Resumen Algoritmos y Estructuras de Datos

Parcial Módulo 1

Gastón Ginestet 5/5/2017

Temas:

1.-Árboles Binarios

2.-Árboles AVL

3.-Árboles de expresión

4.-Árboles Generales

5.-Cola de prioridades(Heap)

1.- Arboles Binarios

Un árbol binario es una colección de nodos, tal que:

- puede estar vacía
- puede estar formada por un nodo distinguido R, llamado raíz y dos sub-árboles T1 y T2, donde la raíz de cada subárbol Ti está conectado a R por medio de una arista.

Descripción y terminología:

- Cada nodo puede tener a lo sumo dos nodos hijos.
- Cuando un nodo no tiene ningún hijo se denomina hoja.
- Los nodos que tienen el mismo nodo padre se denominan hermanos.

Conceptos a usar:

- Camino: desde n_1 hasta n_k , es una secuencia de nodos $n_1, n_2, ..., n_k$ tal que n_i es el padre de n_{i+1} , para $1 \leq i \leq k$.

  La longitud del camino es el número de aristas, es decir k-1.

  Existe un camino de longitud cero desde cada nodo a sí mismo.

  Existe un único camino desde la raíz a cada nodo.

- Profundidad: de n_i es la longitud del único camino desde la raíz hasta n_i.

  La raíz tiene profundidad cero

- Grado de n_i es el número de hijos del nodo n_i .

- Altura de n_i es la longitud del camino más largo desde n_i hasta

una hoja.

Las hojas tienen altura cero.

La altura de un árbol es la altura del nodo raíz.

• Ancestro/Descendiente: si existe un camino desde n_1 a n_2, se

dice que n_1 es ancestro de n_2 y n_2 es descendiente de n_1

• Árbol binario lleno: Dado un árbol binario T de altura h,diremos que T es lleno si cada nodo interno tiene grado 2 y todas las hojas están en el mismo nivel (h).

Es decir, recursivamente, T es lleno si :

1.- T es un nodo simple (árbol binario lleno de altura 0)

o

2.- T es de altura h y sus sub-árboles son llenos de altura h-1.

• Árbol binario completo: Dado un árbol binario T de altura h,diremos que T es completo si es lleno de altura h-1 y el nivel h se completa de izquierda a derecha.

• Cantidad de nodos en un árbol binario lleno:

Sea T un árbol binario lleno de altura h, la cantidad de nodos N es $(2^{h+1} - 1)$

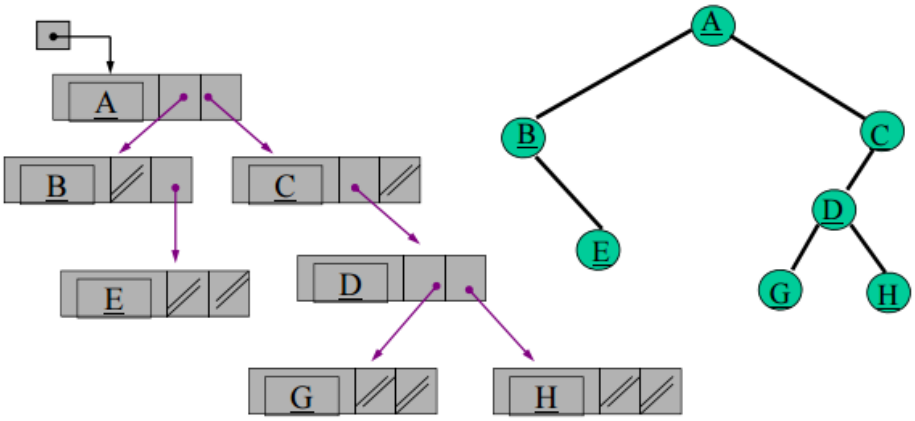
• Cantidad de nodos en un árbol binario completo:

Sea T un árbol binario completo de altura h, la cantidad de nodos N varía entre (2^{h}) y $(2^{h+1} - 1)$

Representación Hijo Izquierdo - Hijo Derecho:

Cada nodo tiene:

- Información propia del nodo
- Referencia a su hijo izquierdo
- Referencia a su hijo derecho



Recorridos:

Preorden

Se procesa primero la raíz y luego sus hijos, izquierdo y derecho.

```
public void printPreorden(){
        System.out.print(this.getDatoRaiz()+" ");
        if(!this.getHijoIzquierdo().esVacio())
                this.getHijoIzquierdo().printPreorden();
        if(!this.getHijoDerecho().esVacio())
                this.getHijoDerecho().printPreorden()
```

Inorden

Se procesa el hijo izquierdo, luego la raíz y último el hijo derecho

```
public void printInorden(){

      if(!this.getHijoIzquierdo().esVacio())

            this.getHijoIzquierdo().printPreorden();

      System.out.print(this.getDatoRaiz()+" ");

      if(!this.getHijoDerecho().esVacio())

            this.getHijoDerecho().printPreorden();}
```

Postorden

Se procesan primero los hijos, izquierdo y derecho, y luego la raíz

```
public void printPostorden(){
      if(!this.getHijoIzquierdo().esVacio())
            this.getHijoIzquierdo().printPreorden();

      if(!this.getHijoDerecho().esVacio())
            this.getHijoDerecho().printPreorden();

      System.out.print(this.getDatoRaiz()+" ");
}
```

Por niveles

Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

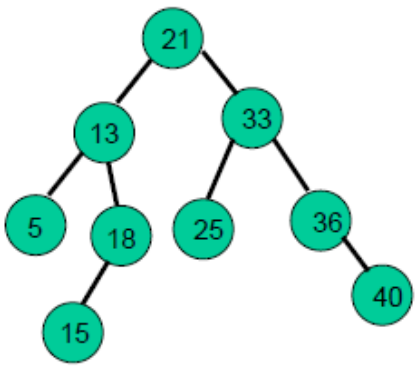
```
      public void recorridoPorNiveles(){
       ArbolBinario<T> arbol = null;
      ColaGenerica<ArbolBinario<T>> cola = new
ColaGenerica<ArbolBinario<T>>();
      cola.encolar(this);
      cola.encolar(null);
      while(!cola.esVacia()){
       arbol=cola.desencolar();
       if(arbol != null){
            System.out.print(arbol.getDatoRaiz()+" ");
            if(!arbol.getHijoIzquierdo().esVacio()){
                 cola.encolar(arbol.getHijoIzquierdo());
                if(!arbol.getHijoDerecho().esVacio()){
                       cola.encolar(arbol.getHijoDerecho());}
                 else
                if(!cola.esVacia()){
                     System.out.println();
                     cola.encolar(null); }}}}
```

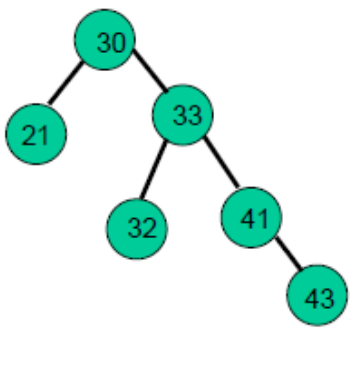
Árbol binario de búsqueda

Definición:

Un árbol binario de búsqueda es una colección de nodos conteniendo claves, que debe cumplir con una propiedad estructural y una de orden.
La propiedad estructural: es un árbol binario.

La propiedad de orden: para cada nodo *N* del árbol se cumple que todos los nodos ubicados en el subárbol izquierdo contienen claves menos que la clave del nodo *N* y los nodos ubicados en el subárbol derecho contienen claves mayores que la clave del nodo *N*





2.- Árboles AVL

Un árbol AVL(Adelson-Velskii-Landis) es un árbol binario de búsqueda que cumple con la condición de estar balanceado.

La propiedad de balanceo que cumple dice:

Para cada nodo del árbol , la diferencia de altura entre el subárbol izquierdo y el subárbol derecho es a lo sumo 1

Características de los Árboles AVL:

La propiedad de balanceo es fácil de mantener y garantiza que la altura del árbol sea de O(*log n*).

En cada nodo del árbol se guarda información de la altura.

La altura del árbol vacío es -1.

Al realizar operaciones de inserción o eliminación se debe actualizar la información de altura de los nodos y recuperar la propiedad de balanceo si fuera necesario, es decir, si hubiera sido destruida.

Big-O	Operations for 10 “things”	Operations for 100 “things”
O(1)	1	1
O(log n)	3	7
O(n)	10	100
O(n log n)	30	700
O(n^2)	100	10000
O(2^n)	1024	2^100
O(n!)	3628800	100!

Operaciones en un AVL

Búsqueda/Recuperación

Inserción

Eliminación

*Al insertar o eliminar un dato del AVL se puede perder la propiedad de balanceo

Se debe preservar el balanceo al realizarse estas operaciones sobre el árbol.

Inserción del el Árbol AVL

La inserción se realiza igual que en un árbol binario de búsqueda

Puede destruirse la propiedad del balanceo ⟶ Rebalancear el árbol

Proceso de inserción:

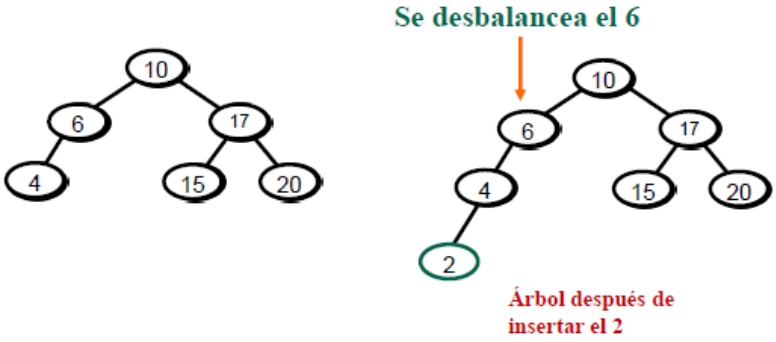
1. Buscar la **posición** en la que debe ser insertado el elemento (de la misma forma que en un Árbol Binario de Búsqueda).
2. Insertar el nuevo nodo con **altura** en **0**.
3. **Retroceder** en el camino obtenido en el paso 1, **verificando** el **balanceo** de los nodos, **rebalanceando** si fuera necesario y **actualizando** las alturas. *Recordar* que un árbol no está balanceado si para alguno de sus nodos, la diferencia de altura de sus subárboles es igual a 2.

Problemas de desbalanceo

Al insertar un elemento se actualiza la información de la altura de los nodos que están en el camino desde el nodo insertado a la raíz .

El desbalanceo sólo se produce en ese camino, ya que sólo esos nodos tienen sus subárboles modificados.

Ejemplo al insertar un nodo



Rebalanceo del árbol

Para restaurar el balanceo del árbol:

Se recorre el camino de búsqueda en orden inverso

Se controla el equilibrio/balanceo de cada nodo

Si está desbalanceo se realiza una modificación simple: rotación

Después de rebalancear el nodo, la inserción termina

Este proceso puede llegar a la raíz

Hay 4 casos posibles de desbalanceo a tener en cuenta , según donde se hizo la inserción . El nodo A es el nodo desbalanceado.

1. Inserción en el Subárbol IZQ del hijo IZQ de A

2. Inserción en el Subárbol DER del hijo IZQ de A

3. Inserción en el Subárbol IZQ del hijo DER de A

4. Inserción en el Subárbol DER del hijo DER de A

La solución para restaurar el balanceo es la ROTACIÓN

La rotación es una modificación simple de la estructura del árbol, que restaura la propiedad de balanceo, preservando el orden de los elementos.

Existen dos clases de rotaciones:

Rotación Simple: Casos 1 y 4 : inserción en el lado externo.

Rotación Doble: Casos 2 y 3 : inserción en el lado interno.

Soluciones simétricas: En cada caso, los subárboles están opuestos.

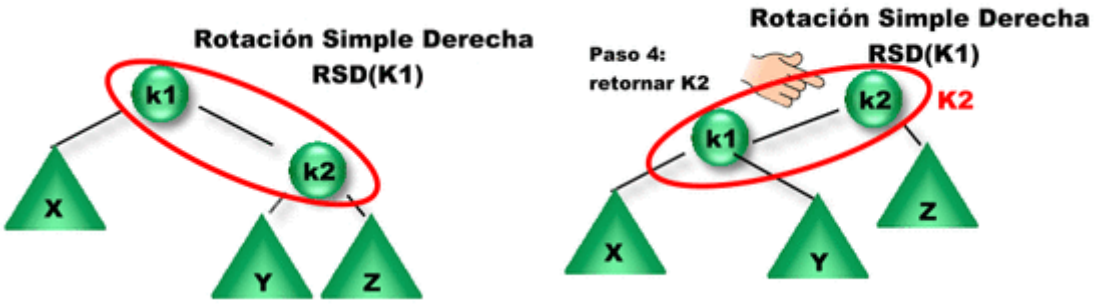
Rotación Simple Izquierda(Caso 1):

Rotación Simple Izquierda (K2)

```
(paso 1)  k1 = k2.hijoIzquierdo

(paso 2)  k2.hijoIzquierdo = k1.hijoDerecho

(paso 3)  k1.hijoDerecho = k2

(paso 4)  retornar K1 para que tome el lugar anterior de
K2 en el árbol
```

Rotación Simple Derecha(Caso 4):

Es simétrico al caso 1, el desbalanceo se produce hacia el lado derecho



Rotación Simple Derecha (K1)

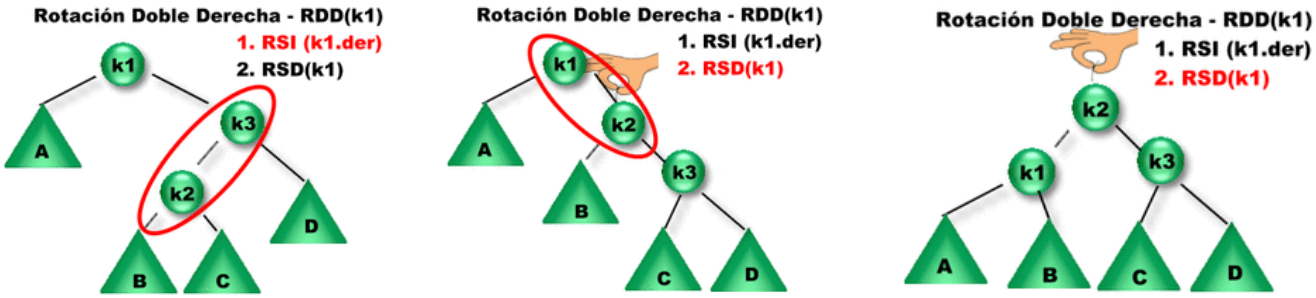
(paso 1) k2 = k1.hijoDerecho

(paso 2) k1.hijoDerecho = k2.hijoIzquierdo

(paso 3) k2.hijoIzquierdo = k1

(paso 4) retornar K2 para que tome el lugar anterior de K1 en el árbol

Rotación Doble Derecha(Caso 2):



Rotación Doble Derecha (K1)

(paso 1) K2 = K1.hijoDerecho

(paso 2) Rotación Simple Izquierda (K2)

(paso 3) Rotación Simple Derecha (K1)

Rotación Doble Izquierda(Caso 3):

Rotación Doble Izquierda (K2)

(paso 1) K1 = K2.hijoIzquierdo

(paso 2) Rotación Simple Derecha (K1)

(paso 3) Rotación Simple Izquierda(K2)

Eliminación en Árboles AVL

Proceso de eliminación:

1. Buscar la **posición** en donde se encuentra el elemento a eliminar (de la misma forma que en un Árbol Binario de Búsqueda).
2. Eliminar el elemento con el mismo procedimiento que en el Árbol Binario de Búsqueda.
3. **Retroceder** en el camino obtenido en el paso 1, **verificando** el **balanceo** de los nodos, **rebalanceando** si fuera necesario y **actualizando** las alturas. *Recordar* que un árbol no está balanceado si para alguno de sus nodos, la diferencia de altura de sus subárboles es igual a 2.

Conclusiones:

Las operaciones de inserción y eliminación de un nodo son similares a las de un árbol binario de búsqueda.

En ambas operaciones se debe actualizar la información de la altura y realizar rotaciones si es necesario.

La inserción provoca una única reestructuración.

La eliminación puede provocar varias reestructuraciones.

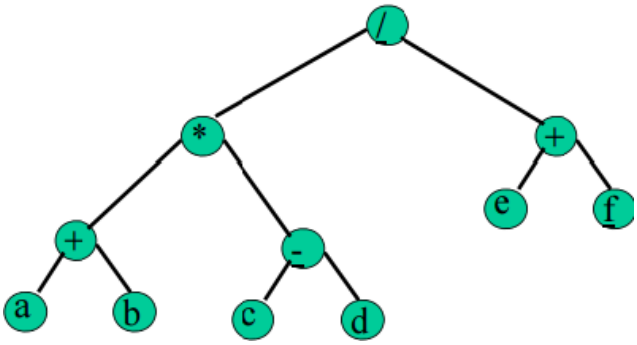
Las operaciones son de O(log n)

3.- Árbol de Expresión

Aplicaciones:

En compiladores para analizar, optimizar y traducir programas
Evaluar expresiones algebraicas o lógicas
No se necesita el uso de paréntesis

Traducir expresiones a notación sufija, prefija e infija
Recorriendo el árbol, obtenemos:



Inorden: (((a + b) * (c – d)) / (e + f))
Preorden: /*+ab-cd+ef
Postorden: ab+cd-*ef+/

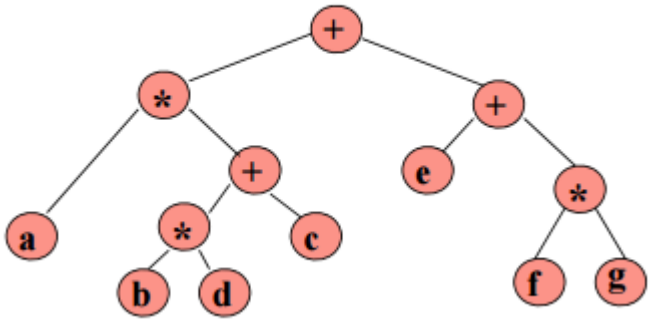
Construcción de un árbol de expresión

A partir de una:

1) Expresión postfija
2) Expresión prefija
3) Expresión infija

Expresión algebraica :

a*(b*d +c) + (e+ f*g)



Expresión prefija: + * a + * b d c + e * f g
Expresión postfija: a b d * c + * e f g * + +
Expresión infija: ((a *((b * d) + c)) + (e + (f * g)))

1) Construcción de un árbol de expresión a partir de una expresión postfija

Algoritmo:

tomo un carácter de la expresión
mientras (existe carácter) hacer
si es un operando → creo un nodo y lo apilo.
si es un operador (lo tomo como la raíz de los dos últimos nodos creados)
→ - creo un nodo R,
- desapilo y lo agrego como hijo derecho de R
- desapilo y lo agrego como hijo izquierdo de R
- apilo R.
tomo otro carácter
fin

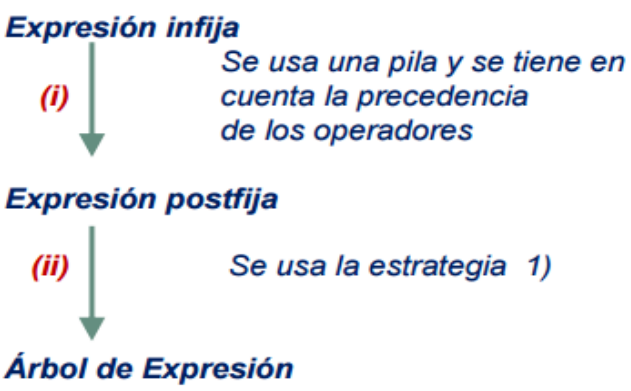
2) Construcción de un árbol de expresión a partir de una expresión prefija

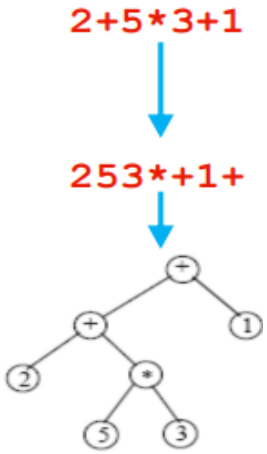
Algoritmo:

ArbolExpresión (A: ArbolBin, exp: string)

si exp nulo → nada.
si es un operador → - creo un nodo raíz R
- ArbolExpresión (subArbIzq de R, exp (sin 1° carácter))
- ArbolExpresión (subArbDer de R, exp (sin 1° carácter))
si es un operando → creo un nodo (hoja)

3) Construcción de un árbol de expresión a partir de una expresión infija





-***Convertir una expresión infija en árbol de expresión: se debe convertir la expresión infija en postfija (i) y a partir de ésta, construir el árbol de expresión (ii).***

(i) Estrategia del Algoritmo para convertir exp. infija en postfija :

a) si es un operando → se coloca en la salida.
b) si es un operador → se maneja una pila según la prioridad del operador en relación al tope de la pila

operador con > prioridad que el tope –> se apila
operador con <= prioridad que el tope –> se desapila elemento colocándolo en la salida.
Se vuelve a comparar el operador con el tope de la pila

c) si es un "(" , ")" → "(" se apila
")" se desapila todo hasta el "(", incluído éste

d) cuando se llega al final de la expresión, se desapilan todos los elementos llevándolos a la salida, hasta que la pila quede vacía.

Operadores ordenados de mayor a menor según su prioridad:

^ (potencia)
***, / (multiplicación y división)**
+, - (suma y resta)

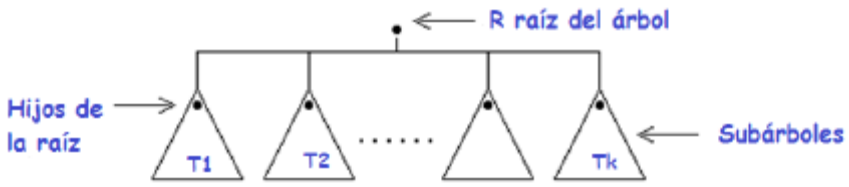
Los " (" siempre se apilan como si tuvieran la mayor prioridad y se desapilan sólo cuando aparece un ") " .

4.- Árboles Generales

Los árboles se pueden definir de dos formas: recursiva y no recursivamente . La definición no recursiva es la mas directa,por lo que comenzamos por ellas. La formulación recursiva nos permite escribir algoritmos simples para manipular árboles.

Se definen a los árboles no recursivos como un conjunto de nodos y otro de aristas orientados que los conectan. Los padres e hijos se definen de forma natural. Una arista orientada conecta el padre con un hijo.

Se definen a los arboles recursivos a aquellos que es o bien vacío o consiste en una raíz y cero o mas subárboles no vacíos $T_1, T_{2,} \ldots, T_k$ cada una de cuyas raíces está conectada por medio de una arista con la raíz.



Descripción y terminología de los Árboles Generales:

Grado del árbol es el grado del nodo con mayor grado.

Árbol lleno: Dado un árbol T de grado k y altura h, diremos que T es lleno si cada nodo interno tiene grado k y todas las hojas están en el mismo nivel (h).

Es decir, recursivamente, T es lleno si :

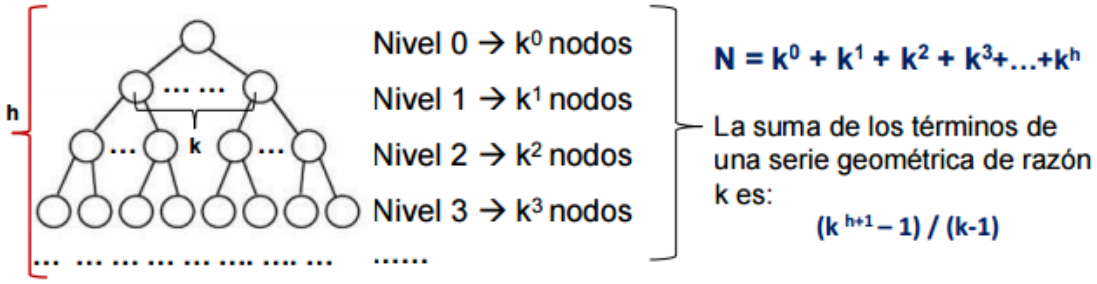
1.- T es un nodo simple (árbol lleno de altura 0), o

2.- T es de altura h y todos sus sub-árboles son llenos de altura h-1.

Árbol completo: Dado un árbol T de grado k y altura h,diremos que T es completo si es lleno de altura h-1 y el nivel h se completa de izquierda a derecha.

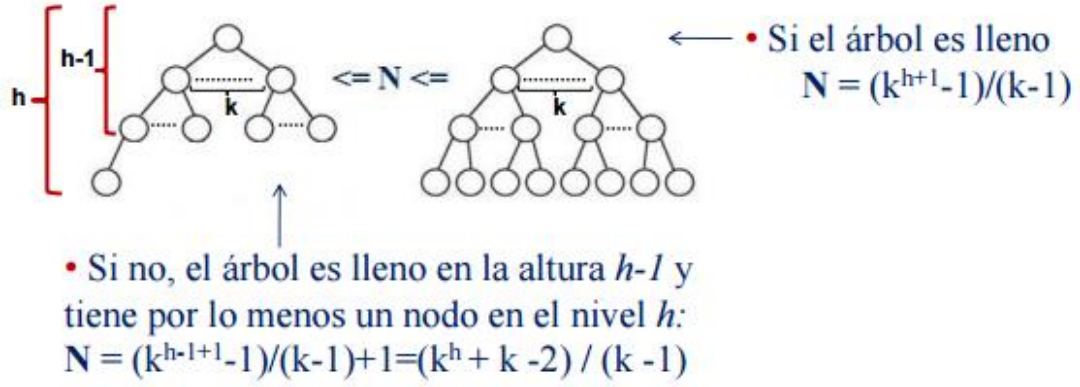
• Cantidad de nodos en un árbol lleno:

Sea T un árbol lleno de grado k y altura h, la cantidad de nodos N es $(k^{h+1} – 1) / (k-1)$ ya que:

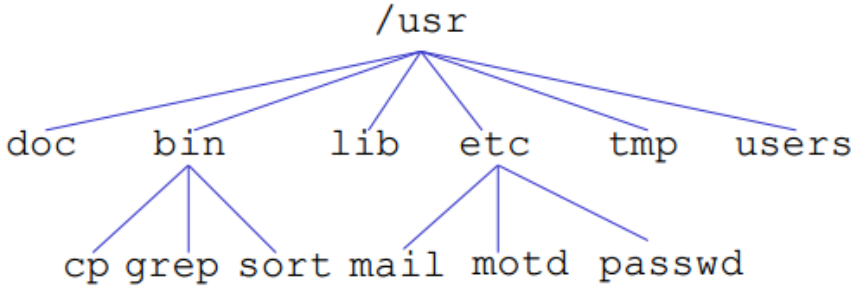


Cantidad de nodos en un árbol completo:

Sea T un árbol completo de grado k y altura h, la cantidad de nodos N varía entre $(k^h + k - 2) / (k-1)$ y $(k^{h+1} – 1) / (k-1)$ ya que …



Ejemplo: Sistema de archivos



Representaciones

Lista de hijos

Cada nodo tiene:

Información propia del nodo

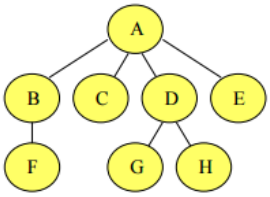
Una lista de todos sus hijos

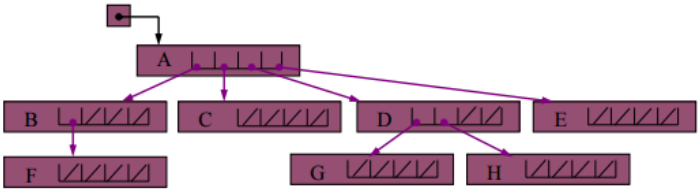
La lista de hijos, puede estar implementada a través de:

Arreglos

Desventaja: espacio ocupado

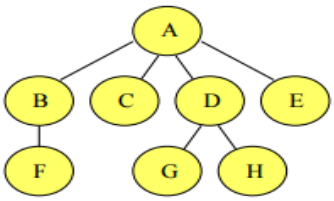
.

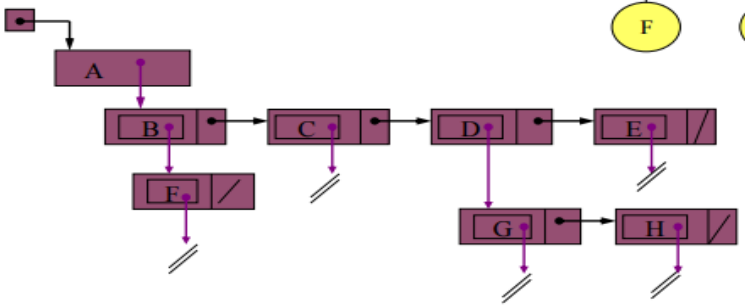




Listas dinámicas

Mayor flexibilidad en el uso.





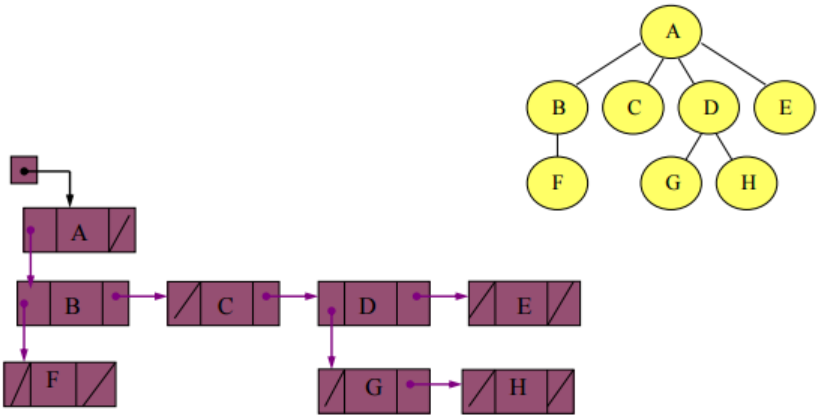
Hijo más izquierdo y hermano derecho

Cada nodo tiene:

Información propia del nodo

Referencia al hijo más izquierdo

Referencia al hermano derecho



Recorridos en Árboles Generales

Recorrido PreOrden:

```
public ListaEnlazadaGenerica<T> preOrden(){
	ListaEnlazadaGenerica<T> lista = new ListaEnlazadaGenerica<>();
	preOrden(lista);
	return lista;
	}

private void preOrden(ListaGenerica<T> lista){
	lista.agregarFinal(this.getDatoRaiz());
	ListaGenerica<ArbolGeneral<T>> listaHijos = this.getHijos();
	listaHijos.comenzar();
	while(!listaHijos.fin()){
		listaHijos.proximo().preOrden(lista);
	}
}
```

Recorrido PostOrden:

```
public ListaEnlazadaGenerica<T> postOrden(){
	ListaEnlazadaGenerica<T> lista = new ListaEnlazadaGenerica<T>();
	postOrden(lista);
	return lista;
	}

private void postOrden(ListaGenerica<T> lista){
	ListaGenerica<ArbolGeneral<T>> listaHijos = this.getHijos();
	listaHijos.comenzar();
	while(!listaHijos.fin()){
		listaHijos.proximo().preOrden(lista);
	}
	lista.agregarFinal(this.getDatoRaiz());

}
```

Recorrido InOrden:

```
	public ListaEnlazadaGenerica<T> inOrden(){
		ListaEnlazadaGenerica<T> lista = new ListaEnlazadaGenerica<T>();
		inOrden(lista);
		return lista;
		}

	private void inOrden(ListaGenerica<T> lista){
		ListaGenerica<ArbolGeneral<T>> listaHijos = this.getHijos();
		listaHijos.comenzar();
		 if(!listaHijos.esVacia()){
			 listaHijos.proximo().inOrden(lista);
		 }

		 lista.agregarFinal(this.getDatoRaiz());

		 while(!listaHijos.fin()){
				listaHijos.proximo().inOrden(lista);
			}
		 }
```

Recorrido Por Niveles:

```
	public ListaEnlazadaGenerica<T> recorridoPorNiveles (){
		ListaEnlazadaGenerica<T> lista = new ListaEnlazadaGenerica<T>();
		recorridoPorNiveles(lista);
		return lista;
		}

	private void recorridoPorNiveles(ListaGenerica<T> lista){
		ArbolGeneral<T> arbol= null;
		ColaGenerica<ArbolGeneral<T>> cola = new
ColaGenerica<ArbolGeneral<T>>();
		cola.encolar(this);
		cola.encolar(null);
		while(!cola.esVacia()){
			arbol=cola.desencolar();
			if(arbol != null){
				lista.agregarFinal(arbol.getDatoRaiz());
				ListaGenerica<ArbolGeneral<T>> lHijos =
arbol.getHijos();
				lHijos.comenzar();
				while(!lHijos.fin()){
					cola.encolar(lHijos.proximo());
				}}
			else
				if(!cola.esVacia()){
					cola.encolar(null);}}}
```

5.- Colas de prioridad

Definición

Una cola de prioridad es una estructura de datos que permite al menos dos operaciones:

Insert

Inserta un elemento en la estructura

DeleteMin

Encuentra , recupera y elimina el elemento mínimo

Cola de impresión

Sistema operativo

Algoritmos de Ordenación

Implementaciones

Lista ordenada

Insert tiene O(N) operaciones y DeleteMin tiene O(1)operaciones

Lista no ordenada

Insert tiene O(1) operaciones yDeleteMin tiene O(N)operaciones

Árbol Binario de Búsqueda

Insert y DeleteMin tienen en promedio O(log N) operaciones

Heap Binaria: Es una implementación de colas de prioridad que no usa punteros y permite implementar ambas operaciones con O(log N) operaciones en el peor caso.

Cumple con dos propiedades:

Propiedad estructural

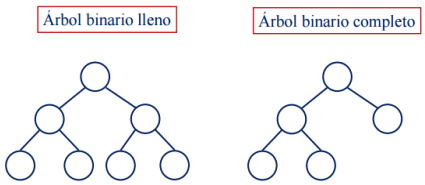
Propiedad de orden

Propiedad estructural

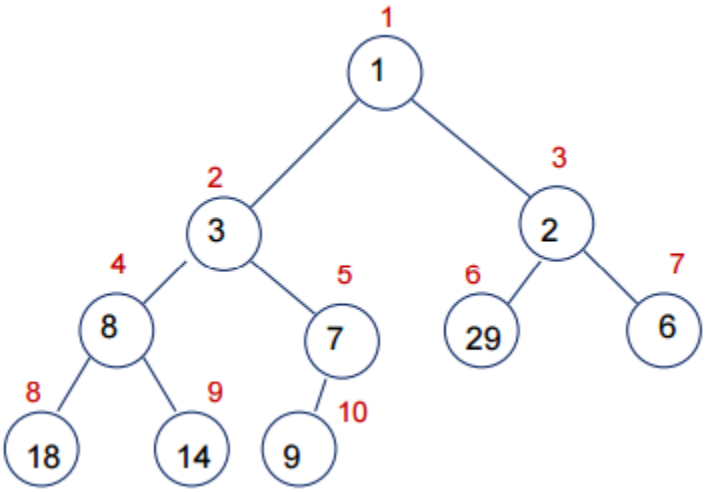
Una heap es un árbol binario completo

En un árbol binario lleno de altura *h*, los nodos internos tienen exactamente 2 hijos y las hojas tienen la misma profundidad

Un árbol binario completo de altura *h* es un árbol binario lleno de altura *h-1* y en el nivel *h*, los nodos se completan de izquierda a derecha



Ejemplo:



El número de nodos *n* de un árbol binario completo de altura *h*, satisface:

$$2^h \leq n \leq (2^{h+1} - 1)$$

Demostración:

Si el árbol es lleno, $n= 2^{h+1}-1$

Si no, el árbol es lleno en la altura h-1 y tiene por lo menos un nodo en el nivel h:

$n= 2^{h-1+1} - 1+ 1 = 2^h$

La altura *h* del árbol es de *O(log n)*

Dado que un árbol binario completo es una estructura de datos regular, puede almacenarse en un arreglo, tal que:

La raíz está almacenada en la posición 1

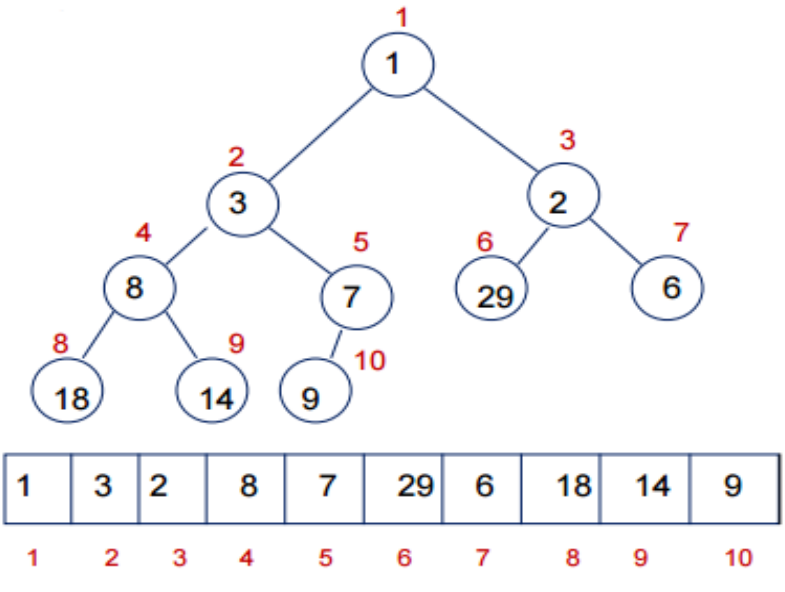
Para un elemento que está en la posición i:

El hijo izquierdo está en la posición 2*i

El hijo derecho está en la posición 2*i + 1

El padre está en la posición [i/2]

El árbol que vimos como ejemplo, puede almacenarse de la siguiente manera:



Propiedad de orden

MinHeap

El elemento mínimo está almacenado en la raíz

El dato almacenado en cada nodo es menor o igual al de sus hijos

MaxHeap

Se usa la propiedad inversa

Implementación de Heap

Una heap H consta de:

Un arreglo que contiene los datos

Un valor que me indica el número de elementos almacenados

Ventaja:

No se necesita usar punteros

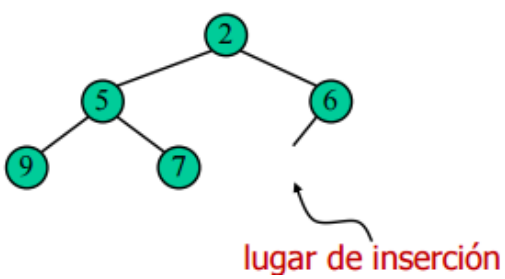
Fácil implementación de las operaciones

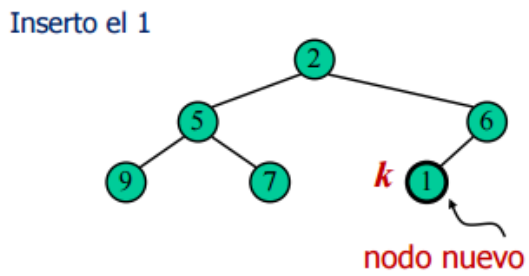
Operación: Insert

El dato se inserta como último ítem en la heap

La propiedad de la heap puede ser violada

Se debe hacer un filtrado hacia arriba para restaurar la propiedad de orden





Filtrado hacia arriba(Percolate Up)

El filtrado hacia arriba restaura la propiedad de orden intercambiando k a lo largo del camino hacia arriba desde el lugar de inserción.

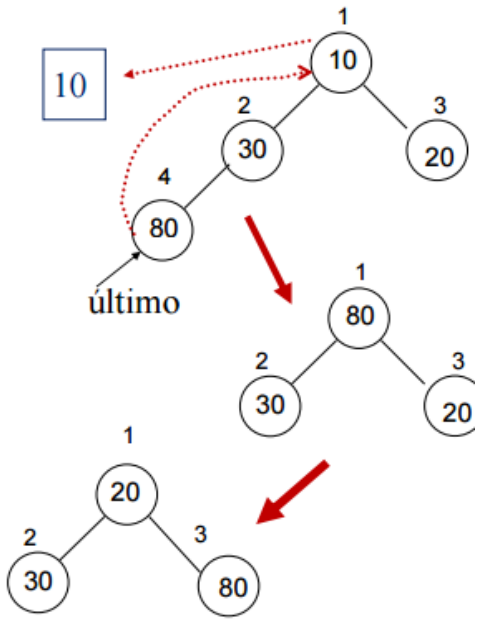
El filtrado termina cuando la clave k alcanza la raíz o un nodo cuyo padre tiene una clave menor.

Ya que el algoritmo recorre la altura de la heap, tiene O(log n) intercambios.

En Java: Operación de Insert y Percolate Up

```
insert (Heap h, Comparable x) {

h.tamaño = h.tamaño + 1;

h.dato[h.tamaño] = x;

percolate_up ( h , h.tamaño )} // end del insert


percolate_up (Heap h, Integer i) {

temp = h.dato[i];

while (i/2 > 0 & h.dato[i/2] > temp ) {

     h.dato[i] = h.dato[i/2];

     i = i/2;}

h.dato[ i ] = temp; // ubicación correcta del elemento a
filtrar

} // end del percolate_up
```

Operación: DeleteMin



Guardo el dato de la raíz

Elimino el último elemento y lo almaceno en la raíz

Se debe hacer un filtrado hacia abajo para restaurar la propiedad de orden

Filtrado hacia abajo(Percolate Down)

Es similar al filtrado hacia arriba

El filtrado hacia abajo restaura la propiedad de orden intercambiando el dato de la raíz hacia abajo a lo largo del camino que contiene los hijos mínimos

El filtrado termina cuando se encuentra el lugar correcto dónde insertarlo

Ya que el algoritmo recorre la altura de la heap, tiene O(log n)

operaciones de intercambio.

En java:

```
delete_min ( Heap h; Comparable e) {

 if (h.tamaño > 0 )   { //  la heap no está vacía
    e := h.dato[1] ;
    h.dato[1] := h.dato[h.tamaño] ;
    h.tamaño :=  h.tamaño - 1;
    percolate_down ( h ; 1);

 }
}  // end del delete_min
```

```
percolate_down ( Heap h, int p) {

    candidato := h.dato[ p ]
    stop_perc := false;
    while (  2* p <= h.tamaño )  and ( not stop_perc) {
        h_min := 2 * p;   //  buscar el hijo con clave menor
        if h_min <> h.tamaño   then
        if ( h.dato[h_min +1] < h.dato[h_min] )
                                          h_min := h_min + 1
        if  candidato  > h.dato [h_min] { // percolate_down
                                          h.dato [p] := h.dato[ h_min ]
                                          p := h_min;
                                          }
        else   stop_perc := true;
     }   // end  { while }
    h.dato[p] := candidato;
}    // end {percolate_down }
```

Otras operaciones

DecreaseKey(x, Δ, H)

Decrementa la clave que está en la posición x de la heap H, en una cantidad Δ

IncreaseKey(x, Δ, H)

Incrementa la clave que está en la posición x de la heap H, en una cantidad Δ

DeleteKey(x)

Elimina la clave que está en la posición x

Puede realizarse:

DecreaseKey(x,∞, H)

DeleteMin(H)

¿Cómo construir una heap a partir de una lista de elementos?

Para construir una heap a partir de una lista de *n* elementos:

1)Se pueden insertar los elementos de a uno

⟶se realizan (*n log n*) operaciones en total

2)Se puede usar un algoritmo de orden lineal, es decir,proporcional a los n elementos BuildHeap

Insertar los elementos desordenados en un árbol binario completo

Filtrar hacia abajo cada uno de elementos

Algoritmo BuildHeap

Para filtrar:

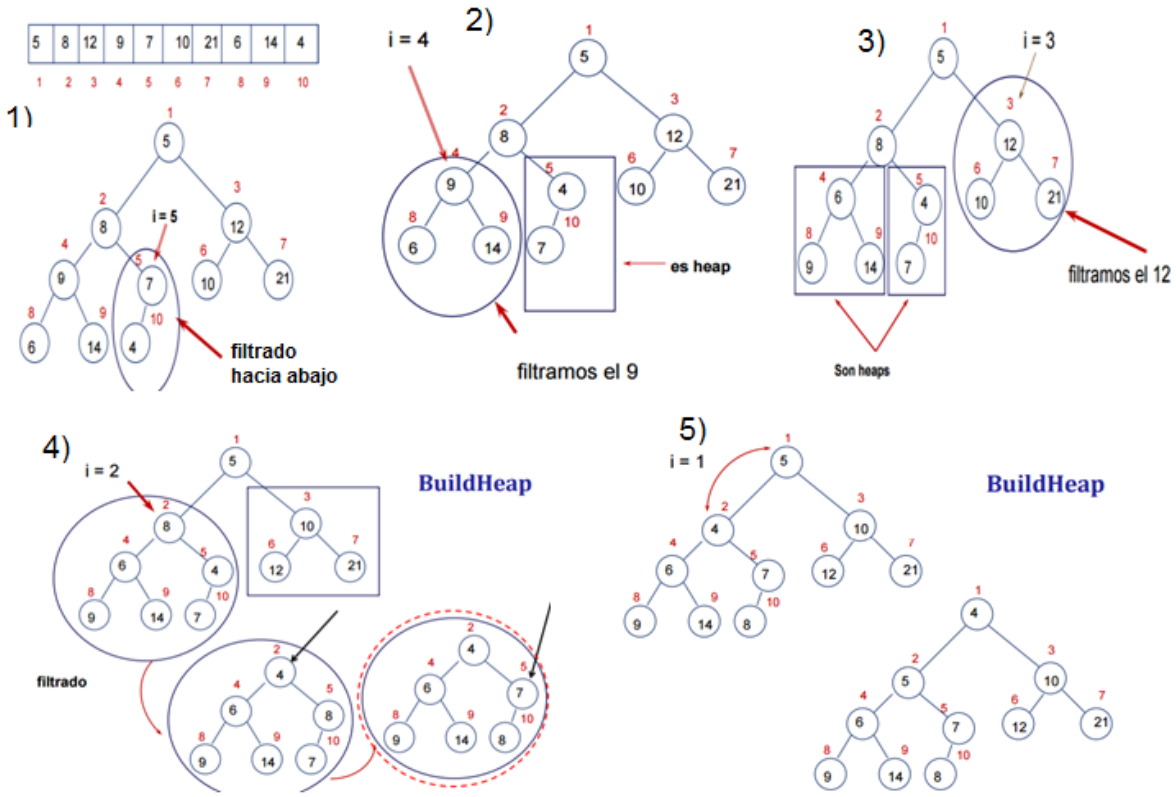
1)se elige el menor de los hijos

2)se compara el menor de los hijos con el padre

Se empieza filtrando desde el elemento que está en la posición (tamaño/2):

*se filtran los nodos que tienen hijos

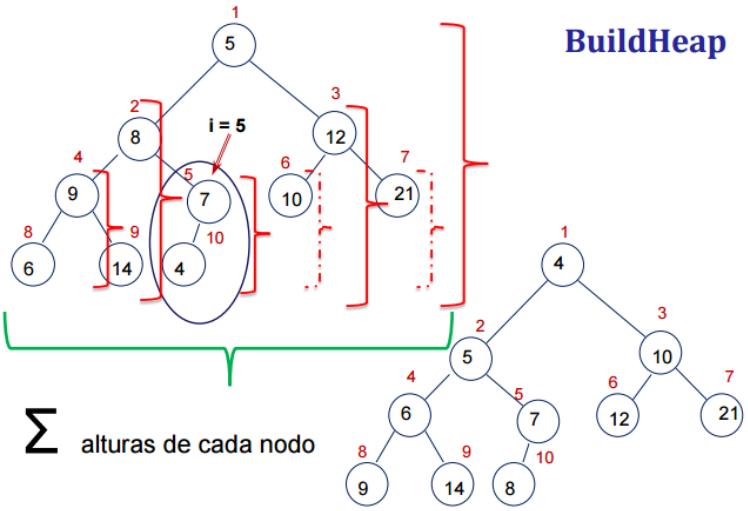
*el resto de los nodos son hojas



Cantidad de operaciones requeridas

En el filtrado de cada nodo recorremos su altura

Para acotar la cantidad de operaciones (tiempo de ejecución) del algoritmo BuildHeap, debemos calcular la suma de las alturas de todos los nodos



Teorema:

En un árbol binario lleno de altura h que contiene $2^{h+1} - 1$ nodos, la suma de las alturas de los nodos es: $2^{h+1} - 1 - (h + 1)$

Demostración:

Un árbol tiene 2^i nodos de altura h – i

$$S = \sum_{i=0}^{h} 2^i(h-i)$$

$S = h + 2\,(h-1) + 4\,(h-2) + 8\,(h-3) + \ldots\ldots 2^{h-1}(1)$

$$S = h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots\ldots\ldots + 2^{h-1}(1) \qquad (A)$$
$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots\ldots\ldots + 2^{h}(1) \qquad (B)$$

Restando las dos igualdades (B) – (A)

$$S = -h + 2(h-(h-1)) + 4((h-1)-(h-2)) + 8((h-2)-(h-3)) + \ldots + 2^{h-1}(2-1) + 2^{h}$$
$$S = -h + 2 + 4 + 8 + 16 + \ldots\ldots\ldots + 2^{h-1} + 2^{h}$$
$$S + 1 = -h + 1 + 2 + 4 + 8 + 16 + \ldots\ldots\ldots + 2^{h-1} + 2^{h}$$
$$S + 1 = -h + (2^{h+1} - 1)$$
$$S = (2^{h+1} - 1) - (h + 1)$$

Un árbol binario completo no es un árbol binario lleno, pero el resultado obtenido es una cota superior de la suma de las alturas de los nodos en un árbol binario completo

Un árbol binario completo tiene entre 2^h y 2^{h+1}- 1 nodos, el teorema implica que esta suma es de O(n) donde n es el número de nodos.

Este resultado muestra que la operación BuildHeap es lineal

Ordenación de vectores usando Heap

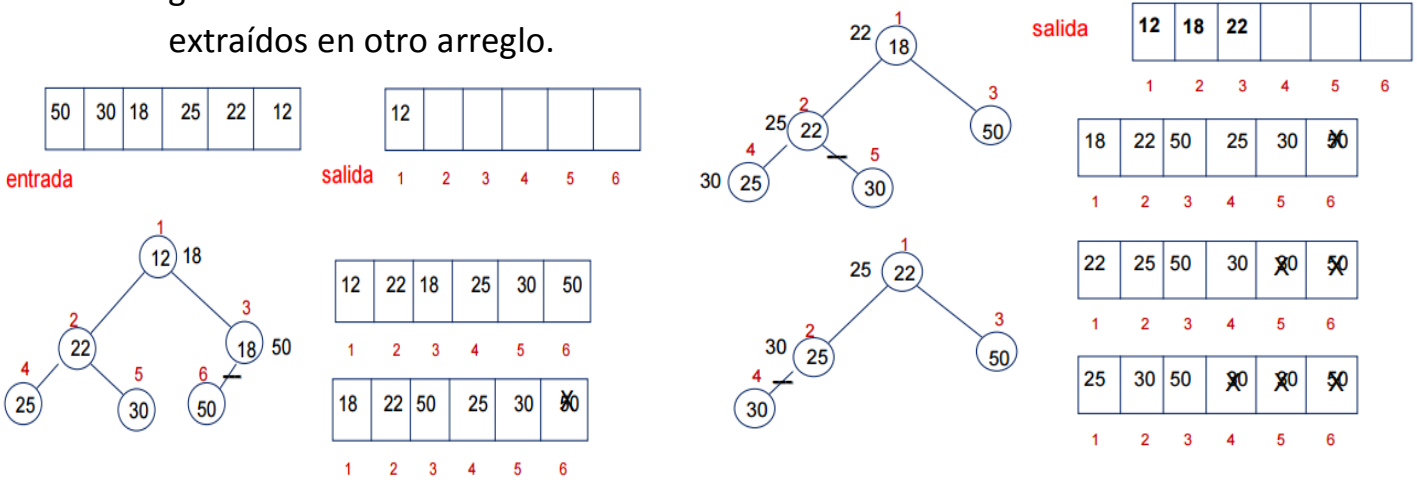
Dado un conjunto de n elementos y se los quiere ordenar en forma creciente, existen dos alternativas:

a) Algoritmo que usa una heap y requiere una cantidad aproximada de (n log n) operaciones.

Construir una MinHeap, realizar n DeleteMin operaciones e ir guardando los elementos extraídos en otro arreglo.

Desventaja: requiere el doble de espacio

Ejemplo: Construir una MinHeap, realizar 6 DeleteMin operaciones e ir guardando los elementos extraídos en otro arreglo.

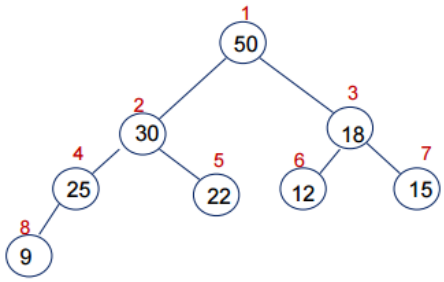


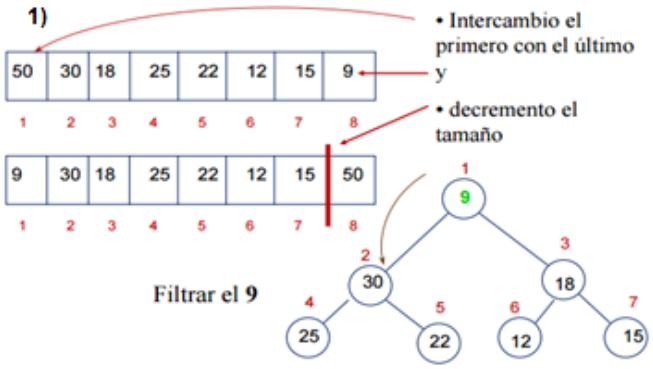
Ordenación de vectores usando Heap:Algoritmo HeapSort

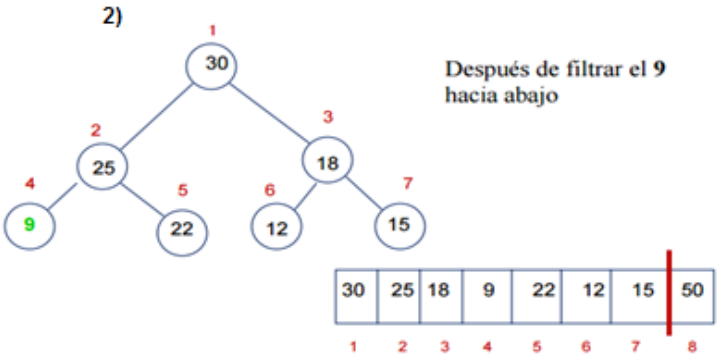
b) Algoritmo HeapSort que requiere una cantidad aproximada de (n log n) operaciones, pero menos espacio.

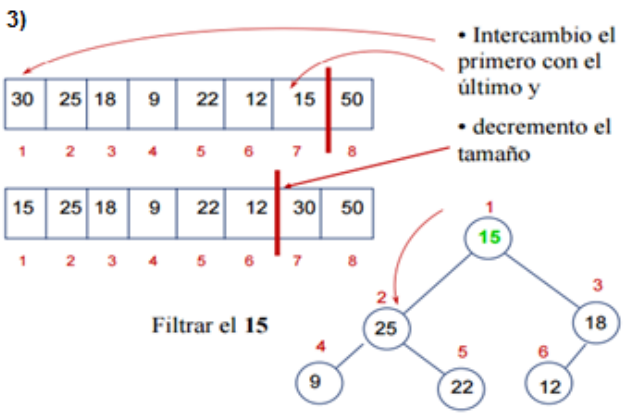
Construir una MaxHeap con los elementos que se desean ordenar,intercambiar el último elemento con el primero, decrementar el tamaño de la heap y filtrar hacia abajo. Usa sólo el espacio de almacenamiento de la heap.

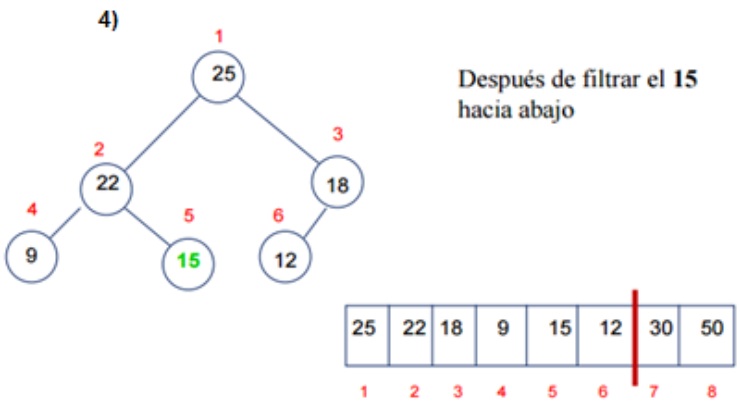
Ejemplo:



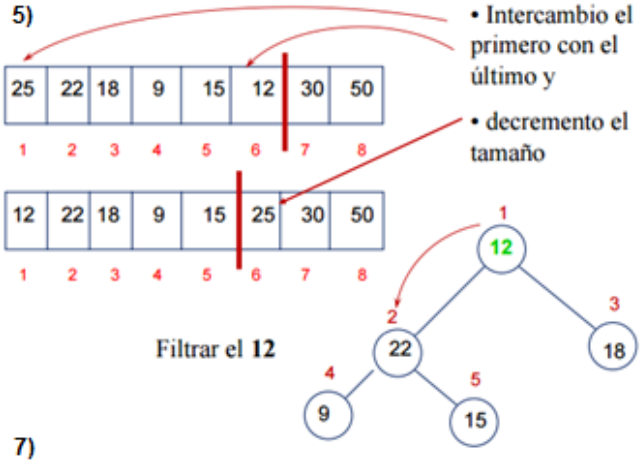






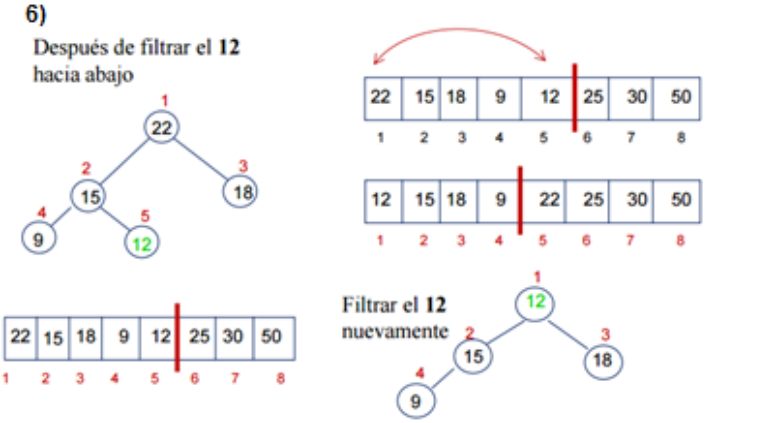


5)



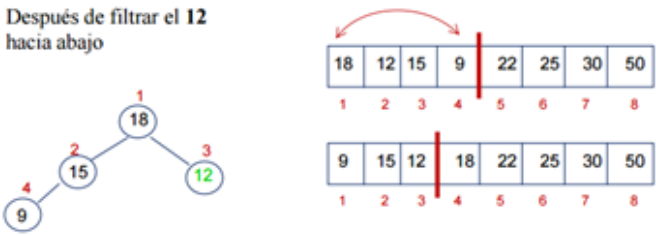
6)

Después de filtrar el **12** hacia abajo



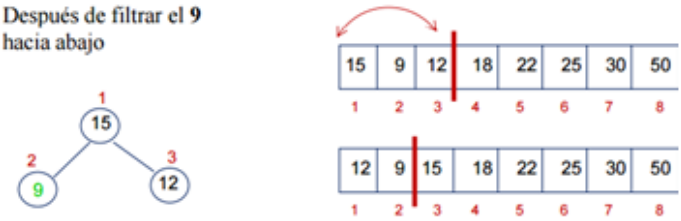
7)

Después de filtrar el **12** hacia abajo



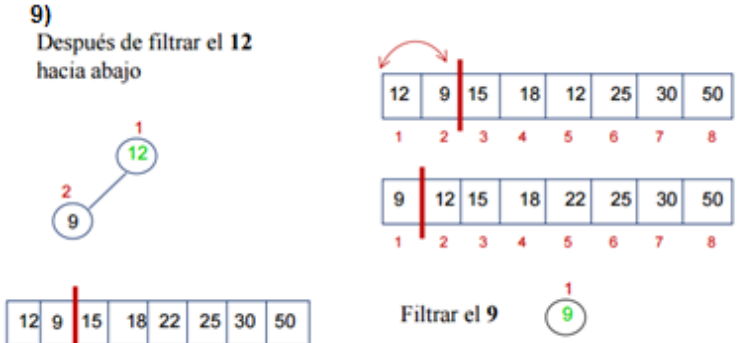
8)

Después de filtrar el **9** hacia abajo



9)

Después de filtrar el **12** hacia abajo



10)