreePtr    : Pointer;
  FreeMix    : Word;
  HeapError  : Pointer;
  ExicProc   : Pointer;
  ExitCode   : Integer;
  ErrorAddr  : Pointer;
  RandSeed   : Longint;
  SaveInt00  : Pointer;
  SaveInt02  : Pointer;
  SaveInt23  : Pointer;
  SaveInt24  : Pointer;
  SaveInt75  : Pointer;
  FileMode   : Byte;
```

La descripción de las diferentes variables se indican en la Tabla B.2.

Tabla B.1. Unidades estándar

Unidad	Uses
System	<i>Ninguna</i>
Printer	<i>Ninguna</i>
Dos	<i>Ninguna</i>
Crt	<i>Ninguna</i>
Graph	<i>Ninguna</i>
Overlay	<i>Ninguna</i>
Turbo3	<i>Crt</i>
Graph3	<i>Crt</i>

Tabla B.2. Variables predeclaradas de *System*

Variable	Descripción
<i>Input/Output</i>	Archivos estándar de entrada/salida.
<i>PrefixSeg</i>	Variable tipo <i>Word</i> que contiene la dirección del segmento del Program Segment Prefix (PSP) creado por DOS cuando se ejecuta el programa.
<i>HeapOrg, HeapPtr,</i> <i>FreePtr, FreeMin, HeapError</i>	Variables utilizadas por Turbo Pascal para la gestión de la asignación dinámica de la memoria.
<i>ExitProc</i>	Variable tipo puntero utilizada para implementar (realizar) procedimientos de salida.
<i>ExitCode</i>	Código de salida.
<i>ErrorAddr</i>	Dirección conectada a un error en la ejecución.
<i>RandSeed</i>	Semilla del generador de números aleatorios.
<i> FileMode</i>	Modo de apertura de archivos
<i>SaveInt00</i>	Se guardan ciertos vectores de interrupción utilizados por el sistema operativo.
<i>SaveInt02</i>	Idem.
<i>SaveInt23</i>	Idem.
<i>SaveInt24</i>	Idem.
<i>SaveInt75</i>	Idem.

B.3. LA UNIDAD PRINTER

La unidad *Printer* ha sido creada para facilitar el manejo de la impresora dentro de un programa. *Printer* declara un archivo de texto llamado *Lst* y lo asocia con el dispositivo (puerto de comunicación) *LPT1* del DOS. Se puede enviar, por consiguiente, datos a la impresora gracias a *Lst* en las sentencias *Write* y *WriteLn*; para ello basta incluir la cláusula *Uses* siguiente en la cabecera del programa

```
uses Printer;
```

y a continuación incluir *Lst* como primer argumento en sentencias *Write*/*WriteLn*, cuyo segundo argumento desea enviar a la impresora

```
WriteLn (Lst, 'esta frase se escribe en la impresora');
```

El uso de *Printer* evita la declaración, asignación, apertura y cierre de un archivo de texto. Esta unidad no exporta ninguna función ni procedimiento.

Ejemplo

```
program PruebaImpresora;
uses Printer;
begin
  WriteLn (Lst, 'Hola amigo Mortimer')
end.
```

B.4. LA UNIDAD DOS

La unidad DOS realiza un gran número de rutinas de utilidad para tratamiento de archivos. Estas rutinas permiten ejecutar funciones relacionadas con el sistema operativo. La declaración de constantes, tipos y variables es la siguiente:

```
const
  {comprueban bits indicadores individuales en el registro Flags}
  {después de una llamada a Intr o a MsDos}

    FCarry      = $0001;
    FParity     = $0004;
    FAuxiliary = $0010;
    FZero       = $0040;
    FSign        = $0080;
    FOverflow   = $0800;

  {constantes utilizadas por los procedimientos de tratamiento de archivos
  durante la apertura y cierre de los mismos}

    fmClosed   = $D7B0;
    fmInput    = $D7B1;
    fmOutput   = $D7B2;
    fmInOut    = $D7B3;

  {atributos de los archivos}

    ReadOnly   = $01;
    Hidden     = $02;
    SysFile    = $04;
    VolumeID   = $08;
    Directory  = $10;
    Archive    = $20;
    AnyFile    = $3F;

type
  {registro utilizado por los procedimientos Intr y MsDos}
  Registers = record
    case integer of
      0 : (AX,BX,CX,DX,BP,SI,DI,DS,ES,FLags:Word);
      1 : (AL,AH,BL,BH,CL,CH,DL,DH : Byte);
    end;
```

```

{informaciones relativas a los archivos tipeados y no tipeados}
FileRec = record
    Handle    : Word;
    Mode      : Word;
    RecSize   : Word;
    Private   : array 1..26 of Byte;
    UserData  : array 1..16 of Byte;
    Name      : array 0..79 of Char;

{informes relativos a los archivos de tipo texto}

TextBuf = array 1..127 of Char;
TextReg = record
    Handle    : Word;
    Mode      : Word;
    BufSize   : Word;
    Privete  : Word;
    BufPos    : Word;
    BufEnd    : Word;
    BufPtr    : TextBuf;
    OpenFunc  : pointer;
    InOutFunc : pointer;
    FlushFunc : pointer;
    CloseFunc : pointer;
    UserData  : array 1..16 of byte;
    Name      : array 0..79 of char;
    Buffer    : TextBuf;
end;

{Estructura utilizada por los procedimientos FindFirst y FindNext}

SearchRec = record
    Fill : array 1..21 of Byte {reservado para DOS}
    Attr : Byte;{atributo del archivo}
    Time : Longint;{fecha y hora}
    Size : Longint;{tamaño del archivo}
    Name : string 12 ;{nombre del archivo}
end;

{Estructura de fechas y horas, utilizadas por PackTime y UnpackTime}
DateTime = record
    Year, Month, Day, Hour, Min, Sec : Word;
end;

{variable de error del DOS}
var
    DosError : Integer;

```

Los tipos y variables predeclarados de la unidad DOS son:

DateTime

Ciertos procedimientos, como **GetFTime**, **SetFTime** o **FindFirst**, utilizan un valor compactado en cuatro bytes que contienen la fecha y hora asociados a un archivo.

UnpackTime y **PackTime** permiten descompactar y compactar fechas y horas y hacen la llamada a la estructura **DateTime**.

FileRec, TextBuf y TextRec

Se utilizan de modo interno por Turbo Pascal y describen características relativas a los archivos.

Registers

Este tipo describe los registros e indicadores del procesador. Se utiliza para la transferencia de información con *Instr* y *MsDos*.

SearchRec

Contiene información sobre los archivos.

DosError

Se utiliza por muchas de las rutinas de la unidad DOS para informe de errores. Los valores almacenados en **DosError** son códigos de error DOS.

- 0 *no hay error*
- 2 *archivo no encontrado*
- 3 *camino no encontrado*
- 5 *acceso prohibido*
- 6 *manipulación no válida*
- 8 *falta memoria*
- 10 *entorno no válido*
- 11 *formato no válido*
- 18 *no existen más archivos*

B.5. PROCEDIMIENTOS Y FUNCIONES DE LA UNIDAD DOS

Los procedimientos y funciones de la unidad DOS que pueden ser utilizados en cualquier programa se enumeran en la Tabla B.3.

Función DiskFree

Proporciona el número de bytes libres en la unidad de disco especificada.

Tabla B.3. Procedimientos y funciones de la unidad DOS

Descripción	Nombre
Procedimientos de soporte de interrupción	GetIntVec Intr MsDos SetIntVec
Procedimientos fecha y hora	GetDate GetFTime GetTime PackTime SetDate SetFTime UnpackTime
Funciones de estado del disco	DiskFree DiskSize
Procedimientos manipulación de archivos	FindFirst FindNext GetFAttr
Procedimientos/Funciones de manipulación de procesos	DosExitCode (función) Execute (procedimiento) Keep (procedimiento)

Formato

DiskFree(unidad: word) : longint

unidad 0, indica la unidad por defecto

1, unidad 1; 2, unidad 3; etc.

longint tipo de datos resultante de la función

Ejemplo

```
uses
  DOS;
begin
  WriteLn (DiskFree(0) div 1024);
  WriteLn ('K-bytes disponibles en la unidad por defecto')
end.
```

Función DiskSize

Devuelve la capacidad total en bytes del disco situado en la unidad especificada.

Formato

DiskSize (unidad: word): longint

unidad 0, unidad por defecto; 1, unidad A; 2, unidad B
longint tipo de dato resultante de la función

Ejemplo

```
uses
  DOS;
begin
  WriteLn (DiskSize(2) div 1024);
  WriteLn ('K-bytes disponibles en la unidad 'B)
end.
```

Función DosExitCode

Esta función permite recuperar el código de salida de un proceso lanzado después del programa actual.

Formato

DosExitCode : word

word tipo de resultado de la función

El byte de menor orden contiene el código devuelto por el proceso, mientras que el byte de mayor orden contiene la información relativa al modo en que se termina el proceso: 0, indica terminación normal; 1, indica interrupción por *CTRL-C*; 2, indica interrupción por hardware; 3, indica terminación por ejecución del procedimiento *Keep*.

Procedimiento Exec

Ejecuta el programa especificado en la línea de órdenes. Realiza una salida temporal hacia el DOS a fin de efectuar un programa .EXE o COM (órdenes ejecutables).

Formato

Exec(Programa,LineaOrdenes : string)

Programa nombre del programa
LineaOrdenes línea de órdenes

Ejemplo

Para ejecutar una orden del DOS se escribe COMMAND.COM en *Programa* y el nombre de la orden como *LíneaOrdenes*.

```
Exec ('\' COMMAND.COM', '/C DIR*.PAS');
```

Un programa debe recurrir a la directiva de compilación \$M para definir el tamaño de la pila antes de emplear EXEC. Por ejemplo, la directiva \$M de un programa define el tamaño de la pila dinámica antes de emplear Exec(\$M 16384,0,0).

Procedimiento FindFirst

Busca en el directorio actual o especificado, la primera entrada que coincida con el nombre del archivo especificado y conjunto atributos.

Formato

```
FindFirst (Camino:string; Attr:byte; var FInfo: SearchRec)
```

Camino	máscara del directorio: '*. *'. '*. Pas', ...
Attr	atributos de archivos
const	
{ constantes de atributos de archivos}	
ReadOnly	= \$01;
Hidden	= \$02;
SysFile	= \$04;
VolumeId	= \$08;
Directory	= \$10;
Archive	= \$20;
AnyFile	= \$3F;

SearchRec se declara en el interfaz de la unidad dos.

```
type {buscar registro utilizado por FindFirst y FindNext}
  SearchRec = record
    Fill : array 1..21 of byte;
    Attr : byte;
    Time : longint;
    Size : longint;
    Name : string 12
  end;
```

Ejemplo

```
uses Dos;
var
  DirInfo : SearchRec;
```

```
begin
  FindFirst ('*.PAS', Archivo, DirectoryInfo);
  while DosError = 0 do
    begin
      WriteLn (DirInfo.Nombre);
      FindNext (DirInfo)
    end
  end;
```

Procedimiento FindNext

Permite encontrar los archivos siguientes, que satisfacen a los criterios de búsqueda de la llamada precedente a **FindFirst** o **FindNext**.

Formato

```
FindNext (var s : SearchRec)
```

Procedimiento GetDate

Devuelve la fecha del reloj interno en el sistema operativo.

Formato

```
GetDate(var Año, Mes, Dia, DiaSemana: word)
```

Año	rango 1980...2099
Mes	rango 1..12
Dia	rango 1..21
DiaSemana	rango 0..6 (0 es domingo, lunes...)

Procedimiento GetFAttr

Devuelve los atributos **Attr** del archivo de cualquier tipo F.

Formato

```
GetAttr (var F; var Attr: word)
```

- F** variable tipo archivo (tipificado, no tipado o texto que ha sido asignado pero no abierto).
- Attr** los diversos atributos corresponden a valores asociados a constantes definidas en el interfaz de la unidad DOS. Los atributos definidos como constantes en la unidad DOS se examinan con la operación AND con las máscaras de atributos de archivos.

```
const
  {constantes de atributos de archivos}
  ReadOnly = $01;
  Hidden   = $02;
  SysFile  = $04;
  VolumeId = $08;
  Directory= $10;
  Archive   = $20;
  AnyFile   = $3F;
```

Procedimiento GetFTime

Devuelve la fecha y hora (en forma compactada) de la última modificación del archivo *F*.

Formato

```
GetFTime (var F; var Time: longint)
```

- F** variable tipo archivo (tipado, no tipado o texto)

Procedimiento GetIntVec

Devuelve la dirección, almacenada en un vector, de la interrupción número *NoInt*.

Formato

```
GetIntVec (NoInt:byte; var vector: pointer)
```

Procedimiento GetTime

Devuelve la hora actual del sistema operativo (reloj interno).

Formato

```
GetTime (var Hora, Minutos, Segundos, Centesimas:word)
```

Hora rango 0..23
Minutos rango 0..59
Segundos rango 0..59
Centésimas rango 0..99

Procedimiento Intr

Este procedimiento permite llamar a la rutina de interrupción número *NoInt*.

Formato

```
Intr (NoInt: byte; var Regs: Registers)
```

NoInt número de interrupción (0..255)
Registers es un registro definido en DOS

Ejemplo

```
type
  Registers = record
    case integer of
      0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags:word);
      1:(AL,AH,BL,BH,CL,CH,DL,DH : byte);
  end;
```

Keep

Termina la ejecución del programa actual y lo guarda en memoria. Esta característica permite realizar programas residentes en memoria.

Formato

```
Keep(CodigoSalida : word)
```

MsDos

Este procedimiento permite llamar a la rutina de interrupción número \$21 de MS-DOS y ejecutar una llamada a una función DOS.

Formato

```
MsDos (var R : Registers)
```

```
type
  Registers = record
    case integer of
      0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: word);
      1: (AL,AH,BL,BH,CL,CH,DL,DH : byte);
  end;
```

PackTime

Este procedimiento efectúa la operación inversa del procedimiento **UnpackTime**. Convierte un registro FechaHora en paquete de 4 bytes.

```
FechaHora = record
  Anio, Mes, Dia, Hora
  Min, Seg           : Word
end;
```

SetDate

Este procedimiento permite modificar la fecha del reloj interno.

Formato

```
SetDate (Año,Mes,Dia,DiaSemana : word)
```

Año rango de 1980..2099
Mes rango de 1..12
Dia rango de 1..31

SetFAttr

Este procedimiento permite modificar los atributos **Attr** del archivo F, cualquiera que sea el tipo.

Formato

```
SetFAttr (var F; Attr : byte)
```

Ejemplo

```
uses DOS;
var
  f : file;
begin
  Assign (f, 'C:\AUTOEXEC.BAT');
  SetFAttr (f, Hidden);
```

```
ReadLn;  
SetFAttr ( f, archivo)  
end
```

Los valores de los atributos se definen como constantes en la unidad DOS.

```
const  
  (constantes de atributos de archivos)  
  ReadOnly = $01;  
  Hidden   = $02;  
  SysFile  = $04;  
  VolumeId = $08;  
  Directory= $10;  
  Archive   = $20;  
  AnyFile   = $3F;
```

SetFTime

Establece la fecha y hora de la última modificación del archivo F.

Formato

```
SetFTime (var F; FechaHora: longint)
```

SetIntVec

Establece un vector interrupción especificado a una dirección especificada.

Formato

```
SetIntVec (NoInt: byte; Vector: pointer)
```

NoInt número de interrupción (0..253)
Vector especifica la dirección

SetTime

Establece la hora actual en el sistema operativo.

Formato

```
SetTime (Hora,Minutos,Segundos,Centesima: word)
```

Hora 00..232, Minuto 0..59, Segundos 0..59, Centésima 0..99

UnpackTime

Este procedimiento convierte la fecha y hora, compactadas en el parámetro Compacto, en una variable FechaHora cuya estructura DateTime está definida en el interface de la unidad DOS.

Formato

```
UnpackTime (Compacto: longint; var FechaHora: DateTime)
```

DateTime es un registro declarado en la unidad Dos

```
DateTime = record
  Year, Month, Day, Hour,
  Min, Sec : word
end;
```

B.6. LA UNIDAD CRT

La unidad *Crt* implementa una amplia gama de rutinas que permiten un control completo de las características de su computadora, tales como control del modo pantalla, códigos de teclado, colores, ventanas y sonido. Una de las grandes ventajas de utilizar *Crt* es el aumento de velocidad y flexibilidad en las operaciones de salida a pantalla.

B.6.1. Archivos de entrada y salida

El código de iniciación de la unidad *Crt* asigna los archivos de texto estándar *input* y *output* para referirse al CRT en lugar de a los archivos de entrada y salida del DOS.

```
AssignCrt (Input); Reset (Input);
AssignCrt (Output); Rewrite (Output);
```

B.6.2. Ventanas

Crt soporta un conjunto completo de ventanas. El procedimiento *Window* permite definir una ventana en cualquier parte de la pantalla. Cuando se escribe en una ventana específica, la ventana actúa como si fuera una pantalla completa, dejando el resto de la pantalla inalterada, es decir, inaccesible. La ventana por defecto es la pantalla completa.

B.6.3. Caracteres especiales

Cuando se escribe en un archivo *Output* que ha sido asignado a *Crt*, existen caracteres de control que tienen significados especiales:

- # 7 *Campana*: emite un pitido de altavoz interno.
- # 8 *Retroceso*: mueve el cursor un carácter a la izquierda.

- # 10 *Avance de linea*: mueve el cursor una linea abajo.
- # 13 *Retorno de carro*: vuelve el cursor al extremo izquierdo de la ventana actual.

B.6.4. Línea de entrada

Cuando se lee de un archivo de texto que ha sido asignado a *Crt* o de un archivo *Input*, el texto se introduce línea a línea. La línea de entrada se almacena en la memoria intermedia interna del archivo de texto, y cuando las variables se leen, esta memoria intermedia se utiliza como archivo fuente. Siempre que se vacía la memoria intermedia, se introduce una línea. Al introducir una línea se debe tener presente la acción de las teclas de edición.

RETROCESO	borra el último carácter introducido.
ESC	borra la línea completa de entrada.
INTRO (ENTER)	termina la línea de entrada y almacena la marca fin de línea (CR+LF) en la memoria intermedia.
CTRL-S	actúa igual que <i>RETROCESO</i> .
CTRL-D	recupera un carácter de la última linea de entrada.
CTRL-A	igual que ESC.
CTRL-F	recuerda la última línea de entrada.
CTRL-Z	termina la linea de entrada y genera una marca fin de archivo.

B.6.5. Constantes, tipos y variables

Constantes modo Crt

Las siguientes constantes se utilizan como parámetros en el procedimiento *TestMode*.

```
const
  BW40    = 0;    {40x25 B/N con adaptador color}
  C40     = 1;    {40x25 color con adaptador color}
  BW80    = 2;    {80x25 B/N con adaptador color}
  C80     = 3;    {80x25 color con adaptador color}
  Mono    = 7;    {80x245 con adaptador monocromo}
  Last    = -1;   {último modo de texto activo}
  Font8x8 = 256;  {juego de caracteres en ROM}
```

C40 y C80 son constantes definidas para la compatibilidad con la versión 3.0.

Constantes color de texto

Las constantes siguientes se utilizan con los procedimientos *TextColor* y *TextBackground*:

```
const
  Black      = 0;
  Blue       = 1;
  Green      = 2;
```

```

Cyan      = 3;
Red       = 4;
Magenta   = 5;
Brown     = 6;
LightGray = 7;
DarkGray   = 8;
LightBlue  = 9;
LightGreen = 10;
LightCyan  = 11;
LightRed   = 12;
LightMagenta = 13;
Yellow    = 14;
White     = 15;
Blink     = 128;

```

Variables

Las variables que siguen permiten controlar la visualización y el desarrollo de los programas:

var CheckBreak : boolean; cuando es **verdadero**, al pulsar *CTRL-BREAK* se abortará el programa; **falso**, no tiene efecto la pulsación de *CTRL-BREAK*.

var CheckEOF : boolean; cuando toma el valor **verdadero**, se genera un carácter de fin de archivo si se pulsa *CTRL-Z* durante la lectura de un archivo asociado al teclado; si toma el valor **falso**, la pulsación de *CTRL-Z* no tiene efecto. **CheckEOF** es falsa por defecto.

var CheckSlow : boolean; facilita la sincronización de la pantalla en determinados adaptadores que producen fluctuaciones en la presentación.

var DirectVideo : boolean; **verdadero**, todas las operaciones de escritura (*write* y *writeln*) en la pantalla se efectúan por acceso directo a la memoria vídeo, en lugar de acceso a través de BIOS considerablemente más lenta.

var TextAttr : byte esta variable contiene el atributo actual de texto; para obtener una visualización parpadeante, amarillo sobre fondo rojo, se puede escribir:

```
TextAttr:=Yellow+Red*16+Blink;
```

var WindMin,WindMax : word estas dos variables contienen las coordenadas de la esquina superior izquierda y de la esquina inferior derecha de la ventana actual:

Lo(WindMin) coordenada *x* esquina superior izquierda.
Hi(WindMin) coordenada *y* de la esquina superior izquierda.

B.6.6. Procedimientos y funciones

AssignCrt

AssignCrt (var F: text)

Asocia un archivo de texto F a la pantalla (CRT); funciona de igual modo que el procedimiento assign, pero no se necesita ningún archivo externo.

CirEol

CirEol

Borra dos caracteres desde el cursor al fin de línea.

CirSer

CirSer

Borra la ventana actual (en su defecto, la pantalla) y posiciona el cursor en la esquina superior izquierda de éste.

Delay

Delay (ms:word)

Detiene la ejecución del programa durante *ms* milisegundos.

DeLine

DeLine

Borra el contenido de la línea en la que se encuentra el cursor.

GotoXY

GotoXY (x,y: byte)

Sitúa el cursor en el punto de coordenadas (x,y)
x comprendido entre 1 y 40/80 caracteres (columnas).
y comprendido entre 1 y 25 (filas).

```
Window (1, 10, 60, 20);
GotoXY (1, 1);
```

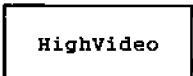
Ejemplo

```
GotoXY(10, 15);
Write('x');
```

```
GotoXY(15, 10);
Write('y');
```

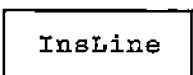
Este programa visualiza una 'x' en la línea 15, columna 10, y una 'y' en la línea 10, columna 15.

HighVideo



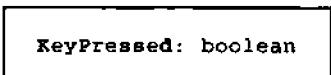
Selecciona caracteres de alta intensidad.

InsLine



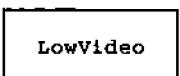
Inserta una línea vacía en la posición del cursor.

KeyPressed



Devuelve el valor **verdadero** si se pulsa una tecla, y **falso** en caso contrario.

LowVideo

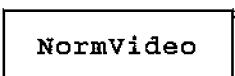


Selecciona caracteres de baja intensidad.

Ejemplo

```
TextColor (LightRed);
WriteLn ('este texto se visualiza en rojo claro');
LowVideo;
WriteLn ('este texto se visualiza en rojo oscuro');
HighVideo;
WriteLn ('este texto se visualiza en rojo claro');
```

NormVideo



Fija el atributo del texto al valor que tenía inmediatamente antes que el programa sea ejecutado.

NoSound

NoSound

Interrumpe la emisión de sonidos emitidos por el altavoz.

ReadKey

ReadKey: char

Lee un carácter del teclado; si la función **KeyPressed** tenía el valor *verdadero* antes de llamar a **ReadKey**, se devuelve un carácter; en caso contrario, la función **ReadKey** espera a que una tecla sea pulsada. El carácter devuelto no se visualiza en pantalla.

Sound

sound(Frec: word)

Emite un sonido de la frecuencia indicada (en hercios); este sonido se emite por el altavoz hasta el momento en que el procedimiento **NoSound** se llama; si no se encuentra **NoSound**, la emisión del sonido prosigue después del fin de la ejecución del programa.

TextBackground

TextBackground (Color; byte)

Selecciona el color de fondo; cada carácter visualizado en la pantalla en modo texto se sitúa en una matriz de puntos; por defecto, estos puntos que constituyen son de color negro y los caracteres visualizados en este fondo son de color blanco. valores de color:

0	negro
1	azul
2	verde
3	cian
4	rojo
5	magenta
6	marrón
7	gris claro

TextColor**TextColor (Color: byte)**

Selecciona el color del carácter de primer plano, es decir, el color del texto; valores de color:

0..7	igual que TextBackground
8	gris oscuro
9	azul claro
10	verde claro
11	cian claro
12	rojo claro
13	magenta claro
14	amarillo
15	blanco
128	parpadeo

TextColor y **TextBackground** permiten personalizar los colores de la pantalla.

TextMode**TextMode (Modo: word)**

Selecciona el modo de texto; los valores posibles del parámetro *modo* son:

0	negro/blanco, 40 caracteres (BW40)
1	color, 40 caracteres (C40)
2	negro/blanco, 80 caracteres (BW80)
3	color 80 caracteres (C80)
7	monocromo, 80 caracteres (MONO)
256	juego de caracteres en ROM (Font8x8)

Toda llamada al procedimiento **TextMode** provoca el borrado de la pantalla.

WhereX**WhereX: byte**

La llamada a esta función devuelve un valor entero que corresponde a la coordenada X del cursor.

WhereY**WhereY: byte**

La llamada a esta función devuelve un valor entero que corresponde a la coordenada Y del cursor.

Window**Window(x₁, y₁, x₂, y₂, : byte)**

Define una ventana de texto en la pantalla:
 esquina superior izquierda (x_1, y_1),
 esquina inferior derecha (x_2, y_2),
 (1,1) corresponde a la esquina superior izquierda de la ventana activa.

B.7. LA UNIDAD STRINGS: FUNCIONES

Esta unidad ya incorporada en *Turbo Pascal for Windows* incorpora un conjunto de 21 funciones para manipulación de cadenas terminadas en nulo. Estas funciones están codificadas en lenguaje ensamblador en linea.

Si desea trabajar con alguna de estas funciones, recuerde que debe incluir siempre una línea similar a ésta:

```
uses Strings;
```

Cuando se hable de cadena en las siguientes funciones se entenderá, salvo indicación expresa, que nos referimos a cadenas terminadas en nulo. A continuación se describen por orden alfabético las diferentes funciones de cadena. Todas ellas contienen su definición en la primera línea. A continuación, una breve descripción de su propósito, y por último, una o varias líneas con un ejemplo simple de su utilización:

StrCat

```
function StrCat(Destino, Fuente: Pchar) : Pchar;
```

Añade una cadena fuente al final de una cadena destino y devuelve un puntero a la cadena destino.

```
StrCat(S, 'Mackoy');
```

StrComp

```
function StrComp(S1,S2 : Pchar) : Integer;
```

Compara dos cadenas, S1 y S2, y devuelve un valor menor que cero si S2 < S1, 0 si S1 = S2 o mayor que cero si S1 > S2.

```
c := StrComp(S1,S2);
```

StrCopy

```
function StrCopy (Destino, Fuente : PChar) : PChar;
```

Copia una cadena *fuente* (Fuente) en una cadena *destino* (Destino) y devuelve un puntero a la cadena destino.

```
StrCopy(S,'Luis');
```

StrDispose

```
function StrDispose (Cad : PChar);
```

Libera el espacio almacenado por una cadena que fue asignada anteriormente con StrNew. Si Cad es nil, no hace nada.

```
StrDispose(P);
```

StrECopy

```
Function StrECopy (Destino, Fuente:PChar) : Pchar;
```

Copia una cadena fuente en una destino, devolviendo un puntero al final de la cadena resultante.

```
StrECopy(S1,S2);
```

StrEnd

```
function StrEnd(Cad : Pchar) : Pchar;
```

Devuelve un puntero al final de una cadena (el carácter nulo de terminación de la cadena).

```
WriteLn('Longitud:', StrEnd(Cad) - Cad);
```

StrIComp

```
function StrIComp (S1, S2 : Pchar) : Integer;
```

Compara dos cadenas sin tener en cuenta mayúsculas/minúsculas.

```
T := StrIComp(S1,S2);
```

StrLCat

```
function StrLCat (S1,S2 : Pchar; LonMax : Word) : Pchar;
```

Añade caracteres de una cadena al final de otra y devuelve una cadena concatenada, no mayor que la longitud máxima especificada en los argumentos.

```
StrLCat(S,'Mortimer', Sizeof(S) - 1);
```

StrLComp

```
function StrLComp(S1, S2 : Pchar; LongMax : Word) : Integer;
```

Compara dos cadenas hasta una longitud dada, sin tener en cuenta el tamaño de la letra.

```
t := StrLComp(S1,S2,10);
```

StrLCopy

```
function StrLCopy (S1,S2 : Pchar; LongMax : Word) : Pchar;
```

Copia caracteres de una cadena a otra y devuelve un puntero a la cadena destino.

```
StrLCopy(S1,S2,7);
```

StrLen

```
function StrLen (Cad : Pchar) : Word;
```

Devuelve la longitud de una cadena (sin contar el carácter de terminación nulo).

```
WriteLn ('Longitud es : ', StrLen (s));
```

StrLIComp

```
function StrLIComp (S1,S2 : Pchar; LongMax : Word) : Integer;
```

Compara dos cadenas hasta una longitud máxima, sin tener en cuenta el tamaño de la letra.

```
t := StrLIComp(S1,S2,10);
```

StrLower

```
function StrLower (S : Pchar) : Pchar;
```

Convierte una cadena a letras minúsculas.

```
WriteLn(StrLower(S));
```

StrMove

```
function StrMove(S1,S2 : Pchar; Cuenta:Word) : Pchar;
```

Copia caracteres de una cadena a otra y devuelve un puntero a la cadena destino.

```
P := StrMove(A,B,2);
```

StrNew

```
function StrNew(Cad : Pchar) : Strings;
```

Asigna una cadena en el montículo (*heap*).

```
P := StrNew(s);
```

StrPas

```
function StrPas (Cad : Pchar) : String;
```

Convierte una cadena terminada en nulo a una cadena estilo Pascal.

```
S := StrPas(Cad);
```

StrCopy

```
function StrCopy (Destino : Pchar; Fuente : String) : Pchar;
```

Copia una cadena Pascal en una cadena terminada en nulo.

```
StrPCopy(A,S);
ReadLn(Cad);
```

StrPos

```
function StrPos(S1,S2 : Pchar) : Pchar;
```

Devuelve un puntero a la primera ocurrencia de una cadena (S2) en otra (S1). Si S2 no está en S1, se devuelve nil.

```
StrPos(S,Sub);
P := StrPos(S, Subcadena);
```

StrRScan

```
function StrRScan (Cad : Pchar; Car : Char) : Pchar;
```

Devuelve un puntero a la última ocurrencia de un carácter dado dentro de una cadena, o nil si el carácter no existe en la cadena.

```
P := StrRScan (Nombre, '*');
if p = nil then
...
```

StrScan

```
function StrScan (Cad : Pchar; Chr : Char) : Pchar;
```

Devuelve un puntero a la primera ocurrencia de un carácter dado dentro de una cadena, o nil si el carácter no existe en la cadena.

```
Cars :=(StrScan(Nombre, '*') < > NIL) OR  
      (StrScan(Nombre, '#') < > nil);
```

StrUpper

```
function StrUpper (Cad : Pchar) : Pchar;
```

Convierte una cadena a letras mayúsculas y devuelve un puntero a la cadena.

```
ReadLn(Cad);  
WriteLn(StrUpper(Cad));  
WriteLn(StrLower(Cad));
```

RESUMEN

La versión 5.X y 6.0 de Turbo Pascal aporta ocho unidades predefinidas, que se llaman unidades *estándar*. Estas unidades contienen un gran número de procedimientos y funciones que completan el lenguaje Turbo Pascal. Los nombres de estas unidades estándar son: SYSTEM, DOS, OVERLAY, CRT, PRINTER, GRAPH, TURBO3 y GRAPH3. La versión 7.0 incluye dos nuevas unidades WINDOWS y STRINGS.

La unidad SYSTEM contiene las definiciones y declaraciones esenciales que emplea todo programa. Turbo Pascal incluye automáticamente los recursos de esta unidad, y el programador no necesita incluir SYSTEM en una cláusula **uses**; su presencia conduce a un error de compilación.

Las otras unidades estándar comprenden rutinas específicas utilizables en ciertos programas. Para poder usar estas rutinas es preciso indicar el nombre de la unidad que la contiene en la cláusula **uses**.

El editor de Turbo Pascal 7.0

Este apéndice recoge todas las órdenes de edición disponibles en el Entorno Integrado de Turbo Pascal 7.0.

ÓRDENES DEL EDITOR

Movimientos del cursor

Orden	Teclas de activación	Teclas alternativas de edición
Avanzar una página	PgDn (AvPág)	CTRL+C
Desplazar (<i>scroll</i>) una línea arriba	CTRL+Z	
Desplazar una linea abajo	CTRL+W	
Carácter a la derecha	→	CTRL+D
Carácter a la izquierda	←	CTRL+S
Principio de linea	CTR+QS	HOME (Inicio)
Final de línea	CTRL+QD	END (Fin)
Principio del archivo	CTRL+QR	CTRL+PgUp
Final de archivo	CTRL+QC	CTRL+PgDn
Línea arriba	↑	CTRL+E
Línea abajo	↓	CTRL+X
Moverse a la posición anterior	CTRL+QP	
Palabra a la derecha	CTRL+→	CTRL+F
Palabra a la izquierda	CTRL+←	CTRL+A
Parte superior de la ventana	CTRL+QE	
Parte inferior de la ventana	CTRL+QX	CTRL+END
Principio de línea	HOME (Inicio)	CTRL+QS
Retroceder una página	PgUp (RePág)	CTRL+R
Avanzar una página	PgDn (AvPág)	CTRL+C

Insertar y borrar

Orden	Teclas de activación	Teclas alternativas de edición
Borrar carácter en posición cursor	DEL (Supr)	CTRL+G
Borrar carácter izquierda cursor	RETROCESO	CTRL+H
Inserta opciones compilador	CTRL+OO	
Borrar línea	CTRL+Y	
Borrar hasta fin de línea	CTRL+QY	
Borrar hasta final de palabra	CTRL+T	
Insertar nueva linea	CTRL+N	
Activar/desactivar modo insertar	INS	CTRL+V

Bloques

Orden	Teclas de activación	Teclas alternativas de edición
Mover al principio del bloque	CTRL+QB	
Mover al final del bloque	CTRL+QK	
Marcar principio de bloque	CTRL+KB	
Marcar final de bloque	CTRL+KK	
Salir a barra de menús	CTRL+KD	
Ocultar/mostrar bloque	CTRL+KH	
Marcar línea	CTRL+KL	
Imprimir bloque seleccionado	CTRL+KP	
Marcar palabra	CTRL+KT	
Borrar bloque marcado	CTRL+KY	CTRL+DEL (Supr)
Copiar bloque marcado	CTRL+KC	CTRL+INS
Mover bloque marcado	CTRL+KV	SHIFT+DEL
Copiar en portapapeles	SHIFT+INS	CTRL+KC
Cortar de cortapapeles	CTRL+DEL	
Borrar bloque	CTRL+DEL	
Sangrar bloque	CTRL+KI	
Pegar de portapapeles	SHIFT+INS	ALT+EP
Leer bloques de disco	CTRL+KR	
No sangrar bloque	CTRL+KV	
Escribir bloque a disco	CTRL+KW	

Ampliación de bloques seleccionados

Orden	Teclas de activación	Teclas alternativas de edición
A la izquierda un carácter	SHIFT+→ (MAYUS←)	
A la derecha un carácter	SHIFT+→	
Fin de linea	SHIFT+END (MAYUS+FIN)	
Principio de linea	SHIFT+HOME	
Igual columna en linea siguiente	SHIFT+↓	
Igual columna en linea anterior	SHIFT+↑	
Avanzar página	SHIFT+PgDn (AvPág)	
Retroceder página	SHIFT+PgUp (RePág)	
A la izquierda una palabra	SHIFT+CTRL+←	
A la derecha una palabra	SHIFT+CTRL+→	
Final de archivo	SHIFT+CTRL+END	
Principio de archivo	SHIFT+CTRL+HOME	
		SHIFT+CTRL+PgDn SHIFT+CTRL+PgUp

Otras órdenes de edición

Orden	Teclas de activación	Teclas alternativas de edición
Activar/desactivar autosangrado	CTRL+OI	
Activar/desactivar tabulaciones	CTRL+OT	
Salida de EID (IDE)		ALT+X
Encontrar posición de marcador	CTRL+Q〃	
Ayuda	F1	
Índice de ayuda	SHIFT+F1	
Insertar carácter de control	CTRL+P	
Ir a posición último error	CTRL+QW	
Abortar operación actual	CTRL+U	
Abortar operación	ESC	
Maximizar ventana		F5
Cargar nuevo archivo		F3
Activar/desactivar modo relleno	CTRL+OF	
Correspondencia de parejas	CTRL+Q[, CTRL+Q]	
Guardar archivo y volver	CTRL+KS	
Buscar texto	CTRL+QF	
Buscar de nuevo	CTRL+L	
Buscar y sustituir texto	CTRL+QA	
Establecer marcadores	CTRL+K〃	
Activar/desactivar modo tabulación	CTRL+OT	
Ayuda búsqueda de tópico	CTRL+FI	
Deshacer	ALT+RETROCESO	
Activar/desactivar sangrado	CTRL+OU	
Visualizar directivas del compilador	CTRL+OO	ALT+EU



El entorno integrado de desarrollo Turbo Pascal 7.0

ÓRDENES DE TECLAS DE FUNCIÓN

Tecla de función/ Secuencia de teclas	Descripción
F1	Obtener ayuda sobre contexto actual.
F2	Guardar un archivo.
F3	Abrir un archivo.
F4	Ejecutar el programa y a continuación detener en la posición actual del cursor.
F5	Commuta ventana de edición entre pantalla completa y parcial.
F6	Mover a la siguiente ventana en la lista de ventanas abiertas.
F7	Traza de un programa paso a paso.
F8	Traza de un programa paso a paso a través de procedimientos y funciones.
F9	Construir el programa compilando todos los archivos requeridos.
F10	Cambiar a línea de menús.
ALT+F1	Visualizar pantalla de ayuda más reciente.
CTRL+F1	Obtener ayuda sobre tópico del lenguaje seleccionado.
CTRL+F2	Reinicializar el programa para ejecutar desde el principio.
ALT+F3	Cerrar ventana activa.
CTRL+F3	Visualizar pila de llamada durante ejecución del programa.
CTRL+F4	Evaluar expresión, valor o función durante ejecución del programa.
ALT+F5	Cambiar entre pantalla de salida y ventana de edición.
SHIFT+F6	Mover a ventana anterior en lista de ventanas abiertas.
CTRL+F7	Añadir variable de observación.
CTRL+F8	Bascular punto de interrupción.
ALT+F9	Compilar archivo.
CTRL+F9	Ejecutar programa, compilado primero si es necesario.

Menú Edit (*Alt-E*)

El menú **Edit** permite moverse por un archivo y entre archivos. Se activa con la secuencia de teclas **ALT-E**.

Orden	Tecla de función/ Secuencia de teclas	Descripción
Undo	ALT-RETROCESO ALT-EU	Deshace (anula) el último movimiento de la orden de borrado.
Redo	ALT-ER	Anula, cancela, la última orden undo .
Cut	ALT-ET SHIFT-DEL (Mayús-Supr)	Corta el bloque marcado de texto de la ventana activa y copia este bloque en el portapapeles.
Copy	ALT-ESC CTRL-INS	Copia el bloque marcado de texto de la ventana activa al portapapeles, dejando el texto en el archivo original.
Paste	ALT-EP SHIFT-INS	Pega texto de portapapeles a la posición del cursor en la ventana activa.
Clear	ALT-EL CTRL-DEL	Borra (limpia) el bloque marcado de texto desde la ventana de edición activa.
Show clipboard	ALT-ES	Abre el portapapeles, de modo que se puede editar su contenido.

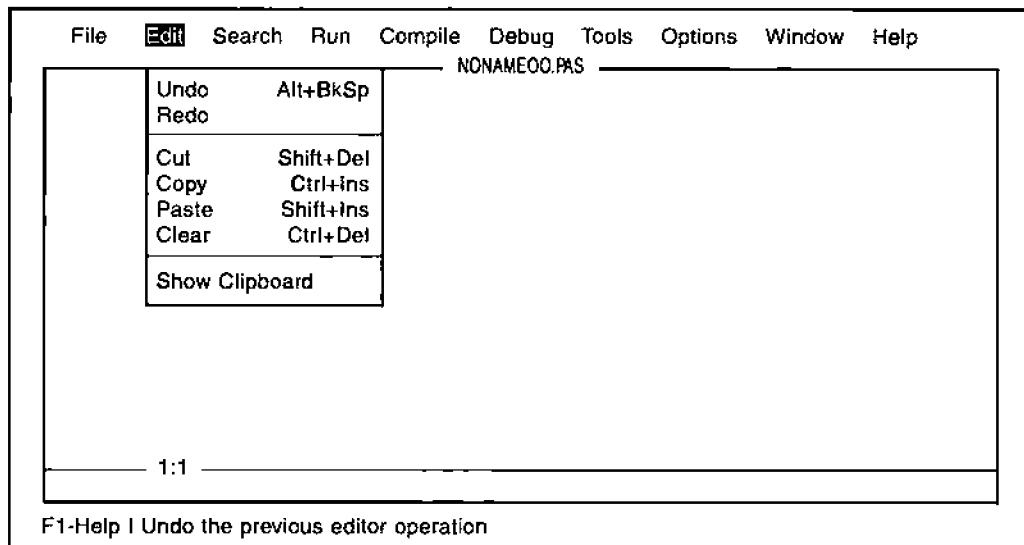


Figura D.2. Menú Edit.

Menú Search (Alt-S)

El menú Search contiene órdenes para encontrar textos específicos o ciertos tipos de texto (tales como errores en tiempo de ejecución).

Orden	Tecla de función/ Secuencia de teclas	Descripción
Find...	ALT-SF	Encuentra el texto específico, utilizando la directriz seleccionada en el siguiente cuadro de diálogo.
Replace...	ALT-SR	Busca el texto especificado en un cuadro de diálogo y sustituye este texto por otro texto especificado.
Search again	ALT-SS CTRL-L	Repite la última orden find o replace.
Go to line number...	ALT-SG	Ir a número de línea especificado en un cuadro de diálogo.
Show last compiler error...	ALT-SC	Muestra el último error del compilador.
Find error...	ALT-SE ALT-F8	Encuentra la posición del código fuente correspondiente a su error en tiempo de ejecución.
Find procedure...	ALT-SP	Encuentra el principio del procedimiento o función que se especifica en un cuadro de diálogo.

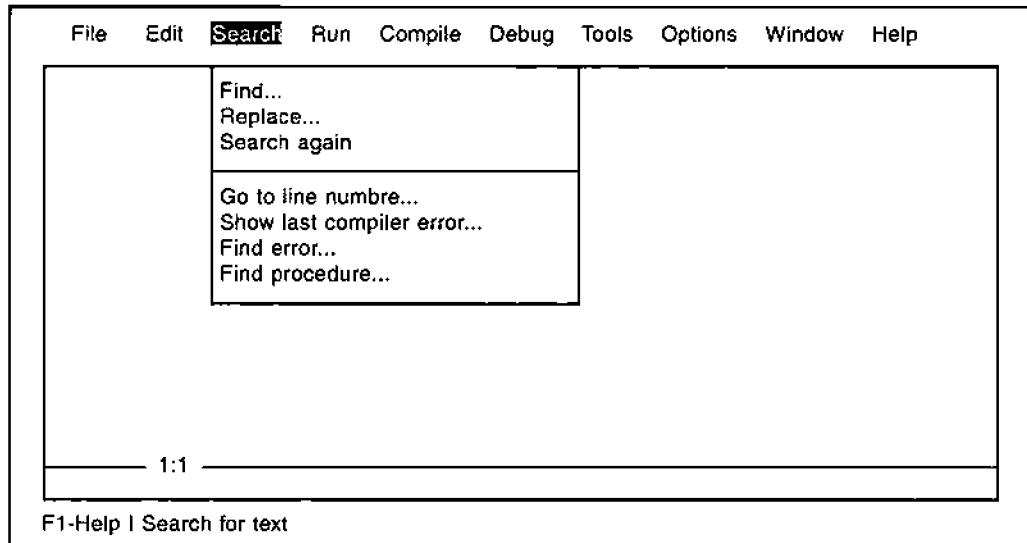


Figura D.3. Menú Search.

Menú Run (*Alt-R*)

Desde este menú se puede ejecutar su programa. Si es necesario, el programa se compilará automáticamente antes de su ejecución. El menú Run se despliega con **ALT-R**.

Orden	Tecla de función/ Secuencia de teclas	Descripción
Run	ALT-RR CTRL-F9	Ejecuta el programa de edición activa, compilando previamente si fuera necesario.
Step over	ALT-RS F8	Ejecuta el programa, en la ventana de edición activa, sentencia a sentencia (realiza la <i>traza</i>). Salta procedimientos y funciones, lo ejecuta como si fuera una única sentencia.
Trace into	ALT-RT F7	Ejecuta el programa, en la ventana de edición activa, sentencia a sentencia, incluyendo la traza de los procedimientos y funciones.
Go to cursor	ALT-RG F4	Ejecuta el programa, deteniéndose en la posición actual del cursor.
Program reset	ALT-RP CTRL-F2	Reinicializa el programa.
Parameters	ALT-RA	Especifica los parámetros que se pasan al programa cuando se ejecuta el entorno integrado de desarrollo, desde la línea de órdenes.

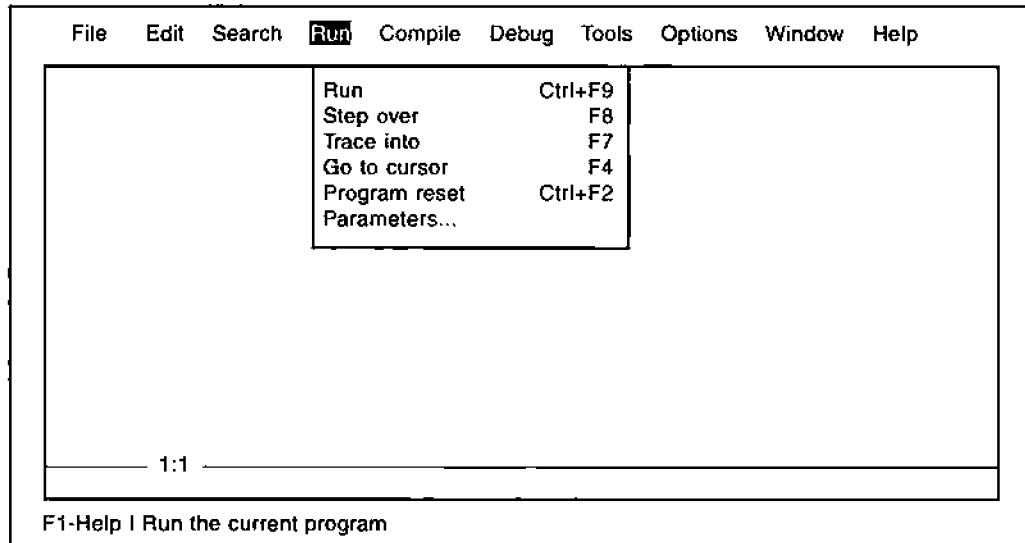


Figura D.4. Menú Run.

Menú Compile (Alt-C)

El menú **Compile** permite compilar programas selectivamente compilando sólo aquellos archivos que necesitan ser compilados. Este menú se activa con ALT-C.

Orden	Tecla de función/ Secuencia de teclas	Descripción
Compile	ALT-CC ALT-F9	Compila el archivo de la ventana activa de edición.
Make	ALT-CM F9	Constituye una versión ejecutable del archivo del programa principal, compilando ese archivo y otros archivos modificados utilizados en el programa.
Build	ALT-CB	Igual que Make , exceptuando que esta orden recompila archivos fuente del programa, sin tener en cuenta si han sido o no modificados.
Destination Memory	ALT-CD	Especifica si la versión ejecutable de un programa se almacena en memoria (<i>Memory</i>) o en disco (<i>Disk</i>).
Primary file...	ALT-CP	Especifica el archivo principal de un programa, cuyo código fuente se encuentra en archivos múltiples.
Clear primary file	ALT-CL	Borra el nombre principal del archivo, de modo que las órdenes Build o Make actúan sobre el archivo en la ventana de edición.
Information...	ALT-CI	Obtiene información sobre el archivo que está siendo compilado. Esta información se visualiza en un cuadro de diálogo.

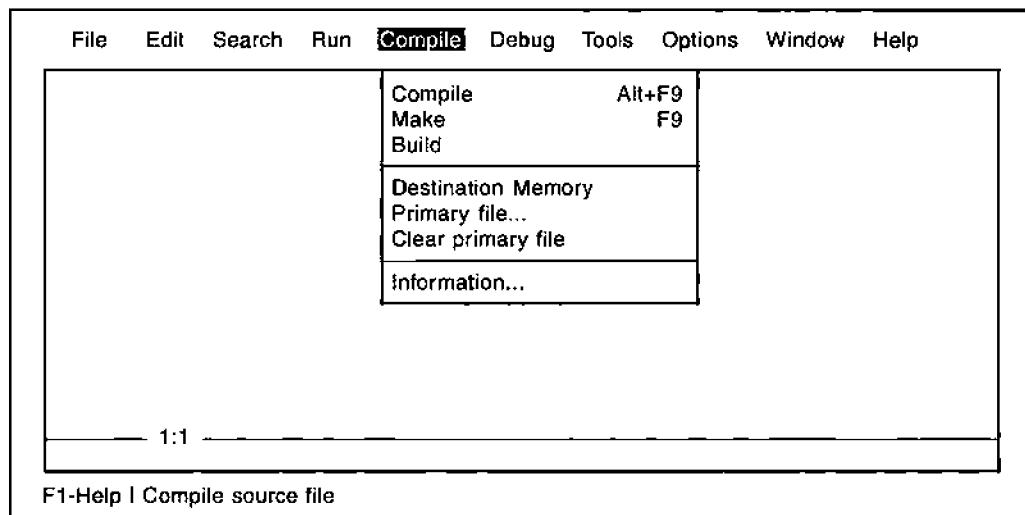


Figura D.5. Menú **Compile**.

Menú Debug (*Alt-D*)

El menú Debug contiene facilidades para depuración de programas. Para desplegar este menú, pulse ALT-D.

Orden	Tecla de función/ Secuencia de teclas	Descripción
Breakpoints...	ALT-DB	Accede a un cuadro de diálogo, en el cual se pueden editar, añadir y borrar puntos de ruptura.
Call stack	ALT-DC CTRL-F3	Abre una ventana que contiene una lista de procedimientos y funciones actualmente llamadas en el programa que se está ejecutando.
Register Watch	ALT-DR ALT-DW	Abre la ventana Register, haciéndola activa. Abre la ventana Watch en la que se especifican variables o puntos de observación.
Output User screen	ALT-DO ALT-DU (ALT-F5)	Abre y activa la ventana de salida (Output). Abre la pantalla de usuario, que utiliza la pantalla completa para visualizar salidas de programas.
Evaluate/modify...	ALT-DE CTRL-F4	Evaluá o modifica una expresión especificada en un cuadro de diálogo.
Add watch...	ALT-DA CTRL-F7	Añade una expresión de observación a la ventana Watch.
Add breakpoint...	ALT-DP CTRL-F8	Añade un punto de ruptura en un cuadro de diálogo.

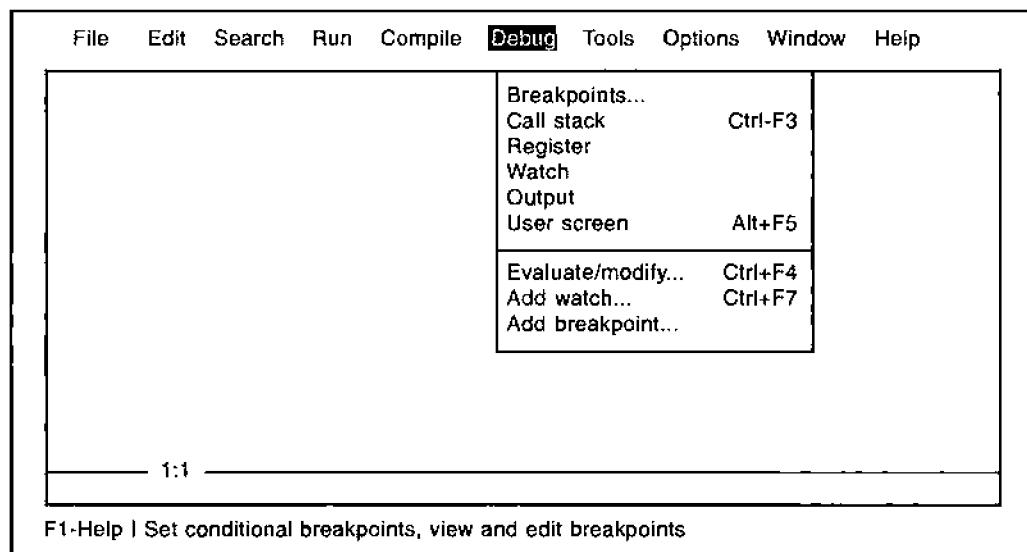


Figura D.6. Menú Debug.

Menú Tools (Alt-T)

Este menú, nuevo en Turbo Pascal 7.0, permite acceder a diferentes programas auxiliares que se pueden utilizar en su trabajo diario. Este menú se visualiza pulsando ALT-T.

Orden	Tecla de función/ Secuencia de teclas	Descripción
Messages	ALT-TM	Abre la ventana de mensajes <i>Messages</i> , de modo que pueda inspeccionar salidas de programas.
Go to next	ALT-TN ALT-F8	Salta al siguiente mensaje la ventana <i>Messages</i> .
Go to previous	ALT-TP ALT-F7	Salta al mensaje anterior con la ventana <i>Messages</i> .
Grep	ALT-TG SHIFT+F2	Invoca a la utilidad Grep, tipo UNIX.

Menú Options (Alt-O)

El menú Options permite modificar su entorno de trabajo y guardar y recuperar información de la configuración. Se activa pulsando ALT-O.

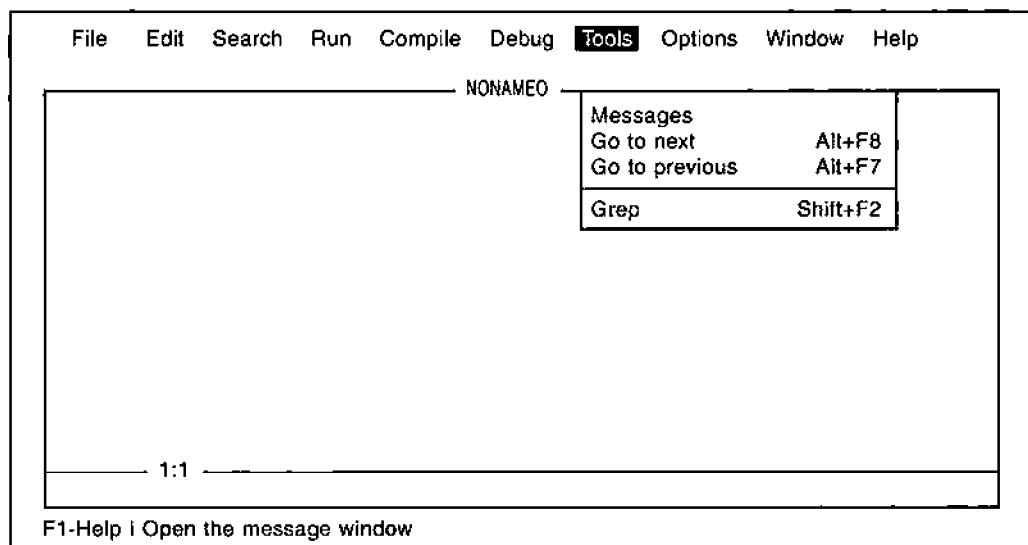


Figura D.7. Menú Tools.

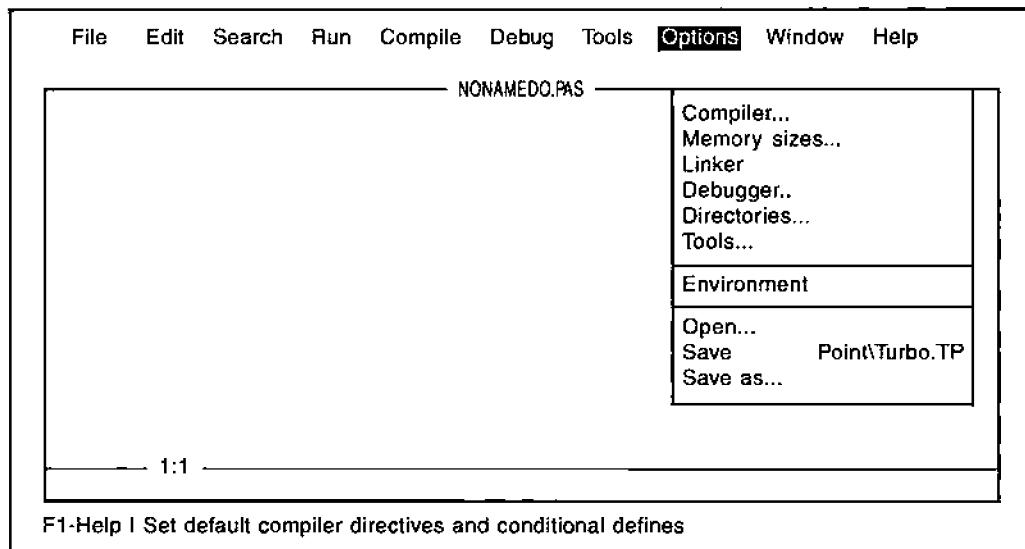


Figura D.8. Menú Options.

Orden	Tecla de función/ Secuencia de teclas	Descripción
Compiler...	ALT-QC	Especificar opciones de compilación.
Memory sizes...	ALT-OM	Especifica los tamaños de memoria por defecto (por ejemplo, de una pila) de un programa.
Linker...	ALT-OL	Especifica las opciones relativas al proceso del enlace.
Debugger...	ALT-OB	Especifica las opciones relativas al depurador integrador.
Directories...	ALT-OD	Especifica los directorios en los que se encuentran o escriben diferentes tipos de archivos (.EXE, .PAS, .TPU, etc.).
Tools...	ALT-OT	Edita, añade o borra herramientas de programas.
Environment/ Preferences	ALT-OE ALT-OEP	Especifica los parámetros del entorno EID. Especifica propiedades específicas de la pantalla.
Open...	ALT-OO	Abre un archivo con información de parámetros.
Save (TURBO..TP)	ALT-OS	Guarda los parámetros de configuración del archivo por defecto.
Save as...	ALT-OA	Guarda los parámetros de configuración del archivo especificado.

Menú Window (**Alt-W**)

El menú **Window** permite controlar el modo en que aparecen las ventanas en la pantalla y también qué ventanas son visibles. El menú **Window** se activa con **ALT-W**.

Orden	Tecla de función/ Secuencia de teclas	Descripción
Tile	ALT-WT	Encaja (en losa) todas las ventanas para que quepan en la pantalla. La ventana seleccionada se obtiene pulsando ALT y su número.
Cascade	ALT-WA	Coloca en cascada las ventanas de edición.
Close all	ALT-WO	Cierra todas las ventanas abiertas.
Refresh display	ALT-WR	Refresca o regenera la pantalla.
Size/Move	ALT-WS CTRL-F6	Cambia el tamaño o posición de la ventana activa, utilizando las ventanas activas para mover la ventana.
Zoom	ALT-WZ F5	Se expande o reduce la ventana activa hasta el tamaño de la pantalla.
Next	ALT-WN F6	Pasa a la siguiente ventana.
Previous	ALT-WP SHIFT-F6	Pasa a la anterior ventana.
Close	ALT-WC ALT-F3	Cierra la ventana activa.
List	ALT-WL ALT-O	Lista todas las ventanas abiertas.

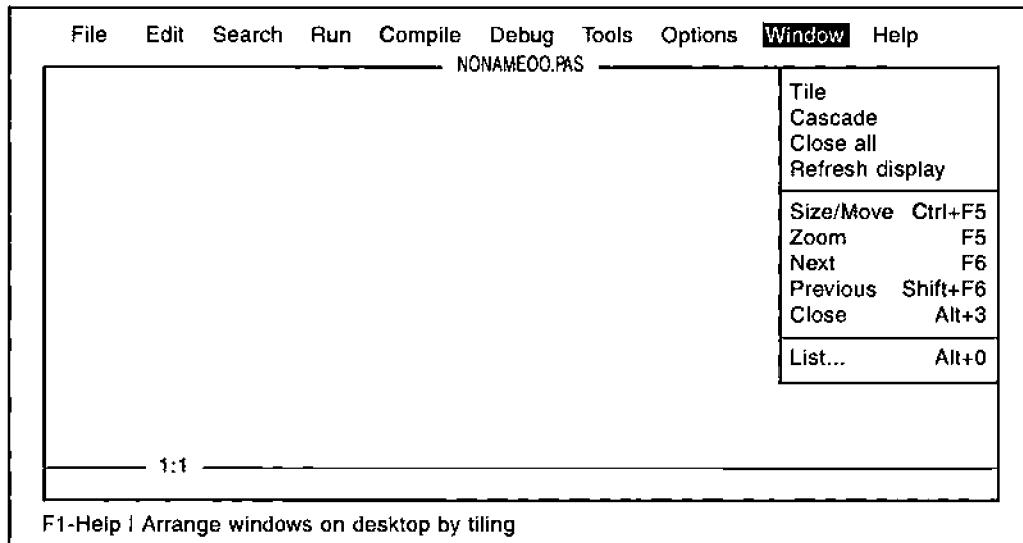


Figura D.9. Menú Window.

Menú Help (*Alt-H*)

El menú Help permite acceder ayuda a través de un índice de conceptos y nombres específicos o a través de una tabla general de contenidos. Este menú se visualiza pulsando ALT-H.

Orden	Tecla de función/ Secuencia de teclas	Descripción
Contents	ALT-HC	Abre la ventana de ayuda (Help), visualizando una tabla de contenidos.
Index	ALT-HI SHIFT-F1	Abre la ventana de ayuda con un índice de palabras reservadas.
Topic search	ALT-HT CTRL-F1	Abre la ventana de ayuda para información sobre la construcción de Pascal en la posición actual del cursor.
Previous topic	ALT-HP ALT-F1	Abre la ventana de ayuda (si ya no está abierta) en la pantalla que proporciona ayuda sobre el sistema de ayuda.
Using help	ALT-HH F1-F1	Abre la ventana de ayuda en una pantalla que proporciona ayuda sobre el sistema de ayuda.
Files	ALT-HF	Especifica archivos a instalar como material de ayuda.
Compiler directives	ALT-HD	Ayuda sobre directivas del compilador.
Reserved words	ALT-HR	Ayuda de palabras reservadas de Turbo Pascal.
Standard units	ALT-HU	Ayuda sobre unidades estándar de Turbo Pascal.
Turbo Pascal Language	ALT-HP	Ayuda sobre el lenguaje Turbo Pascal.
Error messages	ALT-HE	Ayuda sobre mensajes de error.
About...	ALT-HA	Información sobre la versión actual del entorno.

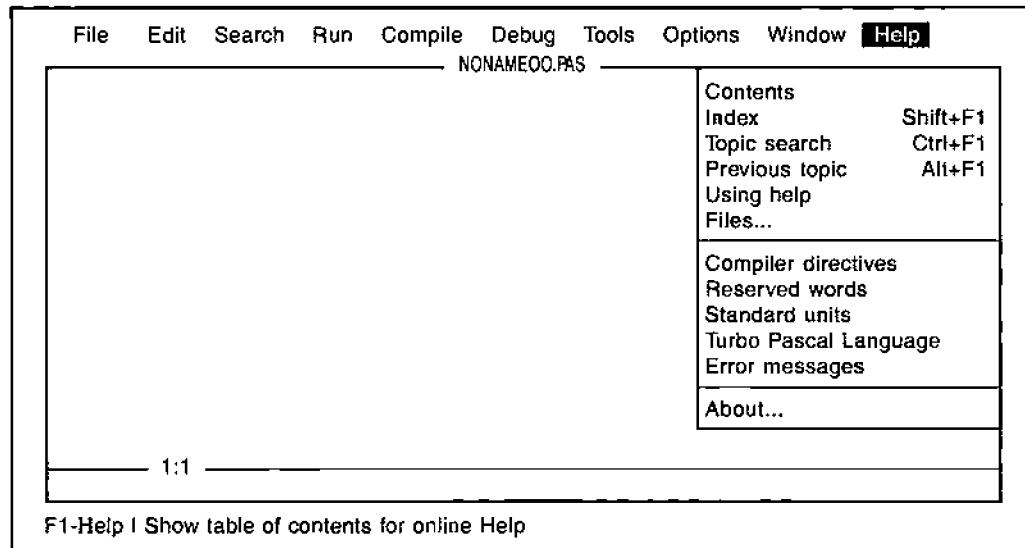


Figura D.10. Menú Help.

OPCIONES DEL EID (IDE) EN LA LÍNEA DE ÓRDENES

Cuando se invoca el entorno integrado de desarrollo (**EID**) desde la línea de órdenes, se pueden especificar ciertas opciones.

Estas opciones comienzan con un símbolo / o bien con un símbolo + o -.

- Si las opciones comienzan con -, se debe dejar al menos un espacio entre opciones consecutivas.
 - Las opciones están especificadas todas como códigos de una letra y no son sensibles a su tamaño.
 - Dependiendo de la opción, se ha de proporcionar un valor al argumento, por ejemplo, un nombre de archivo, un valor numérico, o un signo + o -.
- Situando un signo + o un blanco después de una opción se activa esa opción; situando un signo - después de la opción, se desactiva.

Las opciones de arranque utilizan esta sintaxis:

TPX	<i>[/opciones] [archivos]</i>	<i>modo protegido</i>
TURBO	<i>[/opciones] [archivos]</i>	<i>modo real</i>

Tabla D.1. Opciones en arranque del compilador

Opción	Descripción
<i>/C</i> nombrearchivo	Utiliza el archivo de configuración especificado como argumento.
<i>/D [+ o bien -]</i>	Trabaja en modo dual monitor.
<i>/E</i> número	Cambia el tamaño del montículo del editor al tamaño especificado (en Kb). El argumento debe ser un valor entre 28 (por defecto) y 128 (sólo modo real).
<i>/G [+ o bien -]</i>	Activa (o desactiva) una operación de grabación completa de archivos durante una sesión de depuración.
<i>/L [+ o bien -]</i>	Se debe especificar si se utiliza el EID en una pantalla LCD (tal como una computadora portátil).
<i>/N [+ o bien -]</i>	Activa (o desactiva) verificación de nieve en monitores CGA.
<i>/O</i> número	Cambia el tamaño del montículo de solapamiento (recubrimiento) al tamaño especificado (en Kb). El argumento debe ser un valor entre 64 y 256 con 112 K por defecto (sólo modo real).
<i>/P [+ o bien -]</i>	Se utiliza para controlar intercambio de paletas en adaptadores EGA.
<i>/R [+ o bien -]</i>	Si la opción R está activada, el EID arranca con el último directorio de trabajo en que se encontraba la última.
<i>-Scamino</i>	Especifica un área de intercambio rápida para archivos temporales del EID.
<i>/T [+ o bien -]</i>	Desactiva esta opción para indicar al EID que no cargue la biblioteca en tiempo de ejecución <i>turbo.tpl</i> . Si se desactiva, se debe tener disponible la unidad <i>system.tpu</i> .
<i>/W</i> número	Cambia el tamaño del montículo de la ventana al tamaño especificado (en Kb). El argumento debe ser un valor entre 24 y 64 con 32 K por defecto (sólo modo real).
<i>/X [+ o bien -]</i>	Desactivar esta opción para indicar al EID que no utilice memoria expandida (EMS) (sólo modo real).

Una línea de órdenes válida es:

```
turbo/E 128 test
```

ESTABLECER OPCIONES DE ARRANQUE EN EL EID

Se pueden también establecer las opciones de arranque dentro del propio EID. Para ello:

1. Seleccione la orden **Options | Environment | Startup**. Se visualiza el cuadro de diálogo **Startup Options**.
2. Seleccione la opción adecuada y elija el botón **Ok**.

Realizando estas acciones, sus opciones de arranque serán efectivas la próxima vez que se arranque el EID.

AYUDA EN LÍNEA

El EID tiene un sistema de ayuda en línea que le permite visualizar pantallas de ayuda y que presenta información sobre elementos diversos del lenguaje Pascal.

Así, una de las ayudas más necesarias en el desarrollo de programas es conocer el uso o función de una palabra reservada, un procedimiento o función predefinida. Un ejemplo típico es la obtención de información de la palabra reservada **string**. Para ello,

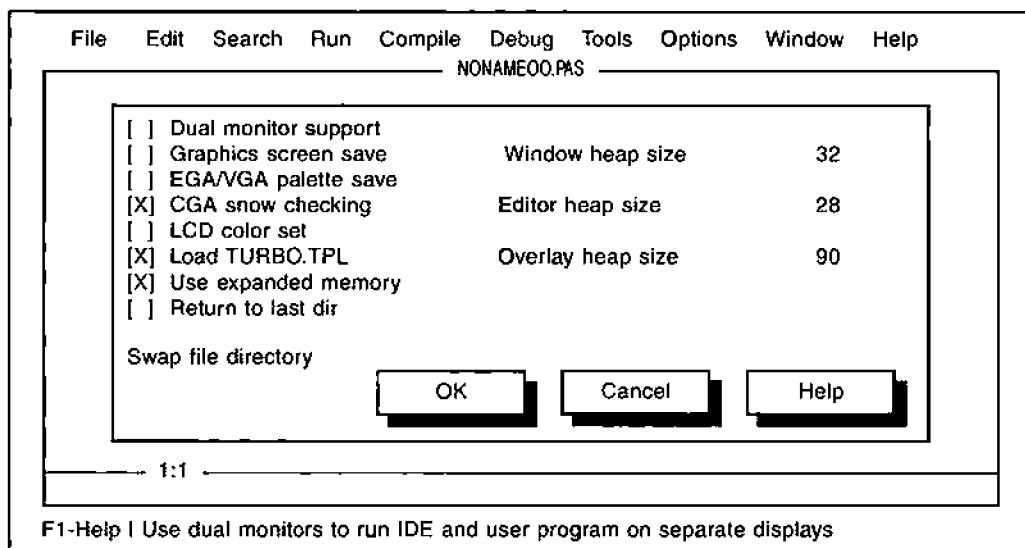


Figura D.11. Opciones de arranque del EID.

si en su pantalla está la palabra **string**, mueva el cursor a una de las letras de la citada palabra y pulse CTRL-F1. Aparece un cuadro de información tal como la Figura D.12.

En este momento se pueden realizar dos tareas:

1. Leer la siguiente información sobre cadenas. Pulsar *AvPág (PgDn)*. Aparecerá otra ventana de ayuda.

```
example:
-'Turbo'
-'That''s all'
...
```

2. Pulsar TAB y se moverá el cursor a la primera palabra coloreada —en amarillo— «*characters*» sombreándose el fondo azul. A continuación pulse INTRO y aparecerá una pantalla de ayuda relativa al tipo carácter Char.

Char type

Variables of the ordinal type Char are used to store ASCII characters.

Otros sistemas para obtener ayuda en línea son:

1. Seleccionar el menú **Help (ALT+H)** y activar la opción deseada.
2. Pulsar F1 y obtendrá información sensible al contexto dependiendo de la tarea que se esté realizando en ese momento: edición, depuración, etc.
3. Pulsar SHIFT+F1 (MAYUS+F1) para visualizar la pantalla con un índice de Turbo Pascal (*Turbo Help Index*)

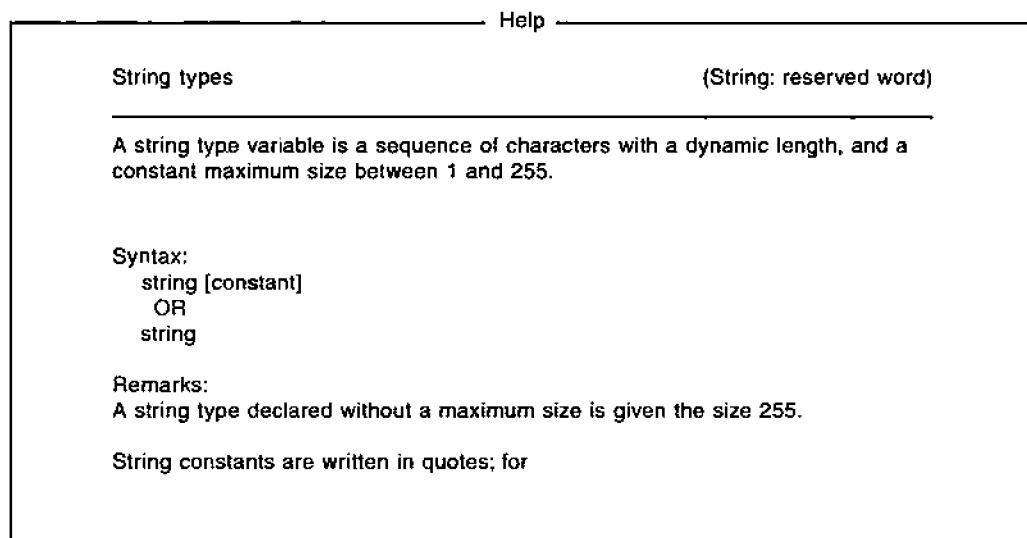


Figura D.12. Pantalla de ayuda de **string**.

SALIDA DEL ENTORNO INTEGRADO DE DESARROLLO (EID)

Existen dos formas para abandonar Turbo Pascal:

1. Salir totalmente del EID. Seleccione **File | Exit**. Si se han realizado cambios que no se han guardado, aparece un cuadro de diálogo (*Information*) que le pide si desea guardar su programa antes de salir.
2. Dejar el entorno temporalmente, al objeto de introducir órdenes en la línea de órdenes del DOS. Se selecciona **File | DOS Shell**. El EID permanece en memoria pero transfiere el control al DOS. Puede entonces ejecutar órdenes DOS e incluso ejecutar otros programas. Cuando desee retornar al EID, teclee **EXIT** en la línea de órdenes y pulse **Intro (Enter)**; el entorno EID aparece en la misma posición en que se quedó.

Depuración de sus programas en Turbo Pascal

Raramente los programas funcionan bien la primera vez que se ejecutan e incluso en numerosas ocasiones ni a la octava ni a la décima vez. De hecho, basta pensar que un programa es perfecto para que acto seguido nos proporcione quebraderos de cabeza en su ejecución. Cuando se pregunta a los programadores profesionales que resuman el proceso de depuración (corrección) de sus programas, la respuesta suele ser la misma: *experiencia*.

Afortunadamente, la programación en Turbo Pascal es diferente. Turbo Pascal proporciona un potente depurador incorporado que le ayuda a examinar el interior de un programa mientras se está ejecutando. Con el depurador se puede seguir la pista (traza) del flujo completo del programa, y se puede ver cómo cambia cada variable en cada paso del camino.

El depurador está construido en el entorno **EID*** de Turbo Pascal, por lo cual se puede editar, compilar y depurar sin dejar Turbo Pascal. Y, además, para programas grandes o complejos que requieran el soporte del lenguaje máquina, Borland proporciona el programa Turbo Debugger.

TIPOS DE ERRORES

Existen tres categorías o tipos de errores: de *compilación*, de *ejecución* y *lógicos*. Los niveles de exactitud —pasos hasta su corrección final— de un programa se indican en la Tabla H.1.

Errores de compilación

Un error de *compilación* (en tiempo de compilación) o de *sintaxis* ocurre cuando se viola una regla de la sintaxis de Pascal: olvidar un punto y coma, no declarar una varia-

* En inglés, IDE (Integrated Development Environment).

ble, palabra reservada mal escrita, ausencia de argumentos, pasar un número incorrecto de parámetros a un procedimiento, asignar un valor real a una variable entera, etc. Turbo Pascal no compilará su programa, es decir, no generará su código máquina hasta que sean reparados todos los errores de compilación.

La mayoría de los errores de compilación son evidentes y fáciles de fijar. Cuando Turbo Pascal detecta uno, el proceso de compilación se detiene y Turbo Pascal visualiza un mensaje en la pantalla que indica la naturaleza del problema y retorna al editor con el cursor posicionado en el lugar donde ha ocurrido el error. Simplemente debe corregir su error y comenzar de nuevo la compilación.

Así, por ejemplo, un error típico es olvidarse de escribir el separador punto y coma al final de una sentencia. Por ejemplo, supongamos tiene escritas las siguientes líneas:

```
WriteLn ('Fin del mensaje')
Saludar
END;
```

Al compilar el programa al que pertenecen esas líneas, el compilador localizará el error y se parará la compilación. Retornará al editor y el cursor se posicionará debajo de la S de Saludar (un determinado procedimiento). Asimismo en la parte superior de la ventana de edición se visualizará el mensaje de error:

Error 85: ";" expected (falta; al final de linea WriteLn)

Todos los mensajes de error producidos por el compilador Turbo Pascal están numerados; en este caso, el código de error correspondiente a la ausencia de un punto y coma es 85 (la lista de los mensajes de error se encuentra en el Apéndice I).

El proceso de compilación debe continuar hasta que su programa esté libre de toda clase de errores de compilación. Durante la compilación, Turbo Pascal visualiza un cuadro especial de mensajes que proporciona información sobre el destino de la compila-

Tabla E.1. Niveles de exactitud de un programa

Nivel	Exactitud de corrección
1	No compila bien.
2	Compila sin errores.
3	Comienza su ejecución pero se detiene con errores en tiempo de ejecución.
4	Se ejecuta pero produce resultados incorrectos.
5	Produce resultados correctos para un conjunto pequeño de datos de entrada normales.
6	Produce resultados correctos para un conjunto de datos normales de entrada bajo todas las condiciones de ejecución normales o esperadas.
7	Produce resultados correctos para datos de entrada extremos, anormales e incorrectos, bajo condiciones de ejecución normales, esperadas y no esperadas.
8	Produce resultados correctos para cualquier posible dato y condiciones de ejecución.

ción y el número de líneas que se han compilado. Después que la compilación ha terminado con éxito, el cuadro le solicita pulsar cualquier tecla para retornar al editor.

Algunos de los mensajes típicos de error y sus posibles causas se analizan a continuación:

Unkown Identifier

Este error significa que Turbo Pascal no puede encontrar un cierto identificador que ha utilizado en su programa —nombre de una variable, constante, procedimiento o algún otro elemento de un programa—. Normalmente se produce por alguna de estas causas:

1. Error de escritura cuando se tecleó el programa. Corrija el identificador. Es un error trivial y casi siempre fácil de corregir.
2. Olvidó nombrar una unidad en la cláusula **USES** de su programa; por ejemplo, olvidó poner el nombre de la unidad **Crt** en un programa que utiliza el procedimiento **ClSer**. (¡Atención! Este error —en concreto— no se puede producir en las versiones anteriores a la 4, ya que no soportaban unidades.)
3. Error en la estructuración de un programa. Esta causa de error es difícil de detectar, ya que el identificador correspondiente es inaccesible en el lugar del programa donde se ha utilizado. Para localizar este error debe tener presente lo siguiente:
 - Los identificadores deben ser declarados antes de ser utilizados. Si la subrutina (procedimiento o función) M utiliza la subrutina N, entonces N debe estar definida en el programa *antes* que la subrutina M.
 - Los identificadores interiores están ocultos al exterior de la subrutina.
 - Incluso dentro de una subrutina, un identificador debe declararse antes de ser utilizado y lo mismo se cumple para el caso de subrutinas.

Missing END or Comment Bracket

Las causas de posibles errores de este tipo se derivan de la estructuración de los programas: pérdidas de palabras **END** (al final de sentencias compuestas, procedimientos, funciones o programas). El mejor método para localizar estos errores es el conocimiento de la sintaxis de Pascal y el seguimiento o traza de la lógica del programa, con la ayuda de un lápiz que facilite las aperturas y cierres de estructuras sintácticas (**BEGIN-END** y **{}**).

Mixed-Up IFs and ELSEs

Este error se produce cuando se utilizan sentencias **IF-THEN-ELSE** o **CASE** anidadas. La causa más frecuente es la mala escritura de estas sentencias debido a la no previsión por el programador de que una sentencia **IF** siempre trata de grabar la sentencia **THEN** y/o **ELSE** más próxima. Al igual que el error «*missing END or Comment Bracket*», el mejor sistema para localizar este error es seguir su lógica con un lapicero.

«;» Expected

Este es un error trivial que se produce por el olvido de la escritura de puntos y coma para separar sentencias en un programa de otras sentencias. Para corregir este error basta con añadir un punto y coma al final de la línea donde falta. Normalmente, como se ha comentado anteriormente, el error de compilación ocurre en la línea inmediatamente después de la línea que no contiene el punto y coma.

Precaución

Antes de probar a ejecutar su programa, realice las siguientes dos tareas:

1. Pulse F2 para grabar la última versión de su programa en disco.
2. Visualice el menú **File** y seleccione la orden **Change dir.** Introduzca la unidad y directorio adecuados.

Errores de ejecución

Cuando ya se ha compilado el programa con éxito, pulse ALT-R para visualizar el menú **Run**, a continuación pulse R para ejecutar la orden **Run**. Para visualizar la salida del programa, pulse ALT-F5 para utilizar la pantalla completa como ventana de salida.

Los errores de ejecución (en tiempo de ejecución) son problemas que el compilador no puede detectar por anticipado, pero que originan la parada del programa antes de su terminación. Valores de entrada imprevistos, cálculos no válidos (división por cero), problemas en acceso a disco, etc., producen errores de ejecución.

Los errores de ejecución terminan el programa y visualizan un mensaje y un número de error. Si el programa se estaba ejecutando en el interior del entorno integrado, Turbo Pascal retorna al editor y lleva el cursor a la línea de código en que se activó el error. Los errores de ejecución pueden ser una incomodidad, pero dado que conoce la línea de código que estaba ejecutando cuando se produjo el error, no es difícil analizar y reparar el problema. Algunos mensajes típicos de error son:

Attemp to Assign Out-Of-Range Value

Este error se produce cuando se intenta utilizar tipos numéricos, enumerados y subrangos que tienen un rango o intervalo limitado de valores. Un ejemplo típico de error es intentar asignar al entero 800 a una variable de tipo byte que sólo toma valores de 0 a 255.

Failure to Handle Nonmatch in CASE Statement

Las sentencias **CASE** en Turbo Pascal no son tan rígidas en su funcionamiento como lo son en Pascal estándar. Cuando la variable selector de **CASE** toma valores diferentes a los casos contemplados, Turbo tiene cos protecciones —en lugar del error que se produ-

ce en Pascal estándar—. La primera protección es la cláusula **ELSE**, que sirve para ejecutar acciones determinadas, y la segunda protección es que en el caso de ausencia de **ELSE** el programa no se detiene y sigue en la siguiente línea.

Failure to Initialize Variable Before Use

Algunas variables necesitan ser inicializadas al principio de su programa o subrutina, otras no. El problema es, pues, localizar estas variables.

Las variables que se utilizan como contadores de bucles **FOR** no tienen que inicializarse; sin embargo, las variables utilizadas para controlar los bucles **WHILE** y **REPEAT** deben inicializarse. Las expresiones y variables que controlan las sentencias **IF** y **CASE** pueden o no necesitar ser inicializadas.

El método más seguro, en caso de duda, es inicializar la variable, utilizando simplemente una sentencia con su valor inicial.

```
S := 0;
Control := 5;
```

Excepción

El único tipo de variable que debe siempre inicializarse es el punto. Una variable puntero se debe fijar a **nil** al principio de un programa o subrutina, y también a **nil** tan pronto como se liberen y no sean ya necesarias.

File Not Found

Se trata de procesar (**RESET** o **APPEND**) un archivo que no existe o bien que existe pero en un directorio distinto al que está buscando su programa. Tal vez la solución más adecuada sea asegurarse que todos los archivos necesarios en la ejecución del programa están en el mismo directorio del disco.

File Not Closed

Si un archivo no se cierra al final de un programa, algunos datos pueden perderse, es decir, aquellos que se encuentren en la memoria intermedia del archivo.

Errores lógicos

Un error lógico se genera cuando el programa hace algo que no desea hacer o no hace lo que se espera que haga. Un error lógico es un fallo de diseño del programa. Los errores lógicos son los más difíciles de detectar y aislar.

A veces, naturalmente, un error lógico es evidente. Si el primer número de una página de un informe es 44, casi con seguridad que se habrá olvidado inicializar su variable

contador de páginas. Los errores lógicos producen resultados incorrectos o abortan los problemas. Su detección requiere un análisis exhaustivo más que un examen de las reglas de Pascal.

Algunos errores lógicos son:

Failure to Use VAR Parameter When Needed

Este es, seguramente, el error más probable cometido en las subrutinas. Recuerde que si se desea pasar una variable a una subrutina existen dos métodos: por valor y por referencia. Cualquier variable que deba ser modificada por una subrutina debe pasarse como un parámetro VAR, es decir, por referencia. Si se olvida pasar una variable con VAR en la lista de parámetros, se producirá este error.

Loop Exit Condition Never Reached

Este error se produce cuando un programa entra en un *bucle sin fin*.

Loop Executes Wrong Number of Times

Este error se produce por dos posibles causas: el bucle nunca se ejecuta, cuando al menos se debe ejecutar una vez, o se ejecuta al menos una vez cuando bajo ciertas condiciones nunca se debe ejecutar. La corrección de este error es una adecuada selección de los bucles WHILE o REPEAT.

EL DEPURADOR INTEGRADO TURBO PASCAL

Las versiones primitivas de Turbo Pascal disponían de pocas herramientas para depuración de programas; entre ellas la sentencia **WriteLn**, que permite visualizar el valor de una variable en un punto de su programa.

El depurador integrado de Turbo Pascal proporciona cuatro herramientas para depuración de programas: capacidad para seguir la ejecución del programa línea; capacidad para observar el valor de una o más variables a medida que se ejecuta el programa; capacidad de cambiar un valor de una variable interior mientras se ejecuta el programa, a fin de encontrar el efecto que ese cambio produce, y la capacidad de establecer *puntos de ruptura* o *interrupción* (*breakpoints*) en los que se detendrá el programa cuando se ejecute, de modo que se pueda inspeccionar los resultados hasta ese punto.

Preparando el depurador para su uso

Antes de comenzar a depurar un programa se deben hacer varias cosas. En primer lugar se deben activar las directivas de compilación **{\$D}** y **{\$L}**. Estos valores por defecto son **{\$D+}** y **{\$L+}**. La opción **{\$D}** puede ser controlada por la orden *Options/Compiler/Debug Information* y la opción **{\$L}** puede ser controlada por *Options/Compile/Local Symbols*.

Finalmente, antes de compilar su programa, debe asegurarse activar la opción *Options/Debugger/Integrated Debugging* que genera información de depuración en el archivo ejecutable.

Las características más sobresalientes del depurador se describen en los párrafos siguientes.

Los puntos de interrupción (*breakpoints*)

Se pueden marcar líneas en su programa como puntos de interrupción (*breakpoints*). Cuando se ejecuta su programa y se alcanza un punto de interrupción en la banda de ejecución. Se pueden entonces examinar variables, comenzar la traza o ejecutar el programa hasta que se encuentre otro punto de interrupción. Se puede también interrumpir en cualquier punto, pulsando CTRL-BREAK (CTRL-INTER). Esta acción tiene el efecto de parada en la línea fuente siguiente, como si de un punto de interrupción se tratara.

Observación/Vigilancia (*Watches*)

Se puede establecer un determinado número de puntos de observación, inspección o vigilancia en la ventana Window. Cada uno puede ser una variable, una estructura de datos o una expresión.

Ir hasta el cursor (F4)

La característica Go to Cursor le permite especificar un punto temporal de parada en su programa. Para utilizar esta característica, sitúe el cursor en una línea específica de su programa y pulse a continuación F4. Turbo Pascal ejecutará su programa hasta que se alcanza la línea de programa que contiene el cursor, en cuyo punto el control retorna al programador. Esta operación facilita saltar sobre bucles y otras secciones de código tediosas.

La traza de un programa (F7)

Turbo Pascal ofrece dos medios para realizar el seguimiento de su programa; el primer método, llamado *paso a paso* (stepping), ejecuta una única línea de código cada vez; si la línea llama a una subrutina, este método trata esa subrutina como si fuera una única sentencia; este método se denomina StepOver. Este método se activa pulsando la tecla de función F7. El segundo método, *traza* (tracing), es similar al método *paso a paso*, excepto que cuando una línea tiene una llamada a una subrutina, la subrutina se ejecuta línea a línea. Este método se activa pulsando F8 (Step).

La pulsación de F7 tiene el mismo efecto que pulsar F8, excepto cuando la linea contiene una llamada a subrutina. En este caso, F7 ejecuta la subrutina linea a linea, mientras que F8 ejecuta la sentencia completa incluyendo la función.

UNA SESIÓN DE DEPURACIÓN

Una vez que está familiarizado con el depurador integrado, es el momento de analizar prácticamente las tareas que lleva consigo la depuración de un programa.

Preparación del depurador

Antes de arrancar la depuración de un programa, el compilador debe ser capaz de generar la tabla de símbolos y la información de números de línea necesarios para sus programas. Las directivas de depuración del compilador `$L+` y `$D+` hacen estas tareas (por defecto están activadas). Estas directivas corresponden a las opciones de menú **Options | Compiler | Local Symbols** y **Options | Compiler | Debug Information**, respectivamente.

Una tabla de símbolos es una pequeña base de datos interna de todos los identificadores utilizados: constantes, tipos, variables, procedimientos e información de números de línea. `{$D+}` genera tablas de números de líneas que hace correspondencia del código objeto con el código fuente y `{$L+}` genera información local de depuración, que significa crear una lista de identificadores locales a cada procedimiento o función, de modo que el depurador puede «recordarlos» mientras está depurando. Si utiliza las directivas juntas, grabarlas de este modo: `{$D, L}`.

Comienzo de la sesión de depuración

El medio más rápido para comenzar la depuración es cargar el programa y ejecutar la orden **Run | Trace Into** o bien pulsar la tecla de función F7. El programa se compila y la barra de ejecución se situará en el primer **begin** del cuerpo principal del programa. A partir de este punto se puede ir sentencia a sentencia, pulsando F7, o bien pulsar F8, con lo que los procedimientos o funciones se ejecutan como si fueran una única sentencia, en lugar de sentencia a sentencia como es el caso cuando se pulsa F7.

Si se conoce previamente el lugar a donde se desea llegar con la comprobación, se puede utilizar F4 que ejecuta el programa hasta la posición actual del cursor y a continuación se detiene y hace una pausa.

Otra opción es establecer punto de interrupción. Lleve el cursor a la línea donde desea se detenga el cursor, a continuación seleccione **Debug | Toggle Breakpoint** o pulse CTRL-F8 y a continuación ejecute su programa (seleccione **Run | Run** o pulse CTRL-F9). Se pueden establecer varios puntos de ruptura en cuyo caso el programa se detendrá siempre que llega a cualquiera de dichos puntos.

Comenzar de nuevo una sesión de depuración

Si está enfrascado en la depuración de un programa y por cualquier causa desea volver a comenzar de nuevo la depuración, seleccione la orden **Program Reset (CTRL-F2)** del menú **Run**. Esta orden reinicializa la depuración en la primera línea del cuerpo principal de su programa y realiza también las siguientes operaciones: cerrar todos los archivos abiertos

por su programa, limpiar (borrar) la pila de cualquier llamada de subrutinas anidadas y libera cualquier espacio que esté utilizando el montículo.

Turbo Pascal ofrece otro método de reiniciar un programa si se hacen cambios durante la depuración. Por ejemplo, si se modifica alguna parte de un programa y se pulsa cualquier orden de ejecución (F7, F8, F4, CTRL-F9, etc.), se obtiene un cuadro de diálogo con el mensaje «*Source modified, rebuild (Y/N)*» (modificado fuente, reconstruir, sí/no). Si se pulsa Y, Turbo Pascal reconstruye su programa y comienza la depuración desde el principio. Si se pulsa N, Turbo Pascal supone que usted conoce lo que está haciendo y continúa la sesión de depuración.

Cualquier cambio en el código fuente *no afecta* a la ejecución del programa hasta que se vuelva a compilar.

Terminar la sesión de depuración

Mientras se está depurando un programa, Turbo Pascal sigue la pista de lo que está haciendo, y aunque puede cargar y editar archivos diferentes mientras está depurando, Turbo Pascal no interpreta la carga de un archivo distinto en el editor como finalización de la sesión de depuración. Así pues, si desea ejecutar o depurar un programa distinto, Turbo Pascal lo sabrá por la elección de la orden Run | Program Reset (CTRL-F2).

Examinando variables

Probablemente una de las cosas más interesantes que puede hacer el depurador integrado es poner una *observación* o *inspección* en una o más variables de su programa. Esta característica permite comprobar los valores de las variables a medida que progresá su programa sin necesidad de insertar sentencia de depuración extras en su código.

Arranque el programa con F7 y pulse cuantas veces lo requiera. Cuando desee añadir variables de observación, elija la orden Debug | Watches | Add Watch (CTRL-F7) para visualizar el cuadro Add Watch. Mediante la combinación adecuada de F7 y CTRL-F7 se pueden seguir fácilmente los valores de expresiones y variables a medida que progresá el programa.

OTRAS NORMAS DE DEPURACIÓN

Ahora que ha aprendido cómo utilizar el depurador, vamos a examinar otras propiedades y características que pueden facilitar la depuración.

Escribiendo programas idóneos para depuración

El depurador integrado sólo puede depurar una sentencia por línea. La unidad de ejecución en el depurador *es una linea* y no una sentencia. Si se tiene diferentes sentencias Pascal en una sola línea, éstas se ejecutarán juntas cuando se pulsa F7. Así, por ejemplo, Turbo Pascal permite escribir la siguiente linea de código:

```
t := 5 * z ; z := m DIV n; s := 4.25; n := 6;
```

Esto significa que si hubiera algún error en alguna de esas sentencias, el depurador integrado no será capaz de indicar cuál es la sentencia que produce el error. Por esta causa, sólo en el caso de que se tenga seguridad plena de que las sentencias son sintácticamente correctas no es conveniente poner diferentes sentencias en una línea, siendo mejor poner una sentencia en cada línea.

```
t := 5;
z := m DIV n;
s := 4.25;
r := 6;
```

Las declaraciones de variable se pueden organizar también de modo que aquellas que probablemente vaya a poner en la ventana *Watch* estén más próximas a la sentencia *begin* inicial del procedimiento o función. Cuando se detenga en ese procedimiento o función, se puede mover rápidamente el cursor a lo largo de la lista utilizando Add Watch (CTRL-F7) para añadir cada variable como una nueva observación (*Watch*).

De modo similar, si existen expresiones que desea examinar o evaluar, con frecuencia, en ciertos puntos de su programa, es conveniente insertarlos como comentarios. Cuando se alcance ese punto, se puede mover el cursor al comienzo de la expresión y copiarla en el cuadro *Add Watch* o *Evaluate and Modify*.

En cualquier forma, la mejor depuración siempre es la preventiva. Un programa bien diseñado y claramente escrito no sólo tendrá menos errores, sino que será más fácil fijar y localizar los que se produzcan. Algunas reglas interesantes a seguir en la escritura de programas son:

- Dividir los programas en módulos. Los módulos (unidades, procedimientos y funciones) deben tener alrededor de 25 líneas; si alguno tuviera más de un número, es conveniente romperlo de nuevo en procedimientos y funciones más pequeñas.
- Siempre que sea posible codificar, probar y depurar sus programas módulos a módulo, antes de integrarlos en dichos programas.
- Tratar de pasar la información sólo a través de parámetros, en lugar de utilizar variables globales. Esta acción evita los efectos laterales y también hace el código más fácil de depurar, ya que se puede examinar fácilmente toda la información que procede y sale de un procedimiento o función dada.
- Es más rentable crear un programa que funcione correctamente en lugar de tratar de hacerlo más rápido.

Requisitos de memoria

El depurador integrado requiere memoria y puede suceder que en el caso de grandes programas el espacio ocupado por el depurador puede necesitarse para programas. En consecuencia, es conveniente utilizar técnicas que permiten optimizar (ahorrar) la memoria.

- Eliminar programas resistentes en memoria RAM, tipo SideKick.
- Modificar CONFIG.SYS para eliminar controladores innecesarios (ANSI.SYS, cachés de disco, etc.) o reducir el número de archivos y memorias intermedias (FILES=20, BUFFERS=20).

- Reconfigurar Turbo Pascal.
 - Si se tiene instalada memoria extendida, asegúrese de que al menos están disponibles 64 K necesarios para el editor.
 - Si no se van a utilizar programas de gráficos, asegúrese que la opción *Graphics Screen Save* está desactivada.
 - Eliminar de TURBO.TPL cualquier unidad no utilizada.
- Organizar el programa mediante recubrimientos. El programa será más lento pero deja más memoria libre para la depuración.
- Al compilar el programa desactive las directivas de comprobación de errores del compilador, tales como *\$S* y *\$D*.

La directiva *{\$L-}* reduce el tamaño de la tabla de símbolos locales. La directiva *{\$D-}* reserva memoria y es conveniente utilizarla si no se necesita en depuración.

TRATAMIENTO DE ERRORES

Además del depurador integrado, Turbo Pascal proporciona diferentes directivas del compilador y características del lenguaje que ayudan a capturar (detectar) errores de programación. Algunas de estas características se describen a continuación.

Comprobación de errores de entrada/salida

Trate de ejecutar el programa Sumar e introduzca los valores 75 y 9z cuando se soliciten y a continuación pulse INTRO.

```
program sumar;
var
  a, b, Suma : Integer;
begin
  WriteLn ('Introduzca dos numeros :');
  ReadLn(a, b);
  Suma := a + b;
  WriteLn ('La suma es', Suma);
  ReadLn;
end.
```

Se obtiene un mensaje de error (el 106) y el cursor se posiciona en la sentencia.

```
ReadLn(a, b);
```

El error se ha producido debido a que se esperaban dos valores enteros y uno de ellos ha sido numérico (9z).

¿Cómo se puede evitar esta comprobación de tipos de datos? Turbo Pascal puede desactivar la verificación automática de errores de E/S; para ello, incluya la directiva de compilación *{\$I-}* en su programa.

Comprobación de rango

Otros errores típicos son aquellos que implican valores fuera de rango o de límites. Por ejemplo, casos típicos son: asignación de un valor demasiado grande a un valor entero, por ejemplo, 75534; asignar un valor a un índice de un array fuera del rango de definición.

La directiva `{$R+}` activa la verificación de fuera de rango. Sin embargo, en ocasiones se necesita violar los límites del rango bien en todo el programa o en segmentos de programa, por ejemplo cuando se trabaja con arrays asignados dinámicamente o con funciones que actúan sobre tipos enumerados como `Succ` y `Pred`. La verificación o comprobación de rangos se consigue situando `{$R+}` al principio del segmento de programa y `{$R-}` al final del mismo segmento. Esta operación suele ser la más empleada, por lo cual sólo se verifican aquellos segmentos de programas donde se requiera la comprobación de rango. Por ejemplo, en el bucle siguiente:

```

while Indice < 101 do
begin
  Indice := Indice+1;
  {$R+}                                activa verificación de rango
  if Lista [Indice] > 0 then
    Lista [Indice] := -1;
  {$R-}                                desactiva verificación de rango
end;

```

DEPURACIÓN ORIENTADA A OBJETOS

Borland ha extendido las propiedades del depurador integrado para soportar depuración orientada a objetos dentro del entorno integrado IDE.

La depuración orientada a objetos dentro del IDE no requiere ninguna preparación especial ni consideraciones especiales de código. El depurador integrado soporta totalmente objetos y elementos objetos que los trata de igual modo que si fueran procedimientos, registros y funciones. Los dos sistemas de depuración, paso a paso (*stepping*) y traza (*tracing*), se realizan mediante llamadas a métodos y examen de los datos objetos.

La traza

Las llamadas a los métodos se tratan por el depurador de modo igual que un procedimiento o función. Por consiguiente, la pulsación de la tecla de función F8 (StepOver) considera la llamada al método como una unidad indivisible y la ejecuta sin visualizar el código interno del método; mientras que F7 (Trace Into) carga el código del método, si está disponible, y sigue la traza a través de las sentencias del método.

Métodos estáticos versus virtuales

No existe diferencia entre el seguimiento o traza de las llamadas a los métodos estáticos y a los métodos virtuales.

Las llamadas a los métodos virtuales se resuelven en tiempo de ejecución, pero como la depuración sucede en tiempo de ejecución, no existe ambigüedad y el depurador integrador siempre conoce el método correcto que debe ejecutar a continuación.

La ventana Call Stack

La ventana Call Stack visualiza los nombres de los métodos con el prefijo del tipo objeto. *Nota:* el prefijo del método es el tipo objeto que define el objeto, no el nombre de la variable de la instancia objeto. Por ejemplo, la ventana Call Stack visualizará *ScrollBar.Init* en lugar de *Init* o *HScrol.Init*.

La ventana Evaluate

Normalmente, cuando un nombre de objeto se especifica en la ventana *Evaluate*, sólo se visualizan los campos datos. En la ventana *Evaluate*, los campos de los datos objetos se visualizarán esencialmente en el mismo formato que un registro fecha. Se aplican los mismos especificados del formato registro y todas las extensiones que son válidas para registros son aceptadas para objetos.

Cuando un nombre específico de método —bien estático o virtual— se evalúa, un valor puntero se visualiza indicando la dirección del código del método.

Dentro del EID, el depurador puede también evaluar u observar el parámetro *Self* y puede seguirse la traza con especificaciones de formato y campo o calificadores de método.

La ventana Watch

Cualquier objeto puede ser asignado a una ventana **Watch**, al igual que un registro todos los calificadores que son válidos para registros son válidos para objetos.

La orden Find Procedure

En Turbo Pascal 6.0 la orden Find Procedure del menú Debug permite la entrada de expresiones que evalúan a una dirección dentro del segmento del código correspondiente. Esto se aplica a variables y parámetros procedurales, así como a métodos objeto.

Mensajes y códigos de error

Como ya conoce el lector, existen tres tipos de errores: *errores de sintaxis o de compilación, errores de ejecución y errores lógicos*.

Errores de compilación (sintaxis)

El formato de los mensajes de error en compilación es

Error 16: Type Identifier expected

Errores en tiempo de ejecución

Algunos errores en el momento de la ejecución hacen que el programa visualice un mensaje de error y terminan.

Run-time error nnn at xxxx:yyyy

nnn número del error
 xxxx:yyyy dirección del error (segmento y desplazamiento)

Los errores en ejecución se dividen en cuatro categorías:

- *errores DOS*: 1-99
- *errores de E/S (I/O)*: 100-149
- *errores críticos*: 150-199
- *errores fatales*: 200-255

Lista de errores de compilación

Turbo Pascal 5.0 a 7.0

1. **Out of memory** (Falta memoria).
 - Si Compile/Destination se pone a Memory, póngalo a Disk en el entorno integrado.

- Si Options/Linker/Link Buffer en el entorno integrado se pone a Memory, fijarla a Disk. Utilizar la opción /L para enlazar al disco en el compilador de línea de órdenes.
 - Si está utilizando alguna utilidad (programa) residente en memoria, tal como SideKick, elimínelos de la memoria.
 - Si está utilizando TURBO.EXE, trate de utilizar en su lugar TPC.EXE.
 - Si ninguna solución anterior elimina el mensaje de error, su programa o unidad son —sencillamente— demasiado grandes para compilarlos con la memoria disponible y tendrá que romperlo en dos o más unidades, más pequeñas.
2. **Identifier expected** (Se esperaba un identificador).
 3. **Unknown identifier** (Identificador desconocido).
 4. **Duplicate identifier** (Identificador duplicado).
 5. **Syntax error** (Error de sintaxis).
 6. **Error in real constant** (Error en constante entera).
 7. **Error in integer constant** (Error en constante entera).
 8. **String constant exceeds line** (Constante de cadena excede la línea).
 9. **Too many nested files** (Demasiados archivos anidados).
 10. **Unexpected end of file** (No se esperaba fin de archivo).
 11. **Line too long** (Línea demasiado larga).
 12. **Type identifier expected** (Se esperaba identificador tipo).
 13. **Too many open files** (Demasiados archivos abiertos).
 14. **Invalid file name** (Nombre de archivo no válido).
 15. **File not found** (Archivo no encontrado).
 16. **Disk full** (Disco lleno).
 17. **Invalid compiler directive** (Directiva del compilador no válida).
 18. **Too many files** (Demasiados archivos).
 19. **Undefined type in pointer definition** (Tipo sin definir en la definición del puntero).
 20. **Variable identifier expected** (Se esperaba identificador de variable).
 21. **Error in type** (Error de tipo).
 22. **Structure too large** (Estructura demasiado grande).
 23. **Set base type out of range** (Tipo base de conjunto fuera de rango).
 24. **File components may not be files** (Existen componentes de archivo que pueden no ser archivos).
 25. **Invalid string length** (Longitud de cadena no válida).
 26. **Type mismatch** (Tipos no coinciden).
 27. **Invalid subrange base type** (Tipo base de subrango inválido).
 28. **Lower bound greater than upper bound** (Límite inferior mayor que el superior).
 29. **Ordinal type expected** (Se esperaba tipo ordinal).
 30. **Integer constant expected** (Se esperaba constante entera).
 31. **Constant expected** (Se esperaba constante).

32. **Integer or real consant expected** (Se esperaba constante real o entera).
33. **Type identifier expected** (Se esperaba identificador de tipo).
34. **Invalid function result type** (Tipo resultado de función no válido).
35. **Label identifier expected** (Se esperaba identificador de etiqueta).
36. **BEGIN expected** (Se esperaba BEGIN).
37. **END expected** (Se esperaba END).
38. **Integer expression expected** (Se esperaba expresión entera).
39. **Ordinal expression expected** (Se esperaba expresión ordinal).
40. **Boolean expression expected** (Se esperaba expresión booleana).
41. **Operand types do not match operator** (Tipos del operando no coinciden con operador).
42. **Error in expression** (Error en expresión).
43. **Illegal assignment** (Asignación ilegal).
44. **Field identifier expected** (Se esperaba identificador de campo).
45. **Object file too large** (Archivo objeto demasiado grande).
46. **Undefined external** (No se definió procedimiento external).
47. **Invalid objet file record** (Registro de archivo objeto no válido).
48. **Code segment too large** (Segmento de código demasiado grande).
49. **Data segment too large** (Segmento de datos demasiado grande).
50. **DO expected** (Se esperaba DO).
51. **Invalid PUBLIC definition** (Definición PUBLIC no válida).
52. **Invalid EXTRN definition** (Definición EXTRN no válida).
53. **Too many EXTRN definition** (Demasiadas definiciones EXTRN).
54. **OF expected** (Se esperaba OF).
55. **INTERFACE expected** (Se esperaba INTERFACE).
56. **Invalid relocatable reference** (Referencia reasignable no válida).
57. **THEN expected** (Se esperaba THEN).
58. **TO or DOWNT0 expected** (Se esperaba TO o DOWNT0).
59. **Undefined forward** (Forward no definido).
60. **Too many procedures** (Demasiados procedimientos).
61. **Invalid typecast** (Typecast no válido).
62. **Division by zero** (División por cero).
63. **Invalid file type** (Tipo de archivo no válido).
64. **Cannot Read or Write variables of this types** (No se pueden leer o escribir variables de este tipo).
65. **Pointer variable expected** (Se esperaba variable de puntero).
66. **String variable expected** (Se esperaba variable de cadena).
67. **String expression expected** (Se esperaba expresión de cadena).
68. **Circular unit reference** (Referencia de unidad circular).
69. **Unit name mismatch** (No coincide nombre de unidad).
70. **Unit version mismatch** (No coincide versión de unidad).
71. **Duplicate unit name** (Nombre de unidad duplicado).
72. **Unit file format error** (Error en formato de archivo de unidad).
73. **Implementation expected** (Se esperaba implementación).

74. **Constant and case types do not match** (No coinciden tipos de case y constantes).
75. **Record variable expected** (Se esperaba variable de archivo).
76. **Constant out of range** (Constante fuera de rango).
77. **File variable expected** (Se esperaba variable de archivo).
78. **Pointer expression expected** (Se esperaba expresión de puntero).
79. **Integer or real expression expected** (Se esperaba expresión real o entera).
80. **Label not within current block** (La etiqueta no está dentro del bloque actual).
81. **Label already defined** (Etiqueta ya definida).
82. **Undefined label in preceding statement part** (Etiqueta sin definir en la parte anterior de la sentencia).
83. **Invalid @ argument** (Argumento @ no válido).
84. **UNIT expected** (Se esperaba UNIT).
85. **";" expected** (Se esperaba ";").
86. ":" expected (Se esperaba ":").
87. **"," expected** (Se esperaba ",").
88. **"(" expected** (Se esperaba "(").
89. **")" expected** (Se esperaba ")").
90. **"=" expected** (Se esperaba "=").
91. **";=" expected** (Se esperaba ";=").
92. "[" or "(" expected (Se esperaba "[" o "(").
93. "]" or ")" expected (Se esperaba "]" o ")").
94. **"." expected** (Se esperaba ".").
95. **".." expected** (Se esperaba "..").
96. **Too many variables** (Demasiadas variables).
97. **Invalid FOR control variable** (Variable de control FOR no válida).
98. **Integer variable expected** (Se esperaba variable entera).
99. **Files are not allowed here** (No se permiten archivos aquí).
100. **String length mismatch** (No coincide la longitud de cadena).
101. **Invalid ordering of fields** (Orden de campos no válido).
102. **String constant expected** (Se esperaba constante de cadena).
103. **Integer or real variable expected** (Se esperaba variable real o entera).
104. **Ordinal variable expected** (Se esperaba variable ordinal).
105. **INLINE error** (Error INLINE).
106. **Character expression expected** (Se esperaba expresión de carácter).
107. **Too many relocation items** (Demasiados elementos de reasignación).
112. **CASE constant out of range** (Constante CASE fuera de rango).
113. **Error in statement** (Error en sentencia).
114. **Cannot call an interrupt procedure** (No se puede invocar un procedimiento de interrupción).
116. **Must be in 8087 mode to compile this** (Debe fijarse el modo 8087 para poderse compilar).
117. **Target address not found** (No se encuentra dirección de destino).
118. **Include files are not allowed here** (No se permiten archivos de inclusión).
120. **NIL expected** (Se esperaba NIL).

121. **Invalid qualifier** (Calificador no válido).
122. **Invalid variable reference** (Referencia de variable no válida).
123. **Too many symbols** (Demasiados símbolos).
124. **Statement part too large** (Parte de sentencia demasiado grande).
125. **Files must be var parameters** (Los archivos deben ser parámetros var).
127. **Too many conditional symbols** (Demasiados símbolos condicionales).
128. **Misplaced conditional directive** (Directiva condicional mal colocada).
129. **ENDIF directive missing** (No se encuentra directiva ENDIF).
130. **Error in initial conditional defines** (Error al definir condiciones iniciales).
131. **Header does not match previous definition** (La cabecera no coincide con la definición previa).
132. **Critical disk error** (Error crítico en el disco).
133. **Cannot evaluate this expression** (No se puede evaluar esta expresión).
134. **Expression incorrectly terminated** (Expresión terminada incorrectamente).
135. **Invalid format specifier** (Especificador de formato no válido).
136. **Invalid indirect reference** (Referencia indirecta no válida).
137. **Structured variables are not allowed here** (No se permiten aquí las variables de estructura).
138. **Cannot evaluate without System unit** (No se puede evaluar sin la unidad Sistema).
139. **Cannot access this symbol** (No se puede acceder a este símbolo).
140. **Invalid floating-point operation** (Operación de punto flotante no válida).
141. **Cannot compile overlays to memory** (No se pueden compilar recubrimientos a la memoria).
142. **Procedure or function variable expected** (Se esperaba variable de función o procedimiento).
143. **Invalid procedure or function reference** (Referencia de función o procedimiento no válidos).
144. **Cannot overlay this unit** (No se puede recubrir esta unidad).

Turbo Pascal 6.0/7.0

146. **File access denied** (Acceso a archivo denegado).
147. **Object type expected** (Se esperaba tipo de objeto).
148. **Local object types are not allowed** (No se permiten tipos de objeto local).
149. **VIRTUAL expected** (Se esperaba VIRTUAL).
150. **Method identifier expected** (Se esperaba identificador de método).
151. **Virtual constructors are not allowed** (No se permiten constructores virtuales).
152. **Constructor identifier expected** (Se esperaba identificador constructor).
153. **Destructor identifier expected** (Se esperaba identificador destructor).
154. **Fail only allowed within constructors** (Solamente se permite el fallo dentro de los constructores).
155. **Invalid combination of opcode and operands** (Combinación no válida del código de operación y operativo).
156. **Memory reference expected** (Referencia esperada a memoria).

157. **Cannot add or subtract relocatable symbols** (No se puede sumar o restar símbolos reubicables).
158. **Invalid register combination** (Combinación no válida de registros).
159. **286/287 instructions are not enabled** (Instrucciones 286/287 no están activas).
160. **Invalid symbol reference** (Referencia no válida a símbolos).
161. **Code generation error** (Error en la generación de código).
162. **ASM expected** (ASM esperado).

Lista de errores en tiempo de ejecución (*Run-time Errors*)

Turbo Pascal 5.5./6.0/7.0

Errores DOS

1. **Invalid function number** (Número no válido de función).
2. **File not found** (Archivo no encontrado).
3. **Path not found** (Vía de acceso no encontrada).
4. **Too many open files** (Demasiados archivos abiertos).
5. **File access denied** (Acceso al archivo negado).
6. **Invalid file handle** (Manipulación de archivo no válida).
12. **Invalid file access code** (Código de acceso al archivo no válido).
15. **Invalid drive number** (Número de unidad no válido).
16. **Cannot remove current directory** (No se puede eliminar el directorio actual).
17. **Cannot rename across drives** (No se puede renombrar entre unidades).
18. **No more files** (No más archivos). (Sólo versión 7.0.)

Errores de E/S (Entrada/Salida)

Estos errores producen la terminación del programa si la sentencia correspondiente fue compilada en el estado **{\$I+}** de la directiva **I**. El programa continúa su ejecución, en el estado **{\$I-}**, y el error se puede consultar en la función **IOResult**.

100. **Disk read error** (Error de lectura de disco).
101. **Disk write error** (Error de escritura de disco).
102. **File not assigned** (Archivo no asignado).
103. **File not open** (Archivo no abierto).
104. **File not open for input** (Archivo no abierto para entrada).
105. **File not open for output** (Archivo no abierto para salida).
106. **Invalid numeric format** (Formato numérico no válido).

Errores críticos

150. **Disk is write-protected** (Protegido contra escritura).
151. **Unknown unit** (Unidad desconocida).

- 152. **Drive not ready** (Unidad no preparada).
- 153. **Unknown command** (Orden desconocida).
- 154. **CRC error in data** (Error CRC en datos).
- 155. **Bad drive request structure length** (Longitud de estructura solicitada en unidad errónea).
- 156. **Disk seek error** (Error en búsqueda de disco).
- 157. **Unknown media type** (Tipo de soporte desconocido).
- 158. **Sector not found** (Sector no encontrado).
- 159. **Printer out of paper** (Falta papel en la impresora).
- 160. **Device write fault** (Fallo de escritura en dispositivo).
- 161. **Device read fault** (Fallo de lectura en dispositivo).
- 162. **Hardware failure** (Fallo en hardware).

Errores fatales

- 200. **Division by zero** (División por cero).
- 201. **Range check error** (Error en comprobación de rango).
- 202. **Stack overflow error** (Error de desbordamiento de pila).
- 203. **Heap overflow error** (Error de desbordamiento de pila).
- 204. **Invalid pointer operation** (Operación de puntero no válida).
- 205. **Floating point overflow** (Desbordamiento de coma flotante).
- 206. **Floating point underflow** (...).
- 207. **Invalid floating point operation** (Operación de coma flotante no válida).
- 208. **Overlay manager not installed** (Manipulador de recubrimiento no instalado).
- 209. **Overlay file read error** (Error de lectura en archivos de recubrimiento).
- 210. **Object not initialized** (Objeto no inicializado). (*Sólo versiones 6.0 y 7.0.*)
- 211. **Call to abstract method** (Llamada a método abstracto). (*Sólo versiones 6.0 y 7.0.*)
- 212. **Stream registration error** (Error en registro de flujo *stream*). (*Sólo versiones 6.0 y 7.0.*)
- 213. **Collection Index out of range** (Índice de Collection fuera de rango). (*Sólo versiones 6.0 y 7.0.*)
- 214. **Collection overflow error** (Error de desbordamiento de Collection). (*Sólo versiones 6.0 y 7.0.*)
- 215. **Aritmetic overflow error** (Error aritmético de desbordamiento). (*Sólo versión 7.0.*)

Manipulación de errores de entrada/salida (E/S)

Errores de *entrada/salida (E/S)* son aquellos que implican la E/S en o desde un programa, archivo o dispositivo. Si el usuario introduce entradas en un formato incorrecto, o si sucede cualquier otro error de E/S, su programa se terminará. Esta situación se puede plantear también, aunque no se produzcan errores; por ejemplo, el usuario pide abrir un archivo que no existe utilizando *reset*.

Errores comunes son:

- Entrada de un dato de tipo inadecuado por teclado durante la ejecución de un programa.
- Intento de lectura o escritura en un archivo no abierto.
- Intento de apertura para lectura (*reset*) de un archivo que no existe en disco.

En lugar de terminar el programa cuando se produce un error de E/S, es preferible que se realice alguna acción en función del error cometido; es decir, tratar adecuadamente el mismo. Turbo Pascal posee la directiva **I** que se utiliza para desactivar la verificación de errores.

El formato **{\$I-}**, situado en cualquier punto de un programa, evita que un programa se termine cuando se producen errores de E/S (desactiva la verificación de errores). La directiva **{\$I+}** activa la verificación de errores.

Dado que es inútil proseguir con un programa cuando se produce un error de E/S, la mejor solución es bifurcar a alguna parte del programa. Esta tarea se facilita mediante la función **IOResult**.

Función IOResult

Devuelve un valor entero que es el estado de la última operación ejecutada de E/S (número de error). Sin embargo, lo que se necesita es saber si ha ocurrido o no un error, y eso se puede hacer verificando *IOResult* para ver si es cero o no cero (cero si no se produce ningún error de E/S).

- Si *IOResult* es cero, no se ha producido ningún error y el programa prosigue con normalidad.
- Si *IOResult* devuelve un valor distinto de cero, entonces el programa debe bifurcar para manipular el error. Después de cada llamada a *IOResult*, el valor de *IOResult* se pone a cero.

Ejemplo

Verificación de escritura correcta en nombre de archivos.

```
repeat
  WriteLn ('Introduzca nombre de archivo:');
  {$I- desactivar verificación de errores}
  ReadLn (NombreArch);
  Assign (NombreVar, NombreArch);
  Reset (NombreVar);
  {$I+ activar errores}
  Error := IOResult;
  if Error <> 0 then
    begin
      WriteLn (NombreArch, 'no se puede encontrar');
      WriteLn ('Pruebe de nuevo');
    end
  until Error=0;
```

Guía de referencia Turbo Borland Pascal

LENGUAJE

Palabras reservadas			
and	file	not	then
array	for	object	to
asm	function	of	type
begin	goto	or	unit
case	if	packed	until
const	implementation	procedure	uses
constructor	in	program	var
destructor	inherited*	record	while
div	inline	repeat	with
do	interface	set	xor
downto	label	shl	
else	mod	shr	
end	nil	string	
Directivas (se pueden redefinir)			
absolute	far	near	virtual
assembler	fotward	private	
external	interrupt	public*	
* Sólo en versión 7.0.			

Identificadores

WriteLn	Exit	CadenaReal	System.MemAvail	Rosas
Un_mes	Nombre_Apellidos		Dia_del_Mes	Pedro_uno

Cadenas de caracteres

'Turbo'	(Turbo)
....	{'}

```
.. {cadena nula}
.. {espacio}
```

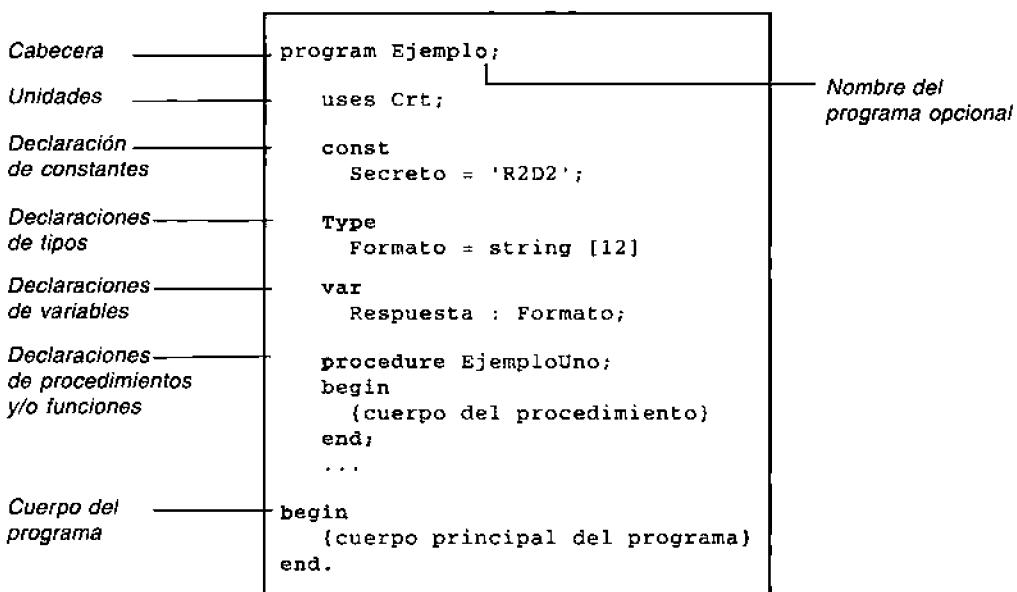
Comentarios

{Cualquier texto encerrado entre llaves}
(* Es también un comentario válido*)

Línea de programa

Las líneas de programas Turbo Pascal tienen una longitud máxima de 126 caracteres.

Estructura de un programa



PROCEDIMIENTOS Y FUNCIONES

Cabecera procedimiento

```
procedure identificador (lista de parámetros);
    (declaraciones de datos)
begin
    (cuerpo del procedimiento)
end;
```

```

cabecera          function identificador (lista de parámetros);
función           {declaraciones de datos}
begin             begin
                  {cuerpo de la función}
end;

```

UNIDADES

Unidades estándar

<i>Crt</i>	Soporte de pantalla y teclados
<i>Dos, WinDOS</i>	Funciones de propósito general DOS
<i>Graph</i>	Rutinas gráficas
<i>Graph3</i>	Gráficos compatibles versión 3
<i>Overlay</i>	Implementa el administrador de solapamientos
<i>Printer</i>	Acceso a la impresora
<i>Strings</i>	Tratamiento de cadenas terminadas en nulo
<i>System</i>	Rutinas de la biblioteca en tiempo de ejecución
<i>Turbo3</i>	Mantener compatibilidad con Turbo Pascal 3.0

Sintaxis de la unidad

```

unit identificador
interface lista de unidades;                      opcional
  {declaraciones públicas más}
  {cabeceras de todas las subrutinas públicas}
implementation
uses lista de unidades;                          opcional
  {declaraciones privadas}
  {implementación de procedimientos y funciones}
begin
  {código de inicialización}                      opcional
end.

```

DISPOSITIVOS

<i>CON</i>	Terminal
<i>PRN</i>	Impresora
<i>AUX</i>	Dispositivo auxiliar
<i>LPT1</i>	Impresora
<i>NUL</i>	Dispositivo nulo

TIPOS DE DATOS

Tipos simples

Boolean (lógicos) true..false

Números enteros

<i>Byte</i>	0..255
<i>Integer</i>	-32.768..32.767
<i>LongInt</i>	2.147.483.648..2.147.483.647
<i>ShortInt</i>	-128..127
<i>Word</i>	0..65.535

Números reales

<i>Real</i>	$2.9 \times 10^{-39}..1.7 \times 10^{38}$
<i>Single</i>	$1.5 \times 10^{-45}..3.4 \times 10^{38}$
<i>Double</i>	$5.0 \times 10^{-324}..1.7 \times 10^{308}$
<i>Extended</i>	$3.4 \times 10^{-4932}..1.1 \times 10^{4932}$
<i>Comp</i>	$-2^{63}+1..2^{63}-1$

Carácter

Char Cualquier carácter ASCII

Cadenas

String Secuencia de hasta 255 caracteres ASCII. Si no se especifica un tamaño, se utilizará por defecto 255.

Punteros

Pointer Dirección de un elemento dato, procedimiento o función.

Tipos subrango

0..99
-128..127
Lunes..Domingo

Tipos enumerados

```
type
  Naipes = {Oros, Espadas, Bastos, Copas};
```

Tipos estructurados

```

Array      array [1..100] of Real
            array [Boolean] of array [1..10] of array [Longitud] of Real;
            packed array [1..10, 1..15] of Boolean

Registro   type Reg = record
                Nombre : String [40];
                Edad : Integer;
                Salario : Real;
            end;

Objeto     type Punto = object
                x,y : Integer;
            end;

Conjunto   Letras := ['A','B','C','D','E']
            NoLetras := []

Puntero    type PunteroReal = ^Real;
            New (P);

Pchar      type Pchar = ^Char;

```

Tipos procedimiento

```

type
  Proc      = procedure;
  InterX   = procedure (var X, Y : Integer);
  ProcCX   = procedure (S : string);
  FuncUna  = function (X : Real) : Real;

```

VARIABLES Y CONSTANTES

Constantes predefinidas

true	<i>verdadero</i>
false	<i>falso</i>
maxint	<i>entero mayor disponible en máquina</i>

Declaración de variables

```

var
  x,y,z : Real;
  i,j,k : Integer;
  Dígito : 0..9;

```

Variables absolutas

```

var
  Cad : string [40];
  LongCad : Byte absolute Cad;

```

Variables predefinidas

<i>Variable</i>	<i>Tipo</i>
input	Archivo Text
Output	Archivo Text
Exitcode	Integer
FileMode	Byte
InOutRes	Integer
RandSeed	LongInt
StackLimit	Word

Constantes con tipos

Tipo simple

```
const
  Maximo : Integer = 425;
  Factor : Real = -12.5;
const
  Min : Integer = 0;
  Max : Integer = 100;
type declaración no válida
  Lista = array [Min..Max] of Integer;
la declaración Lista no es válida ya que Min y Max son constantes de tipos
```

Tipo cadena

```
const
  Cabecera : string[10] = 'Ciudades';
  Nueva Linea : string[2] = #13#10;
  Respuesta : string[5] = 'Si';
```

Tipo estructurado

```
const
  Digits : array [0..9] of Char =
    ('0','1','2','3','4','5','6','7','8','9');
const
  Digits : array [0..9] of Char = '01234567890';
type
  Matriz = array [0..1, 0..1, 0..1] of Integer;
const
  Lista : Matriz = ((0,1), (2,3)), ((4,6), (7,9)));
```

Tipo registro

```
type
  Punto = record
    x,y : Real;
  end;
  Vector = array [0..1] of Punto;
```

```
const
  Origen : Punto = (X:0.0; Y:0.0);
  Linea  : Vecto = ((X:-4.5; Y:1.5), (X:6.4; Y:5.0));
```

Tipo objeto

```
type
  Punto = object
    x,y : Integer;
  end;
const
  Origen : Punto = (x:0; y:0);
```

Tipo conjunto

```
type
  Digitos = set of 0..9;
  Letras  = set of 'A'..'Z';
const
  Pares   : Digitos = [0,2,4,6,8];
  Vocales : Letras  = ['A','E','I','O','U'];
```

Tipo puntero

```
type
  Direccion = (Izda, Dcha, Arriba, Abajo);
  PNodo = ^NodoT;
  NodoT = record
    ...
  end;
const
  S1 : string[4] = 'ABAJO';
  N1 : NodoT = (Siguiente : nil; Symbolo : @S1; Valor : Abajo);
```

Tipo procedimiento

```
type
  ProcError = procedure (CodigoError : Integer);

procedure ErrorPorOmission (CodigoError : Integer); far;

begin
  Writeln ('Error', CodigoError, ',');
end;

const
  ManipuladorErrores : ProcError = ErrorPorOmission;
```

OPERADORES

Aritméticos

Operador	Sintaxis	Significado
+	+expresión	Positivo (unitario)
+	expresión1 + expresión2	Suma (binaria)
-	-expresión	Negativo (unitario)
-	expresión1 - expresión2	Resta (binaria)
*	expresión1 * expresión2	Multiplicación
/	expresión1 / expresión2	División real
DIV	expresión1 DIV expresión2	División entera
MOD	expresión1 MOD expresión2	Resto (módulo)

Lógicos

Operador	Sintaxis	Significado
+	NOT expresión	Complemento
AND	expresión1 AND expresión2	AND
OR	expresión1 OR expresión2	OR inclusiva
XOR	expresión1 XOR expresión2	OR exclusiva

Relacionales

Operador	Sintaxis	Devuelve verdadero si:
=	expresión1 = expresión2	Expresiones son iguales
< >	expresión1 < > expresión2	Expresiones no son iguales
<	expresión1 < expresión2	expresión1 es menor que expresión 2
<=	expresión1 <= expresión2	expresión1 es menor que o igual a expresión2
>	expresión1 > expresión2	expresión1 es mayor que expresión2
>=	expresión1 >= expresión2	expresión1 es mayor o igual que expresión2

De conjunto

Operador	Sintaxis	Devuelve verdadero si:
=	Conjunto1 = Conjunto2	Conjunto1 y Conjunto2 son idénticos Cada elemento de Conjunto1 está contenido en Conjunto2, y cada elemento en Conjunto1 está contenido en Conjunto1
< >	Conjunto1 < > Conjunto2	Uno de los conjuntos contiene al menos un elemento que no está en el otro conjunto

Operador	Sintaxis	Devuelve verdadero si:
<code><=</code>	<code>Conjunto1 <= Conjunto2</code>	Cada elemento del Conjunto1 está también en Conjunto2
<code>>=</code>	<code>Conjunto1 >= Conjunto2</code>	Cada elemento de Conjunto2 está también en Conjunto1
<code>IN</code>	<code>elemento IN Conjunto2</code>	El elemento elemento se encuentra en Conjunto1
<code>-</code>	<code>Conjunto1 - Conjunto2</code>	Diferencia
<code>+</code>	<code>Conjunto1 + Conjunto2</code>	Unión
<code>*</code>	<code>Conjunto1 * Conjunto2</code>	Intersección

Dirección

@ Dirección de variable, procedimiento o función.

Concatenación

+ Concatena dos cadenas.

Prioridad de operadores

Prioridad	Operadores
1 (alta)	<code>@ NOT unitario + -</code>
2	<code>*/DIV, MOD, AND, SHL, SHR</code>
3	<code>binario +, -, OR, XOR</code>
4 (baja)	<code>=, <>, <>, <=, >=, IN</code>

SENTENCIAS

Asignación

`<identificador> := <expresión>`

```
rango := alto - bajo;
cuenta := cuenta + 1;
```

Compuesta

`begin <sentencias> end`

```
begin
  z := 5;
  GetReal ('Valor', ValorReal);
  WriteLn (ValorReal);
end
```

Selectiva (if)

```
if <condición> then
  <sentencia>
[ else
  <sentencia> ]:
```

```
if Cad1 < > Cad2 then
begin
  Cad1 := Cad1 + Cad2;
  writeln ('nueva cadena ', Cad1);
end;
```

Selectiva (if-anidada)

```
if (condición) then
  <sentencia>
else if (condición) then
  <sentencias>
else if (condición) then
  <sentencia>
else
  <sentencia>
```

```
if x >y then
  if x>z then
    Write (x)
  else
    Write (z)
else
  if y>z then
    Write (y)
  else
    Write (z);
writeln ('es el Mayor');
```

Selectiva múltiple (case)

```
case <selector> of
  <lista de constantes>;
    <sentencias 1>;
  <lista de constantes>;
    <sentencias 2>;
    ...
[else
  <sentencia por defecto>]
end;
```

```
case N of
  1,2      : writeln ('Primero');
  3,4,5   : writeln ('Bronce');
  7       : writeln ('Fin');
  else
    writeln ('Fin');
end;
```

Sentencia repetitiva for

1. **for <indice> := <valor inicial> to <valor final> do <sentencia>**

```
for k := 1 to 10 do          for j := 1 to 10 do
  writeln ('cuerpo de bucle'); begin
                           j := j + z;
                           writeln ('pasos ;', pasos : 2);
                           paso := paso + 1
                         end;
                         writeln ('fuera del bucle');
```

2. **for <indice> := <valor final> downto <valor inicial> do <sentencia>**

```
for i := 10 downto 1
begin
  Resultado := i * Resultado;
  WriteLn ('Bucle', i, Resultado);
end;
```

Sentencia repetitiva *while*

while <condición> do	paso := 1;
<sentencia>;	while paso <= 10 do
<sentencia> = sentencia simple	begin
sentencia compuesta	WriteLn ('Pasos:', paso:2);
	paso := paso + 1
	end;
	WriteLn('fuea del bucle');
	While ch < > ' ' do Read(ch);

Sentencia repetitiva *repeat*

repeat	paso := 1;
<sentencia>	repeat
until <condición>	WriteLn('Pasos:',paso:2);
	paso := paso + 1
	until paso > 10
	WriteLn('Futura del bucle');

ÁMBITO Y LOCALIDAD

El *ámbito* de un identificador es el conjunto de módulos (programas o subprogramas) en el que está legalmente declarado el identificador.

```
program Principal
```

```
var
  w,x,y,z : ...;
procedure Uno (var w:...; a:...);
  var
  x : ... ;
begin
  ...
end;
```

variables globales
parámetros locales w,a,x

```

procedure Dos ( c:...; var x... );
var
  z,w : ... ;
begin
  ...
end;

begin {Principal}
  ...
  Uno (y,z);
  ...
  Dos (w,y,x);
  ...
end;

```

Para eliminar posibles efectos laterales, evite utilizar variables globales en subprogramas o identificadores idénticos para cantidades diferentes.

PROCEDIMIENTOS Y FUNCIONES ESTÁNDAR

Turbo Pascal contiene procedimientos y funciones estándar (incorporadas o predefinidas) y variables predeclaradas en la unidad *System*. Todos ellos están declarados en la unidad *System*, en consecuencia no necesita ninguna sentencia *uses* cuando desee utilizar alguno/a de ellos/a.

Funciones aritméticas

Nombre	Sintaxis	Descripción
<i>Abs</i>	<i>Abs (x);</i>	Devuelve el valor (positivo) absoluto de su argumento.
<i>ArcTan</i>	<i>ArcTan (x:Real):Real;</i>	Arco tangente expresado en radianes.
<i>Cos</i>	<i>Cos(x:Real):Real;</i>	Coseno del argumento en radianes.
<i>Exp</i>	<i>Exp(x:Real):Real;</i>	Potencia exponencial del argumento (e^x).
<i>Frac</i>	<i>Frac(x:Real):Real;</i>	Parte decimal de un número real.
<i>Int</i>	<i>Int(x:Real):Real;</i>	Parte entera del argumento.
<i>Ln</i>	<i>Ln(x:Real):Real;</i>	Logaritmo neperiano (base <i>e</i>) del argumento.
<i>Pi</i>	<i>Pi:Real;</i>	Valor de <i>Pi</i> (3.1415926535897932385).
<i>Sin</i>	<i>Sin(x:Real):Real;</i>	Seno del argumento.
<i>Sqr</i>	<i>Sqr(x);</i>	Cuadrado del argumento.
<i>Sqrt</i>	<i>Sqrt(x:Real):Real;</i>	Raíz cuadrada del argumento.

```

y := abs (x)*2;
z := ArcTan(1.75);
Tan := Sin (X)/cos(x);
Pot := Exp(3);
R := Frac(-245.123);
z := Int(345.678);
t := Ln(1.25);
z := ArcTan(Pi);
z := Sin(Pi);
f := Sqr(3.45);
f := Sqrt(1.2345);

```

Funciones de transferencia

Nombre	Sintaxis	Descripción
Chr	Chr (x:Byte):Char;	Devuelve el carácter correspondiente al código ASCII.
Ord	Ord(x):LongInt;	Número ordinal de un tipo ordinal.
Round	Round(x:Real):LongInt;	Redondea un valor real a entero largo.
Trunc	Trunc(x:Real):LongInt;	Trunca un valor de tipo real a entero.

Ejemplos

Write(Chr(i));	Round(5.449)	devuelve 5
Ord('A')	Trunc(-3.14)	devuelve -3
	Trunc(6.5)	devuelve 6

Procedimientos de flujo de control

Nombre	Sintaxis	Descripción
Break	Break;	Termina una sentencia for, while o repeat.
Continue	Continue;	Contorna con la siguiente iteración de una sentencia for, while o repeat.
Exit	Exit;	Termina inmediatamente el bloque actual (procedimiento, función o programa).
Halt	Halt[(CódigoSalida:Word)];	Detiene la ejecución del programa y retorna al sistema operativo.
RunError	RunError[(CódigoError:Byte)];	Detiene la ejecución del programa y genera un error en tiempo de ejecución.

Ejemplos

```
if t='s' then Exit;
if p = nil then
  RunError(204);
```

Procedimientos o funciones ordinales

Nombre	Sintaxis	Descripción
Dec	Dec (var x[:n:LongInt]);	Decrementa una variable.
Inc	Inc(var x[:n:LongInt]);	Incrementa una variable.
High*	High(x);	Devuelve el valor más alto en el rango del argumento que debe ser un tipo array o un tipo cadena.
Low*	Low(x);	
Odd	Odd(x:LongInt):Boolean;	Devuelve el valor más bajo en el rango del argumento (tipo array o cadena).
Pred	Pred(x);	Predecesor del argumento (x de tipo ordinal).
Succ	Succ(x);	Sucesor del argumento (x de tipo ordinal).

Sólo está implementado en la versión 7.0.

Ejemplos

Tratamiento de cadenas

Procedimientos

Nombre	Sintaxis	Descripción
Delete	Delete(var s:string; Pos,Len:Integer);	Borra una subcadena a partir de una posición en una cadena.
Insert	Insert(var s:string; var D: string;Pos:Integer);	Inserta una subcadena en una posición de una cadena.
Str	Str(I:Integer; var s:string); Str(R:Real; var s:string);	Convierte un valor numérico a cadena.
Val	Val(s:string; var R:Real,P: Integer); Val(s:string; var I,P:Integer);	Convierte una cadena a su valor numérico.

Funciones

Nombre	Sintaxis	Descripción
Concat	Concat(s1,s2,...,sn:string); string;	Concatena (une) cadenas.
Copy	Copy(S:string;Pos,Long:Integer):string;	Copia una cadena dada.
Length	Length(s:string):Integer;	Longitud de una cadena.
Pos	Pos(Patron,Fuente:string):Integer;	Posición de la primera ocurrencia de una subcadena.

Ejemplos

```

Cadr := 'computadora';
Delete(Cadr, 4, 3);
WriteLn(Cadr);

Insert('put' Cadr, 4);
Val('32425', r, e);

s := Concat('xyz', 'LUIS');
s := Copy(s, 2, 3);
t := Length(Cadr);

```

Punteros y direcciones

Nombre	Sintaxis	Descripción
Addr	Addr(x) : Pointer;	Devuelve la dirección de un objeto especificado.
Assigned*	Assigned(var P) : Boolean;	Comprueba si una variable procedimiento o puntero es nil.
Cseg	CSeg:Word;	Valor actual del registro CS.
Dseg	Dseg:Word;	Valor actual del registro DS.
Ofs	Ofs(x) : Word;	Desplazamiento de un objeto especificado.
Ptr	Ptr(Set,Ofs:Word):Pointer;	Convierte una base segmento y una dirección de desplazamiento a un valor tipo puntero.
Seg	Seg(x) : Word;	Dirección del segmento de una variable o rutina.
Sptr	Sptr:Word;	Valor actual del registro SP.
Sseg	Sseg:Word;	Valor actual del registro SS.

* Sólo en la versión 7.0.

Ejemplos

```
p := Addr (p);
if Assigned(p) then Writeln('ok');
p := Ptr(Cseg,I);
p := Ptr(Dseg,I);
WriteLn('offset=';Ofs(p2));
abajo := Ptr(0000,$046c);
WriteLn('seg(I)',seg(I));
```

Asignación dinámica

Procedimientos

Nombre	Sintaxis	Descripción
Dispose	Dispose(var p:pointer[,Destructor]*);	
FreeMem	FreeMem(var p:pointer;Tamaño:Word);	
GetMem	GetMem(var p:pointer;Tamaño:Word);	
New	New(var p:pointer[,Init:Constructor]*);	

Funciones

Nombre	Sintaxis	Descripción
MaxAvail	MaxAvail:LongInt;	Tamaño del bloque disponible mayor del montículo (<i>heap</i>).
MemAvail	MemAvail:LongInt;	Cantidad de memoria libre en el montículo (<i>heap</i>).

Ejemplos

```

var
p:^string;
p1:^Real;
while MaxAvail do
GetMem(p,65535);

Dispose(p);
FreeMem(p1,p10);
write('Existen:',MemAvail);

GetMem(p1,2);

New(p);

```

Procedimientos y funciones diversas**Procedimientos**

Nombre	Sintaxis	Descripción
Exclude*	Exclude(var s:set of t; i:t);	Excluye un elemento de un conjunto.
FillChar	FillChar(var x; cuenta:word; valor);	Rellena un número determinado de bytes contiguos con un valor especificado.
Randomize	Randomize;	Inicializa el generador de números aleatorios con una semilla.

* Sólo existe en la versión 7.0.

Funciones

Nombre	Sintaxis	Descripción
Hi	Hi(x):Byte;	Byte de mayor pero del argumento.
Include	Include(var s:set of t;i:T);	Incluye un elemento en un conjunto.
Lo	Lo(x):Byte;	Byte de menor pero del argumento.
Move	Move(var Fuente,Dest:Cuenta: Word);	Copia un número especificado de bytes contiguos de un rango fuente a un rango destino.
ParamCount	ParamCount:Word;	Devuelve el número de parámetros pasados en la línea de órdenes.
ParamStr	ParamStr(Indice):string;	Parámetro de la linea de órdenes.
Random	Random[({Rango:Word})];	Devuelve un número aleatorio.
Sizeof	Sizeof(x):word;	Número de bytes ocupado por el argumento.
Swap	Swap(x);	Intercambia los bytes más altos y menos altos del argumento.
TypeOf	TypeOf(x):Pointer;	Devuelve un puntero a una tabla de métodos virtuales de tipos objeto.
UpCase	UpCase(ch:char):char;	Convierte un carácter a mayúsculas.

Ejemplos

```

Exclude(s,I);

FillChar(A,Sizeof(A),z);

Randomize;

for i := 1 to 10 do
  write(Random(MaxInt):8);

WriteLn('I=',I:5,'Byte alto=',Hi(I):3);

Include(S,I);

WriteLn('byte bajo',Lo(n):6);

Move (car[1],car[50],50);

for i :=1 to ParamCount do
  WriteLn(i:2,':',ParamStr(i))

WriteLn('Real...', Sizeof(Real));

w :=240; w := Swap(w); WriteLn('w='w);

Writeln('Car:UpCase');

for ch := 'a' to 'z' do
  Writeln(ch,UpCase(ch),' ');

```

Procedimiento de tratamiento de archivos

Nombre	Sintaxis	Descripción
ChDir	ChDir(s:string);	Cambia directorio actual.
GetDir	GetDir(D:byte1, var s:string);	Obtiene directorio actual.
MkDir	MkDir(s:string);	Crea un directorio.
RmDir	RmDir(s:string);	Borra un directorio vacío.

Ejemplos

```

ChDir(Camino);
GetDir(1,s);
MkDir(Camino);
RmDir('Aux');

```

Entrada/Salidas

Procedimientos

Nombre	Sintaxis	Descripción
Append	Append(var f:Text);	Abre un archivo de texto, ya existente, para añadir.
Assign	Assign(var f:Nombre);	Asigna el nombre de un archivo externo a una variable archivo.
BlockRead	BlockRead(var f:file;var Buf; Cuenta:Word[;var Resultado:Word]);	Lee uno o más registros de un archivo sin tipos.
BlockWrite	BlockWrite(var f:file;var Buf; Cuenta:Word[;var Resultado:Word]);	Escribe uno o más registros en un archivo sin tipos.
Close	Close (var F);	Cierra un archivo abierto.
Erase	Erase(var F);	Borra un archivo externo.
FileSize	FileSize(var F):LongInt;	Devuelve el tamaño actual de un archivo (no de texto).
Flush	Flush(var F:Text);	Limpia el <i>buffer</i> de un archivo de texto de salida.
Read	Read(F,V1[,V2,...Vn]);	Lee uno o más valores de un archivo.
ReadLn	ReadLn(var F:Text;]V1[,V2,... Vn]);	Igual que Read y después salta al principio de la línea siguiente.
Rename	Rename(var F:NuevoNombre);	Renombra un archivo externo.
Reset	Reset(var F[:file;TamaReg: Word]);	Abre un archivo existente.
ReWrite	ReWrite(var F[:file;TamaReg: Word]);	Crea y abre un archivo nuevo.
Seek	Seek(var F;N:LongInt);	Mueve la posición actual de un archivo a un componente especificado.
SetTextBuf	SetTextBuf(var F:Text;var Buf [; Tamaño:Word]);	Asigna una memoria intermedia de E/S a un archivo de texto.
Truncate	Truncate(var F);	Trunca un archivo con tipos o sin tipos.
Write	Write([var F:Text;]p1[,p2,... pn]); Write(f,v1[,v2,...,vn]); WriteLn([var F:Text;lpl1,p2, ...,pn]);	Escribe en un archivo de texto.
WriteLn		Archivos con tipos. Igual que Write y luego escribe un fin de linea.

Funciones

Nombre	Sintaxis	Descripción
Eof	Eof(var F):Boolean;	Devuelve el estado fin de archivo.
Eoln	Eoln([(var F:Text]):Boolean;	Devuelve el estado fin de linea de un archivo de texto.

Nombre	Sintaxis	Descripción
FilePos	FilePos{var F} :LongInt;	Posición actual de un archivo con o sin tipos.
IOResult	IOResult:Integer;	Estado de la última operación de E/S realizada.
SeekEof	SeekEof{var F:Text}]:Boolean;	Estado fin de archivo de un archivo.
SeekEoln	SeekEoln[{var F:Text}];	Estado fin de linea de un archivo.

Ejemplos

```

var Tf:Text;
    F:File;
Assign(Tf, 'TEST.TXT');
Rewrite(Tf);
Close(Tf);
Append(Tf);
BlockRead(F,A,10,Resultado);
BlockWrite(F,A,1,Resultado);
Erase(F);
Seek(F,FileSize(F));
Flush(F);

While not Eof(Tf) do
begin
  if Eoln(Tf)
    then writeln;
  if Length(s)>0
    then Rename(F,S);
Reset(F);
Seek(Tf,12);
Read(Tf,ch);
Set Text Buf(F,Buffer,512);
Existe :=(IOResult = 0);
while not SeekEof(Tf) do
begin
  ReadLn(Tf,s);
  Inc(C2);
end
while not SeekEoln(Tf) do
begin
  ReadLn(Tf,Ch);
  write(ch);
end;

```

FUNCIONES DEFINIDAS POR EL PROGRAMA

Declaración

```

function <identificador> (<parámetros formales>): <tipo>;
  <declaraciones locales>
begin
  <cuerpo de la función>
end;

```

ARCHIVOS

Preparación de un archivo

- Enlace entre un archivo interno y un archivo externo.
Assign (<identificador de archivo>, <nombre archivo externo>);

Creación de un archivo de registros

- Preparar el sistema para enviar datos de la estructura interna al archivo externo creando y abriendo un nuevo archivo para recibir salida de la computadora.

Rewrite (<identificador de archivo>);

Almacenar datos en un archivo

Write (<identificador de archivo>, <identificador de salida>);

Cerrar un archivo

Close (<identificador de archivo>);

Lectura de un archivo de registros

Procedimiento Reset (Inicializar un archivo)

Reset (<identificador de archivo>);

Lectura de un archivo de registros

Read (<identificador de archivo>, <identificador de entrada>);

Final de archivo

Eof (<identificador de archivo>);

Borrar archivos

Erase (<identificador de archivo>);

Cambiar registros en un archivo de registros

Función de posición FilePos

FilePos (<identificador de archivo>);

Posicionamiento en registro

Seek (<identificador de archivo>, <número de registro>);

DIRECTIVAS DE COMPILEACIÓN

Directivas de conmutación

Directiva	Significado	Valor por defecto	Tipo
A	Alinear datos	{\$A+}	Global
B	Evaluación lógica (<i>boolena</i>)	{\$B-}	Local
D	Información de depuración	{\$D+}	Global
E	Emulación	{\$E+}	Global
F	Force far calls	{\$F-}	Local
G	Generación de código 80286	{\$G-}	Global
I	Verificación de E/S	{\$I+}	Local
L	Generación de información de simbolos locales	{\$L+}	Global
N	Proceso numérico.	{\$N-}	Global
P	Parámetros de cadenas abiertas	{\$P-}	Local
O	Generación de código de solapamiento	{\$O-}	Local
Q	Generación de código de verificación de desbordamiento	{\$Q-}	Local
R	Verificación de rango	{\$R-}	Local
S	Verificación de desbordamiento de pila	{\$S+}	Local
T	Tipos de valores puntero, generados, por el operador @	{\$T-}	Global
V	Verificación de tipos en cadenas pasados como parámetros variables	{\$V+}	Local
X	Activa o desactiva sintaxis extendida de Turbo Pascal	{\$X+}	Global
Y	Información de referencias a simbolos	{\$Y+}	Global

Directivas parámetro

Directiva	Significado	Sintaxis
I	Archivo de inclusión	{\$I <i>nombrearchivo</i> }
L	Enlaza el archivo reforzado con el programa o unidad que se está compilando	{\$L <i>nombrearchivo</i> }
M	Requisitos de asignación de memoria: <i>long</i> , tamaño de la pila; <i>min</i> y <i>max</i> , tamaños mínimo y máximo del montículo	{\$M <i>long,min,max</i> }
O	Nombre de unidad de recubrimiento (solapamiento)	{\$O <i>nombreunidad</i> }

Directivas de compilación condicional

Directiva	Significado
{DEFINE <i>nombre</i> } {SELSE} {SENDIF} {\$IFDEF <i>nombre</i> }	Define un simbolo condicional con el nombre dado. Commuta entre compilación. Termina la compilación condicional iniciada por la última directiva {\$IFXXX}. Compila el texto fuente que le sigue si está definido <i>nombre</i> .

Directiva	Significado
{\$IFDEF} {\$IFOPT <i>conmutador</i> } {\$UNDEF <i>nombre</i> }	Compila el texto fuente que le sigue si <i>nombre</i> no está definido. Verdadero o falso de acuerdo a que la directiva <i>conmutador</i> esté activado o desactivado. Indefine un símbolo condicional definido anteriormente.
{\$IFxxx} ... {\$ENDIF}	{\$IFxxx} ... {\$ELSE} ... {\$ENDIF}
	{\$IFDEF <i>nombre</i> } < <i>sentencias-1</i> > {\$ELSE} < <i>sentencias-2</i> > {\$ENDIF}

Símbolos condicionales

Símbolo	Significado
VER70	Siempre definido, indicando versión 7.0.
MSDOS	Siempre definido, indicando que el sistema operativo es MS-DOS o PC-DOS.
CPU86	Siempre definido, indicando que la CPU pertenece a la familia de procesadores 80 × 86.
CPU87	Definido si un coprocesador matemático está presente en tiempo de compilación.

CONTROL DE DISPOSITIVOS

Las unidades *Crt* y *Printer* son utilizadas con mucha frecuencia por los programadores. *Printer* envía salida a su impresora. La unidad *Crt* implementa una amplia y potente gama de rutinas que le proporcionan un control completo de características de su PC tales como: control del modo de pantalla, códigos de teclado extendido, colores, ventanas y sonido. Por su importancia práctica para el programador, seleccionamos las características más notables utilizadas en programación profesional.

Recuerde que para utilizar la unidad *Crt* o *Printer*, en su programa, debe incluir la cláusula *uses* como cualquier otra unidad:

```
uses Crt, Printer;
```

Caracteres de control

Carácter	Nombre	Descripción
#7	BELL	Emite un sonido (pitido) del altavoz interno.
#8	BS	Retrocede el cursor una columna.
#10	LF	Avanza el cursor una línea abajo.
#13	CR	Retorna el cursor al extremo izquierdo de la línea siguiente.

Teclas de edición de entrada de líneas

Tecla de edición	Descripción
RETROCESO (-)	Borra el último carácter introducido.
ESC	Borra la línea de entrada completa.
ENTER (Intro)	Termina línea de entrada y almacena en almacenamiento temporal. Genera marca fin de linea.
CTRL+S	Igual que Retroceso (<i>Backspace</i>).
CTRL+D	Llama un carácter de la última linea introducida.
CTRL+A	Igual que <i>Esc</i> .
CTRL+F	Llama a la última linea introducida.
CTRL+Z	Termina la última linea introducida y genera una marca fin de archivo.

Funciones

Nombre	Sintaxis	Descripción
KeyPressed	KeyPressed	Devuelve <i>true</i> si se ha pulsado una tecla del teclado; <i>false</i> en caso contrario.
Readkey	Readkey	Lee un carácter del teclado.
WhereX	WhereX	Devuelve coordenada X de la posición actual del cursor.
WhereY	WhereY	Devuelve coordenada Y de la posición actual del cursor.

Ejemplos

```
repeat
    write('Zz');
until KeyPressed;
                                Car := Readkey;
                                gotoxy(1, whereY-1);
```

Nombre	Sintaxis	Descripción
InsLine	InsLine	Inserta una línea vacía en posición del cursor.
LowVideo	LowVideo	Selecciona caracteres de baja intensidad.
NormVideo	NormVideo	Selecciona caracteres normales.
NoSound	NoSound	Desactiva altavoz interno de la computadora.
Sound	Sound(HZ:Word)	Arranca altavoz interno.
TextBackground	TextBackground(Color:Byte)	Selecciona color de fondo.
TextColor	TextColor(Color:Byte)	Selecciona color de carácter (Primer plano).
TextMode	TextMode(Modo:Word)	Selecciona un modo de texto especificado.
Window	Window(x1,y2,x2,y2:Byte)	Define una ventana de texto en pantalla.

Procedimientos

Nombre	Sintaxis	Descripción
AssignCrt	AssignCrt(var F:Text)	Asocia un archivo de texto con la ventana Crt.
ClrEol	ClrEol	Borra todos los caracteres desde la posición del cursor hasta el final de la línea.
ClrScr	ClrScr	Borra la pantalla y cursor a posición inicial.
Delay	Delay(ms:Word)	Retarda un número especificado de segundos.
DelLine	DelLine	Borra la linea que contiene el cursor y mueve todas las líneas inferiores a él, una linea hacia arriba.
GotoXY	GotoXY(x,y:Byte)	Posiciona el cursor en coordenadas (x,y).
HighVideo	HighVideo	Selecciona caracteres de alta intensidad.

```

Gotoxy(1,10);
Write('Mackoy');
ClrEol;
ReadIn(Car);

Sound(440);
Delay(500);
NoSound;

repeat
  x := Succ(Random(71));
  y := Succ(Random(16));
  Window(x,y,x+Random(10),y+Random(10));
  TextBackground(Random8);
  ClrScr;
until KeyPressed;

```

Guía del usuario ISO/ANSI Pascal Estándar

ELEMENTOS DEL LENGUAJE PASCAL

Los elementos del lenguaje Pascal nos permiten construir desde las sentencias más simples hasta las más complejas, así como los distintos tipos de datos, tanto predefinidos como aquellos que son definidos por el usuario. Estos elementos incluyen conjunto de caracteres, palabras reservadas, identificadores estándar, tipos de datos numéricos, carácter, etc.

Cojunto de caracteres

Pascal reconoce letras, dígitos, símbolos especiales y símbolos palabras (palabras reservadas).

Símbolos especiales

```
+ / > . ; <= ) := !
- = [ , ^ >= - " " {
* < ] : <> ( & ' ) }
```

Palabras reservadas

Las palabras reservadas o palabras clave son aquellas a las que Pascal da un significado fijo y predeterminado y que el programador no puede cambiar. Las palabras reservadas en Pascal estándar son las siguientes:

and	div	file	in	of	record	type
array	do	for	label	or	repeat	until
begin	downto	function	mod	packed	set	var
case	else	goto	nil	procedure	then	while
const	end	if	not	program	to	with

Identificadores

Un identificador es una combinación de letras y dígitos que debe comenzar con una letra y que permiten al usuario especificar constantes, variables, procesos, etc. Pascal contiene ciertos identificadores predeclarados o estándar, pero que a diferencia de las palabras reservadas pueden redefinirse por el usuario, aunque no se recomienda esta posibilidad.

Cadena de caracteres

Secuencia de caracteres encerrados entre apóstrofos. Para incluir un apóstrofo en una cadena deberá escribirse el apóstrofo dos veces.

Separadores

Espacios, retornos de carro y sentencias comentario.

Comentarios

Se utilizan para documentar programas y se encierran entre llaves {} o entre símbolos «(*» y «*)». Pueden incluir cualquier carácter excepto una llave.

Etiquetas

Las etiquetas son enteros sin signo utilizados para señalar una sentencia Pascal.

Directivas

Son nombres que sustituyen a bloques de funciones y procedimientos.

TABLA DE IDENTIFICADORES ESTÁNDAR

Los identificadores estándar son de muy diversos tipos. Su característica fundamental es que pueden ser redefinidos por el usuario, aun cuando esta posibilidad no se recomienda porque podría inducir a confusión a otros usuarios o programadores que emplearán un código fuente con los identificadores estándar redefinidos.

Abs	ArcTan	Boolean	Char	Chr	Cos	Dispose	Eof
Eoln	Exp	False	Get	Input	Integer	Ln	MaxInt
New	Odd	Ord	Output	Pack	Page	Pred	Put
Read	ReadLn	Real	Reset	Rewrite	Round	Sin	Sqr
Sqrt	Succ	Text	True	Trunc	Unpack	Write	WriteLn

ESTRUCTURA DE UN PROGRAMA

La estructura general de un programa en Pascal es:

- *Cabecera de programa.*
- *Declaraciones.*
- *Bloque principal.*

Cabecera de un programa

```
program identificador (input, output);
```

Declaraciones

En las declaraciones se definen todos aquellos elementos que luego se utilizarán durante la ejecución del programa (etiquetas, constantes, tipos de datos, variables, subprogramas) de forma similar a como se efectúa en Turbo Pascal. Por ejemplo.

```
const
  Estrella = '*';
  Maximo   = 100;

type
  lista_datos = array[1..Maximo] of real;

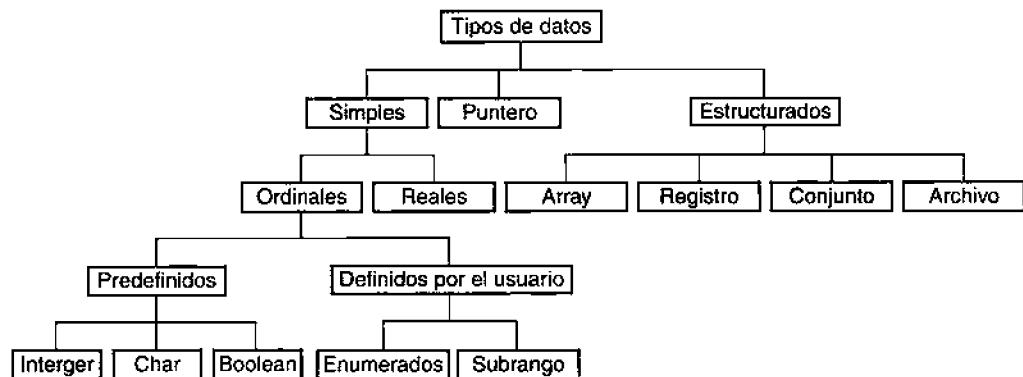
var
  i      : lista_datos;
  x, y : integer;

procedure Cambiar(var a, b : integer);
  var
    aux : integer;
begin
  aux := a;
  a   := b;
  b   := aux
end;
```

Bloque o cuerpo principal

```
begin
  sentencias
end.
```

TIPOS DE DATOS



Definición de tipos

Una sentencia de definición de tipos crea un tipo de dato definido por el usuario.

```
type
  nuevo_identificador = identificador_denotador_tipo;
```

Las definiciones de tipo se pueden utilizar para crear tipos enumerados, subrango, estructurados y tipos punteros, así como sinónimos de tipos simples.

Tipos simples

Los tipos simples que permite Pascal son los tipos numéricos, `integer` (enteros) y `real` (reales), el tipo `char` (carácter), el tipo `boolean` (lógicos o booleanos), todos ellos predefinidos por el lenguaje, y los tipos enumerado y subrango, que al poderse definir por el usuario carecen de identificadores específicos.

Tipos estructurados o compuestos

Son todos aquellos que se forman por la unión de otros de tipo más simple, ya sean homogéneos o heterogéneos. Los más importantes son los arrays, registros, conjuntos y archivos. Dada la gran semejanza con Turbo Pascal se mencionarán únicamente las diferencias.

Tipos estructurados empaquetados

Las estructuras de datos grandes requieren grandes cantidades de espacio de almacenamiento. El empaquetamiento reduce el espacio necesario para tales estructuras, almacenando los datos de modo más eficiente, ocupando el menor espacio posible.

Un tipo de dato se empaqueta si está precedido de la palabra `packed` en una definición de tipo o en una declaración de variable. Por ejemplo:

```

type
  nombre = packed array[1..10] of char;
  adulto = record
    n : nombre;
    sexo : (varon, hembra);           {enumerado}
    edad : 18..65;                  {subrango}
  end;
  fecha = packed record
    dia : 1..31;
    mes : 1..12;
    anno : 1900..2000
  end;

```

El inconveniente de las estructuras empaquetadas es la ralentización en la ejecución del programa.

Tipos Cadena (string)

Las cadenas son una secuencia de caracteres encerrados entre comillas y una variable cadena es una variable de una clase especial de arrays empaquetados con componentes de tipo char.

Una cadena debe ser leída carácter a carácter, aunque puede ser escrita por una sentencia Write o WriteLn. El siguiente programa ilustra el uso de cadenas.

```

program cadenas(input, output);

const
  palabra = 'Hola Mundo';

type
  cadena10 = packed array[1..10] of char;
  cadena20 = packed array[1..20] of char;

var
  indice : integer;
  palentrada : cadena20;
  palsalida : cadena10;
  ch : char;

begin
  for indice := 1 to 20 do
    begin
      Read(ch);                      {lectura de un carácter}
      palentrada[indice] := ch
      {algunas versiones de Pascal permiten leer directamente
      los caracteres con las celdas de un array empaquetado,
      sin necesidad de usar la variable de tipo char}
    end,
  palsalida := palabra;
  WriteLn(palsalida, ';' , palentrada) {escritura cadena completa}
end.

```

Empaquetar y desempaquetar

El empaquetamiento de arrays presenta ventajas en cuanto a la comparación y escritura directa de cadenas, pero el acceso a los elementos de un array empaquetado suele ser más lento y laborioso que a los de uno sin empaquetar, resultando unas veces convenientemente empaquetar y otras desempaquetar arrays. Estas operaciones se pueden realizar mediante los procedimientos predeclarados de transferencia: Pack y Unpack.

```
Pack(array_sin_empaquetar, indice, array_empaquetado)
Unpack(array_empaquetado, array_sin_empaquetar, indice)

Pack(ArrayU, 6, ArrayP);
Unpack(palabra,letras,2);
```

Archivos

Un archivo en Pascal es una estructura de acceso secuencial cuyos componentes son todos del mismo tipo. El tipo del archivo especifica el tipo de los componentes, que es la clase de valores que se pueden almacenar en el archivo. El tipo de componente puede ser cualquiera excepto un tipo archivo o uno estructurado que contenga un tipo archivo. Un archivo se declara como sigue:

```
type
  identificador = file of identificador_de_tipo;
var
  nominas : identificador;
  informe : text;
```

o bien

```
var
  nominas : file of identificador_de_tipo;
  informe : text;
```

El identificador predefinido *text* especifica un tipo especial de archivo de caracteres organizado en líneas de texto. La entrada desde los archivos de texto se hace carácter a carácter. En los archivos de texto los procedimientos Read y Write ofrecen, además de su capacidad para realizar operaciones de lectura y escritura de caracteres, la posibilidad de efectuar conversión automática de tipos; así, cuando se efectúa la lectura de un entero o real, el procedimiento Read examina cada carácter para ver dónde comienza y termina el número en la línea y luego convierte los caracteres del número a un valor numérico. En estos archivos las lecturas numéricas ignoran los blancos hasta encontrar el comienzo del valor numérico y continúan después hasta el primer carácter no numérico. Los procedimientos ReadLn y WriteLn sólo pueden ser utilizados en archivos de texto.

Los archivos estándar de entrada, *input*, y de salida, *output*, están predeclarados como de tipo *text*. No deben ser declarados como variables, ni abierto con *Reset* o *Rewrite*, pero sí aparecer en la cabecera de los programas. Pascal interpreta que

debe leer o escribir en ellos siempre que las sentencias Read o Write no especifiquen un nombre de archivo.

En Pascal hay que considerar que los archivos pueden ser externos o internos. Los archivos externos se mantienen entre las ejecuciones del programa, es decir, son archivos permanentes, y deben especificarse como parámetros en la cabecera del programa. La variable archivo debe ser declarada en la sección de declaración de variables del programa. Por ejemplo, para utilizar un archivo externo denominado archdatos que contiene valores reales, se utilizará la siguiente declaración en el programa principal:

```
var
  archdatos : file of real;
```

Los archivos internos se emplean como almacenamiento auxiliar durante la ejecución de un programa, son archivos temporales que no se listan en la cabecera del mismo.

Pascal utiliza para leer y escribir en los archivos un área de memoria intermedia o buffer en la que sólo cabe un componente del archivo y siempre que se declara una variable de tipo archivo, automáticamente, se crea otra conocida como la variable buffer del archivo, que se representa por el nombre de la variable archivo seguido de una flecha hacia arriba o del símbolo ^ y constituye una manera de inspeccionar o añadir componentes al archivo. Pascal proporciona los procedimientos estándar Get y Put para la manipulación directa de la variable buffer y dado que los procedimientos Read y Write no están definidos en todas las versiones de Pascal estándar para los archivos que no son de texto, es posible que sea necesario utilizar en ellos Get, Put y la variable buffer de archivo.

Read (var_archivo, var_componente)

equivaldría a

```
var_componente := var_archivo^;
Get(var_archivo);
```

y

```
Write(var_archivo, var_componente)
Put(var_archivo)
```

Ejemplo:

```
program mezcla(articulo, nuevoart); {mezcla de archivos}
type
  cadena3 = packed array[1..3] of char;
  registro = record
    codigo : cadena3;
    cantidad : integer;
    precio : real
  end;
  archivo = file of registro;
```

```

var
  articulo, nuevoart, auxiliar : archivo;
begin
  Reset(articulo);
  Reset(nuevoart);
  Rewrite(auxiliar);
  while not eof(articulo) and not eof(nuevoart) do
    begin
      if articulo^.codigo < nuevoart^.codigo then
        begin
          auxiliar^ := articulo^;
          get(articulo)
        end
      else
        begin
          auxiliar^ := nuevoart^;
          get(nuevoart)
        end;
      Put(auxiliar)
    end;
  while not eof(articulo) do
    begin
      auxiliar^ := articulo^;
      Put(auxiliar^);
      get(articulo)
    end;
  while not eof(nuevoart) do
    begin
      auxiliar^ := nuevoart^;
      Put(auxiliar^);
      get(nuevoart)
    end;
  Rewrite(articulo);
  Reset(auxiliar);
  while not eof(auxiliar) do
    begin
      articulo^ := auxiliar^;
      Put(articulo^);
      get(auxiliar)
    end
  end.
end.

```

Punteros

Una variable de tipo puntero es una variable cuyo valor es una dirección. Esta dirección es la posición de un valor en memoria. Las variables dinámicas se pueden crear y destruir en tiempo de ejecución. A las variables dinámicas se accede mediante punteros.

La sintaxis para declarar un tipo puntero es:

```
identificar = ^tipo;
```

Por ejemplo:

```

var
  test : ^integer;

```

define la variable `test` como una variable puntero que apunta a valores enteros. Para acceder al entero al que apunta `test`, se utiliza la sintaxis

```
test^ o bien test
```

TIPOS COMPATIBLES

Los tipos deben ser compatibles en ciertas operaciones. Los tipos son compatibles si una de las siguientes condiciones es verdadera:

1. **Los tipos han sido definidos equivalentes en una sentencia de declaración de tipos.** Así,

```
type
    estado = boolean;
hace a estado y boolean tipos compatibles.
```

2. **Un tipo es un subrango de otro.** Por ejemplo, si

```
type
    alfabeto = 'a'..'z';
alfabeto y char son tipos compatibles.
```

3. **Ambos tipos subrango son del mismo tipo base.** Por ejemplo, si:

```
type
    bajo = 1..50;
    alto = 51..500;
bajo y alto son tipos compatibles ya que ambos son subrango de tipo entero.
```

4. **Ambos tipos son tipos cadena con el mismo número de componentes.**

```
type
    nombre = packed array[1..5]of char;
    palabra = packed array[1..5]of char;
nombre y palabra son compatibles ya que ambos son cadenas de caracteres de cinco caracteres de longitud.
```

5. **Los tipos son tipos conjunto con tipos base compatibles.** Por ejemplo, los tipos definidos por:

```
type
    digitos = set of '0'..'9';
    intervalo = set of '4'..'6';
digitos e intervalo son compatibles ya que tanto las variables como los tipos conjunto son del tipo base char.
```

6. **Los tipos no pueden ser compatibles si uno está empaquetado y el otro no.** Sólo pueden ser compatibles si ambos están empaquetados.

OPERADORES

Constituyen los elementos esenciales que transforman y permiten procesar la información. Pascal estándar utiliza los mismos operadores que Turbo Pascal y el orden de prioridad en las expresiones también es análogo.

Operador	Prioridad	Clasificación
not *, /, div, mod, and +, -, or <, <=, =, < >, >=, >, in	Más alta (se evalúa primero) ↓ Más baja (se evalúa el último)	Negación lógica Operadores de multiplicación Operadores de adición Operadores de relación

Otras operaciones

Operador	Operación	Tipos de operandos	Tipo de resultado
<i>Asignación</i> :=	Asignación	Cualquier tipo assignable	Ninguno
<i>Acceso a variables</i>			
[,]	Índice de array	Array	Tipo componente
:	Selección de campo	Record (registro)	Tipo de campo
↑	Identificación	Puntero	Tipo dominio
↑	Acceso a buffer	Archivo	Tipo componente
<i>Construcción</i>			
[,]	Construcción de conjunto	Tipo base	Conjunto
', '	Construcción de cadenas	Char	Cadena

ENTRADAS Y SALIDAS BÁSICAS

La entrada de un programa es la información real o datos que el programa procesa. La salida constituye el resultado del programa.

Salida con WriteLn y Write

Los formatos generales son:

```
Write (lista de elementos);
WriteLn (lista de elementos);
WriteLn;
```

Con estos formatos la salida es enviada al archivo estándar de salida (*output*).

Entrada (lectura) con Read y ReadLn

Los formatos generales son:

```
Read (lista de variables);
ReadLn (lista de variables);
ReadLn;
```

Con estos formatos los procedimientos Read y ReadLn leen datos del archivo de entrada estándar (*input*) y los asignan a las variables especificadas en la lista.

El comportamiento del procedimiento Read depende del tipo de variable que se está leyendo y presenta ciertas diferencias con el que tiene en Turbo Pascal. Cuando la variable es de tipo char se lee un carácter de la entrada y se le asigna a la variable; si el carácter leído es una marca de fin de línea se asigna un carácter en blanco a la variable. Cuando la variable es numérica se leen una serie de caracteres y se asignan a la variable, siempre que la sucesión se ajuste a la sintaxis correspondiente al tipo de variable; los espacios en blanco, tabuladores y marcas de fin de línea que pudieran preceder a la serie se saltan.

ReadLn se comporta de forma similar a Read, pero, tras la lectura del último parámetro, salta al principio de la siguiente línea.

Sentencias

Las sentencias constituyen el verdadero código ejecutable de un programa. Existen diferentes tipos de sentencias que se organizan en grupos con características afines. Por su semejanza con Turbo Pascal sólo se efectuará una enumeración de los diferentes grupos.

- *Sentencia de asignación*. Asigna el valor de una expresión a la variable situada a la izquierda del operador de asignación.
- *Sentencia procedimiento*. Se utilizan para llamar a procedimientos estándar y definidos por el usuario. Los procedimientos se llaman por su nombre seguido por una lista de argumentos entre paréntesis. El número de elementos en la lista de argumentos debe corresponderse con el número de parámetros en la definición del procedimiento.
- *Sentencia compuesta*. Grupos de sentencias situados entre las palabras *begin* y *end*.
- *Sentencia with*. Permite abreviar la notación necesaria para el acceso a los campos de variables tipo registro.
- *Sentencia if-then-else*. Realiza diferentes acciones en función de una condición o expresión lógica.
- *Sentencia case*. Ejecuta una sentencia entre varias en función del valor de una expresión especificada. Algunas versiones de Pascal tienen una cláusula *otherwise* para especificar las acciones que se llevarán a cabo cuando el valor de la variable que se comprueba no coincide con ninguno de los reseñados en la lista de pruebas. Una sentencia *case* puede sustituirse por varias de tipo *if* anidadas en la rama *else*.

- *Sentencia de bifurcación incondicional.* La sentencia goto salta a una sentencia etiquetada para que continúe por este punto la ejecución de programa. Prácticamente no se utiliza nunca.
- *Sentencia while.* Repite una sentencia mientras que una condición sea verdadera. La condición se comprueba antes de que se ejecute la sentencia.
- *Sentencia repeat.* Ejecuta una sentencia hasta que una condición especificada sea verdadera. La condición se comprueba tras la ejecución de la sentencia.
- *Sentencia for.* Ejecuta una sentencia un número específico de veces. Utilizarán to o down-to para especificar si la variable de control debe incrementarse o decrementarse en cada iteración.

PROCEDIMIENTOS Y FUNCIONES DEFINIDOS POR EL USUARIO

Los procedimientos y funciones son rutinas (subprogramas) que realizan acciones específicas. La diferencia entre procedimientos y funciones es que las funciones devuelven un valor a través de su propio nombre, mientras que los procedimientos no.

Declaración de un procedimiento

```
procedure nombre_procedimiento (lista de declaración de
                                parámetros formales);
  declaraciones
begin
  sentencias
end;
```

Llamada a un procedimiento

```
nombre_procedimiento (lista de parámetros actuales);
```

Declaración de una función

```
function nombre_funcion (lista de declaración de parámetros
                           formales): tipo;
  declaraciones
begin
  sentencias
  nombre_funcion := valor de la función
end;
```

Llamada a una función

```
variable := nombre_función (lista de parámetros actuales);
```

Es posible efectuar la llamada dentro de una sentencia y una expresión.

Parámetros variables

Los parámetros variables son parámetros formales utilizados para representar un parámetro real (actual) cuyo valor puede cambiar durante la ejecución del procedimiento. Los parámetros variables se declaran incluyendo la palabra reservada var antes que el nombre de los parámetros en la cabecera. Los parámetros actuales correspondientes deberán ser variables y su tipo podrá ser cualquiera.

Ejemplo:

```
procedure cambiar (var m, n: integer);
var
  aux : integer;
begin
  if m > n then
    begin
      aux := n;
      n := m;
      m := aux
    end
  end;
end;
```

Parámetros valor

Los parámetros valor constituyen variables independientes con su propia ubicación en memoria, que reciben el valor del argumento al comienzo de la ejecución del subprograma y se declaran en la cabecera de la rutina por una lista de identificadores seguidos por sus tipos. Este tipo puede ser cualquiera excepto un archivo o un tipo estructurado que contenga un archivo como componente.

Ejemplo:

```
function superficie (longitud, anchura: real): real;
begin
  longitud := 12 * longitud;           {convertir pies a pulgadas}
  anchura := 12 * anchura;
  superficie := longitud * anchura
end;
```

Parámetros función

Los parámetros funcionales son parámetros formales que se utilizan para representar una función, parámetro actual, que se puede utilizar durante la ejecución de la rutina que se declara. Es decir, los parámetros funcionales permiten pasar una función a una rutina.

Parámetros procedurales

Los parámetros procedurales son a los parámetros funcionales como los procedimientos son a las funciones. El parámetro formal representa un procedimiento, parámetro actual, que se puede utilizar durante la ejecución de la rutina que se ha declarado.

Directiva forward

Las rutinas definidas por el usuario deben ser declaradas antes de ser referenciadas. La directiva **forward** permite alterar esta norma, indicando al compilador que la definición de la rutina vendrá más adelante en el programa. Es necesaria cuando dos rutinas son mutuamente recursivas.

FUNCIONES INTRÍNSECAS

Las funciones intrínsecas o predefinidas se llaman por identificadores predefinidos y se dividen en cuatro categorías: numéricas, transferencia, ordinales y lógicas.

Funciones	Descripción
<i>Numéricas</i>	
<code>abs(x)</code>	Devuelve el valor absoluto de <i>x</i>
<code>arctan(x)</code>	Devuelve el ángulo en radianes (entre $-pi/2$ y $pi/2$)
<code>cos(x)</code>	Devuelve el coseno de <i>x</i> (<i>x</i> en radianes)
<code>exp(x)</code>	Devuelve e^x ($e = 2.718282$)
<code>sin(x)</code>	Devuelve el seno de <i>x</i> (<i>x</i> en radianes)
<code>sqr(x)</code>	Devuelve el cuadrado de <i>x</i>
<code>sqrt(x)</code>	Devuelve la raíz cuadrada positiva de <i>x</i> , <i>x</i> debe ser mayor o igual a cero
<i>De transferencia</i>	
<code>round(x)</code>	Redondea <i>x</i> (de tipo real) al entero más próximo
<code>trunc(x)</code>	Trunca la expresión <i>x</i> (de tipo real) y devuelve la parte entera
<i>Ordinales</i>	
<code>chr(x)</code>	Devuelve el carácter cuyo ordinal es <i>x</i>
<code>ord(x)</code>	Devuelve el entero que es el número ordinal del valor de <i>x</i> , <i>x</i> puede ser cualquier tipo ordinal
<code>pred(x)</code>	Devuelve el valor cuyo número ordinal es el precedente al valor de <i>x</i>
<code>succ(x)</code>	Devuelve el valor cuyo número ordinal es el siguiente al valor de <i>x</i>
<i>Lógicas</i>	
<code>eof(f)</code>	Devuelve <code>true</code> si se alcanza el final del archivo <i>f</i> . En caso contrario devuelve <code>false</code>
<code>eoln(f)</code>	Devuelve <code>true</code> si se alcanza el final de una línea en <i>f</i> . En caso contrario devuelve <code>false</code> (<i>f</i> debe ser un archivo de texto)
<code>odd(x)</code>	Devuelve <code>true</code> si la expresión es impar, <code>false</code> si es par, <i>x</i> debe ser una expresión entera

PROCEDIMIENTOS INTRÍNSECOS

Los procedimientos intrínsecos o estándar se llaman utilizando identificadores predefinidos. Existen tres categorías: tratamiento de archivos, asignación dinámica y transferencia.

Procedimientos	Descripción
<i>Tratamiento de archivos</i>	
<code>close(x)</code>	Cierra el archivo <i>f</i> . No pertenece a Pascal estándar, pero lo implementan muchas versiones de compiladores de Pascal.
<code>Get(f)</code>	Obtiene el valor del siguiente componente del archivo <i>f</i> en su variable de buffer de archivo.
<code>Page(f)</code>	Sitúa el siguiente texto escrito en un archivo <i>f</i> en una nueva página.
<code>Put(f)</code>	Escribe el buffer del archivo, <i>f</i> [^] , en el archivo <i>f</i> . Después de una llamada a <code>Put(f)</code> , el valor de la función <code>eof(f)</code> será verdadero y el valor de <i>f</i> [^] indefinido.
<code>Read(f, v)</code>	Lee elementos de un archivo o del terminal. La lista de variables son de tipo entero, real, char o un subrango de estos tipos. No está permitida la entrada directa de tipos enumerados, lógicos o cadena.
<code>ReadLn(f, v)</code>	Funciona igual que <code>Read</code> pero el puntero del archivo avanza al primer carácter de la siguiente línea y se utiliza sólo con archivos de texto. Si las variables a leer se omiten se avanza el puntero del archivo.
<code>Reset(f)</code>	Abre un archivo <i>f</i> para lectura y pone el puntero del archivo a la primera posición del archivo.
<code>Rewrite(f)</code>	Crea un archivo <i>f</i> o borra <i>f</i> si ya existe, lo abre para escritura y pone el puntero a la primera posición del archivo.
<code>Write(f, e)</code>	Escribe un elemento o una lista de elementos al final del archivo.
<code>WriteLn(f, e)</code>	Funciona igual que <code>Write</code> , pero añade un retorno de carro y sólo se utiliza con archivos de texto. Si los elementos a escribir se omiten se escribe un retorno de carro.
<i>Asignación dinámica</i>	
<code>New(p)</code>	Asigna dinámicamente espacio de almacenamiento a una nueva variable apuntada. Si <i>p</i> es la variable puntero, <i>p</i> [^] es la variable creada por el procedimiento <code>new</code> .
<code>Dispose(p)</code>	Elimina una variable creada por el procedimiento <code>new</code> . Si <i>p</i> es la variable puntero, el procedimiento <code>dispose</code> suprime la variable <i>p</i> [^] .
<i>Transferencia (empaquetamiento)</i>	
<code>Pack(a, i, b)</code>	Empaquetá el array <i>a</i> , comenzando en la posición <i>i</i> , en el array, empaquetado <i>b</i> .
<code>Unpack</code>	Desempaquetá el array empaquetado <i>b</i> en el array <i>a</i> , comenzando en la posición <i>i</i> .

Construya un programa que permita generar una agenda con números de teléfono, así como realizar consultas.

Codificación

```
program ejercicio_apC_03(input, output);
const
  Max = 10;
```

```

type
  nombre    = packed array [1..15] of char;
  telefono = packed array [1..11] of char;
  conocidos = array [1..Max] of nombre;
  tlfnos   = array [1..Max] of telefono;
var
  amigo     : conocidos;
  numero   : tlfnos;
  auxtfnos : telefono;
  auxamigo : nombre;
  unamigo  : nombre;
  i, j      : 1..Max;
  terminado : boolean;
  otra      : char;

procedure buscar(humano: nombre; listal: conocidos; lista2: tlfnos);
var
  encontrado : boolean;
  primero, ultimo, central: 1..Max;
begin
  encontrado := false;
  primero := 1;
  ultimo := Max;
  while not encontrado and (primero <= ultimo) do
    begin
      central := (primero + ultimo) div 2;
      if humano = listal[central] then
        encontrado := true
      else
        if humano > listal[central] then
          primero := central-1
        else
          ultimo := central+1
    end;
    if encontrado then
      writeln('Su telefono es: ', lista2[central])
    else
      writeln('No consta en la agenda')
  end;

procedure /leerp(var persona :nombre);
var
  i : integer;
  ch : char;
begin
  i := 1;
  while i <= 15 do
    begin
      if not eoln then
        begin
          Read(ch);
          persona[i] := ch;
        end
      else
        persona[i] := ' ';
      i := i+1
    end
  end;

```

```

    end;
  ReadLn
end;

procedure leern(var num : telefono);
var
  i : integer;
  ch : char;
begin
  i := 1;
  while i <= 11 do
  begin
    if not eoln then
    begin
      Read(ch);
      num[i]:=ch;
    end
    else
      num[i] := ' ';
    i := i+1
  end;
  ReadLn
end;

begin
  for i := 1 to Max do
  begin
    Write('Introduce nombre ');
    Leerp(amigo[i]);
    Write('Introduce telefono ');
    Leern(numero[i])
  end;
  for i := 1 to Max-1 do
    for j := i+1 to Max do
      if amigo[i] > amigo[j] then
      begin
        auxamigo := amigo[i];
        auxtfno := numero[i];
        amigo[i] := amigo[j];
        numero[i] := numero[j];
        amigo[j] := auxamigo;
        numero[j] := auxtfno
      end;
  repeat
    Write('Introduce la persona a buscar');
    Leerp(unamigo);
    buscar(unamigo, amigo, numero);
    WriteLn('¿Otra persona? (s/n)');
    ReadLn(otra);
    terminado := otra = 'n'
  until terminado
end.

```

Supuesta la siguiente entrada de datos

-3 * 12.4

averigüe la salida que obtendría tras la ejecución del programa que se muestra a continuación. Observe las diferencias con Turbo Pascal.

```
program ejercicio_apC_04(input, output);
var
  c1, c2, c3, c4 : char;
  i1, i2, i3      : integer;
begin
  ReadLn(c1,i1,c2,c3,i2,c4,i3);
  WriteLn(i1,c3,i3);
end.
```

Ejecución

3 * 4

Escriba un programa que permita efectuar una copia un archivo de texto en otro.

Codificación

```
program ejercicio_apC_05(inicio, destino);
var
  inicio, destino : text;
  letra           : char;
begin
  Reset(inicio);
  Rewrite(destino);
  while not eof(inicio) do
    begin
      while not eoln(inicio) do
        begin
          Read(inicio, letra);
          Write(destino, letra)
        end;
      ReadLn(inicio);
      WriteLn(destino)
    end
  end
end.
```

Diseñe un programa para realizar la actualización de las cuentas de los clientes de un banco. Se recibirá como entrada el número de cuenta y el saldo para cada cliente, además de la cantidad a ingresar o a reintegrar al mismo. La actualización tendrá en cuenta:

- Si es un ingreso, se acumulará en el saldo y se introducirá como cantidad positiva.
- Si es un reintegro o retirada de fondos, se restará del saldo y se introducirá como cantidad negativa.
- De cada número de cuenta se comprobará su fiabilidad según la siguiente técnica. El último dígito de dicho número debe ser la suma de los anteriores módulo 8 para que dicho número sea correcto. El número de cuenta tendrá un máximo de 10 dígitos.

Se pide obtener como salida el número de cuenta y el saldo actualizado, si el número de cuenta es erróneo se debe indicar y no se hará ninguna operación.

Codificación

```

program ejercicio_apC_06(input, output);
type
  numeros = array [1..10] of 0..9;
  cuenta = record
    numcuenta : numeros;
    saldo      : real
  end;
  lista   = array [1..200] of cuenta;
var
  cliente : lista;
  lim     : integer; (no se desea llenar completamente el array)

procedure inicializar(var b: lista);
begin
  var
    i, j : integer;
  begin
    for i := 1 to 200 do
      with b[i] do
        begin
          for j := 1 to 10 do
            numcuenta[j] := 0;
        end
    end;
  end;

function correcto(j: integer; nc: numeros): boolean;
begin
  var
    k, suma : integer;
  begin
    for k := 1 to j - 1 do
      suma := suma + nc[k];
    if suma mod 8 = nc[j] then
      correcto := true
    else
      correcto := false
  end;
end;

procedure actualizar(var uncliente: cuenta; j: integer);
begin
  var
    k      : integer;
    cantidad : real;
  begin
    Write('Cantidad: (+)ingreso, (-)reintegro ');
    ReadLn(cantidad)
    With uncliente do
      begin
        for k := 1 to j do
          Write(numcuenta[k]:1);
        saldo := saldo + cantidad;
        WriteLn('Saldo ', saldo:8:2)
      end
  end;
end;

```

```

procedure leer(var b: lista; var lm: integer);
var
  i, j: integer;
  d : char;      {se leerá el número de cuenta
                  como caracteres}
begin
  repeat
    i := i + 1;
    with b[i] do
      begin
        Write('Deme el numero de cuenta');
        j := 0;
        while not eoln and (j < 10) do
          begin
            j := j + 1;
            Read(d);
            numcuenta[j] := ord(d) - ord('0')
          end;
        ReadLn;
        Write('Deme el saldo');
        ReadLn(saldo)
      end;
    if correcto(j, b[i].numcuenta) then
      actualizar(b[i], j)
    else
      writeln('Numero incorrecto')
  until eof or (i = 200);
  lm := i
end;

begin
  inicializar(cliente);
  leer(cliente, lim)
end.

```

Ejercicios resueltos

Codifique un programa que muestre el dibujo de un pino en pantalla.

Codificación

```

program ejercicio_apC_01(output);
var
  i, j : integer;
begin
  for i := 1 to 25 do
    writeln;
  for i := 1 to 10 do
    begin
      write(' ':40-i);
      for j:= 1 to 2*i-1 do
        write('*');
      writeln
    end;

```

```
for i := 1 to 5 do
begin
  write('**':40);
  writeln
end
end.
```

Diseñe un programa que indique si un número entero es capicúa.

Codificación

```
program ejercicio_apC_02(input, output);
var
  numero,
  copia,
  alreves,
  resto      : integer;

begin
  Write('Introduzca un numero en el rango de los enteros ');
  ReadLn(numero);
  copia := numero;
  alreves := 0;
  while numero <> 0 do
  begin
    resto := numero mod 10;
    alreves := alreves * 10;
    alreves := alreves + resto;
    numero := numero div 10
  end;
  if alreves = copia then
    WriteLn ('El numero es capicua')
  else
    WriteLn ('El numero no es capicua')
end.
```


Índice

- abierto-cerrado*, 93
Abstracción, 3, 4, 5, 24, 84, 97, 107
Ada, 24
Ada 95, 115
Algoritmo, 4
 análisis, 553-571
 árbol de expansión de coste mínimo, 483, 488
 backtracking, 277
 búsqueda de puntos de articulación, 447
 camino más corto, 462, 467
 del aumento de flujo, 476
 de la mochila, 290
 Dijkstra, 455, 461, 464
 diseño, 42
 divide y vencerás, 267
 eficiencia, 553
 Prim y Kruskal, 455, 484, 486
 Ford-Fulkerson, 479
 flujo, 470
 fundamentales, 455
 Kruskal, 486
 ordenación topológica, 456-459
 problema de la mochila, 290
 problema de la selección múltiple, 297
 problema de las ocho reinas, 286
 problema de los matrimonios estables, 300
 problema del laberinto, 293
 problema del salto del caballo, 279, 281
 problema del viajante de comercio, 298
 Torres de Hanoi, 167, 271
 vuelta atrás, 277
 vuelta del caballo, 279
 Warshall, 435, 459
Análisis de algoritmos, 553-571
búsqueda lineal, 536
búsqueda secuencial, 536
de ordenación, 565
eficiencia, 553
notación O-grande, 556
ordenaciones cuadráticas, 514
orden de magnitud, 554
ordenación por burbuja, 507, 566
ordenación por inserción, 567
ordenación por mezcla, 568
ordenación *Radix sort*, 570
ordenación rápida, 520, 569
ordenación por selección, 509, 565
ordenación por inserción, 514
tablas comparativas de tiempos, 571
Análisis, 3
ANSI, 49
Archivos, 581
Apuntador, 126-130. *Véase* Punteros
Archivos, 74
 aumento de la velocidad, 622
 compilados, 79
 de inclusión, 74
 función de direccionamiento *hash*, 622
 indexados, 645
 ordenación, 655
 resolución de colisiones, 625
 texto, 617
Arrays,
 circulares, 231
Aserción, 16
Aserto, 16

Biblioteca de software, 110
Bicola, 239
 estructura, 240

- Booch, 106
Bucle, 19
bug, 39
Burbuja, 503
Búsqueda, 536
 binaria, 536, 540, 563
 análisis, 543
 eficiencia, 542
 binaria recursiva, 544
 conversión de claves, 536
 eficiencia de los algoritmos, 563
 lineal, 536
 análisis, 536
 secuencial, 536, 563
tablas comparativas de tiempos, 571
- C++, 24
Cadena de caracteres, 202
 mediante listas circulares, 202
Calidad del software, 8
 compatibilidad, 9
 corrección, 8
 eficiencia, 8
 extensibilidad, 9
 facilidad de utilización, 8
 factores, 8
 integridad, 8
 reutilización, 9
 robustez, 8
 transportabilidad, 8
 verificabilidad, 8
Ciclo de vida, 3, 9
 análisis, 10
 definición del problema, 10
 depuración, 10
 diseño, 10
 especificación, 10
 implementación, 10
 mantenimiento, 10
 requisitos, 10
Clasificación, 502
Codificación, 12, 15
Cola, 227
 acolar, 256
 bicola, 239
 desacolar, 256
 especificación formal, 228
 Final, 231
- FIFO, 256
implementación, 229
 con arrays circulares, 231
 con listas circulares, 237
 con listas enlazadas, 237
Operaciones, 230, 231
 Añadir, 230
 Borrar, 234
 Cola llena, 230, 233
 Cola vacía, 230, 233
 Quitar, 230, 234
 Siguiente, 233
 TAD, 227, 228, 231
 Frente, 234
- Cola de prioridad, 227, 244-252
implementación, 245
 mediante una lista, 245
 mediante lista de n colas, 246
 problema resuelto con, 247, 252
- Comprensibilidad, 26
Concurrencia, 109
Corrección, 22
Chip de software, 110
- Datos, 12, 96
 flujo de, 12
 predefinidos, 96
Depuración, 3, 23, 38, 40, 41
Depurador, 46
Dijkstra, 13
Diseño, 3, 11
 descendente, 5, 23
dispose, 126, 137, 145, 160
divide y vencerás, 267
Documentación, 3, 35
 del programa, 37
 externa, 37
 interna, 37
 manual de mantenimiento, 35, 36
 manual de usuario, 35
- Eficiencia, 3, 27, 46
Eiffel, 115
Encapsulación, 107
Encapsulamiento, 7, 24

- Errores, 32
 - compilación, 44
 - localización, 39
 - lógicos, 45
 - reparación de, 39
 - sintaxis, 44
 - tiempo de ejecución, 45
 - tratamiento, 323
- Estructura de datos, 5
 - dinámica, 125
 - estática, 126
- es-un*, 108
- EsVacia, 208
- Expresión aritmética, 215
 - evaluación, 215
 - algoritmo de, 216
 - notaciones, 215
 - postfija, 215
 - prefija, 215
- Fibonacci, 264
- Ficheros, 581
 - bases de datos, 584
 - bloque, 585
 - campo, 582
 - clave, 585
 - concepto, 581, 583
 - estructura jerárquica, 581
 - factor de bloqueo, 585
 - operaciones
 - actualización, 591
 - clasificación, 592
 - consulta, 590
 - creación, 590
 - destrucción, 592
 - estallido, 592
 - fusión, 592- 593
 - mantenimiento, 594
 - procesamiento, 596
 - reorganización, 592
 - reunión, 592
 - rotura, 592, 594
 - organización
 - directa, 589
 - secuencial, 588
 - registro, 583
 - físico, 585
- FIFO, 227, 256
- FreeMem, 146
- Función, 32
 - hash*, 623
- Fusión, 546
- gda*, 456
- Genericidad, 110
- getmcm, 145, 146
- goto, 31
- grafo, 417
 - árbol de expansión mínimo, 48
 - camino, 420
 - componentes conexas, 437
 - componentes fuertemente conexas, 439
 - conexo, 421
 - coste mínimo, 483
 - definiciones, 418
 - dirigidos, 418, 419, 493
 - de entrada, 419
 - de Ford y Fulkerson, 472
 - de salida, 419
 - flujo, 470
 - fuertemente conexo, 421
 - lista de adyacencia, 424
 - matriz de adyacencia, 424
 - matriz de caminos, 442, 459
 - no dirigidos, 418, 419
 - realización con listas de adyacencia, 426
 - realización con matriz de adyacencia, 426
 - recorrido, 431
 - en anchura, 431
 - realización, 433
 - en profundidad, 434
 - realización, 435
 - representación, 421
 - gda*, 456
 - TAD, 426
- heap*, 143
- Herencia, 7, 88
- Herramientas de resolución de problemas, 5
- SI, 74
- Identificador, 30, 71

- .Implementación, 12
- include, 74
- Ingeniería de software, 3, 9
- Integración, 13
- Intercalación, 546
- Interfaz, 26, 92
- invariante, 14, 19
- ISO, 49

- Legibilidad, 23, 33
- Lenguaje de programación, 111
 - basado en clases, 114
 - basado en objetos, 114
 - clasificación, 112, 114
 - criterios, 115
 - evolución, 112
 - genealogía, 113
 - híbrido, 115
 - orientado a objetos, 114
- LIFO, 208
- Lista circular, 179
 - especificación, 195
 - implementación, 196
- Lista doblemente enlazada, 179
 - aplicación, 186
 - creación de nodos, 181
 - eliminación de nodos, 184
 - especificación, 179
 - implementación, 180
 - programa Ambulatorio, 192
- Lista enlazada, 153, 208
 - búsqueda, 162
 - especificación formal, 153, 155
 - implementación con arrays, 156
 - implementación con punteros, 159
 - Info, 162
 - iniciar, 162
 - inserción, 164
 - nodo, 154
 - operaciones, 155
 - de dirección, 163
 - recorrido, 168
 - supresión de un elemento, 166
 - TAD lista, 153
 - vacía, 154
 - variable de puntero, 160
- Lista ordenada, 168
 - búsqueda, 168
 - implementación, 168
- Mantenimiento, 3, 14, 15
- Mark, 145
- MaxAvail, 145, 147
- MemAvail, 145, 147
- Memoria, 144
 - asignación, 145
 - liberación, 145
- Mezcla, 546
 - dos listas, 547
- Modelado, 11
 - de datos, 11
- Modelo objeto, 106
- Modificabilidad, 25
- Modula-2, 24
- Modularidad, 5, 6, 23, 25, 83, 89, 107
- Modularización, 91
- Módulo, 5, 24, 53, 86-87, 90
 - acoplamiento, 94
 - cohesión, 95
 - cohesivos, 6, 25, 95
 - diseño, 94
 - estructura, 91
 - implementación, 90
 - interfaz, 90
- Motículo, 143, 523

- new, 126, 130-134, 145, 160
- nil, 138, 151
- Nodo, 134

- \$O, 76
- Objeto, 7, 24, 88, 118
 - beneficios, 118
- Objetos, 683
 - clases, 685
 - conceptos, 683
 - constructor, 717, 721
 - declaración, 692, 693
 - definición, 694
 - destructor, 707, 717
 - dinámicos, 705, 717
 - dispose, 724
 - estructura, 684
 - herencia, 696
 - simple, 698
 - jerarquía, 715

- métodos, 685, 705
 - anulación, 718
 - implementación, 693
 - virtuales, 716
 - liberación de memoria, 707
 - new**, 724
 - OOP**, 683
 - polimorfismo, 714
 - POO**, 683
 - privada, 691
 - private, 725
 - public, 725
 - pública, 691
 - recursivo, 263
 - Self**, 709
 - sintaxis, 684
 - subclase, 697
 - unidades, 694
 - uso, 695
 - variables instancia, 685
- Obsolescencia, 15
- Ocultación de la información, 5, 7, 24, 87, 93
- Ordenación, 501-503
 - binsort*, 529-533
 - burbuja, 503-510
 - análisis, 510
 - heapsort*, 525
 - inserción, 512
 - métodos, 503
 - por montículo, 525-529
 - rápida, 517
 - Quicksort*, 517- 522
 - radixsort*, 533
 - selección, 510
 - Shell*, 514- 517
- Ordenación externa, 655
 - métodos, 655
 - método polifásico, 667
 - mezcla directa, 656
 - mezcla equilibrada, 661
 - mezcla simple, 656
- Orientación a objetos, 105
- Overlay**, 75
- Paquete**, 87
- Parámetros, 31
 - valor, 31
 - variable, 31
- parte-de**, 108
- Pascal, 53, 829
 - guía de usuario, 829-849
- Persistencia, 88, 109
- Persistencia, 829-849
- Pila**, 207
 - cima, 209
 - Cima, 209
 - definición, 207, 224
 - especificación formal, 207
 - Esvacia, 209, 212
 - evaluación de expresiones aritméticas, 215-218, 222
 - algoritmos, 216
 - fondo*, 208
 - implementación, 208
 - con arrays, 208
 - con listas enlazadas, 208
 - con variables dinámicas, 211
 - LIFO**, 208
 - Meter**, 209, 212
 - Operaciones, 208
 - Pilallena, 209
 - Pilavacia, 209, 212
 - Sacar, 212
 - Suprime, 209, 212
 - Utilización, 210
 - Unidad ExpPost, 220
 - Notaciones de expresiones, 215
 - polaca, 215
 - polaca inversa, 215
 - postfija, 215, 216, 220
 - prefija, 215
 - Pilavacia, 208, 212
 - Plantillas, 98
 - Polimorfismo, 7, 8, 88, 108
 - POO**, 88
 - Portabilidad, 49
 - Postfija, 215, 216, 220
 - Postcondición, 12, 16-18
 - Precondición, 12, 16-18
 - Prefija, 215
 - Prioridad, 244
 - cola de, 214-252
 - Problemas de programación, 4
 - resolución, 4
 - Procedimiento, 86
 - recursivo, 263
 - Programa, 15, 29
 - construcción, 52, 61
 - corrección, 22

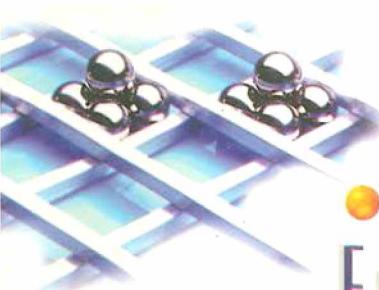
- obsoleto, 15
- reglas para pruebas, 17
- verificación, 16
- Programación, 5
 - a gran escala, 9
 - a pequeña escala, 9
 - equipos, 41
 - estilo, 28
 - estructurada, 116
 - facilidad de uso, 27
 - imperativa, 98
 - orientada a objetos, 9
 - paradigma, 88
 - segura contra fallos, 26
- Programación orientada a objetos, 5, 7, 9, 683
- PROLOG, 89
- Prueba, 3, 13, 15, 42
 - reglas, 17
- Pseudocódigo, 13
- Puntero, 125, 127-130
 - asignación, 135
 - comparación, 139
 - declaración, 127
 - iniciación, 135
 - lista enlazada, 160
 - naturaleza, 138
 - paso con parámetros, 141
 - variable, 127, 134, 160
- Recursión
- Recursividad, 5, 263
 - algoritmos recursivos, 263-265
 - cuándo no utilizar, 264
 - divide y vencerás, 267
 - eliminación de, 266
 - el viajante de comercio, 298
 - generación de permutaciones, 296
 - implementación, 270-277
 - objetos recursivo, 263
 - ocho reinas, 286, 288
 - problema del laberinto, 293
 - problema de la mochila, 290
 - problema de la selección óptima, 297
 - problema de los matrimonios estables, 300-306
 - procedimiento recursivo, 263
- Redundancia, 24
- release*, 146
- Requisitos, 11
 - especificación, 11
- Reutilización, 110
- Simula, 98
- Smalltalk, 114
- Software, 15
 - ciclo de vida, 3, 9, 15
 - construcción, 106
 - evolución, 15
 - IC, 110
 - iteración, 15
 - sistemas de, 23
- Solapamiento, 75
- sort*, 500
- Subprograma, 29
- TAD, 5, 7, 24, 84, 87, 99-102, 207, 227, 426
 - implementación, 101
 - ventajas, 101
- Tecnologías Orientadas a Objetos, 118
 - beneficios, 119
- testing*, 3, 42
- Transportabilidad, 3, 49
- Tipo Abstracto de Dato, 5. *Véase TAD*
- Tipo genérico pointer, 142
- Torres de Hanoi, 267-269
 - concepto, 267
 - resuelto sin recursividad, 271
 - traza de un segmento, 269
- Transportabilidad, 3, 49
- Turbo Pasacal,
 - depuración, 783, 790
 - orientada a objetos, 794
 - depurador integrado, 788
 - editor, 765-767, 779
 - errores, 783
 - códigos, 797
 - mensajes, 797
 - tratamiento, 793
 - guía de referencia, 807
 - menús, 769
 - Compile, 773
 - Debug, 774
 - Edit, 770
 - Help, 778
 - Options, 775

- Run, 774
- Search, 771
- Window, 777

- UCP**, 24, 53, 91
- Unidad, 25, 53, 91, 739-764
 - cabecera, 54
 - compilación, 74
 - concepto, 54
 - creación, 57
 - Crt, 66, 753
 - declaraciones, 62
 - DOS, 67
 - estándar, 739
 - estructura, 54, 58, 72
 - excepciones, 73
- Graph, 68
- implementation, 54, 56
- initialization, 54
- interface, 54-55
- DOS, 741-744
- Crt, 740

- Graph, 740
- Lifo, 241
- Overlay, 77-80, 740
- Printer, 68, 740, 741
- Situación, 69
- Strings, 760
- System, 66, 740
 - uso, 61
 - circular, 64
 - utilización, 65
 - ventajas, 57
- uses, 63

- Vademécum de matemáticas, 73
- var**, 31
- variable, 31
 - dinámica, 126
 - puntero, 125, 127-130, 160
 - operaciones con, 161
 - referenciada, 160
- Vector dinámico, 172
- Verificación, 13



Estructura de Datos

Algoritmos, abstracción y objetos

Luis Joyanes Aguilar • Ignacio Zahonero Martínez

El objetivo fundamental de esta obra es el diseño y construcción de **estructuras de datos** junto con la definición, diseño e implementación de **algoritmos** eficientes, así como las técnicas modernas de resolución de problemas con computadora. Los autores abordan el estudio de las estructuras de datos bajo el enfoque de los tipos abstractos de datos y objetos como una introducción a la programación orientada a objetos.

El libro está dirigido a estudiantes de Ingeniería Informática y Sistemas Computacionales, así como a licenciaturas de Informática y de Ciencias de la Computación, junto a autodidactas que deseen formarse con rigor y profundidad en la disciplina de estructura de datos, y está pensado para su impartición en uno o dos cuatrimestres (semestres) o bien en un curso de duración anual. El libro contiene gran número de ejercicios y problemas de diferentes niveles de complejidad, que ayudarán al lector desde su iniciación a las estructuras de datos hasta los algoritmos más complejos.

- Contiene todos los *descriptores* propuestos por el Consejo de Universidades de España para los planes de estudios de Ingeniería Informática e Ingeniería Técnica Informática, así como los correspondientes a las recomendaciones de los *curricula* tradicionales de ACM correspondientes a los cursos CS2 y los estándares C102.
- Introduce los conceptos de ingeniería de software para la construcción de grandes programas.
- Contiene una revisión del concepto de unidad en Turbo Borland Pascal.

Otras obras de interés de los autores en McGraw-Hill

Programación en Turbo/Borland Pascal 7. 3.^a edición (Luis Joyanes), 1998.

Pascal y Turbo Pascal: Un enfoque práctico (Luis Joyanes, Ignacio Zahonero y Ángel Hermoso), 1995.

Turbo/Borland Pascal 7: Iniciación y Referencia (Luis Joyanes), 1997.

Fundamentos de programación: Libro de problemas en Pascal y Turbo Pascal (Luis Joyanes, Luis Rodríguez y Matilde Fernández), 1997.



9 788440 120429

<http://www.mcgraw-hill.es>

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

A Subsidiary of The McGraw-Hill Companies



ISBN: 84-481-2042-6