

Resumen de programación 3

Tema 5. Estructuras de datos.

Índice:

5.1. Matrices (arrays), pilas y colas.	3
5.2. Registros y punteros (apuntadores)	7
5.3. Listas	8
5.4. Grafos	11
5.5. Árboles	16
5.6. Tablas asociativas	19
5.7. Montículos (heaps)	20
5.8. Montículos binomiales	33
5.9. Particiones	34

Bibliografía:

Se han tomado apuntes de los libros:

- *Fundamentos de algoritmia*. G. Brassard y P. Bratley
- *Estructuras de Datos y Algoritmos*. R. Hernández
- *Esquemas algorítmicos: Enfoque metodológico y problemas resueltos*. J. González y M. Rodríguez

El uso de las estructuras de datos suele ser un factor crucial para el diseño de algoritmos eficientes. Por ello veremos los más usados más adelante.

De modo amplio escribiremos las distintas operaciones de las estructuras de datos, así como el análisis de costes de las mismas. Lo ponemos aunque no haya que estudiarlos con detenimiento.

5.1. Matrices (arrays), pilas y colas.

Definición: Una matriz es una estructura de datos que consta de un número fijo de ítems (o elementos) del mismo tipo.

En una matriz monodimensional, el acceso a todo ítem particular se efectúa especificando un solo índice. A lo largo de todo este resumen (y en temas anteriores también) escribiremos indistintamente tanto matriz como vector, fijándonos que los índices sean uno en el caso de poner el primero.

Ejemplo:

tab: matriz [1..50] de enteros

Aquí, tab es una matriz de 50 enteros indexados desde 1 hasta 50. Por tanto, tab[1] es el primer elemento de la matriz y tab[50] alude al último.

La **propiedad esencial** de esta matriz es que podemos calcular la dirección de cualquier elemento dado en un tiempo *constante*.

Las **operaciones de las matrices** son:

- El tiempo necesario para leer el valor de un solo elemento o para cambiar ese valor se encuentra en $O(1)$ (operaciones elementales).
- Toda operación que implique a todos los elementos de una matriz tenderá a requerir más tiempo a medida que crezca el tamaño de la matriz. Una operación como dar valor inicial a todos los elementos o buscar el mayor elemento tendrá coste $\theta(n)$ (o bien $O(n)$).

Las matrices monodimensionales permiten implementar eficientemente la estructura de datos llamada **pila**. Veremos esta estructura con más detenimiento, aunque insisto que no hay que sabérselo todo de memoria.

Vemos la estructura de datos pila:

Definición: Una pila es una lista dinámica LIFO. La forma de insertar y recuperar elementos hace que el primero en entrar sea el último en salir.

Utilización: Es útil cuando sea importante conservar el orden de inserción y extracción según la propiedad LIFO.

No es útil si el acceso a los elementos es indiscriminado o si debe existir algún tipo de orden en ella dependiendo del valor de los elementos.

Implementaciones:

1. Lista enlazada: Colección de registros que contiene el elemento de información y un apuntador al siguiente.
2. Vectores: Un vector contiene los elementos de información. Se utiliza si podemos estimar el número máximo de elementos que albergamos.

Operaciones:

- Creación:
 1. *fun pila-vacia () dev p:pila*: Devuelve una pila vacía.
- Modificación:
 1. *fun apila (e:elemento, p:pila) dev p:pila*: Añade el elemento a la pila. Se denomina “push”.
 2. *fun desapila (p:pila) dev e:elemento*: Devuelve el elemento situado en la cima de la pila y lo borra de ésta. Se denomina “pop”.
- Consulta:
 1. *fun vacia (p:pila) dev b:booleano*: Comprueba si en la pila hay algún elemento.
 2. *fun llena (p:pila) dev b:booleano*: Comprueba si en la pila está llena, es decir, no caben más elementos.
 3. *fun altura (p:pila) dev n:natural*: Número de elemento en la pila.

El coste asociado según la implementación es:

	Vectores	Punteros
pila-vacia	cte	cte
apila	cte	cte
desapila	cte	cte
vacia	cte	cte
llena	cte	cte
altura	cte	cte

La estructura de datos llamada **cola** también se puede implementar de forma bastante eficiente en una matriz monodimensional. Pasamos a verlo con más detalle como hicimos con la pila:

Definición: Una cola es una lista con dinámica FIFO. Los elementos se insertan en la cola y la recuperación se efectúa en el mismo orden de inserción. El primero en entrar es el primero en salir.

Utilización: Una cola es útil cuando resulta relevante que el orden de inserción y extracción en la estructura se realice según la propiedad FIFO. Análogamente al caso de las pilas, no es de utilidad si el acceso a los elementos es indiscriminado.

Implementaciones: Son las mismas que en el caso de las pilas.

Operaciones: Tiene las suyas propias aunque podríamos usar las de las listas.

- Creación:
 1. *fun cola-vacia () dev p:pila*: Devuelve una cola vacía.
- Modificación:
 1. *fun encolar (e:elemento, c:cola) dev c:cola*: Inserta el elemento en la cola.
 2. *fun desencolar (c:cola) dev e:elemento*: Devuelve el elemento situado al comienzo de la cola y lo borra de ésta.
- Consulta:
 1. *fun vacia (c:cola) dev b:booleano*: Comprueba si en la cola hay algún elemento.
 2. *fun llena (c:cola) dev b:booleano*: Comprueba si en la cola está llena, es decir, no caben más elementos.

El coste asociado según la implementación es:

	Vectores	Punteros
cola-vacia	cte	cte
borrar	cte	$O(n)$
encolar	cte	cte
desencolar	cte	cte
vacía	cte	cte
llena	cte	cte

Tanto para las pilas como para las colas hay una desventaja del uso de las matrices para la implementación y es que normalmente hay que reservar **espacio** para el máximo número de elementos que se prevea. Si reservamos mucho espacio es un desperdicio y si el espacio no es suficiente es difícil reservar más.

Los elementos de una matriz pueden ser de cualquier tipo de longitud fija: entero, booleano, etc.

Ejemplo:

lettab: matriz ['a'..'z'] de valor

No se permite indexar una matriz empleando números reales, ni tampoco estructuras tales como las cadenas o conjuntos.

Podemos declarar matrices con dos o más índices de forma similar a las unidimensionales, las cuales denominaremos bidimensionales o matrices, indistintamente, que será las que tratemos en la asignatura.

Ejemplo:

matriz: matriz [1..20,1..20] de complejo

Una referencia a cualquier elemento requiere ahora dos índices. Ej. matriz [5,7].

Las operaciones de matrices bidimensionales:

- Lectura o modificación del valor de la matriz tiene coste $O(1)$, como las operaciones elementales.
- Dar valor inicial a todos los elementos de la matriz o buscar el elemento mayor requieren un tiempo $\theta(n^2)$ por tener dos dimensiones.

Dijimos que el tiempo necesario para dar un valor inicial a todos los elementos de un vector de tamaño n es $\theta(n)$. Si suponemos que no queremos dar valor inicial a todos los valores, sino que lo único que se precisa saber es si se le ha dado o no valor inicial y obtenemos su valor en tal caso.

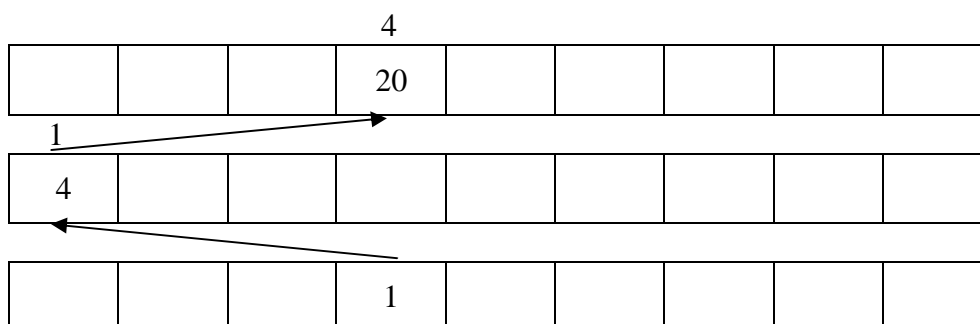
Si estamos dispuestos a emplear más espacio, la técnica denominada inicialización virtual nos permite evitar el tiempo de dar valores a todas las entradas del vector. Tendremos estos componentes:

- $T[1..n]$: Vector que hay que inicializar virtualmente.
- $a[1..n]$ y $b[1..n]$: Vector auxiliares.
- ctr : Contador.

Ejemplo: veremos un ejemplo paso a paso para que se vea más claramente estas componentes:

Al comenzar, damos simplemente a ctr el valor 0 y dejamos los vectores a , b y T con aquellos valores que pudieran contener, o lo que es igual no inicializamos ningún valor en ninguno de estos vectores.

En el paso primero, inicializamos el valor en la posición 4 en T , supongamos 20:



siendo:

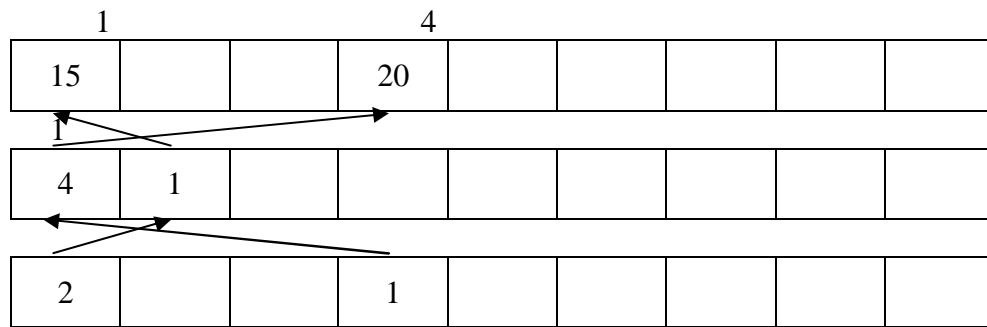
T : El primer vector, que por el diseño no podemos poner el nombre. Indica que se ha inicializado un valor en la posición 4.

$ctr = 1$: El contador se pone a 1, por ser el primer valor.

$a[1] = 4$: Es la segunda matriz de nuestro dibujo. Indica la posición inicializada en primer lugar.

$b[4] = 1$: Es la última matriz. Nos dice qué elemento de T está inicializado en qué posición. En este caso, el 4^a elemento de T es el primero en inicializarse.

En el paso segundo, inicializamos el valor 15, por ejemplo, en la posición 1 en T:



siendo:

T: El primer vector, que por el diseño no podemos poner el nombre. Indica que se ha inicializado otro valor en la posición 1.

$ctr = 2$.

$a[2] = 1$: Es la segunda matriz de nuestro dibujo. Indica la posición inicializada en segundo lugar.

$b[1] = 2$: Es la última matriz. Nos dice qué elemento de T está inicializado en qué posición. En este caso, el 1^{er} elemento de T es el segundo en inicializarse.

Para determinar si se ha asignado un valor a $T[i]$, comprobamos primero si $1 \leq b[i] \leq ctr$. Si no se cumple, no ha sido inicializado. Si se cumple, verificamos si realmente ha sido asignado si $a[b[i]] = i$, siendo afirmativo cuando $T[i]$ ha sido iniciado y si no, no.

5.2. Registros y punteros (apuntadores).

Definición: Un registro es una estructura de datos que consta de un número fijo de elementos, que suelen llamarse *campos* en este contexto y que son de tipos posiblemente distintos.

Ejemplo:

```

tipo persona = registro
  nombre: cadena
  edad: entero
  peso: real
  varon: booleano

```

Podremos referenciar una variable usando la *notación de punto*, como puede ser Juan.edad. Las matrices pueden aparecer como elementos de un registro y los registros se pueden almacenar en matrices.

Ejemplo:

```

clase: matriz [1..50] de persona

```

Las **operaciones de registro** son:

- La dirección de todo elemento particular se puede calcular en un tiempo constante, así que las consultas o modificaciones del valor de un campo se pueden considerar como operaciones elementales.

Se pueden utilizar los registros en conjunción con los punteros.

Ejemplo:

tipo jefe = ^ persona

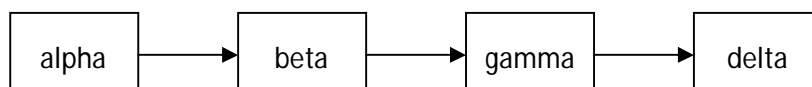
donde jefe es un puntero de un registro cuyo tipo es persona. Para hacer alusiones a los campos de este registro utilizaremos la siguiente notación jefe^.nombre, fijándonos en que la flecha se pone después del nombre del tipo.

Por último, decir que si un puntero tiene el valor especial nulo “nil” entonces no apunta a ningún registro, esto es importante para verlo después en otras estructuras de datos.

5.3. Listas.

Definición: Una lista es una colección de elementos de información dispuestos en un cierto orden.

A diferencia de las matrices y registros, el número de elementos de la lista no suele estar fijado ni suele estar limitado por anticipado (recordemos que era el inconveniente de estas estructuras). Podemos determinar cuál es el primer elemento, cuál es el último, cuál el predecesor y sucesor de esta estructura. En una máquina, el espacio correspondiente a cualquier elemento dado suele denominarse un nodo. La información asociada a cada nodo se muestra dentro del cuadro correspondiente y las flechas muestran los enlaces que van desde el nodo a su sucesor.



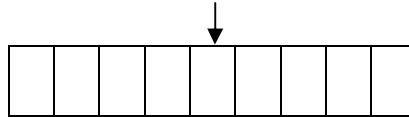
Utilización: El uso más general es el de almacenar elementos de información sin poder estimar el número máximo de elementos de que disponemos. Si la forma de acceso está bien definida es posible que podamos optar por una pila o una cola.

Implementaciones:

1. **Mediante vectores:** Un vector contiene los elementos de información. Se utiliza si podemos estimar el número máximo de elementos que albergamos. Usaremos esta declaración:

tipo lista = registro
contador: 0..longmax
valor: matriz [1..*longmax*] de información

Los ítems (o elementos) de una lista ocupan las posiciones desde valor[1] hasta valor[contador] y el orden de sus elementos es el mismo que el orden de sus índices dentro de la matriz.



La lista no tiene tamaño definido pero se limita según la memoria disponible en el equipo.

Características de esta implementación:

- Asignación estática.
- Inserción costosa: Hay que insertar el elemento y luego desplazar el resto a la derecha. En el caso peor, hay que mover todos los elementos. Sería coste lineal.
- Las búsquedas son eficientes.

Operaciones de esta implementación:

- a) Se puede hallar rápidamente los elementos primero y último de la lista, así que como también el predecesor y sucesor de cualquier elemento dado. El coste es constante ($O(1)$).
- b) Insertar un nuevo elemento o borrar uno requiere un número de operaciones que, en el caso peor, está en el orden del tamaño actual de la lista. Tendría coste lineal ($O(n)$).

Desventaja:

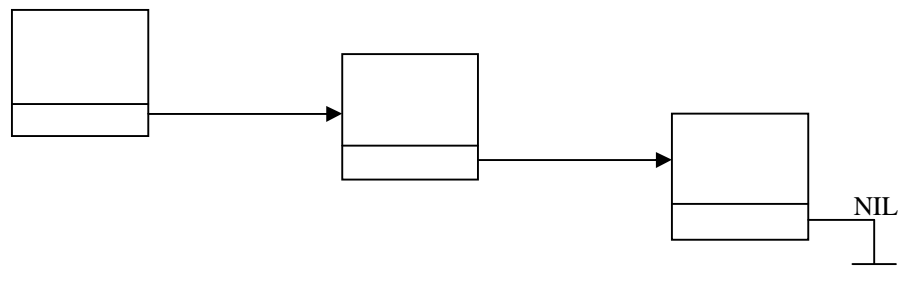
- Todo el almacenamiento precisado está reservado a lo largo de toda la vida del programa.

2. Mediante punteros: Empleamos esta declaración:

```

tipo lista = ^ nodo
tipo nodo = registro
    valor: información
    siguiente: ^ nodo
  
```

Todos los nodos salvo el último incluyen un puntero explícito a su sucesor. El puntero del último nodo tiene el valor especial nulo (“nil” o NIL) para indicar que no apunta a ningún nodo.



Características de esta implementación:

- Asignación dinámica: Según el número de elementos que hagan falta.
- Inserciones eficientes: Sólo hace falta cambiar dos punteros.
- Búsqueda costosa: Pasa al revés que antes.

Operaciones de esta implementación:

- a) Examinar el k-ésimo elemento, para un k arbitrario, se necesita seguir k punteros. Coste $O(k)$.
- b) Inserción de un nuevo nodo o el borrado de un nodo. Coste constante.

Operaciones de las listas en general:

- Creación:
 1. *fun lista-vacia () dev l:lista*: Devuelve una lista vacía.
- Modificación:
 1. *fun añadir (e:elemento, l:lista) dev l:lista*: Añade el elemento a la lista.
 2. *fun resto (l:lista) dev l:lista*: Elimina el primer elemento y devuelve el resto de la lista.
- Consulta:
 1. *fun vacia (l:lista) dev b:booleano*: Comprueba si en la lista hay algún elemento.
 2. *fun miembro (e:elemento, l:lista) dev b:booleano*: Comprueba si el elemento forma parte de la lista.
 3. *fun elemento (p:posición, l:lista) dev e:elemento*: Consulta el elemento de la posición p de la lista, sin modificarlo.
 4. *fun primero (l:lista) dev e:elemento*: Extrae el primer elemento de la lista sin modificarla.

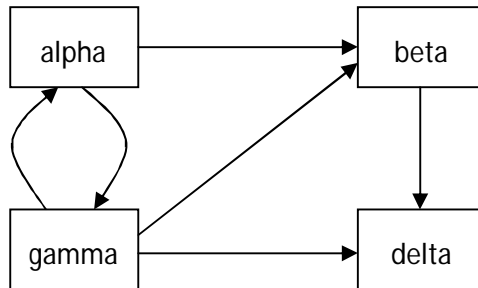
	Vectores	Punteros
lista-vacia	cte	cte
añadir	cte	cte
resto	cte	cte
vacía	cte	cte
miembro	$O(n)$	$O(n)$
elemento	cte	$O(n)$
primero	cte	cte

5.4. Grafos.

Daremos una breve introducción de los grafos, donde se nos darán nociones básicas del mismo, antes de meternos con la definición formal. Este apartado es de los más importantes, sin descartar lo anterior.

Intuitivamente, un **grafo** es un conjunto de nodos unidos por un conjunto de líneas o flechas (llamadas aristas).

Ejemplo:



Conceptos básicos de grafos:

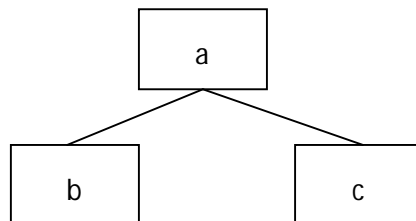
- **Camino:** Sucesión de vértices y aristas que comunican un vértice con otro.
- **Peso:** Valor asociado a una arista. Indica el coste o el valor de uso de dicha arista.
- **Ciclo:** Camino propio que empieza y termina en el mismo vértice.

Tendremos estos tipos de grafos, igualmente pasamos a definirlos:

1. **Dirigidos/no dirigidos:** Podremos formar caminos y ciclos con estos tipos de grafos.

Grafos no dirigidos: Un grafo es no dirigido si la unión entre cualesquiera dos vértices adyacentes es simétrica. Se ve claramente al unirse los nodos mediante líneas sin flechas.

Ejemplo:



Veremos el conjunto de nodos y de aristas del grafo:

$$N = \{a, b, c\}.$$

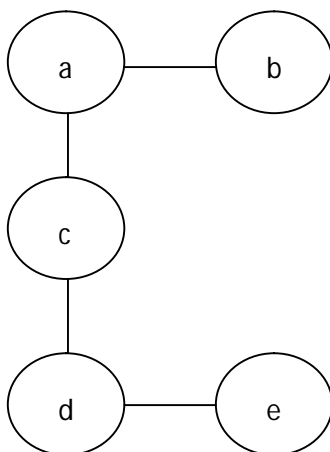
$$A = \{\{a, b\}, \{a, c\}\}.$$

Dirigidos: Los nodos se unen mediante líneas con flecha asociada. En el ejemplo anterior, la única diferencia es que el conjunto de aristas lo denotaremos por (a, b) . Siguiendo el ejemplo anterior tendremos que el conjunto de aristas del grafo será: $A = \{(a, b), (a, c)\}$.

2. Conexo/no conexo:

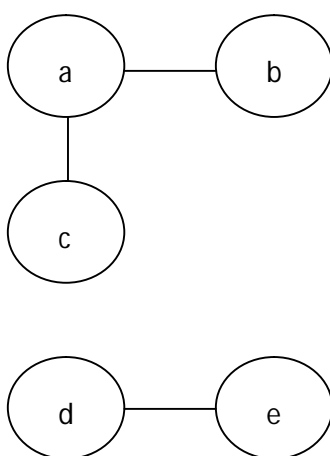
Grafo conexo: Un grafo es conexo cuando siempre hay al menos un camino entre cualesquiera dos vértices, es decir, si todos los nodos están conectados por alguna arista.

Ejemplo:



Grafo no conexo: No todos los nodos están conectados.

Ejemplo:

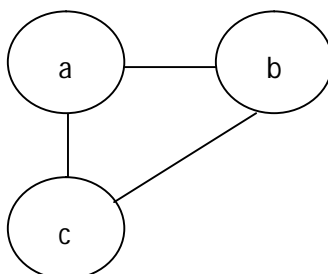


Un grafo dirigido es **fuertemente conexo** si se puede pasar desde cualquier nodo hasta cualquier otro siguiendo una secuencia de aristas, pero respetando esta vez el sentido de las flechas.

3. Cíclico/acíclico:

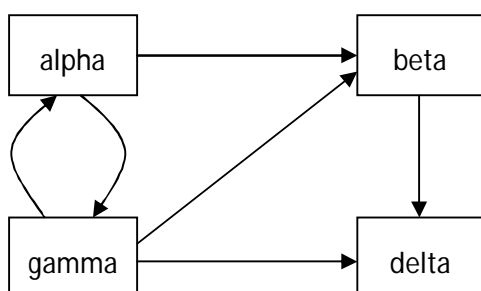
Cíclico: Puede crear un ciclo desde un nodo al otro. En este ejemplo será un grafo no dirigido, pero puede ser el dirigido según el sentido de las flechas.

Ejemplo:



Acíclico: Es justo lo contrario, no se crearía ningún ciclo.

Definición formal de grafo: Un grafo es una pareja $G = \langle N, A \rangle$ en donde N es un conjunto de nodos y A es un conjunto de aristas. Pondremos un ejemplo que nos servirá a lo largo del tema:



$N = \{\text{alpha}, \text{beta}, \text{gamma}, \text{delta}\}.$

$A = \left\{ (\text{alpha}, \text{beta}), (\text{alpha}, \text{gamma}), (\text{beta}, \text{delta}), (\text{gamma}, \text{alpha}), \right. \\ \left. (\text{gamma}, \text{beta}), (\text{gamma}, \text{delta}) \right\}.$

Utilización: Se utiliza para la representación de los elementos de información y de la relación entre ellos. Un ejemplo puede ser la representación de las ciudades (vértices) y las carreteras (aristas) así como las distancias (peso) que hay entre ellas.

Implementaciones: Tendremos dos tipos:

1. **Matriz de adyacencia:** Un registro contiene, por un lado, un vector con los elementos de información contenido en los vértices y, por el otro lado, una matriz que puede tener valores lógicos indicando existencia o no de aristas, o bien un valor que indique el peso o coste de dicha arista.

tipo grafoadya = registro
valor: matriz $[1..numnodos]$ de información
adyacente: matriz $[1..numnodos, 1..numnodos]$ de boolean

Si existe arista de i a j entonces $adyacente[i, j] = \text{verdadero}$, en caso contrario $adyacente[i, j] = \text{falso}$.

En un grafo **no** dirigido, tal y como dijimos en su definición anteriormente, $adyacente[i, j] = adyacente[j, i]$, es decir, la matriz de adyacencia es simétrica (una mitad es igual a la otra).

Operaciones de esta implementación:

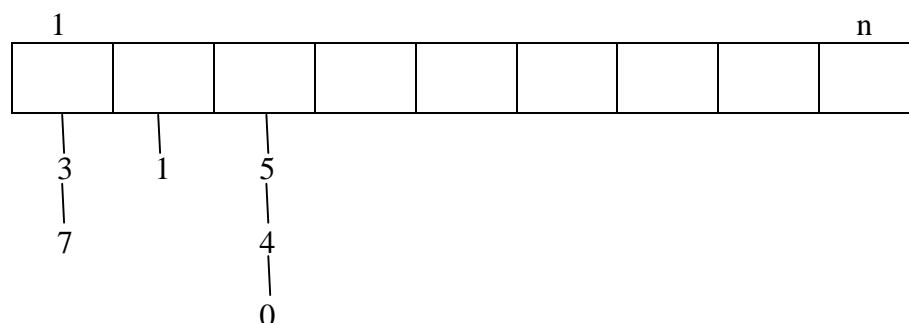
- Para saber si existe arista entre i y j hay que buscar un valor de la matriz. Coste constante, $O(1)$.
- Si deseamos examinar todos los nodos que están conectados con algún nodo dado hay que recorrer toda una fila completa en el caso de un grafo no dirigido o bien tanto una fila completa como una columna completa en el caso de un grafo dirigido. Tiene coste $\theta(numnodos)$.
- El espacio requerido para representar un grafo es cuadrático. Coste $\theta(numnodos^2)$.

2. **Array de registros (lista de adyacencia):** Una matriz de vértices contiene el valor de estos y una lista de sus vértices sucesores.

tipo grafolista = matriz $[1..numnodos]$ de
registro
valor: información
adyacente: lista

Aquí se asocia a cada nodo i una lista formada por sus vecinos, esto es, una lista formada por aquellos nodos j , tales que exista una arista de i a j (para grafo dirigido) o entre i y j (para grafo no dirigido).

Ejemplo:



Operaciones de esta implementación:

- Puede ser que sea posible examinar todos los vecinos de un nodo dado en menos de numnodos operaciones. $O(\text{numnodos})$.
- Determinar si existe o no una conexión directa entre dos nodos dados i y j nos obliga a recorrer la lista de vecinos del nodo i (y también del j , si fuera un grafo dirigido), lo cual es menos eficiente que buscar un valor booleano en una matriz. Equivale a buscar el nodo. El coste es $O(\text{numnodos})$, que es peor que antes.

Para pocas aristas ocupa menos espacio que la implementación anterior.

Operaciones de grafos en general:

- Creación:
 1. *fun grafo-vacío () dev g:grafo*: Devuelve un grafo vacío.
- Modificación:
 1. *fun sucesores (v:vértice, g:grafo) dev l:lista*: Devuelve una lista con los vértices adyacentes a v .
 2. *fun peso (v_1, v_2 :vértice, g:grafo) dev p:peso*: Peso asociado a la arista que une los vértices dados.
 3. *fun añadir-arista (v_1, v_2 :vértice, p:peso, g:grafo) dev g:grafo*: Añade una arista entre los vértices dados y le asigna el peso p .
 4. *fun añadir-vértice (v:vértice, g:grafo) dev g:grafo*: Añade el vértice v al grafo g .
 5. *fun borrar-arista (v_1, v_2 :vértice, g:grafo) dev g:grafo*: Elimina la arista que une los vértices dados.
 6. *fun borrar-vértice (v:vértice, g:grafo) dev g:grafo*: Borra el vértice del grafo y todas las aristas que partan o lleguen de él.
- Consulta:
 1. *fun adyacente (v_1, v_2 :vértice, g:grafo) dev b:boolean*: Comprueba si los vértices v_1 y v_2 son adyacentes.

	Matriz de adyacencia	Lista de adyacencia
grafo-vacío	cte	cte
sucesores	$O(n)$	cte
peso	cte	$O(n)$
añadir-arista	cte	cte
borrar-arista	cte	$O(n)$
añadir-vértice	cte	cte
borrar-vértice	$O(n)$	$O(n)$
adyacente	cte	$O(n)$

5.5. Árboles.

Un **árbol** es un grafo acíclico, conexo y no dirigido. Se puede definir como un grafo no dirigido en el cual existe exactamente un camino entre todo par de nodos dado.

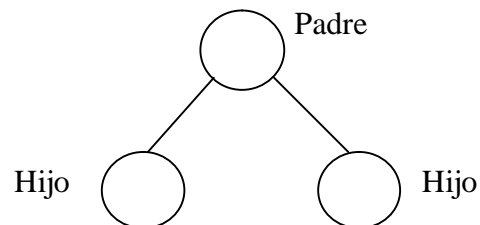
Se emplean las mismas implementaciones que un grafo.

Propiedades de los árboles:

- Un árbol con n nodos contiene exactamente $n - 1$ aristas.
- Si se añade una única arista a un árbol, entonces el grafo resultante contiene un único ciclo.
- Si se elimina una única arista de un árbol, entonces el grafo resultante ya no es un conexo.

Nos interesamos por los árboles con raíz, en los cuales hay un nodo llamado raíz, que es especial. La raíz la dibujaremos en la parte superior y luego su descendencia, como un árbol genealógico. Usaremos el término árbol en todas las ocasiones y en todo el resto de asignatura.

Ejemplo:

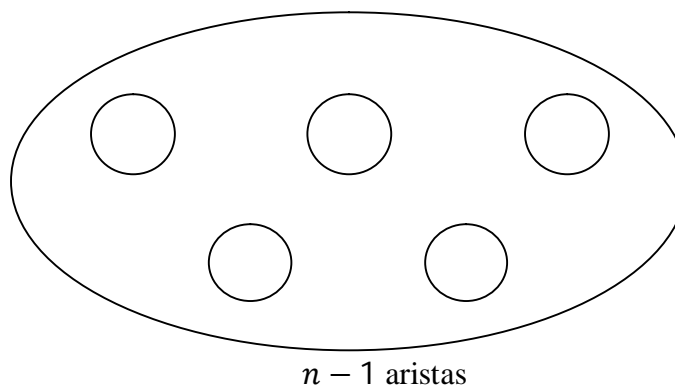


Una de las propiedades del árbol la pasamos a ver con más detenimiento:

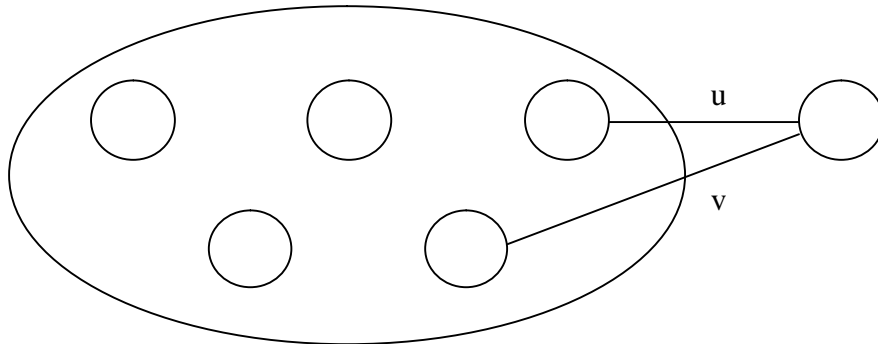
- Un árbol con n nodos contiene exactamente $n - 1$ aristas.

Para comprobarlo usaremos la **demostración por inducción**, que dimos brevemente en el tema 1, aunque aquí lo trataremos con más detenimiento:

Hipótesis de inducción: Suponemos que los nodos están conectados por las aristas.



Si se añade un nodo más, entonces se añade una sola arista (u), ya que si añadimos otra más (v) crea un ciclo. Se pasaría a $n + 1$ nodos y n aristas.



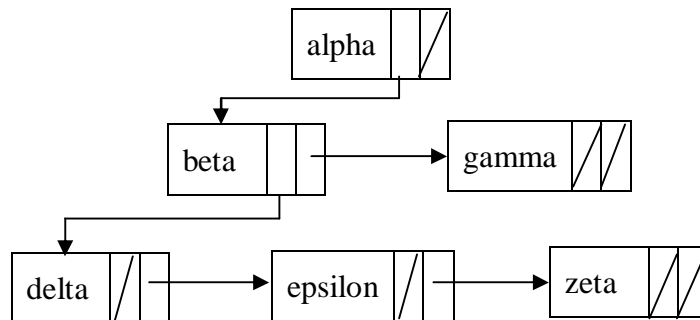
Implementaciones:

1. **Con punteros al hijo mayor y al hermano siguiente.** Se usan nodos del tipo:

```

tipo nodoarbol1 = registro
  valor: información
  hijo_mayor, hermano_siguiente: ^ nodoarbol1
  
```

Ejemplo:



La ventaja es que todos los nodos se pueden representar utilizando la misma estructura registro, independientemente del número de hijos y hermanos que posean.

2. **Con punteros al padre.** Su representación es esta:

```

tipo nodoarbol2 = registro
  valor: información
  padre: ^ nodoarbol2
  
```

Cada nodo contiene un único puntero que lleva a su padre. Esta representación es la más económica en términos de espacio, pero es poco eficiente a no ser que todas las operaciones del árbol impliquen comenzar de un nodo y subir, sin descender nunca.

Si queremos acelerar operaciones que queremos efectuar a base de añadir punteros suplementarios. Por el contrario, se incrementa el espacio.

Tendremos ocasión de usar *árboles binarios*, de 0, 1 y 2 hijos, distinguiendo entre hijo izquierdo y derecho.

3. **Árbol k-ario.** Generalizando si en cada nodo del árbol no puede haber más de k hijos, se trata de un árbol k -ario. Esta representación emplea estos nodos:

tipo nodo- k -ario = registro
valor: información
hijo: matriz $[1..k]$ de \wedge nodo- k -ario

En el caso de un *árbol binario* (de 0 a 2 hijos) tenemos:

tipo nodo-binario = registro
valor: información
hijo-izquierdo, hijo-derecho: \wedge nodo-binario

Un árbol binario es un **árbol de búsqueda** si el valor contenido en todos los nodos internos es mayor o igual que los valores contenidos en su hijo izquierdo o en cualquiera de los descendientes de ese hijo y menor o igual que los valores contenidos en su hijo derecho o en cualquiera de los descendientes de ese hijo. Sólo mostraremos brevemente los conceptos del árbol de búsqueda sin entrar en más detalle.

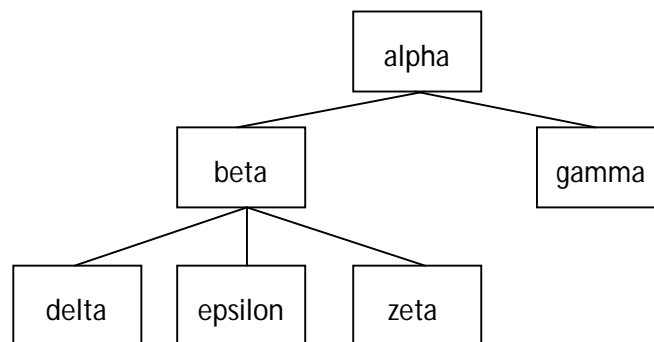
Operaciones del árbol de búsqueda:

- Borrar un nodo o añadir un nuevo valor es fácil pero luego hay que recuperar la propiedad del montículo. Se vuelve un árbol desequilibrado con ramas largas y delgadas, cuya búsqueda será entonces lineal.
- Para equilibrar el árbol tendremos coste $O(\log n)$ en el caso peor, siendo n el número de nodos que hay en el árbol.

Conceptos sobre árboles:

- **Altura de un nodo:** Es el número de aristas que hay en el camino más largo que vaya desde el nodo en cuestión hasta una hoja.
- **Profundidad de un nodo:** Es el número de aristas que hay en el camino que va desde el nodo raíz hasta el nodo en cuestión.
- **Nivel de un nodo:** Es igual a la altura de la raíz del árbol menos la profundidad del nodo estudiado.

Una aplicación de estos conceptos puede ser el siguiente **ejemplo**:



Nodo	Altura	Profundidad	Nivel
alpha	2	0	2
beta	1	1	1
gamma	0	1	1
delta	0	2	0
épsilon	0	2	0
zeta	0	2	0

Los valores que se nos dan son todos sacados de la definición, aunque nos pararemos en deducir el nivel, en este caso veremos el de alpha y beta con más calma.

Nivel de alpha = altura de alpha – profundidad de alpha = $2 - 0 = 2$.

Nivel de beta = altura de alpha – profundidad de beta = $2 - 1 = 1$.

5.6. Tablas asociativas.

Brevemente daremos las nociones básicas en este apartado.

Definición: Una tabla asociativa es igual a una matriz, salvo que su índice no está restringido a encontrarse entre dos cotas predeterminadas.

Para implementarlo usaremos una lista:

```

tipo lista_tabla = ^ nodo_tabla
tipo nodo_tabla = registro
    indice: tipo_indice
    valor: información
    siguiente: ^ nodo_tabla
  
```

Esta implementación es ineficiente en el caso peor, requiriendo un tiempo que se encuentra en $\Omega(n^2)$.

Todos los compiladores utilizan una tabla asociativa para implementar la *tabla de símbolos*, que contiene los identificadores que se utilizan en el programa que hay que compilar.

Veremos varios conceptos:

- **Función de dispersión:** Es una función $h: U \rightarrow \{0, 1, 2, \dots, N - 1\}$ que debe dispersar todos los índices probables: $h(x)$.
- **Colisión:** ocurre cuando $x \neq y$ pero $h(x) = h(y)$.
- **Factor de carga:** Es m/N , donde m es el número de índices distintos que se han almacenado en la tabla y N es el tamaño de la matriz que se emplea para implementarla.
- **Redispersión:** Lo usaremos para mantener valores pequeños del factor de carga. Consiste en volver a hacer la dispersión con otra función distinta.

5.7. Montículos (heaps).

Esta es una estructura de las más importantes que daremos a lo largo del curso, por lo que nos centraremos en las distintas propiedades y operaciones.

Definición: Un montículo es un tipo especial de árbol que tiene la propiedad particular que se puede implementar en una matriz sin punteros explícitos.

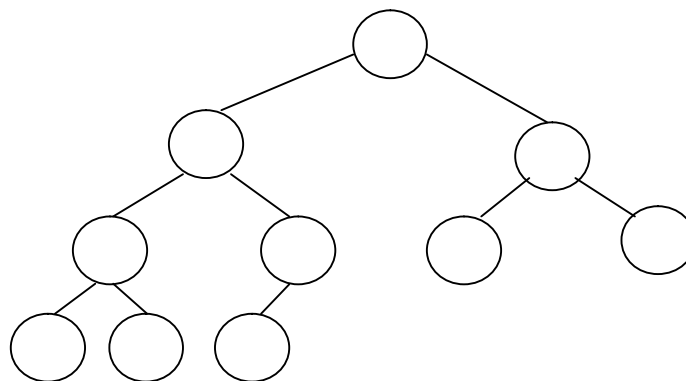
Tiene estas **características**:

- Es binario, significa que tiene de 0 a 2 hijos cada nodo.
- Cada uno de los nodos incluye un elemento de información llamado valor del nodo, siendo éste mayor o igual que los valores de sus hijos.
- Es esencialmente completo: Todo nodo interno, con la posible excepción de un nodo especial, tiene exactamente dos hijos.

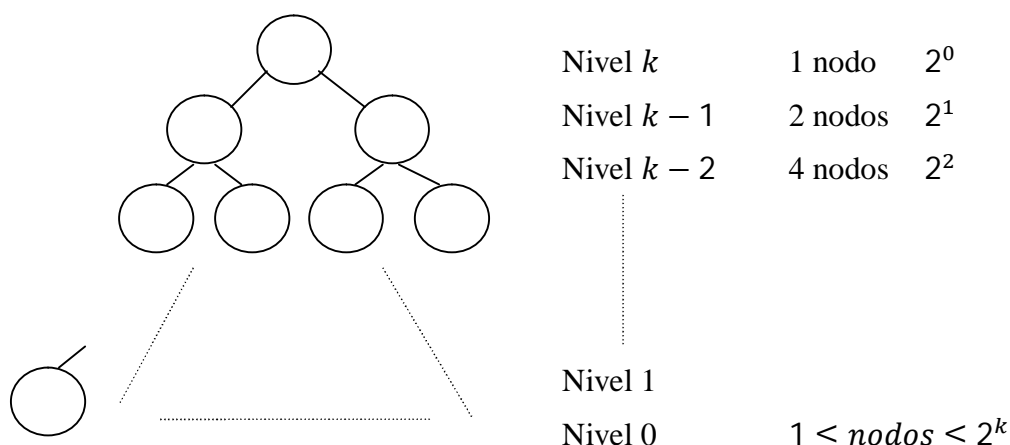
Pasamos esta última característica con más detenimiento:

Árbol binario esencialmente completo: Es un árbol binario en el que todos los nodos finales hasta el penúltimo están completos (con mayor número de nodos posibles) y el último está incompleto, aunque ordenado. Se irá rellenando de arriba a abajo y de izquierda a derecha.

Ejemplo:



Calculamos el número de nodos que tendrá un árbol binario, como sigue:



Se van rellenando los nodos hasta completar el nivel 0.

Si contamos todos los nodos hasta el penúltimo nivel:

$$\sum_{i=0}^{k-1} 2^i = \frac{1-2^{k-1+1}}{1-2} = 2^k - 1 \text{ nodos.}$$

Para resolver la serie tendremos que seguir esta fórmula, que ya hemos aplicado previamente:

$$\sum_{i=0}^k r^i = \frac{1-r^{k+1}}{1-r}.$$

Por tanto, para el árbol contiene estos nodos $2^k \leq n \leq 2^{k+1}$.

La altura del árbol que contiene n nodos es $k = \lfloor \log_2 n \rfloor$, lo que significa que es un redondeo por abajo, quedándonos con la parte entera.

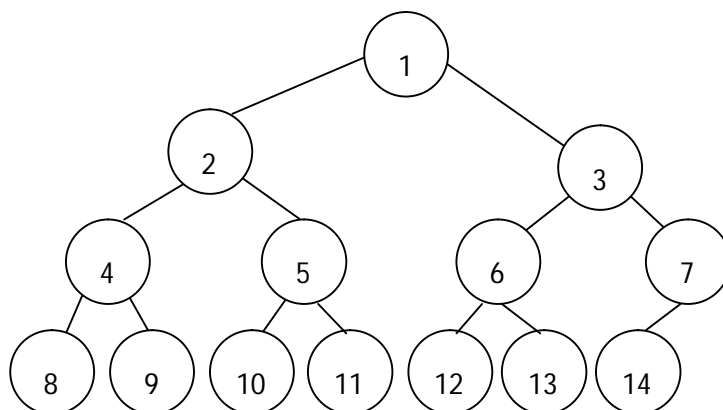
Ejemplo: si tenemos tres nodos, la altura sería $\lfloor \log_2 3 \rfloor = 1$.

Es importantísimo tener bien claro la **propiedad del montículo**:

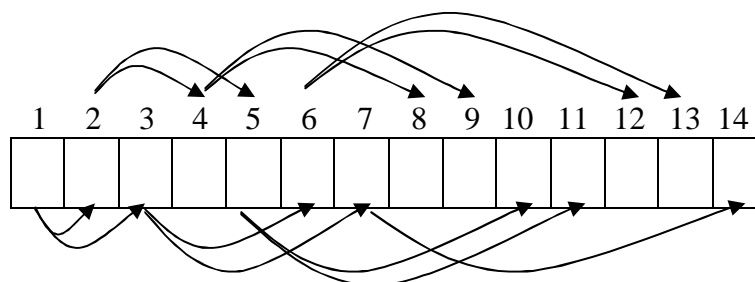
El nodo i es el padre de $\left\{ \frac{2 * i}{2 * i + 1} \right\}$.

El nodo i es el hijo del nodo $i \div 2$ (o $i \div 2$).

Ejemplo: tenemos este montículo:



Queremos verificar si cumple estas condiciones, para ello lo representamos en forma de vector:



Las flechas indican cuál es el padre y cuál es el hijo, es decir, los hijos del nodo 1 son las posiciones 2 y 3.

El número del nodo indica el orden de inserción en el árbol.

Por tanto, el nodo en la posición $T[i]$ es el padre de $\left\{ \begin{matrix} T[2 * i] \\ T[2 * i + 1] \end{matrix} \right\}$, que son posiciones en el vector.

El nodo en la posición $T[i]$ es el hijo del nodo $T[i \text{ div } 2]$ o $T[i \div 2]$.

Estas son la propiedad del montículo para los vectores. No haría falta representarlo con punteros, ya que lo haríamos con un vector en la que cada posición indica cual es el padre e hijo.

En resumen y para los vectores sería:

$$\left\{ \begin{matrix} T[i] \geq T[2 * i] \\ T[i] \geq T[2 * i + 1] \end{matrix} \right\} \quad \text{Hijos con respecto a padres.}$$

$$T[i] \leq T[i \text{ div } 2]. \quad \text{Padres con respecto a hijos.}$$

Consideramos montículos de máximos (padre es mayor o igual que el hijo), aunque hay también de mínimos, que es justo lo contrario.

Utilización: La principal utilidad de un montículo es la de estar permanentemente organizado de forma que nos proporcione de manera eficiente el elemento de mayor (o menor) valor, que en este caso es la raíz del árbol binario. A este estado se le conoce como **propiedad del montículo** y debe restaurarse después de cualquier modificación del montículo.

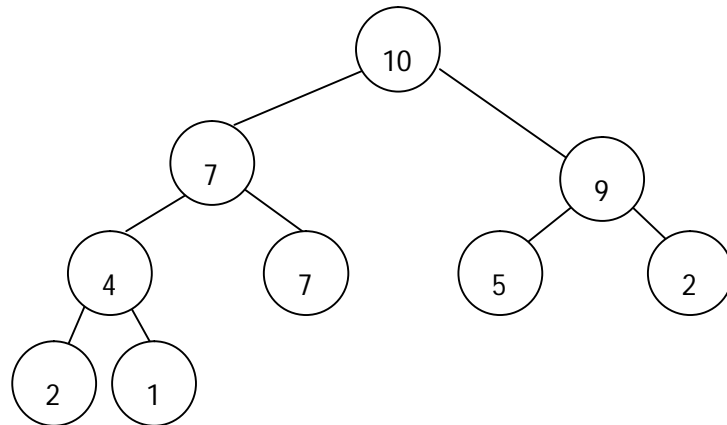
Un ejemplo típico de uso de montículos es la implementación de las *colas de prioridad*.

Insistimos en el asunto de las propiedades del montículo, ya que es básico su dominio. Pondremos unos ejemplos para verificar si el vector dado es montículo o no. Seguiremos este procedimiento:

1. Imaginemos que nos dan este vector:

10 7 9 4 7 5 2 2 1

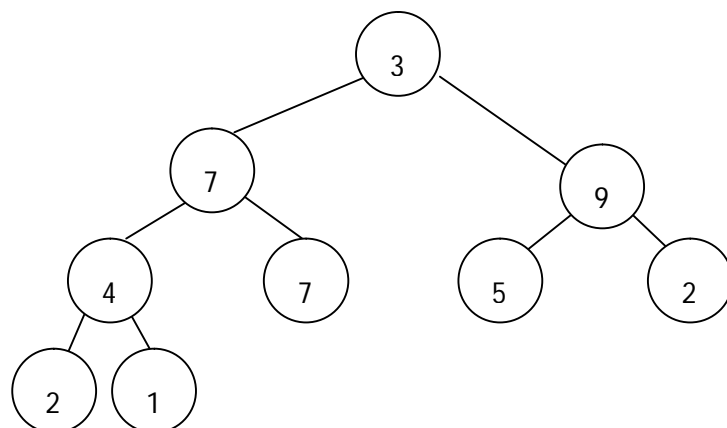
El árbol del mismo sería:



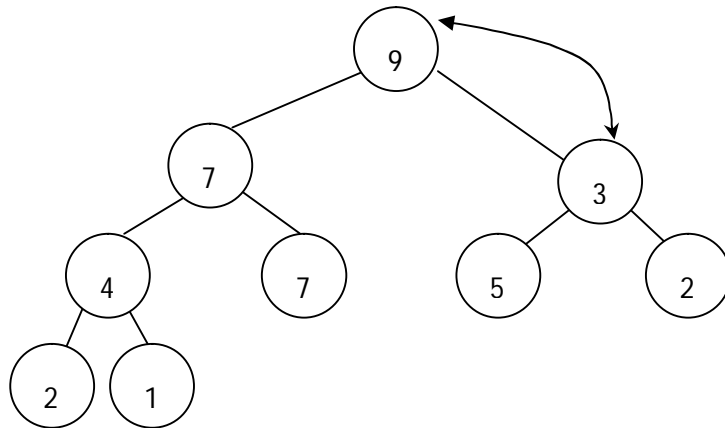
Vemos que cada padre es mayor o igual que sus hijos. Por tanto, es montículo.

2. Ahora nos da este otro:

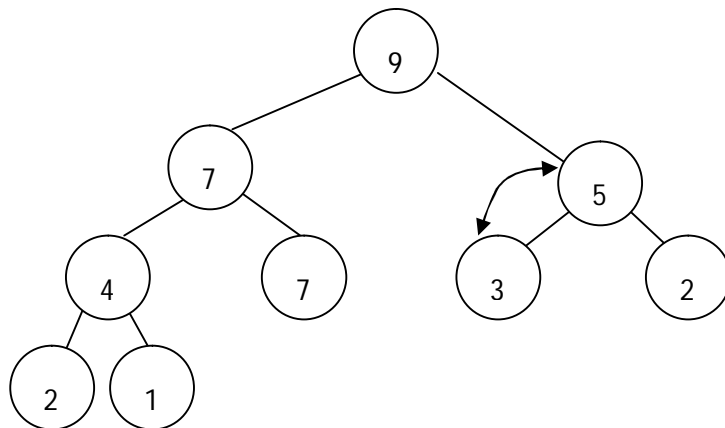
3 7 9 4 7 5 2 2 1



Observamos que el nodo 1 es menor que sus hijos, por lo que no es montículo. Para arreglarlo lo hundimos (de arriba abajo), para ello lo intercambiamos con el hijo mayor, que es 9, en este caso.

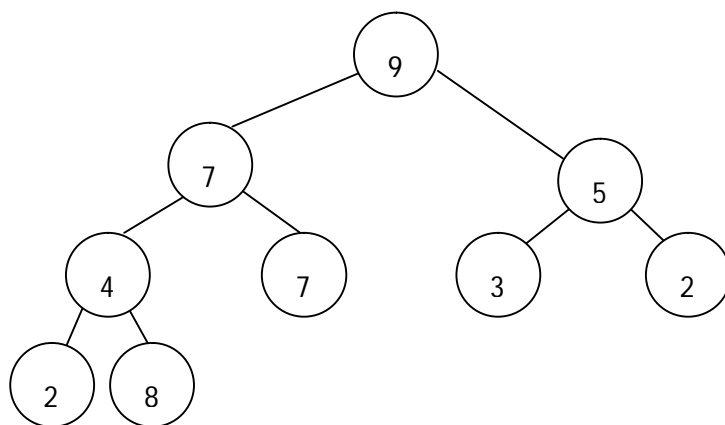


Sigue sin cumplirse la propiedad del montículo, por lo que hundimos de nuevo el nodo 3, intercambiándolo con el hijo mayor, quedando así:

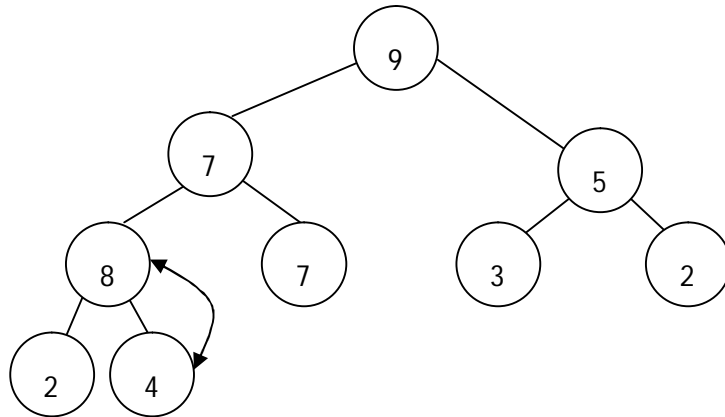


Con este último intercambio ya cumple la propiedad del montículo.

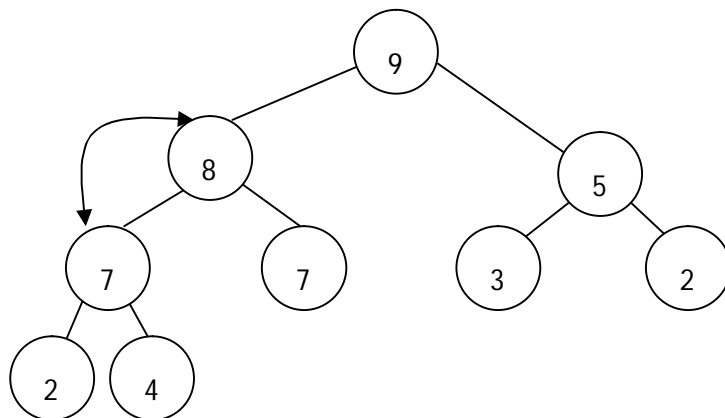
Agregamos nodo nuevo en el vector siguiente, nos fijamos que es la última posición y el valor es un 8:



Flotamos el nodo último para que cumpla la propiedad del montículo y lo intercambiamos con el padre:



A continuación, hay que hacer otro cambio para que cumpla de nuevo la propiedad del montículo:



Para hacer los dos cambios hay que hacer una única operación de hundir o flotar. Ya con estos cambios se cumple la propiedad del montículo.

Operaciones del montículo:

Hemos visto anteriormente una aplicación de flotar y hundir en esos ejemplos. Pasamos a escribir las operaciones, con los algoritmos y a explicarlos. Las más importantes y en las que se basaran el resto de algoritmos es el de **hundir** y **flotar**. Empezamos por la modificación de montículo, que usa ambos, que más abajo lo describiremos. Es **IMPORTANTE** el saber estos algoritmos (razonando y no de memoria), teniendo en cuenta la ***propiedad del montículo***.

procedimiento modificar-montículo ($T[1..n], i, v$)

- { $T[1..n]$ es un montículo. $T[i]$ recibe el valor v y se vuelve a establecer la propiedad del montículo. Suponemos $1 \leq i \leq n$ }
- $x \leftarrow T[i];$
- $T[i] \leftarrow v;$
- si $v < x$ entonces hundir (T, i);
- si no flotar (T, i);

Este procedimiento restaura la propiedad del montículo, o bien hundiendo o bien flotando el nodo. El coste de este algoritmo es el de hundir o flotar, que son $\theta(\log n)$.

procedimiento hundir ($T[1..n], i$)

- { Este procedimiento hunde el nodo i para establecer la propiedad del montículo en $T[1..n]$. suponemos que T seria un montículo si $T[i]$ fuera lo suficientemente grande. También suponemos que $1 \leq i \leq n$ }
- $k \leftarrow i;$
- repetir
 - $j \leftarrow k;$
 - { Buscar el hijo mayor del nodo j }
 - si $2 * j \leq n$ y $T[2 * j] > T[k]$ entonces $k \leftarrow 2 * j$
 - si $2 * j < n$ y $T[2 * j + 1] > T[k]$ entonces $k \leftarrow 2 * j + 1$
 - intercambiar $T[j]$ y $T[k]$
 - { si $j = k$, entonces el nodo ha llegado a su posición final }
- hasta que $j = k$

El coste es de $\theta(\log n)$

procedimiento flotar ($T[1..n], i$)

- { Este procedimiento flota el nodo i para establecer la propiedad del montículo en $T[1..n]$. suponemos que T seria un montículo si $T[i]$ fuera lo suficientemente grande. También suponemos que $1 \leq i \leq n$ }
- $k \leftarrow i;$
- repetir
 - $j \leftarrow k;$
 - si $j > 1$ y $T[j \div 2] < T[k]$ entonces $k \leftarrow j \div 2$
 - intercambiar $T[j]$ y $T[k]$
 - { si $j = k$, entonces el nodo ha llegado a su posición final }
- hasta que $j = k$

El coste es de $\theta(\log n)$, como pasa con el algoritmo de hundir un nodo.

Debido a su importancia recordamos *de nuevo* la **utilización** del montículo, ya que lo usaremos en muchas aplicaciones prácticas. Un **montículo** es una estructura ideal para hallar el mayor de un conjunto, para eliminarlo, para añadir un nodo nuevo o para modificar un nodo.

Una aplicación muy utilizada es la lista de prioridad dinámica (como dijimos antes *cola de prioridad*), en la que el valor del nodo de la prioridad del suceso

correspondiente, el suceso de prioridad más alta se encuentra siempre en la raíz del montículo y la prioridad de un suceso se puede modificar dinámicamente en cualquier momento.

Seguimos con más operaciones del montículo, no menos importantes que las anteriores:

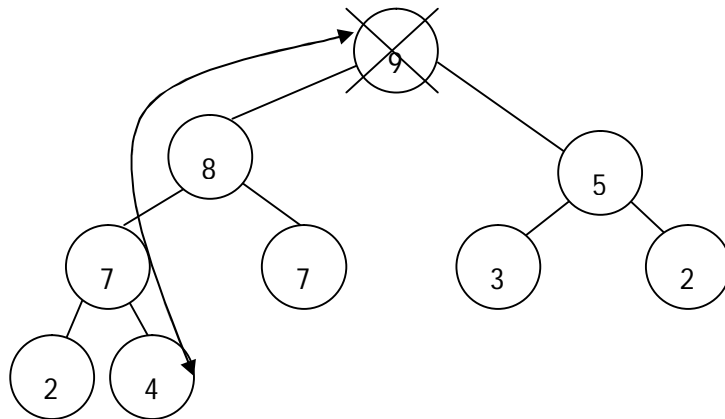
```
funcion buscar-max ( $T[1..n]$ )
{ Proporciona el mayor elemento del montículo  $T[1..n]$  }
devolver  $T[1]$ ;
```

Esta función devuelve un valor de un elemento en una matriz, que recordemos es una interpretación muy utilizada de los montículos. En este caso, al ser montículo de máximo, en la raíz estará el mayor elemento. Recapitulando teníamos que el coste de acceso al vector es constante, $O(1)$.

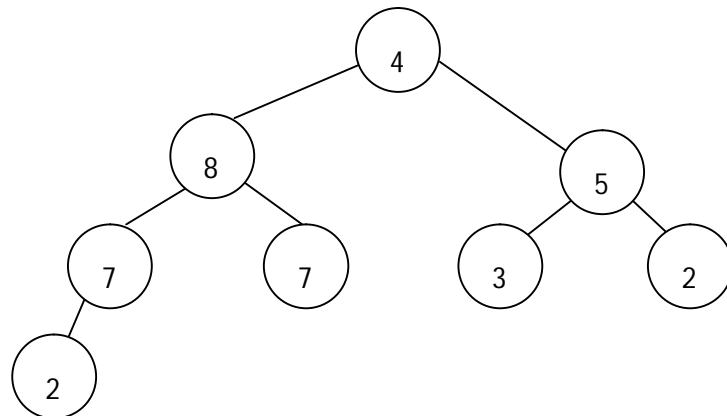
```
funcion borrar-max ( $T[1..n]$ )
{ Elimina el mayor elemento del montículo  $T[1..n]$  y restaura la
  propiedad del montículo en  $T[1..n-1]$  }
 $T[1] \leftarrow T[n]$ ;
hundir ( $T[1..n-1], 1$ );
```

Si queremos borrar el máximo (el primer elemento) del montículo tendremos que restaurar la propiedad del mismo, para eso hundimos la raíz hasta que se haga eso. Veremos paso a paso este algoritmo, para que así quede más claro:

1^{er} paso. Intercambio con el último elemento. Siguiendo con el ejemplo anterior tendremos:



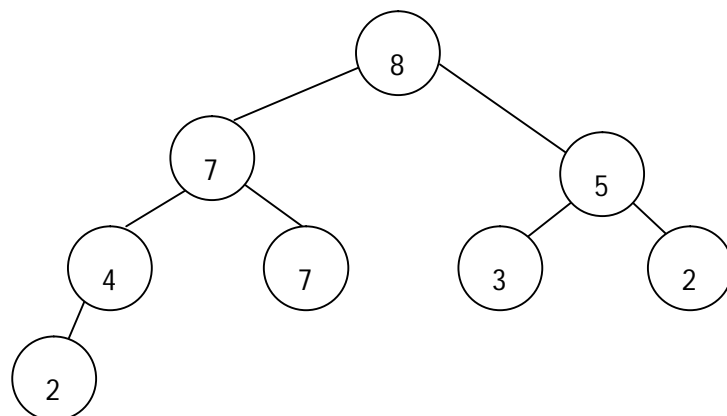
Quitamos el primer elemento e intercambiamos el último elemento con el primero, el que previamente hemos quitado. Por tanto, nos quedaría así:



2º paso. Hundimos (de arriba abajo, recuérdese como regla nemotécnica el del buzo y el agua) el primer elemento, que como vimos en los ejemplos anteriores no cumple la condición de montículo (de máximos). Nos evitamos hacer los pasos, aunque los describiremos, para saber porque el montículo será ese:

1. Intercambiamos 4 con 8, que es su hijo mayor ($T[1]$ con $T[2]$).
2. Intercambiamos 4 con 7 ($T[2]$ con $T[4]$).

En este segundo intercambio ya es montículo, por lo que ponemos como quedaría:



Un asunto importante es que estos intercambios (es decir, hundir dos veces dos nodos) se hace en una misma llamada hasta que acabe de recorrerse todo el montículo.

procedimiento añadir-nodo ($T[1..n], v$)

{ Añade un elemento cuyo valor es v y restaura la propiedad del montículo en $T[1..n - 1]$ }

$T[n + 1] \leftarrow v$;

flotar ($T[1..n + 1], n + 1$)

Si quisiéramos añadir un nodo habría que flotar, como resaltamos de nuevo aunque de tres saltos se hace en una llamada. El coste de nuevo lo determina *flotar*, que es $\theta(\log n)$.

procedimiento crear-montículo-lento ($T[1..n]$)
 { Este procedimiento transforma la matriz $T[1..n]$ en un montículo,
 aunque de forma más bien ineficiente }
 para $i \leftarrow 2$ hasta n hacer flotar ($T[1..i], i$)

Iríamos recorriendo el montículo desde la posición 2 hasta flotar (de arriba abajo) todos los elementos y que se restaure la propiedad del montículo.

Al ser un procedimiento lento, veremos otro más rápido, que es más eficiente. A continuación los compararemos:

procedimiento crear-montículo ($T[1..n]$)
 { Este procedimiento transforma la matriz $T[1..n]$ en un montículo }
 para $i \leftarrow \lfloor n/2 \rfloor$ bajando hasta 1 hundir (T, i)

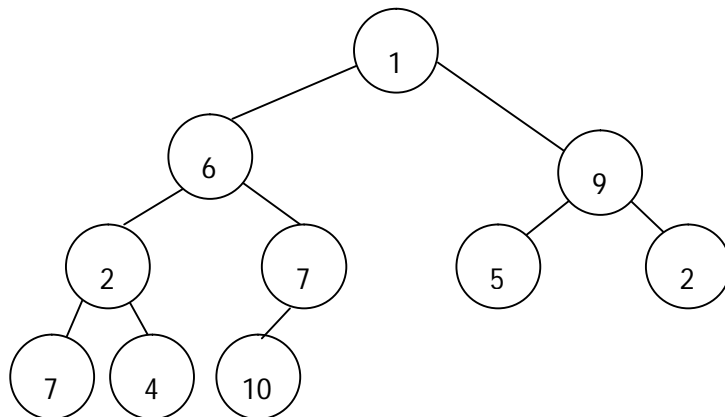
Observando ambos algoritmos vemos que mientras el primero flota haciendo un bucle mayor (de 2 a n) el segundo hunde de $\lfloor n/2 \rfloor$ bajando a 1, por lo que se reduce a priori el número de operaciones. Veremos este segundo y comprenderemos porque es más eficiente.

Ejemplo del segundo algoritmo, más eficiente:

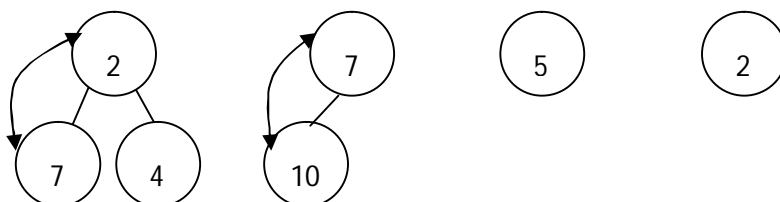
- Inicialmente nos dan la situación siguiente:

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

Pasamos este vector a un árbol binario (montículo):

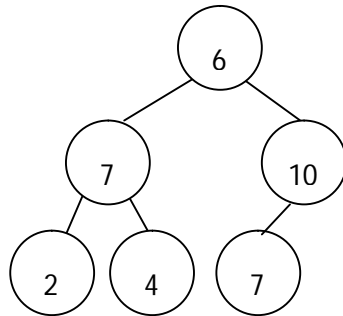


- Primer paso: Convertimos en montículos aquellos subárboles cuyas raíces se encuentran en el nivel 1. Recordamos que se intercambiarán con su hijo mayor.



Segundo paso: Los subárboles de nivel inmediatamente superior se transforman en montículos, hundiendo (de arriba abajo) una vez más sus raíces. Se cuentan con las modificaciones del paso anterior. En todos los niveles salvo el último, hundimos la raíz de dicho subárbol.

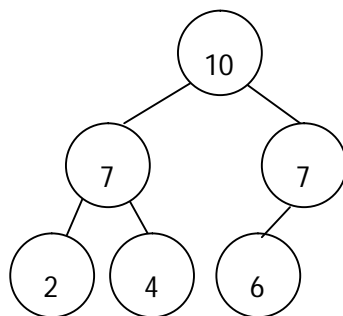
Empezaríamos por el subárbol derecho, ya que es el elemento con mayor índice. Observamos que está ordenado, por lo que continuamos por el subárbol izquierdo (menor índice, ojito) como sigue:



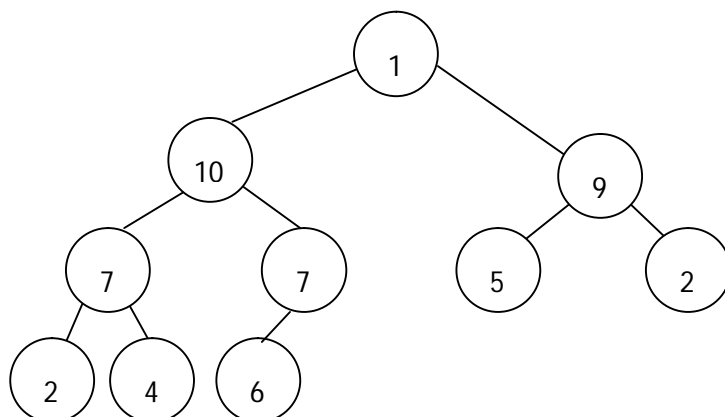
Como antes vemos los intercambios en el subárbol izquierdo que se han realizado:

1. Intercambiamos el nodo con valor 6 con el 10 (su hijo mayor).
2. Intercambiamos el nuevo nodo con valor 6 con el 7.

Quedará así:



- Último paso: Nos queda hundir la raíz, por lo que una vez hecho lo anterior, ya estamos dispuestos a hacerlo. Ya cumple la propiedad del montículo.



De nuevo, no pintaremos los grafos, pero diremos que hemos hecho (recordemos de nuevo que eso se hace en una misma llamada a hundir):

1. Intercambiamos nodo posición 1 (valor 1) con el posición 2 (valor 10).
2. Intercambiamos nuevo nodo valor 1 con valor 7 (posición 4).
3. Intercambiamos nodo 1 (posición 4) con valor 4 (posición 9).

Nuestro resultado en forma de matriz será:

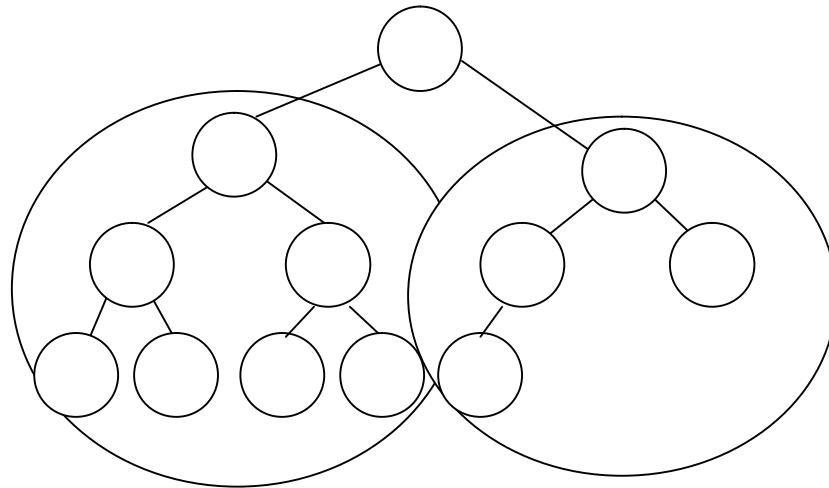
1	2	3	4	5	6	7	8	9	10
10	7	9	4	7	5	2	2	1	6

Para finalizar las operaciones del montículo analizaremos el coste del algoritmo de crear montículo. Queda decir que ambos dos tendrán coste lineal, lo único que les diferenciara es que la constante multiplicativa del “lento” es mucho mayor (hace más operaciones) que la del rápido.

Para analizar el coste tendremos esta afirmación: El algoritmo construye un montículo en tiempo lineal.

La **demonstración** es:

Sea $t(n)$ el tiempo requerido para construir un montículo de altura k como máximo. Para construir el montículo, el algoritmo transforma primero los dos subárboles asociados a la raíz en árboles de altura $k - 1$ como máximo:



Entonces, el algoritmo hunde la raíz por una ruta cuya longitud es k como máximo, lo cual requiere un tiempo $s(k) \in O(k)$ en el caso peor.

Tenemos, por tanto, la recurrencia siguiente:

$$t(k) \leq 2 * t(k - 1) + s(k).$$

siendo:

a: Número de llamadas recursivas = 2

b: Reducción del subproblema en cada llamada = 1

c * n^k: Coste de llamadas externas a la recursividad, que sería:

$$c * n^k = 1 \Rightarrow k = 0.$$

Recordemos cuales son los casos de la resolución de la recurrencia por sustracción:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

Estamos en el tercer caso, porque $a = 2$, por lo que el coste es $\theta(a^{n \text{ div } b}) = \theta(2^k)$.

Necesitamos saber el coste para el número de nodos, para ello teníamos $k = \lfloor \log n \rfloor$, sustituyendo tendremos $\theta(2^{\log_2 n}) = \theta(n)$. Se verá la demostración de este coste en los ejercicios del tema 7, el esquema de divide y vencerás.

Con esto queda demostrado que el coste de crear montículo es lineal, y como dijimos antes tiene más constante multiplicativa el algoritmo lento que el que hemos visto paso a paso.

Por último, para acabar el apartado de montículos veremos el algoritmo de ordenación por montículo (heapsort), cuyo algoritmo es el siguiente:

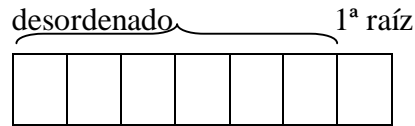
```
procedimiento ordenación por montículo (T[1..n])
{ T es la matriz que hay que ordenar }
crear-montículo (T);
para i ← n bajando hasta 2 hacer
    intercambiar T[1] y T[i]
    hundir (T[1..i - 1], 1)
```

Para analizar el coste realizaremos estos pasos:

1. Análisis del funcionamiento.
2. Análisis del coste propiamente dicho.

Analizaremos el funcionamiento como sigue:

- a) Pasamos el último elemento al primero, es algo así como quitar la raíz que veíamos antes.
- b) Hundimos la raíz hasta que se cumpla la propiedad del montículo. Esto quiere decir que ese último elemento ya estaría ordenado, por ser el mayor de todos los elementos.



- c) De nuevo pasamos el penúltimo elemento al primero (a la raíz). Luego hundimos estos elementos. Ya estarían las dos últimas posiciones ordenadas en el vector (montículo).
- d) Sucesivamente haremos esto, hasta ordenarlos. Nos fijamos que acaba en el segundo elemento, ya que no haría falta continuar más.

Una vez visto el funcionamiento, pasamos a **analizar el coste**, que será lo siguiente:

- Crear montículo (T): $\theta(n)$.
- Hacer el bucle "para": realiza n veces hundir la raíz a lo largo de un camino una longitud $\log(n)$, que es el coste en el caso peor. Por tanto, el coste es $\theta(n * \log(n))$.

Por lo visto anteriormente, el coste del algoritmo de ordenar montículo es $\theta(n * \log(n))$.

Recordemos que en algunas ocasiones usaremos un montículo invertido (o de mínimos, o la raíz es el mínimo) en el que el nodo interno es menor o igual que los valores de sus hijos. Nuestro montículo habitual es el montículo de máximos.

Al haber operaciones que no resultan adecuadas para manejar listas dinámicas de prioridad usaremos montículos binomiales, que los daremos en el siguiente apartado.

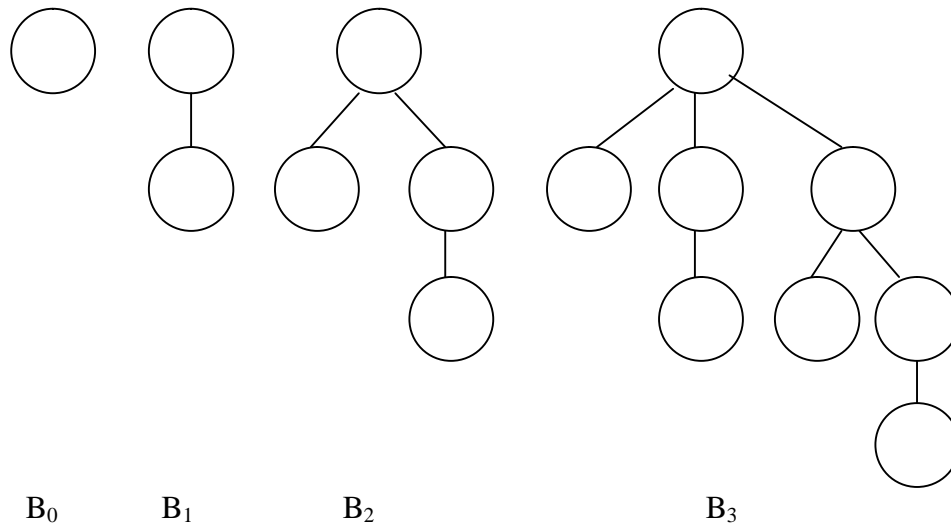
5.8. Montículos binomiales.

En un **montículo ordinario** que contenga n elementos, buscar el mayor de ellos requiere un tiempo que está en $O(1)$. Borrar el mayor elemento o insertar uno nuevo requiere un tiempo que está en $O(\log(n))$. Sin embargo, fusionar dos montículos que tengan entre los dos n elementos, requiere un tiempo que está en $O(n)$.

En un **montículo binomial**, la búsqueda del mayor elemento sigue necesitando un tiempo que está en $O(1)$ y el borrado del mayor elemento $O(\log(n))$, igual que antes. Sin embargo, la fusión de dos de estos montículos sólo requiere un tiempo en $O(\log(n))$ y la inserción de un nuevo elemento sólo requiere un tiempo en $O(1)$.

Definición de árbol binomial: El i -ésimo árbol binomial B_i con $i \geq 0$, se define recursivamente como aquél que consta de un nodo raíz con i hijos, en donde el j -ésimo hijo, $1 \leq j \leq i$, es a su vez la raíz de un árbol binomial B_{j-1} .

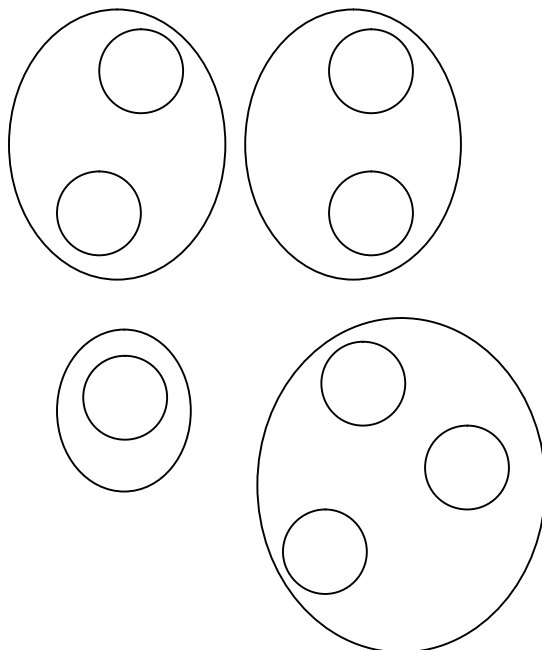
Ejemplo de árbol binomial:



5.9. Particiones.

Supongamos que se tienen N objetos numerados de 1 a N . deseamos agrupar estos objetos en conjuntos disjuntos, de tal manera que en todo momento cada objeto se encuentre exactamente en un conjunto.

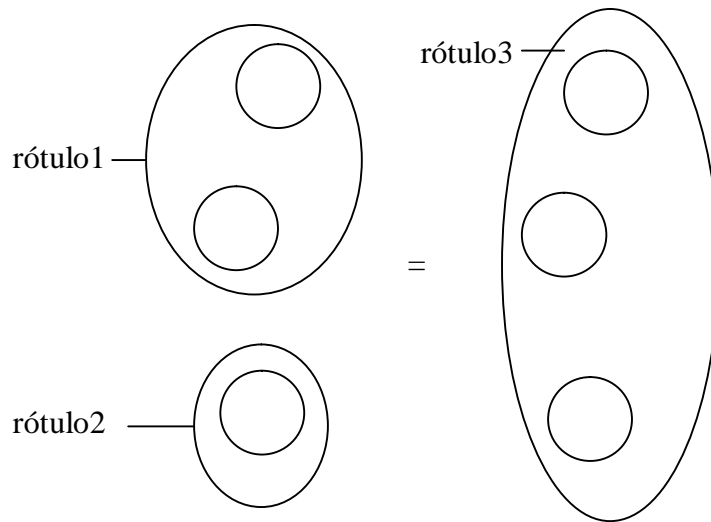
Ejemplo de particiones:



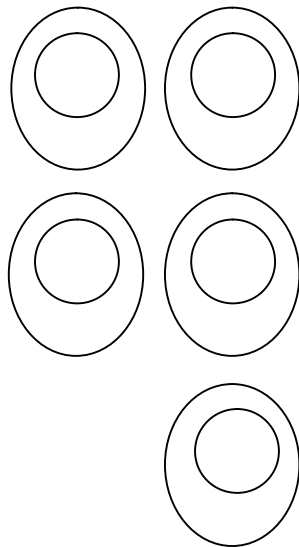
Cada conjunto lleva su rótulo asociado.

Operaciones:

- Buscar: Busca en qué conjunto está contenido un elemento. Devolvemos el rótulo de este conjunto.
- Fusionar: Se le pasan dos rótulos y construye un único conjunto con un rótulo nuevo. Ej. Fusionar (rótulo1, rótulo2).

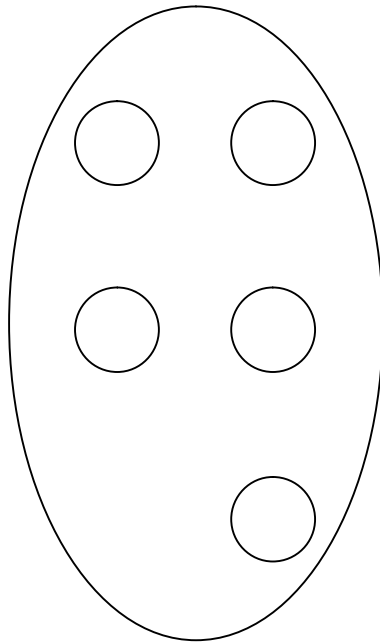


Se nos da que inicialmente los N objetos se encuentran en N conjuntos diferentes, cada uno contiene exactamente un objeto:



Pretendemos que queden todos los objetos en un único conjunto mediante operaciones de búsquedas y fusiones, para ello necesitamos **N búsquedas y $N - 1$ fusiones**. Esto es importante, ya que de ello depende lo que a continuación daremos en este apartado.

La partición resultante quedaría así:



Tendremos distintas **implementaciones**, que en ocasiones serán mejoras de la anterior. Es decir, aunque asumimos que son implementaciones distintas en el fondo son mejoras de las anteriores.

NOTA: Tomaremos la cota superior de todas las funciones al analizar el coste, ya que en principio tendremos el mismo resultado que al poner el coste exacto (recordemos que es la cota superior y la cota inferior).

Las **implementaciones** serán:

1. La primera implementación: El rótulo de un conjunto es el menor de sus elementos.

La notación que usaremos será $\text{conjunto}[i]$, que es el rótulo que contiene al elemento i .

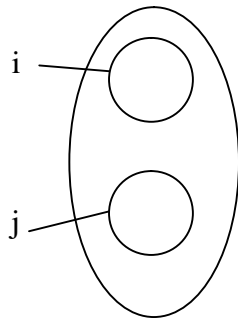
Operaciones:

```
funcion buscar1 (x)
{ Busca el rótulo del conjunto que contiene x }
devolver conjunto[x];
```

Esta función tiene coste $O(1)$, al devolver una posición en un array.

```
procedimiento fusionar1 (a,b)
{ Fusiona los conjuntos rotulados como a y b, suponemos que
   $a \neq b$  }
 $i \leftarrow \min(a, b)$ ;
 $j \leftarrow \max(a, b)$ ;
para  $k \leftarrow 1$  hasta N hacer
  si  $\text{conjunto}[k] = j$  entonces  $\text{conjunto}[k] \leftarrow i$ 
```

Al fusionar los dos conjuntos tendremos que saber cuál de los dos rótulos a ó b es el menor y le asignamos el valor al nuevo rótulo después de fusionarlo. Gráficamente tendríamos algo así tras fusionar:



El coste máximo será $O(N)$, determinado por el bucle “para”.

Según las conclusiones anteriores el coste de 1 búsqueda es $O(1)$ y el de 1 fusión es $O(N)$, por lo que para nuestro problema de N búsquedas y $N - 1$ fusiones tendremos:

$$\begin{array}{lll} N \text{ búsquedas} & O(1) & \text{—} O(N) \\ N - 1 \text{ fusiones} & O(N) & \text{—} O(N^2) \end{array}$$

Aplicando la regla del máximo, visto en temas anteriores, tendremos que el coste del problema de esta implementación es $O(N^2)$.

2. La segunda implementación: Cada conjunto es un árbol, en el cual cada nodo contiene un puntero a su padre, como nodoarbol2. Tendremos este convenio:

Si $\text{conjunto}[i] = i \Rightarrow i$ es rótulo de un conjunto y raíz.
 Si $\text{conjunto}[i] = j \neq i \Rightarrow j$ es el padre de i en algún árbol.

Ejemplo:

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

Buscamos conjunto 1: Conjunto 1 es una raíz de un conjunto, es su rótulo por ser $\text{conjunto}[i] = i$.

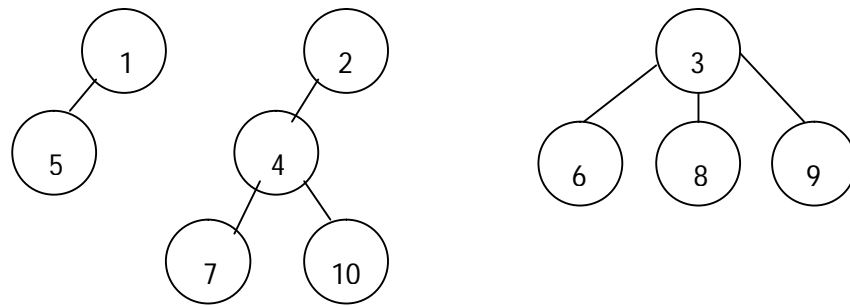
Buscamos conjunto 2: Ocurre igual que el 1, es su rótulo.

Buscamos conjunto 3: Igual que los anteriores.

Buscamos conjunto 4: Observamos que $\text{conjunto}[4] = 2$, por lo que el padre de 4 es el 2.

Y así sucesivamente, hasta finalizar el recorrido.

En forma de árbol, el resultado nos quedaría así:



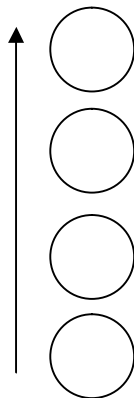
Las operaciones de buscar y fusionar serán:

```

funcion buscar2 (x)
{ Busca el rótulo del conjunto que contiene x }
  r ← x;
  mientras conjunto[r] ≠ r hacer r ← conjunto[r]
  devolver r
  
```

Iremos buscando en todos los conjuntos hasta encontrar el padre para llegar a la raíz.

En el caso peor, tiene coste lineal ($O(N)$), porque llegaría a ser un árbol de altura n, buscando el último elemento del árbol.

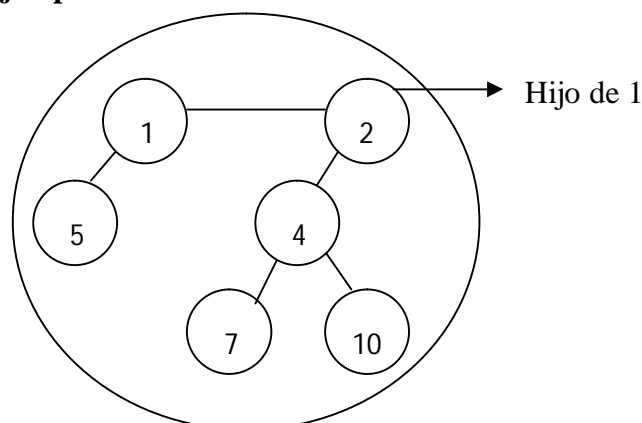


```

procedimiento fusionar2 (a,b)
{ Fusiona los conjuntos rotulados como a y b, suponemos que
  a ≠ b }
  si a < b entonces conjunto[b] ← a
  si no conjunto [a] ← b
  
```

Bastaría con cambiar un conjunto y su rótulo. Entonces, el coste sería constante, $O(1)$.

Ejemplo:



Con estas modificaciones quedaría así el nuevo vector:

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

Según las conclusiones anteriores el coste de 1 búsqueda es $O(N)$ y el de 1 fusión es $O(1)$, por lo que para nuestro problema de N búsquedas y $N - 1$ fusiones tendremos:

$$\begin{array}{lll} N \text{ búsquedas} & O(N) & \text{—} O(N^2) \\ N - 1 \text{ fusiones} & O(1) & \text{—} O(N) \end{array}$$

Como hemos visto antes, el coste según la regla del máximo es $O(N^2)$.

3. La tercera implementación: Se mejora, ya que en vez de tener como rótulo el menor de los elementos en cada paso. Tenemos el de menor altura, que sería el hijo del de mayor altura. No se varía buscar2, pero sí fusionar2. Ambas funciones serían:

funcion buscar2 (x)

{ Busca el rótulo del conjunto que contiene x }

$r \leftarrow x$;

mientras conjunto[r] \neq r hacer $r \leftarrow$ conjunto[r]

devolver r

procedimiento fusionar3 (a, b)

{ Fusiona los conjuntos rotulados como a y b, suponemos que $a \neq b$ }

si altura[a] = altura[b] entonces

altura[a] \leftarrow altura[a]+1

conjunto[b] \leftarrow a

si no

si altura[a] > altura[b] entonces

conjunto[b] \leftarrow a

si no conjunto[a] \leftarrow b;

Al cabo de una secuencia arbitraria de búsquedas y fusiones, con árbol de k nodos tiene una altura máxima $\lceil \log k \rceil$.

Para ejecutar una secuencia arbitraria de n operaciones buscar2 y $N-1$ operaciones fusionar3 comenzando a partir de la situación inicial es $O(n * \log n)$. En el libro nos comentan como coste exacto, como pusimos anteriormente tomaremos cota superior, porque es lo mismo.

4. La cuarta implementación: Añadimos **compresión de caminos** (cuidado que es de comprimir, no de comprender).

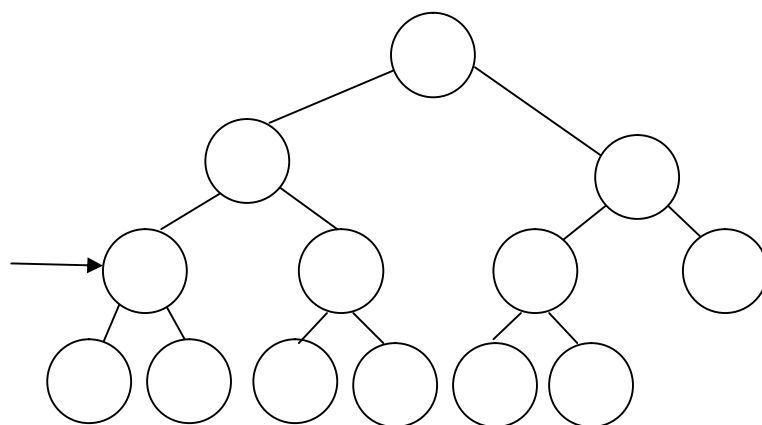
Cuando se está intentando determinar el conjunto que contiene un cierto elemento x ; primero se recorren las aristas del árbol que suben desde x hasta la raíz. Una vez que se conoce la raíz, podemos recorrer una vez más las mismas aristas, modificando esta vez cada nodo encontrado por el camino de tal manera que su puntero señale ahora directamente a la raíz.

Modificaremos buscar2 como sigue:

```

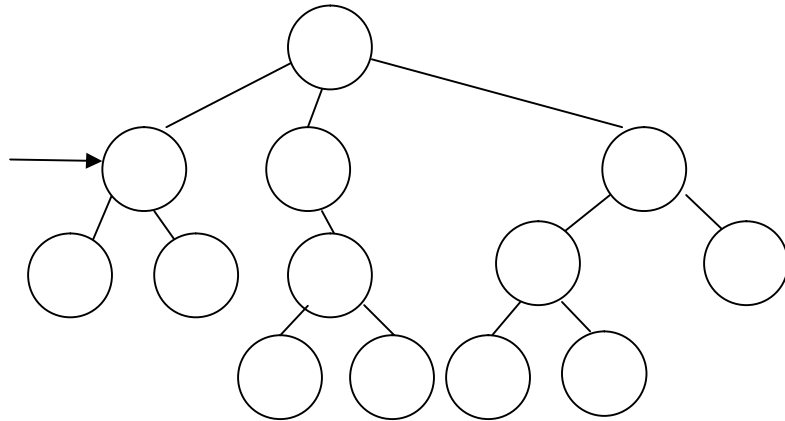
funcion buscar3 (x)
{ Busca el rótulo del conjunto que contiene el elemento x }
 $r \leftarrow x$ ;
mientras conjunto[r]  $\neq$  r hacer  $r \leftarrow$  conjunto[r]
{ r es la raíz del árbol }
Mientras  $i \neq r$  hacer
     $j \leftarrow$  conjunto[i]
    conjunto[i]  $\leftarrow$  r
     $i \leftarrow j$ ;
devolver r
    
```

Explicamos brevemente la compresión de caminos:

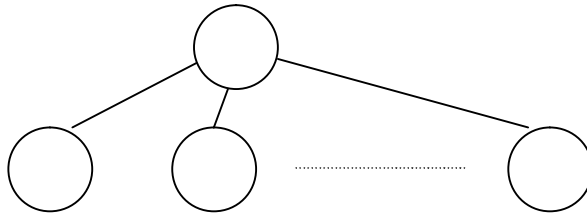


Encontramos ese nodo y vemos que el nodo padre sería el del nivel superior y el padre de este es el nodo raíz.

Se cambiaría el puntero de este nodo:



Si hiciéramos muchas compresiones de caminos, o lo que es igual, muchas búsquedas el árbol llegaría a algo así:



Llegaría la búsqueda casi a coste lineal $O(N)$, en vez de $O(\log n)$.