Estructuras de datos
Clase teórica 9

UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE MEDELLÍN

Contenido

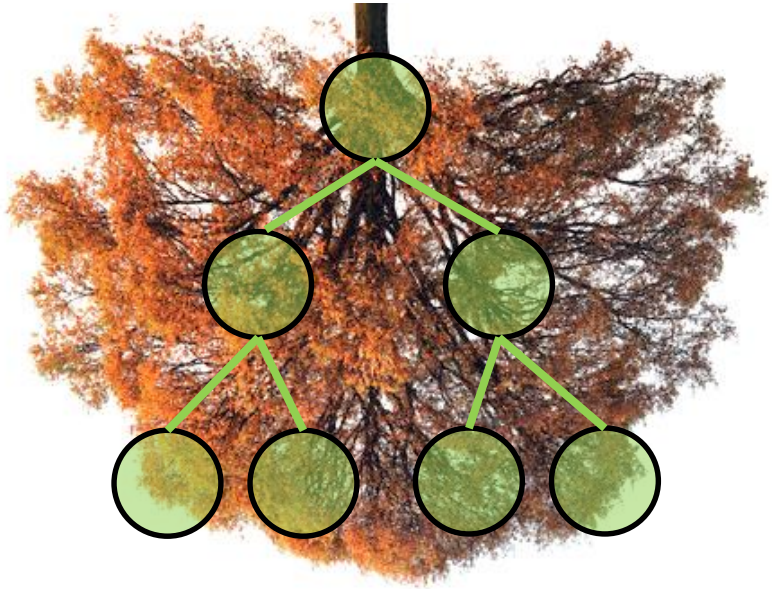
- Montículos binarios
- Colas con prioridad
- Ordenamiento heapSort

Material elaborado por: Julián Moreno

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Montículo

En el fondo, un montículo es igual a un árbol en cuanto a su estructura (relación jerárquica no lineal de uno a muchos), sin embargo, la metáfora utilizada es diferente: una agrupación piramidal de elementos.



Montículo binario

En los montículos binarios, como su nombre lo indica, cada nodo tiene a lo sumo dos “hijos”, es decir, igual que en los árboles binarios. La posición de un elemento cumple una regla, sin embargo es muy diferente a la de un árbol binario de búsqueda:

Regla 1:
Si el montículo es ascendente la regla es que los hijos de un padre deben ser mayores o iguales a este. Por consiguiente, en la parte más alta del montículo, en la cima (lo que en el árbol sería la raíz), siempre se encuentra el elemento más pequeño.

Nota 1: No existe un orden entre los elementos que se encuentran en un mismo nivel.

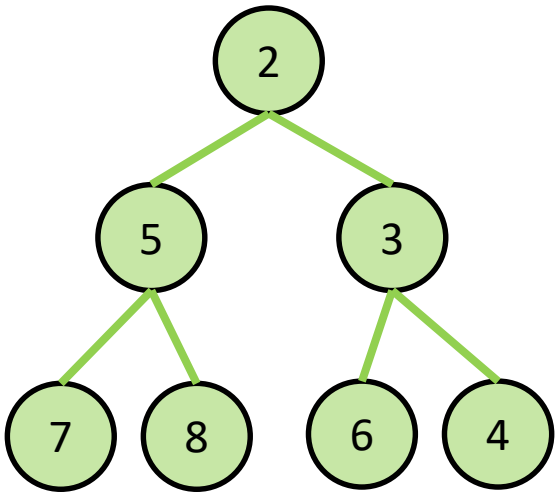
Nota 2: En el resto de esta clase nos referiremos a montículos ascendentes, sabiendo que en el caso de los descendentes las operaciones son análogas

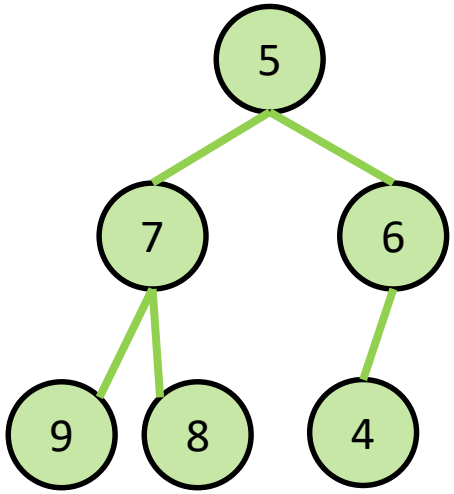
Montículo binario

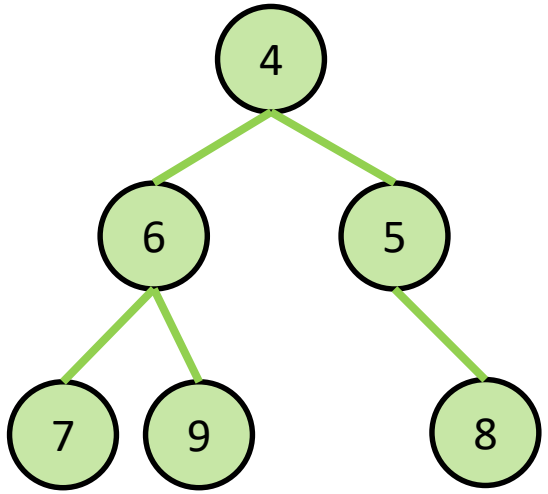
Adicional a la regla anterior, un montículo binario puede verse como un árbol binario donde se debe cumplir que:

Regla 2:

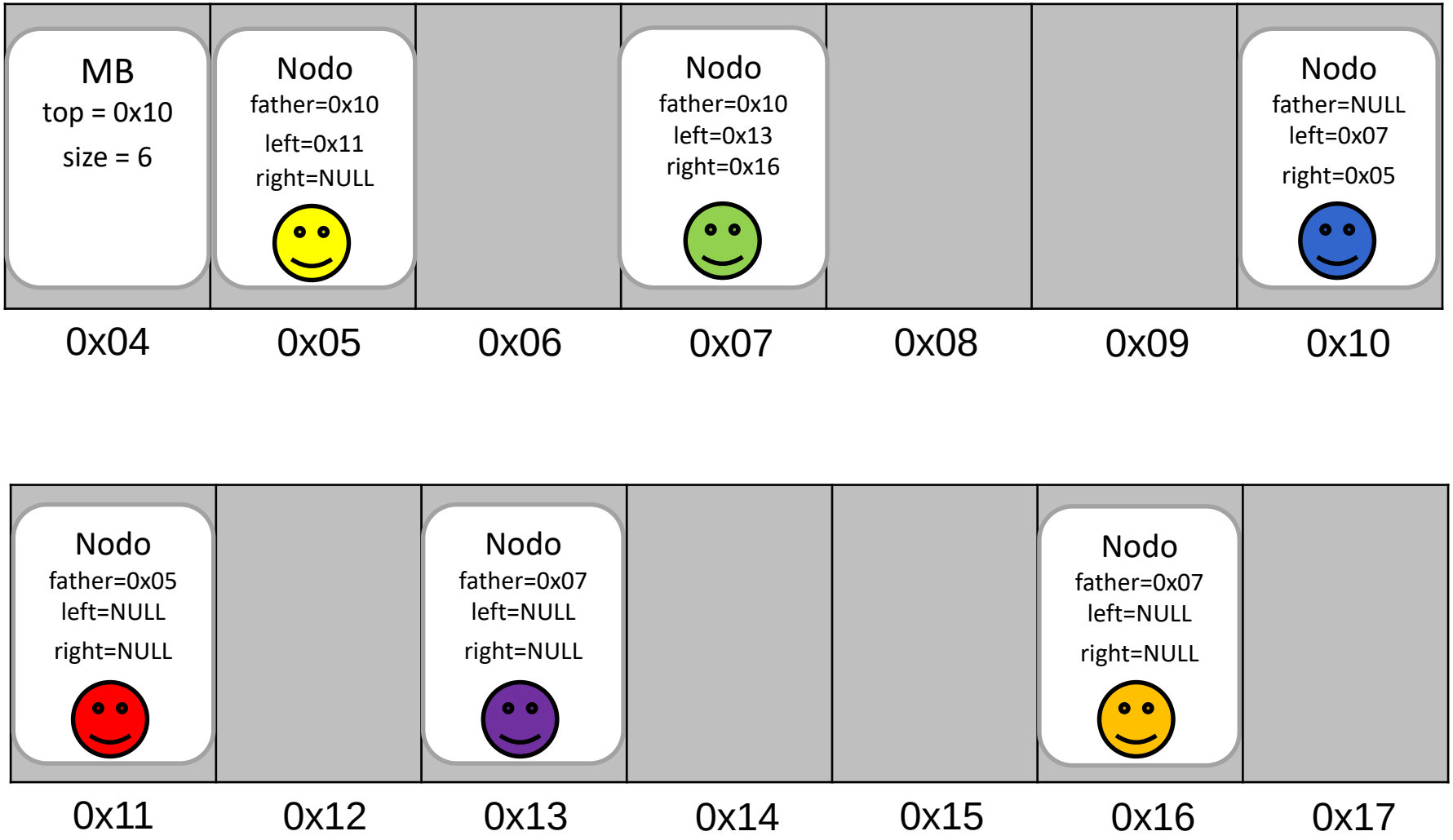
- Todos los niveles son completos excepto el último; y
- En el último nivel los nodos se sitúan lo más a la izquierda posible







Estructura de un montículo binario



Inserción en un montículo binario

Al insertar un elemento se deben garantizar dos cosas:

1. Se mantenga la estructura del montículo (árbol binario semi-completo)
2. Se mantenga el orden del montículo (todo padre menor o igual que sus hijos o viceversa)

Para lograrlo es necesario realizar un proceso en dos pasos:

Paso 1: Agregar el nuevo elemento en el último nivel y a la derecha del más a la derecha o, si el último nivel está completo, como hijo izquierdo del más a la izquierda de ese nivel.

Paso 2: En caso que se incumpla la propiedad de orden se deben reorganizar los elementos “subiendo” el nuevo por medio de intercambios hasta que quede en el lugar que le corresponde.

Paso 1: Agregación

```
p = new nodo(e) // e es el elemento a insertar
n++
if (n == 1)
   top = p
else{
   a = n
   while (a > 1){
      r.add(a%2) //r es una pila
      a = a/2    //división entera
   }
   q = top
   for(i=0; i<r.size()-1; i++){
      if(r.pop() == 0)
         q = q.left
      else
         q = q.right
   }
   if (r.pop() == 0)
      q.left = p
   else
      q.right = p
}
```

¿Cuál es la eficiencia de este algoritmo? *O(log(n))*

Paso 2: Re-ordenamiento

```
if (p != top){
   while(p.father.elm > p.elm){
      aux = p.father.elm
      p.father.elm = p.elm
      p.elm = aux
      p = p.father
      if (p == top)
         break
   }
}
```

¿Cuál es la eficiencia de este algoritmo? *O(log(n))*

Por tanto ¿ Cuál es la eficiencia total de la inserción?

O(log(n)) + O(log(n)) = O(log(n))

Búsqueda en un montículo binario

Se deben considerar dos casos: 1) cuando se busca el menor elemento y, 2) cuando se busca cualquier otro.

El primer caso es sumamente simple pues según las reglas que vimos previamente dicho elemento estará en la cima del montículo, del cual se tiene referencia, y por tanto la eficiencia será *O(1)*.

En el segundo caso, dado que no existe un orden entre los elementos que se encuentran en un mismo nivel del montículo, no es posible usar un algoritmo eficiente como en el caso del árbol binario de búsqueda. Por tanto es necesario recurrir a un recorrido por TODO el montículo (sea pre-orden, en-orden, o post-orden), el cual ya sabemos, tiene una eficiencia *O(n)*.

Búsqueda en un montículo binario

El segundo caso sería algo más o menos así (pre-orden):

```
buscar(elm e){
   encontrado = false
   buscar(top, e)
   return encontrado
}

buscar(nodo p, elm e){
   if (p.elm == e)
      encontrado = true
   else if (p.left != NULL)
      buscar(p.left, e)
   else if (p.right != NULL)
      buscar(p.right, e)
}
```

Borrado en un montículo binario

Se deben considerar dos casos: 1) cuando se borra la cima, es decir, el menor elemento y, 2) cuando se borra cualquier otro.

El primer caso requiere de dos pasos

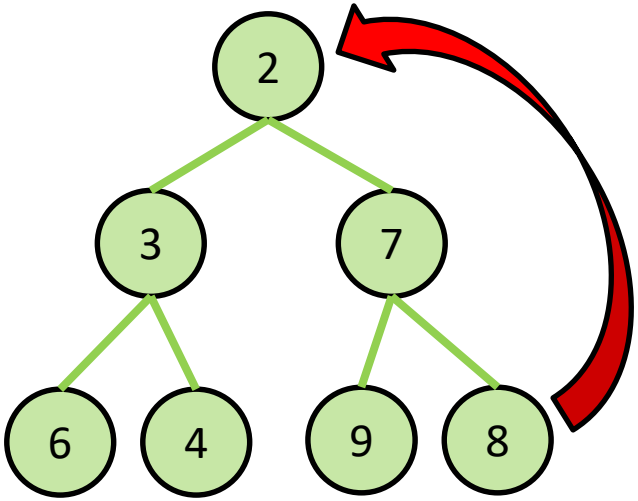
Paso 1: Reemplazar la cima del montículo por el elemento más abajo más a la derecha. Encontrar dicho elemento toma *O(log(n))* y hacer el reemplazo *O(1)*, es decir, todo junto es *O(log(n)*

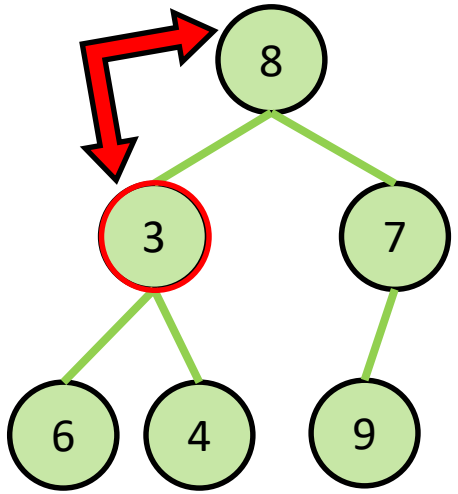
Paso 2: Similar a como cuando se inserta, se debe "bajar" dicho elemento por el camino de valores mínimos hasta la posición que le corresponda. En caso de empate se elige el izquierdo. En el peor de los casos se debe bajar hasta el último nivel, y por tanto, es O(log(n))

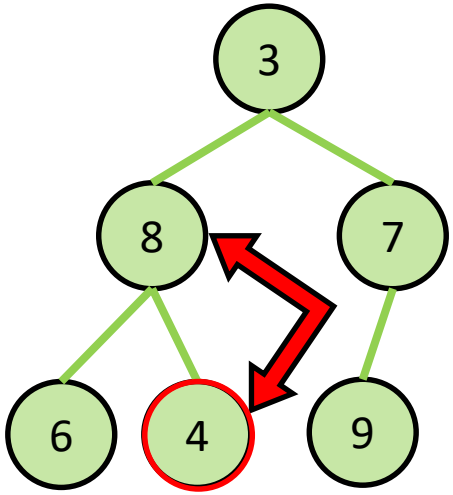
Al juntar los dos pasos se tiene *O(log(n))* + *O(log(n))* = *O(log(n))*

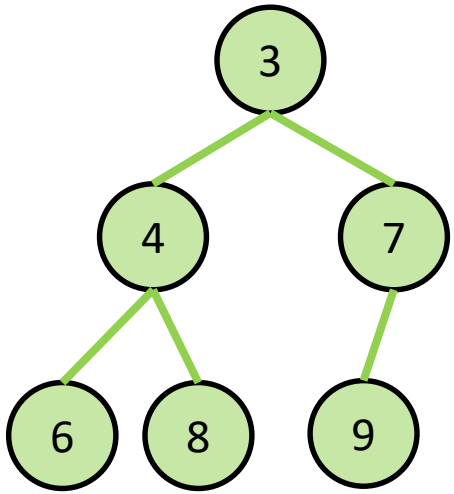
Borrado en un montículo binario

Ejemplo:









Borrado en un montículo binario

En el segundo caso de borrado (cualquier otro elemento que no sea la cima del montículo) se debe incluir un paso previo: buscar el elemento. Entonces:

Paso 0: Buscar el elemento, lo cual ya vimos es *O(n).* Si dicho elemento está lo más abajo más a la derecha del montículo se borra y listo, en caso contrario se sigue con los pasos 1 y 2

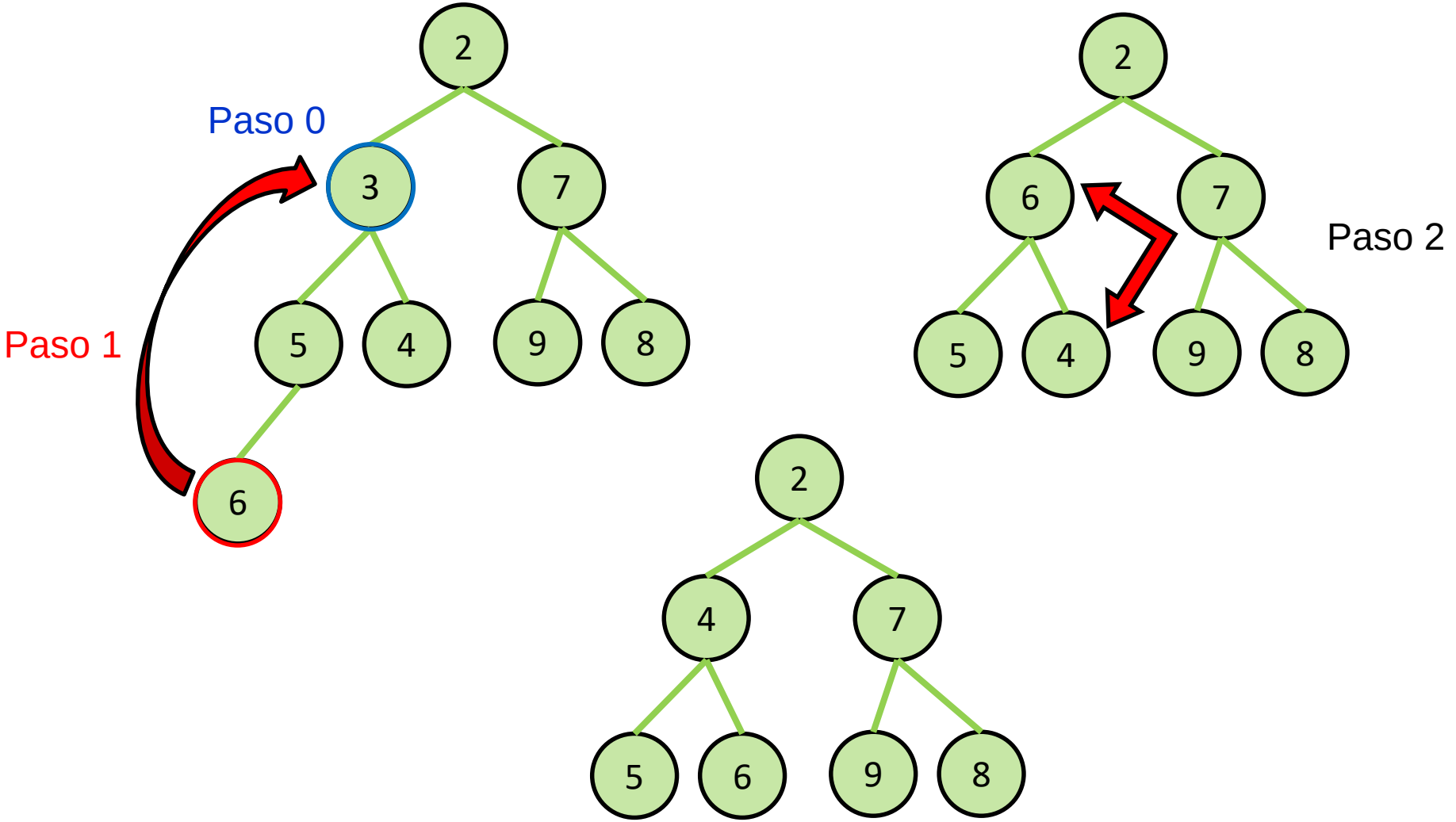
Paso 1: Reemplazar dicho elemento por el más abajo más a la derecha del montículo, lo cual en el peor de los casos es *O(log(n)).*

Paso 2: Se baja o sube dicho elemento hasta la posición que le corresponda, lo cual nuevamente es en el peor de los casos *O(log(n))*.

Al juntar los tres pasos se tiene $O(n)+O(log(n))+O(log(n)) = O(n)$

Borrado en un montículo binario

Ejemplo: borrar el 3



Colas con prioridad

El comportamiento del montículo binario, así como su eficiencia, hacen que muchos lenguajes (incluido Java) se basen en él para implementar las colas con prioridad.

En una cola con prioridad, como su nombre lo indica, el elemento a salir (pop) no es el primero en entrar (FIFO) sino aquel con mayor prioridad (valor mayor o menor según se necesite).

¿Por qué se usa esta alternativa en vez de usar listas enlazadas como en las colas FIFO?

Pues porque, si se hiciera así, el push se reduciría a *O(1)* pero el pop se aumentaría a *O(N),* o bien el pop se reduciría a *O(1)* pero el push se aumentaría a *O(N)*

Ordenamiento mediante heapSort

Un montículo binario puede ser usado para ordenar un arreglo. ¿Cómo?, muy sencillo: se insertan uno a uno los elementos del arreglo en un montículo binario hasta que este quede con los *n* elementos. Luego se extrae *n* veces la cima del montículo reinsertándolos al arreglo de forma secuencial.

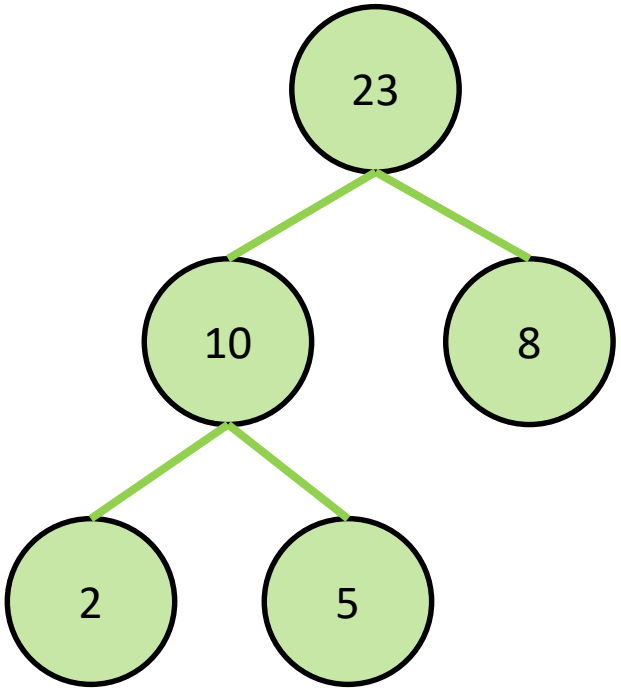
```
//sea A el arreglo que se desea ordenar
//y B un montículo binario inicialmente vacío
for (i=0; i<n; i++){
   B.add(A[i])
}
for (i=0; i<n; i++){
   A[i] = B.poll()
}
```

¿Cuál es la eficiencia de este algoritmo?

*n*O(log(n)) + n*(O(1)+O(log(n)) = O(n*log(n))*

Ejercicio

Si se insertan los siguientes elementos en un montículo binario descendente (en ese orden estricto): 10, 23, 8, 37, 5, 41, 2 y luego se borra dos veces la cima: ¿cómo queda el montículo?

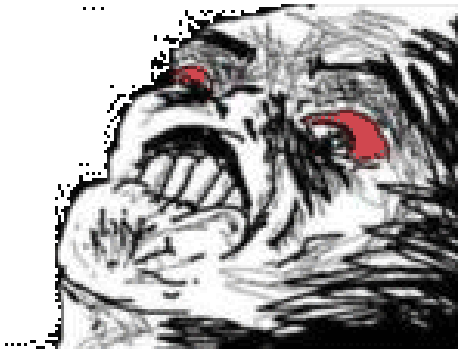


Dato de interés

Partiendo del hecho que un montículo binario corresponde a un árbol binario semi-completo recargado a la izquierda, la mayoría de los lenguajes lo implementan, no como un árbol, sino como un arreglo

Lo anterior garantiza las mismas eficiencias en las operaciones de inserción, búsqueda y borrado, pero con dos ventajas:

1. Mejores constantes
2. Menos consumo de memoria en la búsqueda



Queda de tarea analizar como serian la inserción, búsqueda y borrado con esta implementación.

Tabla resumen

Recapitulando la clase de hoy, tenemos que:

Estructura	Inserción	Indexación	Búsqueda	Borrado
Montículo binario implementado mediante árbol	*O(log(n))*	No aplica	*O(1)* refiriéndonos únicamente a la cima *O(n)* en caso contrario	*O(log(n))* refiriéndonos únicamente a la cima *O(n)* en caso contrario

Además analizamos que la eficiencia del algoritmo heapSort es *O(n*log(n))*