Informe tecnico DSIC-DD/4/00, DSIC, UPV, 2000

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
UNIVERSIDAD POLITÉCNICA DE VALENCIA

P.O. Box: 22012 E-46071 Valencia (SPAIN)

Documentación Docente

Asignatura:	Algoritmos y Estructuras de Datos 3
Temas:	Todos
Curso:	2^o
Centro:	Escuela y Facultad de Informática

Ref. No.:	DD-DSIC-4/00	**Versión:**	1	**Pág.:**	139
Título:	Algoritmos y Estructuras de Datos 3				
Autor(es):	María José Castro Bleda				
Fecha:	Octubre, 2000				
Materia:	Algoritmos y estructuras de datos				

Teoría ☒ **Apuntes** ☐
Problemas ☒
Prácticas de Laboratorio ☐ **Transparencias** ☐

V

V^o B^o
responsable Unidad Docente

Autor(es):

Prólogo

El presente documento recoge los aspectos fundamentales del contenido de la asignatura Algoritmos y Estructuras de Datos 3, que se imparte en segundo curso de la Escuela Universitaria de Informática y la Facultad de Informática de la Universidad Politécnica de Valencia. Muchos de los ejemplos y ejercicios propuestos se han extraído de exámenes de la asignatura.

Índice General

Prólogo **i**

Índice General **iii**

Índice de Tablas **vii**

Índice de Figuras **ix**

1 Divide y Vencerás **1**

1.1 La aproximación Divide y Vencerás (DyV) 3
1.2 Mergesort: Algoritmo de ordenación por fusión 3
  1.2.1 Problema de la ordenación 3
  1.2.2 Mergesort: Estrategia y algoritmo 5
  1.2.3 Análisis de Mergesort 7
  1.2.4 División de subproblemas no equilibrada 8
  1.2.5 Mejoras sobre el Mergesort 9
1.3 Análisis de los algoritmos DyV 10
  1.3.1 Corrección y eficiencia de los algoritmos DyV 10
  1.3.2 Complejidad temporal de los algoritmos DyV 10
1.4 Quicksort: Algoritmo de ordenación por partición 12
  1.4.1 Quicksort: Estrategia y algoritmo 12
  1.4.2 Análisis de Quicksort 14
  1.4.3 Comparación de algoritmos de ordenación 19
1.5 Select: Búsqueda del k-ésimo menor elemento 19
  1.5.1 Select: Estrategia y algoritmo 19
  1.5.2 Análisis de Select 20
1.6 Determinación del umbral de recursión en los algoritmos DyV 23
1.7 Otros problemas 25
  1.7.1 Búsqueda binaria (o dicotómica) 25
  1.7.2 Búsqueda dicotómica en 2D 25
  1.7.3 Búsqueda en cadenas de caracteres 26
  1.7.4 Mínimo de una función cóncava 28
  1.7.5 Búsqueda del máximo y del mínimo 30

1.7.6 Producto de enteros grandes 32
1.7.7 Cálculo de la potencia de un número 34
1.7.8 Multiplicación de matrices 34

2 Grafos y árboles 37
2.1 Grafos: Definiciones básicas 39
2.2 Representación de grafos 41
2.2.1 Matriz de adyacencias 41
2.2.2 Listas de adyacencia 41
2.3 Árboles: Definiciones básicas 46
2.3.1 Árboles enraizados 46
2.4 Árboles binarios 47
2.4.1 Propiedades de los árboles binarios completos 48
2.4.2 Propiedades de los árboles binarios 49
2.5 Representación de árboles 49
2.6 Recorridos de árboles binarios 54
2.6.1 Ejemplo árboles binarios: árbol de expresiones 56
2.7 Recorridos de grafos 57
2.7.1 Recorrido en profundidad de un grafo 57
2.7.2 Recorrido en amplitud de un grafo 59
2.8 Orden topológico en grafos acíclicos 61

3 Conjuntos 65
3.1 Conceptos generales 67
3.1.1 Operaciones con conjuntos 67
3.1.2 Algunos tipos de conjuntos dinámicos 68
3.2 Representaciones básicas 68
3.2.1 Ejemplo: Lista doblemente enlazada 69
3.3 Tablas de dispersión 71
3.3.1 Tablas de direccionamiento directo 71
3.3.2 Tablas de Dispersión 72
3.3.3 Ejercicios 77
3.4 Árboles binarios de búsqueda 79
3.4.1 Búsqueda de una clave en un ABB 80
3.4.2 Mínimo y máximo en un ABB 82
3.4.3 Sucesor y predecesor en un ABB (en secuencia *inorden*) 82
3.4.4 Insertar en un ABB 83
3.4.5 Borrar en un ABB 85
3.4.6 Operaciones con ABBs (diccionarios y colas de prioridad) 87
3.4.7 Ejercicios 87
3.5 Árboles parcialmente ordenados: montículos 89
3.5.1 Implementación de un Heap 89
3.5.2 Operaciones básicas sobre Heaps 90

3.5.3 Reconstrucción de un Heap: Heapify 91
3.5.4 Construir un Heap: Build-Heap 94
3.5.5 Extraer el máximo elemento en un Heap: Heap-Extract-Max 96
3.5.6 Insertar un elemento en un Heap: Heap-Insert 97
3.5.7 Algoritmo de ordenación Heapsort 99
3.5.8 Ejercicios 101
3.6 Implementación de conjuntos. Resumen 102
3.7 Conjuntos disjuntos: MFset 104
3.7.1 Operaciones sobre MFsets 104
3.7.2 Representación en árbol de MFsets 105
3.7.3 Implementación operaciones sobre MFsets 106
3.7.4 Mejoras del coste. Heurísticos 108
3.7.5 Ejercicios 113

4 Algoritmos voraces 115
4.1 Introducción. Ámbito de aplicación 117
4.1.1 Ejemplo: Cajero automático 117
4.2 Esquema general voraz 117
4.3 El problema de la mochila con fraccionamiento 118
4.4 Ejercicios 121

5 Algoritmos de grafos 123
5.1 Algoritmos sobre grafos 125
5.2 Árbol de recubrimiento de coste mínimo 126
5.2.1 Algoritmo de Kruskal 126
5.2.2 Algoritmo de Prim 128
5.2.3 Comparación entre Prim y Kruskal 130
5.3 Caminos de mínimo peso en grafos dirigidos 132
5.3.1 Algoritmo de Dijkstra 132

Bibliografía 139

Índice de Tablas

1.1 Algoritmos de ordenación 4
1.2 Mergesort vs. Quicksort (I) 19
1.3 Mergesort vs. Quicksort (II) 19
1.4 Mergesort vs. Quicksort (III) 19

3.1 Operaciones con conjuntos 67
3.2 Estudio comparativo de costes con listas doblemente enlazadas 70
3.3 Operaciones con tablas de direccionamiento directo 71
3.4 Operaciones con tablas de dispersión 75
3.5 Operaciones con ABBs 80
3.6 Operaciones con Heaps 91
3.7 Coste operaciones de conjuntos con distintas estructuras 102
3.8 Operaciones con MFsets 104

Índice de Figuras

1.1 Esquema DyV . . . 3
1.2 Algoritmo Insertionsort . . . 4
1.3 Algoritmo Selectionsort . . . 4
1.4 Funcionamiento del algoritmo Mergesort . . . 5
1.5 Algoritmo Mergesort . . . 5
1.6 Algoritmo Merge . . . 6
1.7 Traza del algoritmo Merge . . . 6
1.8 Traza del algoritmo Mergesort . . . 7
1.9 DyV: División de subproblemas equilibrada . . . 8
1.10 DyV: División de subproblemas no equilibrada . . . 9
1.11 Funcionamiento del algoritmo Quicksort . . . 13
1.12 Algoritmo Quicksort . . . 13
1.13 Algoritmo Partition . . . 13
1.14 Traza del algoritmo Partition . . . 14
1.15 Traza del algoritmo Quicksort . . . 15
1.16 Algoritmo Select . . . 20
1.17 Traza del algoritmo Select . . . 21
1.18 Algoritmo SelectIt . . . 21
1.19 Algoritmo SelectUm . . . 24
1.20 Algoritmos BinSrch y BinSrchIt . . . 26
1.21 Algoritmo BinSrch2D . . . 27
1.22 Algoritmo SrchString2 . . . 28
1.23 Algoritmo Max-Min_DyV . . . 31
1.24 Traza del algoritmo Max-Min_DyV . . . 31
1.25 Algoritmo MultG . . . 32
1.26 Algoritmos ExpoSec y ExpoDyV . . . 34

2.1 Ejemplo de grafo dirigido . . . 39
2.2 Ejemplo de grafo no dirigido . . . 39
2.3 Ejemplos de subgrafos . . . 40
2.4 Ejemplo de componentes conexas . . . 41
2.5 Representación de grafos con matrices de adyacencia . . . 42
2.6 Representación de grafos con listas de adyacencia . . . 43
2.7 Ejemplo de árbol, grafo y bosque . . . 46

2.8 Ejemplo de árbol enraizado 47
2.9 Altura de un árbol 47
2.10 Altura de árboles binarios 48
2.11 Ejemplo de árbol binario completo 49
2.12 Representación de árboles (I) 50
2.13 Representación de árboles (II) 51
2.14 Representación de árboles (III) 52
2.15 Representación de árboles binarios (I) 53
2.16 Representación de árboles binarios (II) 54
2.17 Algoritmos Preorden, Inorden y Postorden 55
2.18 Ejemplo de recorridos de árboles 55
2.19 Árbol de expresiones 56
2.20 Algoritmo DFS 57
2.21 Recorrido en profundidad de un grafo: Ejemplo 58
2.22 Algoritmo BFS 59
2.23 Recorrido en amplitud de un grafo: Ejemplo 60
2.24 Ejemplo de ordenamiento topológico 61
2.25 Algoritmo OrdenTopológico 62
2.26 Orden topológico en grafos acíclicos: Ejemplo 63

3.1 Tabla de direccionamiento directo 72
3.2 Tablas de dispersión: Uso de una función de dispersión 73
3.3 Tablas de dispersión: Ejemplo de colisiones 73
3.4 Tablas de dispersión: Resolución de colisiones por encadenamiento 74
3.5 Tablas de dispersión: Ejemplo direccionamiento abierto 78
3.6 ABB: Ejemplo 79
3.7 ABB: Algoritmo Tree-Search 80
3.8 ABB: Ejemplo Tree-Search 81
3.9 ABB: Algoritmo Tree-Search recursivo 81
3.10 ABB: Ejemplo búsqueda mínimo 82
3.11 ABB: Algoritmo Tree-Successor 83
3.12 ABB: Ejemplo sucesor 83
3.13 ABB: Algoritmo Tree-Insert 84
3.14 ABB: Ejemplo Tree-Insert 84
3.15 ABB: Ejemplo borrado 86
3.16 ABB: Algoritmo Tree-Delete 86
3.17 Heap: Ejemplo 89
3.18 Heap: Implementación de un Heap 90
3.19 Heap: Ejemplo Heapify 92
3.20 Heap: Algoritmo Heapify 93
3.21 Heap: Ejemplo Build-Heap 94
3.22 Heap: Algoritmo Build-Heap 95
3.23 Heap: Ejemplo Heap-Extract-Max 96

3.24 Heap: Algoritmo Heap-Extract-Max 97
3.25 Heap: Ejemplo Heap-Insert 98
3.26 Heap: Algoritmo Heap-Insert 98
3.27 Algoritmo Heapsort 99
3.28 Ejemplo Heapsort 100
3.29 MFset: Ejemplo 105
3.30 MFset: Representación 106
3.31 MFset: Ejemplos operaciones 107
3.32 MFset: Algoritmos MF-Build-Set, MF-Merge y MF-Find 107
3.33 MFset: Caso peor 109
3.34 MFset: Algoritmo MF-Merge por rango 110
3.35 MFset: Ejemplo MF-Merge por rango 111
3.36 MFset: Algoritmo MF-Find con compresión de caminos 112
3.37 MFset: Ejemplo MF-Find con compresión de caminos 112

4.1 Esquema Voraz 118
4.2 Algoritmo Mochila-fraccional 119

5.1 Ejemplo de árbol de recubrimiento 126
5.2 Algoritmo de Kruskal 128
5.3 Algoritmo de Prim 130
5.4 Ejemplo de grafo dirigido y ponderado 132
5.5 Algoritmo de Dijkstra, versión preliminar 134
5.6 Traza del algoritmo de Dijkstra 134
5.7 Corrección del algoritmo de Dijkstra 135
5.8 Algoritmo de Dijkstra 136
5.9 Algoritmo de Dijkstra, recuperación camino 137

Capítulo 1

Divide y Vencerás

Objetivos

- Estudio de la estrategia de diseño de algoritmos recursivos DyV.
- Estudio de los algoritmos de ordenación rápida Mergesort y Quicksort.
- Estudio de otros algoritmos clásicos: multiplicación de matrices, búsquedas...

Contenidos

1 La aproximación Divide y Vencerás (DyV)
2 Mergesort: Ordenación por fusión
3 Quicksort: Ordenación por partición
4 Select: Búsqueda del k-ésimo menor elemento
5 Otros problemas

Bibliografía

- *Introduction to Algorithms*, de Cormen, Leiserson y Rivest (capítulos 1.3, 8 y 10)
- *Fundamentos de Algoritmia*, de Brassard y Bratley (capítulo 7)
- *Computer algorithms*, de Horowitz, Sahni y Rajasekaran (capítulo 3)

1.1 La aproximación Divide y Vencerás (DyV)

Supongamos que tenemos un problema de talla n que queremos resolver. La técnica DyV consiste en *dividir* el problema en un cierto número de problemas más pequeños (*subproblemas*), *solucionar (vencer)* esos subproblemas y después *combinar* las soluciones de los subproblemas para buscar la solución del problema original.

Si al dividir el problema en subproblemas, éstos aún son de una talla “suficientemente grande”, entonces podemos volver a aplicar la técnica de DyV a estos subproblemas. Normalmente, estos subproblemas son la misma naturaleza que el problema del cual proceden, lo que justifica la naturaleza recursiva de estos esquemas.

DIVIDIR el problema en subproblemas.

VENCER (resolver) los subproblemas de forma recursiva. Si éstos son de un tamaño suficientemente pequeño, resolverlos de forma directa.

COMBINAR las soluciones para obtener la solución al problema original.

Figura 1.1: Esquema DyV recursivo.

- Parte “díficil” o “dura” del problema: dividir (ejem.: Quicksort) o combinar (ejem.: Mergesort).
- Eficiencia: problema de la división equilibrada.

1.2 Mergesort: Algoritmo de ordenación por fusión

1.2.1 Problema de la ordenación

Entrada: Una secuencia de n números $\{a_1, a_2, \ldots, a_n\}$ dispuestos en un vector A.

Salida: Una permutación (reordenación) $\{a'_1, a'_2, \ldots, a'_n\}$ de la secuencia de entrada tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Ejemplo de instancia: $\{31, 41, 59, 26, 41, 58\} \rightarrow \{26, 31, 41, 41, 58, 59\}$.

Ejercicio. Traza. Realiza una traza de Insertionsort y Selectionsort con el vector $A = \{5, 2, 4, 6, 1, 3, 2, 6\}$.

Tabla 1.1: Algunos algoritmos de ordenación ("por comparación"): inserción, selección directa, Mergesort, Quicksort, Heapsort.

Algoritmo	**Coste**	**Mem. auxiliar**
Insertionsort (véase fig. 1.2)	$\Omega(n), O(n^2)$	–
Selectionsort (véase fig. 1.3)	$\Theta(n^2)$	–
Mergesort (véase fig. 1.5)	$\Theta(n \log n)$	Sí
Heapsort (véase fig. 3.27)	$\Theta(n \log n)$	–
Quicksort (véase fig. 1.12)	$\Omega(n \log n), O(n^2), \Theta(n \log n)$	–

```
Insertionsort $(A, p, r)$
  for $j := p + 1$ to $r$ do
    $key := A[j]$ {Insertar $A[j]$ en $A[1..j-1]$}
    $i := j - 1$
    while $(i > 0) \wedge (A[i] > key)$ do
      $A[i+1] := A[i]$
      $i := i - 1$
    end while
    $A[i+1] := key$
  end for
```

Figura 1.2: Algoritmo de ordenación Insertionsort. Llamada inicial: Insertionsort$(A, 1, n)$, siendo n el número de elementos del vector A. Coste: $\Omega(n), O(n^2)$.

```
Selectionsort $(A, p, r)$
  for $j := p$ to $r - 1$ do
    $min\_i := j$; $min\_key := A[j]$
    for $i := j + 1$ to $r$ do
      if $A[i] < min\_key$ then
        $min\_i := i$; $min\_key := A[i]$
      end if
    end for
    $A[min\_i] := A[j]$; $A[j] := min\_key$
  end for
```

Figura 1.3: Algoritmo de ordenación Selectionsort. Llamada inicial: Selectionsort$(A, 1, n)$, siendo n el número de elementos del vector A. Coste: $\Theta(n^2)$.

1.2.2 Mergesort: Estrategia y algoritmo

El algoritmo de ordenación Mergesort sigue exactamente el esquema de DyV:

> **Dividir:** Divide la secuencia de n elementos a ordenar en dos subsecuencias de $n/2$ elementos cada una.
>
> **Vencer:** Ordena las dos subsecuencias recursivamente utilizando Mergesort.
>
> **Combinar:** Combina las dos subsecuencias ordenadas para generar la solución.

Nota: La recursión finaliza cuando la secuencia a ordenar contiene un único elemento, en cuyo caso la secuencia ya está ordenada.

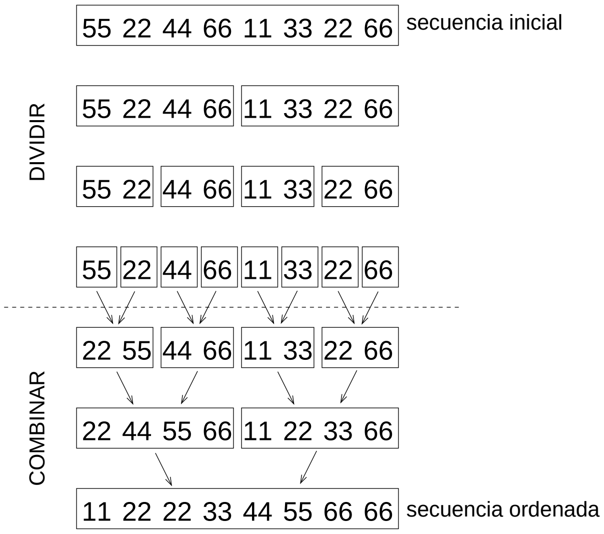


Figura 1.4: Funcionamiento del algoritmo Mergesort con la secuencia {55, 22, 44, 66, 11, 33, 22, 66}.

```
Mergesort($A, p, r$)
  if $p < r$ then
    $q := \lfloor (p+r)/2 \rfloor$        {Divide}
    Mergesort($A, p, q$)         {Resuelve}
    Mergesort($A, q+1, r$)       {Resuelve}
    Merge($A, p, q, r$)           {Combina}
  end if
```

Figura 1.5: Algoritmo de ordenación Mergesort. Llamada inicial: Mergesort$(A, 1, n)$, siendo n el número de elementos del vector A.

```
Merge($A, p, q, r$)
  $i := p; j := q + 1; k := 1$
  while $(i \leq q) \wedge (j \leq r)$ do
    if $A[i] \leq A[j]$ then
      $B[k] := A[i]; i := i + 1$
    else
      $B[k] := A[j]; j := j + 1$
    end if
    $k := k + 1$
  end while
  while $(i \leq q)$ {...quedan todavia elementos de $A[i..q]$} do
    $B[k] := A[i]; i := i + 1; k := k + 1$
  end while
  while $(j \leq r)$ {...quedan todavia elementos de $A[j..r]$} do
    $B[k] := A[j]; j := j + 1; k := k + 1$
  end while
  for $k := p$ to $r$ do {...volcamos $B$ en $A$}
    $A[k] := B[k - p + 1]$
  end for
```

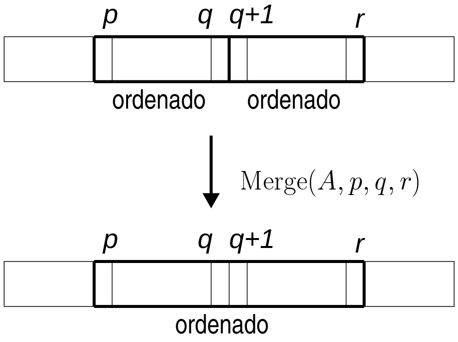


Figura 1.6: Algoritmo de fusión Merge. Coste: $\Theta(n)$.

A

22	44	55	66	**11**	22	33	66
22	44	55	66	–	**22**	33	66
–	**44**	55	66	–	**22**	33	66
–	**44**	55	66	–	–	**33**	66
–	**44**	55	66	–	–	–	**66**
–	–	**55**	66	–	–	–	**66**
–	–	–	**66**	–	–	–	**66**
–	–	–	–	–	–	–	**66**
–	–	–	–	–	–	–	–

B

11							
11	**22**						
11	22	**22**					
11	22	22	**33**				
11	22	22	33	**44**			
11	22	22	33	44	**55**		
11	22	22	33	44	55	**66**	
11	22	22	33	44	55	66	**66**

Figura 1.7: Traza del algoritmo Merge con las subsecuencias {22, 44, 55, 66} y {11, 22, 33, 66}.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Mergesort($A, 1, 8$)	55	22	44	66	11	33	22	66
Mergesort($A, 1, 4$)	55	22	44	66				
Mergesort($A, 1, 2$)	55	22						
Mergesort($A, 1, 1$)	55							
Mergesort($A, 2, 2$)		22						
Merge($A, 1, 1, 2$)//	**22**	**55**						
Mergesort($A, 3, 4$)			44	66				
Mergesort($A, 3, 3$)			44					
Mergesort($A, 4, 4$)				66				
Merge($A, 3, 3, 4$)//			**44**	**66**				
Merge($A, 1, 2, 4$)//	**22**	**44**	**55**	**66**				
Mergesort($A, 5, 8$)					11	33	22	66
...						...		
					11	**22**	**33**	**66**
Merge($A, 1, 4, 8$)//	**11**	**22**	**22**	**33**	**44**	**55**	**66**	**66**

Figura 1.8: Traza del algoritmo Mergesort con la secuencia $\{55, 22, 44, 66, 11, 33, 22, 66\}$.

Ejercicio. Traza. Realiza una traza de Mergesort con $A = \{3, 41, 52, 26, 38, 57, 9, 49\}$.

1.2.3 Análisis de Mergesort

Problema del balance en el Mergesort: Una propiedad importante que ha de cumplir un algoritmo DyV para resultar eficiente es que la partición del espacio de trabajo ha de dar lugar a dos mitades lo más parecidas posible respecto a la talla, tal y como ocurre en Mergesort.

Sea $T(n)$ el tiempo de ejecución de Mergesort. Si $n = 1$, el tiempo es constante. Si $n > 1$ (supondremos por simplicidad que n es una potencia de 2):

Dividir: Calcula el índice medio del vector: $\Theta(1)$.

Vencer: Resuelve recursivamente los dos subproblemas, cada uno de ellos de talla $n/2$: $2T(n/2)$.

Combinar: Merge (fusión) de un vector de n elementos: $\Theta(n)$.

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ 2T(n/2) + c_2 n + c_3, & \text{si } n > 1 \end{cases}$$

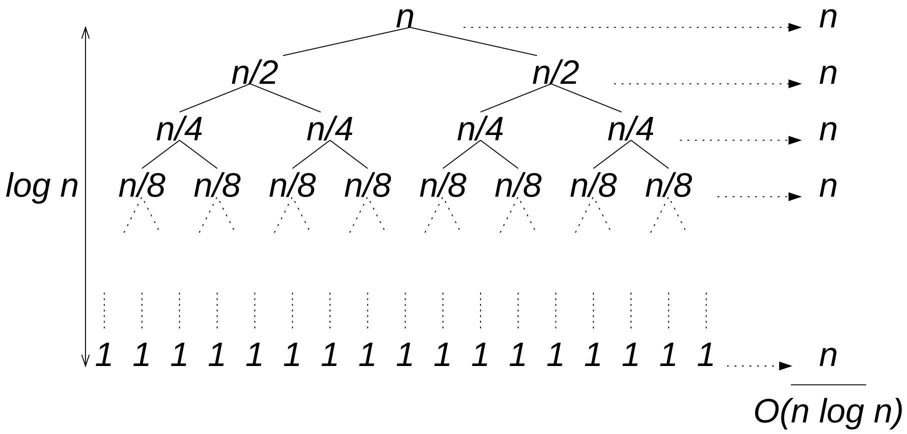


Figura 1.9: División de subproblemas equilibrada.

Podemos ahora resolver la ecuación por desplegado:

$$\begin{aligned}
T(n) &= 2T(n/2) + c_2 n + c_3 \\
&= 2(2T(n/2^2) + c_2 n/2 + c_3) + c_2 n + c_3 = 2^2 T(n/2^2) + 2c_2 n + c_3(2+1) \\
&= 2^2(2T(n/2^3) + c_2 n/2^2 + c_3) + 2c_2 n + c_3(2+1) = 2^3 T(n/2^3) + 3c_2 n + c_3(2^2+2+1) \\
&\vdots \quad \{i \text{ iteraciones}\} \\
&= 2^i T(n/2^i) + i c_2 n + c_3(2^{i-1} + \ldots + 2^2 + 2 + 1)
\end{aligned}$$

El desplegado finaliza cuando i es igual a $\log n$, pues entonces $n/2^i$ es igual a 1, y $T(1)$ no se ha definido recursivamente. Entonces,

$$\begin{aligned}
T(n) &= \{n/2^i = 1, i = \log n\}; \sum_{j=0}^{i-1} 2^j = 2^i - 1 \\
&= nc_1 + c_2 n \log n + c_3(n-1) \in \Theta(n \log n)
\end{aligned}$$

Por tanto, el coste de Mergesort es $\Theta(n \log n)$.

1.2.4 División de subproblemas no equilibrada

Supongamos un algoritmo Mergesort_bad, donde la división de problemas en subproblemas no esté equilibrada:

```
Mergesort_bad $(A, p, r)$
  if $p < r$ then
    Mergesort_bad $(A, p, p)$
    Mergesort_bad $(A, p + 1, r)$
    Merge$(A, p, p, r)$
  end if
```

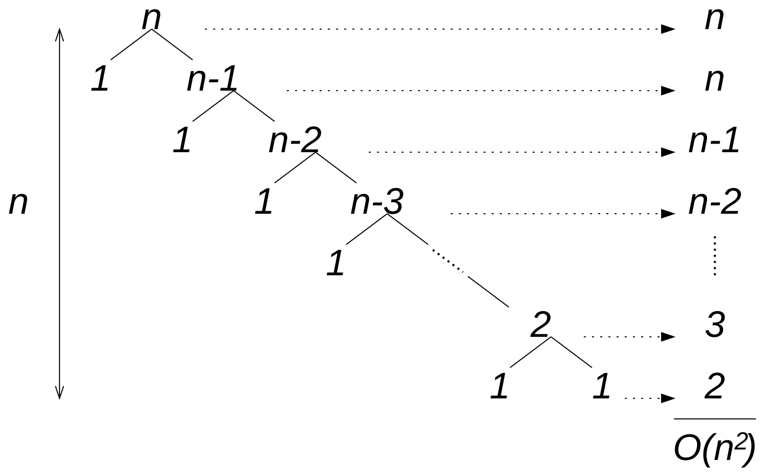


Figura 1.10: División de subproblemas no equilibrada.

La relación de recurrencia del algoritmo Mergesort_bad sería:

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ T(1) + T(n-1) + c_2 n + c_3, & \text{si } n > 1 \end{cases}$$

$$T(n) = \begin{cases} c_1, & \text{si } n \leq 1; \\ T(n-1) + c_2 n + c_4, & \text{si } n > 1 \end{cases}$$

¡¡Coste: $\Theta(n^2)$!!

1.2.5 Mejoras sobre el Mergesort

- Uno de los inconvenientes es que se necesita un espacio $2n$ debido al Merge. Existe un algoritmo de fusión que no utiliza un vector adicional en $O(n)$, pero con una constante muy alta que no lo hace apropiado [HSR98].
- Otro inconveniente es la talla de la pila de recursión: la máxima profundidad de la pila será proporcional a $\log n$.
- Una mejora sobre el Mergesort consiste en no realizar llamadas recursivas cuando la talla del vector es pequeña, haciendo una ordenación por inserción o selección directa.
- Se pueden evitar movimientos redundantes de datos haciendo llamadas alternativamente sobre el vector original y el vector auxiliar.

1.3 Análisis de los algoritmos DyV

1.3.1 Corrección y eficiencia de los algoritmos DyV

Corrección de los algoritmos DyV

- La corrección se realiza por inducción sobre el tamaño del problema.
- Consiste en demostrar que la operación “combinar” es correcta si las soluciones de las que parte también lo son.

Corrección de Mergesort

El tamaño del problema será el número de elementos del vector, $n = r - p + 1$.

B.I.: Si $n = 1$, el vector está ordenado.

H.I.: Para tallas menores que n, Mergesort ordena el vector.

P.I.: Para una talla n:

- Primera llamada: $n' = q - p + 1 = [(p + r)/2] - p + 1 = [(r - p + 1)/2]$
- Segunda llamada: $n'' = r - (q + 1) + 1 = r - [(p + r)/2] = [(r - p)/2]$
- n' y n'' son menores que n. Por H.I., Mergesort ordena los subvectores.
- Por la especificación de Merge, también se ordena el vector.

1.3.2 Complejidad temporal de los algoritmos DyV

- **Dividir** el problema original de talla n en a subproblemas de tamaño n/b. Coste $D(n)$.
  Mergesort: Calcula el índice medio del vector. Coste $\Theta(1)$.
- **Resolver** los a subproblemas de tamaño n/b. Coste $aT(n/b)$.
  Mergesort: Resuelve recursivamente los dos subproblemas de tamaño $n/2$. Coste $2T(n/2)$.
- **Combinar**. Coste $C(n)$.
  Mergesort: Merge (fusión) de un vector de n elementos. Coste $\Theta(n)$.

Relación de recurrencia general:

$$T(n) = \begin{cases} c, & \text{si } n \leq n_0; \\ aT(n/b) + D(n) + C(n), & \text{si } n > n_0 \end{cases}$$

$$T(n) = \begin{cases} c, & \text{si } n \leq n_0; \\ aT(n/b) + \Theta(n^k), & \text{si } n > n_0 \end{cases}$$

Teorema: La solución a la ecuación $T(n) = aT(n/b) + O(n^k)$, con $a \geq 1$ y $b > 1$, es: (Dem. [Wei95, pág. 383])

$$\begin{array}{rl} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \log n) & \text{si } a = b^k \\ O(n^k) & \text{si } a < b^k \end{array}$$

Teorema: La solución a la ecuación $T(n) = aT(n/b) + O(n^k \log_p n)$, con $a \geq 1$, $b > 1$ y $p \geq 0$, es:

$$\begin{array}{rl} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \log^{p+1} n) & \text{si } a = b^k \\ O(n^k \log^p n) & \text{si } a < b^k \end{array}$$

Ejercicio. Costes. Demuestra el teorema para $k = 1$: $T(n) = aT(n/b) + O(n)$, con $a \geq 1$ y $b > 1$, es:

$$\begin{array}{rl} O(n^{\log_b a}) & \text{si } a > b \\ O(n \log n) & \text{si } a = b \\ O(n) & \text{si } a < b \end{array}$$

Ejemplos:

a) $a = 1, b = 2, k = 1.$ $T(n) \in O(n)$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ T(n/2) + n, & \text{si } n > 1 \end{cases}$$

b) $a = 2, b = 2, k = 1.$ $T(n) \in O(n \log n)$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ 2T(n/2) + n, & \text{si } n > 1 \end{cases}$$

c) $a = 3, b = 2, k = 1.$ $T(n) \in O(n^{\log_2 3}) = O(n^{1.59})$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ 3T(n/2) + n, & \text{si } n > 1 \end{cases}$$

d) $a = 4, b = 2, k = 1. \qquad T(n) \in O(n^2)$

$$T(n) = \begin{cases} 1, & \text{si } n \leq 1; \\ 4T(n/2) + n, & \text{si } n > 1 \end{cases}$$

1.4 Quicksort: Algoritmo de ordenación por partición

1.4.1 Quicksort: Estrategia y algoritmo

Quicksort, al igual que Mergesort, se basa en la estrategia DyV. La diferencia estriba en que la parte costosa del algoritmo está en el paso de “dividir”. Los tres pasos de la recursión para ordenar un subvector $A[p..r]$ son:

Dividir: El vector $A[p..r]$ se particiona (reorganiza) en dos subvectores $A[p..q]$ y $A[q+1..r]$, de forma que los elementos de $A[p..q]$ son menores o iguales que los de $A[q+1..r]$. El índice q se calcula también en el procedimiento de partición.

Vencer: Los dos subvectores $A[p..q]$ y $A[q+1..r]$ se ordenan recursivamente utilizando Quicksort.

Combinar: Como los subvectores se ordenan en su lugar correspondiente, no hay que hacer nada para combinar las soluciones: el vector completo $A[p..r]$ ya está ordenado.

El objetivo de la función de partición es reorganizar el vector $A[p..r]$ dejando en $A[p..q]$ elementos menores o iguales que en $A[q+1..r]$, de forma que las particiones resulten lo más equilibradas posible. Posibles soluciones:

- Mejor algoritmo posible: Elegir como pivote la mediana (deja ambas partes equilibradas) → algoritmo muy costoso
- Elegir como pivote el primer elemento $A[p]$ → algoritmo Partition
- Otras opciones: pivote aleatorio, mediana de un subconjunto de elementos, ...

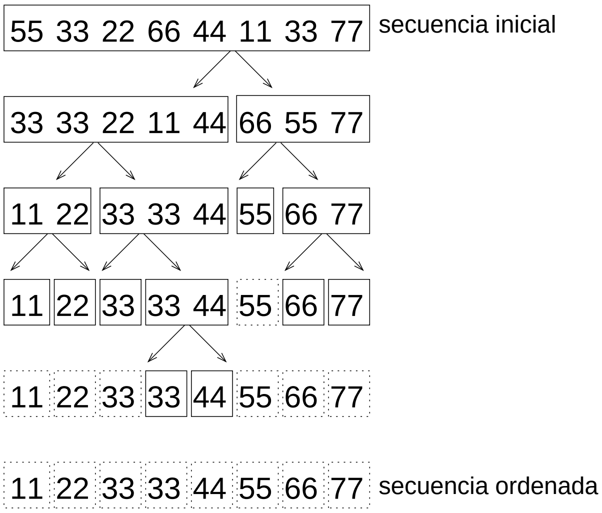


Figura 1.11: Funcionamiento del algoritmo Quicksort con la secuencia {55, 33, 22, 66, 44, 11, 33, 77}.

```
Quicksort($A, p, r$)
   if $p < r$ then
      $q$ := Partition($A, p, r$)
      Quicksort($A, p, q$)
      Quicksort($A, q + 1, r$)
   end if
```

Figura 1.12: Algoritmo Quicksort. Llamada inicial: Quicksort$(A, 1, n)$, siendo n el número de elementos del vector A.

```
Partition($A, p, r$)
   $piv := A[p]; i := p - 1; j := r + 1$
   repeat
      repeat
         $j := j - 1$
      until $A[j] \leq piv$
      repeat
         $i := i + 1$
      until $A[i] \geq piv$
      if $i < j$ then
         $A[i] \longleftrightarrow A[j]$
      end if
   until $i \geq j$
   return $j$
```

Figura 1.13: Algoritmo de partición Partition. Coste: $\Theta(n)$.

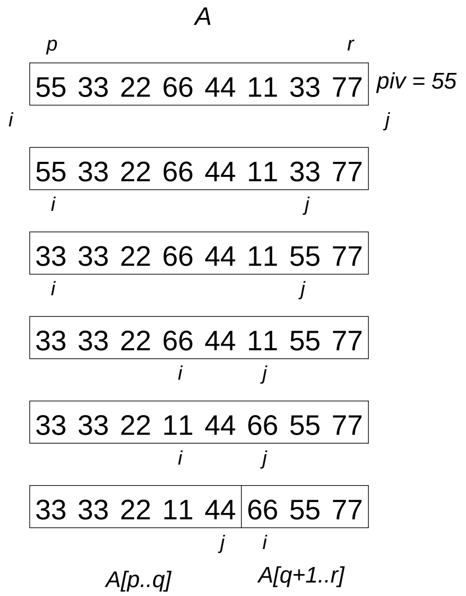


Figura 1.14: Traza del algoritmo Partition con la secuencia $\{55, 33, 22, 66, 44, 11, 33, 77\}$.

1.4.2 Análisis de Quicksort

Sea n el número de elementos del vector. El coste de Partition es $O(n)$ y reorganiza el vector en dos subvectores de tamaño i y $(n - i)$. La relación de recurrencia es:

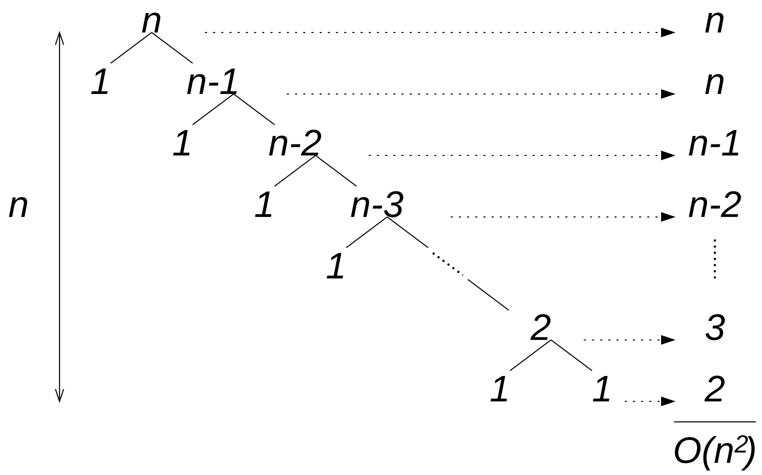
$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ T(i) + T(n-i) + k'n, & \text{si } n > 1 \end{cases}$$

Quicksort: Análisis (Peor caso: Partición desequilibrada)

Peor caso: supongamos que todos los elementos del vector son diferentes y que están ordenados de menor a mayor. En este caso, el algoritmo Partition divide el vector original en dos subvectores: uno con un único elemento (el mínimo, que ha actuado como pivote), mientras que el otro contiene el resto de elementos.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Quicksort$(A, 1, 8)$	55	33	22	66	44	11	33	77
Partition$(A, 1, 8) \to 5$	33	33	22	11	44	66	55	77
Quicksort$(A, 1, 5)$	33	33	22	11	44			
Partition$(A, 1, 5) \to 2$	11	22	33	33	44			
Quicksort$(A, 1, 2)$	11	22						
Partition$(A, 1, 2) \to 1$	11	22						
Quicksort$(A, 1, 1)//$	**11**							
Quicksort$(A, 2, 2)//$		**22**						
Quicksort$(A, 3, 5)$			33	33	44			
Partition$(A, 3, 5) \to 3$			33	33	44			
Quicksort$(A, 3, 3)//$			**33**					
Quicksort$(A, 4, 5)$				33	44			
Partition$(A, 4, 5) \to 4$				33	44			
Quicksort$(A, 4, 4)//$				**33**				
Quicksort$(A, 5, 5)//$					**44**			
Quicksort$(A, 6, 8)$						66	55	77
...							...	
						55	**66**	**77**
	11	**22**	**33**	**33**	**44**	**55**	**66**	**77**

Figura 1.15: Traza del algoritmo Quicksort con la secuencia $\{55, 33, 22, 66, 44, 11, 33, 77\}$.

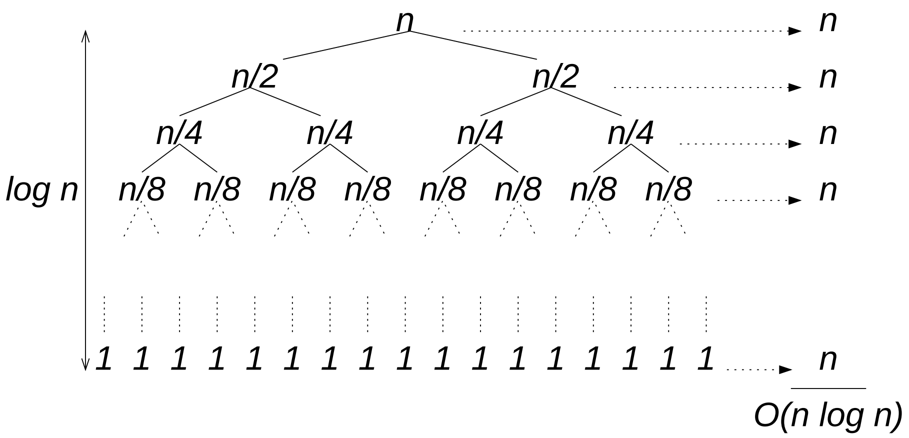


$$T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ T(1) + T(n-1) + k'n, & \text{si } n > 1 \end{cases}$$

$$
\begin{aligned}
T(n) &= T(n-1) + c_2 n + c_3 \\
&= (T(n-2) + c_2(n-1) + c_3) + c_2 n + c_3 \\
&= T(n-2) + ((n-1)+n)c_2 + 2c_3 \\
&= (T(n-3) + c_2(n-2) + c_3) + ((n-1)+n)c_2 + 2c_3 \\
&= T(n-3) + ((n-2)+(n-1)+n)c_2 + 3c_3 \\
&\vdots \\
&= T(1) + c_2 \sum_{i=2}^{n} i + (n-1)c_3 \\
&= c_1 + c_2(n+2)(n-1)/2 + c_3(n-1) \in O(n^2)
\end{aligned}
$$

Quicksort: Análisis (Mejor caso: Partición equilibrada)

Mejor caso: será áquel que deja en el que en cada llamada a Partition deja las dos partes equilibradas.

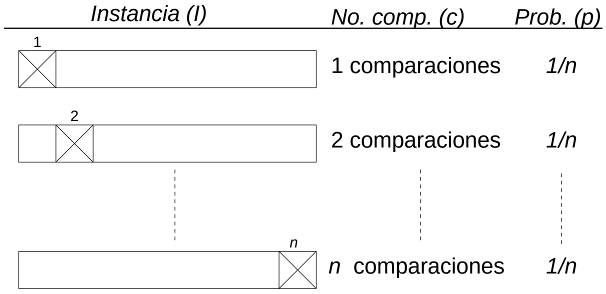


$$
T(n) = \begin{cases} k, & \text{si } n \le 1; \\ 2T(n/2) + k'n, & \text{si } n > 1 \end{cases}
$$

Análisis de coste promedio

Ejemplo: Sea un vector A de enteros con n elementos (todos diferentes). Supongamos que deseamos conocer la posición de un elemento x que sabemos puede estar ocupando cualquiera de las posiciones.

```
pos $(A, x)$
    $i := 1$
    while $i \le n$ do
        if $A[i] \neq x$ then
            $i := i + 1$
        else
            return $i$
        end if
    end while
```



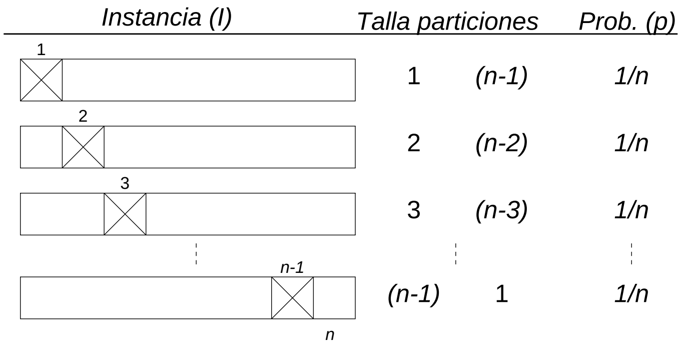
Instancia (I)	No. comp. (c)	Prob. (p)
1	1 comparaciones	1/n
2	2 comparaciones	1/n
⋮	⋮	⋮
n	n comparaciones	1/n

Coste medio:

$$\begin{aligned} \sum_{\forall i \in I} p_i c_i &= 1\frac{1}{n} + 2\frac{1}{n} + \ldots + n\frac{1}{n} \\ &= \sum_{i=1}^{n} i\frac{1}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2} \in O(n) \end{aligned}$$

Quicksort: Análisis (Caso promedio)

El coste promedio de Quicksort es $O(n \log n)$. Vamos a demostrarlo.



Instancia (I)	Talla particiones		Prob. (p)
1	1	(n-1)	1/n
2	2	(n-2)	1/n
3	3	(n-3)	1/n
⋮	⋮		⋮
n-1, n	(n-1)	1	1/n

$$
\begin{aligned}
T(n) = k'n \quad &+ \quad \frac{1}{n}(T(1) + T(n-1)) \\
&+ \quad \frac{1}{n}(T(2) + T(n-2)) \\
&+ \quad \frac{1}{n}(T(3) + T(n-3)) \\
&\vdots \\
&+ \quad \frac{1}{n}(T(n-1) + T(1)) \\
= k'n \quad &+ \quad \frac{2}{n}\sum_{i=1}^{n-1} T(i)
\end{aligned}
$$

$$
T(n) = \begin{cases} k, & \text{si } n \leq 1; \\ k'n + 2/n \sum_{i=1}^{n-1} T(i), & \text{si } n > 1 \end{cases}
$$

Se puede demostrar que, $\forall n \geq 2$, $T(n) \leq k''n \log n$, siendo $k'' = 2k' + k$. Por tanto, el coste promedio de Quicksort es $O(n \log n)$.

$\boxed{n = 2}$

$$
T(2) \quad = \quad 2k' + \sum_{i=1}^{1} T(i) = 2k' + k \leq (2k' + k)2 \log 2
$$

$\boxed{n > 2}$

$$
\begin{aligned}
T(n) \quad &= \quad k'n + 2/n \sum_{i=1}^{n-1} T(i) = k'n + 2/nT(1) + 2/n \sum_{i=2}^{n-1} T(i) \\
(H.I.) \quad &\leq \quad k'n + 2k/n + 2/n \sum_{i=2}^{n-1} k''i \log i \leq k''n \log n \in O(n \log n)
\end{aligned}
$$

1.4.3 Comparación de algoritmos de ordenación

Tabla 1.2: Mergesort vs. Quicksort (Sun 10/30). Media de resultados de 50 instancias aleatorias (en milisegundos) [HSR98].

n	1000	2000	3000	4000	5000	10000
Mergesort	72.8	167.2	275.1	378.5	500.6	1073.6
Quicksort	36.6	85.1	138.9	205.7	269.9	645.2

Tabla 1.3: Mergesort vs. Quicksort (Sun 10/30). Peor resultado de 50 instancias aleatorias (en milisegundos) [HSR98].

n	1000	2000	3000	4000	5000	10000
Mergesort	105.7	206.4	335.2	422.1	589.9	1167.6
Quicksort	41.6	97.1	158.6	244.9	397.8	738.1

Tabla 1.4: Quicksort con el primer elemento como pivote o con pivote aleatorio (Sun 10/30). Instancias ordenadas (peor caso teórico). Media de resultados de 100 ejecuciones (en milisegundos) [HSR98].

n	1000	2000	3000	4000	5000
Quicksort	195.5	759.2	1728	3165	4829
RQuicksort	9.4	21.0	30.5	41.6	52.8

1.5 Select: Búsqueda del k-ésimo menor elemento

1.5.1 Select: Estrategia y algoritmo

Problema: Dado un vector de n elementos, el problema de la selección consiste en buscar el k-ésimo menor elemento.

Solución directa: ordenar los n elementos y acceder al k-ésimo. Coste: $O(n \log n)$.

Solución DyV: ¿Es posible encontrar un algoritmo más eficiente? Utilizando la idea del algoritmo Partition:

Dividir (Partition): El vector $A[p..r]$ se particiona (reorganiza) en dos subvectores $A[p..q]$ y $A[q+1..r]$, de forma que los elementos de $A[p..q]$ son menores o iguales que el pivote y los de $A[q+1..r]$ son mayores o iguales.

Combinar: Si $k \leq q$, entonces el k-ésimo menor estará en $A[p..q]$. Si no, estará en $A[q+1..r]$.

Vencer: Buscamos en el subvector correspondiente haciendo llamadas recursivas al algoritmo.

Nota: La recursión finaliza cuando el subvector contiene un único elemento, en cuyo caso habremos encontrado el k-ésimo menor.

```
Select($A, p, r, k$)
  if $p = r$ then
    return $A[p]$
  end if
  $q :=$ Partition($A, p, r$)
  $i := q - p + 1$   {$i$ es el número de elementos en el primer subvector}
  if $k \leq i$ then
    return Select($A, p, q, k$)   {...buscamos en $A[p..q]$}
  else
    return Select($A, q+1, r, k-i$)   {...buscamos en $A[q+1..r]$}
  end if
```

Figura 1.16: Algoritmo Select. Llamada inicial: Select$(A, 1, n, k)$, siendo n el número de elementos del vector A.

Ejercicio. Traza. Realiza una traza de Select y SelectIt con $A = \{31, 23, 90, 0, 77, 52, 49, 87, 60, 15\}$ y $k = 7$.

1.5.2 Análisis de Select

Supondremos que todos los elementos son diferentes.

Análisis de Select (Peor caso: Desequilibrado)

Peor caso: vector ordenado de forma no decreciente y buscamos el elemento mayor ($k = n$). Coste: $O(n^2)$.

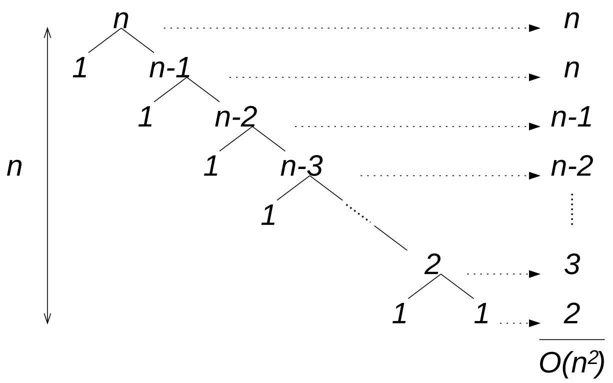
Instancia: $A = \{55, 33, 22, 66, 44, 11, 33, 77\}$; $k = 3$
Vector ordenado: $A' = \{11, 22, \mathbf{33}, 33, 44, 55, 66, 77\}$

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Select$(A, 1, 8, 3)$	55	33	22	66	44	11	33	77
Partition$(A, 1, 8) \to 5$	33	33	22	11	44	66	55	77
Select$(A, 1, 5, 3)$	33	33	22	11	44			
Partition$(A, 1, 5) \to 2$	11	22	33	33	44			
Select$(A, 3, 5, 1)$			33	3	44			
Partition$(A, 3, 5) \to 3$			33	33	44			
Select$(A, 3, 3, 1)$			**33**					

Figura 1.17: Traza del algoritmo Select con la secuencia $\{55, 33, 22, 66, 44, 11, 33, 77\}$ y $k = 3$.

```
SelectIt($A, p, r, k$)
  while $p < r$ do
    $q$ := Partition($A, p, r$)
    if $k \leq q$ then
      $r := q$ {...buscamos en $A[p..q]$}
    else
      $p := q + 1$ {...buscamos en $A[q + 1..r]$}
    end if
  end while
  return $A[p]$
```

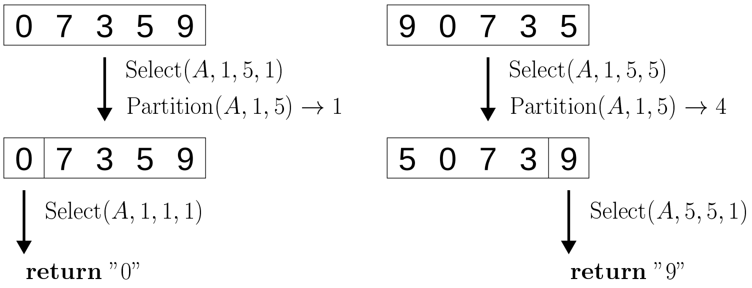
Figura 1.18: Algoritmo SelectIt. Llamada inicial: Select$(A, 1, n, k)$, siendo n el número de elementos del vector A.



$$T(n) = \begin{cases} c, & \text{si } n \leq 1; \\ T(n-1) + c'n, & \text{si } n > 1 \end{cases}$$

Análisis de Select (Mejor caso)

Mejor caso: el elemento a buscar es el menor ($k = 1$) o mayor ($k = n$) y éste actúa como pivote → Una única llamada a Partition. Coste: $O(n)$.



Select: Análisis (Caso promedio)

- Peor caso $\in O(n^2)$
- Mejor caso $\in O(n)$
- Promedio: con ciertas asunciones de aleatoriedad, el algoritmo Select tiene un coste $\in \Theta(n)$ [CLR90, pág. 188], [HSR98, pág. 167]

Teorema [HS95, pág. 167] Sea $T^k(n)$ el coste promedio de Select$(A, 1, n, k)$. Este promedio se calcula sobre las $n!$ diferentes permutaciones de n elementos diferentes.

Definimos

- $T(n) = \frac{1}{n}\sum_{k=1}^{n} T^k(n)$ $\quad T(n)$ es el tiempo promedio de Select
- $R(n) = \max_k(T^k(n))$, siendo $T(n) \leq R(n)$

Vamos a demostrar que $T(n) \in O(n)$.

Demostración: Una llamada a Partition: el pivote es el k-ésimo menor elemento con probabilidad $1/n$, $1 \leq k \leq n$. Esta afirmación se sigue de la asunción de distribución equiprobable de los elementos en el vector.

El coste de Select es:

$$T^k(n) \leq cn + \frac{1}{n}(\sum_{i=1}^{k-1} T^{k-1}(n-k) + \sum_{i=k+1}^{n} T^k(k-1)), \qquad n \geq 2$$

$$
\begin{aligned}
R(n) &\leq cn + \frac{1}{n}\max_k(\sum_{i=1}^{k-1} R(n-k) + \sum_{i=k+1}^{n} R(k-1)), \qquad n \geq 2 \\
&\leq cn + \frac{1}{n}\max_k(\sum_{n-k+1}^{n-1} R(k) + \sum_{k}^{n-1} R(k))
\end{aligned}
$$

Por inducción, se sigue que $R(n) < 4cn$, por tanto, $T(n) \in O(n)$.

1.6 Determinación del umbral de recursión en los algoritmos DyV

- Una mejora sobre los algoritmos DyV consiste en no realizar llamadas recursivas cuando la talla del vector es pequeña.

```
if $talla > n_0$ then
    algoritmo DyV
else
    algoritmo directo
end if
```

- Ejemplos:
  - Mergesort y Quicksort: cuando la talla es menor que un umbral, ordenar el subvector por inserción o selección directa.
  - Select: cuando la talla es menor que un umbral, ordenar el subvector por inserción o selección directa y devolver directamente el k-ésimo menor elemento.

Este umbral de recursión se debe determinar experimentalmente.

Ejercicio. Select: Variante umbral. Implementa el algoritmo Select utilizando un umbral de recursión. Determina experimentalmente el umbral de recursión.

```
SelectUm($A, p, r, k$)
  if $r - p + 1 > n_0$ then
    $q :=$ Partition($A, p, r$)
    $i := q - p + 1$   {i es el número de elementos en el primer subvector}
    if $k \le i$ then
      return SelectUm($A, p, q, k$)   {...buscamos en $A[p..q]$}
    else
      return SelectUm($A, q + 1, r, k - i$)   {...buscamos en $A[q + 1..r]$}
    end if
  else {si el subvector es menor que el umbral, ordenamos}
    Insertionsort ($A, p, r$)
    return $A[k]$
  end if
```

Figura 1.19: Algoritmo Select variante umbral de recursión.

1.7 Otros problemas

- Búsqueda binaria (o dicotómica)
- Búsqueda dicotómica en 2D
- Búsqueda en cadenas de caracteres
- Mínimo de una función cóncava
- Búsqueda del máximo y del mínimo
- Producto de enteros grandes
- Cálculo de la potencia de un número
- Multiplicación de matrices

1.7.1 Búsqueda binaria (o dicotómica)

Problema: Dado un vector ordenado A por orden no decreciente de n enteros, buscar un elemento x.

Solución directa: búsqueda secuencial. Coste: $O(n)$.

Solución DyV: aprovechando que el vector está ordenado, se puede llegar a un algoritmo $O(\log n)$.

Ejercicio. Traza. Realiza una traza del algoritmo BinSrch y BinSrchIt con A = {-5, -2, 0, 3,8, 8, 9, 12, 12, 26, 31} y $x = 12$.

1.7.2 Búsqueda dicotómica en 2D

Problema: Sea M una matriz cuadrada de enteros cuya dimensión es n. Esta matriz cumple que $(\forall j : 1 \leq j < n : M[i,j] < M[i,j+1])$ para toda fila i, $1 \leq i \leq n$, y además $(\forall i : 1 \leq i < n : M[i,n] < M[i+1,1])$. La siguiente matriz cumple esta condición:

-30	-14	-2	0
2	3	10	12
15	16	22	23
25	30	38	45

Dado un elemento x de la matriz, escribe un algoritmo "Divide y Vencerás" que en $O(\log n)$ devuelva la celda de la matriz en la que se encuentra este valor.

Solución: Como los elementos están ordenados por filas de izquierda a derecha y por columnas de arriba a abajo:

- encontrar la fila donde está el elmento con BinSrch y

```
BinSrch($A, p, r, x$)
  if $p = r$ then
    if $x = A[p]$ then return $p$
    else return 0
    end if
  else
    $q := \lfloor (p+r)/2 \rfloor$
    if $x = A[q]$ then
      return q
    else
      if $x < A[q]$ then
        return BinSrch($A, p, q-1, x$)
      else
        return BinSrch($A, q+1, r, x$)
      end if
    end if
  end if
```

```
BinSrchIt($A, p, r, x$)
  $i := p; j := r$
  while $i \leq j$ do
    $k := \lfloor (i+j)/2 \rfloor$
    if $x = A[k]$ then
      return $k$
    else
      if $x < A[k]$ then
        $j := k - 1$
      else
        $i := k + 1$
      end if
    end if
  end while
  return 0
```

Figura 1.20: Algoritmo BinSrch y su versión iterativa, BinSrchIt. Dado el vector $A[p..r]$ en orden no decreciente, devuelve i si $x = A[i]$; si no, devuelve 0.

- a continuación encontrar la columna, también con BinSrch.

El algoritmo contendrá dos bucles, cada uno de los cuales tiene un coste $O(\log n)$, que es por tanto el coste del algoritmo.

1.7.3 Búsqueda en cadenas de caracteres

Problema: Dada una cadena ordenada x de n símbolos y otra cadena también ordenada y que contiene exactamente 2 símbolos, se desea obtener un índice $j \in \mathbf{N}$ tal que $0 < j < n$, $x_j = y_1$, $x_{j+1} = y_2$, o bien $j = 0$ si y no se encuentra en x.

Siguiendo la estrategia de "Divide y Vencerás", desarrolla algoritmos para resolver el problema propuesto en los siguientes casos:

a) Se asume que en ninguna de las dos cadenas (x, y) hay símbolos repetidos. *Ejem.:*
$x =$"`A D E F H P R W`": $y =$"`H P`" $\rightarrow j = 5$; $y =$"`F P`" $\rightarrow j = 0$.

b) Se asume que tanto x como y pueden tener símbolos repetidos. *Ejem.:*
$x =$"`A D D D D D R S V`": $y =$"`D D`" $\rightarrow j = 2$; $y =$"`D S`" $\rightarrow j = 0$.
Nótese que con $y =$"`D D`", otras soluciones correctas pueden ser: $j = 3$ o $j = 4$.

En ambos casos los costes temporales de los algoritmos a desarrollar deberán ser $O(\log n)$ en el peor de los casos.

```
BinSrch2D($x, M$)
  $i := 1$
  $j := n$
  $encontrado := false$
  **repeat**
    $f := \lfloor (i+j)/2 \rfloor$
    **if** $x \geq M[f,1]$ and $x \leq M[f,n]$ **then**
      $encontrado := true$
    **else**
      **if** $x < M[f,1]$ **then**
        $j := f - 1$
      **else**
        $i := f + 1$
      **end if**
    **end if**
  **until** $encontrado$
  {El elemento $x$ está en la fila $f$}
  $i := 1$
  $j := n$
  **repeat**
    $c := \lfloor (i+j)/2 \rfloor$
    **if** $x < M[f,c]$ **then**
      $j := c - 1$
    **else**
      $i := c + 1$
    **end if**
  **until** $M[f,c] = x$
  **return** $(f, c)$
```

Figura 1.21: Algoritmo BinSrch2D.

Solución: El caso a) puede resolverse del mismo modo que el caso b), o bien usando directamente un algoritmo convencional de búsqueda dicotómica que busque y_1 en x. En este último caso, sea i el resultado obtenido de la búsqueda dicotómica ($i = 0$ si y_1 no está en x). La solución al problema propuesto será:

$$j = \begin{cases} i, & \text{si } 0 < i < |x| \wedge x_{i+1} = y_2; \\ 0, & \text{en cualquier otro caso.} \end{cases}$$

El caso b) (y también el a)) pueden resolverse mediante "Divide y Vencerás" usando una técnica similar a la empleada en la búsqueda dicotómica: Sea $x_{i..k}$ una subcadena de x en la que se quiere encontrar y. Se divide la cadena $x_{i..k}$ en dos subcadenas $x_{i..j}$ y $x_{j+1..k}$ de (aproximadamente) la misma talla calculando $j = \lfloor (i+k)/2 \rfloor$. Si $y_1 = x_j \wedge y_2 = x_{j+1}$

la solución es j. En caso contrario, si $x_j \geq y_2$, se tiene la garantía que y sólo puede estar en $x_{i..j}$, mientras que si $x_{j+1} \leq y_1$ se puede asegurar que y sólo puede estar en $x_{j+1..k}$ (por estar x e y ordenadas). Finalmente, si ninguna de estas condiciones se cumple, entonces y no está en x y se dará $j = 0$ como solución.

Siguiendo este planteamiento, la función SrchString2 descrita a continuación asume que tanto x como y son datos globalmente accesibles y almacenados en sendos vectores. Invocando a esta función como SrchString2$(1, |x|)$, se obtiene un natural que es una solución al problema propuesto.

```
SrchString2($i, k$)
  if $k - i + 1 < 2$ then
    return 0
  else
    $j = \lfloor (i + k)/2 \rfloor$
    if $y_1 = x_j \wedge y_2 = x_{j+1}$ then
      return $j$
    else
      if $x_j \geq y_2$ then
        return SrchString2($i, j$)
      else
        if $x_{j+1} \leq y_1$ then
          return SrchString2($j + 1, k$)
        else
          return 0
        end if
      end if
    end if
  end if
```

Figura 1.22: Algoritmo SrchString2.

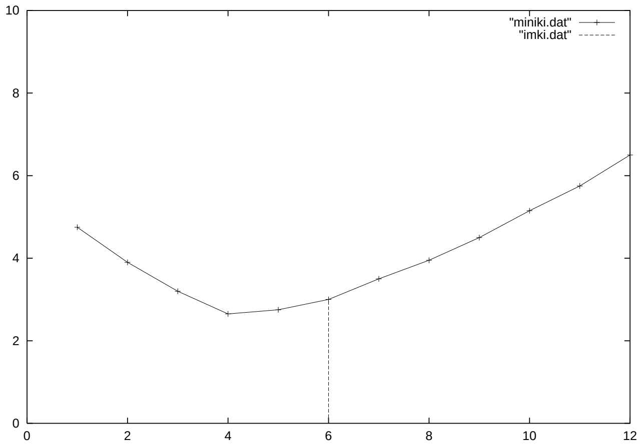
El coste de este algoritmo responde a una recurrencia clásica de Divide y Vencerás de la forma $T(n) \leq T(n/2) + c$, cuya solución es $O(\log n)$.

1.7.4 Mínimo de una función cóncava

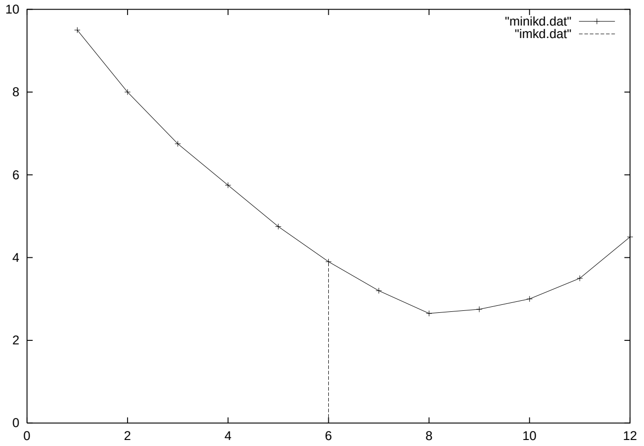
Problema: Dado un vector A de n enteros positivos que se ajusta al perfil de una curva cóncava, $\forall k, 1 \leq k \leq n \mid (\forall j, 1 \leq j < k, A[j] > A[j+1]) \wedge (\forall j, k < j \leq n, A[j-1] < A[j])$, se desea encontrar el mínimo.

Solución directa: buscar el mínimo. Coste: $O(n)$.

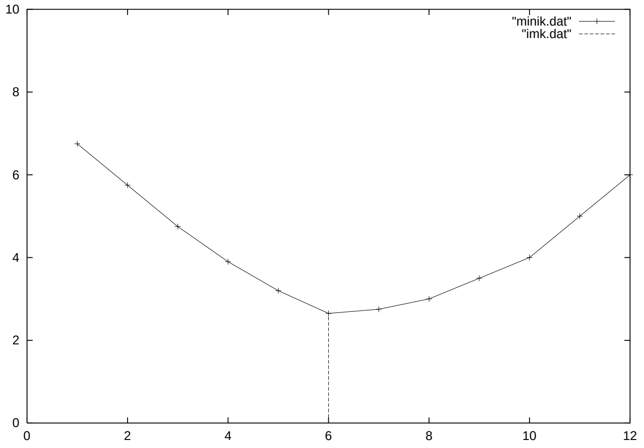
Solución DyV: aprovechando la forma de la curva, se puede llegar a un algoritmo $O(\log n)$.



$A[k-1] < A[k] < A[k+1]$
mín. en la mitad izquierda



$A[k-1] > A[k] > A[k+1]$
mín. en mitad derecha



$A[k-1] > A[k] < A[k+1]$
mín. en k

Ejercicio. Función cóncava. Escribe un algoritmo para obtener el mínimo de un conjunto de puntos que se ajustan a una función cóncava. Estudia su eficiencia e impleméntalo.

$$T(n) = \begin{cases} c_1, & \text{si } n = 1; \\ T(n/2) + c_2, & \text{si } n > 1 \end{cases}$$

$$T(n) \in O(\log n)$$

1.7.5 Búsqueda del máximo y del mínimo

Problema: Búsqueda del máximo y del mínimo en un vector no ordenado de n enteros (siendo n potencia de 2).

Solución: Una solución inmediata para el problema sería:

```
Max-Min($A$)
   $max := A[1]; min := A[1]$
   for $i := 2$ to $n$ do
      if $A[i] > max$ then
         $max := A[i]$
      end if
      if $A[i] < min$ then
         $min := A[i]$
      end if
   end for
   return $(max, min)$
```

Solución DyV: El algoritmo Max-Min_DyV encuentra el máximo y el mínimo con menos comparaciones.

Número de comparaciones entre elementos del vector:

- Max-Min → $2(n-1)$
- Max-Min_DyV → $T(n) = 3/2n - 2$. Reducción: 25%

$$T(n) = \begin{cases} 1, & \text{si } n = 2; \\ 2T(n/2) + 2, & \text{si } n > 2 \end{cases}$$

```
Max-Min_DyV($A, p, r$)
   if $r - p \leq 1$ then
      if $A[r] > A[p]$ then
         $zmax := A[r]; zmin := A[p]$
      else
         $zmax := A[p]; zmin := A[r]$
      end if
      return $(zmax, zmin)$
   else
      {Buscamos el min y el max por separado en las dos
      mitades y después buscamos el máximo de los dos max
      y el mínimo de los dos min}
      $q := \lfloor (p + r)/2 \rfloor$
      $(xmax, xmin) :=$ Max-Min_DyV($A, p, q$)
      $(ymax, ymin) :=$ Max-Min_DyV($A, q + 1, r$)
      $zmax := \max(xmax, ymax)$
      $zmin := \min(xmin, ymin)$
      return $(zmax, zmin)$
   end if
```

Figura 1.23: Algoritmo Max-Min_DyV. Llamada inicial: Max-Min_DyV($A, 1, n$).

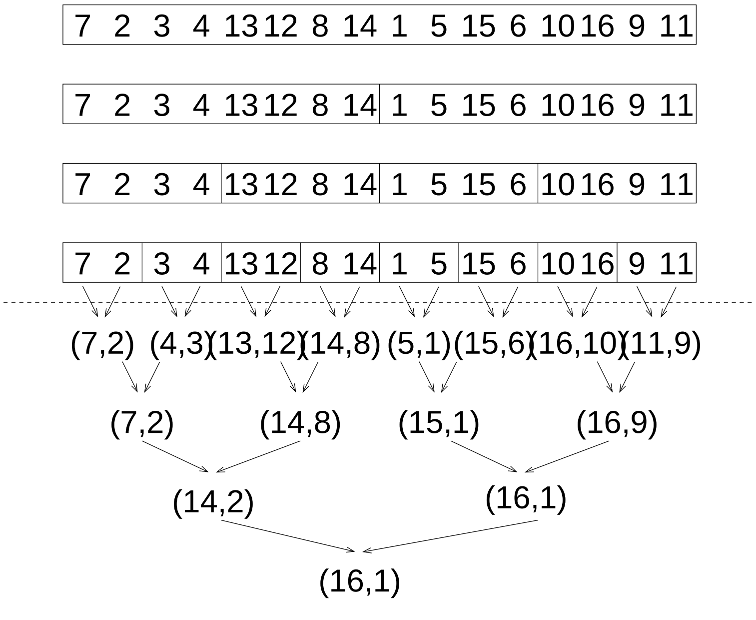


Figura 1.24: Traza del algoritmo Max-Min_DyV con la instancia {7, 2, 3, 4, 13, 12, 8, 14, 1, 5, 15, 6, 10, 16, 9, 11}.

1.7.6 Producto de enteros grandes

Problema: Dados dos enteros u y v de n cifras, calcular el producto de u y v, $u \cdot v$.

Algoritmo directo: $\Theta(n^2)$

Resolución por DyV (primera propuesta):

a) Dividimos cada número en otros dos (de igual talla), donde $0 \leq x < 10^s, 0 \leq z < 10^s, s = \lfloor n/2 \rfloor$.

$$\boxed{\begin{array}{l} u = 10^s w + x \\ v = 10^s y + z \end{array}}$$

Ejem.: $u = 9387421 = 9387 \cdot 10^3 + 421$ y $v = 3324578 = 3324 \cdot 10^3 + 578$.

b) Realizamos el producto con esta descomposición: $\boxed{u \cdot v = 10^{2s} wy + 10^s(wz + xy) + xz}$

→ El algoritmo MultG ha reducido la multiplicación de 2 números de n cifras a 4 multiplicaciones de números de $n/2$ cifras. ¿Coste?

```
MultG($u, v$){Dados los enteros u y v, calcula $u \cdot v$}
  $n$ := max(length($u$), lenght($v$))
  if $n = 1$ then
    return $u \cdot v$
  else
    $s := \lfloor n/2 \rfloor$
    $w := \lfloor u/10^s \rfloor$ ; $x := u - w \cdot 10^s$
    $y := \lfloor v/10^s \rfloor$ ; $z := v - y \cdot 10^s$
    return MultG($w, y$) $\cdot 10^{2s}$ + (MultG($w, z$) +
    MultG($x, y$)) $\cdot 10^s$ + MultG($x, z$)
  end if
```

Figura 1.25: Algoritmo MultG.

El coste de este algoritmo es (las operaciones que aparecen en el algoritmo pueden hacerse con un coste lineal: desplazamientos (multiplicaciones por potencias de 10), adiciones y sustraciones):

$$T(n) = \begin{cases} c, & \text{si } n \leq k; \\ 4T(n/2) + c'n + c'', & \text{si } n > k \end{cases}$$

Ejercicio. Coste. Demuestra que $T(n) \in O(n^2)$. Realiza una traza con $u \cdot v = 373472243 \cdot 323$.

Resolución por DyV (Mejora del algoritmo MultG): Menos multiplicaciones, más sumas

$$\begin{aligned}
\boxed{u \cdot v} &= 10^{2s}wy + 10^{s}(wz + xy) + xz \\
&\quad \{r = (w+x)(y+z) = wy + (wz + xy) + xz\} \\
&= 10^{2s}wy + 10^{s}((w+x)(y+z) - wy - xz) + xz \\
&= \boxed{10^{2s}p + 10^{s}(r - p - q) + q}
\end{aligned}$$

donde:

1: $p = wy$; $p := \mathrm{MultG}(w, y)$
2: $q = xz$; $q := \mathrm{MultG}(x, z)$
3: $r = (w+x)(y+z)$; $r := \mathrm{MultG}((w+x), (y+z))$

El coste de este nuevo algoritmo es:

$$T(n) = \begin{cases} c, & \text{si } n \leq k; \\ 3T(n/2) + c'n + c'', & \text{si } n > k \end{cases}$$

Ejercicio. Algoritmo. Demuestra que $T(n) \in O(n^{\log_2 3}) = O(n^{1.59})$.

Algoritmo mejorado MultG $\in O(n^{1.59})$:

- Se ha reducido el producto de dos números de n cifras a:
  3 multiplicaciones de números de $n/2$ cifras
  +
  desplazamientos (multiplicaciones por potencias de 10), adiciones y sustracciones
- Aumenta el número de sumas: “merece la pena” cuando los operandos son *muy grandes* (en ese caso, el tiempo de las sumas es “despreciable” frente al tiempo de una multiplicación)
- Aplicación a *criptografía* [BB97, sección 7.8]

1.7.7 Cálculo de la potencia de un número

Problema: Dados los enteros a y n, calcular $x = a^n$.

Algoritmo directo

```
ExpoSec($a, n$)
  {Calcula $x = a \cdot a \cdot \ldots \cdot a$}
  $r := a$
  for $i := 1$ to $n - 1$ do
    $r := a \cdot r$
  end for
  return $r$
```

$n - 1$ productos → $\Theta(n)$
(problemas de desbordamiento)

Algoritmo DyV

```
ExpoDyV($a, n$)
  {Calcula $x = a^{n/2} \cdot a^{n/2}$ recursivamente}
  if $n = 0$ then
    return 1
  else
    if $n$ es impar then
      return $a \cdot$ ExpoDyV($a, n - 1$)
    else
      $b :=$ ExpoDyV($a, n/2$)
      return $b \cdot b$
    end if
  end if
```

El coste es $\Theta(\log n)$

Figura 1.26: Algoritmos de cálculo de la potencia, dados los enteros a y n, devuelve a^n. Solución secuencial, ExpoSec, y solución DyV, ExpoDyV.

1.7.8 Multiplicación de matrices

Problema: Multiplicación de matrices cuadradas $n \times n$: $C = A \times B$.

Algoritmo directo: $\Theta(n^3)$

```
for $i := 1$ to $n$ do
  for $j := 1$ to $n$ do
    $C[i, j] := 0$
    for $k := 1$ to $n$ do
      $C[i, j] := A[i, k] \cdot B[k, j] + C[i, j]$
    end for
  end for
end for
```

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{12}b_{22} \end{pmatrix}$$

$n = 2$: 8 productos y 4 sumas de $n/2 \times n/2 \to \Theta(n^3)$

Resolución por DyV: El algoritmo de Strassen

Consideremos las operaciones siguientes, cada una de las cuales necesita una única multiplicación:

$$\begin{aligned}
m_1 &= (a_{11} + a_{22}) \cdot (b_{11} + b_{22}) \\
m_2 &= (a_{21} + a_{22}) \cdot b_{11} \\
m_3 &= a_{11} \cdot (b_{12} - b_{22}) \\
m_4 &= a_{22} \cdot (b_{21} - b_{11}) \\
m_5 &= (a_{11} + a_{12}) \cdot b_{22} \\
m_6 &= (a_{21} + a_{11}) \cdot (b_{11} + b_{12}) \\
m_7 &= (a_{12} + a_{22}) \cdot (b_{21} + b_{22})
\end{aligned}$$

$$C = \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

$n = 2$: 7 productos y 18 sumas de $n/2 \times n/2 \to \Theta(n^{\log 7}) = \Theta(n^{2.81})$
(Las matrices se pueden sumar y restar en un tiempo $\Theta(n^2)$)

$$T(n) = \begin{cases} k, & \text{si } n = 1; \\ 7T(n/2) + k'n^2, & \text{si } n > 1 \end{cases}$$

Ejercicio. Strassen. Verifica que el producto matricial $A \times B$ está dado por la matriz anterior C. Demuestra que $T(n) \in \Theta(n^{2.81})$.

Capítulo 2

Grafos y árboles

Objetivos

- Definiciones, representación y recorrido de árboles y grafos.

Contenidos

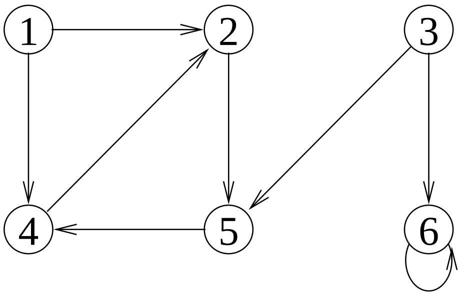
1 Grafos: Definiciones básicas
2 Representación de grafos
3 Árboles: Definiciones básicas
4 Árboles binarios
5 Representación de árboles
6 Recorridos de árboles binarios
7 Recorrido de grafos
8 Orden topológico en grafos acíclicos

Bibliografía

- *Introduction to Algorithms*, de Cormen, Leiserson y Rivest (secciones 5.4, 5.5, 11.4 y 23.1)
- *Estructuras de datos y algoritmos*, de Aho, Hopcroft y Ullman (capítulos 3, 6 y 7)

2.1 Grafos: Definiciones básicas

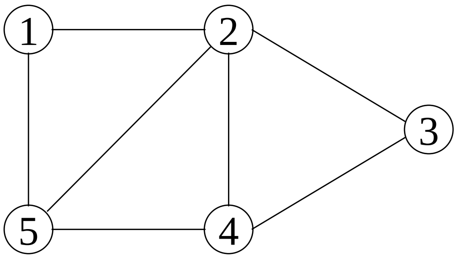
- **Grafo dirigido (g.d.):** es un par $G = (V, A)$ donde V es un conjunto finito de elementos llamados *vértices* y $A \subseteq V \times V$ es un conjunto de "pares ordenados" de vértices llamados *aristas*.
  Si $a = (u, v)$ es una arista de G, se dice que *a entra* o *incide* en v y que *a sale* o *emerge* de u.
  Si (u, v) es una arista de G, se dice que el vértice v es *adyacente* a u.



$$V = \{1, 2, 3, 4, 5, 6\}$$
$$A = \{(1, 2), (1, 4), (2, 5), (3, 5), (3, 6), (4, 2), (5, 4), (6, 6)\}$$

Figura 2.1: Ejemplo de grafo dirigido $G = (V, A)$.

- **Grafo no dirigido (g.n.d.):** es un par $G = (V, A)$ donde V es un conjunto finito de *vértices* y $A \subseteq \{\{u, v\} \mid u, v \in V \wedge v \neq u\}$ es un conjunto de "pares no ordenados" de vértices.
  Equivalentemente, G es un grafo no dirigido si G es un grafo dirigido y $\forall u, v \in V$, $(u, v) \in A \Leftrightarrow (v, u) \in A$.
  Si $a = \{u, v\}$ es una arista no dirigida, se dice que *a une* a u y v y que *a incide* en u y v.
  Si $\{u, v\}$ es una arista de G, se dice que el vértice v es *adyacente* a u. La relación es simétrica.



$$V = \{1, 2, 3, 4, 5\}$$
$$A = \{\{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{4, 5\}\}$$

Figura 2.2: Ejemplo de grafo no dirigido $G = (V, A)$.

En lo que sigue nos referiremos siempre a grafos dirigidos, a menos que se indique lo contrario.

- **Grado:** Para todo vértice v, **grado de entrada** es el número de aristas que inciden en v; **grado de salida** es el número de aristas que emergen de v; **grado** es la suma de los grados de entrada y salida de v. El **grado de un grafo** es el máximo grado de sus vértices.

  Ejem.: En el grafo de la figura 2.1, el grado de entrada del vértice 2 es 2; el grado de salida es 1; el grado del vértice es 3. El grado del grafo es 3.

- **Camino desde un vértice $u \in V$ a un vértice $v \in V$:** es una secuencia $\langle v_0, v_1, \ldots, v_k \rangle$ de vértices de $G = (V, A)$ tal que $v_0 = u$, $v_k = v$, $(v_i, v_{i+1}) \in A$, $0 \leq i < k$. La **longitud de un camino** $\langle v_0, v_1, \ldots, v_k \rangle$ es el número de aristas que lo forman.
- **Camino simple:** es un camino en el que todos sus vértices son distintos.
- **Ciclo:** es un camino simple $\langle v_0, v_1, \ldots, v_k \rangle$ tal que $v_0 = v_k$ y el camino contiene al menos una arista. Un **bucle** es un ciclo de longitud 1.
- **Grafo acíclico:** es un grafo sin ciclos.

  Ejem.: En el grafo de la figura 2.1, el camino $\langle 1, 4, 2, 5 \rangle$ es simple y tiene longitud 3. El camino $\langle 1, 2, 5, 4, 2 \rangle$ no es simple.

  Ejem.: En el grafo de la figura 2.1, el camino $\langle 2, 5, 4, 2 \rangle$ es un ciclo de longitud 3. El ciclo $\langle 6, 6 \rangle$ es un bucle.

- **Subgrafo:** $G' = (V', A')$ es un subgrafo de $G = (V, A)$ si $V' \subseteq V \wedge A' \subseteq A$. Dado $V' \subseteq V$, el subgrafo de G **inducido** por V' es $G' = (V', A')$ tal que $A' = \{(u, v) \in A \mid u, v \in V'\}$.

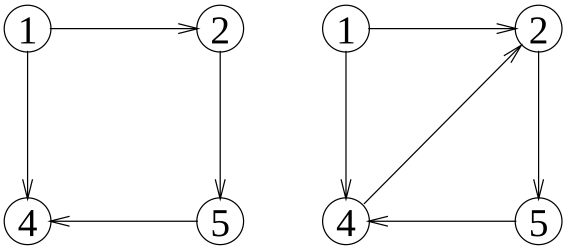


Figura 2.3: Subgrafo y subgrafo inducido $\{1, 2, 4, 5\}$ del grafo de la figura 2.1.

- **Vértice alcanzable desde un vértice u:** es cualquier vértice v para el que existe un camino de u a v.
- Un grafo no dirigido es **conexo** si $\forall u, v \in V$, v es alcanzable desde u. Las **componentes conexas en un grafo** son las clases de equivalencia de vértices según la relación *"ser alcanzable"*.

  Un grafo dirigido es **fuertemente conexo** si $\forall u, v \in V$, v es alcanzable desde u.

Las **componentes fuertemente conexas en un grafo dirigido** son las clases de equivalencia de vértices según la relación *"ser mutuamente alcanzable"*.

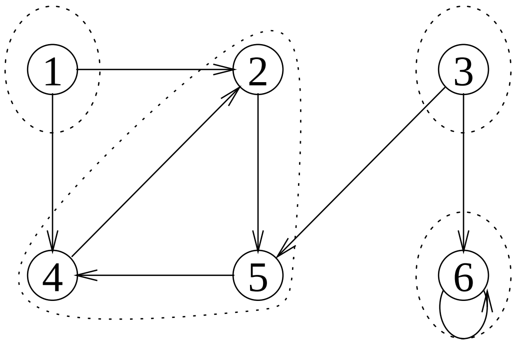


Figura 2.4: Ejemplo de componentes conexas en el grafo de la figura 2.1.

- **Grafo completo:** es un grafo $G = (V, A)$ en el que $\forall u, v \in V$, $u \neq v$, $(u, v) \in A$.
- **Grafo etiquetado:** es un grafo $G = (V, A)$ acompañado de una función $f : A \rightarrow E$, donce E es un conjunto cuyas componentes se denominan *etiquetas*.
- **Grafo ponderado:** es un grafo etiquetado con números reales ($E \equiv \mathbf{R}$).

2.2 Representación de grafos

2.2.1 Matriz de adyacencias

Un grafo $G = (V, A)$ se representa como G: matriz$[V, V]$ de booleanos. La componente $G[u, v]$ es 1 si $(u, v) \in A$; sino $G[u, v] = 0$.

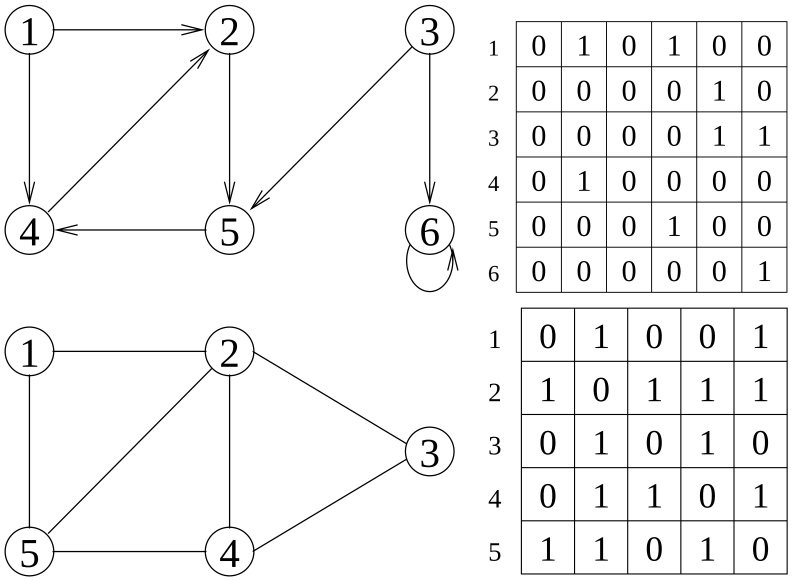
- Memoria: $O(|V|^2) \rightarrow$ grafos densos $|A| \approx |V|^2$.
- Tiempo de acceso: $O(1)$.

2.2.2 Listas de adyacencia

Un grafo $G = (V, A)$ se representa como un *vector de listas de vértices* indexado por vértices; es decir, G: vector$[V]$ de V. Cada componente $G[v]$ es una lista de los vértices emergentes y/o incidentes de/a $v \in V$.

- Memoria: $O(|V| + |A|) \rightarrow$ grafos dispersos $|A| <<< |V|^2$.
- Tiempo de acceso: $O(grado(G))$.

```
#define n 128
typedef char grafMatriz[n][n];
grafMatriz GM;
```



1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Figura 2.5: Representación de un grafo dirigido y no dirigido mediante matriz de adyacencias: Ejemplo.

```
#define n 128
typedef struct arist {
    int estado;
    struct arist *sig;
} Arista;
typedef Arista *grafListas[n];
grafListas GL;
```

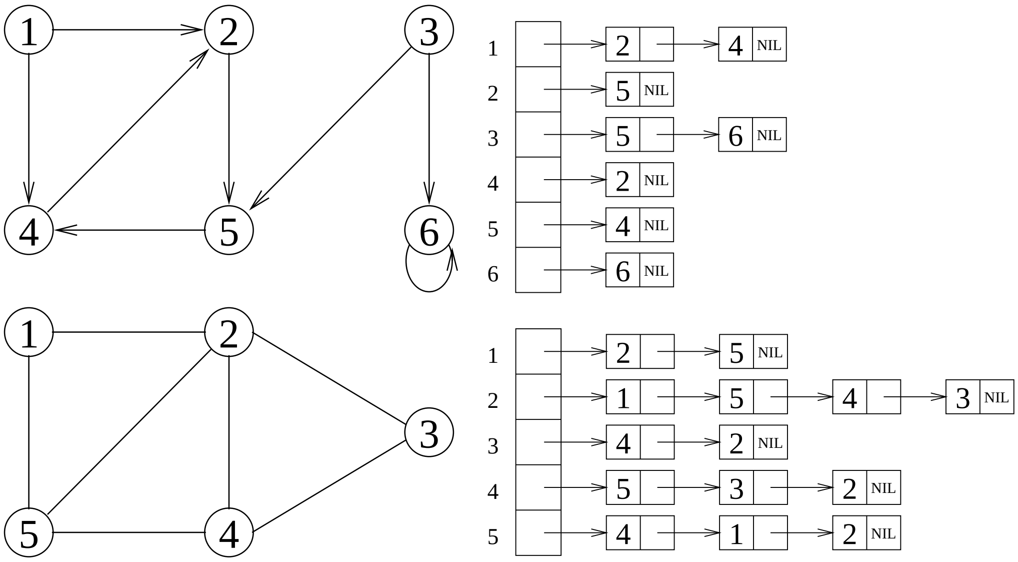
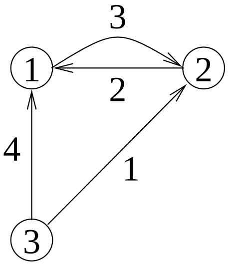


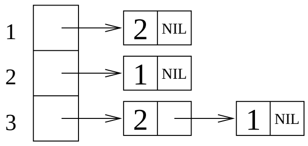
Figura 2.6: Representación de un grafo dirigido y no dirigido mediante listas de adyacencias: Ejemplo.

Ejercicio. Representación de grafos. Supongamos que tenemos un grafo dirigido sin bucles $G = (V, A)$ (donde V es el conjunto de vértices, numerados de 1 a $|V|$, y A es el conjunto de aristas, numeradas de 1 a $|A|$) representado en una matriz M de dimensiones $|V| \times |A|$, donde:

$m_{ij} = 1$ si del vértice i parte la arista j,
$m_{ij} = -1$ si al vértice i llega la arista j,
$m_{ij} = 0$ la arista j no incide en el vértice i.



$$M = \begin{pmatrix} 0 & -1 & 1 & -1 \\ -1 & 1 & -1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$



Escribe un algoritmo que pase de esta forma de representación a una representación en forma de lista de adyacencia. Calcula el coste del algoritmo justificándolo adecuadamente.

Ejercicio. Grafo traspuesto. El grafo *traspuesto* de un grafo dirigido $G = (V, A)$ (V es el conjunto de vértices y A es el conjunto de aristas) es un grafo $G^t = (V, A^t)$ donde $(u, v) \in A^t$ si y sólo si $(v, u) \in A$. Escribe dos algoritmos que permitan construir el grafo traspuesto G^t a partir de otro grafo G: en el primer algoritmo los grafos se representarán mediante matrices de adyacencia, mientras que en el segundo algoritmo los grafos se representarán mediante listas de adyacencia. Calcula el coste computacional de los dos algoritmos.

Ejercicio. Cuadrado de un grafo. El *cuadrado* de un grafo dirigido $G = (V, A)$ se define como $G^2 = (V, A')$ tal que $(u, w) \in A'$ si y solo si para algún $v \in V$ se cumple que $(u, v) \in A$ y $(v, w) \in A$ en el grafo G. Escribe un algoritmo para calcular G^2 a partir de G, suponiendo que este último está representado como una matriz de adyacencia. Calcula el coste temporal del algoritmo resultante, justificándolo adecuadamente.

Ejercicio. Diferencia simétrica. Dados dos grafos, $G_1 = (V_1, A_1)$ y $G_2 = (V_2, A_2)$, con $V_1 = V_2$, se desea obtener un nuevo grafo $G = (V, A)$ que sea la *diferencia simétrica* de ambos, tal que el conjunto de vértices V del grafo resultante coincida con V_1 y V_2, y el conjunto de aristas A contenga las aristas de A_1 que no estén en A_2 junto con las de A_2 que no estén en A_1. Escribe un algoritmo que construya el grafo resultante G, suponiendo que los grafos están representados mediante matrices de adyacencia. Analiza el coste espacial y temporal del algoritmo.

Ejercicio. Grafo bipartido. Se dice que un grafo $G = (V, A)$ es bipartido si:

- Existen un par de conjuntos $V1$, $V2$ tales que V es la unión de $V1$ y $V2$, y la intersección de $V1$ y $V2$ es vacía.
- Para todo arco (a, b) de A, a está en $V1$, y b está en $V2$.

Se debe escribir un algoritmo que diga si un grafo G, del que se supone que sus vértices están numerados de 1 a n, es bipartido:

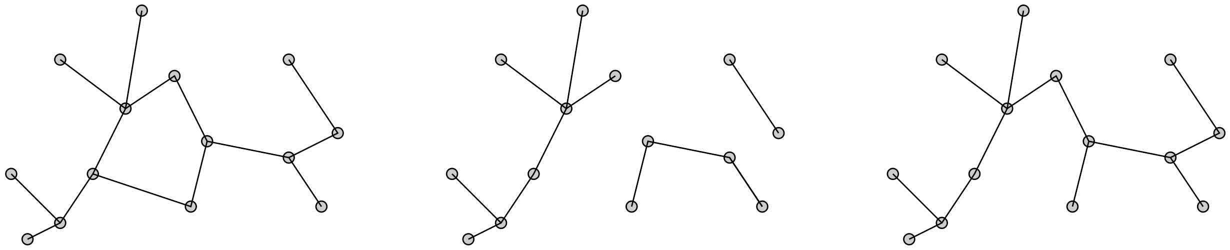
a) En el caso de que G esté representado por su matriz de adyacencia.
b) En el caso de que G esté representado por su lista de adyacencia.

Discute el coste del algoritmo para las dos representaciones.

2.3 Árboles: Definiciones básicas

La estructura de datos “árbol” nos va a servir para representar una colección de objetos. Ejemplos comunes de colecciones de datos representadas con árboles: índice de un libro, árboles genealógicos...

- **Bosque:** es un grafo no dirigido acíclico.
- **Árbol libre:** es un grafo no dirigido acíclico conexo.
- **Árbol de recubrimiento de un grafo no dirigido $G = (V, A)$:** es un árbol libre $T = (V', A')$ tal que $V' = V$ y $A' \subseteq A$.



Grafo Bosque Árbol

Figura 2.7: Ejemplo de árbol, grafo y bosque.

Teorema (Propiedades de los árboles libres): *(Demostración en [CLR90, pág. 91-93].) Sea $G = (V, A)$ un grafo no dirigido. Los siguientes predicados son equivalentes:*

1. *G es un árbol libre.*
2. *Cualquier par de vértices está conectados por un único camino.*
3. *G es conexo, pero si se elimina cualquier arista de A, el grafo resultante no es conexo.*
4. *G es conexo y tiene $|V| - 1$ aristas.*
5. *G es acíclico, y tiene $|V| - 1$ aristas.*
6. *G es acíclico, pero si se añade una arista, se crea un ciclo.*

2.3.1 Árboles enraizados

Árbol enraizado: es un árbol libre con un vértice (o nodo) distinguido denominado **raíz**.

- Si existe un camino de un nodo x a un nodo y en un árbol T, se dice que x es **antecesor** de y, y que y es **sucesor** de x.
- Si (x, y) es el último arco en el camino desde la raíz r del árbol T hasta el nodo y, entonces x es el **padre** de y, e y es el **hijo** de x. La raíz es el único nodo en T que no tiene padre. Si dos nodos tienen el mismo padre son **hermanos**.

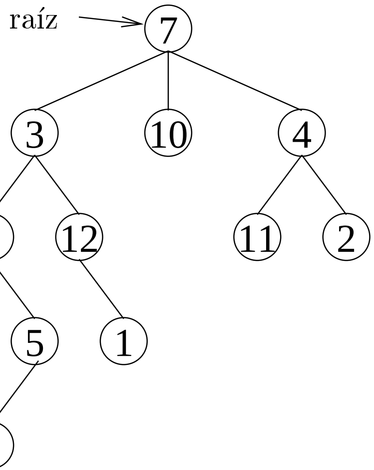


Figura 2.8: Ejemplo de árbol enraizado.

- Un nodo sin hijos lo denominaremos **hoja**. El resto son nodos **internos**.
- El **grado** de un nodo es el número de hijos que tiene.
- Se llama **profundidad** de un nodo a la longitud del camino desde la raíz a ese nodo.
- La **altura** de un árbol es la profundidad del nodo más profundo.

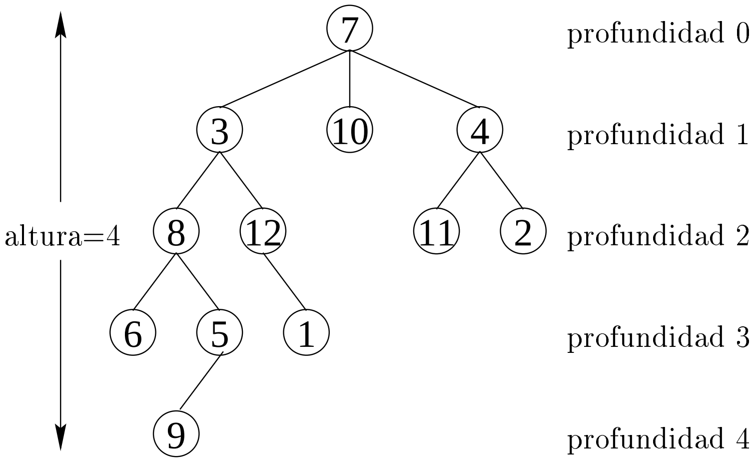


Figura 2.9: Altura de un árbol.

2.4 Árboles binarios

Un **árbol binario** es un árbol en el que el máximo número de hijos de cada nodo es 2 (**hijo izquierdo** e **hijo derecho**). (Nodo x: padre(x); hizquierdo(x); hderecho(x))

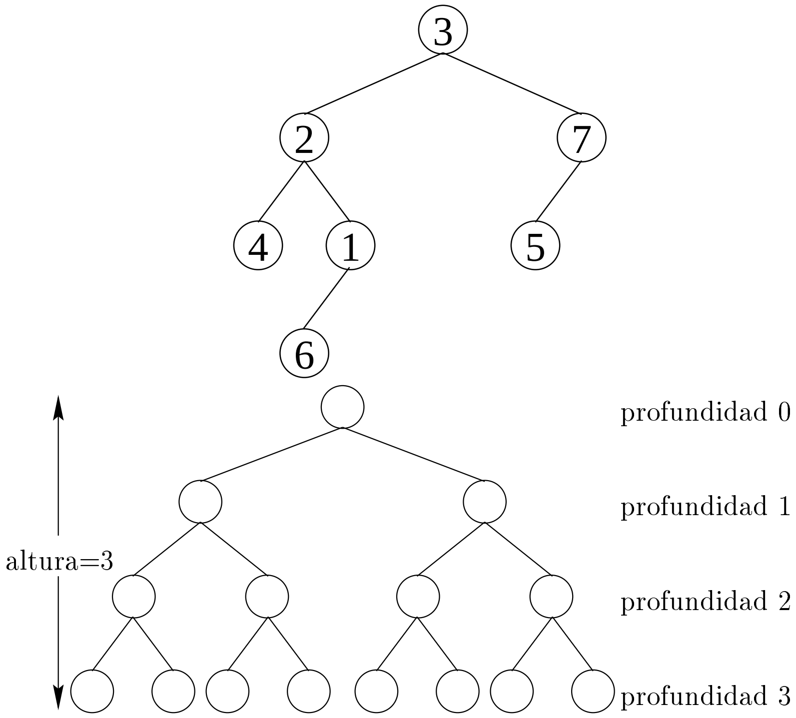


Figura 2.10: Altura de un árbol binario y de un árbol binario completo.

- Un árbol binario se dice que es **completo** (o lleno) si todas las hojas tienen la misma profundidad y todos los nodos internos tienen grado 2.
- Un árbol binario es **casi-completo** si el árbol es completo, a excepción quizás en el nivel de las hojas, en el cual todas las hojas están tan a la izquierda como sea posible.
- Un árbol binario es **equilibrado** si para todo nodo la diferencia de alturas del subárbol izquierdo y del subárbol derecho es como mucho uno.

2.4.1 Propiedades de los árboles binarios completos

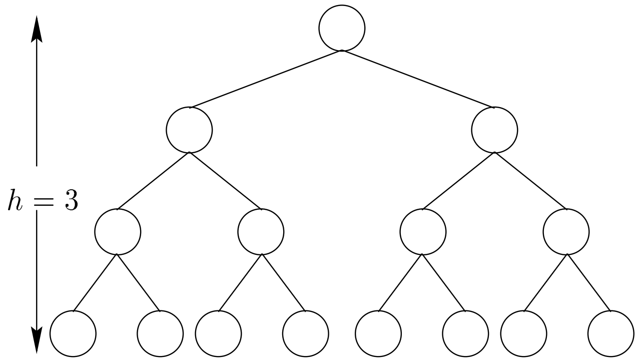
- Número de nodos: La raíz tiene 2 hijos de profundidad 1, cada uno de los cuales tienen 2 hijos de profundidad 2, etc. → Nodos de profundidad $i = 2^i$.

$$\text{hojas: } 2^h \quad \text{y} \quad \text{nodos internos: } 1 + 2^2 + \ldots + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

$$\boxed{\text{nodos total: } n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1}$$

- … y la altura

$$\boxed{n = 2^{h+1} - 1 \qquad \rightarrow \qquad h = \lfloor \log_2 n \rfloor}$$



Prof.	Núm. nodos
0	$1 = 2^0$
1	$2 = 2^1$
2	$4 = 2^2$
3	$8 = 2^3$

Figura 2.11: Árbol binario completo de altura $h = 3$, con 8 hojas y 7 nodos internos.

2.4.2 Propiedades de los árboles binarios

- El máximo número de nodos de profundidad i es 2^i.
- El máximo número de nodos en un árbol binario de altura h es $2^{h+1} - 1$. El máximo número de nodos internos es $2^h - 1$. El máximo número de hojas es 2^h.
- La altura mínima de un árbol binario con n nodos es $\lfloor \log_2 n \rfloor$.
- Para cualquier árbol binario no vacío, si n_0 es el número de hojas y n_2 es el número de nodos de grado 2, entonces $n_0 = n_2 + 1$.
- Para cualquier árbol binario completo, si n_0 es el número de hojas y n_2 es el número de nodos internos, entonces $n_0 = n_2 + 1$.

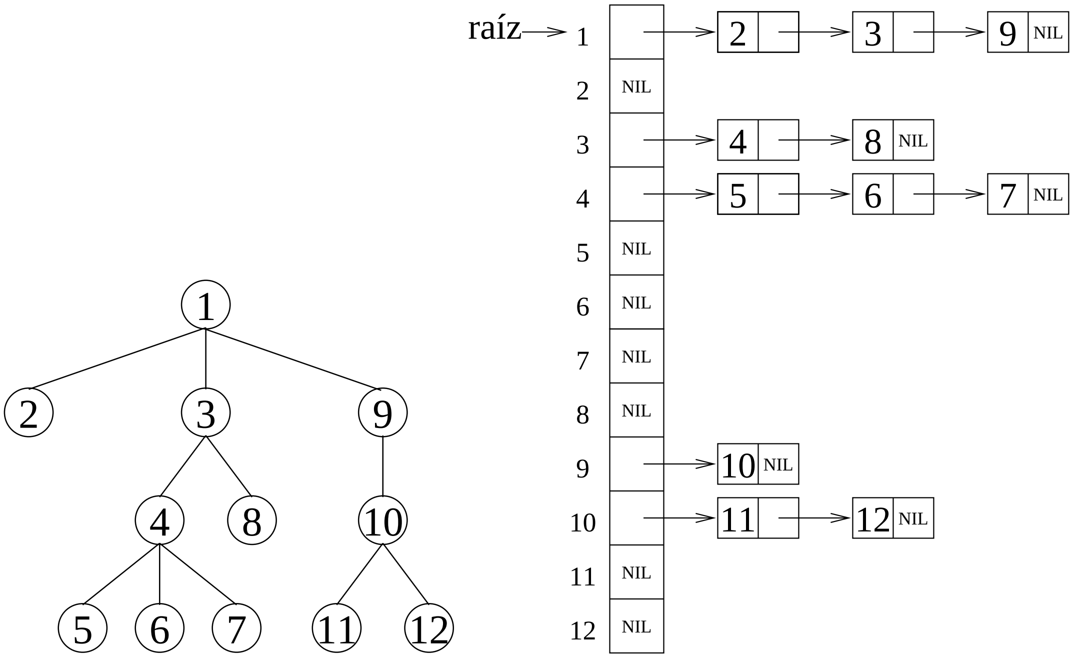
Ejercicio. Árboles k-arios. Un árbol k-ario es un árbol en el que el máximo número de hijos de cada nodo es k (así, un árbol binario es un árbol k-ario con $k = 2$). ¿Cuántas hojas tiene un árbol k-ario completo de altura h? ¿Y cuántos nodos internos?

2.5 Representación de árboles

- Árboles (número de hijos sin acotar)
  - Listas (ordenadas) de hijos
  - Hijo más a la izquierda-Hermano derecha: variables dinámicas o vectores
  - Con vectores y dirección del padre
  - Otros...
- Árboles binarios

- Hijo izquierdo e Hijo derecho para cada nodo: variables dinámicas (“Práctica ABB”) o vectores
- Árbol binario (casi-completo): vector (“heaps”)
- Otros...

Listas (ordenadas) de hijos



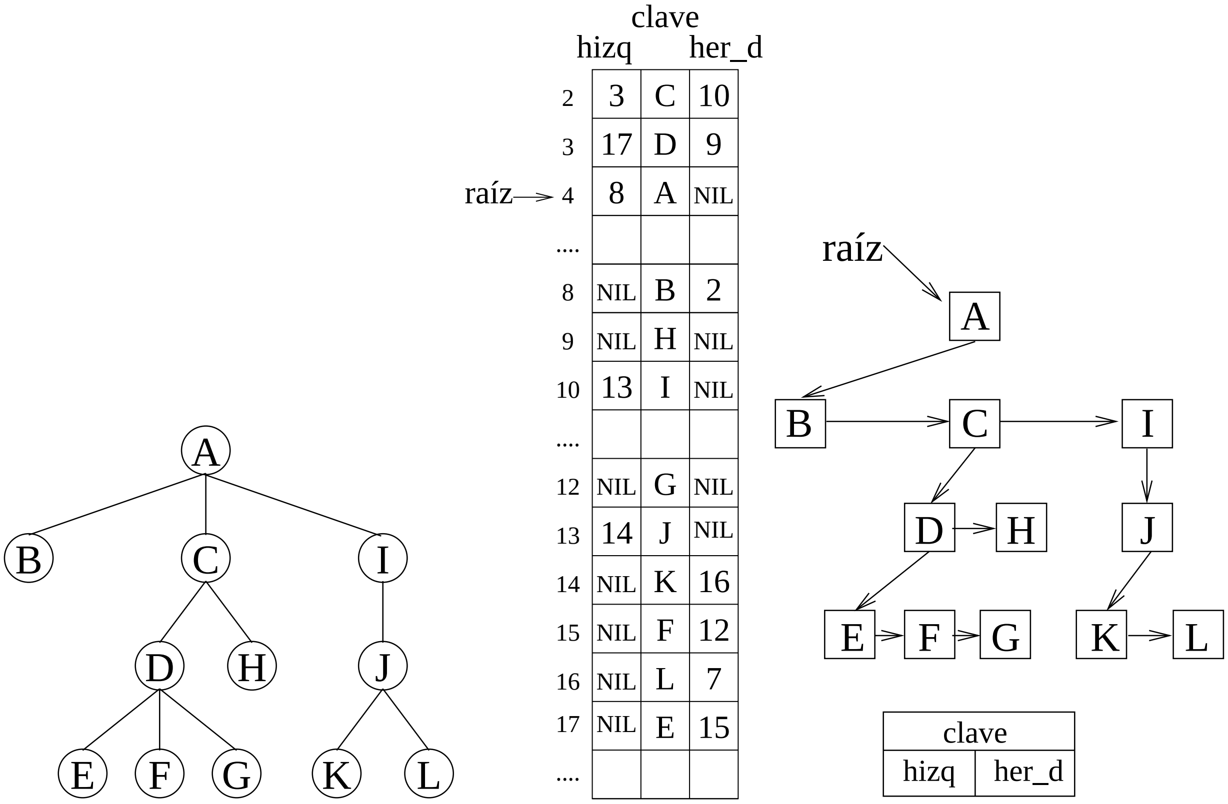
Inconvenientes:

- Se desperdicia memoria: las hojas no tienen hijos.
- Dificultad de la soperaciones que requieran información del padre

Figura 2.12: Representación de árboles (I).

Hijo más a la izquierda-Hermano derecha: variables dinámicas o vectores

(opcionalmente, puntero al padre)

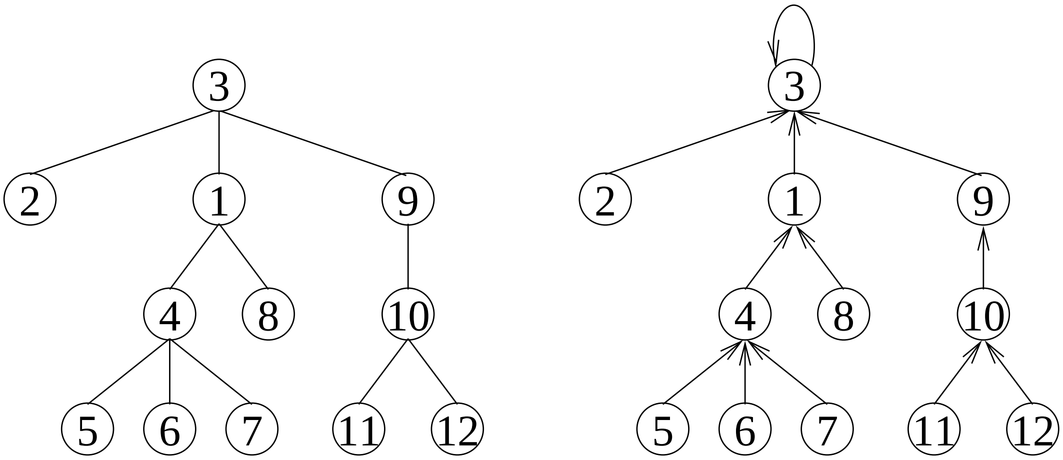


	hizq	clave	her_d
2	3	C	10
3	17	D	9
raíz→ 4	8	A	NIL
....			
8	NIL	B	2
9	NIL	H	NIL
10	13	I	NIL
....			
12	NIL	G	NIL
13	14	J	NIL
14	NIL	K	16
15	NIL	F	12
16	NIL	L	7
17	NIL	E	15
....			

Representación más adecuada cuando hay que realizar actualizaciones sobre el padre, esto es, crear y borrar subárboles, y no se tiene información sobre la talla del árbol.

Figura 2.13: Representación de árboles (II).

Con vectores y dirección del padre



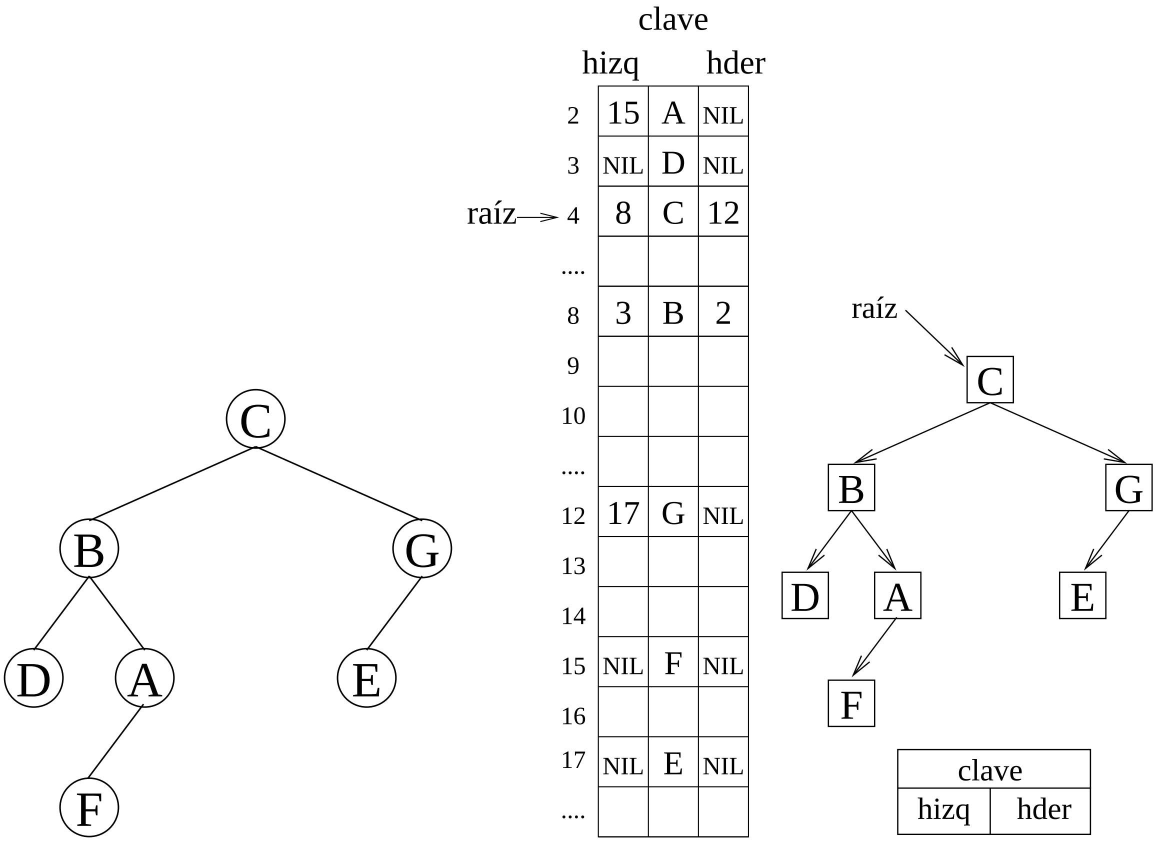
1	2	3	4	5	6	7	8	9	10	11	12
3	3	3	1	4	4	4	1	3	9	10	10

El nodo i es la raíz del árbol si $T[i] = i$. Cada nodo apunta a su padre: $T[i] = \text{padre}(i)$.

Figura 2.14: Representación de árboles (III).

Árboles binarios (hijo izquierdo-hijo derecho): variables dinámicas o vectores

(opcionalmente, puntero al padre)



	hizq	clave	hder
2	15	A	NIL
3	NIL	D	NIL
raíz → 4	8	C	12
....			
8	3	B	2
9			
10			
....			
12	17	G	NIL
13			
14			
15	NIL	F	NIL
16			
17	NIL	E	NIL
....			

```
/* ab.c:Representación árboles binarios con variables dinámicas  */
#include <stdio.h>
#include <stdlib.h>
#define GETMEM(cantidad,tipo) (tipo *)malloc(cantidad*sizeof(tipo))

/* ============================================================ */
typedef struct _ab {           /* Definición tipo árbol binario */
  int  info;
  struct _ab *hizq, *hder;     /* struct _ab ..., *padre        */
} AB;

AB *arbol;                                  /* Variable tipo AB */

/* ============================================================ */
AB *ABCreaVacio() {  return(NULL);  }

int ABEsVacio(AB *AB) {  return(AB == NULL);  }

AB *hizquierdo(AB *AB) {  return(AB->hizq);  }

AB *hderecho(AB *AB) {  return(AB->hder);  }

int key(AB *AB) {  return(AB->info);  }
```

Figura 2.15: Representación de árboles binarios (I).

Árboles binarios (casi-completos): vector

raíz: $T[1]$
padre$(i) = T[\lfloor i/2 \rfloor]$
hizquierdo$(i) = T[2i]$
hderecho$(i) = T[2i+1]$

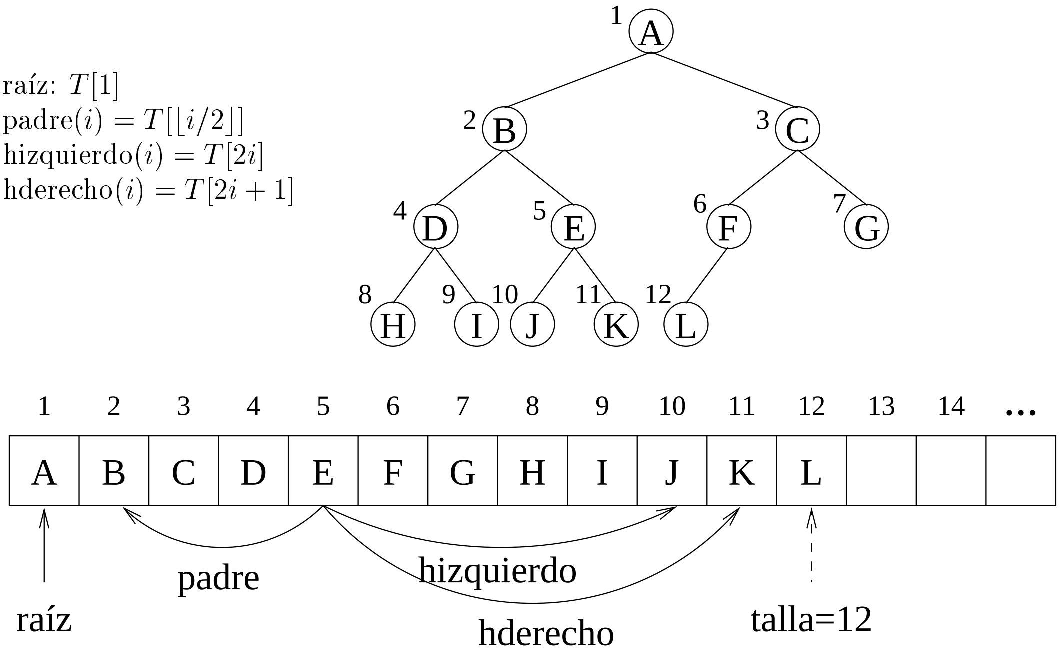


Figura 2.16: Representación de árboles binarios (II).

Ejercicio. Representaciones de árboles. Define, en un lenguaje de programación, las estructuras de datos necesarias para cada una de las representaciones de árboles vistas. Implementa, asimismo, las operaciones necesarias para acceder al padre de un nodo o a sus hijos.

Ejercicio. Árboles k-arios. Modifica la representación de árboles binarios "hijo izquierdo-hijo derecho" para árboles k-arios.

Ejercicio. Altura de un árbol binario. Implementa, para una representación dada de árbol binario, una función que devuelva su altura.

Ejercicio. Árboles binarios. Dado un árbol binario A, con $n \geq 0$ nodos, escribe un algoritmo Divide y Vencerás para determinar en tiempo lineal si el árbol A es completo.

2.6 Recorridos de árboles binarios

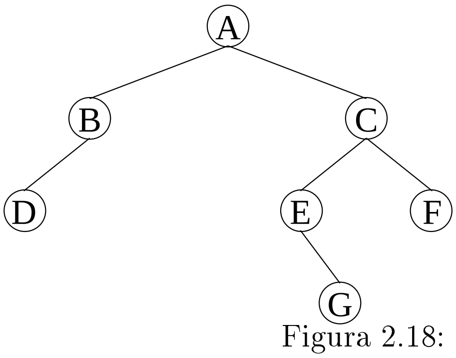
- Preorden: La clave de la raíz se imprime antes de los valores de sus subárboles.
- Inorden: La clave de la raíz se imprime entre los valores de su subárbol izquierdo y los del subárbol derecho.

```
Preorden($x$)
   if $x \neq$ NIL then
      print key($x$)
      Preorden(hizquierdo($x$))
      Preorden(hderecho($x$))
   end if

Inorden($x$)
   if $x \neq$ NIL then
      Inorden(hizquierdo($x$))
      print key($x$)
      Inorden(hderecho($x$))
   end if

Postorden($x$)
   if $x \neq$ NIL then
      Postorden(hizquierdo($x$))
      Postorden(hderecho($x$))
      print key($x$)
   end if
```

Figura 2.17: Algoritmos de recorrido de árboles binarios: Preorden, Inorden y Postorden.

- Postorden: La clave de la raíz se imprime después de los valores de sus subárboles.

El coste de todos los algoritmos es $\Theta(n)$, siendo n el número de nodos del árbol → después de la llamada inicial, la función se llama recursivamente exactamente 2 veces para cada nodo del árbol: una vez para su hijo izquierdo y otra para su hijo derecho.



- Recorrido en preorden: A B D C E G F
- Recorrido en inorden: D B A E G C F
- Recorrido en postorden: D B G E F C A

Figura 2.18: Ejemplo de recorridos de árboles.

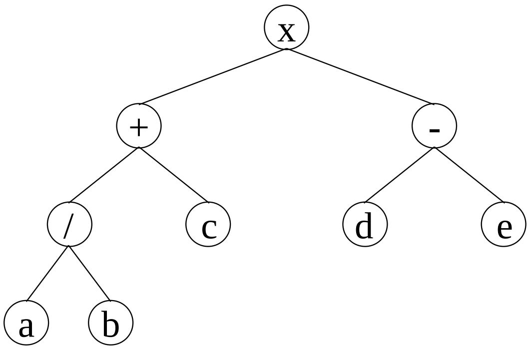
Ejercicio. Implementación. Implementa, en un lenguaje de programación determinado, todos los recorridos vistos para una representación de árbol binario basada en variable dinámica (campos del nodo: clave, puntero hijo izquierdo, puntero hijo derecho).

Ejercicio. Preorden/Inorden iterativo. Escribe un algoritmo iterativo para realizar un recorrido en Preorden y en Inorden. *Pista:* utiliza una pila auxiliar.

2.6.1 Ejemplo árboles binarios: árbol de expresiones

Utilización de la estructura AB para representar expresiones aritméticas con operadores binarios.

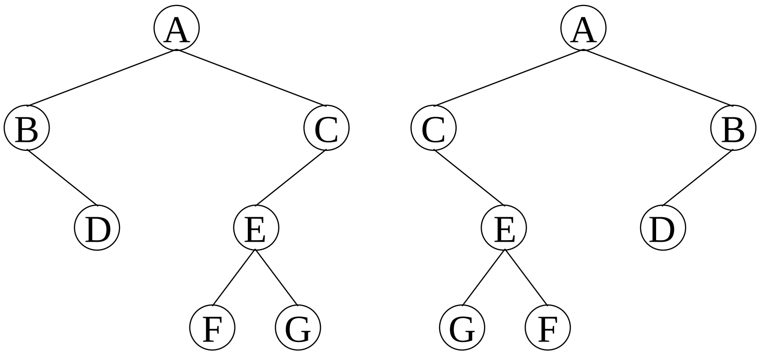
- raíz: operador principal
- nodos internos: operadores de subexpresiones
- hojas: operandos
- (niveles: precedencia relativa de evaluación)



- Recorrido en preorden: $\times + / a\ b\ c - d\ e$ notación prefija
- Recorrido en inorden: $(((a/b)\ +\ c)\ \times\ (d-e))$ notación infija
- Recorrido en postorden: $a\ b\ /\ c\ +\ d\ e\ -\ \times$ notación postfija o polaca

Figura 2.19: Árbol de expresiones.

Ejercicio. Espejo. Decimos que un AB e B es el *espejo* o imagen especular de otro AB A si las raíces de A y B son iguales y el nodo que aparece como hijo izquierdo en A, aparece como hijo derecho (del mismo padre) en B, y viceversa. Ejemplo:



Diseña un algoritmo que obtenga la imagen especular de un AB. Determina el coste.

2.7 Recorridos de grafos

Método para recorrer de forma sistemática y eficiente un grafo.

- Recorrido en profundidad: generalización del recorrido en preorden de un árbol
- Recorrido en amplitud: generalización del recorrido en niveles de un árbol

En todos los algoritmos de recorrido de grafos supondremos que el grafo está implementado con listas de adyacencia.

2.7.1 Recorrido en profundidad de un grafo

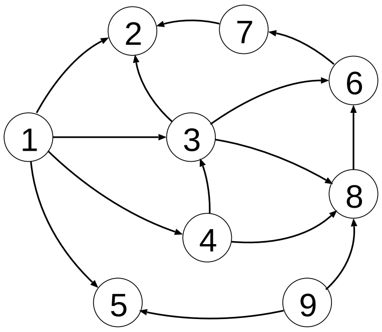
Dado un grafo $G = (V, A)$ y un vértice $v \in V$, la estrategia de recorrido en profundidad ("Depth-First Search (DFS)"), explora sistemáticamente las aristas de G de manera que primero se visitan los vértices adyacentes a los visitados más recientemente. De esta forma, se va "profundizando" en el grafo; es decir, alejándose progresivamente de v.

Esta estrategia admite una implementación simple de forma recursiva, utilizando globalmente un *contador* (n) y un *vector de naturales* (R) para "marcar" los vértices ya visitados ($R[v] > 0$) y almacenar el orden de recorrido.

```
RecorridoEnProfundidad($G$)
  $n := 0$
  for all $v \in V$ do
    $R[v] := 0$
  end for
  for all $v \in V$ do
    if $R[v] = 0$ then
      DFS($v$)
    end if
  end for
  return ($R$)


DFS($v$)
  $n := n + 1$; $R[v] := n$
  for all $w \in$ adyacentes($v$) do
    if $R[w] = 0$ then
      DFS($w$)
    end if
  end for
```

Figura 2.20: Algoritmo de recorrido en profundidad de un grafo. Coste temporal: $O(|V| + |A|)$.

Ejercicio. Coste. Estudia el coste del algoritmo suponiendo que el grafo se representa con una matriz de adyacencia.



nodos	R								
v/w	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0
1/2,3,4,5	1	-	-	-	-	-	-	-	-
2	-	2	-	-	-	-	-	-	-
3/2,6,8	-	-	3	-	-	-	-	-	-
6/7	-	-	-	-	-	4	-	-	-
7/2	-	-	-	-	-	-	5	-	-
8/6	-	-	-	-	-	-	-	6	-
4/3,8	-	-	-	7	-	-	-	-	-
5/–	-	-	-	-	8	-	-	-	-
9/5,8	-	-	-	-	-	-	-	-	9

R	1	2	3	7	8	4	5	6	9

Orden de Visita de Nodos: 1, 2, 3, 6, 7, 8, 4, 5, 9

Figura 2.21: Recorrido en profundidad de un grafo: Ejemplo.

2.7.2 Recorrido en amplitud de un grafo

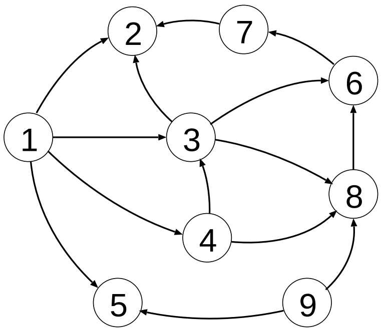
Dado un grafo $G = (V, A)$ y un vértice $v \in V$, la estrategia de recorrido en amplitud o anchura ("Breadth-First Search (BFS)"), explora sistemáticamente las aristas de G de manera que primero se visitan los vértices "más cercanos" a v.

La función RecorridoEnAnchura obtiene un recorrido en anchura de G llamando a la rutina BFS. Se utiliza globalmente un *contador* (n) y un *vector de naturales* (R) para "marcar" los vértices ya visitados ($R[v] > 0$) y almacenar el orden de recorrido. BFS usa una cola auxiliar (Q) para gestionar los vértices no visitados.

```
RecorridoEnAnchura($G$)
   $n := 0$
   for all $v \in V$ do
      $R[v] := 0$
   end for
   for all $v \in V$ do
      if $R[v] = 0$ then
         BFS($v$)
      end if
   end for
   return ($R$)
```

```
BFS($v$)
   $n := n + 1$; $R[v] := n$; encolar($Q, v$)
   while $Q \neq \phi_{\mathbf{Q}}$ do
      $u :=$ cabeza($Q$); descabezar($Q$)
      for all $w \in$ adyacentes($u$) do
         if $R[w] = 0$ then
            $n := n + 1$; $R[w] := n$; encolar($Q, w$)
         end if
      end for
   end while
```

Figura 2.22: Algoritmo de recorrido en amplitud de un grafo. Coste temporal: $O(|V|+|A|)$.



v	u	w	1	2	3	4	5	6	7	8	9	
1			1	0	0	0	0	0	0	0	0	$< 1 >$
	1		-	-	-	-	-	-	-	-	-	$<>$
		2	-	2	-	-	-	-	-	-	-	$< 2 >$
		3	-	-	3	-	-	-	-	-	-	$< 2, 3 >$
		4	-	-	-	4	-	-	-	-	-	$< 2, 3, 4 >$
		5	-	-	-	-	5	-	-	-	-	$< 2, 3, 4, 5 >$
	2		-	-	-	-	-	-	-	-	-	$< 3, 4, 5 >$
	3		-	-	-	-	-	-	-	-	-	$< 4, 5 >$
		2	-	-	-	-	-	-	-	-	-	—
		6	-	-	-	-	-	6	-	-	-	$< 4, 5, 6 >$
		8	-	-	-	-	-	-	-	7	-	$< 4, 5, 6, 8 >$
	4		-	-	-	-	-	-	-	-	-	$< 5, 6, 8 >$
		3	-	-	-	-	-	-	-	-	-	—
		8	-	-	-	-	-	-	-	-	-	—
	5		-	-	-	-	-	-	-	-	-	$< 6, 8 >$
	6		-	-	-	-	-	-	-	-	-	$< 8 >$
		7	-	-	-	-	-	-	8	-	-	$< 8, 7 >$
	8		-	-	-	-	-	-	-	-	-	$< 7 >$
		6	-	-	-	-	-	-	-	-	-	—
	7		-	-	-	-	-	-	-	-	-	$<>$
9			-	-	-	-	-	-	-	-	9	$< 9 >$
	9		-	-	-	-	-	-	-	-	-	$<>$
	9	5	-	-	-	-	-	-	-	-	-	—
	9	8	-	-	-	-	-	-	-	-	-	—

	1	2	3	4	5	6	8	7	9	

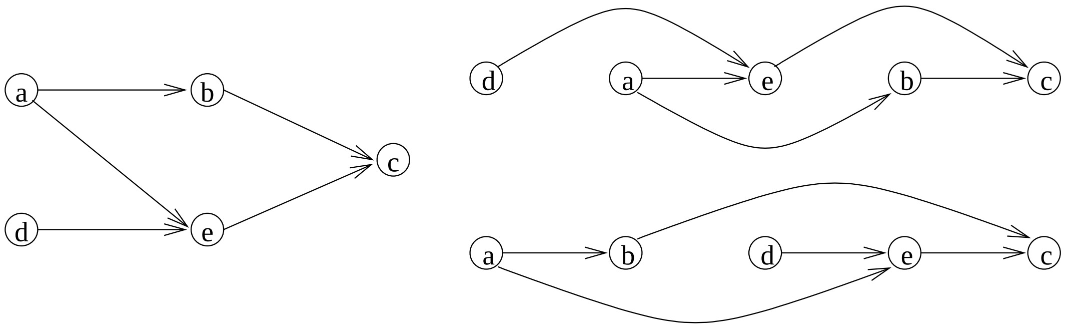
Orden de Visita de Nodos: 1, 2, 3, 4, 5, 6, 8, 7, 9

Figura 2.23: Recorrido en amplitud de un grafo: Ejemplo.

2.8 Orden topológico en grafos acíclicos

Un **orden topológico** de un grafo dirigido acíclico $G = (V, A)$ es una ordenación de los vértices de forma que si $(u, v) \in A$, entonces u aparece *antes* que v. (La solución no es única.) Ejem.: prerrequisitos de los estudios.

¡No es posible la ordenación toplógica cuando existen ciclos!



Orden no único.
Ordenación de vértices en eje horizontal con las aristas de izquierda a derecha.

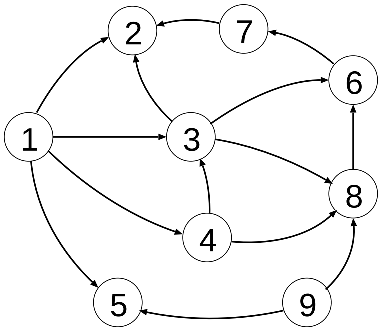
Figura 2.24: Ejemplo de ordenamiento topológico.

Dado un grafo acíclico $G = (V, A)$, el *recorrido en profundidad* puede usarse directamente para ordenar los vértices de V según un orden (parcial) "$\prec$" tal que, $\forall u, v \in V$, si $(u, v) \in A$ entonces $u \prec v$. Basta con ir imprimiendo los *vértices completamente explorados* por DFS.

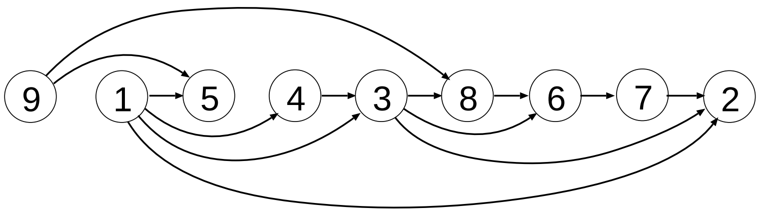
Ejercicio. Coste. Estudia el coste del algoritmo suponiendo que el grafo se representa con una matriz de adyacencia.

```
OrdenTopológico($G$)
  $n := 0$
  for all $v \in V$ do
    $R[v] = 0$
  end for
  for all $v \in V$ do
    if $R[v] = 0$ then
      DFS($v$)
    end if
  end for

DFS($v$)
  $n := n + 1$; $R[v] = n$
  for all $w \in$ adyacentes($v$) do
    if $R[w] = 0$ then
      DFS($w$)
    end if
  end for
  print $v$
```

Figura 2.25: Algoritmo de orden topológico en grafos acíclicos. Coste temporal: $O(|V| + |A|)$.



nodos	R									
v/w	1	2	3	4	5	6	7	8	9	P
——	0	0	0	0	0	0	0	0	0	$<>$
1/2,3,4,5	1	-	-	-	-	-	-	-	-	$<>$
2	-	2	-	-	-	-	-	-	-	$< 2 >$
3/2,6,8	-	-	3	-	-	-	-	-	-	$< 2 >$
6/7	-	-	-	-	-	4	-	-	-	$< 2 >$
7/2	-	-	-	-	-	-	5	-	-	$< 7, 2 >$
——	-	-	-	-	-	-	-	-	-	$< 6, 7, 2 >$
8/6	-	-	-	-	-	-	-	6	-	$< 8, 6, 7, 2 >$
——	-	-	-	-	-	-	-	-	-	$< 3, 8, 6, 7, 2 >$
4/3,8	-	-	-	7	-	-	-	-	-	$< 4, 3, 8, 6, 7, 2 >$
5/–	-	-	-	-	8	-	-	-	-	$< 5, 4, 3, 8, 6, 7, 2 >$
——	-	-	-	-	-	-	-	-	-	$< 1, 5, 4, 3, 8, 6, 7, 2 >$
9/5,8	-	-	-	-	-	-	-	-	9	$< 9, 1, 5, 4, 3, 8, 6, 7, 2 >$



Grafo ordenado topológicamente

Figura 2.26: Orden topológico en grafos acíclicos: Ejemplo.

Capítulo 3

Conjuntos

Objetivos

- Estudiar los conjuntos desde un punto de vista algorítmico (conjuntos dinámicos).
- Estudiar técnicas de representación de conjuntos (algoritmos de manipulación).
- Estudiar los tipos de datos: diccionario, cola de prioridad.
- Aprender a seleccionar el tipo de datos y la implementación más adecuados al problema a resolver.

Contenidos

1 Conceptos generales. Operaciones con conjuntos
2 Representaciones básicas
3 Tablas de dispersión
4 Árboles binarios de búsqueda
5 Árboles parcialmente ordenados: Montículos (Heaps)
6 Conjuntos disjuntos: MFset

Bibliografía

- *Introduction to Algorithms*, de Cormen, Leiserson y Rivest (capítulos 7, 11.1, 11.2, 12, 13 y 22)
- *Fundamentos de Algoritmia*, de Brassard y Bratley (capítulo 5)
- *Estructuras de datos y algoritmos*, de Aho, Hopcroft y Ullman (capítulos 4 y 5)

3.1 Conceptos generales

- Colección de elementos, donde cada elemento es un ítem o un conjunto. Todos los elementos de un conjunto son diferentes.
- Elementos de un conjunto (dinámico). Típicamente:
  - clave: campo de identificación
  - información satélite: asociada al objeto
- Algunos conjuntos se caracterizan porque en el conjunto de claves se puede establecer una relación de orden total (ejem.: enteros, reales, palabras, …)
  - En este caso se puede definir el elemento mínimo (o máximo) de un conjunto, o el sucesor (o predecesor).

3.1.1 Operaciones con conjuntos

- **Consultoras (“queries”):** Buscar (Search), Mínimo (Minimum), Máximo (Maximum), Predecesor (Predecessor), Sucesor (Successor)
- **Modificadoras:** Insertar (Insert), Borrar (Delete)

Tabla 3.1: Operaciones con conjuntos. x es la dirección del objeto de clave key(x) y S es un conjunto dinámico. El coste de las operaciones viene dado en función del tamaño del conjunto $|S| = n$.

Search(S, k)	dado un conjunto S y una clave k devuelve la posición x tal que key(x) $= k$, o un valor especial si $x \notin S$
Insert(S, x)	añade el elemento apuntado por x en el conjunto S (x ya inicializado)
Delete(S, x)	borra el elemento apuntado por x de S. ¡¡¡Delete(S, k)!!!
Minimum(S)	devuelve el elemento de S cuya clave sea mínima
Maximum(S)	devuelve el elemento de S cuya clave sea máxima
DeleteMin(S)	borra el elemento de S cuya clave sea mínima
DeleteMax(S)	borra el elemento de S cuya clave sea máxima
Predecessor(S, x)	devuelve el elemento de clave inmediatamente inferior a la de x o un valor especial si x es el mínimo
Successor(S, x)	devuelve el elemento de clave inmediatamente superior a la de x o un valor especial si x es el máximo

3.1.2 Algunos tipos de conjuntos dinámicos

- **Diccionarios**
  - Operaciones: Search, Insert, Delete
  - Ejemplo: tabla de símbolos
  - Tablas de dispersión, ABB
- **Colas de prioridad**
  - Operaciones: Insert, DeleteMax (DeleteMin)
  - Ejemplo: cola de procesos
  - Heaps, ABB

  Aplicaciones de colas de prioridad:

  - Simulaciones (las claves corresponden a eventos temporales, que se deben procesar en orden cronológico)
  - Job scheduling (las claves corresponden a la prioridad de los procesos que indican el proceso que se debe procesar a continuación)
  - Computación numérica (las claves son errores computacionales, que indican el error mayor que se debe resolver en un momento dado)

  En realidad, las colas de prioridad es una generalización de pilas y colas.

- **Conjuntos disjuntos** (conjuntos con número de elementos fijo)
  - Operaciones: crear, unión, búsqueda del conjunto
  - Ejemplo: componentes conexas
  - MF-Set

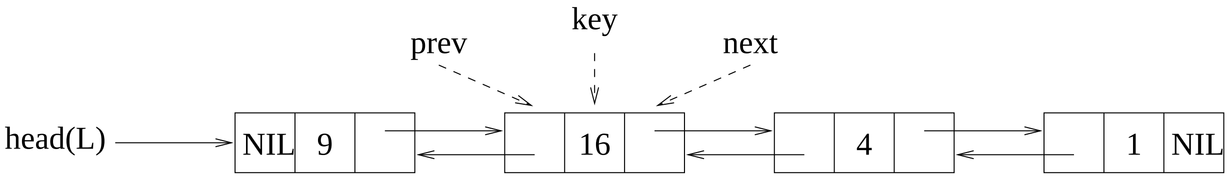
3.2 Representaciones básicas

- Pilas y Colas: Se inserta y se borra un elemento predeterminado
  - Pila: política LIFO (“Pila platos”)
    - Implementación: vector, punteros
    - Operaciones: push, top. Coste $O(1)$
  - Cola: política FIFO (“Cola banco”)
    - Implementación: vector circular, punteros
    - Operaciones: enqueue, dequeue. Coste $O(1)$

Ejercicio. Pilas y colas. Define las estructuras de datos para implementar pilas y colas, tanto con vectores como con memoria dinámica, y diseña las operaciones.

- Listas enlazadas: representación simple y flexible para conjuntos dinámicos. "Soportan" todas las operaciones de conjuntos (aunque de forma no necesariamente eficiente)
  - simplemente enlazadas, doblemente enlazadas, ordenadas, circulares, ...

3.2.1 Ejemplo: Lista doblemente enlazada



- Una lista L tiene el atributo "head" para acceder al principio de la misma (head(L))
- Si x es un elemento de la lista:
  - key(x) es la clave
  - next(x) apunta a su sucesor
  - prev(x) apunta a su predecesor
- Si prev(x) = NIL, x no tiene predecesor, es el primer elemento de la lista
- Si next(x) = NIL, x no tiene sucesor, es el último elemento de la lista

```
List-Search($L, k$) {Encuentra el primer elemento con clave $k$ en $L$ y devuelve su posición. Si no lo encuentra devuelve NIL}
    $x :=$ head($L$)
    while $x \neq$ NIL $\wedge$ key($x$) $\neq k$ do
        $x :=$ next($x$)
    end while
    return $x$
Coste: $O(n)$
```

```
List-Insert($L, x$) {Dado la dirección de un elemento $x$, lo inserta en la cabeza de la lista $L$}
    next($x$) := head($L$)
    if head($L$) $\neq$ NIL then
        prev(head($L$)) := $x$
    end if
    head($L$) := $x$
```

```
    prev($x$) := NIL
Coste: $O(1)$
```

```
List-Delete($L, x$) {Dado la dirección de un elemento $x$, lo borra de la lista $L$}
    if prev($x$) $\neq$ NIL then
        next(prev($x$)) := next($x$)
    else
        head($L$) := next($x$)
    end if
    if next($x$) $\neq$ NIL then
        prev(next($x$)) := prev($x$)
    end if
Coste: $O(1)$. ¡OJO! List-Delete($L, k$): primero buscar, coste $O(n)$
```

Tabla 3.2: Estudio comparativo de los costes de las operaciones con conjuntos utilizando listas doblemente enlazadas. S conjunto, $n = |S|$, x puntero a objeto, k clave del objeto.

Operación	**Lista**	**Lista ordenada**
List-Search(S, k)	$O(n)$	$O(n)$
List-Insert(S, x)	$O(1)$	$O(n)$
List-Delete(S, x)	$O(1)$	$O(1)$
List-Minimum(S), List-Maximum(S)	$O(n)$	$O(1)$
List-Successor(S, x), List-Predecessor(S, x)	$O(n)$	$O(1)$

Ejercicio. Listas. Costes operaciones con listas simplemente enlazadas, ordenadas y no ordenadas.

Ejercicio. Implementa. Define e implementa las operaciones con listas sin utilizar punteros. Define una estrategia para gestionar el espacio libre.

3.3 Tablas de dispersión

- Estructura de datos especialmente diseñada para la implementación de DICCIONARIOS (operaciones Search, Insert, Delete en un tiempo esperado $O(1)$).
- Contenidos:
  - Diccionarios
  - Tablas de acceso directo (VECTORES)
  - Tablas de dispersión (GENERALIZACIÓN VECTORES)
    - Concepto de dispersión ("hashing")
    - Dispersión por encadenamiento: funciones de dispersión, operaciones, coste
    - Dispersión por direccionamiento abierto

3.3.1 Tablas de direccionamiento directo

- Si universo de claves pequeño $U = \{0, 1, .., m-1\} \rightarrow$ utilizar 1 posición en la tabla por cada posible elemento del conjunto
- El conjunto se representa con un vector T: $T[k]$ es el puntero al elemento x de clave k (o NIL). Se utiliza la clave del elemento para indexarlo en el vector.

Tabla 3.3: Operaciones con tablas de direccionamiento directo: buscar, insertar y borrar. Todas las operaciones tienen coste $O(1)$.

```
Direct-Adress-Search($T, k$)
    return $T[k]$

Direct-Adress-Insert($T, x$)
    return $T[\text{key}(x)] := x$

Direct-Adress-Delete($T, x$)
    return $T[\text{key}(x)] :=$ NIL
```

Ejercicio. Vector de bits. Un vector de n bits ocupa menos espacio que un vector de n punteros. ¿Cómo se utilizaría un vector de bits para representar un conjunto dinámico (con un universo de claves pequeño) donde los elementos *no* tienen asociada información satélite?

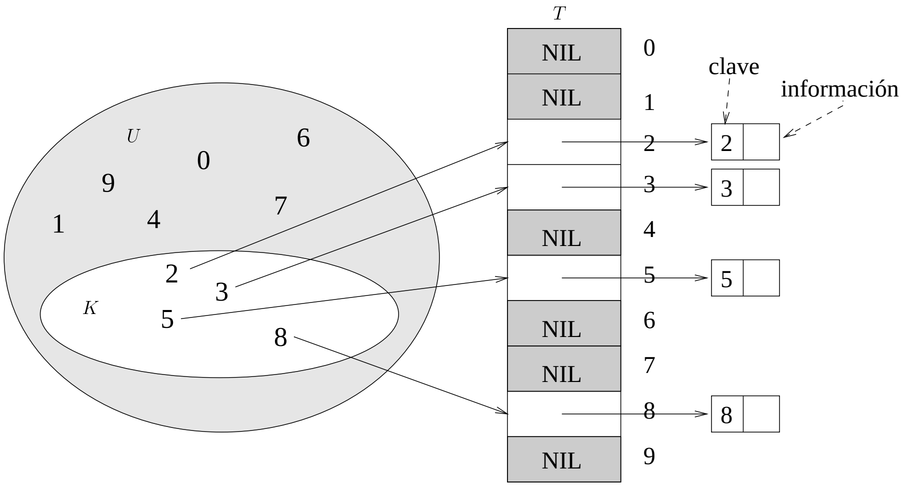


Figura 3.1: Implementación de un conjunto dinámico con una tabla de direccionamiento directo.

3.3.2 Tablas de Dispersión

¿Qué ocurre si U es grande y/o el número de elementos $|S| \ll |U|$? *HASHING*

Se utiliza un vector $T[0..m-1]$, con $m \ll |U|$. Un elemento de clave k se almacena en la posición $h(k)$ de la tabla T. T es una tabla de dispersión (tabla "hash") y h es la función de dispersión ($h : U \rightarrow [0..m-1]$). A cada una de las $T[j]$ se les llama cubeta (bucket, slot).

Se utiliza una función de dispersión h para direccionar las claves en las cubetas de la tabla de dispersión. Se pueden generar **colisiones**.

Resolución de colisiones

- Definir una función de dispersión que minimice el número de colisiones (Dispersión efectiva).
- Dos métodos para tratar las colisiones:
  - Encadenamiento
  - Direccionamiento abierto

```
typedef struct snode {
   char *palabra;
   struct snode *sig;
} tnode;
typedef tnode *ttabla;
```

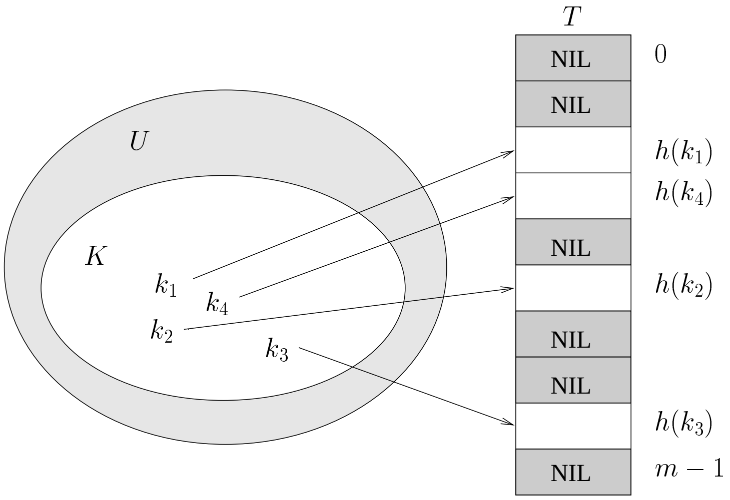


Figura 3.2: Uso de una función de dispersión para direccionar claves en las cubetas de una tabla.

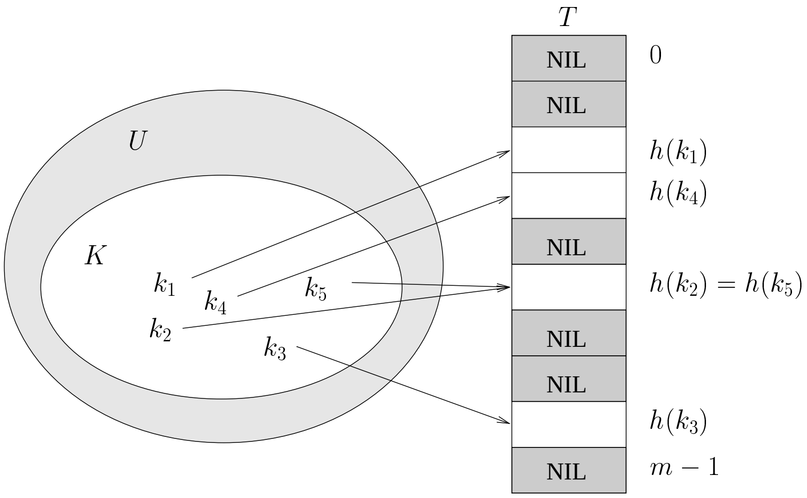
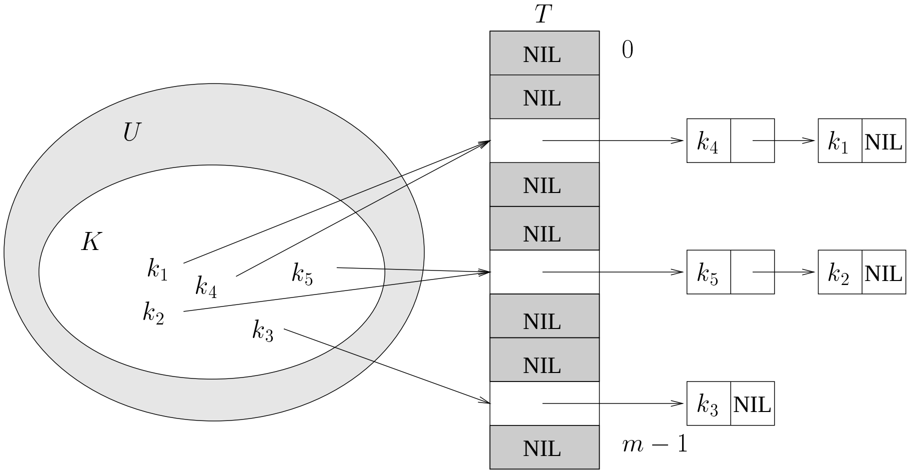


Figura 3.3: Ejemplo de colisiones. Las claves k_2 y k_5 se dispersan a la misma cubeta, $h(k_2) = h(k_5)$, por lo que se produce una *colisión*.

Resolución de colisiones por encadenamiento

Se encadenan los elementos que tienen el mismo valor de la función de dispersión.
$T[j]$: puntero a la cabeza de lista de aquellos elementos cuya función hash es j.



$$S = \{84, 7, 22, 14, 34, 6, 64, 9\}; \text{ tabla con 10 cubetas; } h(k) = k \bmod 10$$
$$h(84) = 4, h(7) = 7, h(22) = 2, h(14) = 4, h(34) = 4, h(6) = 6, h(64) = 4, h(9) = 9$$

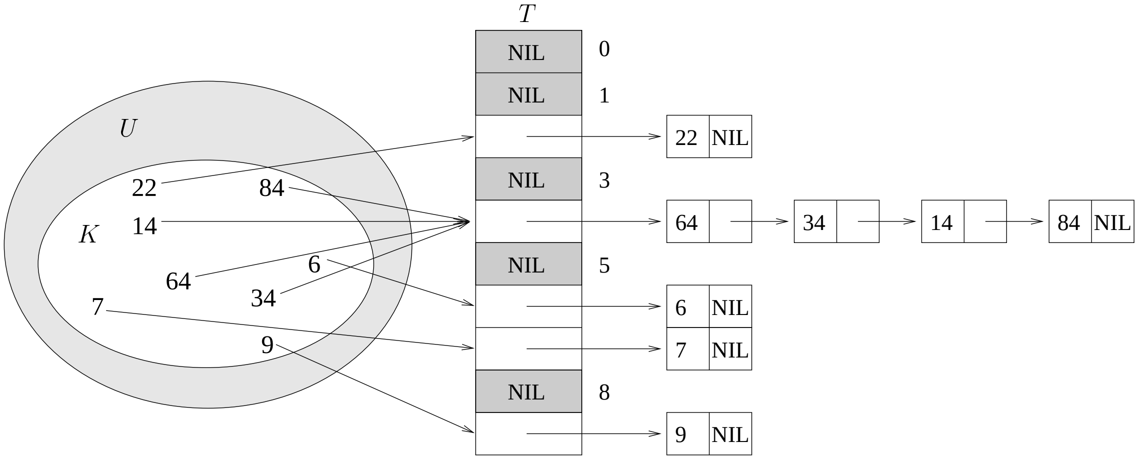


Figura 3.4: Ejemplo resolución de colisiones por encadenamiento.

Operaciones: buscar, insertar y borrar.

Tabla 3.4: Operaciones con tablas de dispersión.

```
Chained-Hash-Insert($T, x$)
    Insert $x$ en la cabeza de la lista $T[h(\mathrm{key}(x))]$

Chained-Hash-Search($T, k$)
    Search el elemento con clave $k$ en la lista $T[h(k)]$

Chained-Hash-Delete($T, x$)
    Delete $x$ de la lista $T[h(\mathrm{key}(x))]$
```

- Insertar: $O(1)$
- Buscar: $O(n)$ en el peor caso (todos los elementos de están en la misma cubeta). Tiempo esperado: $O(1)$
- Borrar: $O(1)$ (con listas doblemente enlazadas). Con listas simples: coste igual a buscar. Si no tenemos la dirección (sólo la clave) hay que buscar previamente el elemento: el coste es el de Buscar.

Análisis del coste medio de Buscar

- Dada un tabla de m cubetas que almacena n elementos, definimos el **factor de carga** de la tabla como $\alpha = n/m$ (número medio de elementos almacenados en una lista)
- Peor caso de Chained-Hash-Search: $O(n)$ (todos los elementos de están en la misma cubeta)
- Análisis del coste medio:
  Asumimos "hashing uniforme simple": La probabilidad de que un elemento de U se almacene en cualquier posición de la tabla es la misma.
  Asumimos que el coste de calcular h es $O(1)$
  Suponemos que las colisiones se han resuelto por encadenamiento

  $$\boxed{\text{El coste de Chained-Hash-Search es } O(1+\alpha)}$$

  - Si n es proporcional a m, $n \in \Theta(m)$, $\alpha = n/m = O(m)/m \in \Theta(1)$
  - Y por lo tanto el coste de Chained-Hash-Search es $O(1)$
  - Así, todas las operaciones del tipo **diccionario** (buscar, insertar y borrar) se pueden efectuar con **coste constante** $O(1)$

La función de dispersión h

"Hashing uniforme simple": La probabilidad de que un elemento de U se almacene en cualquier posición de la tabla es la misma.

$$\sum_{k|h(k)=j} P(k) = \frac{1}{m}, \qquad \text{para } j = 0, 1, \ldots, m-1$$

- **Método de la División:** $\boxed{h(k) = k \bmod m}$
  Restricción: m número primo no cercano a una potencia de 2.

  Ejemplo: se desea almacenar $n \approx 2000$ elementos. Podemos examinar una media de 3 elementos en una búsqueda

  - Seleccionamos tamaño tabla: $\alpha = 3 = 2000/m$, $m = 666.66.. \rightarrow m = 701$ (primo cercano a 666.66.. pero no cercano a una potencia de 2)
  - Función $h(k) = k \bmod 701$
  - Ejem.: $h(1600) = 1600 \bmod 701 = 198$

- **Método de la Multiplicación:** $\boxed{h(k) = \lfloor m(kA \bmod 1) \rfloor, \quad 0 < A < 1}$
  Selección m no crítica (m se suele tomar una potencia de 2, facilidad operaciones)
  $A \approx (\sqrt{5} - 1)/2 = 0.6180339887...$

  Ejemplo: se desea almacenar $n \approx 3000$ elementos. Podemos examinar una media de 3 elementos en una búsqueda

  - Seleccionamos tamaño tabla: $\alpha = 3 = 3000/m$, $m = 1000 \rightarrow m = 1024$ (potencia de 2)
  - Función $h(k) = \lfloor 1024(k \cdot 0.6180339887... \bmod 1) \rfloor$
  - Ejem.:

$$\begin{aligned} h(1600) &= \lfloor 1024(1600 \cdot 0.6180339887... \bmod 1) \rfloor \\ &= \lfloor 1024(988.84800... \bmod 1) \rfloor \\ &= \lfloor 1024 \cdot 0.84800 \rfloor \\ &= \lfloor 868.352 \rfloor = 868 \end{aligned}$$

Ejemplo: Funciones hash sobre cadenas

```
$f1(s)$
  $x := 0$
  for $i := 1$ to $|s|$ do $x := x + \mathrm{ord}(s[i])$
```

```
  return $x$ mod $m$
```

- No distribuye bien las claves si el tamaño de T es grande. Ejemplo: $m = 10007$ y $|s| < 8$; $0 \leq f1(s) \leq 1016 = 127 \cdot 8$
- $f1(abc) = f1(bca)$

```
$f2(s)$
  return ord($s$[1]) + 255 ord($s$[2]) + 2552 ord($s$[3])) mod $m$
```

- Si $m = 10007$, sólo se aprovecha un 28% de la tabla. De las $26 \cdot 26 \cdot 26 = 17576$ combinaciones posibles sólo se observan 2851
- $f2(clase) = f2(claudicar)$

```
$f3(s)$
  $x$ := ord($s$[1])
  for $i := 2$ to $|s|$ do
    $x := (x \cdot 256) + ord(s[i]))$ mod $m$
  end for
  return $x$
```

- Cálculo más costoso (bucle con mod)

```
$f4(s)$
  $x$ := ord($s$[1])
  return ord$s$[1] mod $m$
```

- ¡¡Terrible!!

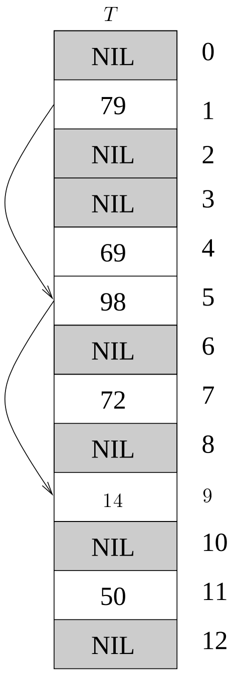
Resolución de colisiones por direccionamiento abierto

Ejemplo “double-hashing”: $\boxed{h(k,i) = (h_1(k) + ih_2(k)) \bmod m}$

Ejercicio. Operaciones. Escribe las funciones Hash-Insert(T,x) y Hash-Search(T,k) para este tipo de tablas de dispersión.

3.3.3 Ejercicios

Ejercicio. Tablas de dispersión. ¿Qué supondría en una tabla de dispersión de talla $m = 1001$ tener una función de dispersión para la cual la variable *talla_de_las_cubetas* tuviera una media $k = 100$ y una varianza 0?



Ejemplo:
$m = 13, \quad h_1(k) = k \bmod 13, \quad h_2(k) = 1 + (k \bmod 11)$
Insertar elemento de clave **14**:

- $h_1(14) = 14 \bmod 13 = 1$; $h_2(14) = 1 + (14 \bmod 11) = 4$
- $i = 0 \rightarrow h(14, 0) = 1 \bmod 13 = 1$
- $i = 1 \rightarrow h(14, 1) = (1 + 4) \bmod 13 = 5$
- $i = 2 \rightarrow h(14, 2) = (1 + 2 \cdot 4) \bmod 13 = \mathbf{9}$

Figura 3.5: Ejemplo direccionamiento abierto.

Ejercicio. Tablas de dispersión. Dos ficheros $F1$ y $F2$, ambos con el mismo número de palabras, se dispersan en sendas tablas de dispersión, ambas de la misma dimensión, y usando la misma función de dispersión. Se mide la ocupación de las cubetas en ambos casos, y se obtiene:

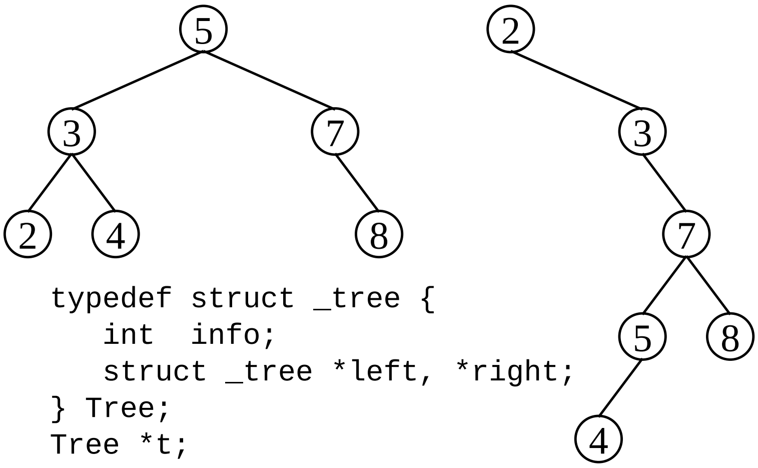
	$F1$	$F2$
Media	4.25	4.25
Desviación típica	0.84	16.73

¿Para cuál de los dos ficheros es más adecuada la función de dispersión usada? Justificad la respuesta.

3.4 Árboles binarios de búsqueda

Un **árbol binario es de búsqueda** si:

- Todos los elementos del subárbol izquierdo son menores que el de la raíz
- Todos los elementos del subárbol derecho son mayores que el de la raíz



```
typedef struct _tree {
   int  info;
   struct _tree *left, *right;
} Tree;
Tree *t;
```

Figura 3.6: Ejemplo de árboles binarios de búsqueda: Representación del conjunto $S = \{5,$ 2, 7, 4, 8, 3$\}$.

```
Inorden($x$)
   if $x \neq$ NIL then
      Inorden(hizquierdo($x$))
      print key($x$)
      Inorden(hderecho($x$))
   end if
```

Propiedad: Si se listan los elementos según recorrido en inorden resulta una secuencia ordenada (demo: por inducción)

2, 3, 4, 5, 7, 8 ← ¡Secuencia ordenada!

Ejercicio. ABB. Dibujar árboles binarios de búsqueda de alturas 2, 3, 4, 5 y 6 para el conjunto de claves $\{1, 4, 5, 10, 16, 17, 21\}$.

Ejercicio. ABB y Heaps. ¿Cuál es la diferencia entre la propiedad de los ABB y la de Heaps? ¿Puede utilizarse la propiedad de Heaps para listar de forma ordenada las claves de un árbol con n nodos en tiempo $O(n)$?

Tabla 3.5: Operaciones con ABBs (diccionarios y colas de prioridad). T es un ABB y x es la dirección del objeto de clave key(x). El coste de todas las operaciones es $O(h)$, siendo h la altura del ABB. Árbol binario con n nodos: completo $O(\log n)$; lista: $O(n)$.

buscar	Tree-Search(T, k)	busca el elemento de clave k en el ABB T
insertar	Tree-Insert(T, x)	añade el elemento apuntado por x en una posición adecuada en un ABB T (x ya inicializado)
borrar	Tree-Delete(T, x)	borra el elemento apuntado por x del ABB T
mínimo	Tree-Minimum(T)	devuelve el elemento del ABB T cuya clave sea mínima
máximo	Tree-Maximum(T)	devuelve el elemento del ABB T cuya clave sea máxima
predecesor	Tree-Predecessor(x)	devuelve el elemento de clave inmediatamente inferior a la de x o un valor especial si x es el mínimo
sucesor	Tree-Successor(x)	devuelve el elemento de clave inmediatamente superior a la de x o un valor especial si x es el máximo

3.4.1 Búsqueda de una clave en un ABB

```
Tree-Search($x, k$) {Busca el elemento de clave k en el ABB x}
  while $x \neq$ NIL $\wedge$ $k \neq$ key($x$) do
    if $k <$ key($x$) then
      $x :=$ hizquierdo($x$)
    else
      $x :=$ hderecho($x$)
    end if
  end while
  return $x$
```

Figura 3.7: ABB: Tree-Search. Llamada inicial: Tree-Search(raíz(T), k). Coste $O(h)$, siendo h la altura del árbol.

Ejercicio. ABB secuencias. Supongamos que tenemos los números del 1 al 1000 en un ABB y queremos buscar el número 363. ¿Cuál de las siguientes secuencias de nodos no puede ser la secuencia de nodos examinada?

2, 252, 401, 398, 330, 344, 397, 363
924, 220, 911, 244, 898, 258, 362, 363
925, 202, 911, 240, 912, 245, 363
2, 399, 387, 219, 266, 382, 381, 278, 363
935, 278, 347, 621, 299, 392, 358, 363

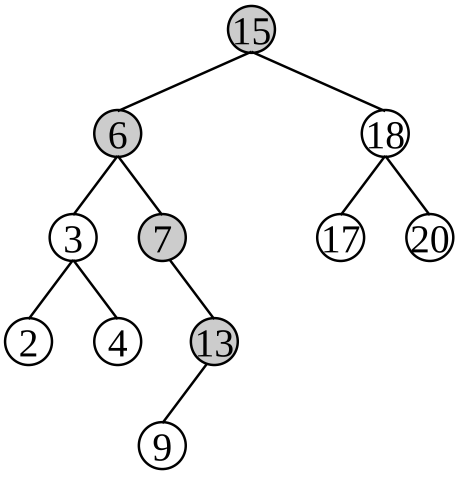
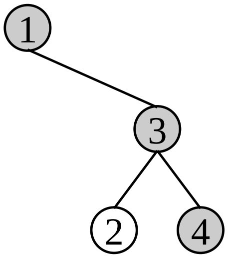


Figura 3.8: Ejemplo Tree-Search.

```
Tree-Search_R($x, k$) {$x$ es un puntero a la raíz de un ABB y $k$ es una clave}
  {Devuelve un puntero al nodo con clave $k$ si existe o NILsi no existe}
  if $x = \text{NIL} \vee k = \text{key}(x)$ then
    return $x$
  end if
  if $k < \text{key}(x)$ then
    return Tree-Search_R(hizquierdo($x$), $k$)
  else
    return Tree-Search_R(hderecho($x$), $k$)
  end if
```

Figura 3.9: Versión recursiva de Tree-Search.

Ejercicio. Falsa propiedad. Demuestra que la siguiente propiedad no es cierta: Supongamos que la búsqueda de un elemento de clave k en un ABB termina en una hoja. Considera tres conjuntos: A, las claves a la izquierda del camino de búsqueda; B, las claves del camino de búsqueda; y C, las claves a la derecha del camino de búsqueda. Encuentra un contraejemplo para demostrar que no se cumple que para cualquier $a \in A$, $b \in B$ y $c \in C$, $a \leq b \leq c$.



$A = \{2\}$; $B = \{1, 3, 4\}$; $C = \{\} \Rightarrow 2 > 1$

3.4.2 Mínimo y máximo en un ABB

- Elemento mínimo en un ABB: nodo que no tiene ningún hijo a la izquierda y que no pertenece a ningún subárbol derecho de ningún nodo

```
Tree-Minimum($x$) {Devuelve un puntero al nodo con la clave más pequeña del árbol $x$, $x \neq$ NIL}
    while hizquierdo($x$) $\neq$ NIL do
        $x$ := hizquierdo($x$)
    end while
    return $x$
```

- Elemento máximo...

```
Tree-Maximum($x$) {Devuelve un puntero al nodo con la clave mayor del árbol $x$, $x \neq$ NIL}
    while hderecho($x$) $\neq$ NIL do
        $x$ := hderecho($x$)
    end while
    return $x$
```

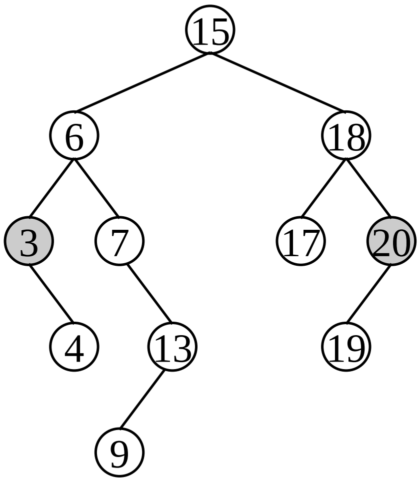


Figura 3.10: Ejemplo búsqueda mínimo. Llamada inicial (con $x \neq$ NIL): Tree-Minimum(raíz(T)). Coste $O(h)$, siendo h la altura del árbol.

3.4.3 Sucesor y predecesor en un ABB (en secuencia *inorden*)

Sucesor en *inorden* de x: nodo con menor clave mayor que key(x)

- Sucesor del 15: mínimo elemento en el subárbol derecho del nodo → 17
- Sucesor del 13: no tiene subárbol derecho, con lo que su sucesor es el menor antecesor cuyo hijo izquierdo sea también un antecesor → 15

Ejercicio. Predecesor. Escribe el algoritmo Tree-Predecessor.

```
Tree-Successor($x$) {Devuelve un puntero al sucesor del nodo $x$ si existe o NIL si la clave de $x$ es la mayor del ABB }
  if hderecho($x$) $\neq$ NIL then
    return Tree-Minimum(hderecho($x$))
  end if
  {Buscar el menor antecesor de $x$ cuyo hijo izquierdo también sea antecesor de $x$}
  $y$ := padre($x$)
  while $y \neq$ NIL $\wedge$ $x$ = hderecho($y$) do
    $x := y$; $y$ := padre($y$)
  end while
  return $y$
```

Figura 3.11: Algoritmo Tree-Successor. Coste $O(h)$.

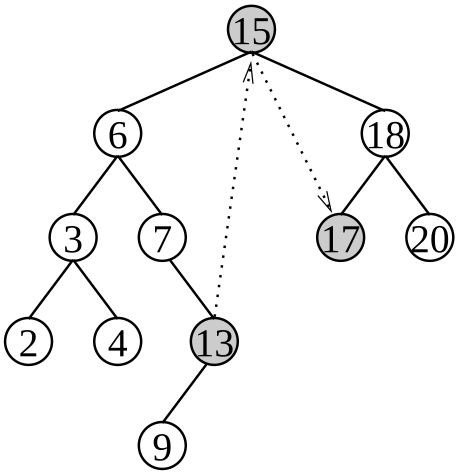


Figura 3.12: Ejemplo sucesor en ABB. Recorrido inorden: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20.

3.4.4 Insertar en un ABB

Ejercicio. Versión recursiva. Escribe una versión recursiva de Tree-Insert.

Ejercicio. Algoritmo de ordenación. Dada una secuencia de n enteros, diseña un algoritmo de ordenación utilizando un ABB. Estudia el coste del algoritmo.

```
Tree-Insert($T, z$) {Inserta el elemento apuntado por $z$ en una posición adecuada en un ABB $T$, key($z$) $= k$, hizquierdo($z$) $=$ NIL, hderecho($z$) $=$ NIL}
  $y :=$ NIL; $x :=$ raíz($T$)
  while $x \neq nil$ do
    $y := x$
    if key($z$) $<$ key($x$) then
      $x :=$ hizquierdo($x$)
    else
      $x :=$ hderecho($x$)
    end if
  end while
  padre($z$) $:= y${ ...si puntero al padre}
  if $y =$ NIL then
    raíz($T$) $:= z$
  else
    if key($z$) $<$ key($y$) then
      hizquierdo($y$) $:= z$
    else
      hderecho($y$) $:= z$
    end if
  end if
```

Figura 3.13: Algoritmo Tree-Insert. Coste $O(h)$.

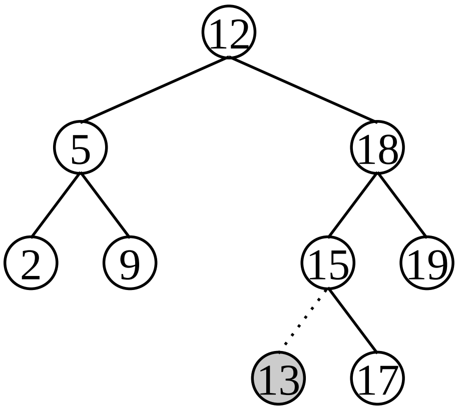
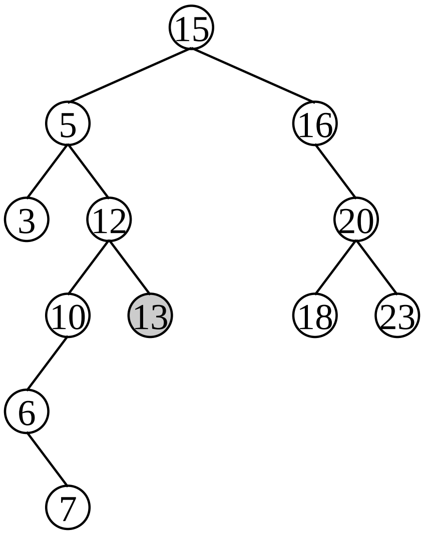
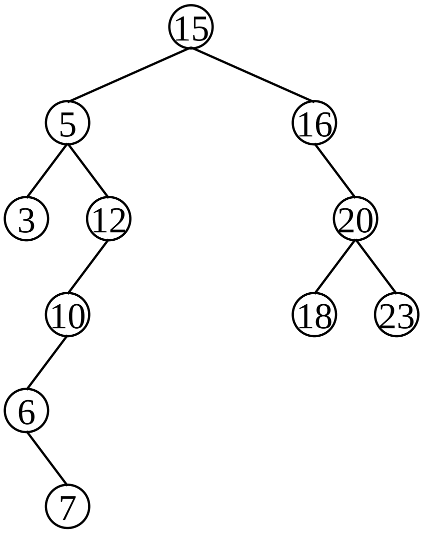


Figura 3.14: Ejemplo Tree-Insert.

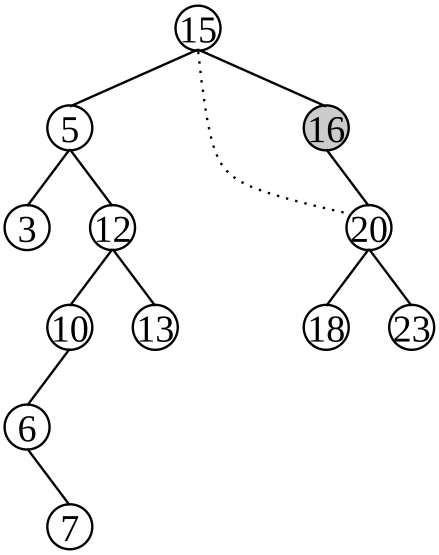
3.4.5 Borrar en un ABB

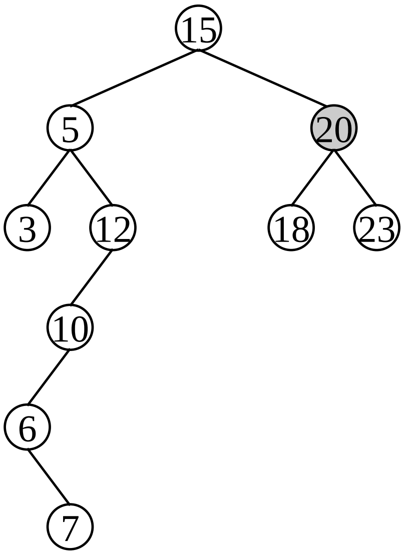
Caso 1. El elemento es una hoja: se borra y ya está. Tree-Delete$(T, 13)$





Caso 2. El elemento tiene un hijo: se borra y su hijo ocupa su posición. Tree-Delete$(T, 16)$





Caso 3. El elemento tiene dos hijos: se borra y su lugar es ocupado por el menor hijo del subárbol derecho (o el mayor hijo del subárbol izquierdo). Tree-Delete$(T, 5)$

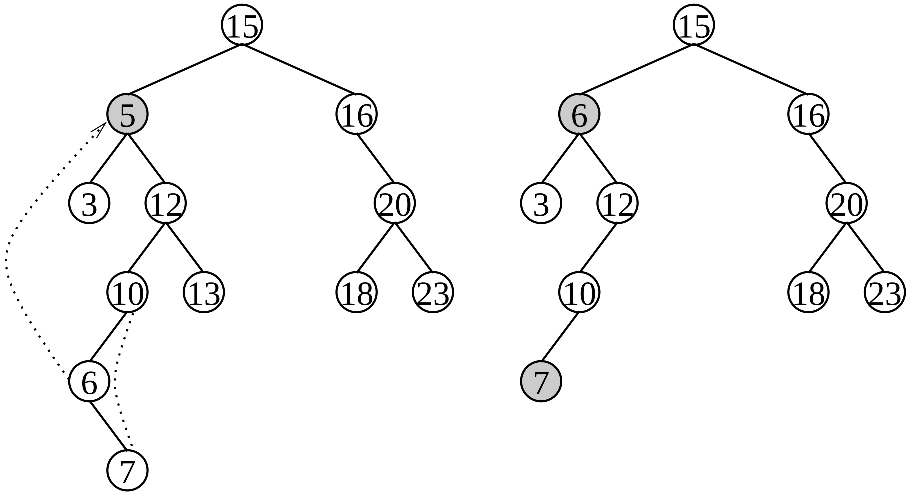


Figura 3.15: Ejemplo borrado en ABB. Casos 1, 2 y 3.

```
Tree-Delete($T, z$) {Borra el elemento apuntado por $z$}
    if hizquierdo($z$) $=$ NIL $\vee$ hderecho($z$) $=$ NIL then $y := z$
    else $y :=$ Tree-Successor($z$)
    if hizquierdo($y$) $\neq$ NIL then $x :=$ hizquierdo($y$)
    else $x :=$ hderecho($y$)
    if $x \neq$ NIL then padre($x$) $:=$ padre($y$)
    if padre($y$) $=$ NIL then
        raíz($T$) $:= x$
    else
        if $y =$ hizquierdo(padre($y$)) then hizquierdo(padre($y$)) $:= x$
        else hderecho(padre($y$)) $:= x$
    end if
    if$y \neq z$ then key($z$) $:=$ key($y$) {Si $y$ tiene otros campos, copiar}
    return $y$
```

Figura 3.16: Algoritmo Tree-Delete. Coste $O(h)$.

Ejercicio. Borrar máximo. Escribe un algoritmo iterativo que borre el nodo de máximo valor en un árbol binario de búsqueda.

3.4.6 Operaciones con ABBs (diccionarios y colas de prioridad)

Tree-Search(T, k)
Tree-Insert(T, x)
Tree-Delete(T, x)
Tree-DeleteMin(T, x), Tree-DeleteMax(T, x)
Tree-Minimum(T), Tree-Maximum(T)
Tree-Predecessor(x), Tree-Successor(x)

Coste operaciones: $O(h)$, siendo h la altura del ABB ($h \in O(\log n)$; $h \in O(n)$)

$$\boxed{\text{Altura de un ``ABB construido aleatoriamente'': } O(\log n)}$$

Árboles AVL, árboles rojos-negros, árboles 2-3, ... ← altura $O(\log n)$

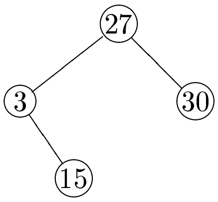
3.4.7 Ejercicios

Ejercicio. Vec2Arb. Dado un conjunto de enteros almacenados de forma creciente en un vector, escribe un algoritmo "divide y vencerás" eficiente que disponga los elementos en un árbol de búsqueda de altura mínima.
Estrategia: dejar el elemento que está en la posición central m del subvector $v[i..j]$ como el elemento a insertar en la raíz, mientras que los elementos $v[i..m-1]$ se insertarán en el subárbol izquierdo y los elementos $v[m+1..j]$ en el subárbol derecho.

Ejercicio. ABB con campo adicional. Dado un nodo de un ABB, se define un campo adicional `talla_hizquierdo` como el número de nodos que hay en el subárbol izquierdo más uno. Diseña un algoritmo iterativo para insertar un elemento e en un ABB de estas características, asumiendo que no existe en el árbol.

Ejercicio. Implementación. Se tiene un árbol binario de búsqueda de enteros representado en forma de vector que se ha definido como:

```
#define N ...
#define vacio -1

typedef struct {
     int hizq, hder;
     int elem;
} nodo;

typedef struct {
     int raiz;
     int disponibles;
     nodo v[N];
} arbol;
```



	hizq	elem	hder	
0	-1			
1	-1	30	-1	
2	4	27	1	**raíz**
3	6			**disponibles**
4	-1	3	5	
5	-1	15	-1	
6	0			

Los campos `hizq` y `hder` indican la posición del hijo izquierdo y derecho, respectivamente. El campo `disponibles` apunta al primer nodo de una lista enlazada de nodos que no forman parte del árbol. En un nodo de la lista de nodos disponibles, los campos `hder` y `elem` no están definidos, y el campo `hizq` apunta al siguiente nodo disponible, o a `vacio` si es el último.

Escribe un algoritmo iterativo para insertar un elemento x en un árbol A, sabiendo que $x \notin A$.

Ejercicio. ABB. Modifica el código del algoritmo de inserción en un árbol binario de búsqueda para que adicionalmente cuente el número de veces que ha aparecido una palabra en un texto.

3.5 Árboles parcialmente ordenados: montículos

Un árbol binario es **casi-completo** si el árbol es completo, a excepción quizás en el nivel de las hojas, en el cual todas las hojas están tan a la izquierda como sea posible.

Árbol **parcialmente ordenado (Montículo o Heap)**: Es un árbol binario casi-completo, con la *propiedad de orden*: el valor de cualquier nodo es mayor o igual –MaxHeap– (o menor o igual –MinHeap–) que el de sus nodos hijos.

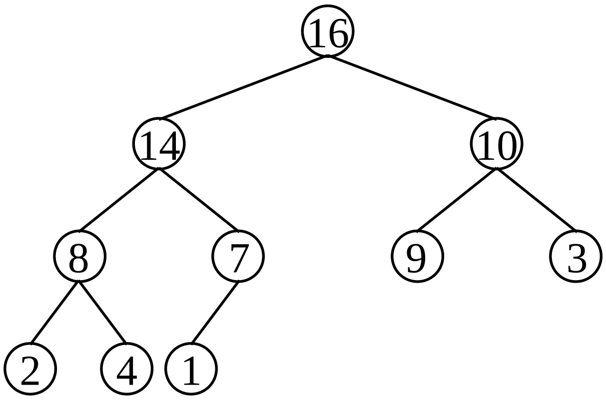


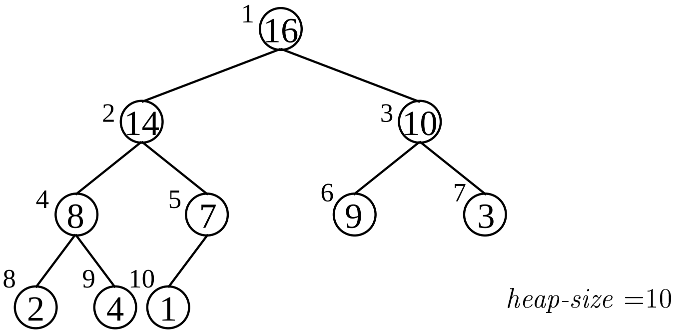
Figura 3.17: Ejemplo de un Heap.

- $A[1]$ es la raíz y dado un nodo $A[i]$,
  - si $2i \leq n$: $A[2i]$ hijo izquierdo
  - si $2i+1 \leq n$: $A[2i+1]$ hijo derecho
  - si $i \neq 1$: $A[\lfloor i/2 \rfloor]$ padre
- $\boxed{A[\text{padre}(i)] \geq A[i], \quad 2 \leq i \leq n}$
- Heap de n elementos: altura $\Theta(\log n)$

Propiedades:	todas las ramas del árbol son secuencias ordenadas, la raíz del árbol es el nodo de máximo (mínimo) valor, y todo subárbol de un Heap es, a su vez, un Heap.

3.5.1 Implementación de un Heap

Un AB casi-completo, y por lo tanto un Heap, admite una representación en forma de vector, numerando sus nodos de arriba abajo y de izquierda a derecha, fijando este orden la posición en el vector.



	1	2	3	4	5	6	7	8	9	10	11	...	
A	16	14	10	8	7	9	3	2	4	1			

```
#define maxTallaHeap 1024
int A[maxTllHeap+1];         /* Representación vectorial de un AB casicompleto */
#define tallaH A[0]   /* La talla definida se almacena en la componente 0 de A */
                      /* Ojo!!  esto sólo es posible porque element == int ... */
#define pad(i) (int)(i/2)         /* funciones inline para manejo de A como AB */
#define izq(i)      (2*i)
#define der(i)      (2*i+1)
```

Figura 3.18: Implementación de un Heap sobre un vector.

Ejercicio. Elementos en Heap. Suponiendo que todos los elementos de un MaxHeap son diferentes,

a) ¿dónde estará el máximo?

b) ¿dónde estará el mínimo?

c) ¿cualquier elemento que sea una hoja es menor que un elemento de un nodo interno?

d) ¿es un vector ordenado de forma decreciente un MaxHeap?

e) ¿es la secuencia $\{23, 17, 14, 6, 13, 10, 1, 5, 7, 12\}$ un MaxHeap?

3.5.2 Operaciones básicas sobre Heaps

Demostrarán la utilidad de esta estructura para:

- el diseño de un algoritmo de **ordenación rápido** (Heapsort)
- la representación de **colas de prioridad**

Insert(S, x)	añade el elemento apuntado por x en el conjunto S
Maximum(S)	devuelve el elemento de S cuya clave sea máxima
DeleteMax(S)	elimina y devuelve el elemento de S cuya clave sea máxima

Tabla 3.6: Operaciones básicas sobre Heaps.

Heapify ("Ajustar")	es la operación clave para mantener la propiedad de Heap. $O(\log n)$
Build-Heap ("Construir heap")	para convertir un vector, en principio desordenado, en un Heap. $O(n)$
Heapsort	para ordenar un vector. $O(n \log n)$
Heap-Maximum ("Consulta máximo")	devuelve el máximo (`return` $A[1]$). $O(1)$
Heap-Extract-Max ("Eliminar Máximo")	para extraer el máximo del conjunto y eliminarlo. $O(\log n)$
Heap-Insert ("Insertar")	para insertar en el conjunto. $O(\log n)$

3.5.3 Reconstrucción de un Heap: Heapify

Problema: Se tiene un Heap A y sea i, $1 \leq i \leq n$, una posición del vector tal que el subárbol izquierdo $(2i)$ y el derecho $(2i+1)$ son Heaps. Se desea transformar el subárbol que parte del nodo i en un Heap.

Estrategia Heapify: "hunde" el elemento $A[i]$ en el heap de forma que el subárbol que tiene por raíz el elemento con índice i se convierte en un Heap.

Coste Heapify: si h es la altura del árbol que tiene por raíz el elemento con índice i,

$$T(h) = \begin{cases} c_1, & \text{si } h = 0; \\ c_2, & \text{si } A[i] \text{ es un Heap;} \\ T(h-i) + c_3, & \text{si } A[i] \text{ no es un Heap} \end{cases}$$

Mejor caso: $\Omega(1)$ si $A[i]$ es un heap
Peor caso: $O(h) \in O(\log n)$, $n = \textit{heap-size}[A]$

Ejercicio. Traza. Hacer una traza de Heapify$(A, 3)$ sobre el vector $A = \{27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\}$.

Ejercicio. Heapify. ¿Cuál es el efecto de la acción Heapify(A, i) cuando el elemento $A[i]$ es mayor que sus hijos?

Ejercicio. Heapify. ¿Cuál es el efecto de la acción Heapify(A, i) para $i > (\textit{heap-size}(A)/2)$?

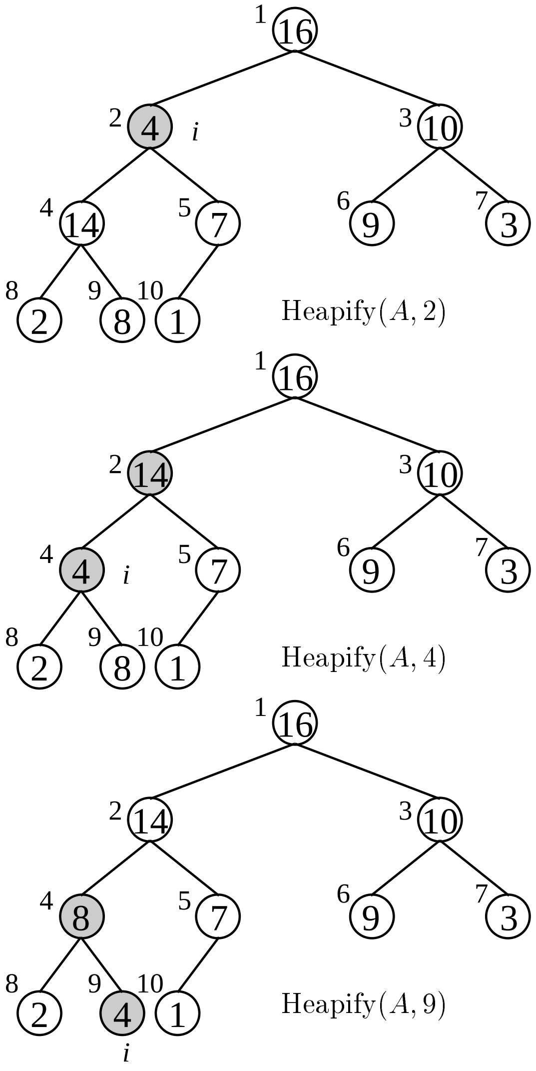


Figura 3.19: Ejemplo Heapify.

```
Heapify$(A, i)$ {left$(i)$ y right$(i)$ son Heaps }
    $l :=$ left$(i)$; $r :=$ right$(i)$;
    if $l \leq$ heap-size$(A) \wedge A[l] > A[i]$ then
        $largest := l$
    else
        $largest := i$
    end if
    if $r \leq$ heap-size$(A) \wedge A[r] > A[largest]$ then
        $largest := r$
    end if
    if $largest \neq i$ then
        $A[i] \longleftrightarrow A[largest]$
        Heapify$(A, largest)$
    end if{$A[i..$heap-size$(A)]$ es un Heap }
```

Figura 3.20: Algoritmo Heapify.

3.5.4 Construir un Heap: Build-Heap

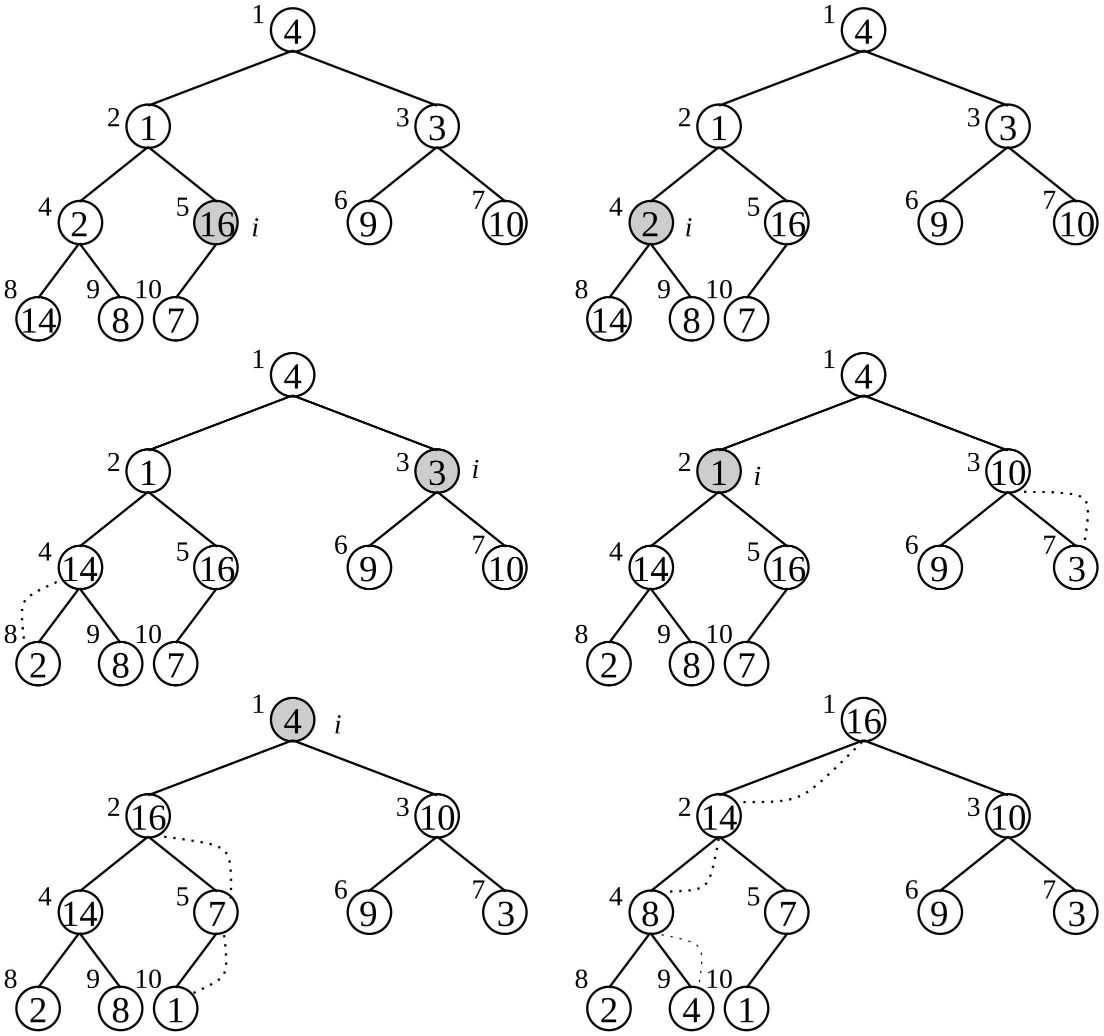


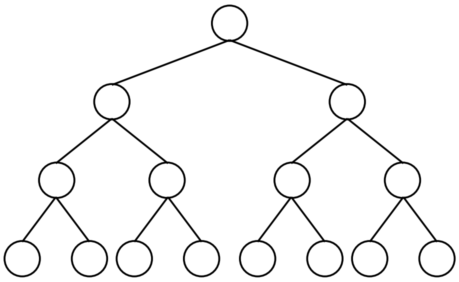
Figura 3.21: Construcción de un Heap. Ejemplo. Build-Heap: convierte un vector A en un Heap. Construye un Heap a partir del vector $A = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$.

Complejidad temporal de Build-Heap:

- Cota NO ajustada: $O(n)$ llamadas a Heapify: $O(n \log n)$
- $\boxed{\text{Cota ajustada: } O(n)}$
  - Coste Heapify de un nodo es proporcional a su altura $O(h)$
  - Propiedad: En un Heap de n elementos hay, como mucho, $2^{\lfloor \log n \rfloor}/2^h$ nodos de altura h.

```
Build-Heap($A$) {Convierte un vector $A[1..n]$ en
  un Heap }
  heap-size($A$) := $n$
  **for** $i := \lfloor n/2 \rfloor$ downto 1 **do**
    Heapify($A, i$)
  **end for**
```

Figura 3.22: Algoritmo Build-Heap.



Altura	Núm. nodos en h
$h = \lfloor \log 15 \rfloor - 0 = 3$	$2^0 = 2^{\lfloor \log 15 \rfloor}/2^3$
$h = \lfloor \log 15 \rfloor - 1 = 2$	$2^1 = 2^{\lfloor \log 15 \rfloor}/2^2$
$h = \lfloor \log 15 \rfloor - 2 = 1$	$2^2 = 2^{\lfloor \log 15 \rfloor}/2^1$
$h = \lfloor \log 15 \rfloor - 3 = 0$	$2^3 = 2^{\lfloor \log 15 \rfloor}/2^0$

$$\begin{aligned} T(n) &\leq \sum_{h=0}^{\lfloor \log n \rfloor} \frac{2^{\lfloor \log n \rfloor}}{2^h} O(h) \leq O\left(\sum_{h=0}^{\log n} 2^{\log n} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\log n} h \left(\frac{1}{2} \right)^h \right) \\ &= O\left(n \sum_{h=0}^{\infty} h \left(\frac{1}{2} \right)^h \right) \in O(n) \end{aligned}$$

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \quad 0 < x < 1, \qquad \text{si } x = 1/2, \quad \sum_{k=0}^{\infty} k \left(\frac{1}{2} \right)^k = \frac{1/2}{(1-1/2)^2} = 2$$

Ejercicio. Traza Build-Heap. Realiza una traza de Build-Heap con el vector $A = \{5, 3, 17, 10, 84, 19, 6, 22, 9\}$.

3.5.5 Extraer el máximo elemento en un Heap: Heap-Extract-Max

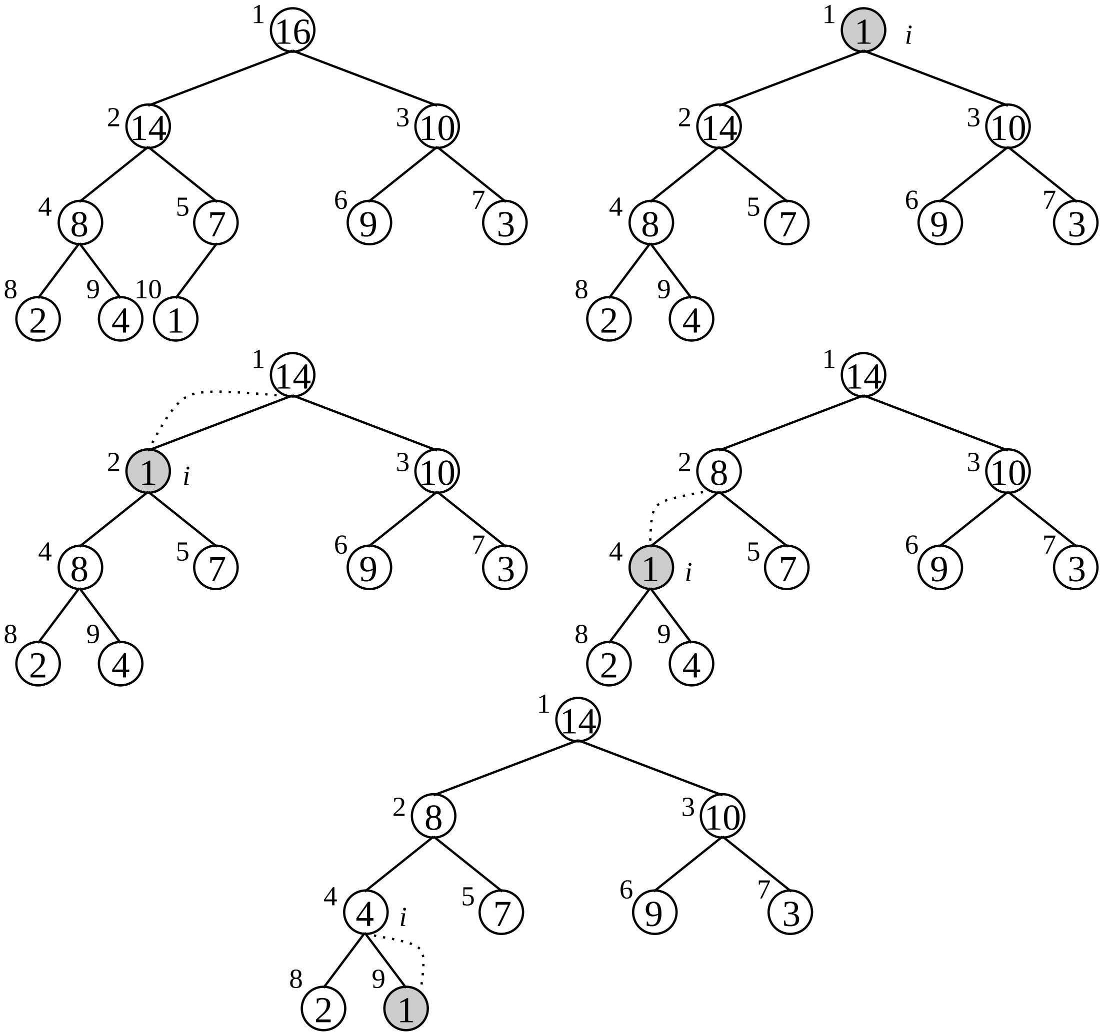


Figura 3.23: Ejemplo de Heap-Extract-Max.

```
Heap-Extract-Max($A$) {Devuelve el máximo de A y lo borra}
  if $heap\text{-}size(A) < 1$ then
    return Error “Heap underfow”
  end if
  $max := A[1]$
  $A[1] := A[heap\text{-}size(A)]$
  $heap\text{-}size(A) := heap\text{-}size(A) - 1$
  Heapify($A, 1$)
  return $max$
```

Figura 3.24: Algoritmo Heap-Extract-Max. El coste temporal $O(\log n)$, ya que realiza sólo un trabajo constante antes de llamar a Heapify, que tiene un coste $O(\log n)$.

3.5.6 Insertar un elemento en un Heap: Heap-Insert

Ejercicio. Traza Insertar. Hacer una traza de Heap-Insert$(A, 3)$ sobre el Heap $A = \{15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\}$.

Ejercicio. Traza Heap-Extract-Max. Hacer una traza de Heap-Extract-Max sobre el mismo Heap.

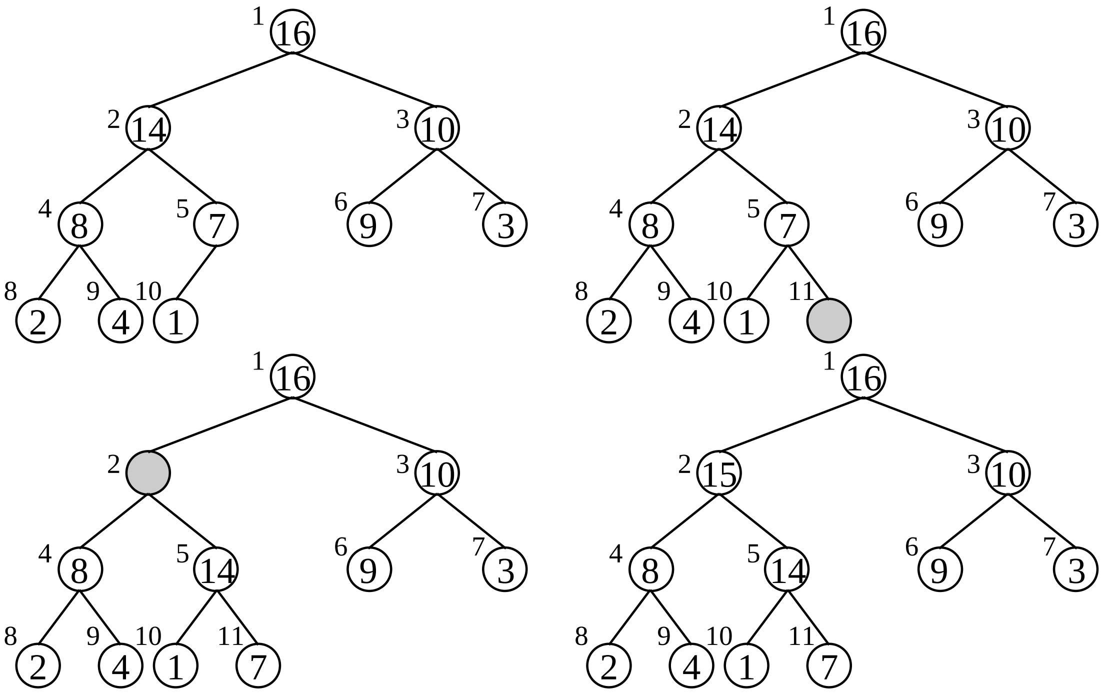


Figura 3.25: Ejemplo: Heap-Insert$(A, 15)$.

```
Heap-Insert$(A, key)$ {Añade el elemento key al Heap A}
    $heap\text{-}size(A) := heap\text{-}size(A) + 1$
    $i := heap\text{-}size(A)$
    **while** $i > 1 \wedge A[\text{parent}(i)] < key$ **do**
        $A[i] := A[\text{parent}(i)]$
        $i := \text{parent}(i)$
    **end while**
    $A[i] := key$
```

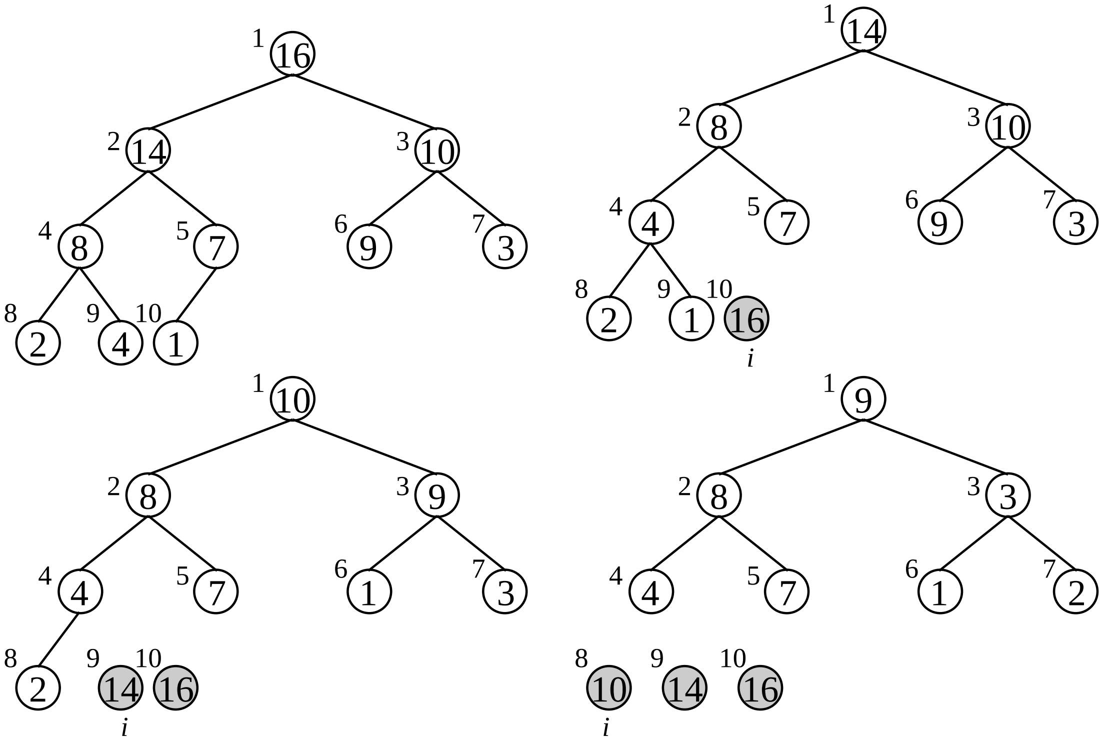
Figura 3.26: Algoritmo Heap-Insert. El coste temporal es $O(\log n)$, ya que el camino trazado de la nueva hoja a la raíz del heap tiene talla $O(\log n)$.

3.5.7 Algoritmo de ordenación Heapsort

Estrategia Heapsort: Se insertan los elementos en un MaxHeap y se va eliminando sucesivamente el mayor para obtener los elementos en orden inverso.

```
Heapsort($A$) {Ordena el vector $A[1..n]$}
    Build-Heap($A$)
    for $i$ := heap-size($A$) downto 2 do
        $A[1] \longleftrightarrow A[i]$
        heap-size($A$) := heap-size($A$) − 1
        Heapify (A,1)
    end for
```

Figura 3.27: Algoritmo Heapsort. La complejidad temporal es el coste de Build-Heap + coste del bucle ($n-1$ llamadas a Heapify), por tanto $T(n) \in O(n) + (n-1)O(\log n) \in O(n \log n)$.



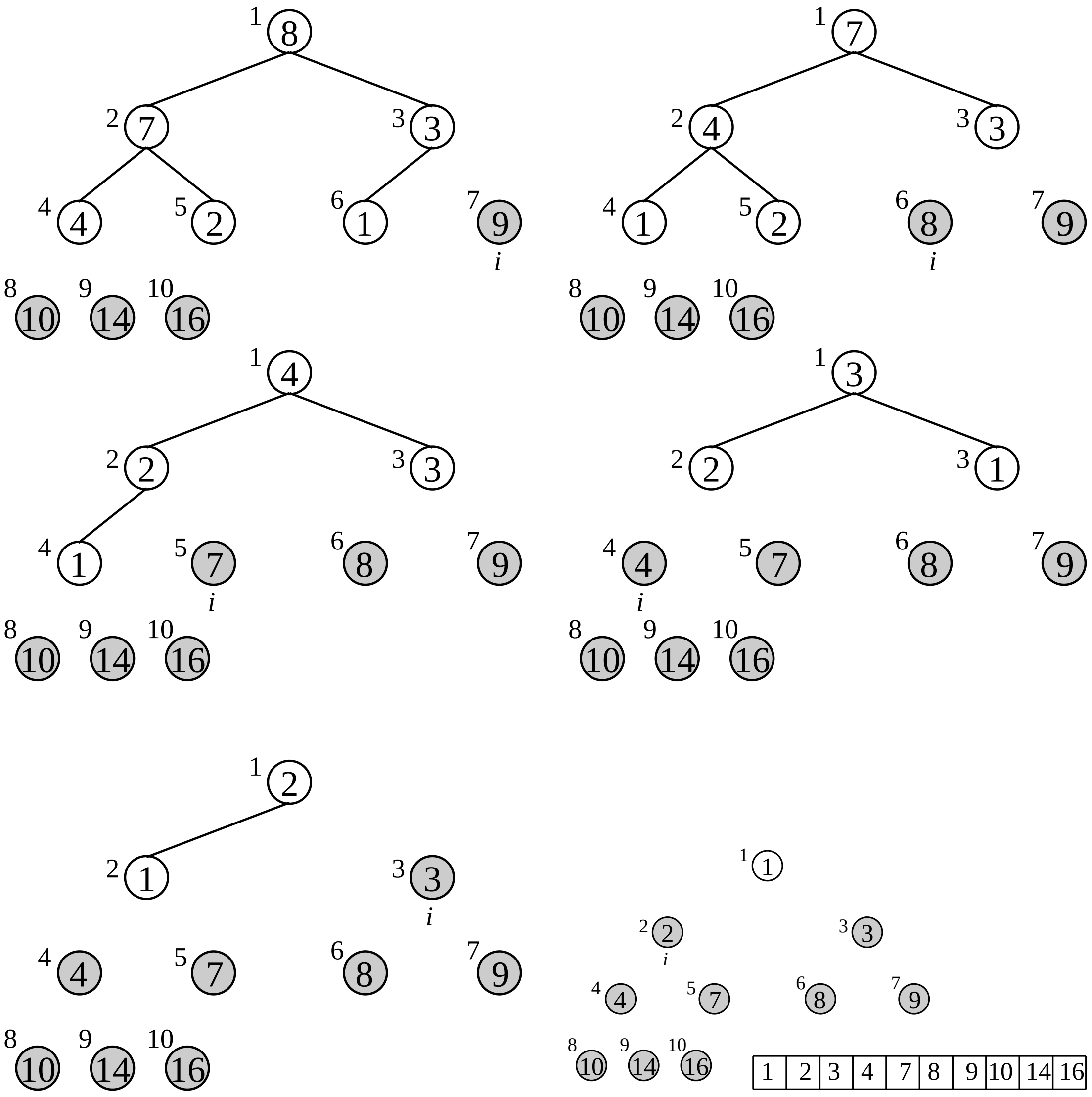


Figura 3.28: Ejemplo Heapsort.

Propiedades Heapsort:

- “sort in-place” (como Quicksort; Mergesort necesita espacio adicional)
- Se garantiza el coste $O(n \log n)$, independientemente de la entrada → no hay peor caso (a diferencia de Quicksort). El “precio” que hay que pagar es que realiza más comparaciones que Quicksort, por lo que tiende a ser más lento en la práctica.

Ejercicio. Traza Heapsort. Hacer una traza de Heapsort sobre el vector $A = \{$ 5, 13, 2, 25, 7, 17, 20, 8, 4$\}$.

Ejercicio. Coste Heapsort. ¿Cuál es el coste de Heapsort sobre un vector de longitud n que esté ya ordenado de forma creciente? ¿Y si está ordenado de forma decreciente? ¿Y si todos sus elementos son iguales?

3.5.8 Ejercicios

Ejercicio. Heap. Diseña un algoritmo que borre el elemento que ocupa la posición i-ésima en un MaxHeap. Analiza la complejidad temporal del algoritmo.

```
Heap-Delete($A, i$) {Borra el elemento $A[i]$}
    if $heap\text{-}size(A) < 1$ then
        return Error "Heap underfow"
    end if
    $item := A[i]$
    $A[i] := A[heap\text{-}size(A)]$
    $heap\text{-}size(A) := heap\text{-}size(A) - 1$
    while $i > 1 \land A[i] > A[\mathrm{padre}(i)]$ do
        $A[i] \longleftrightarrow A[\mathrm{padre}(i)]$
        $i := \mathrm{padre}(i)$
    end while
    Heapify($A, i$)
    return $item$
```

El coste de Heap-Delete presenta un mejor caso cuando el elemento que se coloca en $A[i]$ no viola la propiedad del Heap. En el peor caso, el coste es idéntico al del borrado del máximo, es decir, $O(\log n)$, siendo n la talla del Heap.

Ejercicio. Heap. Diseña un algoritmo para resolver el problema de encontrar el k-ésimo menor elemento de un vector de talla n, con $n \geq 1$, utilizando un Heap y cuyo coste sea $O(n + k \log n)$.

Ejercicio. Heap. Realiza una traza de las operaciones siguientes sobre un Heap, inicialmente vacío: Heap-Insert(10), Heap-Insert(12), Heap-Insert(21), Heap-Insert(40), Heap-Insert(25), Heap-Insert(70), Heap-Insert(88), Heap-Extract-Max, Heap-Extract-Max.

Ejercicio. Heap. Dado un conjunto de $n > 0$ enteros representados en un Max-Heap, diseña un algoritmo que devuelva el mínimo del conjunto, haciendo el menor número de comparaciones posible.

Ejercicio. Heap. Escribe un algoritmo eficiente que elimine el mínimo elemento de un Max-Heap (heap en el que cualquier elemento es mayor o igual que sus hijos).

Ejercicio. Heap. Simula el comportamiento de una pila con un MaxHeap.

Ejercicio. Heap. Construye un MaxHeap a partir de otros dos.

3.6 Implementación de conjuntos. Resumen

Tabla 3.7: Coste operaciones de conjuntos con distintas estructuras. S conjunto, $n = |S|$, x puntero a objeto, k clave del objeto. D representa diccionario y C cola de prioridad.

		Coste promedio			
	Operación	Lista ordenada	ABB ($h \in O(\log n)$)	Tabla hash	Heap
D	Search(S, k)	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$
D, C	Insert(S, x)	$O(n)$	$O(\log n)$	$O(1)$	$O(\log n)$
D	Delete(S, x)	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$
C	Minimum(S)	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$
C	DeleteMin(S)	$O(1)$	$O(\log n)$	$O(n)$	$O(\log n)$

Ejercicio. Conjuntos. Tenemos un conjunto de enteros (sin elementos repetidos) en el que se están realizando continuamente operaciones de eliminar el máximo, añadir elementos y comprobar si un elemento dado pertenece o no al conjunto. Tenemos la posibilidad de utilizar las siguientes estructuras de datos para representar el conjunto:

(a) un heap,

(b) un árbol binario de búsqueda (balanceado),

(c) un heap combinado con una tabla de dispersión en la que se guarda cada uno de los elementos junto con la posición que ocupa al heap.

¿Qué estructura de datos no utilizarías y por qué?

Ejercicio. Búsqueda de los k menores elementos. De un vector v de enteros cuya talla es n deseamos obtener los k menores elementos (con $k \leq n$). Para ello disponemos de los siguientes algoritmos:

a) Ordenar ascendentemente el vector utilizando Quicksort y devolver los k primeros elementos.

b) Transformar el vector v en un MinHeap y extraer k veces el mínimo.

c) Introducir los elementos en un ABB y hacer un recorrido en inorden hasta obtener k elementos.

Se pide estudiar el coste temporal asintótico de estos tres algoritmos y en función del coste determinar en qué condiciones es preferible cada uno de ellos.

Ejercicio. Multiconjunto. Se tiene un multiconjunto C de enteros de talla $n \geq 0$ (es decir, puede haber elementos repetidos). Si se eliminan los elementos repetidos de C se obtiene un conjunto de talla m, $0 \leq m \leq n$. Se desea listar los elementos de C sin repeticiones, ordenados por frecuencia absoluta de mayor a menor.

Ejemplo: si $C = \{3, 5, 5, 1\}$, la solución puede ser $5, 1, 3$ o bien $5, 3, 1$.

Para solucionar este problema disponemos de los siguientes algoritmos:

Algoritmo 1:	Insertar los elementos en una tabla de dispersión actualizando la frecuencia de aparición adecuadamente. Pasar los elementos a un vector. Ordenar los elementos por frecuencia de aparición con Heapsort. Listar los elementos del vector ordenado.
Algoritmo 2:	Insertar los elementos en un ABB actualizando la frecuencia de aparición adecuadamente. Pasar los elementos a otro ABB, tomando la frecuencia como clave (supondremos que este ABB admite elementos con claves repetidas). Listar los elementos haciendo un recorrido en inorden del último árbol.

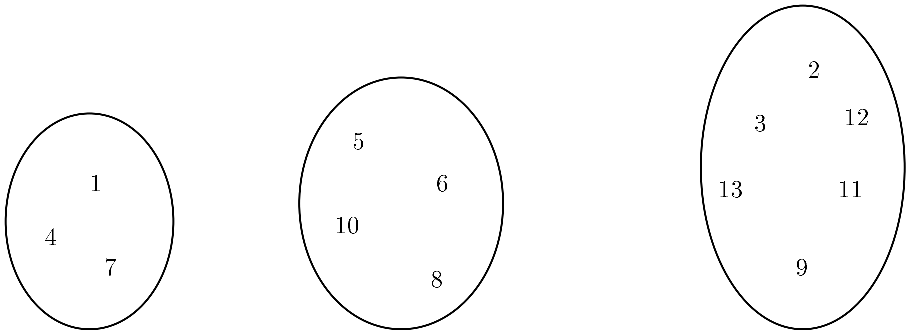
Se pide estudiar coste temporal cuando $m \approx n$ y cuando $m <<<< n$.

3.7 Conjuntos disjuntos: MFset

El problema de la Unión-Búsqueda (Merge-Find) requiere el desarrollo de estructuras de datos eficientes para agrupar n elementos distintos en una colección de conjuntos disjuntos $S = \{S_1, S_2, \ldots, S_k\}$ con dos tipos de operaciones: unión de dos conjuntos disjuntos y búsqueda para saber a qué conjunto pertenece un elemento.

Un MFset ("Merge-Find Set") es una estructura tipo conjunto en la que los elementos están organizados en subconjuntos **disjuntos** y el número de elementos es **fijo** (no se añaden ni se borran). Las operaciones características son:

- Combina (MF-Merge): unión de dos conjuntos disjuntos
- Encuentra (MF-Find): dado un elemento debe determinar a qué conjunto pertenece



$1 \approx 7 \approx 4$ $\quad 8 \approx 5 \approx 6 \approx 10$ $\quad 3 \approx 13 \approx 12 \approx 9 \approx 11 \approx 2$

Cada subconjunto se **identifica** por algún miembro.

Aplicaciones (relaciones de equivalencia): Equivalencia entre autómatas finitos, Componentes conexas de un gnd, Obtención del árbol de recubrimiento de mínimo peso en un gnd (Kruskal).

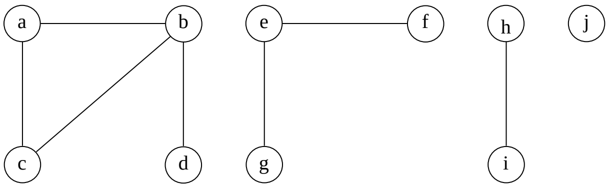
3.7.1 Operaciones sobre MFsets

Tabla 3.8: Operaciones sobre MFsets. x e y son elementos (objetos) del conjunto.

MF-Build-Set(x)	crea un nuevo conjunto cuyo único miembro es x
MF-Merge(x, y)	une los conjuntos a los que pertenecen x e y
MF-Find(x)	devuelve un puntero al representante del conjunto al que pertenece x

Ejemplo. Aplicación MFsets: Componentes conexas de un grafo

```
ComponentesConexas($V, A$)
  {Devuelve las componentes conexas del gnd $G = (V, A)$}
  for all $v \in V$ do
    MF-Build-Set($v$)
  end for
  for all $(u, v) \in A$ do
    if MF-Find($u$) $\neq$ MF-Find($v$) then
      MF-Merge($u, v$)
    end if
  end for
```



Arco	Colección de conjuntos disjuntos									
	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(a, b)	$\{a, c, b, d\}$				$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(e, f)	$\{a, c, b, d\}$				$\{e, g, f\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, c, b, d\}$				$\{e, g, f\}$			$\{h, i\}$		$\{j\}$

Figura 3.29: Ejemplo utilización MFset: componentes conexas.

3.7.2 Representación en árbol de MFsets

- Cada subconjunto es un árbol:
  - Los nodos del árbol son los elementos del conjunto.
  - Cada nodo apunta al padre.
  - La raíz del árbol puede usarse para nombrar el conjunto.
- El MFset estará representado por una colección de árboles (un bosque).

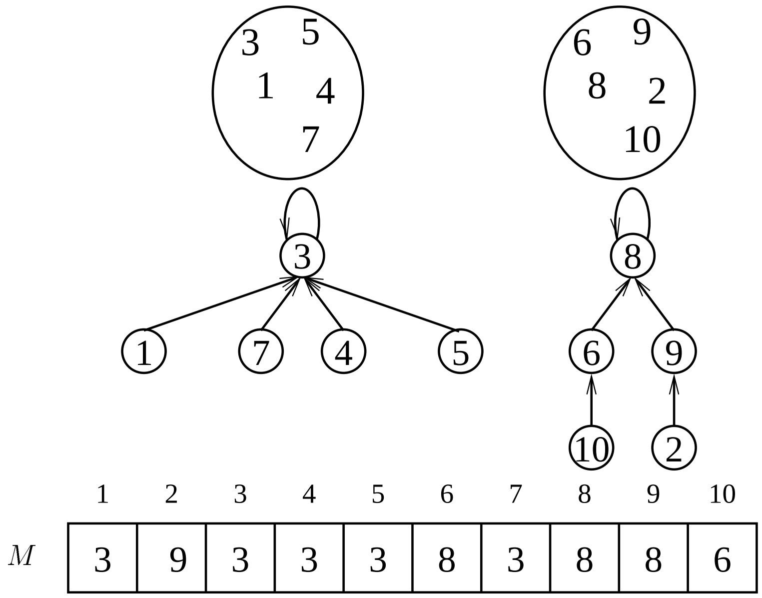


Figura 3.30: Representación de MFset.

Los árboles no son necesariamente binarios, pero su representación es muy fácil ya que sólo necesitaremos un apuntador al padre:

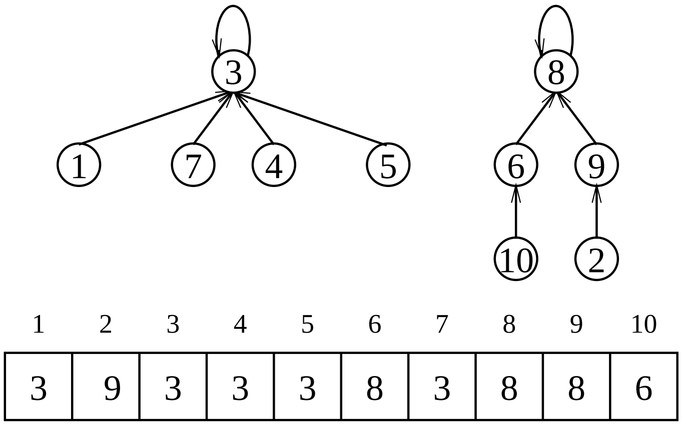
$$M \;=\; \texttt{vector}[1..n] \texttt{ de enteros}$$

- $M[x]$: puntero al padre del elemento x
- si $M[x] = x$, x es la raíz

3.7.3 Implementación operaciones sobre MFsets

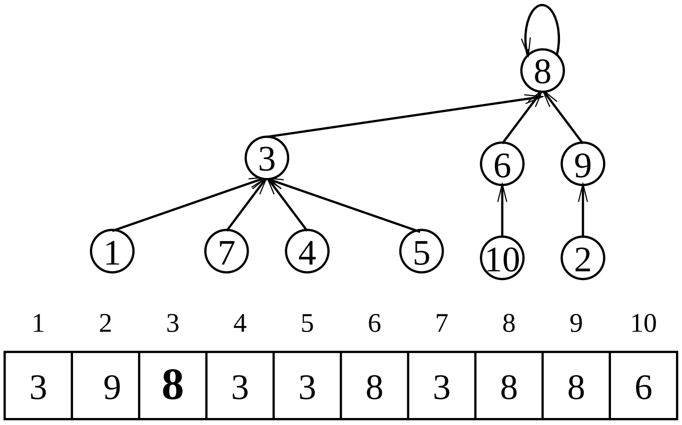
$M \;=\; \texttt{vector}[1..n] \texttt{ de enteros};$
n es el número de elementos
x e y son elementos del conjunto
$M[x]$: puntero al padre del elemento x
si $M[x] = x$, x es la raíz

- MF-Find(x): devuelve un puntero al representante del conjunto al que pertence x
  - devolver la raíz del árbol que contiene a x.
  - Coste proporcional a la profundidad del nodo, que en el PC es $n - 1$.
- MF-Merge(x, y): une los conjuntos a los que pertenecen x e y
  - hacer que la raíz de un árbol sea hijo del otro.
  - Coste constante $O(1)$ (si x e y son raices).



1	2	3	4	5	6	7	8	9	10
3	9	3	3	3	8	3	8	8	6

MF-Find(10) = 8; MF-Find(9) = 8; MF-Find(4) = 3; MF-Find(3) = 3



1	2	3	4	5	6	7	8	9	10
3	9	**8**	3	3	8	3	8	8	6

MF-Merge(3, 8)

Figura 3.31: Operaciones sobre MFsets. Ejemplos.

```
MF-Build-Set($x$)
    $M[x] := x$
```

```
MF-Merge($x, y$)
    $u :=$ MF-Find($x$); $v :=$ MF-Find($y$)
    $M[u] := v$
```

```
MF-Find($x$)
    $i := x$
    **while** $i \neq M[i]$ **do**
        $i := M[i]$
    **end while**
    **return** $M[i]$
```

```
MF-Find($x$){ Versión recursiva}
    **if** $x \neq M[x]$ **then**
        MF-Find($M[x]$)
    **end if**
    **return** $M[x]$
```

Figura 3.32: Algoritmos MF-Build-Set, MF-Merge y MF-Find con la implementación M = `vector[1..n] de enteros`.

Operaciones sobre MFsets. Caso peor

Una secuencia de m operaciones MF-Merge y MF-Find tiene un coste $O(mn)$ en el peor caso (se puede crear un árbol degenerado en una lista de n nodos).

0. MF-Build-Set (1), ...
1. MF-Merge (1,2), MF-Find (1)
2. MF-Merge (2,3), MF-Find (1)
3. MF-Merge (3,4), MF-Find (1)
4. MF-Merge (4,5), MF-Find (1)

	1	*2*	*3*	*4*	*5*
$\{\{1\},\{2\},\{3\},\{4\},\{5\}\}$	1	2	3	4	5
$\{\{1,2\},\{3\},\{4\},\{5\}\}$	2	2	3	4	5
$\{\{1,2,3\},\{4\}\{5\}\}$	2	3	3	4	5
$\{\{1,2,3,4\},\{5\}\}$	2	3	4	4	5
$\{\{1,2,3,4,5\}\}$	2	3	4	5	5

3.7.4 Mejoras del coste. Heurísticos

Una secuencia de m operaciones MF-Merge y MF-Find tiene un coste $O(mn)$ en el peor caso (árbol como lista).

Mejoras del coste a base de **reducir la altura del árbol**: m operaciones MF-Merge y MF-Find tendrán un coste casi lineal con m:

- Combinar por Rango
- Comprimir Caminos

Combinar por Rango

Idea: hacer la operación MF-Merge de forma que *la raíz del árbol con menos nodos apunte a la raíz del árbol con más nodos.* Para ello en cada nodo se mantiene un rango que aproxima el logaritmo de la talla del subárbol (cota de la altura del nodo). El empate se rompe arbitrariamente.

Se puede demostrar que si se realiza la unión por rango, la altura de un árbol de n elementos nunca será superior a $\lceil \log n \rceil$.

Compresión de Caminos

Idea: el efecto de la Compresión de Caminos sobre la operación MF-Find es que todo nodo en el camino de x a la raíz pasa a apuntar a la raíz del árbol.

Permite realizar las m operaciones de búsqueda con coste $O(m\,\alpha(m,n))$, donde $\alpha(m,n)$ es una inversa de la función de Ackerman, que crece muy lentamente.

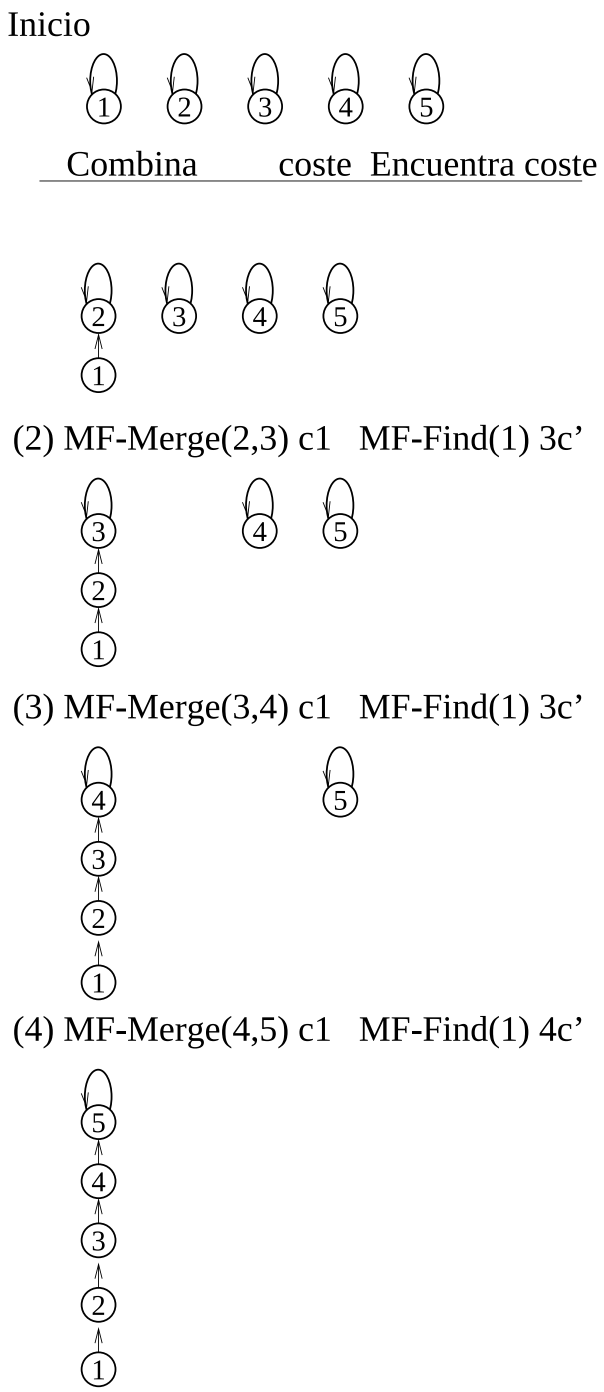
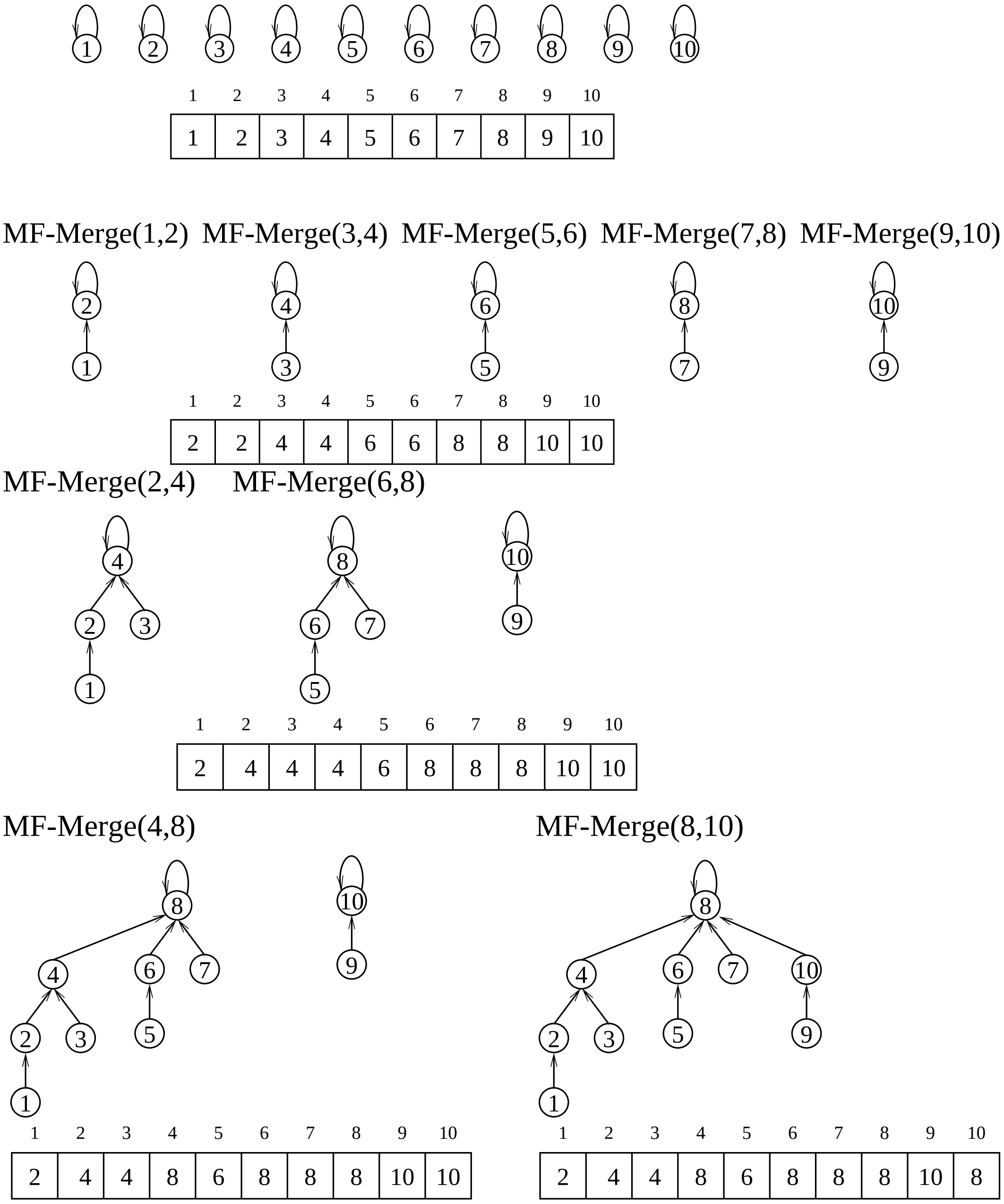


Figura 3.33: Caso peor: Coste secuencia de $n-1$ operaciones MF-Merge y MF-Find: $coste = \sum_{i=1}^{n-1} c + \sum_{i=1}^{n-1} ic\prime \in O(n^2)$.

```
MF-Build-Set($x$)
    $M[x] := x$
    $rango[x] := 0$
```

```
MF-Merge($x, y$)
    $u :=$ MF-Find($x$); $v :=$ MF-Find($y$)
    if $rango[u] > rango[v]$ then
        $M[v] := u$
    else
        $M[u] := v$
        if $rango[u] = rango[v]$ then
            $rango[v] := rango[v] + 1$
        end if
    end if
```

Figura 3.34: Algoritmo de combinación por rango.



1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	6	7	8	9	10
2	2	4	4	6	6	8	8	10	10

1	2	3	4	5	6	7	8	9	10
2	4	4	4	6	8	8	8	10	10

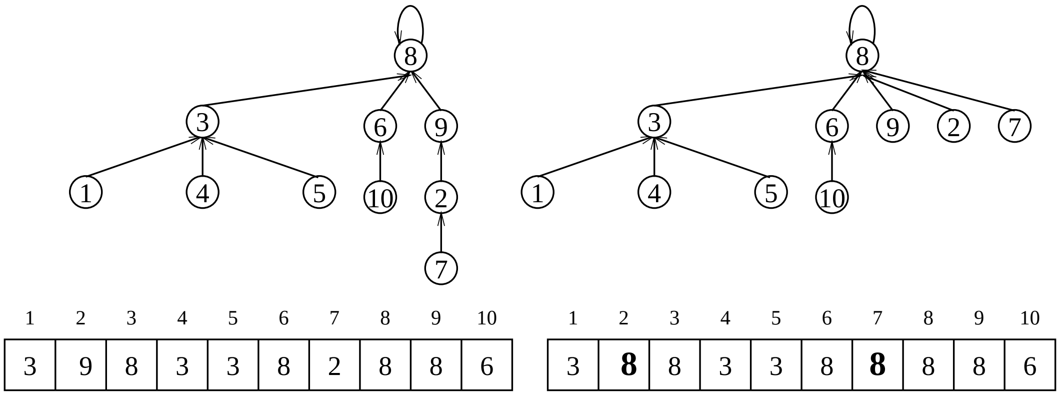
1	2	3	4	5	6	7	8	9	10
2	4	4	8	6	8	8	8	10	10

1	2	3	4	5	6	7	8	9	10
2	4	4	8	6	8	8	8	10	8

Figura 3.35: Combinación por Rango. Ejemplo.

```
MF-Find($x$)
    $r := x$
    while $r \neq M[r]$ do
        $r := M[r]$
    end while{$r$ es la raíz}
    $i := x${Compresión de caminos}
    while $i \neq r$ do
        $j := M[i]$
        $M[i] := r$
        $i := j$
    end while
    return $r$
```

Figura 3.36: Algoritmo de búsqueda con compresión de caminos.



1	2	3	4	5	6	7	8	9	10
3	9	8	3	3	8	2	8	8	6

1	2	3	4	5	6	7	8	9	10
3	**8**	8	3	3	8	**8**	8	8	6

Figura 3.37: Ejemplo búsqueda con compresión de caminos: MF-Find(7).

Coste amortizado

Secuencia de operaciones: $n-1$ operaciones MF-Merge (por rango)
m operaciones MF-Find

- Coste total si *no* se usa compresión de caminos: $O(m \log n)$
- Coste total si se usa compresión de caminos: $O(m\,\alpha(m+n,n))$. Lo usual es que $\alpha(m+n,n) \leq 4$ (ejem.: $\alpha(2^{65536}) = 5$) y, por lo tanto:
    - coste total: $O(m)$
    - coste amortizado: $O(1)$

3.7.5 Ejercicios

Ejercicio. Traza operaciones MFset. Escribe el $^{\text{o}}$mfset que se obtiene a partir del dado, después de ejecutar cada una de las operaciones de la secuencia indicada. Utiliza las técnicas de unión por rangos o por altura y compresión de caminos. En la unión, en caso de empate, la raíz del primer argumento pasa a ser hijo de la raíz del segundo argumento.

1	2	3	4	5	6	7	8	9	10
1	1	3	3	1	6	2	1	8	6

1. MF-Merge$(3,6)$
2. MF-Find(4)
3. MF-Merge$(1,6)$
4. MF-Find(10)

Ejercicio. MFset: Componentes conexas. Sea $G = (V, A)$ un grafo no dirigido con $n > 0$ nodos. Dado que la relación "ser dos nodos mutuamente conectados" es reflexiva, simétrica y transitiva, se pueden calcular sus componentes conexas en una variable S de tipo MF-Set, indexada por los nodos de V, de la siguiente forma:

- Se inicializa S con n subconjuntos disjuntos, uno por cada nodo de V.
- Para cada arco (u, v) de A, se unen los subconjuntos disjuntos a los que pertenecen u y v.

Se pide:

1. Escribe el algoritmo que se ha descrito, suponiendo que G está representado por su matriz de adyacencia A. Calcula el coste temporal del algoritmo.
2. Escribe un algoritmo que (a partir de este MF-Set S), obtenga el número de componentes conexas de G. Calcula el coste temporal del algoritmo.

Capítulo 4

Algoritmos voraces

Objetivos

- Estudio de la técnica de diseño de algoritmos voraces.
- Estudio de algunos problemas clásicos: mochila con fraccionamiento, algoritmo de Dijkstra, algoritmos de Kruskal y Prim.

Contenidos

1 Introducción. Ámbito de aplicación
2 Esquema general voraz
3 El problema de la mochila con fraccionamiento

Bibliografía

- *Introduction to Algorithms*, de Cormen, Leiserson y Rivest (sección 17.2)
- *Fundamentos de Algoritmia*, de Brassard y Bratley (capítulo 6)
- *Computer algorithms*, de Horowitz, Sahni y Rajasekaran (capítulo 4)

4.1 Introducción. Ámbito de aplicación

Se aplica normalmente a **problemas de optimización**: Búsqueda del valor **óptimo** (máximo o mínimo) de una cierta **función objetivo** en un **dominio** determinado. Ejemplo: problema de la mochila con fraccionamiento.

La solución se puede presentar como una secuencia de decisiones

- Medida local de optimización
- Las decisiones son irreversibles

No siempre encuentra la solución óptima, pero en ocasiones permite encontrar una solución aproximada con un coste computacional bajo.

Estrategia voraz: en cada paso del algoritmo se debe realizar una selección entre las existentes. La estrategia voraz aconseja realizar la elección que es mejor en ese momento. Tal estrategia no garantiza, de forma general, que encuentra la solución óptima al problema.

4.1.1 Ejemplo: Cajero automático

Cajero automático: Suministrar la cantidad de billetes solicitada de forma que el n^{o} total de billetes sea mínimo.

Ejemplo: $M = 11000$ pts. $B = \{1000, 2000, 5000\}$

- 11×1000 (11 billetes)
- $5 \times 2000 + 1 \times 1000$ (6 billetes)
- $2 \times 5000 + 1 \times 1000$ (3 billetes)

La estrategia voraz consiste en ir seleccionando el billete de mayor valor:

- A veces no hay solución; p.e. si $M = 3000$ y no hay billetes de 1000
- A veces no la encuentra; p.e. si $M = 11000$ y no hay billetes de 1000
- A veces encuentra una factible pero no óptima; p.e. $B = \{100, 500, 1100, 5000\}$ y $M = 6500$. La solución que se obtendría es: $1 \times 5000 + 1 \times 1100 + 4 \times 100$ (6 billetes) y la óptima es: $1 \times 5000 + 3 \times 500$ (4 billetes)

4.2 Esquema general voraz

C	Conjunto de elementos o candidatos
S	Conjunto de elementos de la solución en curso
solución(S)	¿Es S solución?
factible(S)	¿Es S completable o factible?
selecciona(C)	Seleccionar un candidato
f	Función objetivo

```
Voraz($C$)
  $S := \emptyset$ {conjunto vacío}
  while NOT(solución($S$)) $\wedge C \neq \emptyset$ do
    $x :=$ selecciona($C$)
    $C := C - \{x\}$
    if factible($S \cup \{x\}$) then
      $S := S \cup \{x\}$
    end if
  end while
  if solución($S$) then
    return $S$
  else
    return "No hay solución"
  end if
```

Figura 4.1: Esquema Voraz.

Algoritmo voraz: Obtiene un subconjunto de C que optimice la función objetivo. Para ello:

1. Seleccionar en cada instante un candidato
2. Añadir el candidato a la solución en curso si es completable, sino rechazarlo
3. Repetir 1 y 2 hasta obtener la solución

¡Importante!
- Un algoritmo voraz no siempre proporciona la solución óptima.
- Las decisiones son irreversibles.

Ejercicio. Cajero. Diseña una solución voraz para el problema del cajero.

4.3 El problema de la mochila con fraccionamiento

Tenemos una mochila de capacidad M y N objetos para incluir en ella. Cada objeto tiene un peso p_i y un beneficio b_i, $1 \leq i \leq N$.

Problema: ¿Cómo llenar la mochila de forma que el beneficio total de los elementos que contenga sea máximo? Los objetos se pueden fraccionar.

Planteamiento del problema

- Datos del problema: capacidad M y N objetos de peso $(p_1, \dots, p_N)$ y beneficio $(b_1, \dots, b_N)$

- Resultado: $(x_1, \ldots, x_N)$, donde $0 \leq x_i \leq 1$, $1 \leq i \leq N$
- Maximizar el beneficio $\sum_{i=1}^{N} b_i x_i$ con la restricción de que los objetos quepan en la mochila $\sum_{i=1}^{N} p_i x_i \leq M$

Ejemplo: $M = 20$, $p = (18, 15, 10)$, $b = (25, 24, 15)$. Soluciones factibles:

	solución	Peso total	Beneficio total
(a)	(1/2, 1/3, 1/4)	16.50	24.25
(b)	(1, 2/15, 0)	20.00	28.20
(c)	(0, 2/3, 1)	20.00	31.00
(d)	(0, 1, 1/2)	20.00	31.50

La solución óptima debe llenar la mochila al completo.

Criterio de selección de los objetos:

- El de mayor beneficio (b)
- El de menor peso (c)
- El de mayor relación beneficio/peso (d)

```
Mochila-fraccional($p, b, M, N$) {p es el vector de pesos, b de beneficios y M es el tamaño de la
  mochila. N es el número de objetos}
  $C := \{1, 2, \ldots, N\}$
  for $i := 1$ to $N$ do
    $solucion[i] := 0$
  end for
  while $(C \neq \emptyset) \wedge (M > 0)$ do
    /*$i$ :=elemento de $C$ con máximo beneficio $b_i$ {Orden decreciente $b_i \leftarrow$ subóptimo}*/
    /*$i$ :=elemento de $C$ con mínimo peso $p_i$ {Orden creciente $p_i \leftarrow$ subóptimo}*/
    $i :=$ elemento de $C$ con máxima relación $b_i/p_i$ {Orden decreciente $b_i/p_i \leftarrow$ óptimo}
    $C := C - \{i\}$
    if $p[i] \leq M$ then
      $solucion[i] := 1$; $M := M - p[i]$
    else
      $solucion[i] := M/p[i]$; $M := 0$
    end if
  end while
  return $solucion$
```

Figura 4.2: Algoritmo Mochila-fraccional.

Ejercicio. Corrección Mochila-fraccional. Demostrar que si los objetos se seleccionan por orden decreciente de beneficio/peso, el algoritmo Mochila-fraccional encuentra una solución óptima. (Consulta [BB97]: Teorema 6.5.1, pág. 230.)

Ejemplo: $p = (30, 18, 15, 10)$**,** $b = (25, 13, 12, 10)$ **y** $M = 50$

(1) Selección: Objeto de mayor beneficio b_i

Step	M	C	*solucion*			
	50	$\{1,2,3,4\}$	0	0	0	0
1	20	$\{2,3,4\}$	1	0	0	0
2	2	$\{3,4\}$	1	1	0	0
3	0	$\{4\}$	1	1	2/15	0

Beneficio $= 25 + 13 + 2/15 \cdot 12 = 39.6$

(2) Selección: Objeto de menor peso p_i

Step	M	C	*solucion*			
	50	$\{1,2,3,4\}$	0	0	0	0
1	40	$\{1,2,3\}$	0	0	0	1
2	25	$\{1,2\}$	0	0	1	1
3	7	$\{1\}$	0	1	1	1
4	0	$\emptyset$	7/30	1	1	1

Beneficio $= 7/30 \cdot 25 + 13 + 12 + 10 = 40.83$

(3) Selección: Objeto con mayor beneficio unitario $b_i/p_i = (5/30, 13/18, 12/15, 10/10)$

Step	M	C	*solucion*			
	50	$\{1,2,3,4\}$	0	0	0	0
1	40	$\{1,2,3\}$	0	0	0	1
2	10	$\{2,3\}$	1	0	0	1
3	0	$\{2\}$	1	0	10/15	1

Beneficio $= 25 + 10/15 \cdot 12 + 10 = \boxed{42.99}$

Coste del algoritmo Mochila-fraccional:

Versión 0: El proceso de selección requiere un coste $O(N)$ y por lo tanto el coste del algoritmo sería $O(N^2)$.

Versión 1: Si los objetos se ordenan previamente de acuerdo a la relación b/p (lo que tiene un coste $O(N \log N)$), el coste de la selección es $O(1)$ y el coste del algoritmo completo $O(N \log N)$.

Versión 2: Se puede utilizar un MaxHeap para representar el conjunto de candidatos, el coste es $O(N \log N)$.

¿En qué ocasiones es mejor la última opción?

Ejercicio. Mochila-fraccional. Escribe una versión del algoritmo Mochila-fraccional que utilice un MaxHeap para representar el conjunto de candidatos.

4.4 Ejercicios

Ejercicio. Ficheros en disco. Se tiene un disco de tamaño T y N ficheros que se desean cargar en el disco. El tamaño total de los ficheros puede ser mayor que T. Se desearía un algoritmo voraz que decidiera qué ficheros almacenar en el disco, suponiendo que:

- Se dispone de un vector $t[1..N]$ con los tamaños de los N ficheros.
- La solución se da como un vector $s[1..N]$, en el que $s[i] = 1$ indica que el fichero i-ésimo se almacena en disco, y $s[i] = 0$ en caso contrario.
- Se desea maximizar el coficiente de ocupación del disco definido como:

$$\big(s[1]t[1] + s[2]t[2] + \ldots + s[N]t[N]\big)/T$$

Para ello, se propone el siguiente algoritmo: "Ordenar los ficheros por orden *decreciente* de tamaño. Suponiendo los ficheros numerados de acuerdo con dicho orden, desde el primero hasta el último, se van cargando mientras quede espacio en el disco. Si un determinado fichero no cabe, se descarta y se pasa al siguiente."
Se pide:

a) Discutir el coste temporal del algoritmo propuesto.
b) Dar un contraejemplo para demostrar que este algoritmo no siempre obtiene la solución óptima.

Ejercicio. Almacenamiento óptimo en cinta. Dados una cinta de longitud L y n ficheros $f_1, f_2, \ldots, f_n$ de longitudes $l_1, l_2, \ldots, l_n$, ¿cuál es la secuencia de almacenamiento que minimiza el tiempo medio de acceso? Asumiremos:

- La cinta siempre está al principio para cualquier acceso
- Si el orden de almacenamiento es $i_1, i_2, \ldots, i_n$, el tiempo necesario para acceder al programa i_j será $t_j = \sum_{k=1}^{j} l_{ik}$
- Si la probabilidad de acceder a cualquier programa es la misma, el tiempo medio de acceso es $TMA = \frac{1}{n} \sum_{j=1}^{n} t_j$

Capítulo 5

Algoritmos de grafos

Objetivos

- Representación de grafos: matriz de adyacencia y listas de adyacencia (tema 2).
- Recorridos de grafos en amplitud y en profundidad. Aplicaciones recorrido en profundidad: orden topológico en un grafo dirigido acíclico (tema 2).
- Árbol de expansión de coste mínimo: algoritmo de Kruskal y el algoritmo de Prim (algoritmos voraces).
- Problema de la búsqueda del camino mínimo en un grafo: Dijkstra.

Contenidos

1 El problema del árbol de recubrimiento de coste mínimo.
Algoritmos de Kruskal y Prim
2 El problema del camino mínimo. Algoritmo de Dijkstra

Bibliografía

- *Introduction to Algorithms*, de Cormen, Leiserson y Rivest (capítulos 24 y 25)

5.1 Algoritmos sobre grafos

- Representación de grafos: matriz de adyacencia y listas de adyacencia (tema 2).
- Recorridos de grafos en amplitud y en profundidad. Aplicaciones recorrido en profundidad: orden topológico en un grafo dirigido acíclico (tema 2).
- Árbol de expansión de coste mínimo: algoritmo de Kruskal y el algoritmo de Prim (algoritmos voraces).
- Problema de la búsqueda del camino mínimo en un grafo:
  - Búsqueda del camino más corto en un grafo (dirigido y ponderado) desde un vértice origen al resto de vértices del grafo o Camino más corto hasta un vértice destino desde el resto de vértices del grafo.
    - Aristas de peso positivo: algoritmo de Dijkstra.
    - Si el grafo tiene aristas de peso negativo, pero no tiene ciclos de peso negativo: algoritmo de Bellman-Ford.
    - ¡OJO! El algoritmo de recorrido en profundidad (DFS) es un algoritmo que encuentra el camino más corto entre un vértice dado y el resto de vértices de un grafo (dirigido o no dirigido) no ponderado (esto es, grafos en los que cada arista se considera con peso 1). El coste de este algoritmo es $O(|V| + |A|)$ con una representación con listas de adyacencia y $O(|V|^2)$ con matriz de adyacencia.
  - Si el grafo es dirigido y acíclico: Camino más corto entre dos vértices del grafo (más eficiente que la aplicación de los algoritmos anteriores).
  - Problema del camino más corto entre todo par de vértices: algoritmo de Floyd-Warshall (más eficiente que aplicación repetida de los algoritmos anteriores). Asume representación del grafo en forma de matriz de adyacencia y se permiten aristas de peso negativo, aunque no ciclos de peso negativo. El coste de este algoritmo es del orden $O(|V|^3)$.

5.2 Árbol de recubrimiento de coste mínimo

Árbol de recubrimiento de un grafo no dirigido $G = (V, A)$: es un árbol libre $T = (V', A')$ tal que $V' = V$ y $A' \subseteq A$.

Problema: Dado un grafo conexo ponderado no dirigido $G = (V, A, p)$, encontrar un árbol de recubrimiento de G, $T = (V, A')$, tal que la suma de los pesos de las $|V| - 1$ aristas de T sea mínimo.

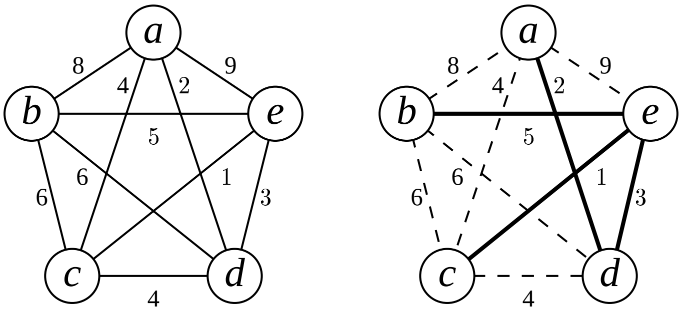


Figura 5.1: Ejemplo de árbol de recubrimiento de coste mínimo.

Ejem. aplicación: Establecer una red eléctrica entre ciertas ciudades de forma que el coste de la red sea mínimo.

Solución con algoritmos voraces (que encuentran la solución óptima): algoritmos de Kruskal y Prim. Kruskal: similar al algoritmo de de búsqueda de componentes conexas. Prim: similar al algoritmo de Dijkstra.

5.2.1 Algoritmo de Kruskal

Algoritmo de Kruskal:

- El grafo parcial formado por los nodos de G y los arcos de T consta de varias componentes conexas.
- Los elementos de T que se incluyen en una componente conexa forman un AEM para los nodos de esa componente.
- Al final del algoritmo sólo queda una componente conexa, así que T es el AEM sobre G.
- Para construir componentes conexas cada vez más grandes se seleccionan los arcos por orden creciente de pesos:
  - Si el arco seleccionado une dos componentes distintas se incluye en T (las dos componentes conexas forman ahora una sola).
  - En caso contrario se rechaza porque formaría un ciclo.

Idea voraz: construir incrementalmente un *bosque* (de recubrimiento), seleccionando en cada paso una arista $(u, v) \in A$ tal que:

- *No* se cree ningún ciclo.
- Produzca el menor incremento de peso posible.

El árbol de recubrimiento se obtiene cuando se han seleccionado exactamente $|V| - 1$ aristas válidas.

```
Kruskal_0($V, A, p$)
  $A' := \emptyset$
  while $|A'| < |V| - 1$ do
    $(u, v) := \text{argmin}_{(x,y) \in A - A'}\, p(x, y)$
    if $(u, v)$ no crea ciclo en $A'$ then
      $A' := A' \cup \{(u, v)\}$
    end if
  end while
  return $A'$
```

Problema: ¿Cómo verificar eficientemente la condición de *"no crear ciclo"*?
Solución: Mantener una colección de subconjuntos (disjuntos) con los vértices de cada árbol del bosque: *Una arista* (u, v) *no creará ciclo si* u *y* v *están en distintos subconjuntos* → componentes conexas → estructurar el conjunto de aristas seleccionadas como un MFset de vértices.

Problema: ¿Cómo seleccionar eficientemente la arista de menor peso en cada iteración?
Solución: Manteniendo las aristas en un MinHeap organizadas según su peso.

Análisis del coste del algoritmo Kruskal: El coste es $O(|A| \log |A|)$

construir $\lvert V \rvert$ MFsets	$O(\lvert V \rvert)$
construir MinHeap	$+O(\lvert A \rvert)$
$O(\lvert A \rvert)$ borrados en MinHeap	$+O(\lvert A \rvert \log \lvert A \rvert)$
$O(\lvert A \rvert)$ operaciones MFset	$+O(\lvert A \rvert)$

En general, el coste es bastante inferior a la cota $O(|A| \log |A|)$: Si m es el número de iteraciones del bucle **while**, típicamente $m \approx |V|$ y si $|V| \ll |A| \leq |V|^2$, en la práctica, el coste está más cercano a $|A| + |V| \log |V|$.

```
Kruskal($V, A, p$)
   $A' := \emptyset$
   for each $v \in V$ do
      MF-Build-Set($v$)
   end for
   $H$ := Build-Heap($A, p$)
   while $|A'| < |V| - 1$ do
      $(u, v)$ := Heap-Extract-Min($H$)
      if MF-Find($u$) $\neq$ MF-Find($v$) then
      {si $(u, v)$ no crea ciclo en $A'$}
         $A' := A' \cup \{(u, v)\}$
         MF-Merge($u, v$)
      end if
   end while
   return $A'$
```

Figura 5.2: Algoritmo de Kruskal.

5.2.2 Algoritmo de Prim

Idea voraz: Empezando por cualquier vértice, construir incrementalmente un árbol de recubrimiento, seleccionando en cada paso una arista $(u, v) \in A$ tal que:

- Si se añade (u, v) al conjunto de aristas A' obtenido hasta el momento *no* se cree ningún ciclo.
- Produzca el menor incremento de peso posible.
- Los arcos de T (Árbol de extensión parcial) forman un único árbol.
- El árbol toma como raíz un vértice cualquiera del grafo y a partir de éste se extiende por todos los vértices de G.
- En cada paso se selecciona el arco de menor peso que une un vértice presente en T con uno que no lo esté.

```
Prim_0($V, A, p$)
    $A' := \emptyset$
    $u :=$ elemento arbitrario de $V$; $S = \{u\}$
    while $S \neq V$ do
        $(u, v) := \mathrm{argmin}_{(x,y) \in S \times (V-S)} \, p(x, y)$
        $S := S \cup \{v\}$
        $A' := A' \cup \{(u, v)\}$
    end while
    return $A'$
```

```
$(u, v) := \mathrm{argmin}_{(x,y) \in S \times (V-S)} \, p(x, y) \equiv$
    $m := +\infty$
    for $x \in S$ do
        for $y \in V - S$ do
            if $p(x, y) < m$ then
                $m := p(x, y)$
                $(u, v) := (x, y)$
            end if
        end for
    end for
    Coste: $O(|V|^2)$
```

¡¡¡El coste de este algoritmo es $O(|V|^3)$!!!

Implementaciones y Análisis

- Bucle principal: $|V| - 1$ iteraciones
- Búsqueda del arco factible del menor peso:
  - Proporcional a $|A|$
  - Proporcional a $|V|$
- Mejora del algoritmo:
  - Guardar para cada nodo de $V - B$ cuál es el nodo de B más próximo
  - La actualización de los nodos más próximos consiste en recorrer $V - B$ y, para cada nodo, comprobar si el añadido a B está más cerca que el de la iteración anterior

Problema: El coste cúbico del algoritmo básico de Prim está dominado por el coste cuadrático de la operación de selección (argmin), que debe considerar todos los pares de vértices de S (ya seleccionados) y $V - S$ (por seleccionar).

Solución: Esta operación puede implementarse con un coste lineal mediante la idea de Prim:

Mantener para cada vértice no seleccionado $v \in V - S$ *el identificador de un vértice ya seleccionado* $u \in S$ *más "próximo" a él; es decir,*

$$\forall v \in V - S, \qquad P[v] = u = \underset{x \in S}{argmin}\, p(x, v)$$

Estos identificadores se pueden mantener en un vector P. Como en cada iteración sólo se añade un nuevo vértice a S, P puede actualizarse en $|V - S| = O(|V|)$ pasos.

```
Prim($V, A, p$)
  $A' := \emptyset$
  $u :=$ elemento arbitrario de $V$; $S := \{u\}$
  for $v \in V - S$ do
    $P[v] := u$
  end for
  while $S \neq V$ do
    $v := \text{argmin}_{y \in V-S}\, p(P[y], y)$; $u := P[v]$
    $S := S \cup \{v\}$
    $A' := A' \cup \{(u, v)\}$
    $x := v$
    for $y \in V - S$ do
      if $p(x, y) < p(P[y], y)$ then
        $P[y] := x$
      end if
    end for
  end while
  return $A'$
```

Figura 5.3: Algoritmo de Prim. Coste $O(|V|^2)$. Si se utiliza un MinHeap para representar $V - S$, coste $O(|A| \log |V|)$.

5.2.3 Comparación entre Prim y Kruskal

- Algoritmo de Prim:
  - Versión sencilla de coste $O(|V|^2)$: Grafos densos
  - Listas de adyacencias y Heap $O(|A| \log |V|)$
  - Listas de adyacencias y Heap de Fibonacci $O(|A| + |V| \log |V|)$
- Algoritmo de Kruskal:

- Con MFset: $O(|A| \log |A|)$

	PRIM	KRUSKAL
Grafo Denso ($|A| \approx |V|^2$)	$O(|V|^2)$	$O(|V|^2 \log |V|)$
Grafo Disperso ($|A| \approx |V|$)	$O(|V| \log |V|)$	$O(|V| \log |V|)$

Ejercicio. Grafos. El siguiente algoritmo basado en el algoritmo de recorrido primero en profundidad permite obtener un árbol de expansión de un grafo ponderado, no dirigido y conexo:

```
ST($G$) {G es un grafo ponderado, no dirigido y conexo, cuyos nodos se
  numeran de 1 a n.}
  for $v := 1$ to $n$ do
    $flag[v] := non\_visited$
  end for
  $T := \emptyset$ {T es un conjunto de aristas.}
  RPP($G, 1, T$)
  return T

RPP($G, v, T$) {G es un grafo ponderado, no dirigido y conexo, cuyos
  nodos se numeran de 1 a n, y v es un nodo.}
  $flag[v] := visited$
  for all $w \in$ Adj($v$) do
    if $flag[w] = non\_visited$ then
      $T := T \cup \{(v, w)\}$
      RPP($G, w, T$)
    end if
  end for
```

Demostrar mediante un contraejemplo que el árbol obtenido no es necesariamente de coste mínimo.

5.3 Caminos de mínimo peso en grafos dirigidos

Dado un grafo $G = (V, A)$ dirigido y ponderado por una función $p : A \to \mathbf{R}^{\geq 0}$, se define el *peso* de un camino $v_0, v_1, \ldots, v_k$ como la suma de los pesos de sus aristas:

$$p(v_0, v_1, \ldots, v_k) = \sum_{i=1}^{k} p(v_{i-1}, v_i)$$

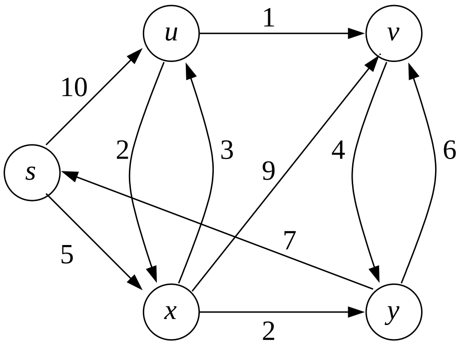


Figura 5.4: Ejemplo de grafo dirigido y ponderado.

Problema: Se quiere calcular el camino de mínimo peso (*camino más corto*) y su peso asociado desde un vértice origen (que denominaremos s) al resto de vértices del grafo.

Otras variantes: • Camino más corto desde todos los vértices a uno destino
• Camino más corto de un vértice u a un vértice v
• Camino más corto entre todos los pares de vértices
Problema: aristas con peso negativo → ciclos de peso negativo.
Dijkstra: aristas de peso positivo
(Bellman-Ford: aristas de peso negativo, sin ciclos negativos)

5.3.1 Algoritmo de Dijkstra

Resuelve el problema de los caminos más cortos con un solo origen sobre un grafo dirigido y ponderado en el que los pesos de los arcos son no negativos. Algoritmo voraz:

- Se mantiene un conjunto de vértices S cuya distancia más corta desde el origen ya es conocida. Inicialmente, $S = \{s\}$.
- En cada paso se añade a S el vértice cuya distancia desde el origen es la más corta posible pasando únicamente por vértices de S. Se utiliza un vector D para guardar la longitud del camino más corto que llega a cada vértice del grafo pasando únicamente por vértices de S.
- Al final del algoritmo $S = V$ y el vector D contiene las distancias más cortas desde el origen a cada uno de los demás vértices.

Idea voraz: empezando en el vértice origen s, construir incrementalmente caminos a los demás vértices seleccionando en cada paso un vértice v no seleccionado anteriormente tal que:

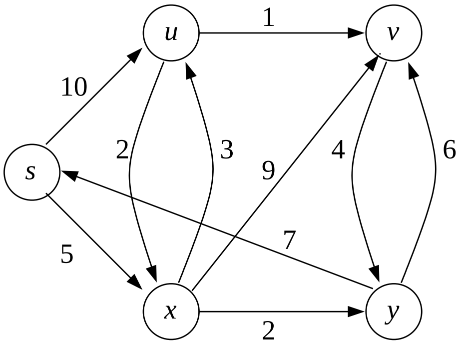
- Exista algún vértice $u \in V$ ya seleccionado previamente tal que $(u, v) \in A$.
- Al añadir (u, v) al camino que terminaba en u se produzca el menor incremento de peso posible.

El algoritmo mantiene un conjunto de vértices S cuyo peso del camino más corto desde el origen s ya es conocido. El algoritmo va seleccionando el vértice $u \in V - S$ con la mejor estimación del camino mínimo, lo inserta en S y actualiza los mejores caminos desde s a los vértices de S. Se utiliza un vector auxiliar D para mantener el menor peso de un camino de s a cada vértice v.

```
Dijkstra_0($V, A, s, p$)
  {$D[v]$ =coste mínimo de ir desde s hasta v pasando únicamente por
  vértices de $S$}
  for each $v \in V$ do
    $D[v] := \infty$
  end for
  $D[s] := 0$
  $S := \emptyset$
  while $S \neq V$ do
    $u := \text{argmin}_{(x,y) \in S \times (V-S)} D[x] + p(x, y)$ {añade a $S$ el vértice $u$
    cuya distancia desde origen es la más corta posible}
    $S := S \cup \{u\}$
    for each $v \in Ady(u)$ do
      $D[v] := \min(D[v], D[u] + p(u, v))$
    end for
  end while
  return $D$
```

```
Dijkstra_1$(V, A, s, p)$
  $\{D[v]$ =coste mínimo de ir desde $s$ hasta $v$ pasando únicamente por vértices de $S$. $Q$ es un vector$\}$
  for each $v \in V$ do
    $D[v] := \infty$
  end for
  $D[s] := 0$
  $S := \emptyset$
  $Q := V$
  while $Q \neq \emptyset$ do
    $u :=$ DeleteMin$(Q)$ $\{$mínimo según $D\}$
    $S := S \cup \{u\}$
    for each $v \in Ady(u) \mid v \in Q$ do
      $D[v] := \min(D[v], D[u] + p(u, v))$
    end for
  end while
  return $D$
```

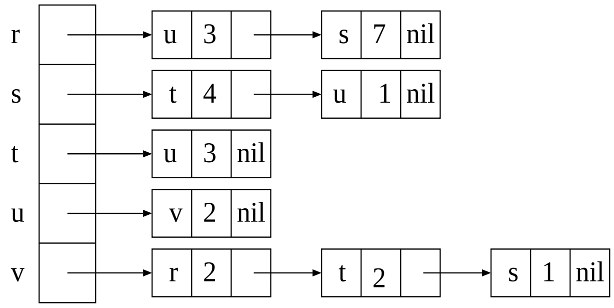
Figura 5.5: Algoritmo de Dijkstra, version preliminar.



Iteración	u	S	Q	$D[s]$	$D[u]$	$D[v]$	$D[x]$	$D[y]$
0	-	$\emptyset$	$\{s, u, v, x, y\}$	0	∞	∞	∞	∞
1	s	$\{s\}$	$\{u, v, x, y\}$	0	10	∞	5	∞
2	x	$\{s, x\}$	$\{u, v, y\}$	0	8	14	5	7
3	y	$\{s, x, y\}$	$\{u, v\}$	0	8	13	5	7
4	u	$\{s, x, y, u\}$	$\{v\}$	0	8	9	5	7
5	v	$\{s, x, y, u, v\}$	$\emptyset$	0	8	9	5	7

Figura 5.6: Traza del algoritmo de Dijkstra.

Ejercicio. Traza Dijkstra. Dado el siguiente grafo dirigido y ponderado (representado mediante listas de adyacencias):



a) Escribir la traza del algoritmo de Dijkstra para el cálculo de las distancias mínimas desde el nodo r a todos los nodos del grafo.

b) Dar los caminos asociados a las distancias mínimas calculadas en a).

Corrección del algoritmo de Dijkstra

$D[v]$ es la distancia del camino más corto desde el origen s al vértice v: No puede haber ningún camino más corto hasta v que pase por vértices que no están en S. (Demostración en CLR90: Teorema 25.10, pág. 529.)

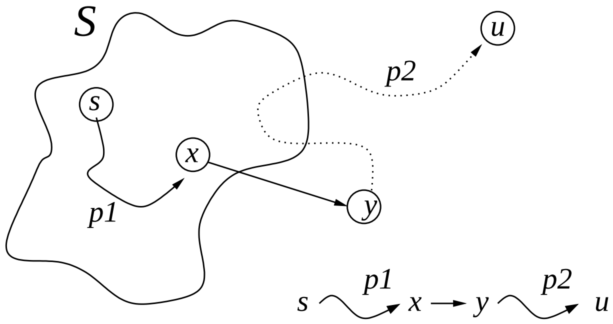


Figura 5.7: Corrección del algoritmo de Dijkstra.

Coste del algoritmo de Dijkstra

- Versión del algoritmo Dijkstra_1: G implementado como una lista de adyacencia... y Q como un *vector*:
  - Existen $|V|$ operaciones de extracción del mínimo en el vector, con un coste $O(|V|)$.

  - Cada vértice $v \in V$ se inserta en S exactamente una vez, de forma que cada arista en la lista de adyacencia se examina una única vez (bucle **for each** $v \in Ady(u)$). Debido a que el número total de aristas en G es $|A|$, existen $|A|$ iteraciones de este bucle, y cada iteración tiene un coste $O(1)$.

  Por tanto, el coste total del algoritmo es $O(|V|^2 + |A|) = O(|V|^2)$.

- Nueva versión del algoritmo Dijkstra (modificado): G implementado como una lista de adyacencia... y Q como un *MinHeap* ordenado según D (*algoritmo de Dijkstra modificado*):

```
Dijkstra($V, A, s, p$)
  {$D[v]$=coste mínimo de ir desde $s$ hasta $v$ pasando
  únicamente por vértices de $S$. $Q$ MinHeap }
  for each $v \in V$ do $D[v] := \infty$ end for
  $D[s] := 0$; $S := \emptyset$; $Q :=$ Build-Heap($V, D$)
  while $Q \neq \emptyset$ do
    $u :=$ Heap-Extract-Min($Q$)
    $S := S \cup \{u\}$
    for each $v \in Ady(u) \mid v \in Q$ do
      if $D[v] > D[u] + p(u,v)$ then
        $D[v] := D[u] + p(u,v)$
        $Q :=$ Modifica-Heap($D[v]$)
      end if
    end for
  end while
  return $D$
```

Figura 5.8: Algoritmo de Dijkstra modificado.

  - Existen $|V|$ operaciones de extracción del mínimo en un Heap, con un coste $O(\log|V|)$. Hay que añadir el tiempo de construir el Heap, $O(|V|)$.
  - En cada iteración del bucle **for each** $v \in Ady(u)$, si $D[v] > D[u] + p(u,v)$, se deberá modificar $D[v]$ y, por tanto, modificar el Heap, con un coste $O(\log|V|)$. El número de veces que se ejecuta el bucle es $O(|A|)$.

  Por tanto, el tiempo total del algoritmo será $O((|V| + |A|)\log|V|) = O(|A|\log|V|)$.

- G implementado como una lista de adyacencia... y Q se representa como un *Heap de Fibonacci*: coste total $O(|V|\log|V| + |A|)$.

Ejercicio. Dijkstra. Escribe la operación Modifica-Heap($D[v]$) añadiendo la estructura que indexa los vértices en el MinHeap.

Obtención del valor y del camino más corto

Se utiliza un vector auxiliar π de talla $|V|$, y en $\pi[u]$ se almacena el predecesor al vértice u en el camino.

```
Dijkstra_camino($V, A, s, p$)
  {Se utiliza un vector $\pi$ de talla $|V|$, y en $\pi[u]$ se almacena
  el predecesor al vértice $u$ en el camino}
  for each $v \in V$ do
    $D[v] := \infty$; $\pi[v] := NIL$
  end for
  $D[s] := 0$; $S := \emptyset$; $Q := V$
  while $Q \neq \emptyset$ do
    $u :=$ DeleteMin($Q$)
    $S := S \cup \{u\}$
    for each $v \in Ady(u) \mid v \in Q$ do
      if $D[v] > D[u] + p(u, v)$ then
        $D[v] := D[u] + p(u, v)$
        $\pi[v] := u$
      end if
    end for
  end while
  return $(D, \pi)$
```

Figura 5.9: Algoritmo de Dijkstra modificado con recuperación del camino.

Ejercicio. Caminos mínimos. Dado un grafo dirigido $G = (V, A)$ (V es el conjunto de nodos y A es el conjunto de arcos) ponderado por $p : V \times V \to \mathbb{R}^{\geq 0}$, escribe un algoritmo que calcule los costes de los caminos mínimos entre cualquier par de nodos del grafo haciendo uso únicamente del algoritmo de Dijkstra. Escribe el coste del algoritmo resultante razonándolo adecuadamente.

Bibliografía

[BB97] G. Brassard and P. Bratley. *Fundamentos de Algoritmia.* Prentice-Hall, 1997.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[HS95] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures in C++.* Computer Science Press, W. H. Freeman, 1995.

[HSR98] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer algorithms.* Computer Science Press, W. H. Freeman, 1998.

[Wei95] Mark Allen Weiss. *Estructuras de datos y algoritmos.* Addison-Wesley Iberoamericana, 1995.