Programación y Estructuras de Datos Avanzadas

Capítulo 2: Estructuras de datos avanzadas

Capítulo 2: Estructuras de datos avanzadas

- Continuación de las estructuras estudiadas en 1º
- Se utilizarán en algunos de los esquemas algorítmicos

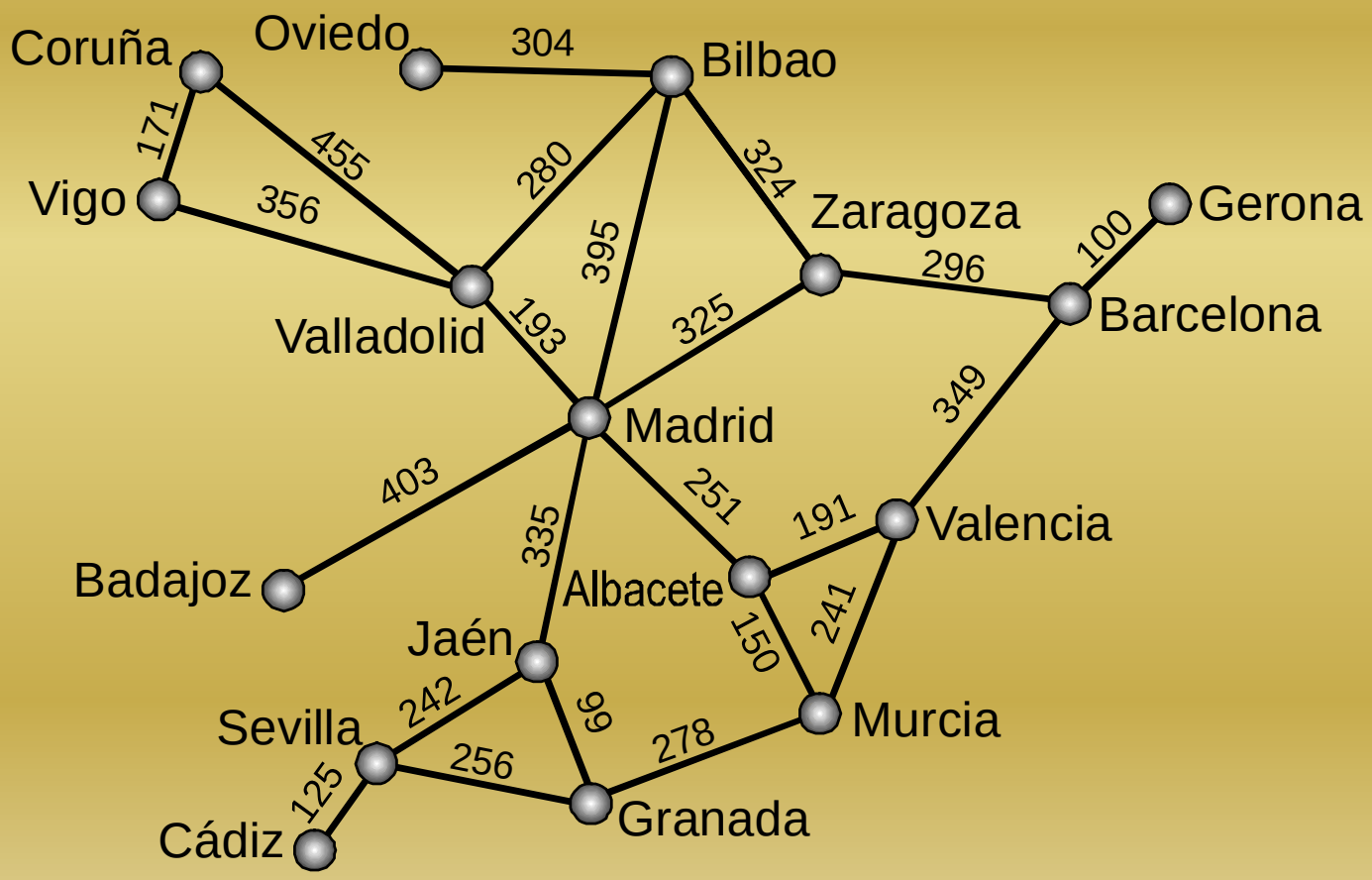
  - Grafos (algoritmos voraces, vuelta atrás y ramificación y poda)

  - Montículos (algoritmos de ramificación y poda)

  - Tablas de dispersión o tablas Hash (no se utilizarán en el resto de los temas)

2.1 Grafos

Ejemplo: grafo de carreteras entre ciudades



Problemas que se pueden plantear

- ¿Cuál es el camino más corto de Murcia a Badajoz?
- ¿Existen caminos entre todos los pares de ciudades?
- ¿Cuál es la ciudad más lejana a Barcelona?
- ¿Cuál es la ciudad más céntrica?
- ¿Cuántos caminos distintos existen de Sevilla a Zaragoza?
- ¿Cómo hacer un tour entre todas las ciudades en el menor tiempo posible?

2.1.1 Definiciones básicas

- **Un grafo G** es una tupla G= <N, A>, donde N es un conjunto finito de **vértices** o **nodos** y A es un conjunto finito de **aristas** o **arcos**.
- Cada **arista** es un par **(v, w)** o **<v,w>,** donde v, w ∈ N.
- Modelan una relación entre los objetos que intervienen.

Tipos de grafos

- **Grafo no dirigido.**
  Las aristas/líneas no están orientadas:
  (v, w) = (w, v)

N={1,2,3,4,5,6}
A={(1,2),(1,6),(2,3),(2,4),(6,4),(4,5)}

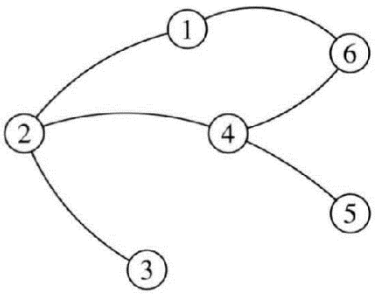


Figura 2.1: *Grafo no dirigido*

- **Grafos dirigidos (o digrafos).**
  Los arcos/flechas están orientados (pares ordenados):
  <v, w> ≠ <w, v>
  <v, w> ⇒ sentido de la flecha de v a w

Bucle o lazo

N={1,2,3,4,5,6}
A={<2,1>,<1,6>,<6,4>,<4,5>,<3,2>,<2,4>,<4,4>}

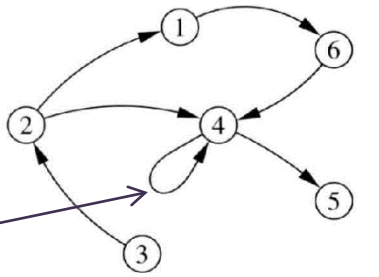


Figura 2.2: *Grafo dirigido*

- **Nodos adyacentes a un nodo v:** todos los nodos unidos a v mediante una arista.
- **En grafos dirigidos:**
  - **Nodos adyacentes a v:** todos los w con <v, w> ∈ A.
  - **Nodos adyacentes de v:** todos los u con <u, v> ∈ A (no se suele utilizar).
- **Grado de un vértice v:** número de arcos que inciden en él.
- **Para grafos dirigidos:**
  - **Grado de entrada de v:** nº de aristas con <x, v>
  - **Grado de salida de v:** nº de aristas con <v, x>

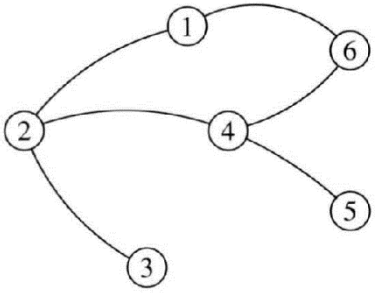


Figura 2.1: *Grafo no dirigido*

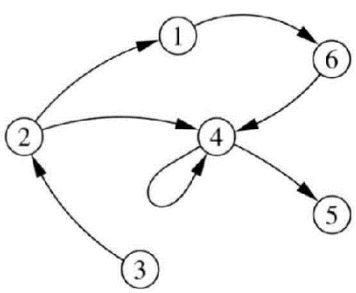


Figura 2.2: *Grafo dirigido*

Ejemplos:

Nodos adyacentes al 6:	1 y 4	4
Grado del nodo 2:	3	1 (Entrada) y 2 (Salida)

- **Camino de un vértice w_1 a w_q:** es una secuencia $w_1, w_2, ..., w_q \in N$, tal que todas las aristas $<w_1, w_2>, <w_2, w_3>, ..., <w_{q-1}, w_q> \in A$.
- **Longitud de un camino:** número de aristas del camino = nº de nodos -1.
- **Camino simple:** aquel en el que todas las aristas son distintas (los vértices también son distintos excepto el primero y el último si es un ciclo). En caso contrario es un **camino compuesto.**
- **Ciclo o circuito:** es un camino simple que empieza y termina en el mismo nodo.
- Dos nodos están **conectados** si existe un camino que los une.

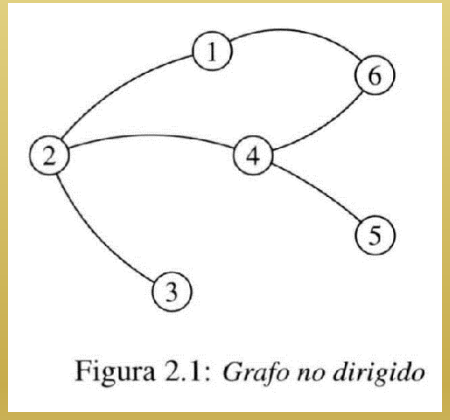


Figura 2.1: *Grafo no dirigido*

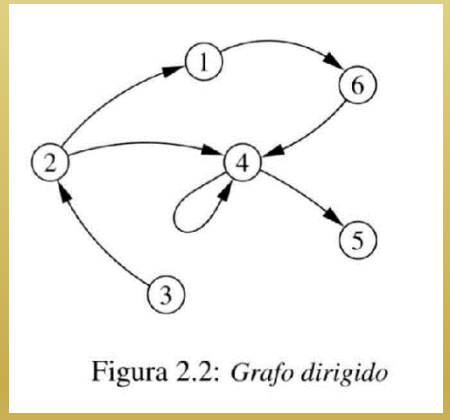
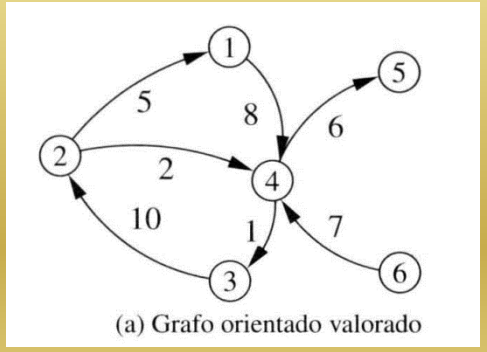


Figura 2.2: *Grafo dirigido*

Ejemplos:		
Camino simple de long. 3:	(2,1),(1,6),(6,4)	<2,1>,<1,6>,<6,4>
Camino compuesto:	(3,2),(2,4),(4,2)	---
Ciclo:	(2,1),(1,6),(6,4),(4,2)	---

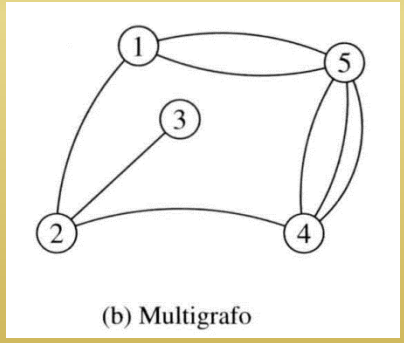
2.1.2 Tipos de grafos (según sus propiedades)



(a) Grafo orientado valorado

- Grafo **nulo** → sin nodos/vértices
- Grafo **acíclico** → sin ciclos
- Un grafo está **etiquetado** o **valorado** si cada arista tiene asociada una etiqueta o valor de cierto tipo.

  G= (N, A, W), con W: A → TipoEtiq
  (asocia un valor a cada arista)



(b) Multigrafo

- Un grafo es **simple** si entre cada par de nodos existe como mucho una arista, en caso contrario es un **multigrafo**.
- Un **subgrafo** de G=(N, A) es un grafo G'=(N', A') tal que N'⊆N y A'⊆A.
- Un grafo es **conexo** si hay un camino entre cualquier par de nodos.
- Si es un grafo dirigido, se llama **fuertemente conexo** si para cualquier par de nodos distintos (u,v) existe un camino de u a v y de v a u

- Una **componente (fuertemente) conexa** de un grafo G es un subgrafo maximal (fuertemente) conexo *Ojo porque un nodo es siempre un subgrafo (fuertemente) conexo, pero puede no ser maximal.*

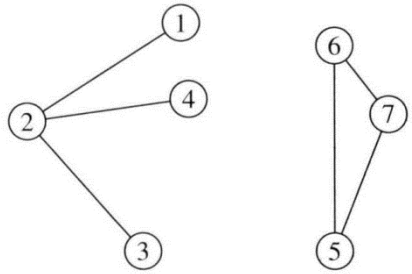


Figura 2.4: *Grafo no dirigido con dos componentes conexas*

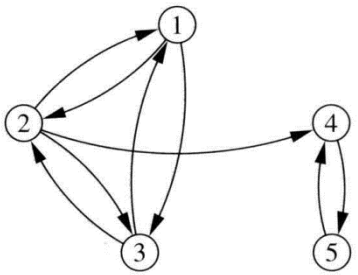


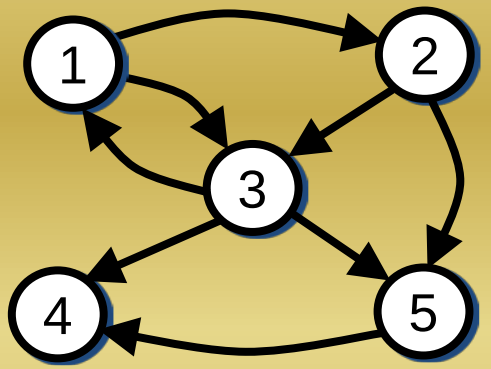
Figura 2.5: *Grafo dirigido con dos componentes fuertemente conexas*

- Un grafo será **denso** o **disperso** en función de si tiene muchas o pocas aristas.
- Número **máximo de aristas** (sin contar bucles) en un grafo de n nodos.
  - No dirigido: n(n-1)/2 (**grafo completo**)
    (si el grafo es conexo el nº mínimo de aristas es n-1)
  - Dirigido: n(n-1)
    (si el grado es fuertemente conexo el nº mínimo de aristas es 2[n-1])
- **Árbol**: grafo acíclico, conexo y no dirigido (nº de aristas = n-1)

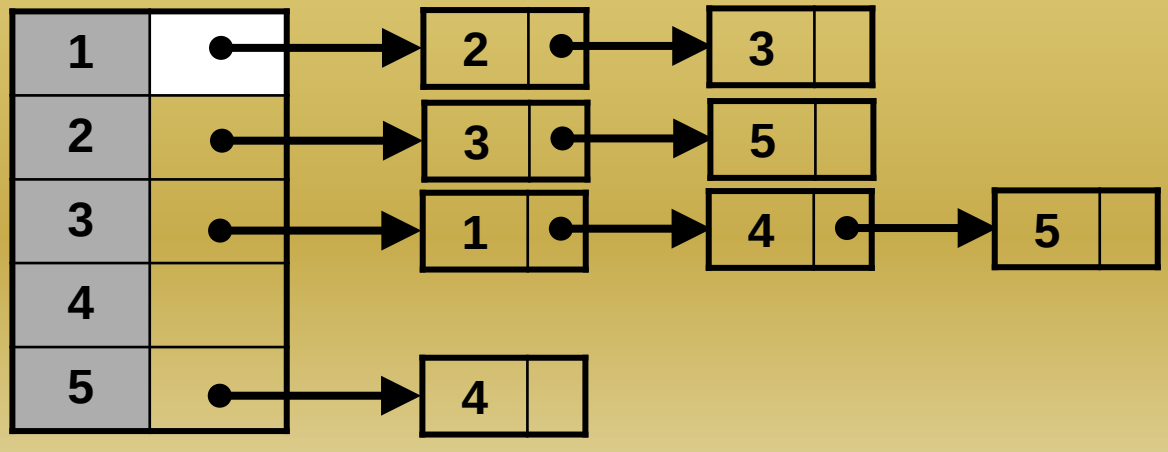
2.1.3 Representación de grafos

– Mediante **matrices de adyacencia**.

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



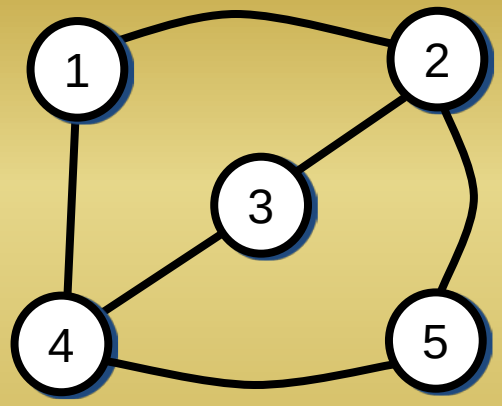
– Mediante **listas de adyacencia.**



Matrices de adyacencia

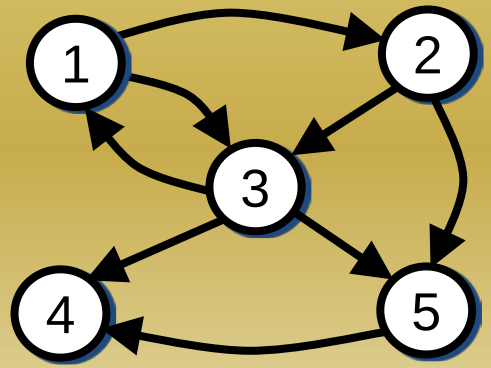
– Grafo no etiquetado/valorado: matriz MA de tamaño *n*x*n*, donde *n* es el nº de nodos y MA[i,j]=1 (o verdadero) si existe la arista (i,j) –o el arco <i,j> en un grafo dirigido– y MA[i,j]=0 (o falso) en caso contrario.

Grafo no dirigido: matriz simétrica
Puedo usar sólo la matriz triangular superior o inferior



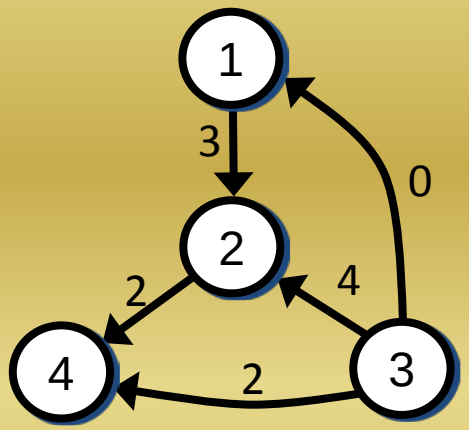
M	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

Grafo dirigido: matriz no simétrica



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

– Grafo etiquetado o valorado: MA[i,j]=valor_arista si existe la arista y MA[i,j]=valor_inválido (0, ∞,…) en caso contrario



M	1	2	3	4
1	∞	3	∞	∞
2	∞	∞	∞	2
3	0	4	∞	2
4	∞	∞	∞	∞

Grafo de *n* nodos y *a* aristas → uso de memoria:

- Requiere un espacio de memoria de $\theta(n^2)$.

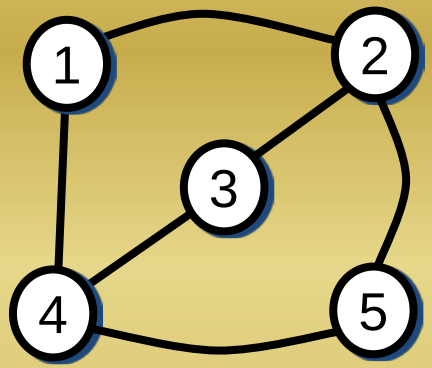
- **Ventajas:**
  - Representación y operaciones muy sencillas.
  - Eficiente para el acceso a una arista dada O(1).

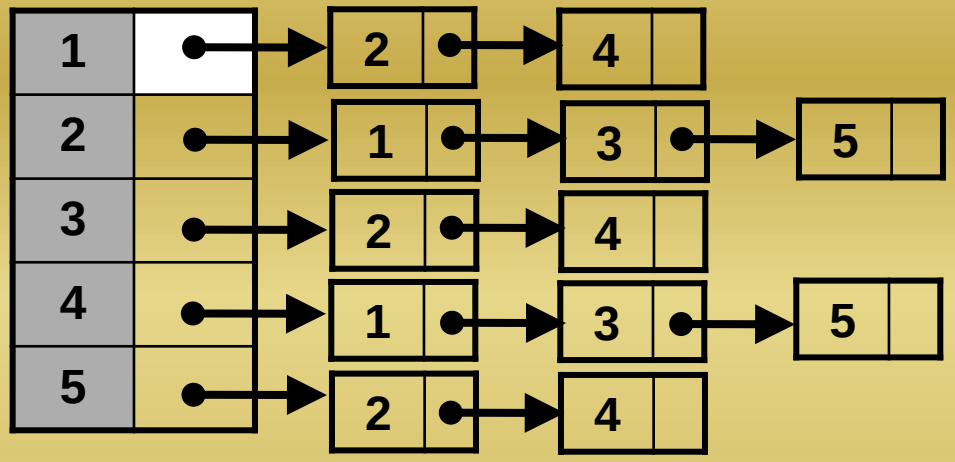
- **Desventajas:**
  - Algunas operaciones costosas:
    - ❖ Conocer los nodos adyacentes de un nodo dado O(n).
    - ❖ Determinar cuantas aristas tiene un grafo $O(n^2)$.
  - Si el grafo es disperso ($a<<n^2$) se desperdicia mucha memoria. Además si el grafo es no dirigido se desperdicia la mitad de la matriz.

Listas de adyacencia

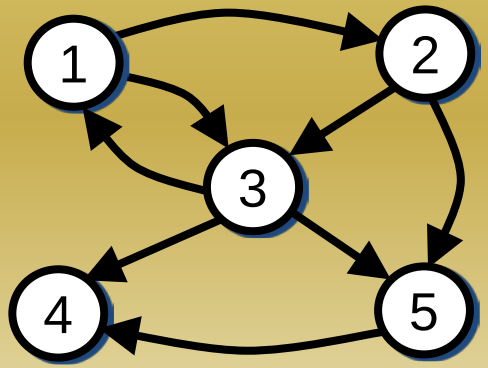
– Para un grafo de *n* nodos → **array de *n* listas**: la lista del nodo i tiene tantos elementos como nodos adyacentes tenga el nodo i.

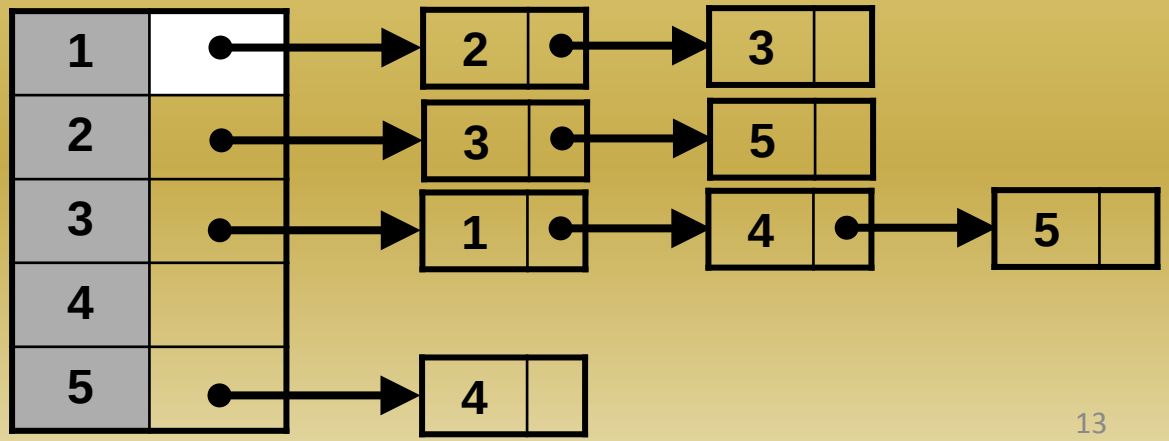
Grafo no dirigido:
se repiten aristas



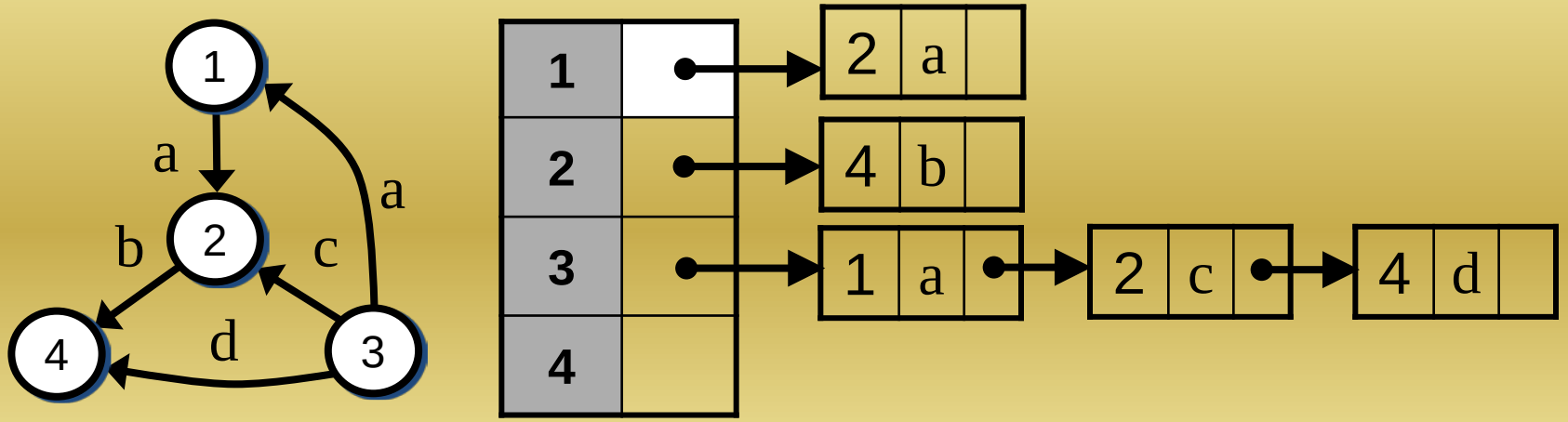


Grafo dirigido:





Grafo etiquetado/valorado: se añade un campo a cada elemento de la listas



Grafo de *n* nodos y *a* aristas → uso de memoria:

- Coste en espacio θ(*n*+*a*), *n*+*a* es el nº de punteros que se van a necesitar (*n* listas apuntando a un total de *a* aristas)

- **Ventajas:**
  - Más adecuada cuando el grafo es disperso ($a << n^2$)
  - Coste en espacio O(*n*+*a*) (normalmente < n^2)
  - Recorrido de todas las aristas de un grafo O(*n*+*a*)
  - Muy eficiente conocer los nodos adyacentes a un nodo dado O(1)

- **Desventajas:**
  - Representación más compleja.
  - Ineficiente para encontrar las aristas que llegan a un nodo.

Funciones de manipulación de grafos

- **Crear grafo:** devuelve un grafo vacío → **fun** CrearGrafo(): grafo
- **Añadir arista:** añade una arista entre los vértices *u* y v y le asigna de peso *p* → **fun** AñadirArista (u,v:vértice, p:peso, g:grafo): grafo
- **Añadir vértice:** añade el vértice *v* al grafo *g* → **fun** AñadirVertice (v:vértice, g:grafo): grafo
- **Borrar arista:** elimina la arista entre los vértices → **fun** BorrarArista (v1 ,v2:vértice, g:grafo): grafo
- **Borrar vértice:** borra el vértice *v* al grafo *g* y todas las aristas que partan o lleguen a él → **fun** BorrarVertice (v:vértice, g:grafo): grafo
- **Adyacente?:** comprueba si dos vértices son adyacentes → **fun** Adyacente? (v1,v2:vértice, g:grafo): bool
- **Adyacentes:** devuelve una lista con los vértices adyacentes a *v* → **fun** Adyacentes (v:vértice, g:grafo): lista
- **Etiqueta:** devuelve la etiqueta o peso asociado a la arista que une los vértices → **fun** Etiqueta (v1,v2:vértice, g:grafo): etiqueta

	Matriz de adyacencia	Lista de adyacencia
CrearGrafo	$O(1)$	$O(1)$
AñadirArista	$O(1)$	$O(1)$
AñadirVertice	$O(n)$	$O(1)$
BorrarArista	$O(1)$	$O(n)$
BorrarVertice	$O(n)$	$O(n+a)$
Adyacente?	$O(1)$	$O(n)$
Adyacentes	$O(n)$	$O(1)$
Etiqueta	$O(1)$	$O(n)$
Recorrido de todas las aristas	$O(n^2)$	$O(n+a)$

En general es preferible:

Grafo denso → matriz adyacencia

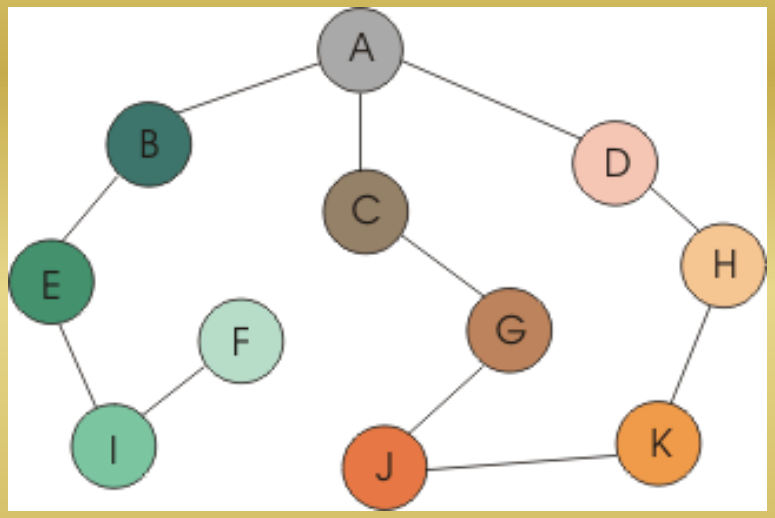
Grafo disperso → lista adyacencia

2.1.4 Recorrido de grafos

- Muchos de los problemas se pueden representar mediante grafos: la solución buscada estará en un nodo o en un camino → recorrido
- El recorrido de grafos es similar al de los árboles pero hay que tener en cuenta que los grafos:
  - pueden tener ciclos (necesito llevar la cuenta de los nodos ya visitados)
  - pueden no ser conexos (necesito poder empezar desde cualquier nodo no visitado)
- Se parte de un nodo dado y se visitan los vértices del grafo de manera ordenada y sistemática, *moviéndose* por las aristas.
- El recorrido de grafos recorridos son una **herramienta** útil para resolver muchos problemas sobre grafos.
- Estos algoritmos son válidos tanto para grafos dirigidos como no dirigidos.

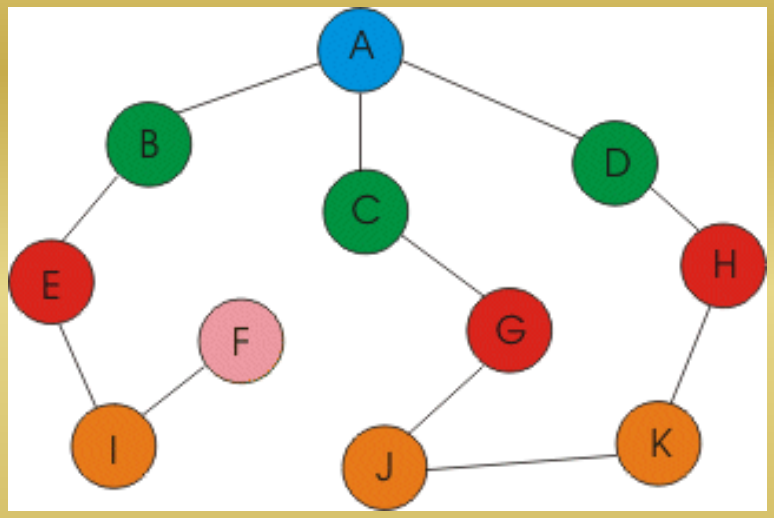
Tipos de recorridos

Recorrido en profundidad. Equivalente a un recorrido en preorden de un árbol.



Orden recorrido: A-B-E-I-F-C-G-J-K-H-D
(verdes-marrones de oscuro a claro)

Recorrido en amplitud o anchura. Equivalente a recorrer un árbol por niveles.



Orden recorrido: A-B-C-D-E-G-H-I-J-K-F
(verdes-rojos-naranjas-rosa)

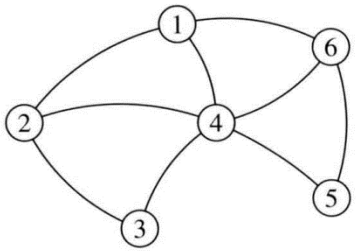
Se suponen que los nodos están almacenados en la estructura de datos por orden alfabético A-B-C-D-E-F. **Es fundamental el orden en que los nodos están almacenados en las estructuras de datos (para el cálculo de adyacentes y del nodo inicial)**. Si, por ejemplo, el nodo D estuviera antes que el C, en la búsqueda en profundidad se tomaría primero la rama del D (con lo que el último en visitarse sería el C), y en la búsqueda en anchura se exploraría antes el H que el G.

a) Recorrido en profundidad (depth-first search)

Recorrido en profundidad recursivo

- Recorre todos los nodos de una componente conexa del grafo al que pertenece el nodo *v.*

```
fun RecProfundidadRecursivo(v: nodo, visitado: Vector)
    var
        w: nodo
    fvar
    visitado[v] ← cierto
    para cada w adyacente a v hacer
        si ¬ visitado[w] entonces
            RecProfundidadRecursivo(w, visitado)
        fsi
    fpara
ffun
```



(a) Grafo no dirigido

RecProfundidadRecursivo(1)
RecProfundidadRecursivo(2)
RecProfundidadRecursivo(3)
RecProfundidadRecursivo(4)
RecProfundidadRecursivo(5)
RecProfundidadRecursivo(6)

(b) Secuencia de llamadas del recorrido en profundidad partiendo del nodo 1

Figura 2.6: *Ejemplo de recorrido en profundidad de un grafo*

Función auxiliar para invocar el recorrido

- Marca todos los nodos como no visitados
- Invoca el algoritmo recursivo con cada uno de los nodos no visitados
- No necesaria en árboles.

```
tipo Vector = matriz[0..n] de booleano
fun RecorridoProfundidad(G = ⟨N,A⟩: grafo)
    var
        visitado: Vector
        v: nodo
    fvar
    para cada v ∈ N hacer
        visitado[v] ← falso
    fpara
    para cada v ∈ N hacer
        si ¬ visitado[v] entonces
            RecProfundidadRecursivo(v, visitado)
        fsi
    fpara
ffun
```

Coste grafo *n* nodos y *a* aristas:

- Matriz de adyacencia → **O(n^2)**
- Lista de adyacencia → **O(*n*+*a*)**

Suponemos nodos ordenados numéricamente

Recorrido en profundidad iterativo

Versión iterativa utilizando una pila

```
fun RecProfundidadIterativo(v: nodo, visitado: Vector)
    var
        w: nodo
        P: TPila
    fvar
    P ← PilaVacía
    visitado[v] ← cierto
    Apilar(v,P)
    mientras ¬ vacia(P) hacer
        u ← Cima(P)
        Desapilar(P)
        para cada w adyacente a u hacer
            si ¬ visitado[w] entonces
                visitado[w] ← cierto
                Apilar(w,P)
            fsi
        fpara
    fmientras
ffun
```

Función auxiliar

```
tipo Vector = matriz[0..n] de booleano
fun RecorridoProfundidad(G = ⟨N,A⟩: grafo)
    var
        visitado: Vector
        v: nodo
    fvar
    para cada v ∈ N hacer
        visitado[v] ← falso
    fpara
    para cada v ∈ N hacer
        si ¬ visitado[v] entonces
            RecProfundidadIterativo(v, visitado)
        fsi
    fpara
ffun
```

- Coste del recorrido del grafo similar al del algoritmo recursivo.
- Se puede modificar el algoritmo para almacenar el orden en el que se visitan los nodos.

b) Recorrido en amplitud o en anchura (breadth-first search)

- En vez de una pila se utiliza una **cola**.
- El orden es equivalente al recorrido por niveles en un árbol, primero se recorren los nodos situados a una arista del inicial, luego los que están a dos, etc…

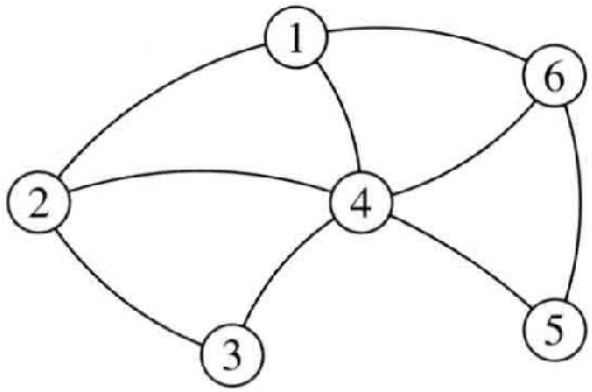
Recorrido en anchura iterativo

```
fun RecAnchura(v: nodo, visitado: Vector)
    var
        u,w: nodo
        Q: TCola
    fvar
     Q ← ColaVacía
     visitado[v] ← cierto
     Encolar(v,Q)
     mientras ¬ vacia(Q) hacer
        u ← Primero(Q)
        Desencolar(u,Q)
        para cada w adyacente a u hacer
            si ¬ visitado[w] entonces
                visitado[w] ← cierto
                Encolar(w,Q)
            fsi
        fpara
    fmientras
ffun
```

Función auxiliar para invocar el recorrido

```
tipo Vector = matriz[0..n] de booleano
fun RecorridoAnchura(G = ⟨N,A⟩: grafo)
    var
        visitado: Vector
        v: nodo
    fvar
     para cada v ∈ N hacer
        visitado[v] ← falso
    fpara
    para cada v ∈ N hacer
        si ¬ visitado[v] entonces
            RecAnchura(v, visitado)
        fsi
    fpara
ffun
```

- Coste similar al del recorrido en profundidad.
- Se utiliza cuando hay que hacer una exploración parcial de un grafo infinito o cuando se busca un nodo solución con profundidad mínima (p.ej. menor nº de movimientos).



(a) Grafo no dirigido

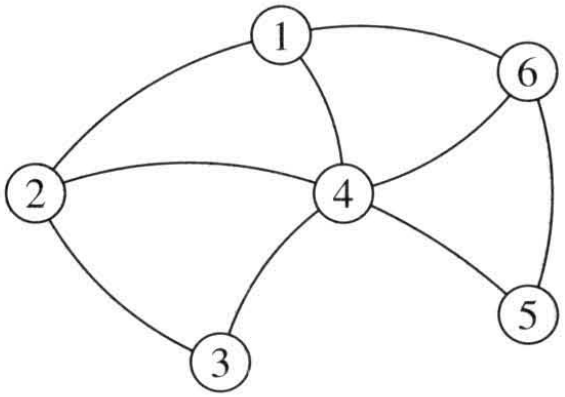
(1)
(2,4,6)
(4,6,3)
(6,3,5)
(3,5)
(5)
()

(b) Contenido de la cola del recorrido en anchura partiendo del nodo 1

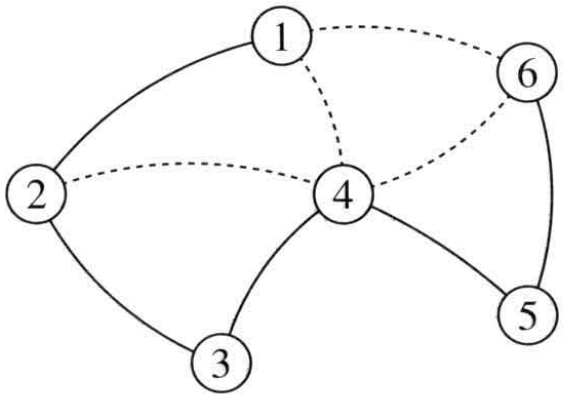
Figura 2.7: *Ejemplo de recorrido en anchura de un grafo*

2.1.5 Árboles de recubrimiento

- Los recorridos en profundidad o en anchura de un **grafo conexo** $\rightarrow$ **árbol de recubrimiento asociado** (árbol que incluye todos los nodos del grafo)
  - Las aristas de no forman parte del árbol son las que no se usan en el recorrido del grafo (porque conducían a nodos ya visitados)
  - El árbol resultante depende del orden de los nodos en la estructura (cálculo de adyacentes) así como del nodo inicial.



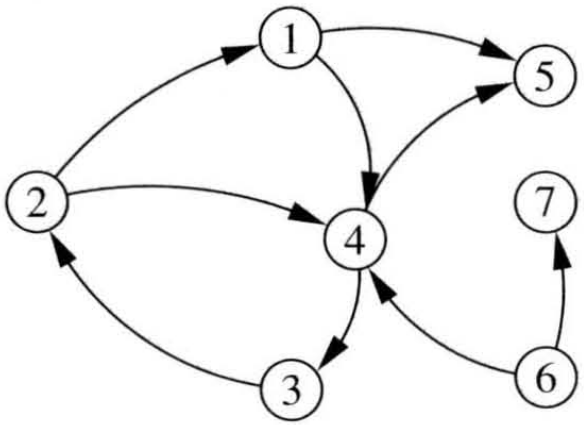
(a) Grafo no dirigido



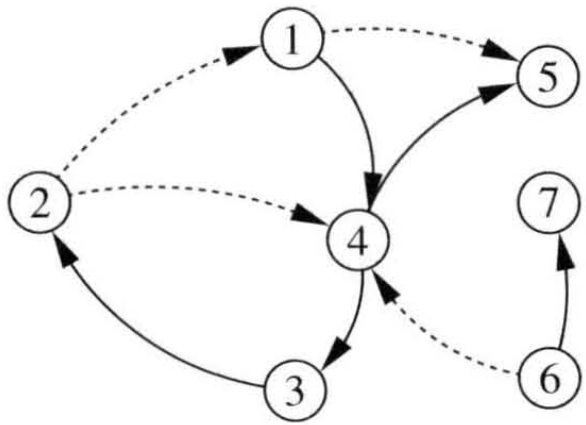
(b) Aristas del árbol de recubrimiento asociado

Figura 2.8: *Ejemplo de árbol de recubrimiento a partir de un recorrido en profundidad*

- **Grafo no conexo → bosque de recubrimiento asociado** (un árbol por cada componente conexa del grafo)



(a) Grafo dirigido no conexo



(b) Bosque de recubrimiento asociado

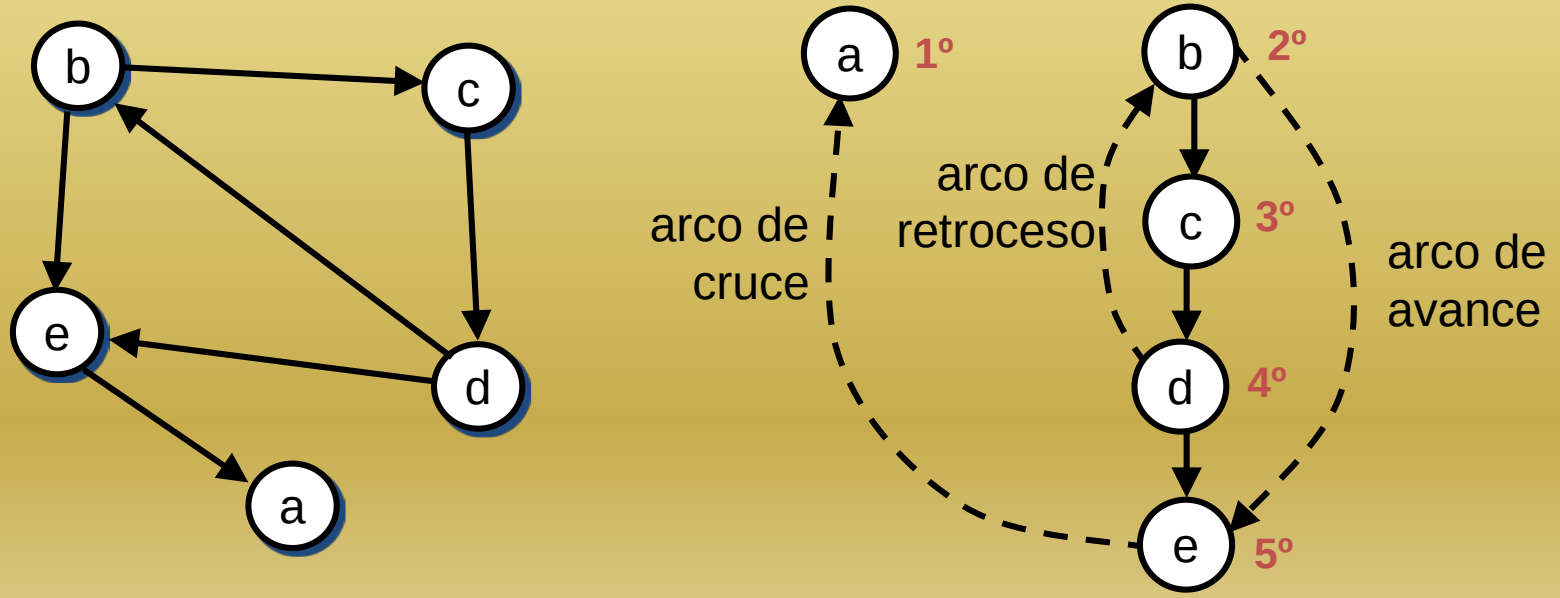
Figura 2.9: *Ejemplo de bosque de recubrimiento a partir de un recorrido en profundidad*

- Secuencia de llamadas recursivas suponiendo que los nodos están ordenados numéricamente. El bosque resultante también depende de este orden.
- En el bosque habría que sustituir las flechas por líneas.

```
RecProfundidadRecursivo(1)
    RecProfundidadRecursivo(4)
        RecProfundidadRecursivo(3)
            RecProfundidadRecursivo(2)
        RecProfundidadRecursivo(5)
RecProfundidadRecursivo(6)
    RecProfundidadRecursivo(7)
```

- En el **bosque de expansión de un grafo dirigido**, los **arcos que no se incluyen en el árbol** pueden clasificarse de 3 tipos:
  - **Arco de avance** <v,w>: w es descendiente de v en uno de los árboles del bosque.
  - **Arco de retroceso** <v,w>: v es descendiente de w en uno de los árboles del bosque.
  - **Arco de cruce** <v,w>: no se cumple ninguna de las condiciones anteriores.

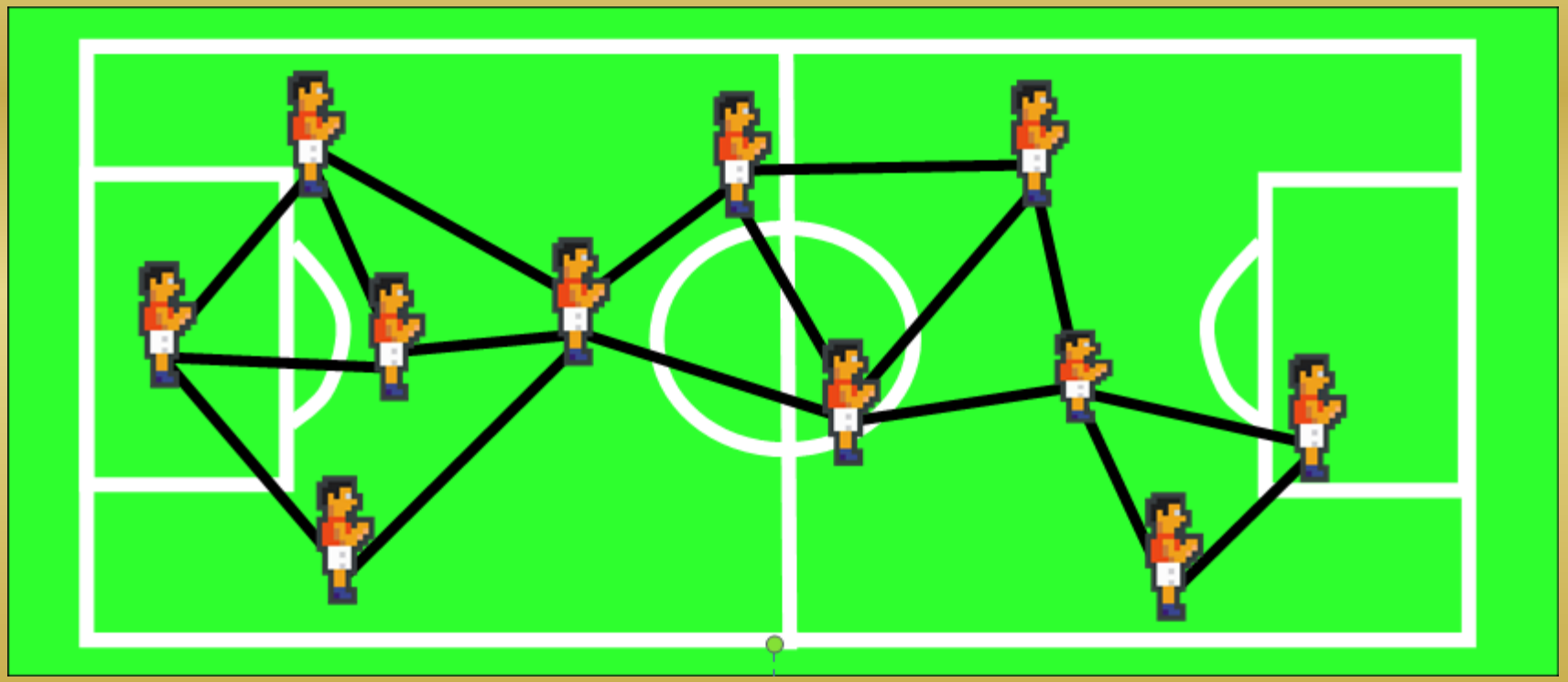
  Todos ellos conducen a **nodos ya incluidos en el bosque de recubrimiento**.



Orden de los nodos: alfabético (si empezase por el *b* obtendría un solo árbol)

2.1.6 Puntos de articulación

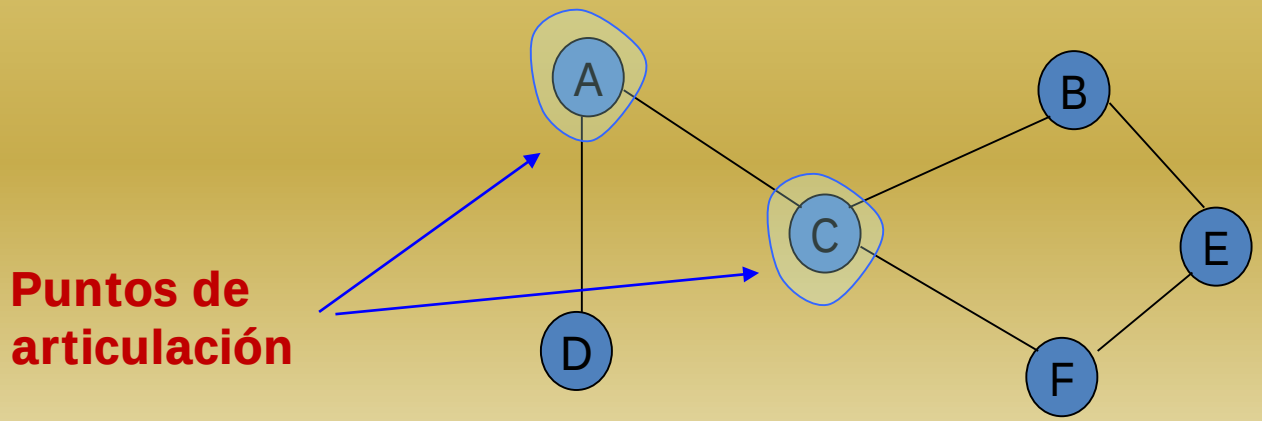
- **Ejemplo:** grafo de estrategias de pase del balón del Real Murcia.



- **¿Qué jugador, o jugadores, desconectan al equipo si los eliminamos?**
- Escribir un algoritmo que lo calcule.

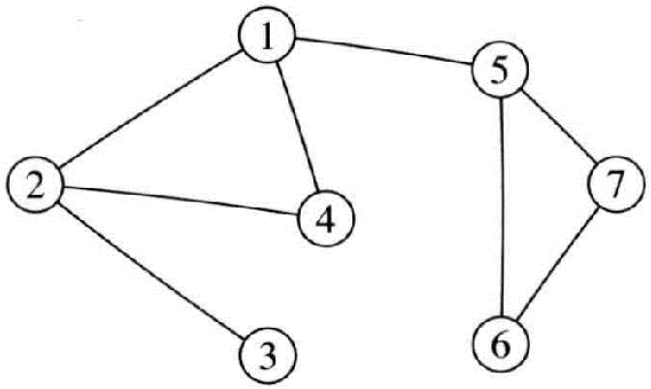
2.1.6 Puntos de articulación

- En un grafo **no dirigido conexo** existen vértices que si se eliminan
  - "Desconectan" al grafo dividiéndolo en componentes conexas
- Estos vértices "clave" → **puntos de articulación**
- La **conectividad de un grafo**
  - Es el número de nodos que se necesitan eliminar para dejar a un grafo "desconectado". Grafo de conectividad *k* se pueden eliminar *k*-1 nodos cualesquiera sin "desconectar" el grafo.
  - Ligado al concepto de redundancia (p. ej. red de comunicaciones)
- Grafo **biconexo** → no tiene puntos de articulación (conectividad ≥ 2)

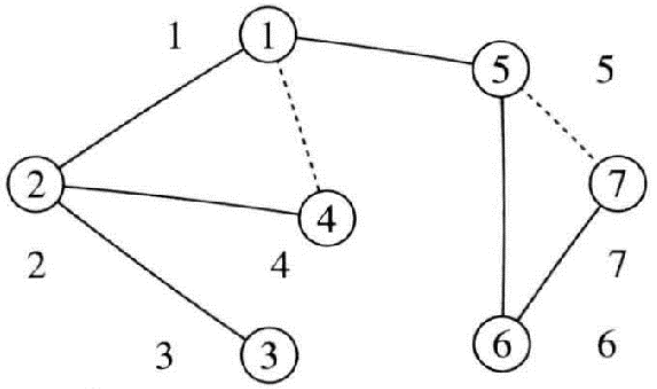


Búsqueda de los puntos de articulación de un grafo

- Basado en el recorrido en profundidad del grafo.
- Si no se encuentran $\rightarrow$ grafo biconexo.
- **Procedimiento:**
  1) Realizar recorrido en profundidad numerando, según se recorren, los nodos del árbol de recubrimiento asociado (numOrden[]).



(a) Grafo no dirigido conexo



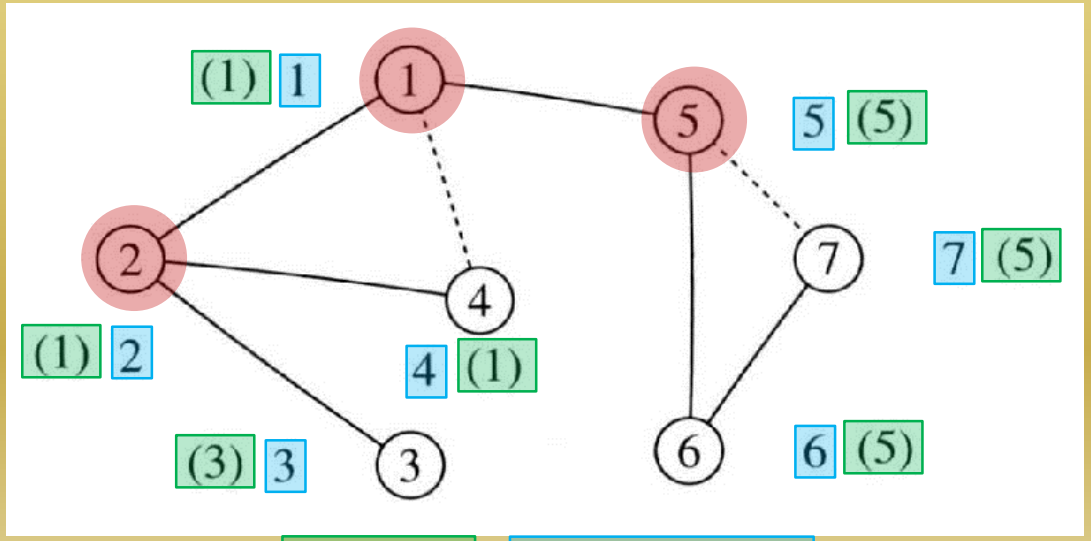
(b) Árbol de recubrimiento asociado con los nodos numerados en orden de recorrido

Figura 2.11: *Ejemplo de grafo con puntos de articulación y su árbol de recubrimiento asociado*

2) Recorremos el AR en **postorden** (recorre los subárboles de izqda. a drcha. en postorden y luego el nodo raíz) y para cada nodo *v* calcula bajo[*v*]

bajo[v] → ***valor de numOrden*** del nodo más alto al que podemos llegar desde v en el AR descendiendo por 0 o más aristas del árbol y ascendiendo como máximo por una arista que no pertenece al árbol.

$$bajo[v] = \min \begin{cases} numOrden[v] \\ numOrden[w],\ \forall w \text{ t.q. } \exists \text{ una arista de retroceso } (v,w) \in G \\ bajo[x] \text{ para cualquier hijo } x \text{ de } v \end{cases}$$



(*bajo*[*v*]) y *numOrden*[*v*]

3) **Cálculo puntos articulación**

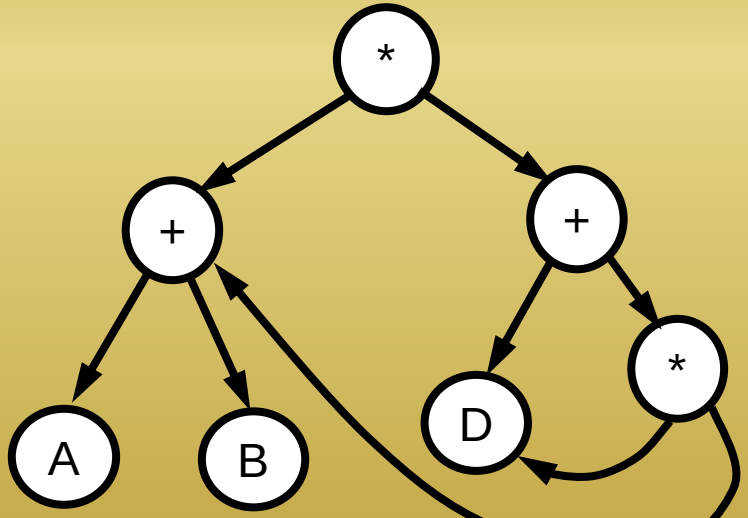
- La **raíz** del AR es un punto de articulación **si tiene más de un hijo**.

- Cualquier otro **nodo *v* *es* un punto de articulación si y sólo si tiene un hijo *w* tal que *bajo*[*w*]≥*numOrden*[*v*].**

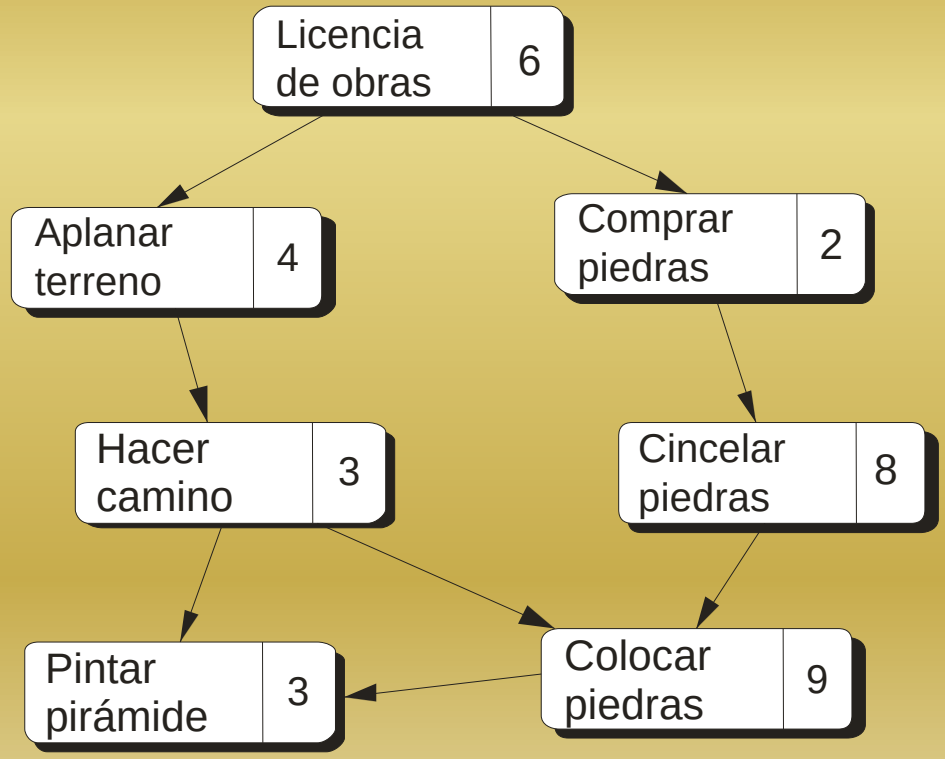
Puntos de articulación: 1,2,5

2.1.7 Ordenación topológica de un grafo dirigido acíclico

- **Definición**: un **grafo dirigido acíclico** (**GDA**) es un grafo dirigido sin ciclos
- **Ejemplos:** grafo de planificación de tareas, expresiones aritméticas (con subexpresiones comunes), grafo de prerrequisitos, etc.



(A+B)*(D+D*(A+B))



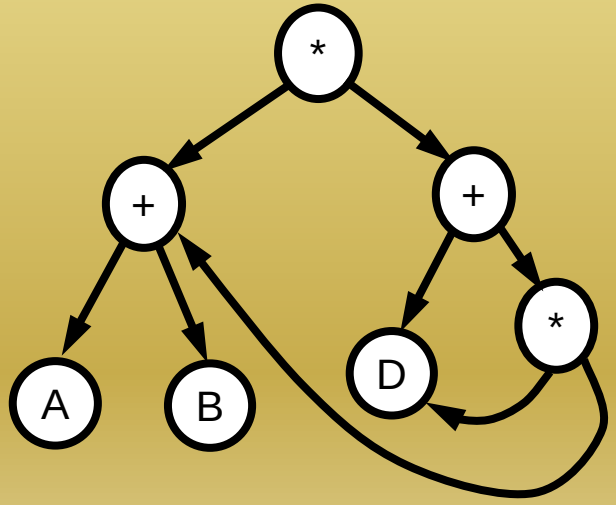
- **Recorrido en orden topológico:** es un tipo de recorrido aplicable solamente a gda.

  - Idea → un vértice sólo se visita después de haber sido visitados **todos** sus predecesores en el grafo.

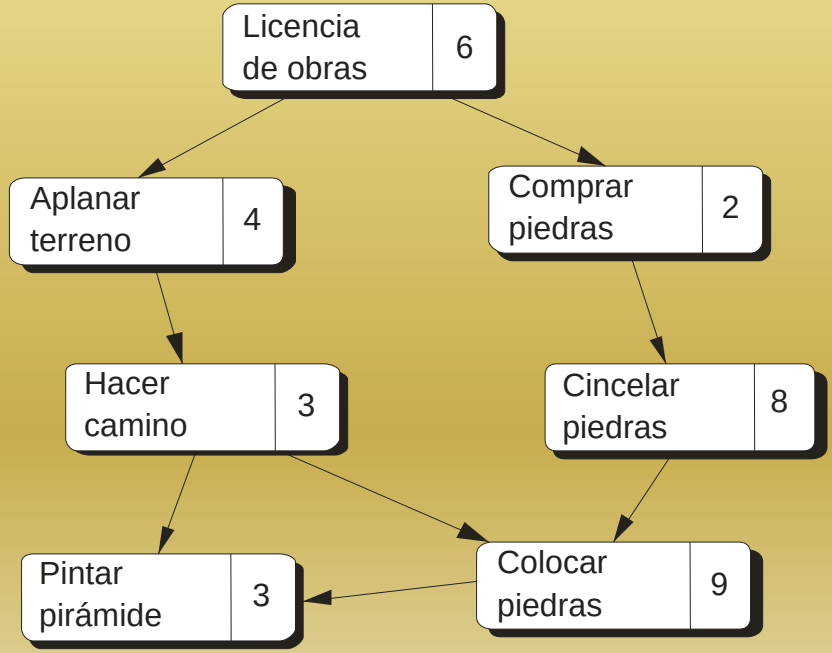
- **Numeración en orden topológico: ntop[v].** Si existe una arista <v, w> entonces **ntop**[v] < **ntop**[w].
- Puede existir más de un orden válido.
- ¿Cuál es el significado del orden topológico?

  - Grafo de tareas: es un posible orden de ejecución de las tareas, respetando las precedencias.

  - Expresión aritmética: orden para evaluar el resultado total de la expresión (de mayor a menor **ntop**).



(A+B)*(D+D*(A+B))



- **Recorrido en orden topológico → recorrido en postorden invertido**

```
fun RecProfundidadRecursivoOrdenTopologico(v: nodo, visitado: Vector)
    var
        w: nodo
    fvar
    visitado[v] ← cierto
    para cada w adyacente a v hacer
        si ¬visitado[w] entonces
                RecProfundidadRecursivoOrdenTopologico(w, visitado)
        fsi
    fpara
    escribir(v)
ffun
```

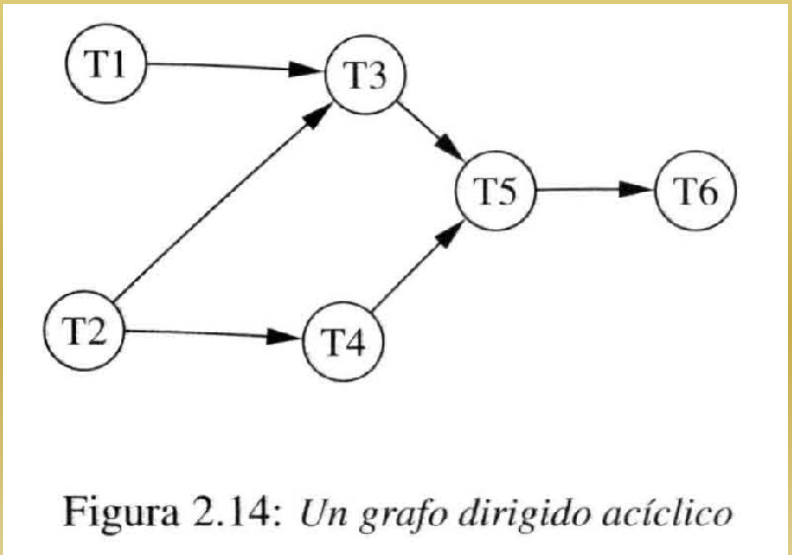


Figura 2.14: *Un grafo dirigido acíclico*

- La función anterior imprimiría los nodos en orden topológico inverso:

  T6 T5 T3 T1 T4 T2

- Invirtiendo dicha lista tendríamos el **orden topológico**.

  T2 T4 T1 T3 T5 T6

2.1.8 Camino más corto desde la raíz a cualquier nodo

- **Definición**: dado un grado *G*, la distancia más corta entre sus nodos *u* y *v* es el mínimo número de aristas en cualquier camino entre *u* y *v.*
  - Sea *G*=<*N*,*A*> y ***ARA***=<*N*,*A*'> el **árbol de recubrimiento asociado al recorrido en anchura de *G***.
  - Para cada nodo w, la longitud del camino desde la raíz hasta w en ARA coincide con la longitud del camino más corto desde la raíz hasta *w* en G.
- **Asociado al recorrido en anchura**.
- **Ejemplo:**
  - Alcanzar, en un menor nº de pasos, un determinado nº entero E partiendo de 2 y aplicando sólo 2 posibles operaciones: multiplicación por 3 y división entera entre 2.

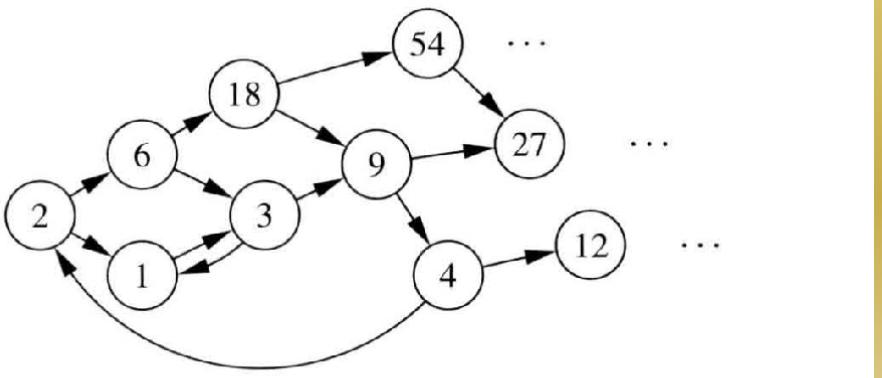


Figura 2.15: *Un grafo dirigido infinito*

- Sup. E=4, el recorrido en anchura daría:

  2÷2×3×3÷2=4
- Un recorrido en profundidad podría no alcanzar la solución nunca (aunque no pidiesen el camino más corto≡menor nº de pasos).

2.2 Montículos (Heaps)

- Definición de **montículo (binario): árbol binario esencialmente completo** cuyos nodos satisfacen la llamada **propiedad del montículo**.
  - Árbol binario esencialmente completo (los nodos internos tienen siempre 2 hijos con la excepción de un único nodo cuando el nº de elementos es **par**).
  - Cada nodo contiene un valor mayor o igual que el de sus nodos hijos (montículo de máximos), o menor o igual (montículo de mínimos/invertido) ← **propiedad del montículo**.
- **Prop. fundamental:** la cima del montículo (nodo raíz) contiene el mayor elemento (o menor en el montículo de mínimos) → elemento más prometedor en **colas de prioridad**
- **Implementación** (montículo maximal de *n* nodos) → **vector T[1..*n*]** donde:
  - El nodo raíz → T[1]. Es el mayor de los elementos. Profundidad del árbol: $\lfloor \log_2 n \rfloor$
  - Los hijos del elemento T[*i*] → T[2*i*] y T[2*i*+1]. Cumplen T[*i*]≥T[2*i*] y T[*i*]≥T[2*i*+1].
  - El padre del nodo T[*i*] → T[*i div* 2].
  - Los nodos de profundidad *k* → situados en las posiciones [2^k..2^{k+1}-1] del vector.
- Los elementos del vector no están ordenados. El objetivo de esta estructura es tener en todo momento el mayor elemento en la raíz y contar con operaciones de manipulación muy eficientes.

Ejemplo de montículo:

El vector *m*=[6,5,4,4,1,3,2] representa el montículo:

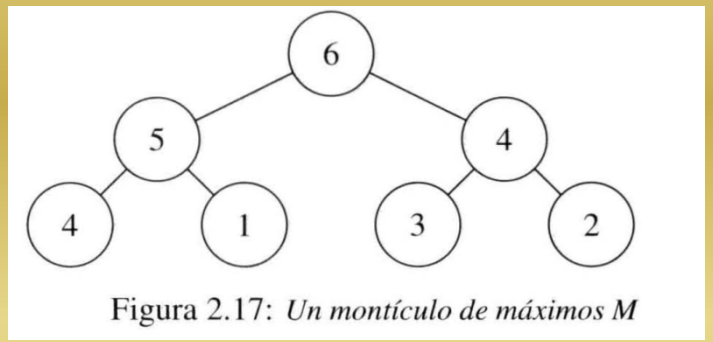


Figura 2.17: *Un montículo de máximos M*

Los elementos se disponen en un árbol binario por niveles de izquierda a derecha

Tras añadir el elemento 7 quedaría *m*=[7,6,4,5,1,3,2,4]

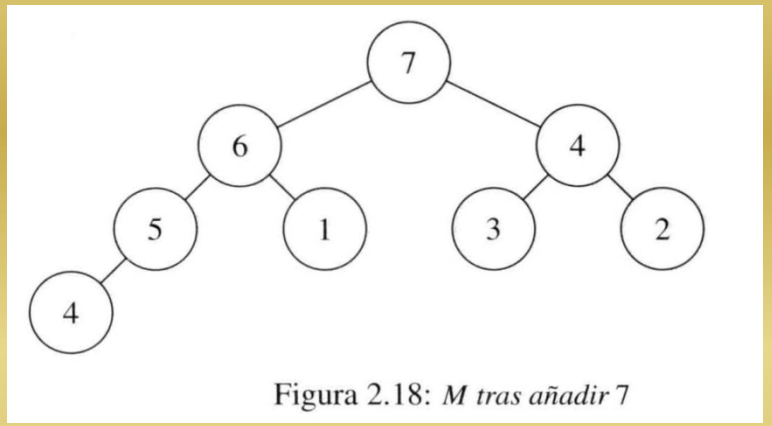


Figura 2.18: *M tras añadir 7*

- Ojo, cuando se añade o elimina un elemento hay que **restaurar la propiedad del montículo**.
- Aplicaciones:
  - Algoritmos de ordenación como Heapsort (eficiente en el caso peor)
  - Algoritmos de selección: búsqueda de mínimos, máximos, medianas…
  - Algoritmos basados en grafos: almacenando los nodos o aristas en montículos puede reducirse la complejidad del algoritmo (Algoritmo de Prim o Dijkstra).
  - Algoritmos de ramificación y poda $\rightarrow$ colas de prioridad.

2.2.1 Implementación y operaciones sobre elementos del montículo

- **Estructura de datos**: registro con tres campos:

```
registro monticulo
    T: vector [1..n] de entero;
    c: natural;
    MAX: natural;
fregistro
```

- **Operaciones:**
  - **CreaMonticuloVacio:** devuelve un montículo vacío → O(1)
  - **MonticuloVacio?:** devuelve **cierto** si el montículo está vacío → O(1)
  - **Flotar:** reubica el elemento *i-esimo* del vector en caso de que este sea mayor que el padre → O(log n)
  - **Hundir:** reubica el elemento *i-esimo* del vector en caso de que éste sea menor que alguno de sus hijos. En tal caso, intercambia su valor por el del mayor de sus hijos → O(log n)
  - **Insertar**: inserta un elemento en el montículo y lo flota hasta restaurar la propiedad de montículo → O(log n)
  - **Primero**: devuelve la cima del montículo sin modificarlo → O(1)
  - **ObtenerCima**: devuelve la cima del montículo, la elimina del mismo y recompone la propiedad de montículo O(log n)

Función *CreaMontículoVacio*

La función devuelve un montículo vacío con el contador iniciado a 0 elementos.

```
fun CreaMonticuloVacio(m: monticulo, n:entero)
    m.T ← null
    m.c ← 0
    m.MAX ← n
ffun
```

Función *MontículoVacio?*

```
fun MonticuloVacio?(m: monticulo)
    si (m.c = 0) entonces dev cierto sino dev falso
ffun
```

Función *Flotar*

La función flotar reubica el elemento *i-esimo* del vector *T* del montículo en caso de que éste sea mayor que el padre, hasta que esté correctamente situado en el montículo y se haya restablecido la *propiedad de montículo*. El proceso de flotar se utiliza para la inserción de un elemento nuevo en el montículo.

```
fun flotar(T: vector, i:natural)
    var
        padre:natural
    fvar
    padre ← i div 2
    mientras (i>1) ∧ (T[padre] < T[i]) hacer
        intercambiar (T[i],T[padre])
        i ← padre
        padre ← i div 2
    fmientras
ffun
```

Supongamos que tenemos el siguiente caso, en el que aplicamos la función *flotar* al montículo siguiente (en oscuro se señala aquel nodo que no cumple la propiedad de montículo):

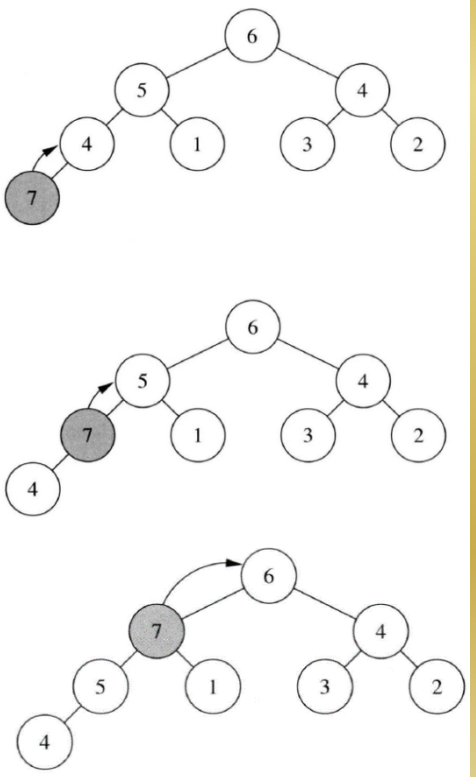


Figura 2.20: *Operación* flotar

Función *Hundir*

La función *Hundir* reubica el elemento *i-ésimo* del vector *T* del montículo *m* en caso de que este sea menor que alguno de sus hijos. En tal caso, intercambia su valor por el del mayor de sus hijos.

```
fun Hundir(T:vector, i:natural)
    var
        hi,hd,p:natural
    fvar
    repetir
        hi ← 2*i
        hd ← 2*i+1
        p ← i
    si (hd ≤ m.MAX)∧(T[hd] > T[i]) entonces
        i ← hd
        fsi
        si (hi ≤ m.MAX) ∧ (T[hi] > T[i]) entonces
            i ← hi
        fsi
        intercambiar (T[p],T[i])
    hasta p=i;
ffun
```

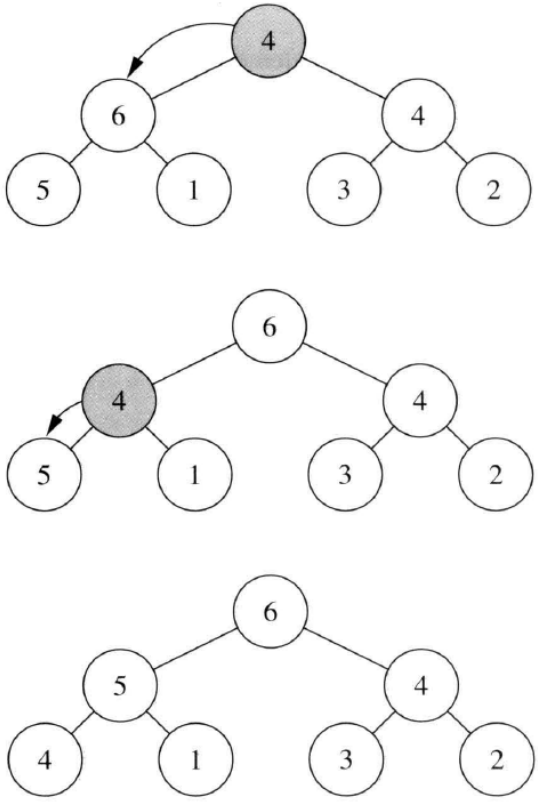


Figura 2.21: *Operación* hundir

Función *Insertar*

La manera más intuitiva de inserción de un elemento en un montículo se realiza incorporando el elemento al final del mismo y aplicando la operación *Flotar*.

```
fun Insertar(e:elemento; m: montículo): monticulo
    si m.c = m.MAX entonces
        error(MonticuloLleno)
    sino
        m.c ← m.c + 1
        m.T[m.c] ← e
        Flotar(m.T,m.c)
    fsi
ffun
```

Función *Primero*

Esta función devuelve el valor que hay en la cima del montículo sin eliminarla.

```
fun Primero(m: montículo): elemento
    si m.c = 0 entonces dev error
    sino
        dev m.T[1]
    fsi
ffun
```

Función *ObtenerCima*

El borrado de la cima del montículo devuelve el valor de esta y elimina la cima de la primera posición del montículo. Para recuperar la propiedad de montículo, se pone en su lugar el último elemento del montículo (el de más profundidad y más a la derecha en el árbol binario) y se realiza sobre el mismo la función *Hundir*.

```
fun ObtenerCima(m: montículo): elemento
    var
        e:elemento
    fvar
    si m.c ≠ 0 entonces
        e ← m.T[1]
        m.T[1] ← m.T[m.c]
        m.c ← m.c - 1
        Hundir(m.T,1)
        dev e
    fsi
ffun
```

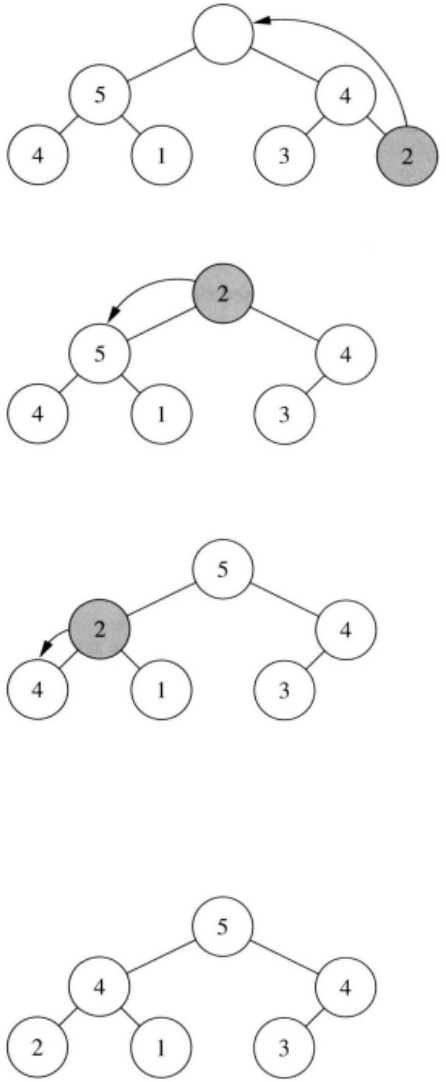


Figura 2.22: *Operación de la recuperación de la propiedad de montículo tras el borrado de la cima*

2.2.2 Eficiencia en la creación de montículos a partir de un vector

Función CreaMontículo

- Partimos de un **vector T[1..n]** y queremos crear un **montículo m[1..n]** de forma eficiente.
  - **Solución 1 → Función CreaMonticulo:** flota los [2..n] elementos del vector

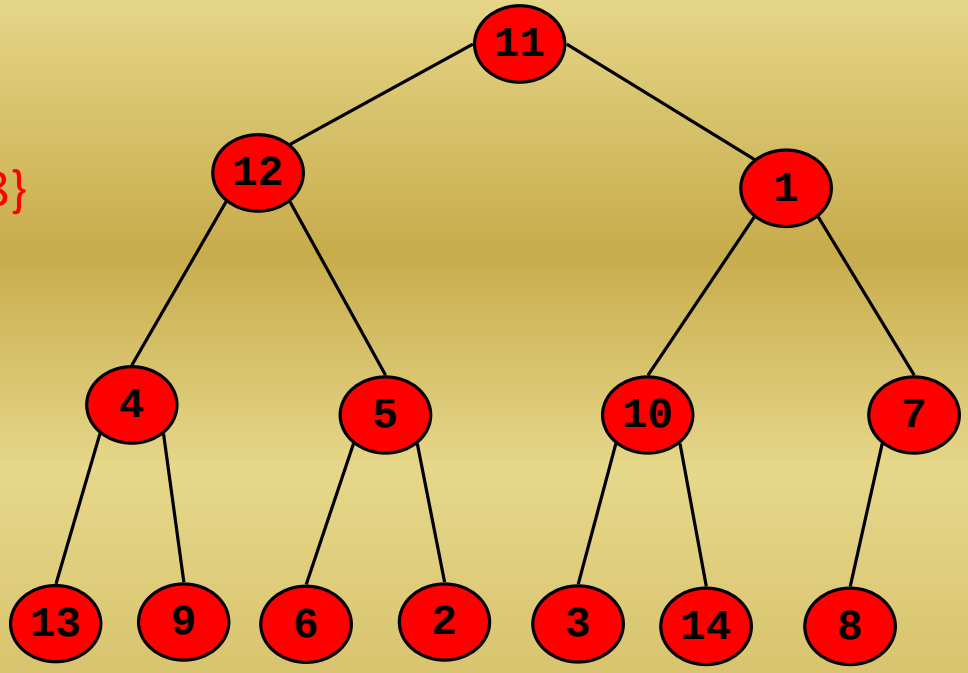
```
fun CreaMonticulo(T:vector[1..n]): monticulo
    var
        m: monticulo
    fvar
    m ← CreaMonticuloVacio();
    para i ← 2 hasta n hacer
        Flotar(T,i);
    fpara
    m.T ← T
    m.c ← n
    dev(m)
ffun
```

Coste: n-1 operaciones flotar, de coste promedio O(log n) → **O(n log n)**

(el coste de flotar depende de donde esté el elemento y cuanto haya que "flotarlo")

¿Se puede hacer mejor?

Ejemplo:
T[1..n]={11,12,1,4,5,10,7,13,9,6,2,3,14,8}



No es un montículo (no cumple la propiedad)

Idea: en vez de flotar nodos (que nos ahorramos el primero) hundimos nodos, por lo que no nos hará falta hundir los n/2 últimos elementos del vector (que ya están en el último nivel, aunque eso no significa que estén se vayan a quedar ahí)

- **Solución 2 → Función CreaMontículo:** hunde los [1..n/2] elementos del vector

```
fun CreaMonticuloLineal(T:vector[1..n]): monticulo
    var
        m:monticulo
    fvar
    m ← CreaMonticuloVacio();
    para i ← n/2 hasta 1 paso -1 hacer
        hundir(T,i);
    fpara
    m.T ← T
    m.c ← n
    dev(m)
ffun
```

Coste:

- Sup. árbol binario completo (*n* nodos, *k* niveles) $n=2^{k.}$
- Nivel *k*: $n/2=2^{k-1}$ hojas que no hay que hundir..
- Nivel *k*-1: $n/4=2^{k-1}$ nodos requerirán, como mucho, 1 iteración de hundir.
- Nivel *k*-2: $n/8=2^{k-2}$ nodos que requerirán, como mucho, 2 iteraciones de hundir.
- Nivel *i*: $n/2^{i+1}=2^{k-i-1}$ nodos requerirán, como mucho, *i* iteraciones en el bucle.

$$\sum_{i=1}^{k-1} i2^{k-i-1} = n - \log n - 1 \in O(n) \qquad n = 2^k$$

Ordenación basada en montículos: algoritmo Heapsort

- La estructura de datos Montículo → creada para algoritmo de ordenación eficiente
- Procedimiento para ordenar un vector T[1..n]:
  - Se crea un montículo (de mínimos o máximos) a partir del vector (**CreaMonticuloLineal**)
  - Se van eliminando las raíces (y restaurando de su vez la propiedad del montículo –**ObtenerCima** –) y añadiendo en orden al nuevo vector ordenado.

```
fun Heapsort(T:vector[1..n] de entero): vector[1..n] de entero
    var
        e:entero
        M:monticulo
        S: vector[1..n]
    fvar
    M ← CreaMonticuloLineal(T);
    para i ← 1 hasta n hacer
        e ← ObtenerCima(M)
        S[i] ← e
    fpara
    dev S
ffun
```

Coste:

- Creación montículo: O(n)

- ObtenerCima: O (log n), pero como se hace n veces:

→ **O(n log n)**

(constante multiplicativa peor que Mergesort y Quicksort)

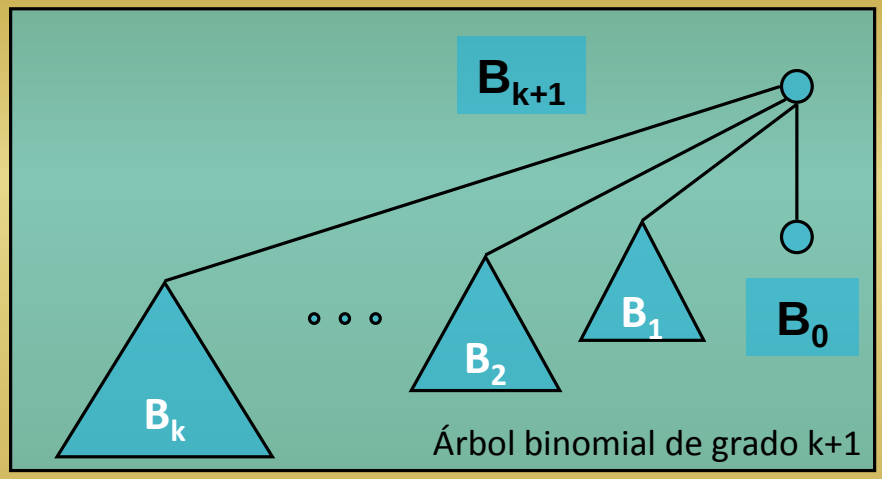
2.2.3 Otros tipos de montículos

a) Montículo binomial:

- Colección de árboles binomiales que verifican la propiedad del montículo.

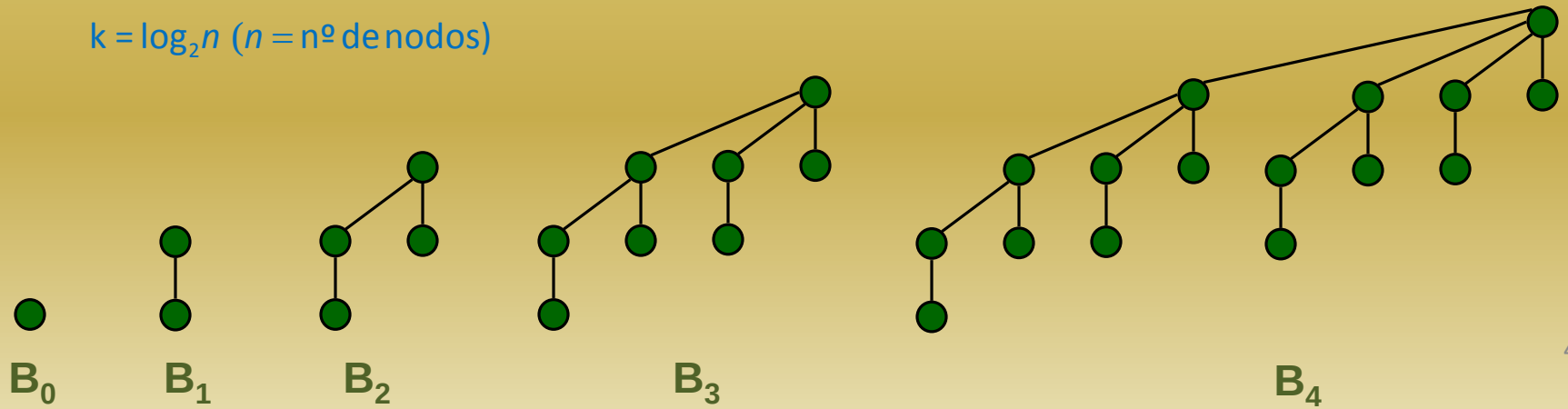
Árboles binomiales

- Propiedades árbol binomial de orden k (B_k):
  - Número de nodos = 2^k.
  - Altura del árbol = grado de la raíz = k.
  - Eliminado la raíz obtenemos árboles binomiales de grados k-1 hasta 0: B_{k-1}, … , B_0.



- Corolario:
  - Profundidad del árbol:

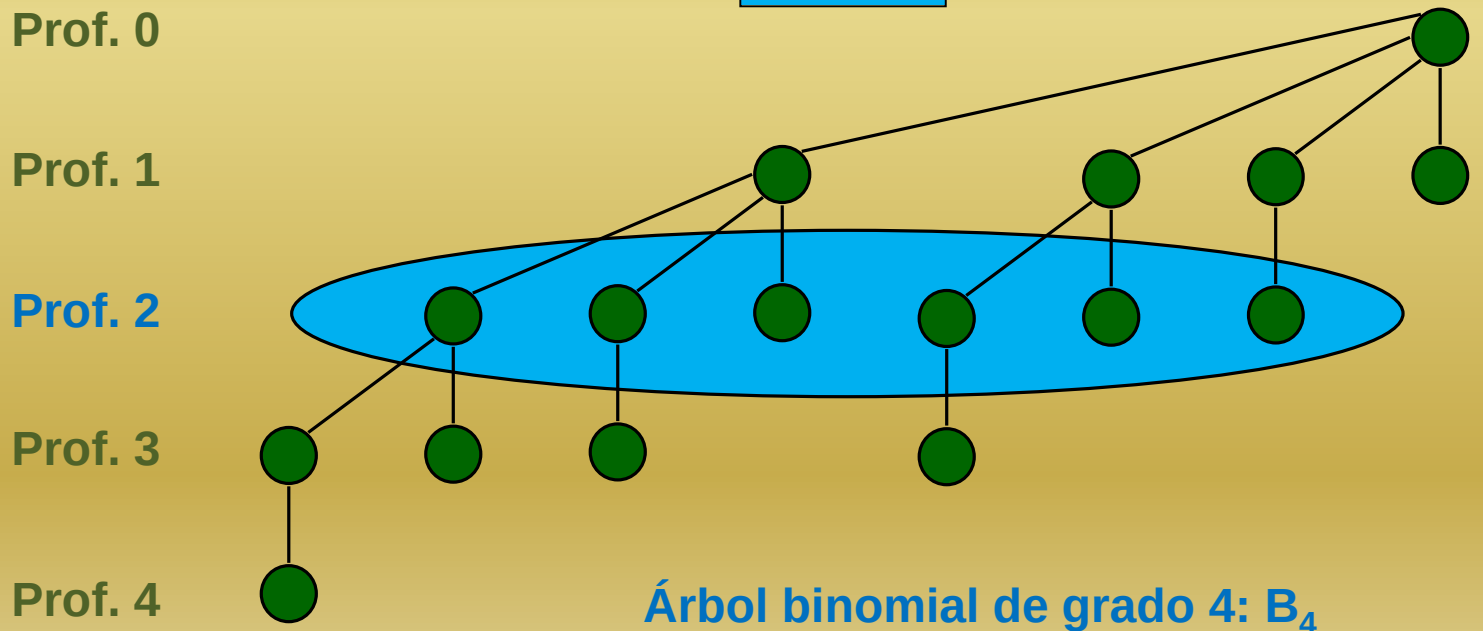
$k = \log_2 n$ (n = nº de nodos)



- Propiedad adicional que le da nombre a la estructura:

B_k tiene $\binom{k}{i}$ nodos a profundidad *i*

$$\binom{4}{2} = 6$$

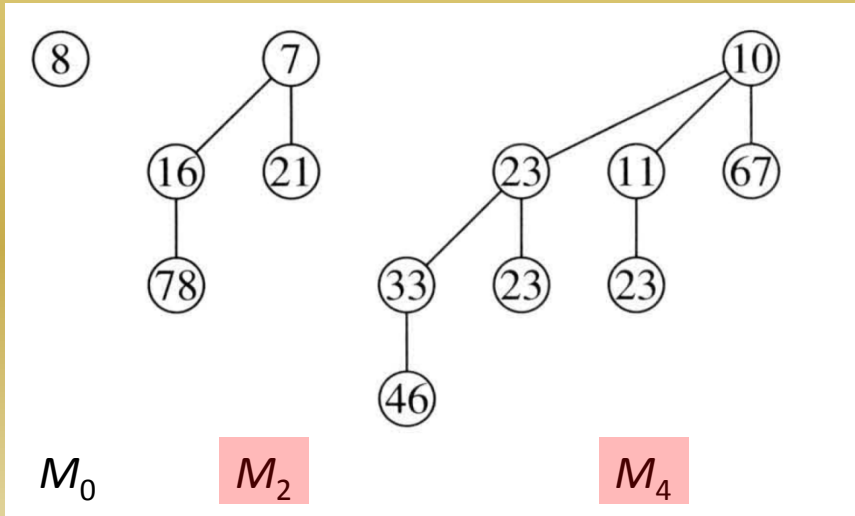


Árbol binomial de grado 4: B_4

a) Montículo binomial:

- Secuencia de árboles binomiales M_0, M_1,… M_k que verifican la propiedad del montículo.
  - En cada árbol binomial, los hijos son siempre menores (mayores) o iguales que el padre para un montículo de máximos (mínimos).
  - Para todo k>0 existe al menos un árbol binomial en el montículo con una raíz de grado *k*.
  - Puede haber solo 0 ó 1 árbol binomial por cada orden, incluido el orden 0.
  - Montículo binomial de n elementos → como máximo $\log_2 n+1$ árboles binomiales

Ejemplo: montículo binomial de 13 elementos: $13_{10}=1101_2 \rightarrow M_0$, M_2 y M_3



Coste:

- Mejora la eficiencia en algunas operaciones como la unión de 2 montículos.
- Veáse tabla siguiente.

b) Montículo de Fibonacci:

Comparativa coste operaciones montículo

		Montículos		
Operación	Lista Enlazada	Binario	Binomial	Fibonacci*
crear vacío	O(1)	O(1)	O(1)	O(1)
insertar	O(1)	O(log n)	O(log n)	O(1)
buscar-min/max	O(n)	O(1)	O(log n)	O(1)
borrar-min/max	O(n)	O(log n)	O(log n)	O(log n)
unión	O(1)	O(n)	O(log n)	O(1)
reducir clave	O(1)	O(log n)	O(log n)	O(1)
borrar nodo	O(n)	O(log n)	O(log n)	O(log n)
está vacío?	O(1)	O(1)	O(1)	O(1)

* En la mayoría de las operaciones se trata de coste amortizado

Montículos de Fibonacci

Situaciones de interés:

- Problemas en los que las operaciones de borrado del mínimo y de borrado de cualquier elemento son poco frecuentes en proporción con el resto.
- Ejemplo: muchos algoritmos de grafos en los que se precisan colas con prioridades y con la ejecución frecuente de la operación de reducción de clave.
  - Algoritmo de Dijkstra para el cálculo de caminos mínimos
  - Algoritmo de Prim para el cálculo de árboles de recubrimiento de coste mínimo

Desde un punto de vista práctico:

- Las constantes multiplicativas en el coste y la complejidad de su programación los hacen menos aconsejables que los montículos "ordinarios" en muchas aplicaciones.
- Por tanto, salvo que se manejen MUCHAS claves, tienen un interés eminentemente teórico.

Montículos de Fibonacci:

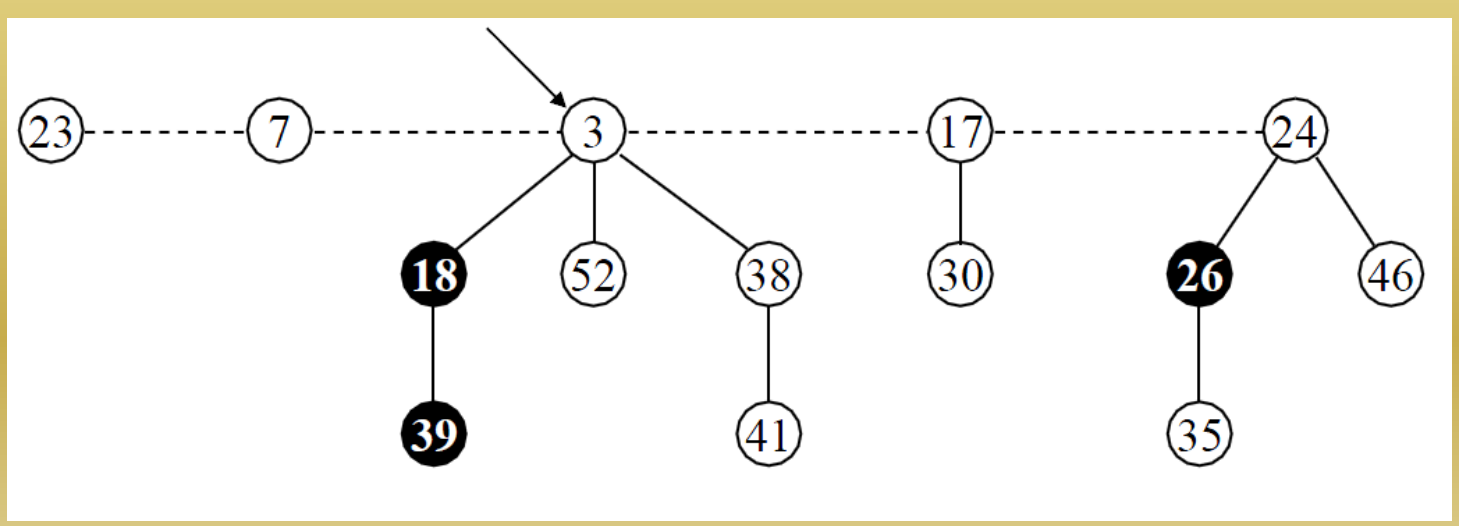
¿Qué son?

– Podría decirse que son una *versión perezosa* de los montículos binomiales.

- Si no se ejecutan operaciones de borrado ni de reducción de claves en un montículo de Fibonacci, entonces cada uno de sus árboles es un árbol binomial.
- Tienen una estructura "más relajada", permitiendo retrasar la reorganización de la estructura hasta el momento más conveniente para reducir el coste amortizado.
- Como resultado de esta estructura, algunas operaciones pueden llevar mucho tiempo mientras que otras se hacen muy deprisa. En el análisis del coste de ejecución amortizado pretendemos que las operaciones muy rápidas tarden un poco más de lo que tardan. Este tiempo extra se resta después al tiempo de ejecución de operaciones más lenta

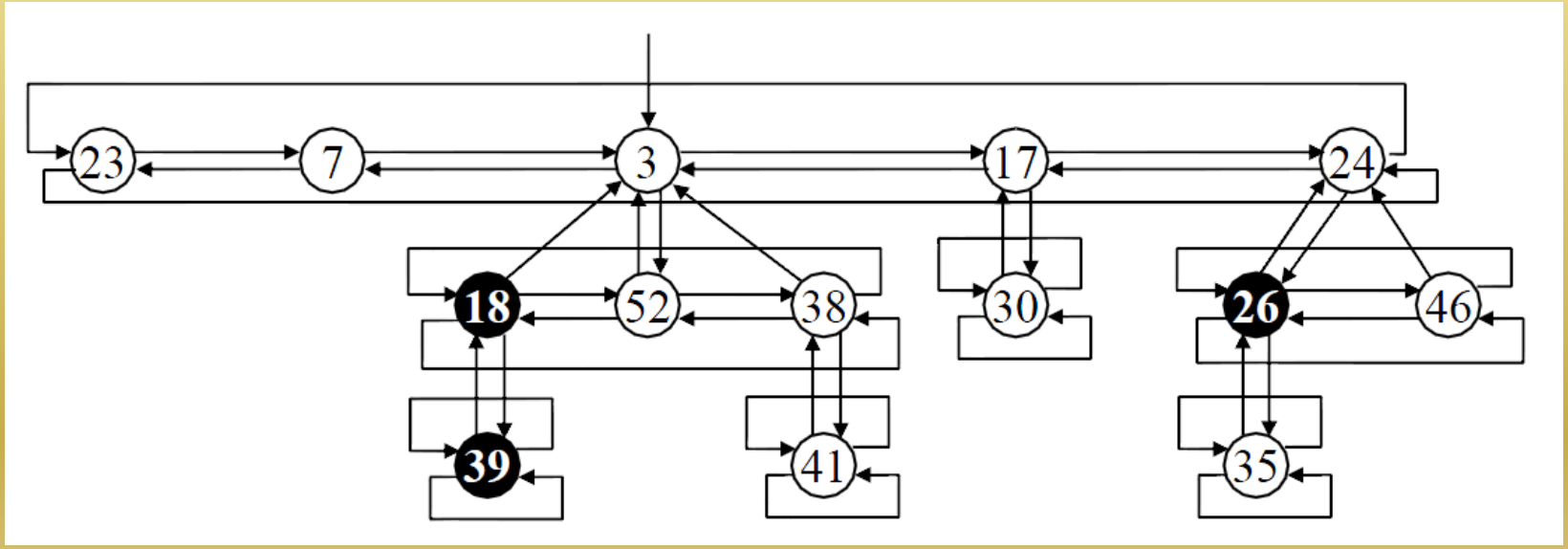
Estructura de un montículo de Fibonacci:

- Es un conjunto de *árboles parcialmente ordenados*, es decir, la clave de todo nodo es mayor o igual que la de su padre.
- Los árboles no precisan ser binomiales y tampoco están ordenados → mucho más flexibles que éstos.
- Sin embargo, hay cierto orden para que la estructura sea rápida → El nº de hijos de cada nodo es menor que log n y el tamaño de los subárboles con raíz en un nodo de grado k es al menos F_{k+2}, siendo F_k el k-ésimo número de la serie de Fibonacci.
- Se accede por un puntero a la raíz de clave mínima.



Representación en memoria:

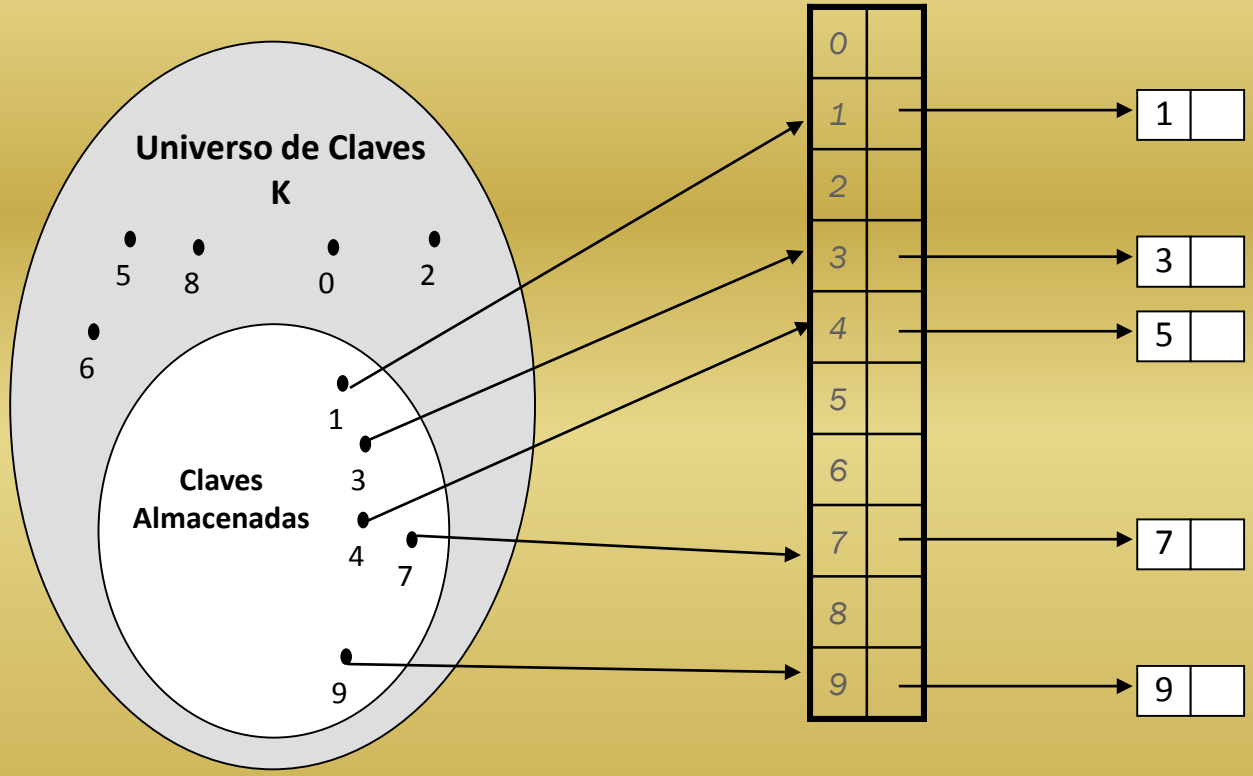
- Cada nodo tiene un puntero al padre y un puntero a alguno de sus hijos.
- Los hijos de un nodo se enlazan con una lista circular doblemente enlazada (cada nodo tiene un puntero a su hermano izquierdo y al derecho). El orden en esa lista es arbitrario.



2.3 Tablas de dispersión (Hash)

- Muchas aplicaciones requieren un conjunto dinámico que soporte las operaciones de un diccionario: ***Inserción, Búsqueda, Eliminación***.
- Es posible hacer uso de una lista enlazada con un tiempo O(n).
- Otra alternativa es el uso de arrays que nos permiten acceso a sus elementos en orden O(1).
- Una opción sería usar un array tan grande como el rango de posibles claves, de tal manera que la clave del elemento determinara su posición en la tabla ***(Tablas de Direccionamiento Directo***). La desventaja es el espacio de memoria requerido en tal estrategia.

Tablas de Direccionamiento Directo (búsqueda por clave directa)



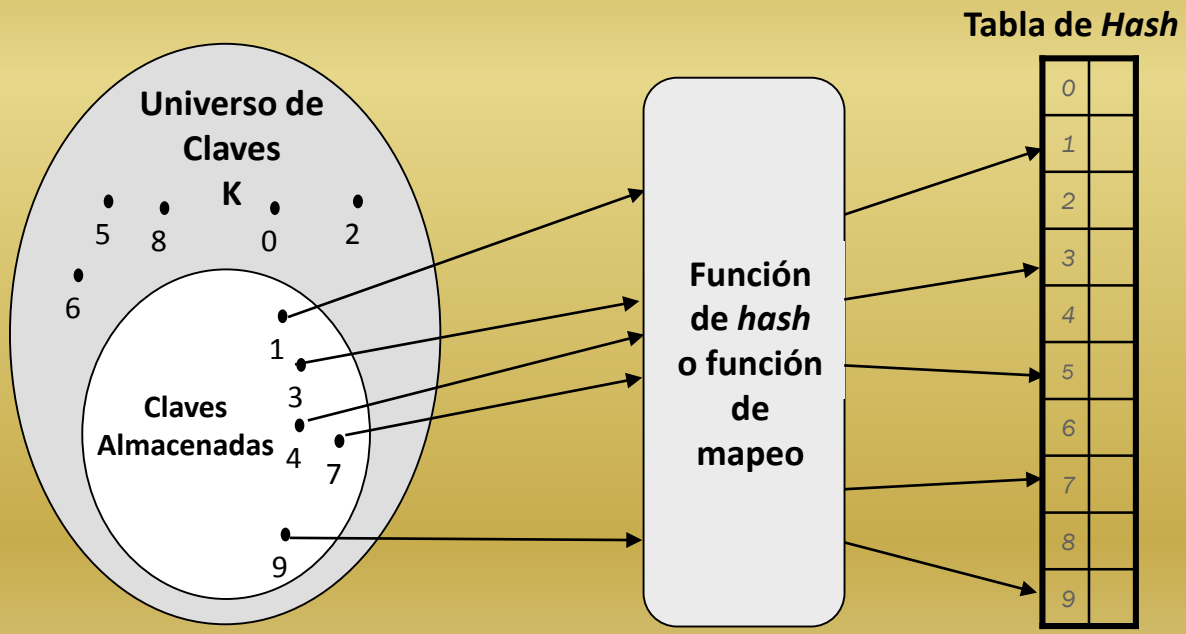
¿Qué ocurre si el conjunto de las posibles claves es mucho más grande que las claves efectivamente almacenadas?

- **Se requeriría una estructura muy grande para almacenar un número relativamente pequeño de elementos**
  - **Mal uso del espacio de almacenamiento**

Tablas Hash

- Solución → usar un array menor, y convertir las claves en uso en índices de la tabla a través de una función. Esta función de mapeo es la *función hash*. La tabla así organizada es la *tabla hash*.
- Ejemplo: fichero de datos de p.ej. 10^3 personas indexados por DNI.
  - Dominio de claves posibles: 00000000 hasta 99999999.
  - Conjunto de índices de la tabla Hash: 000 hasta 999.

Func. hash: h(DNI)



- Es posible que dos claves conduzcan al mismo mapeo, es decir, que la función de *hash* produzca el mismo resultado para dos claves diferentes **(colisión):** $k_1 \neq k_2$; $h(k_1) = h(k_2)$

2.3.1 Funciones Hash

- Una **función *Hash*** asocia una clave con una posición en la tabla *Hash*. Es una aplicación entre el conjunto dominio de las claves *K* y el conjunto dominio de direcciones *D*.

$$H : K \rightarrow D$$
$$k \rightarrow h(k)$$

  - Deben repartir equiprobablemente los valores
  - La función ha de poder calcularse de forma eficiente.
  - Cambios pequeños en clave $\rightarrow$ cambios significativos en la función *Hash*.

- La mayor dificultad es repartir las colisiones adecuadamente. Paradoja del cumpleaños $\rightarrow$ aplicando 23 claves a una tabla de 365 valores, las probabilidades de colisión para una función Hash equiprobable son del 50%.

- **Propiedades**
  - **Sentido único:** no se puede calcular k a partir de h(k) –fundamental en criptografía–.
  - **Colisiones:** son inevitables aunque han de minimizarse.
  - **Distribución de datos:** cómo tratar de distribuir uniformemente los datos.

Tipos de funciones Hash

- Como el rango de la función de ***hash*** es un **número natural**, en el conjunto {0,1,2,..,*M*-1}**,** ésta debe transformar la clave en un número natural en ese rango **(*M es el nº de elementos/índices de la tabla hash).***
- Si se trata de claves enteras, el problema está más o menos resuelto.
- Si se trata de secuencia de caracteres o *strings*, se puede interpretar cada carácter como un número en base 128 (los números ASCII van del 0 al 127) y el *string* completo como un número en base 128.

  Así por ejemplo la clave ***"pt"*** puede ser transformada a ***(112 *128^1 + 116 * 128^0) = 14452***. ***(ASCII(p)=112 y ASCII(t)=116)***.

- En adelante supondremos que las claves son números naturales (o ya han sido transformadas a números naturales)

a) Función módulo (o división)

- Este método consiste en tomar el resto de la división por el número de entradas de la tabla *M*. Asumiendo índices [0..*M*-1]:

$$h(k) = k \bmod M$$

- **Importante → elección de *M***
  - No se recomienda que sea potencia de 2 (el valor de hash dependería sólo de los bits menos significativos de *k*)
  - Utilizar nº primos cercanos a una potencia de 2 (así el resultado depende de más bits de *k*).
- Método muy sencillo pero hay que tener en cuenta el dominio de claves para evitar un número alto de colisiones.
- Ejemplo: Sea *M*=97 (índices [1—96]) y tenemos las claves k_1=7259 y k_2=8708

$$h(k_1) = 7259 \bmod 97 = 81$$
$$h(k_2) = 8708 \bmod 97 = 75$$

c) Función cuadrado

- Este método eleva al cuadrado el valor de la clave y elimina los *c* bits centrales del resultado, dando un valor en el rango [0..2^c-1].
- Ejemplo:
  - Tenemos registros cuya clave sea 4 dígitos numéricos en base 10 [0000..9999].
  - Tenemos una tabla Hash de 100 elementos [0..99] $\rightarrow$ c=2.

$$k=5678 \rightarrow k^2=32239684 \rightarrow \text{dígitos centrales } 39=h(k)$$

d) Función plegado (compresión)

- Este método toma partes de la clave y opera sobre ellas, por ejemplo realizando operaciones XOR. Proceso rápido y aplicable a claves no numéricas aunque puede llevar a muchas colisiones.
- Ejemplo: XOR entre caracteres

$$h(UNED) = U \oplus N \oplus E \oplus D = (01010101 \oplus 01001110) \oplus (01000101 \oplus 01000100)$$

$$= 00011011 \oplus 00000001 = 00011010$$

e) Función multiplicación

- El método opera en 2 pasos:
  - Se normaliza la clave de entrada multiplicándola por una constante $\phi \in (0..1)$ y se extrae la parte fraccionaria (mod 1).

    Valor de ϕ recomendado, razón áurea $\phi_a = \frac{\sqrt{5}-1}{2} \simeq 0.618$
  - Se multiplica el valor obtenido por el número *M* de entradas de la tabla, truncando el resultado.

- Ejemplo: Sea ϕ=0.618 y *M*=1000

$$h(3) = \lfloor 1000((0.618 \times 3)\mathbf{mod1}) \rfloor = \lfloor 1000((1.854)\mathbf{mod1}) \rfloor = 854$$

Otros valores serían $h(1) = 618$, $h(4) = 472$, $h(5) = 90$, etc.

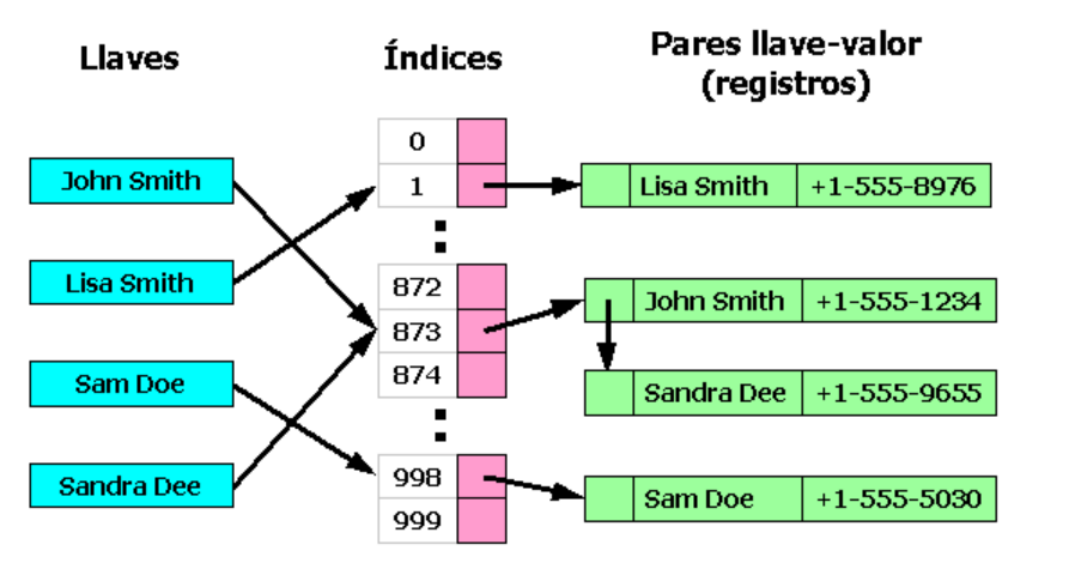
2.3.2 Resolución de colisiones

- Se requiere almacenar los registros que comparten clave de manera que en caso de colisión, sean accesibles igualmente. Dos métodos: *hashing abierto y hashing cerrado*
- La elección de uno u otro depende del *factor de carga* de la tabla $\delta \in [0,1]$:

$$\delta = \frac{n}{M} \ (n \text{ número de índices ocupados})$$

a) Hashing abierto (si el factor de carga es muy elevado):

- Consiste en tener en cada posición de la tabla, una lista de los elementos que, de acuerdo a la función de hash, correspondan a dicha posición.
- El peor caso de hashing abierto nos conduce a una lista con todas las claves en una única lista. El peor caso para búsqueda es así O(n).



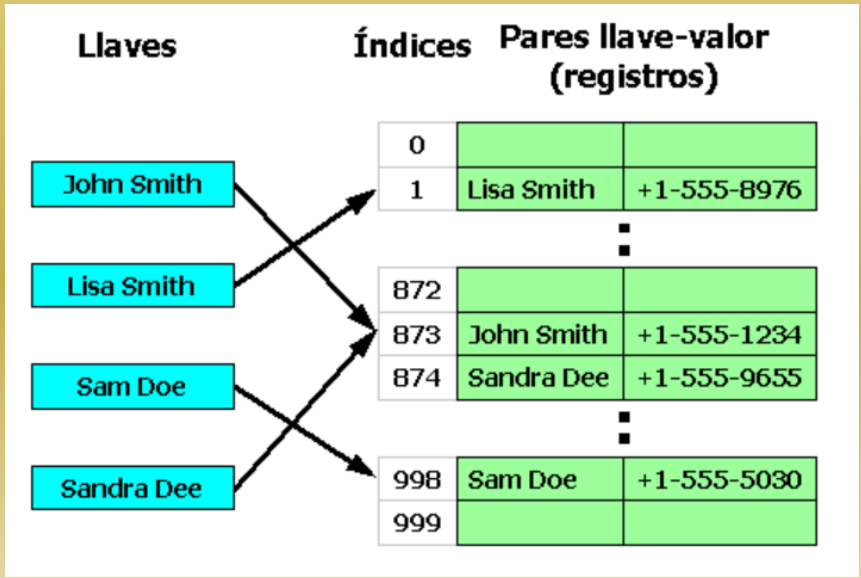
b) Hashing cerrado:

- El hash cerrado soluciona las colisiones buscando celdas alternativas hasta encontrar una vacía.

$$dir(k, i) = (h(k) + f(i)) \bmod M$$

donde ***f(i)*** es la función que determina la **estrategia para la resolución de colisiones** e ***i*** es el **número de celda analizada** que se va incrementando hasta encontrar el hueco (*i*=0,1,2..)

- ***Ventaja:*** Elimina totalmente los apuntadores. Se libera así espacio de memoria, el que puede ser usado en más entradas de la tabla.
- ***Desventaja:*** Si la aplicación realiza eliminaciones frecuentes, puede degradarse el rendimiento de la misma. Se requiere una tabla más grande. Para garantizar el funcionamiento correcto, se requiere que la tabla de *hash* tenga, por lo menos, el **50% del espacio disponible**.
- Dentro de la dispersión cerrada hay tres estrategias distintas para localizar posiciones alternativas en la tabla: la ***exploración lineal***, la ***exploración cuadrática*** y la ***dispersión doble***



Estrategias de exploración de la tabla en Hashing cerrado

- ***Recorrido lineal (o exploración lineal)***

- En este tipo de estrategia la función f() es una función lineal de *i*, por ejemplo: f(*i*)=*i*. Esto significa que las celdas se recorren en secuencia buscando una celda vacía; es decir, si hay una colisión se prueba en la celda siguiente y así sucesivamente hasta encontrar una vacía. La nueva dirección apuntada por la función de *hash* quedaría de la siguiente manera:

$$dir(k,i) = (h(k) + i) \bmod M$$

- Es decir, a la dirección obtenida por la función hash h'(k) *se le añade un incremento lineal*.
- Una desventaja de este método el tiempo que se tarda en encontrar una celda vacía y la formación de largas secuencias de celdas ocupadas, llamada efecto **agrupamiento primario**.
- **Ejemplo:** *h*(*k*)=*k* mod *M,* con *M*=10. Claves: {89, 18, 49, 58, 69}

	Tabla vacía	Después de 89	Después de 18	Después de 49	Después de 58	Después de 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

- *dir*(89,0)=9 → libre
- *dir*(18,0)=8 → libre
- *dir*(49,0)=9 → colisión;
  dir(49,1)=(9+1) mod 10=0 → libre
- *dir*(58,0)=8 → colisión;
  dir(58,1)=(8+1) mod 10=9 → colisión;
  dir(58,2)=(8+2) mod 10=0 → colisión;
  dir(58,3)=(8+3) mod 10=1 → libre
- *dir*(69,0)=9 → colisión;
  dir(69,1)=(9+1) mod 10=0 → colisión;
  dir(69,2)=(9+2) mod 10=1 → colisión;
  dir(69,3)=(9+3) mod 10=2 → libre

Estrategias de exploración de la tabla en Hashing cerrado

- ***Recorrido cuadrático (exploración cuadrática)***

- Este método de resolución de colisiones elimina el problema del agrupamiento primario.
- Se utiliza una función de colisiones cuadrática: $f(i) = c \cdot i^2$. La función de hashing es:

$$dir(k,i) = (h(k) + c \cdot i^2) \bmod M$$

- **Ejemplo:** $h(k)=k \bmod M$, con $M=10$. Claves: {89, 18, 49, 58, 69}, $c=1$.
  - $dir(89,0)=9 \rightarrow$ libre
  - $dir(18,0)=8 \rightarrow$ libre
  - $dir(49,0)=9 \rightarrow$ colisión; $dir(49,1)=(9+1) \bmod 10=0 \rightarrow$ libre
  - $dir(58,0)=8 \rightarrow$ colisión; $dir(58,1)=(8+1^2) \bmod 10=9 \rightarrow$ colisión; $dir(58,2)=(8+2^2) \bmod 10=2 \rightarrow$ libre
  - $dir(69,0)=9 \rightarrow$ colisión; $dir(69,1)=(9+1^2) \bmod 10=0 \rightarrow$ colisión; $dir(69,2)=(9+2^2) \bmod 10=3 \rightarrow$ libre

	Tabla vacía	Después de 89	Después de 18	Después de 49	Después de 58	Después de 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

- ***Recorrido mediante doble hashing (dispersión doble)***

– La función de colisiones para la dispersión doble es: ***f(i)*** = ***i·h'(k)***. Se aplica una segunda función de dispersión a *k*, y luego se prueba a distancias *h'*(x), 2*h'*(x), ... La nueva dirección queda:

$$dir(k,i) = (h(k) + i \cdot h'(k)) \bmod M$$

– Importante la elección de *h'*(*k*)

- *h'* (x)≠ 0, ya que de lo contrario se producen colisiones.
- La función *h'* debe ser diferente de la función *h.*
- Los valores de *h'(x)* deben ser primos relativos de *M* (es decir, que no compartan factores primos), para que los índices de la tabla se ocupen en su totalidad. Si *M* es primo, cualquier función puede ser usada como *h'.*
- En general la función ***h'(k) = R - (k mod R)*** con ***R*** un número primo menor que ***M***, da buenos resultados.

– **Ejemplo:** *h*(*k*)=*k* mod *M,* con *M*=10. Claves: {89, 18, 49, 58, 69}, *R*=7 → *h'*(*k*)=7-(*k* mod 7*)*.

	Tabla vacía	Después de 89	Después de 18	Después de 49	Después de 58	Después de 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

- *dir*(89,0)=9 → libre
- *dir*(18,0)=8 → libre
- *dir*(49,0)=9 → colisión;
  dir(49,1)=(9+h'(49)) mod 10=(9+7) mod 10=6 → libre
- *dir*(58,0)=8 → colisión;
  dir(58,1)=(8+h'(58)) mod 10=(8+5) mod 10=3 → libre
- *dir*(69,0)=9 → colisión;
  dir(69,1)=(9+h'(69)) mod 10=(9+1) mod 10=0 → libre

¡Muchas menos colisiones!

Coste de las operaciones de búsqueda (búsqueda no exitosa=inserción)

TIPO	EXITOSA	NO EXITOSA
Exploración lineal	$\frac{1}{2}\left(1+\frac{1}{(1-\delta)^2}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\delta)^2}\right)$
Doble Hashing	$-\frac{ln(1-\delta)}{\delta}$	$\frac{1}{1-\delta}$
Sondeo cuadrático	$-\frac{ln(1-\delta)}{\delta}$	$\frac{1}{1-\delta}$