CAPITULO 2: ESTRUCTURA DE DATOS AVANZADAS

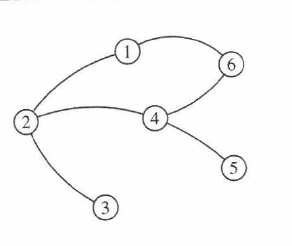
1. **Grafos.**

Un grafo es una colección de nodos o vértices(objetos) unidos por líneas o aristas (relaciones entre objetos). Se puede definir como G=<N, A>, donde N es un conjunto finito de vértices o nodos y A es un conjunto finito de líneas o aristas. Los grafos permiten modelar problemas en los que existe una relación relevante entre los objetos que intervienen.

* 1. **Definiciones básicas**

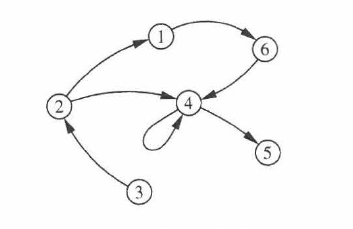
**NODO NO DIRIGIDO**

* Sus **aristas** no tienen sentido.
* La unión de dos vértices se representa entre paréntesis. Ej.
* La representación podría verse como:
* El grado de un vértice es el número de arista que salen o entran en él. En el caso de un bucle cuenta por 2.
* Un camino en un grafo no dirigido es una secuencia de aristas consecutivas.
* nº máx. aristas (no se cuentan los bucles) = n(n-1)/2.



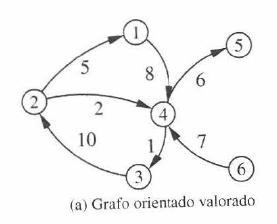
**GRAFO DIRIGIDO**

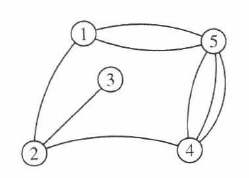
* Sus **arcos** tienen sentido.
* La unión de dos vértices se representa entre corchetes angulares, es importante el orden en el que se representan. Ej.
* La representación formal sería:
* A la arista se le denomina bucle o lazo.
* Grado de entrada, número de aristas que entran a ese vértice, y el grado de salida que es el número de arcos que salen de él.
* Un camino es una secuencia finita de arcos entre dos vértices
* Nº máx de aristas aristas (no se cuentan los bucles) = n(n-1)

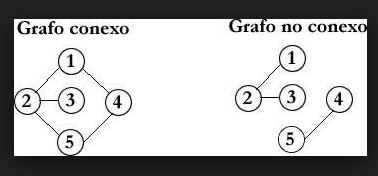


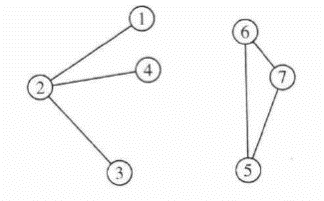
COSAS COMUNES EN GRAFOS DIRIGIDOS Y NO DIRIGIDOS

* Se dice que dos nodos son adyacentes si siendo distintos existe una arista o arco que los une.
* Longitud del camino es el número de aristas que contiene.
* Un camino simple es aquel que NO usa más de una vez la misma arista.
* Un camino complejo es aquel que usa más de una vez la misma arista.
* Un ciclo o circuito es un camino simple que empieza y termina en el mismo vértice.
* Dos nodos o vértices están conectados si existe un camino que los une.
  1. **Tipos de grafos.**
* **Grafo nulo:** Un grafo que no tiene vértices.
* **Grafo acíclico:** grafo que no contiene ciclos. Por ejemplo, los árboles son acíclicos.
* **Grafos etiquetados o valorados:** son aquellos en los que sus aristas o vértices tiene un nombre, valor o descripción asociada. Consiste en la tripleta: . Donde P es una función que a cada arista de A le asigna un valor de un cierto tipo. Por ejemplo, en los anteriores grafos hemos visto que cada vértice tiene un número asociado, se pueden interpretar como su nombre o identificador.



* **Grafo simple:** entre cada par de vértices existe a lo sumo una arista.
* **Multígrafo:** Si no es grafo simple. AL menos un vértice tiene más de una arista.
* **Sub-grafo:** Un sub-grafo de un grafo G es un grafo cuyo conjunto de vértices es un subconjunto del de G, cuyo conjunto de aristas es un subconjunto del conjunto de las aristas de G.
* Grafo **no dirigido conexo**: si para cualquier par de índices distintos existe un camino que los contiene.

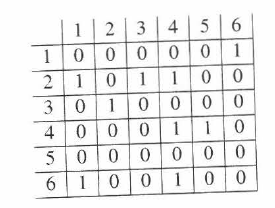
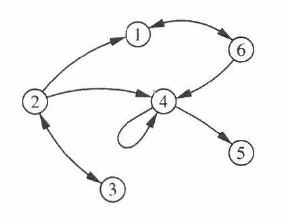


* Un grafo **dirigido** es **débilmente conexo** si al reemplazar todas sus aristas dirigidas por aristas no dirigidas resulta un grafo conexo.
* Un grafo **dirigido** es **conexo**: si contiene un camino entre cualquier par de vértices
* Un grafo **dirigido** es **fuertemente conexo** si para cualquier par de vértices (n y m) existe un camino que une n y m en ambas direcciones, es decir, que existe un arco desde “n” como origen hasta “m” como destino e inversa.
* Denominamos componente conexa de un grafo a un sub-grafo conexo maximal (G’), es decir, que no existe otro grafo conexo G’’ tal que G’ sea subconjunto de G’’.
* Denominaremos componente fuertemente conexa, a un subgrafo maximal fuertemente conexo de un grafo dirigido que no es fuertemente conexo.
* **Árbol libre:** es un grafo acíclico, conexo y no dirigido, estos árboles no tienen raíz y los hijos no están ordenados, para obtener un árbol general hay que seleccionar un nodo como raíz y establecer algún orden entre los hijos de cada nodo
  1. **representación de los grafos.**

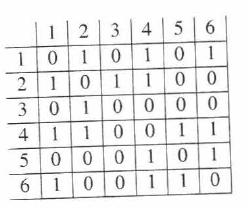
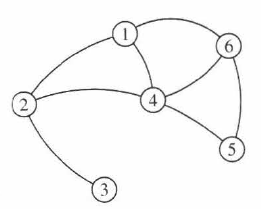
Hay diversas opciones para representar un grafo, en este texto se presenta las dos más comunes:

**MATRIZ DE ADYACENCIA**

Se a un grafo con n vértices. La matriz de adyacencia asociada a G, MA, es una matriz cuadrada de elementos tal que si la arista ( y ] = 0 si no existe tal arista en G. Los valores 0 y 1 se pueden sustituir por valores booleanos. Por ejemplo:



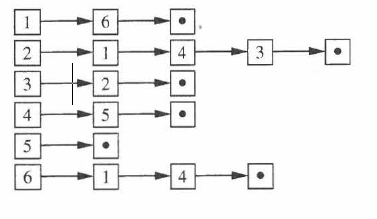
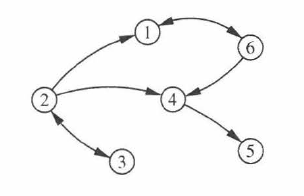
La matriz de adyacencia de un grafo no dirigido es simétrica, por lo que con la matriz triangular superior o inferior es suficiente. Por ejemplo:



**Un grafo valorado o etiquetado** se puede representar mediante una matriz de adyacencia si en lugar del valor 1 (o cierto) para representar la existencia de una arista se utiliza su valor o etiqueta. Si el cero no es una de las etiquetas se puede utilizar para representar la no existencia de una arista, en caso contrario se deberá utilizar un carácter que no pertenezca al conjunto de posibles valores.

**LISTAS DE ADYACENCIA**

Son un array de n listas, siendo n el número de nodos, por lo que contiene tantas listas como nodos hay en el grado. Las listas del nodo i tendrá tantos elementos como nodos adyacentes tenga el nodo i. Dentro de cada lista, los nodos no están ordenados. El acceso a los nodos adyacentes a uno dado se traduce en acceder y recorrer la lista asociada a dicho nodo. En el caso de grafos etiquetados las listas deberían incluir un campo adicional para almacenar la etiqueta asociada a cada arista. Veamos el siguiente ejemplo:



**FUNCIONES DE MANIPULACION DE GRAFOS**

Se muestran los prototipos de funciones que implementan las operaciones básicas de manipulación de grafos:

* **Crear grafo:** devuelve un grafo vacío.
* **Añadir arista:** añade una arista entre los vértices u y v y le asigna el peso(etiqueta) p.
* **Añadir vértice:** añade el vértice v al grafo g.
* **Borrar arista:** elimina la arista entre los vértices.
* **Borrar vértice**: borra el vértice v al grafo g y todas las aristas que partan o lleguen a él.
* **¿Es adyacente?:** comprueba si dos vértices son adyacentes.
* **Adyacentes**. Devuelve una lista con los vértices adyacentes a un determinado vértice.
* **Etiqueta:** Devuelve la etiqueta o peso asociado a la arista que une los vértices.

|  |  |  |
| --- | --- | --- |
|  | **Matriz de adyacencia** | **Lista de adyacencia** |
| Crear grafo |  |  |
| Añadir arista |  |  |
| Añadir vértice |  |  |
| Borrar Arista |  |  |
| Borrar Vértice |  |  |
| ¿Adyacente? |  |  |
| Adyacentes |  |  |
| Etiquetas |  |  |

**COSTES DE GRAFOS**

* n es el número de nodos y a es el número de aristas.
* Si el grafo tiene pocas aristas las listas de adyacencia resultan menos costosas en espacio, sin embargo, cuando “a” se acerca a el coste del espacio es del mismo orden. En el caso de que suceda lo último, es más conveniente utilizar una matriz de adyacencia teniendo en cuenta que es más sencilla de trata ya que no utiliza punteros.
* En cuanto al coste de tiempo dependiendo de que operaciones se necesiten será más apropiado una representación que otra.
* En general, para grafos dispersos (con pocas aristas) se utilizan las listas de adyacencia, para grafos densos, con muchas aristas, la matriz de adyacencia resulta la más apropiada.
  1. **Recorrido de grafos.**

Vamos a ver dos maneras de recorrer los grafos, aplicables tanto a grafos dirigidos como no dirigidos:

* **Recorridos en profundidad:** Inicialmente se marcan todos los nodos como no visitados y se selecciona un nodo “u” como punto de partida. A continuación, se marca como visitado y se accede a un nodo no visitado “v “adyacente al nodo “u”. Se procede recursivamente con el nodo “v”. El recorrido termina cuando todos los nodos están marcados como visitados. Sería el equivalente a pre-orden en un árbol, es decir, primero la raíz, luego los hijos.

Vamos a ver como se implementaría este recorrido con pseudo-código (más en apéndice A del libro).

**tipo** Vector = matriz[0…n] de booleano

**fun** RecorridoProfundidad(G = )

**var**

visitado: Vector

v: nodo

**fvar**

**para** **cada** v N **hacer**

visitado[v]<- falso

**fpara**

**para** **cada** v N hacer

**si**¬ visitado[v] **entonces**

RecProdfundidadRecursivo(v, visitado);

**fsi**

**fpara**

**ffun**

**fun** RecProdfundidadRecursivo(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**

este recorrido se puede realizar tanto en grafos dirigidos como no dirigidos, la única diferencia es lo que se entiende por adyacente en uno u otro tipo de grafo. Si el grafo es dirigido, el nodo w es adyacente al nodo u si existe el arco .

También se puede representar de forma iterativa, con una estructura de datos Pila LIFO con las operaciones del Tipo de Datos Abstracto (TA) pila. El cálculo en costes es similar a la versión recursiva.

**Coste de recorrido en profundidad**: Si consideramos que n es el número de nodos y a es el número de aristas de un grafo, en su recorrido en profundidad sólo se hace una llamada a *RecProfundadRecursivo* por cada nodo, por lo tanto, el método recursivo se ejecuta “n” veces.

Si la representación es mediante una matriz de adyacencia, los adyacentes a un nodo se obtienen recorriendo una fila de la matriz n x n, por tanto, el coste de la búsqueda en profundidad será O(n^2).

Si la representación es mediante una lista de adyacencia, para obtener los adyacentes a un nodo habrá que recorrer su lista de adyacentes, el tiempo estaría en O(n+a).

* **Recorrido en amplitud o anchura:** inicialmente se marcan todos los nodos como no visitados y se selección un nodo “u” como punto de partida. A continuación, se marca como visitado y se visitan todos los nodos no visitados adyacentes al nodo “u”, a continuación, se hace lo mismo con cada uno de los nodos adyacentes recién visitados. Se puede considerar que es un recorrido por niveles.

El recorrido en anchura se emplea cuando hay que realizar una exploración parcial de un grafo infinito o potencialmente muy grande, y también para calcular el camino más corto desde un punto del grafo hasta otro. Este recorrido no es de naturaleza recursiva, se necesita un cola para poder realizarla.

**fun** RecAnchura (v: nodo, visitado: Vector)

**var**

u,w : nodo

Q:TCola

**Fvar**

Q <- Cola vacía

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**

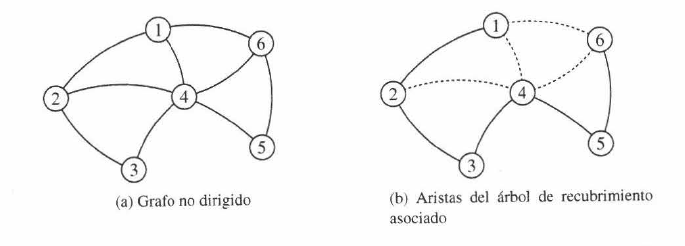
**ffun**

**Nota: la llamada a este método se realiza de manera similar a la expuesta en el recorrido de profundidad.**

**Coste de recorrido en anchura:** sucede lo mismo con el recorrido en profundidad: O(n^2) para el grafo que se presenta en forma de matriz adyacencia, O (n + a) en forma de lista de adyacencia.

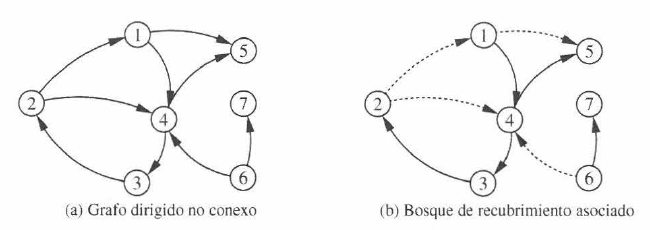
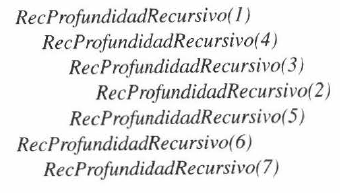
* 1. **Árboles de recubrimiento.**

Los recorridos en profundidad o en anchura de **un grafo conexo** le asocian un árbol de recubrimiento. Así, las aristas del grafo se dividen en dos conjuntos: las que pertenecen al árbol de recubrimiento y las que no, las que no pertenecen son las que no se usan en el recorrido del grafo, el nodo inicial del recorrido es la raíz del árbol.

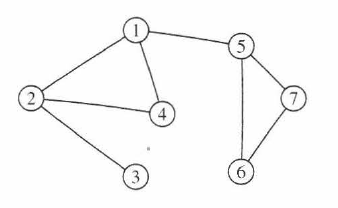


En la figura (b) las aristas en línea discontinuas representan las que no se han utilizado en el recorrido en el recorrido en profundidad sobre la figura (a), debido a que conducían a nodos que ya estaban visitados. Las aristas en línea continua son las que sí se han utilizado en el recorrido, en el sentido de que conducían a nodos sin visitar, son las que forman parte del árbol de recubrimiento.

Si el **grafo no es conexo**, el recorrido en profundidad le asocia un bosque de árboles, uno por cada componente conexa al árbol. Por ejemplo: En el siguiente grafo tendríamos 2 componentes conexas: una formada por los nodos [1,2,3,4,5] y la otra por los nodos [6,7]. Si se hace un recorrido en profundidad tomando el nodo 1 como origen, se observa que se tiene un bosque de recubrimiento formado por 2 árboles, uno por cada componente conexa. Gráficamente hablando, esto se traduce como:



Según en el orden en que se examinen los adyacentes pueden producirse unos u otros árboles o bosques de recubrimiento según el orden en el que se examinen los adyacentes.

* 1. **Puntos de articulación.**

Dado un grafo conexo, un nodo “u” es un punto de articulación si al eliminar ese nodo “u” y todas sus aristas, el grafo deja de ser conexo. Por ejemplo: en la imagen de la derecha los nodos 1,5 y 2 son puntos de articulación, dado que, si los eliminamos, el grafo deja de ser conexo.

Un grafo conexo sin puntos de articulación se llama bi-conexo.Encontrar los puntos de articulación de un grafo es una tarea básica en problemas de conectividad de grafos. Un grafo tiene conectividad k si la eliminación de k-1 nodos cualesquiera no lo desconecta. Un grafo biconexo tiene al menos conectividad 2. Cuanto mayor sea el valor de “k”, es decir, su conectividad, será más resistente a posibles fallos en los nodos.

La búsqueda de los puntos de articulación de un grafo se basa en el recorrido en profundidad. En el caso de que no se encuentren se concluirá que el grafo es bi-conexo. Los pasos para hallar los puntos de articulación de un grafo son los siguientes:

1. Realizar el recorrido en profundidad del grafo numerando los nodos según avanza en el recorrido. Como resultado tendremos el grafo inicial G, un árbol de recubrimiento asociado, AR, junto con el número de orden (es decir, el primer nodo que se recorre tiene el número 1 asociado, el segundo el 2, así sucesivamente) que le asigna el recorrido a cada nodo.
2. Recorriendo el árbol de recubrimiento en post-orden (primero izq, luego derecho, luego raíz) se calcula para cada nodo visitado el valor bajo[v], el cual, es el valor mínimo de los tres valores siguientes:
   * numOrden[v]
   * numOrden[w] para cualquier w tal que haya una arista de retroceso (v, w) en G que no esté en AR.
   * Bajo [x] para cualquier hijo x de v.
3. Se calculan los puntos de articulación de la siguiente manera:
   1. Si la raíz del AR tiene más de un hijo, ese nodo es un punto de articulación.
   2. Cualquier otro nodo v es un punto de articulación si tiene un hijo w tal que bajo[w] >= numOrden[v].
   3. Se calcula los puntos de articulación de la siguiente manera, si un nodo no tiene hijos en un árbol de articulación, **este nodo no puede ser un punto de articulación**.
   4. **Ordenación topológica de un grafo dirigido acíclico(gda)**.

Un gda es un grafo dirigido que no tiene ciclos. Los gda permiten modelar de manera natural ciertos problemas cuya resolución requiere la realización de una serie de tareas en un orden especifico. Por ejemplo, cuando haya que realizar una serie de tareas de forma que una no puede empezar hasta que otra u otras acaben.

En un gda la existencia del arco indica que el nodo “u” precede al nodo “v” en una ordenación lineal. Una ordenación topológica de una gda es una secuencia válida para las tareas representadas y todo gda tienes al menos una. Uno de los algoritmos que permite calcular un orden topológico de una gda es el algoritmo de Kahn. Expliquémoslo detalladamente:

* Debe haber al menos un nodo sin arcos entrantes.
* Se ejecuta la siguiente función, la cual, función devuelve un array que indica el orden topológico para el grafo de entrada

**tipo** vector **=** matriz[identNodo\_1 … identNodo\_n]de naturales

**fun** OrdenTopologicoKahn(G = grafo)

**var**

u,v: nodo

Q : TCola

Orden: Vector

**Fvar**

Q <- {uN: Gradoentrante (A,u) = 0} /**mete en una cola nodos con grado . entrante cero.**

i <- 1

**mientas** ¬ Vacia(Q) **hacer**

u <- Primero(Q)

Desencolar(u,Q)

orden[u] <- i

i <- i + 1

**para** **cada** (u,v) **hacer**

A <- A \{(u,v)}

**si** GradoEntrante(A,v) = 0 **entonces**

Encolar(v,Q)

**fsi**

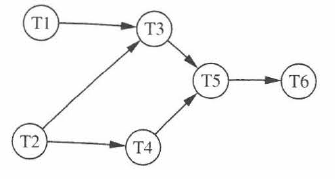
**fpara**

**fmientras**

**dev** orden

**ffun**

Nota: El coste de este algoritmo está en , siendo n el número de nodos y “a” el número de aristas. Veamos un ejemplo:



Este grafo representa la planificación de tareas. Aplicando la función anterior y empezando por el nodo T1, el array orden reflejaría el siguiente orden topológico:

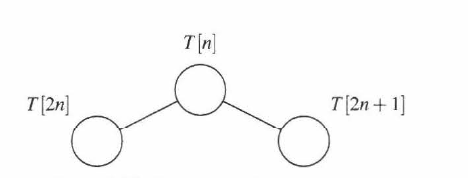
* 1. **Camino más corto desde la raíz a cualquier otro nodo.**

Se define como la distancia más corta entre los vértices “u” y “v” de un grafo G como el mínimo número de aristas en cualquier camino entre “u” y “v” en ese grafo G. Sea y sea el árbol de recubrimiento asociado al recorrido en anchura de G. **la longitud del camino desde la raíz hasta un nodo cualquier w en ARA, coincide con la longitud del camino más corto desde la raíz hasta w en G.**  Ejemplo: Problema de atravesar un laberinto.

1. **Montículos.**

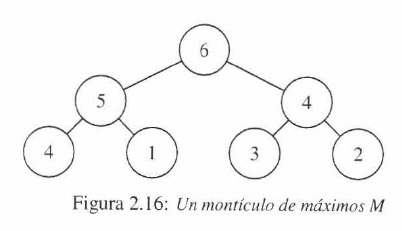
Los montículos son un tipo especial árbol binario (cada nodo tiene como máximo 2 hijos) que se implementan sobre vectores con las siguientes propiedades:

* Es un árbol balanceado(AVL) y completo (nodos internos siempre con dos hijos) con la posible excepción de un único nodo cuanto el número de elementos es par.
* Cada nodo contiene un valor mayor o igual que de sus nodos hijos (montículo de máximos), o menos o igual (montículo de mínimos). A está propiedad se le denomina **propiedad de montículo,** la cual, permite tener en la cima del montículo (el nodo raíz) el elemento mayor (o menor si se trata de un montículo de mínimos), siendo está la utilidad fundamental del montículo.
* Los nodos de profundidad k (longitud camino raíz-nodo) están situados en las posiciones y siguientes del vector hasta la
* Para acceder al nodo padre del nodo T[i] se accede al elemento Asumiremos sin pérdida de generalidad que los montículos de ejemplo del resto del capítulo implementan montículos de máximos.
* La implementación no se realiza utilizando árboles, sino vectores, cuyo recorrido es más eficiente. El vector T [1…n] implementa un montículo cuando:



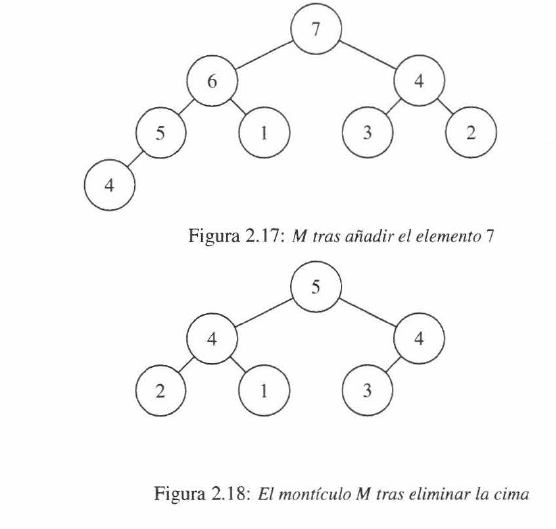
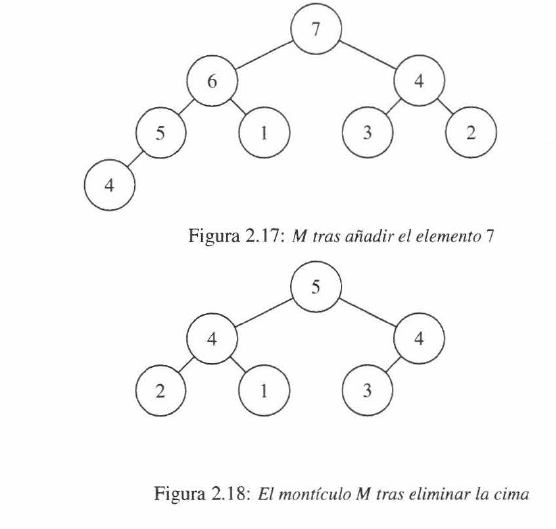
* El nodo raíz es el elemento T [1] del vector y contiene el mayor (o menor) de los elementos del montículo.
* Los nodos hijos del elemento T[i] que son respectivamente T[2i] y T[2i+1] cumplen que para el caso de los montículos máximos.

Al cumplirse la propiedad de montículo que permite tener en la cima el elemento más prometedor, la principal utilidad de esta estructura de datos es la de proporcionar un método eficiente de implementar colas de prioridad. El montículo fue desarrollado por Floyd como estructura de datos de apoyo a un método de ordenación denominad HeapSort (ordenación en montículo), el cual, se verá con mayor detalle en el capítulo 6.



Este montículo de la izquierda se puede representar con el vector:

Como se puede comprobar los montículos no son elementos ordenados, sino que en la cima de este está el elemento de mayor valor. La peculiaridad de los montículos es que es una estructura e datos que permite de manera muy eficiente insertar y borrar un elementos y restaurar la propiedad de montículo.



Una estructura de montículo tiene aplicaciones importantes en otros algoritmos conocidos:

* Algoritmos de ordenación como Heapsort, uno de los algoritmos más relevantes con un coste eficiente en el caso peor.
* Algoritmos de selección: la búsqueda de mínimos, máximos, medianas o el k-ésimo mayor elemento pueden realizarse en tiempo lineal mediante el uso de montículos.
* Algoritmos basados en grafos: usando montículos como estructuras internas de almacenamiento de nodos o aristas, la complejidad puede reducirse en un orden polinómico. Ejemplos de este tipo de algoritmos son la creación de árboles de recubrimiento mínimo de Prim o el problema del camino más corto de Dijkstra.

1. **Implementación y operaciones sobre los elementos del montículo.**

La estructura de datos para implementar el montículo consta de un registro con tres elementos:

**registro** montículo

T: **vector**[1..n] enteros;

C: natural;

MAX: natural

**fregistro**

* Un vector
* Un contador c para el número de elementos del montículo.
* Un valor para el tamaño máximo del montículo.

El montículo tiene las operaciones básicas siguientes:

* **CrearMonticulovacio**: devuelve un montículo vacío.

**fun** CreaMonticuloVacio(m: monticulo)

m.T <- null

m.c <- 0

m.Max <-n

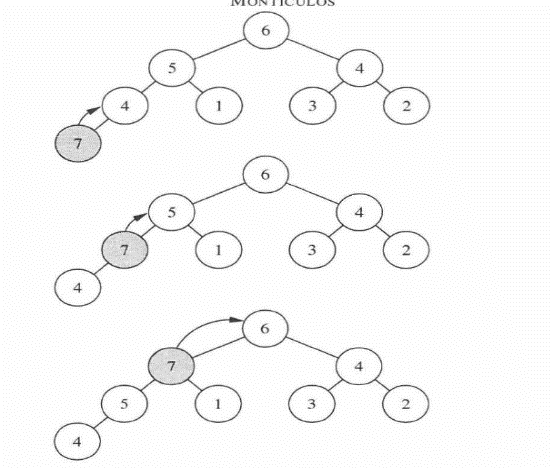
**ffun**

* **MonticuloVacio?:** devuelve **cierto** si el montículo está vacío.

**fun** monticuloVacio?(m: monticulo):bool

**si** (m.c = 0) **entonces dev cierto sino dev falso**

**ffun**

* **Flotar:** reubica el elemento i-esimo del vector 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.

**fun flotar**(T: vector, i: natural)

**mientras** (i > 1) (T[i div 2] < T[i]) **hacer**

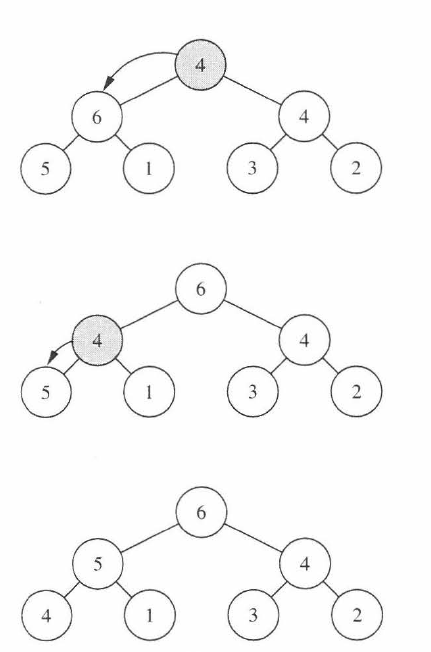
intercambiar(T[i], T[i div 2])

i <- i div 2

**fmientras**

**ffun**

* **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.



**fun** Hundir(m\_ montículo, i: natural)

**var**

**u,hi,hd, p : natural**

**fvar**

**repetir**

**hi <- 2\*i; -> hijo izquierdo**

**hd <- 2\*i +1; -> hijo derecho**

**p <- i; -> padre**

**si** (hd m.c) (m.T[hd] > m.T[i]) **entonces**

i <- hd

**fsi**

**si** (hi m.c) (m.T[hi] > m.T[i]) **entonces**

**i <- hi**

**fsi**

intercambiar (m.T[p], m.T[i])

**hasta** p = i;

**ffun**

**fmientras**

**ffun**

* **Insertar**: inserta un elemento en el montículo y lo flota para restaurar la propiedad del montículo. Se realiza incorporando el elemento al final del montículo y aplicando la operación flotar.

**fun** insertar(e: elemento, m: monticulo): montículo

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**

* **Primero**: devuelve la cima del montículo sin modificarlo.

**fun** Primero(m: monticulo): elemento

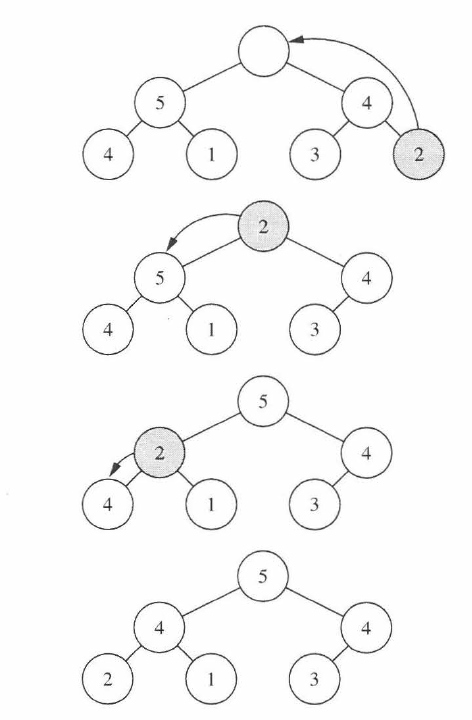
**si** m.c = 0 **entonces dev error**

**sino**

**dev m.T[1]**

**fsi**

**ffun**

* **ObtenerCima**: devuelve la cima del montículo, la elimina del mismo y recompone propiedad del montículo.

**fun** obtenerCima(m: monticulo): elemento

**var**

**e: elemento**

**fvar**

**si m.c entonces**

**e <- m.T[1]**

**m.c <- m.c – 1**

**Hundir (m.T,1)**

**dev e**

**fsi**

**ffun**

1. **eficiencia en la creación de montículos a partir de un vector.**

la creación de un montículo a partir de una colección de valores puede optimizarse para tener un coste lineal. Veamos algunos procedimientos para realizar esta operación y analicemos su eficiencia.

FUNCION CREAMONTICULO

La primera opción de creación del montículo a partir de un vector se puede realizar utilizando la función flotar, que inserta cada elemento al final del montículo y recupera la propiedad de este. Para crear un montículo a partir de un vector V [1...n] mediante inserciones, usamos una simple iteración.

**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**

Si n es el número de elementos del montículo, la profundidad media es , lo que equivale a una complejidad . El i-ésimo objeto requerirá como mucho intercambios en el árbol, por lo que:

Sin embargo, hay una manera más eficiente de crear un montículo a partir de los elementos de un vector mediante el procedimiento Hundir.

**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**

El último nivel del árbol contiene los n/2 últimos elementos del vector, y estos no pueden hundirse másm por lo que se parte del penúltimo nivel, es decir, desde i = n/2. Esto tendrá una complejidad lineal O(n)

ORDENACIÓN BASADA EN MONTÍCULOS: ALGORITMO DE HEAPSORT

El concepto de montículo como estructura de datos fue desarrollado ad hoc como apoyo a la creación de un algoritmo de ordenación eficiente. Con un montículo disponemos de una estructura de datos en la que encontrar el mínimo es una operación de coste constante. Una vez extraído el primer elemento, restaurar la propiedad de montículo tiene un coste O (log n). El funcionamiento de este algoritmo es como sigue:

**fun** HeapSort(T:vector[1..n]): vectir[1..n]

**var**

e: entero

M: montículo

S: vector[1..n]

**fvar**

M <- CreaMonticuloLineal (T);

**para** i <- 1 **hasta** n **hacer**

**e <- obtenerCima(M**);

S[i] <- e

**fpara;**

dev S

**ffun**

**En cada paso del bucle se extrae el primer elemento y se restaura la propiedad de montículo con un coste total O (log n). Como realizamos n operaciones de coste O (log n), el algoritmo tiene un coste global O(n log n).**

* Se convierte el vector en un montículo de máximos.
* Se selecciona el máximo (la cima del montículo) y se incorpora al vector S solución. El montículo pierde un el elemento y el vector S lo incorpora.
* Se restaura la propiedad de montículo sobre los elementos que restan. Así estamos en condiciones de volver a seleccionar el máximo.
* Al finalizar el vector S contiene le vector de entrada ordenado de mayor a menor.

1. **Otros tipos de montículos.**

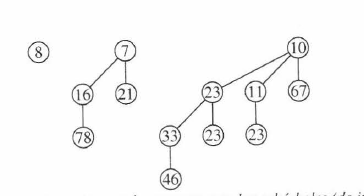
El montículo que describimos en este capítulo se denomina montículo binario y como hemos indicado se usa para la implementación de colas de prioridad y que son de utilidad en otros algoritmos. Además, hay otros tipos de montículos, entre los que destacan los siguientes:

**MONTICULO BINOMIAL**

Se trata de una colección de árboles binomiales que se definen de manera recursiva.

* En el caso de un único elemento, un árbol binomial de orden 0 es un nodo.
* Un árbol binomial de orden k tiene una raíz de grado k (número de hijos) y los hijos son raíces de árboles binomiales de orden
* De esta forma es posibles construir un árbol binomial de orden “k” a partir de dos árboles k -1, agregando a uno de ellos como el hijo más a la izquierda del otro. En general, un árbol binomial de orden “k” contendrá 2^k nodos, y tendrá una altura k. Los árboles de montículo binomial tienen las siguientes propiedades:
  + Los hijos son siempre mayores(menores) o iguales que el padre para un montículo de mínimos (máximos).
  + Para todo k > 0 existe al menos un árbol binomial en el montículo con una raíz de grado k.
  + Cada subárbol del montículo binomial cumple que: la raíz del subárbol tiene en la cima el elemento mínimo(máximo).
  + Puede haber solo 0 ó 1 árbol binomial por cada orden, incluido el orden 0.

Nota: los montículos binomiales mejoran la eficiencia frente a una implementación de montículos básicos.

****

Aquí podemos ver 3 subárboles, de izquierda a derecha (

* En el montículo binomial vemos que se cumple que la raíz (número 10) es de grado 3, y los hijos de esta raíz cumplen que son de grado

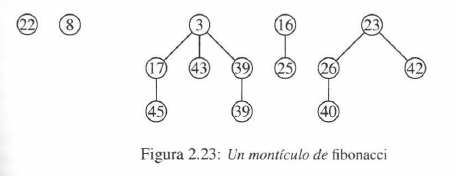
(3-1(23), 3-2(11), 3 – 3(67)).

* Además, los nodos (23, 11, 67) también son montículos binomiales.

**MONTÍCULO DE FIBONACCI**

Se trata de una colección de árboles en la que cada uno de ellos satisface la propiedad de montículo. La estructura es más flexible que los montículos binomiales, ya que no restringe la existencia o no de árboles o la profundidad de los niveles (puede haber un montículo de Fibonacci con todos sus elementos en un solo árbol). Esta flexibilidad va a permitir realizar operaciones como la mezcla de montículos, que se realiza concatenando las dos listas de árboles de cada montículo. Las condiciones son las siguientes:

* El número de hijos de cada nodo se mantiene menor que log n, siendo n el tamaño del montículo.
* El tamaño de los sub-arboles con raíz en un nodo de grado k es al menos , siendo el k-ésimo número de la serie de Fibonacci.

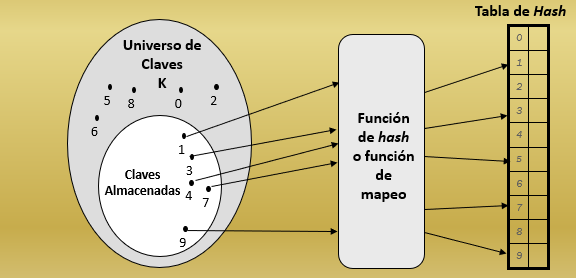


La raíz de un montículo de Fibonacci es el menor elemento de primer nivel (el valor 3). El primer nivel está formado por las raíces de los árboles (elementos 22, 8,3,16,23) que forman la primera lista circular.

A continuación, veamos algunos costes de operaciones conocidas para ilustrar la eficiencia de este tipo de estructuras.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Lista enlazada | montículo | Montículo binomial | Montículo de Fibonacci. |
| insertar |  |  |  |  |
| Mínimo(máximo) |  |  |  |  |
| borrar |  |  |  |  |

* 1. **Tablas de dispersión (hash).**

Las tablas *hash* o tablas de dispersión permiten el almacenamiento de datos pertenecientes a un dominio potencialmente muy grande de registros, pero del que sólo usamos y referenciamos un número reducido. El uso de tablas hash permite un ahorro considerable de memoria y es eficiente, siempre que el número de registros del subconjunto se mantenga dentro de los previsto.

Para lograr la reducción del espacio de claves se hace corresponder a cada clave directa, una clave en el subconjunto de índices de la tabla hash, mediante una función hash. Entra dentro de lo previsible que dos o más claves diferentes del dominio inicial, proporcionen la misma clave en el índice de la tabla hash, lo que se conoce como colisión. La función Hash asocia claves a valores hash, que a su vez son usados como índices en la tabla hash. Las dos propiedades fundamentales son:

* El índice o clave permite encontrar o asociar un rango de elementos.
* La búsqueda no compara por valores clave directamente, sino que aplica una función h(k) que nos da directamente la localización de la clave k en la estructura o tabla hash.
* Se acceden a los elementos de la tabla haciendo operaciones algorítmicas que transforman las claves en direcciones de la tabla.
* En resumen, las funciones hash son por tanto funciones que transforman claves o llaves en direcciones que referencias a un documentos o registro en la tabla o estructura donde están almacenados.
  1. **Funciones Hash.**

Una función Hash asocia una clave o llave con una posición en la tabla hash. La función hash es una aplicación entre el conjunto dominio de las claves X y el conjunto dominio de direcciones D de la estructura de datos.

Donde h(k) debe tener las siguientes propiedades:

* Debe repartir equis probablemente los valores, es decir, la
* La función ha de poder calcularse de forma eficiente.
* Cambios pequeños en las claves o llaves suponen cambios significativos en la función hash.

Propiedades de las funciones hash.

Las funciones hash son útiles en la medida que proporcionan un mecanismo de direccionamiento que cumple una serie de requisitos. Básicamente nos centraremos en estas propiedades:

* **Sentido único**: las funciones hash son funciones que en general no mantienen en el resultado la semejanza entre dos entradas. (útil para criptografía). no se puede calcular k, a partir de h(k).
* **Colisiones:** las colisiones son inevitables ya que el espacio de valores se salida es mucho menor que el espacio de valores entrada.
* **Distribución de datos:** se deben distribuir de manera uniforme.

**Tipos de funciones Hash**

**FUNCIÓN MÓDULO**

La función módulo calcula el resto de la división de la clave por un valor M+1, siendo M el número de índices existentes en la tabla hash. Es decir:

* La elección del valor de M es importante, debe mantener la uniformidad.
* Es recomendable utilizar números primos cercanos a una potencia de 2 (así el resultado depende de más bits de k).
* No se recomienda que sea potencia de 2 (el valor de hash dependería sólo de los bits menos significativos de k).

**FUNCIÓN CUADRADO**

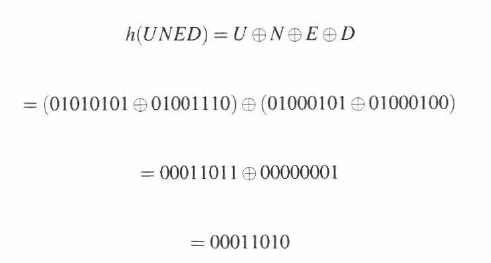
Este método eleva al cuadro el valor de la clave y escoge los “c” bits centrales, dando un valor en el rango. [0…]. Por ejemplo:

* Tenemos registros cuya clave de 4 dígitos números en base 10 [0000 hasta 9999].
* Tenemos una tabla Hash de 100 elementos [0…99]

Si la llave es

**FUNCIÓN PLEGADO (COMPRENSIÓN)**

Este método toma partes de la clave y opera sobre ellas, por ejemplo, realizando operaciones XOR. Consiste en un proceso rápido y aplicable a claves no numéricas, aunque puede llevar a muchas colisiones. Por ejemplo:

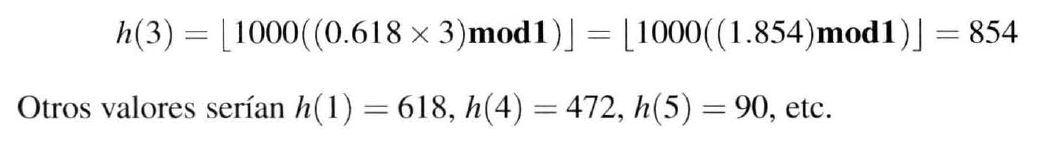


**FUNCIÓN MULTIPLICACIÓN**

El método opera en dos pasos:

* Se normaliza la clave de entrada multiplicándola por una constante φ∈(0..1).
* Se multiplica el valor obtenido por el número M de entradas de la tabla.
* Truncamos la parte fraccionaria (representada por mod1)
* Valor de φ recomendado, razón áurea:

Veamos el siguiente ejemplo:



* 1. **Colisiones**

En el caso de funciones hash para la ordenación, el caso de que dos claves colisiones es más frecuente, aunque se trata de evitar igualmente y se procura que la colisión sea equiprobable. Además, hay que tener en cuenta que hacer en caso de conflicto, es lo que se conoce como resolución de colisiones. En este caso se requiere almacenar los registros que comparten clave de manera que, en caso de colisión, sean accesibles igualmente.

**ESTRATEGIAS PARA LA RESOLUCIÓN DE COLISIONES**

La gestión de colisiones tiene dos estrategias diferenciadas.

* **Hashing abierto**: uso de espacio adicional a la propia tabla hash para los valores colisionados. Este utiliza estructuras dinámicas externas a la tabla para el almacenamiento de las claves que han generado colisiones. Hay varias maneras de implementar este tipo de Hashing:
  + **Niveles (buckets)**: dada la tabla hash, se añaden niveles o dimensiones adicionales. En lugar de una table clave-valor se añaden dimensiones adicionales para valores que colisionen.
  + **Encadenamiento directo**: En este caso se crea una estructura lineal para aquellos valores colisionados.
* **Hashing cerrado**: utilizar la propia tabla hash para almacenarlos, pero siguiendo diversas estrategias. Permite resolver la colisión mediante la búsqueda en ubicaciones alternativas en la tabla, hasta que encontramos un sitio libre de la misma. Se debe determinar que hay un sitio libre en la tabla con la presencia de un valor que lo determine, y si es así, se ubica el valor en la posición indicada por la función hash. Los métodos más empleados son:
  + **Recorrido lineal**: en general, a la dirección obtenida por la función hash h’(k) se le añade un incremento lineal que proporciona otra dirección en la tabla, la principal desventaja de este método es que es tendente a la creación de agrupaciones (clusters):

* + **Recorrido cuadrático**: hay otro método basado en una expresión cuadrática basada en la función hash h’(k) que permite mayor dispersión de las colisiones por la tabla, al mismo tiempo que proporciona un recorrido completo por la misma. Así si usamos:
  + **Recorrido mediante doble Hashing**: Dada una función hash , el doble Hashing utiliza una segunda función llamada función de paso tal que:

Para que este método funcione ha de cumplirse que:

* + - La función debe ser diferente de la función
    - Los valores de deben ser primos relativos de m (es decir que no comparten factores primos) para que los índices de la tablan se ocupen en su totalidad.

NOTA: en el libro entre la página 52 y 54 hay ejemplos interesantes de visualizar para comprender mejor las tablas hash.