

Ejercicios resueltos de programación 3

Tema 5. Estructuras de datos.

Los ejercicios de este tema no son muchos, no obstante veremos el *índice* como en el resto de temas. Al no existir problemas de este tema, pondremos un nuevo apartado en el que habrá distintas cuestiones sin solución.

1. Introducción teórica	3
2. Cuestiones de exámenes	5

Introducción teórica:

Veremos estas nociones de teoría correspondiente con los **montículos**, ya que es muy importante el tenerlo claro, habiendo multitud de ejercicios que se resuelven de esta manera, aunque luego insistamos de nuevo con esta misma teoría en los ejercicios:

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

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

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

En resumen y para los vectores sería:

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

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

Relacionado con el ejercicio es que tenemos dos posibles algoritmos para crear un montículo. El primero de ellos será:

```
proc crear-montículo-lento( $T[1..n]$ )  
  para  $i \leftarrow 2$  hasta  $n$  hacer flotar ( $T[1..i], i$ )  
fproc
```

Para seguir el recordatorio veremos el algoritmo iterativo de flotar una posición, que será:

```
proc flotar ( $T[1..n], i$ )  
   $k \leftarrow i$ ;  
  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  $j = k$   
fproc
```

El segundo algoritmo será el siguiente:

```
proc crear-montículo ( $V[1..N]$ )  
  para  $i \leftarrow [n/2]$  bajando hasta 1 hacer hundir ( $T, i$ )  
fproc
```

Vimos en la teoría que uno de ellos es ineficiente con respecto a ello, lo que supone que haría más operaciones de *flotar* y aunque tiene coste lineal la constante multiplicativa es mayor. Evidentemente, es el primero de ellos, lo cual no nos valdría para el ejercicio que queremos resolver. Por tanto, escogeremos el algoritmo segundo que hemos escrito, ya que será más eficiente.

Pasamos a escribir el procedimiento para *hundir* un elemento del montículo:

```
proc hundir ( $T[1..n], i$ )  
   $k \leftarrow i$ ;  
  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  $2 * j \leq n$  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  $j = k$   
fproc
```

Estas funciones serán bastantes importantes el saberlos, ya que nos harán falta en numerosos ejercicios. Por lo que serán de gran importancia el sabérselos (no memorizarlos, por supuesto). Como regla nemotécnica (a mí personalmente me vale) el hundir es ir de arriba abajo (como los buzos) y flotar es ir de abajo a arriba. Es necesario el tener claro estas ideas, ya que son críticas en los montículos.

Otra regla nemotécnica importante y que me ha liado mucho es asumir el significado de estas variables:

- i**: Indica valor inicial que, dada por argumento, la posición que se flota o hunde.
- J**: Indica la posición que se refrescará tras intercambiar dentro del bucle "repetir".
- k**: Indica la nueva posición que asumirá el elemento, previo al intercambio, igualmente dentro del bucle "repetir" o ,dicho de otra manera, la posición a la que se intercambiará el elemento una vez que se cumple la propiedad del montículo bien sea con hundir o flotar.

Vemos que se sale del bucle cuando $j = k$, por lo que ya no se podrá intercambiar dos posiciones.

Por último, decir que siempre consideraremos **montículos de máximos** (la raíz es el máximo elemento), salvo cuando nos lo digan expresamente, ante ese caso tendremos **montículos de mínimos**. No queda decir que las operaciones son las mismas, sólo que cambian el signo.

La idea es considerar el vector como un montículo. Empezamos convirtiendo inicialmente en montículos los subárboles del nivel más bajo, para progresivamente seguir modificando en niveles superiores sucesivos.

1ª parte. Cuestiones de exámenes:

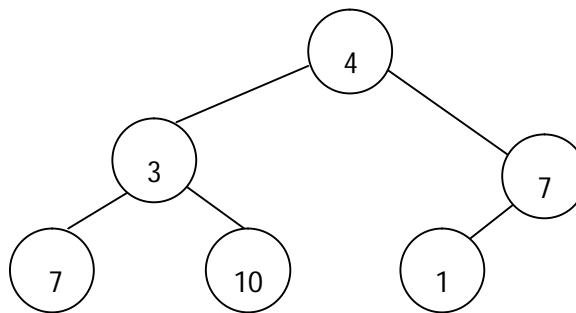
Septiembre 2000-reserva (ejercicio 3)

Enunciado: Explicar cómo funciona el algoritmo que dota a un vector la propiedad del montículo en tiempo lineal (no es necesario demostrar que ese es el coste).

Respuesta: Tenemos en mente los algoritmos vistos en la sección de teoría, siendo uno de ellos ineficiente con respecto al otro, lo que supone que haría más operaciones de *flotar* y aunque tiene coste lineal la constante multiplicativa sería mayor. Evidentemente, es el primero de ellos, lo cual no nos valdría para el ejercicio que queremos resolver. Por tanto, escogeremos el algoritmo segundo que hemos escrito, ya que será más eficiente.

```
proc crear-montículo ( $T[1..N]$ )  
  para  $i \leftarrow [n/2]$  bajando hasta 1 hacer hundir ( $T, i$ )  
fproc
```

Veremos el funcionamiento con un **ejemplo**. Se nos da este vector $V = [4,3,7,7,10,1]$. En forma de árbol será:

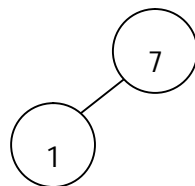


Vemos que al ser el elemento en la posición 2 no cumple la propiedad del montículo de máximos, con lo que aplicaremos el procedimiento antes visto.

Tenemos estos pasos:

1^{er} paso. Empezaremos por $i = 3$, es decir, por la posición 3 en el montículo.

Hacemos una llamada a hundir ($V, 3$). Tendremos que hundir esta posición para ordenarla. Tendremos este subárbol:

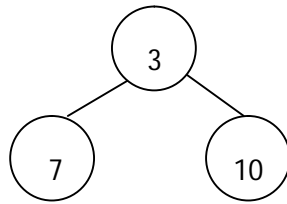


Vemos, por tanto, que cumple la propiedad del montículo, en la que el padre es mayor o igual que el hijo, por lo que no habrá ningún intercambio de posiciones.

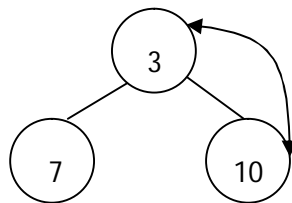
El vector no se modifica.

2º paso. Continuaremos por $i = 2$.

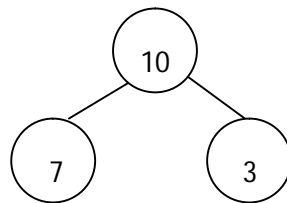
Haremos llamada a hundir ($V, 2$). Veremos de nuevo el subárbol para así ver qué posición hay que hundir:



Observamos que el nodo padre es menor que los hijos por lo que **NO** cumple la propiedad del montículo. Siguiendo el algoritmo hay que intercambiarlo con el hijo mayor, por lo que intercambiamos estas posiciones:



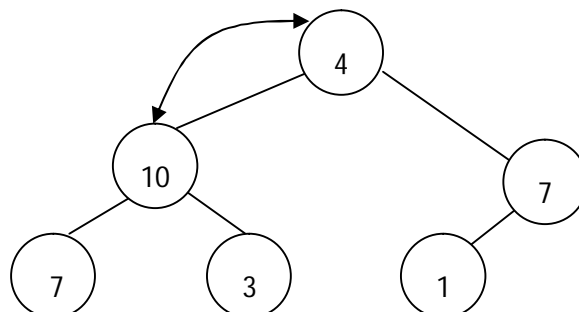
Con lo que el resultado quedaría así:



El vector tras este paso es $V = [4, 10, 7, 7, 3, 1]$.

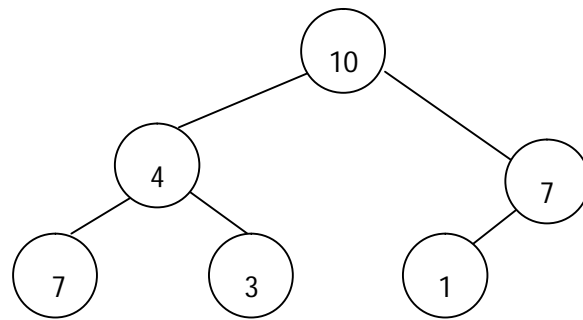
3º paso. Seguimos por $i = 1$ o lo que es lo mismo la raíz.

Hacemos llamada a hundir ($V, 1$). Nuestro subárbol será:



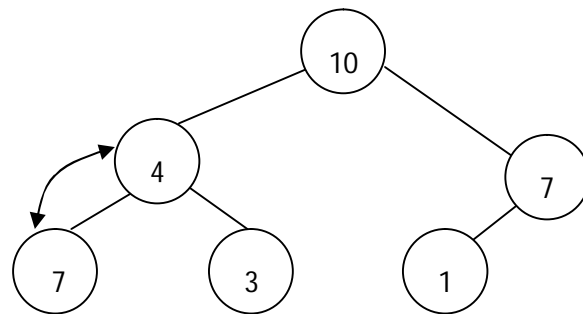
Vemos que el nodo valor 4 (nuestra raíz) es menor que cualquiera de sus hijos, por lo que a priori no cumple la propiedad del montículo. Seguiremos la misma filosofía de antes e intercambiamos con el hijo mayor, que de nuevo es la posición del vector 2.

Tras intercambiarlo, el montículo quedará:

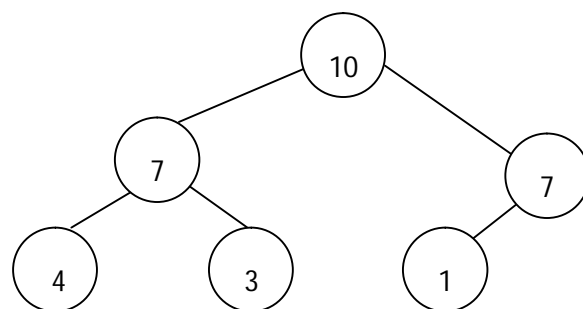


Y el vector será $V = [10, 4, 7, 7, 3, 1]$.

Viéndolo de nuevo, observamos que sigue sin cumplirse la propiedad del montículo. Nuestro algoritmo de *hundir* seguirá por la posición 2, que es la del último intercambio. Vemos, por tanto, que no cumple la propiedad del montículo de nuevo en dicha posición, por lo que se intercambia con el hijo mayor, como sigue:



El montículo resultante del intercambio será:



En este caso, vemos que ya cumple la propiedad del montículo en todo el árbol. Por tanto, ya se ha creado un montículo y el vector resultante será:

$V = [10, 7, 7, 4, 3, 1]$.

Nótese varios aspectos importantes:

- Al intercambiar con la posición 2 observamos que el subárbol derecho se deja sin explorar. Esto significa que ya cumple la propiedad previamente analizada en el primer paso (cuando $i = 3$).
- Hemos seguido el planteamiento utilizando árboles, pero tendremos presente que una de las representaciones más utilizadas es con vectores. Dicho esto, podríamos haber hecho esto mismo de forma grafica usando vectores y viendo así las posiciones de intercambio.

Febrero 2001-1ª (ejercicio 1)

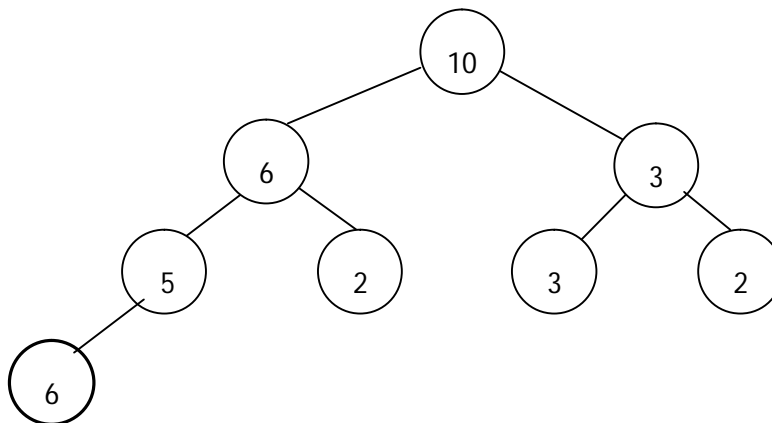
Enunciado: Dado el siguiente montículo $[10,6,3,5,2,3,2]$ se pide insertar el valor 6 describiendo toda la secuencia de cambios en el mismo.

Respuesta: Tendremos el algoritmo de inserción siguiente:

```
proc añadir-nodo ( $T[1..n], v$ )  
   $T[n + 1] \leftarrow v$ ;  
  flotar ( $T[1..n + 1], n + 1$ )  
fproc
```

Observamos que se usa la función *flotar* (de abajo a arriba) para que al insertar el nuevo nodo se recupere la propiedad del montículo. En nuestro caso, tenemos que añadir un nodo al montículo $[10,6,3,5,2,3,2]$.

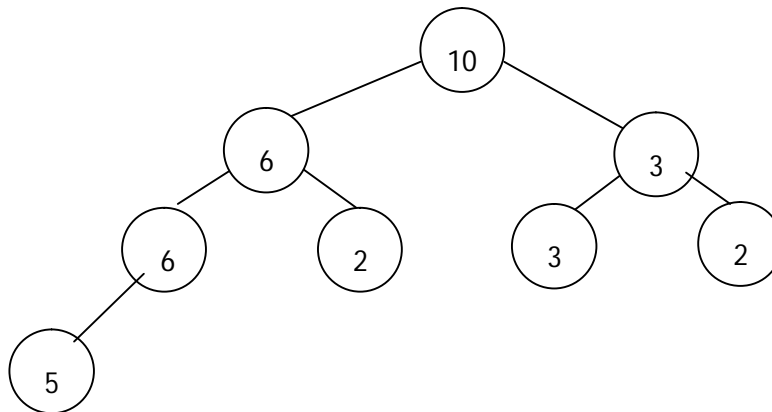
Pasamos como en el ejercicio anterior el vector a un árbol:



El nuevo vector tras insertar el elemento es $[10,6,3,5,2,3,2,6]$

Resaltamos el nodo insertado en **negrita**. Vemos que habrá que hacer una llamada a la función *flotar* según el procedimiento anterior, que será *flotar* ($T[1..8], 8$). Como antes, seguiremos estos pasos, que se harán en la misma llamada de *flotar* (no confundir con el ejercicio anterior, que eran distintas llamadas a las funciones).

1. Se compara $T[8]$ con su padre, que es $T[8 \text{ DIV } 2] = T[4]$. Como es mayor se intercambia resultando:



El vector, por el momento es $[10, 6, 3, 6, 2, 3, 2, 5]$.

2. Se comparará, en este caso, $T[4]$ con su padre, que recordemos es $T[4 \text{ DIV } 2] = T[2]$ y observamos que ambos son iguales. Por tanto, el bucle finalizará, al coincidir j con k y ya será montículo.

Febrero 2001-2ª (ejercicio 3) (parecido a ejercicio 1 de Dic. 02)

Enunciado: Dado un montículo $T[1..n]$, programar completamente en pseudocódigo una función recursiva flotar (T, i) para flotar el elemento de la posición i del vector T . Explicar cómo usar esta función para insertar un elemento en el montículo.

Respuesta: Para realizar la primera parte del ejercicio tendremos que recordar el algoritmo iterativo de flotar:

```

proc flotar ( $T[1..n], i$ )
   $k \leftarrow i$ ;
  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  $j = k$ 
fproc

```

Por lo que, casi siendo una transcripción directa tendremos el algoritmo recursivo de flotar:

```

proc flotar-rec ( $T[1..n], i$ )
  si  $(i > 1)$  and  $(T[i] > T[i \text{ DIV } 2])$  entonces
    intercambiar  $T[i]$  y  $T[i \text{ DIV } 2]$ 
    flotar-rec ( $T, i \text{ DIV } 2$ )
  fsi
fproc

```

La segunda parte de este ejercicio es exactamente igual al anterior, por lo que evitamos dar más detalles del mismo.

Febrero 2002-1ª (ejercicio 3) (Igual a ejercicio 2 Sept. 01-reserva)

Enunciado: Programar en pseudocódigo un algoritmo recursivo para la operación de hundir un elemento en un montículo.

Respuesta: Tendremos al igual que en el ejercicio anterior que partir del algoritmo iterativo de *hundir*. Recordamos, de nuevo, que es:

```
proc hundir ( $T[1..n], i$ )
   $k \leftarrow i$ ;
  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  $2 * j + 1 \leq n$  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  $j = k$ 
fproc
```

Intentaremos razonar de modo similar a como hemos hecho en el algoritmo recursivo de *flotar*, pero nos topamos con un inconveniente, por el que tendremos que averiguar cuál de los dos hijos es el mayor, es decir, el de mayor valor, por lo que deberemos almacenar en una variable temporal el valor del primero hijo para compararlo con el segundo e intercambiar el mayor. Veámos de modo práctico esta explicación previamente.

```
proc hundir-rec ( $T[1..n], i$ )
   $hmayor \leftarrow i$ ;
  { Buscar el hijo mayor del nodo  $i$  }
  si  $(2 * i \leq n)$  y  $(T[2 * i] > T[hmayor])$  entonces
     $hmayor = 2 * i$ ;
  fsi
  si  $(2 * i + 1 \leq n)$  y  $(T[2 * i + 1] > T[hmayor])$  entonces
     $hmayor = 2 * i + 1$ ;
  fsi
  { Si cualquier hijo es estrictamente mayor que el padre }
  si  $(hmayor > i)$  entonces
    intercambiar  $T[i]$  y  $T[hmayor]$ 
    hundir-rec ( $T, hmayor$ )
  fsi
fproc
```

Febrero 2002-1ª (ejercicio 3)

Enunciado: Programar en pseudocódigo todos los procedimientos necesarios para fusionar dos conjuntos implementados con montículos. Suponer que los conjuntos no pueden tener elementos repetidos.

Respuesta: Este ejercicio no tengo ni idea de cómo se puede resolver, sólo sé que pueden pedir los códigos que se han visto en teoría de los montículos, pero en este caso piden fusionar dos conjuntos implementados con montículos, lo cual desconozco. He buscado en internet, en el libro, en muchos sitios más que solución tiene este ejercicio y no he podido llegar a ninguna conclusión.

Diciembre 2003 (ejercicio 2)

Enunciado: ¿Qué diferencias hay entre un montículo y un árbol binario de búsqueda?

Respuesta: Para responder a esta pregunta vamos a definir que es cada uno de ellos y así se verá más adecuadamente sus diferencias. No hay solución oficial de esta pregunta, por lo que se ha tomado apuntes del libro base casi directamente (*Fundamentos de Algoritmia* de G. Brassard y P. Bratley). Pasamos, pues, a las definiciones y a continuación a las diferencias:

Un **árbol de búsqueda** es aquél en el que 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.

El **montículo** es un árbol binario esencialmente completo, cada uno de cuyos nodos incluye un elemento de información denominado valor del nodo y que tiene la propiedad consistente en que el valor de cada nodo interno es mayor o igual que los valores de sus hijos. Esto se llama **propiedad del montículo**.

Al eliminar o añadir nodos al árbol binario de búsqueda se puede volver *desequilibrado*, con lo que muchos nodos pueden tener un solo hijo, así que sus ramas se vuelven largas y delgadas. No resulta eficiente hacer búsquedas en el árbol desequilibrado, ya que el coste es lineal, porque hay que recorrer todos los nodos. Para equilibrarlo requiere un coste de $O(\log n)$, en el caso peor, siendo n el número de nodos que hay en el árbol.

Al igual que pasa con el árbol de búsqueda al quitar nodos o añadirlos en un **montículo** hay que restaurar la propiedad del montículo, lo que implica hundir o flotar, cuyo coste es $O(\log n)$.

Observamos cómo hemos puesto en ocasiones anteriores que tomaremos la cota superior, en vez del coste exacto, esto lo haremos porque al fin y al cabo el coste exacto es una unión de cota superior y cota inferior. En el montículo será coste exacto, pero lo dejaremos así.

Septiembre 2004 (ejercicio 2)

Enunciado: Implementar una versión recursiva de una función que tome un vector y le dé estructura de montículo.

Resultado:

Tendremos que hacer la versión recursiva de crear montículo. Hemos tomado de nuevo la solución que ponen (no sabemos si es oficial o no) con algunas modificaciones propias, aunque nos fijamos que el algoritmo que han tomado es el de crear montículo "lento", en el que hay más llamadas.

Recordemos la función *iterativa* de crear montículo:

```
proc crear-montículo-lento ( $T[1..n]$ )  
  para  $i \leftarrow 2$  hasta  $n$  hacer flotar ( $T[1..i], i$ )  
fproc
```

La función *recursiva* de crear montículo será, por tanto, la siguiente:

```
proc crear-montículo-rec ( $T[1..n], i$ )  
  si  $i > n$  entonces  
    devuelve ( $T$ )  
  si no  
    flotar ( $T, i$ )  
    crear-montículo-rec ( $T, i+1$ )  
  fsi  
fproc
```

Escribiremos la función recursiva de flotar, aunque lo hemos escrito previamente. Es otra solución posible, por lo que es interesante tenerla en cuenta:

```
proc flotar-rec ( $T[1..n]$ :tipo monticulo, VAR  $i$ :integer)  
  VAR  $i\_padre$ : integer;  
   $i\_padre = i \text{ div } 2$ ;  
  si ( $i > 1$ ) and  $T[i] > T[i\_padre]$  entonces  
    intercambiar  $T[i]$  y  $T[i\_padre]$   
    flotar-rec ( $T, i\_padre$ )  
  fsi  
fproc
```

La llamada inicial de la función de creación de montículo será:

```
crear-monticulo-rec ( $T, 2$ )
```

Otra versión es sin usar funciones auxiliares es sustituir el código de flotar por ser versión iterativa e insertarla en la función anterior.

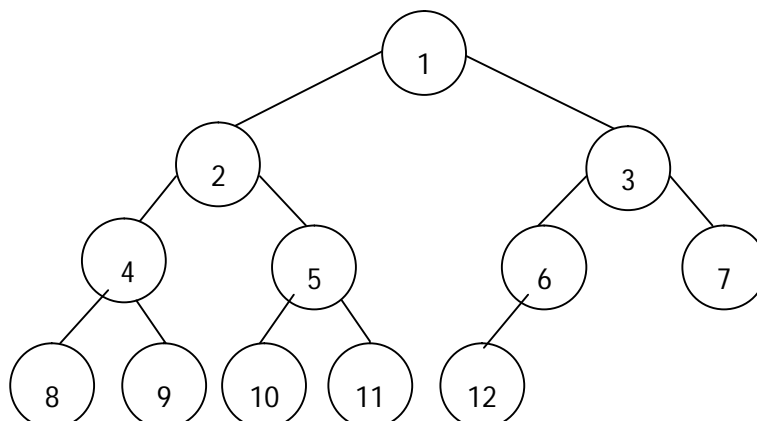
Septiembre 2004-reserva (ejercicio 2)

Enunciado: Sea $T[1..12]$ una matriz tal que $T[i] = i$ para todo $i \leq 12$. Crear un montículo en tiempo lineal, especificando cada paso y mostrando en todo momento el estado de la matriz T .

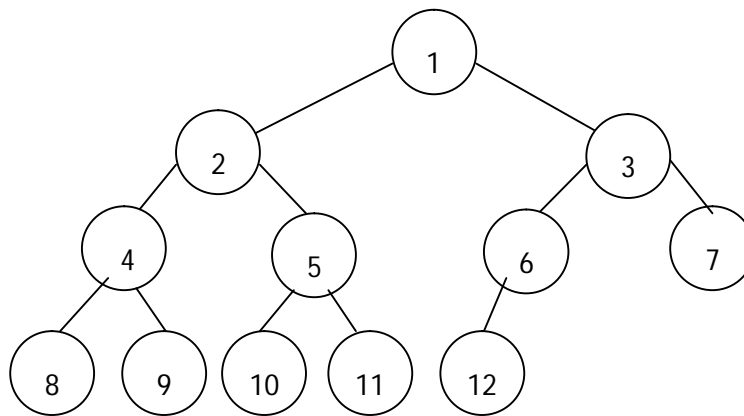
Resultado:

Hemos visto en ejercicios anteriores los distintos algoritmos con los que se resuelven estos ejercicios, por tanto, resolveremos el problema especificando paso a paso:

Inicialmente, el montículo será:

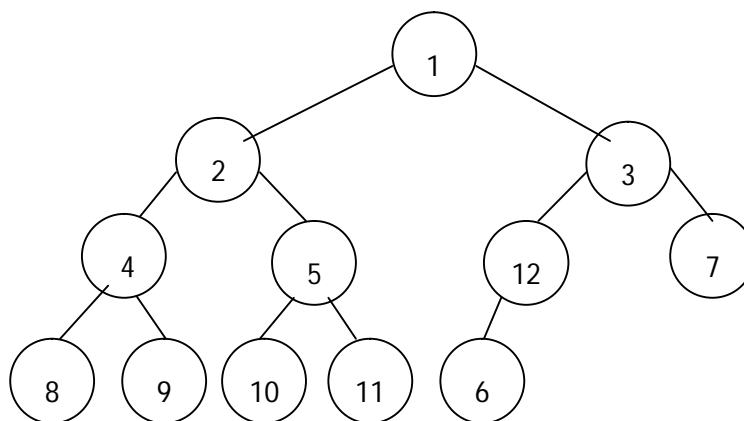


1^{er} paso: Empezaremos por la mitad del montículo, es decir, con $i = 6$:



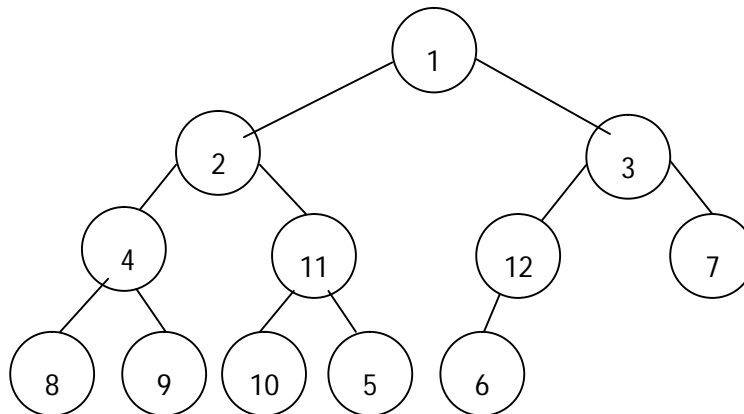
Intercambiamos la posición 6 con la 12, porque al usar el algoritmo de crear montículo rápido, hundiremos (en vez de flotar).

2^o paso: Seguiremos por $i = 5$:



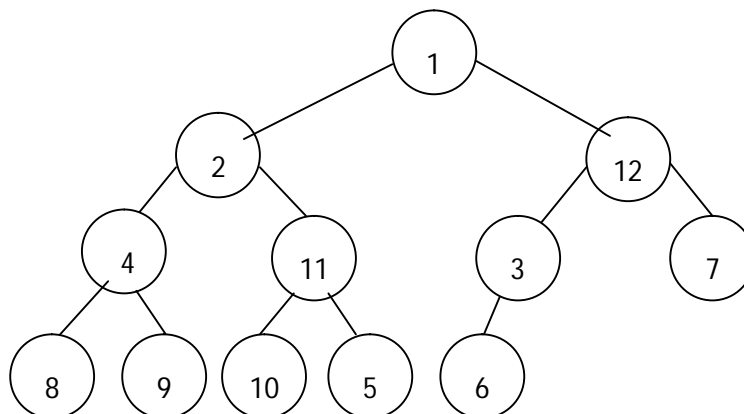
En este caso, intercambiaremos el valor de 5 con el hijo mayor, que es 11. En este caso, por el momento, coincidirán las posiciones con los valores.

3^{er} paso: Seguiremos con $i = 4$, que será el valor 4 y se nuevo hundimos ese valor hasta que cumpla la propiedad del montículo en el árbol:



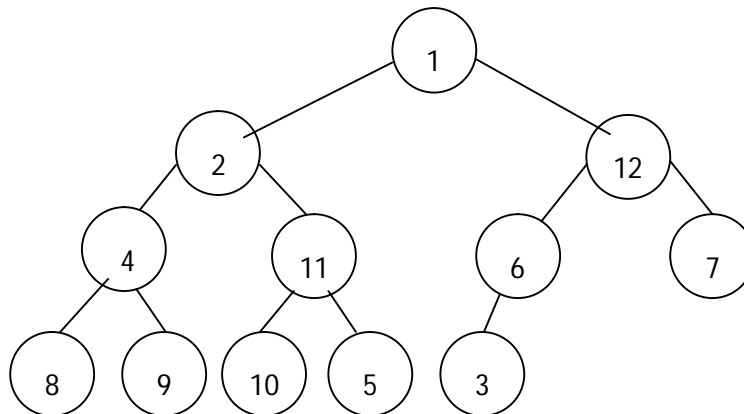
Intercambiaremos de nuevo con el hijo mayor, que es 9.

4^o paso: Nos vamos a $i = 3$ y observamos en el grafo anterior que tendremos que intercambiarlo con la posición 6, es decir, el valor 12. Tendremos, por tanto,

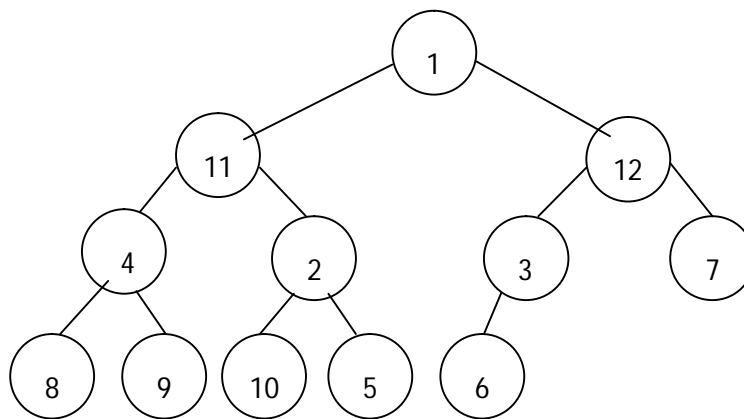


De nuevo, vemos que al intercambiarlo no se cumple la propiedad del montículo, por lo que de nuevo intercambiamos 3 con 6 (posición 6 con la 12, al tener solo un hijo). Lo haremos en la misma llamada a hundir, mucho cuidado con eso.

5º paso: El penúltimo paso será aquel en que $i = 2$.

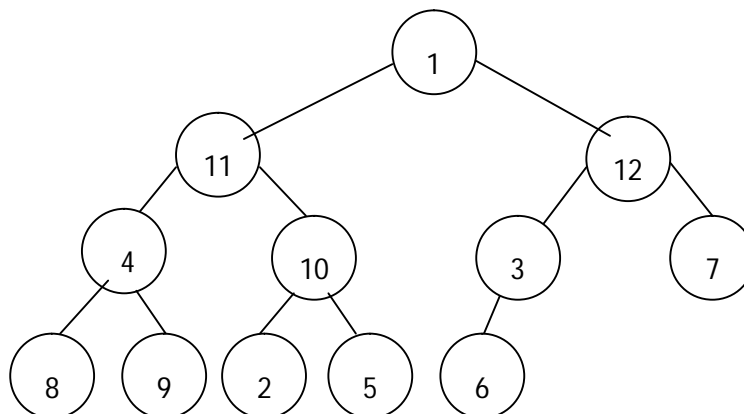


Intercambiaremos la posición 2 con la 5, que es su hijo mayor, resultando:

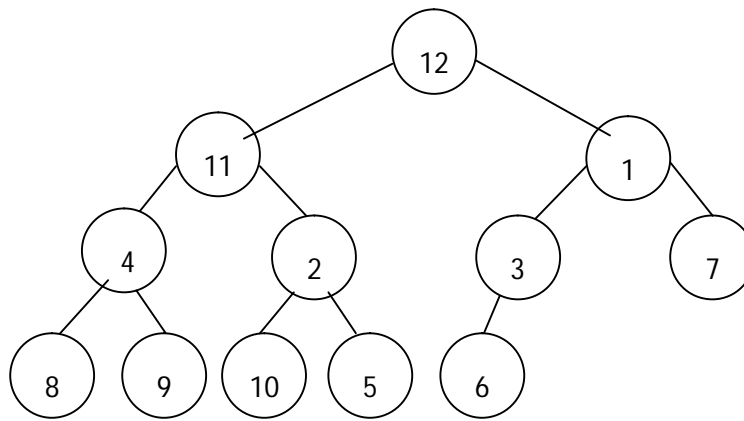


Al igual que el paso anterior necesitaremos hundir de nuevo este valor intercambiado, por lo que será la posición 5 con la 10, su hijo mayor.

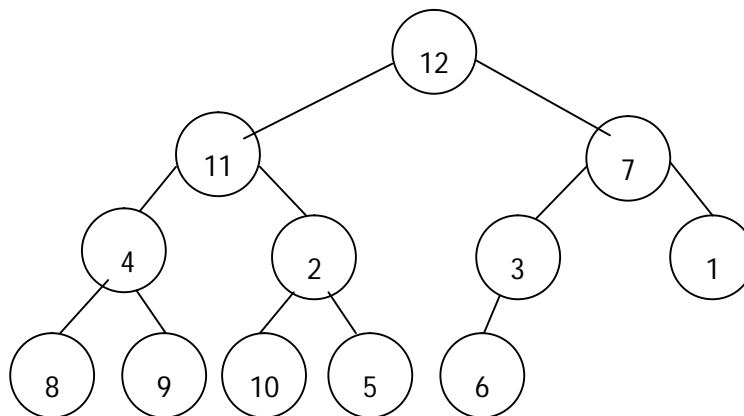
6º paso y último: Estamos en $i = 1$, que es el nodo raíz, por lo que tendremos que ver este árbol:



Intercambiaremos la posición 1 con la 3 (el valor 1 con el 12), quedando:



Vemos de nuevo que no cumple la propiedad del montículo, por lo que intercambiaremos la posición 3 con la 7 (el valor 1 con el 7), quedando el montículo completamente creado así:



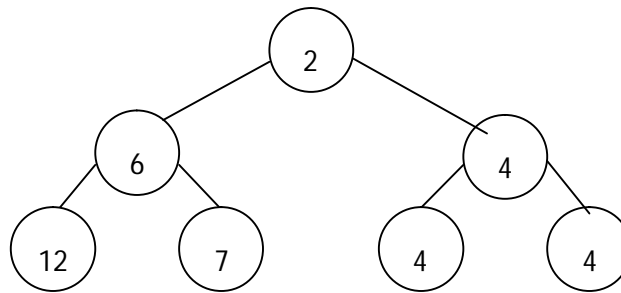
No hemos puesto los vectores intermedios tras hundir los elementos, pero pondremos el vector final, que será [12,11,7,4,2,3,1,8,9,10,5,6].

Septiembre 2005-reserva (ejercicio 1)

Enunciado: Dado $m = [2, 6, 4, 12, 7, 4, 4]$. Comprobar si es o no montículo de mínimos. Si no lo es, programar una función para convertirlo en montículo y aplicarlo a 'm'. Si no, programar una función de añadir elemento mediante la función "flotar" y aplicarlo al valor 3. En ambos casos, escribir el montículo resultante y detallar todos los pasos.

Resultado:

Verificaremos si es montículo de mínimos, que recordemos que es aquél en el que la raíz es el menor elemento. Tendremos en forma de árbol lo siguiente:



Verificaremos que cumple la propiedad del montículo en todo ello, que será

$$\left\{ \begin{array}{l} T[i] \leq T[2 * i] \\ T[i] \leq T[2 * i + 1] \end{array} \right\} \text{ Hijos con respecto a padres.}$$

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

Observaremos que cumple la propiedad del montículo en todo el árbol. Por tanto, ya es correcto.

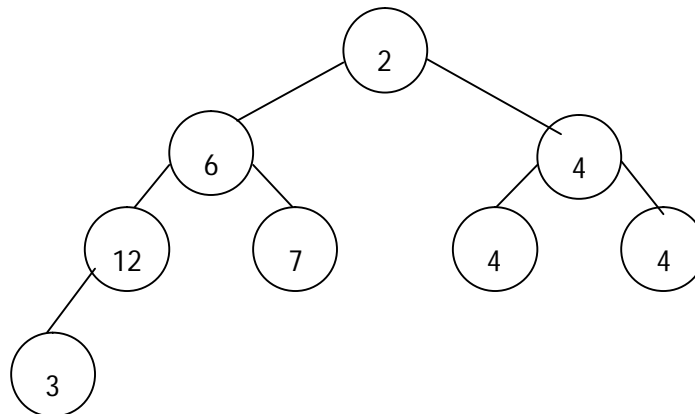
Añadimos un elemento al final del montículo. Recordemos el algoritmo de *añadir-nodo*:

```
proc añadir-nodo (T[1..n], v)
  T[n + 1] ← v;
  flotar (T[1..n + 1], n + 1)
fproc
```

En nuestro caso, variaremos conforme al montículo de mínimos la función *flotar*, como sigue:

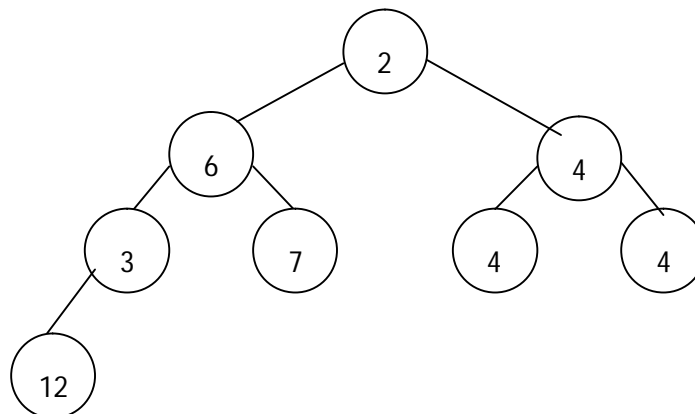
```
proc flotar (T[1..n], i)
  k ← i;
  repetir
    j ← k;
    si j > 1 y T[j ÷ 2] > T[k] entonces k ← j ÷ 2
    intercambiar T[j] y T[k]
    { si j = k, entonces el nodo ha llegado a su posición final }
  hasta que j = k
fproc
```

Al añadir el elemento 3, tendremos este árbol, para convertirlo en montículo:



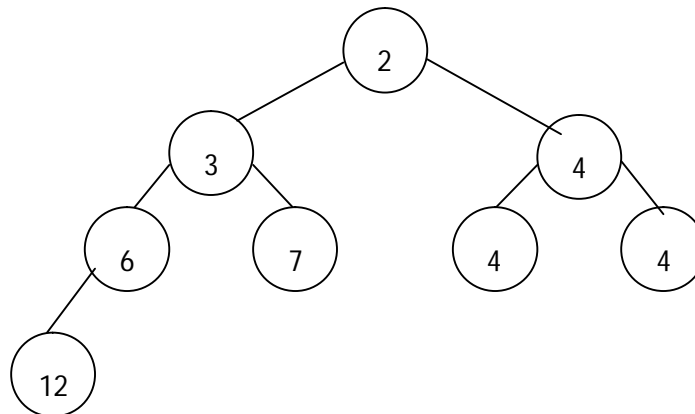
Observamos como en los ejercicios anteriores, que no cumple la propiedad del montículo, por lo que hay que flotar ese elemento (de abajo a arriba, ojito). Haremos estos pasos:

1º paso. Intercambiamos 3 con 12 (posición 8 con 4):



De nuevo no cumple la propiedad del montículo, por lo que tendremos que hacer otro paso más.

2º paso. Intercambiamos 3 con 6 (posición 4 con 2):



Vemos que al acabar esta llamada a flotar ya queda restaurada la propiedad del montículo en todo el árbol, por lo que se da por finalizado el ejercicio.

Febrero 2008-2ª (ejercicio 2)

Enunciado: Sea $T[1..n]$ con k elementos ($k < n$) un montículo de mínimos. Se pide programar una función "flotar" que dado un nuevo elemento $T[k + 1]$ restaure la propiedad del montículo en T . Una función iterativa que lo resuelva puntuará cero puntos.

Respuesta:

Tendremos la siguiente función que se asemeja enormemente a la dada en el ejercicio 3 de Febrero de 2001-2ª semana. Lo vamos a tratar, ya que es interesante verlo con más calma:

```
proc flotar-rec (T[1..n]:tipo monticulo,VAR i:integer)
  VAR i_padre: integer;
  i_padre = i div 2;
  si (i > 1) and T[i] < T[i_padre] entonces
    intercambiar T[i] y T[i_padre]
    flotar-rec (T,i _padre)
  fsi
fproc
```

Nos fijamos que con respecto a los algoritmos antes puestos, sólo cambiará en el signo $>$, puesto antes y ahora al ser un montículo de mínimos será $<$.