Colas de prioridad

Agenda

- Aplicaciones
- Definición
- Distintas implementaciones
- Heap Binaria
  - Propiedad Estructural
  - Propiedad de Orden
  - Implementación
- Operaciones: Insert, DeleteMin, Operaciones adicionales
- Construcción de una Heap: operación BuildHeap
  - Eficiencia
- HeapSort

Aplicaciones

- Cola de impresión
- Sistema Operativo
- Algoritmos de Ordenación

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

Implementaciones

- ✔ Lista ordenada
  - Insert tiene O(N) operaciones
  - DeleteMin tiene O(1) operaciones
- ✔ Lista no ordenada
  - Insert tiene O(1) operaciones
  - DeleteMin 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

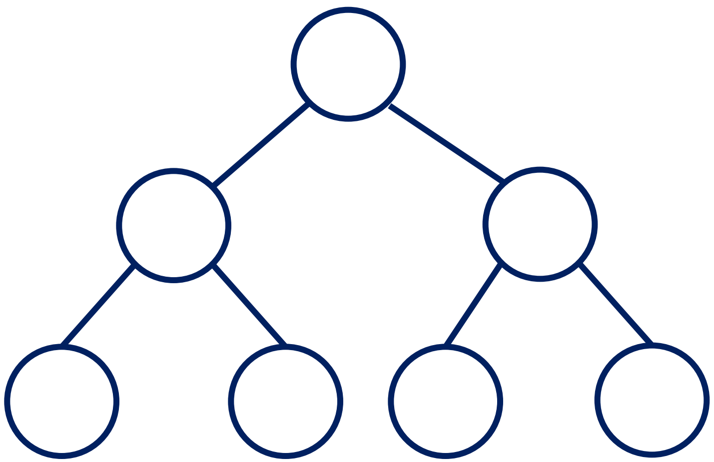
Una heap es un árbol binario completo

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

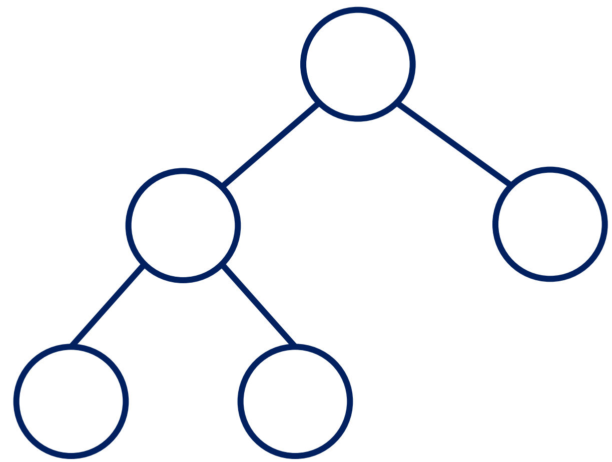
- Un árbol binario completo de altura *h* es un árbol binario lleno de altura *h-1* y en el nivel *h*, los nodos se completan de izquierda a derecha

Propiedad estructural (cont.)

Árbol binario lleno

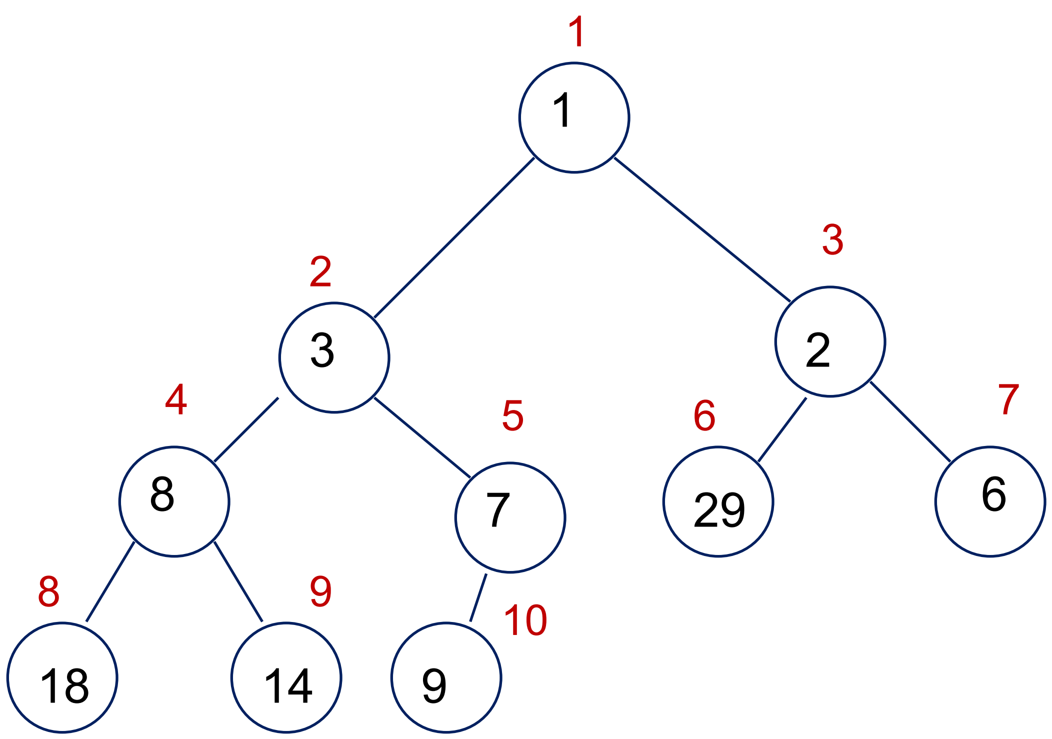


Árbol binario completo



Propiedad estructural (cont.)

Ejemplo:



Propiedad estructural (cont.)

- 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:*

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

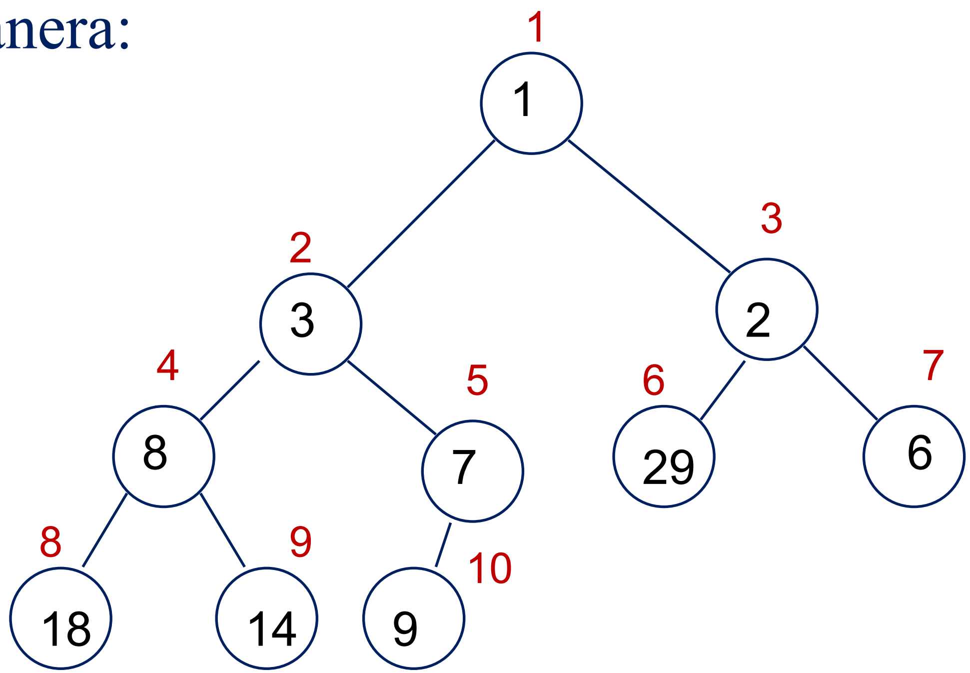
La altura *h* del árbol es de **O(log n)**

Propiedad estructural (cont.)

- 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 $\lfloor i/2 \rfloor$

Propiedad estructural (cont.)

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



1	3	2	8	7	29	6	18	14	9
1	2	3	4	5	6	7	8	9	10

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:

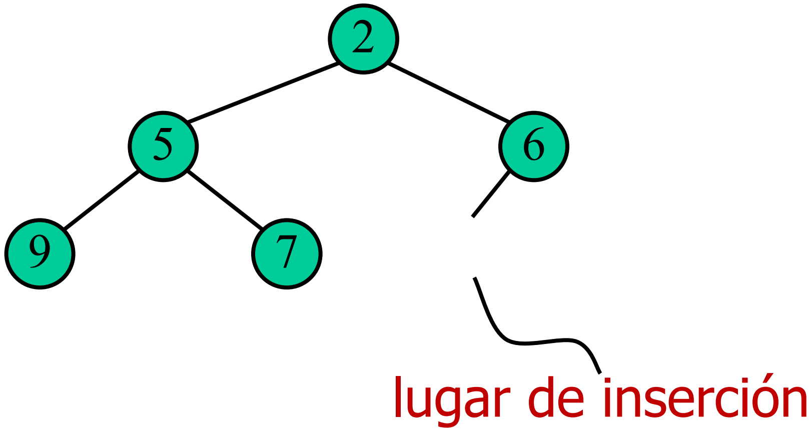
- ✓ *Un arreglo que contiene los datos*
- ✓ *Un valor que me indica el número de elementos almacenados*

Ventaja:

- ✓ No se necesita usar punteros
- ✓ Fácil implementación de las operaciones

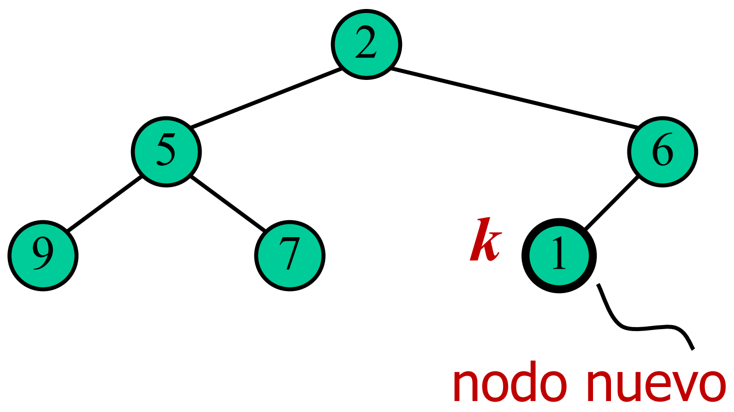
Operación: Insert

- El dato se inserta como último ítem en la heap
    - La propiedad de la heap puede ser violada



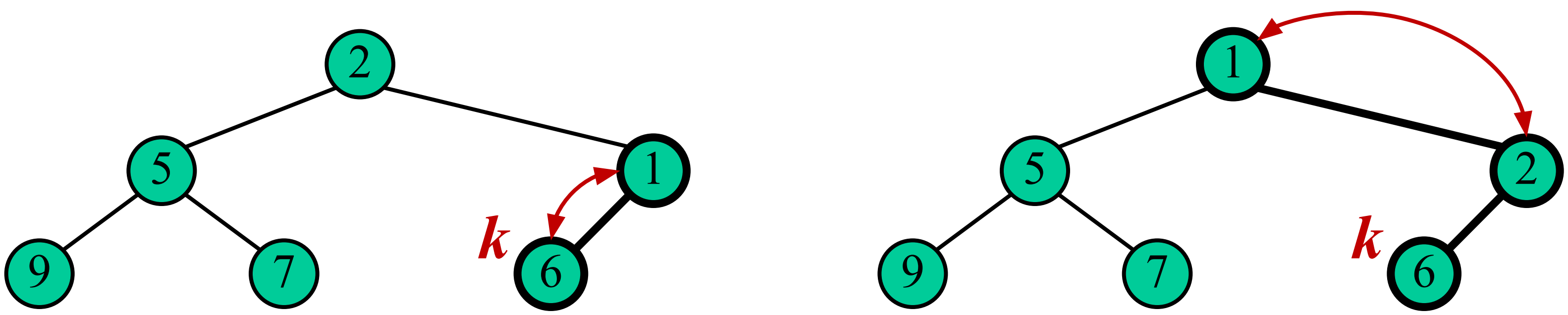
- Se debe hacer un filtrado hacia arriba para restaurar la propiedad de orden

Inserto el 1



Insert: 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



Operación: insert (Versión 1)

Filtrado hacia arriba
o Percolate_up

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

    h.tamaño  =  h.tamaño + 1;
    n  =  h.tamaño;

    while ( n / 2 > 0 & h.dato[n/2]  > x ) {
        h.dato[n]  = h.dato[n/2];
        n = n/2;
        }
    h.dato[n] = x;    // ubicación correcta de "x"

}  // end del insert
```

Operación: percolate_up

```
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: insert (Versión 2)

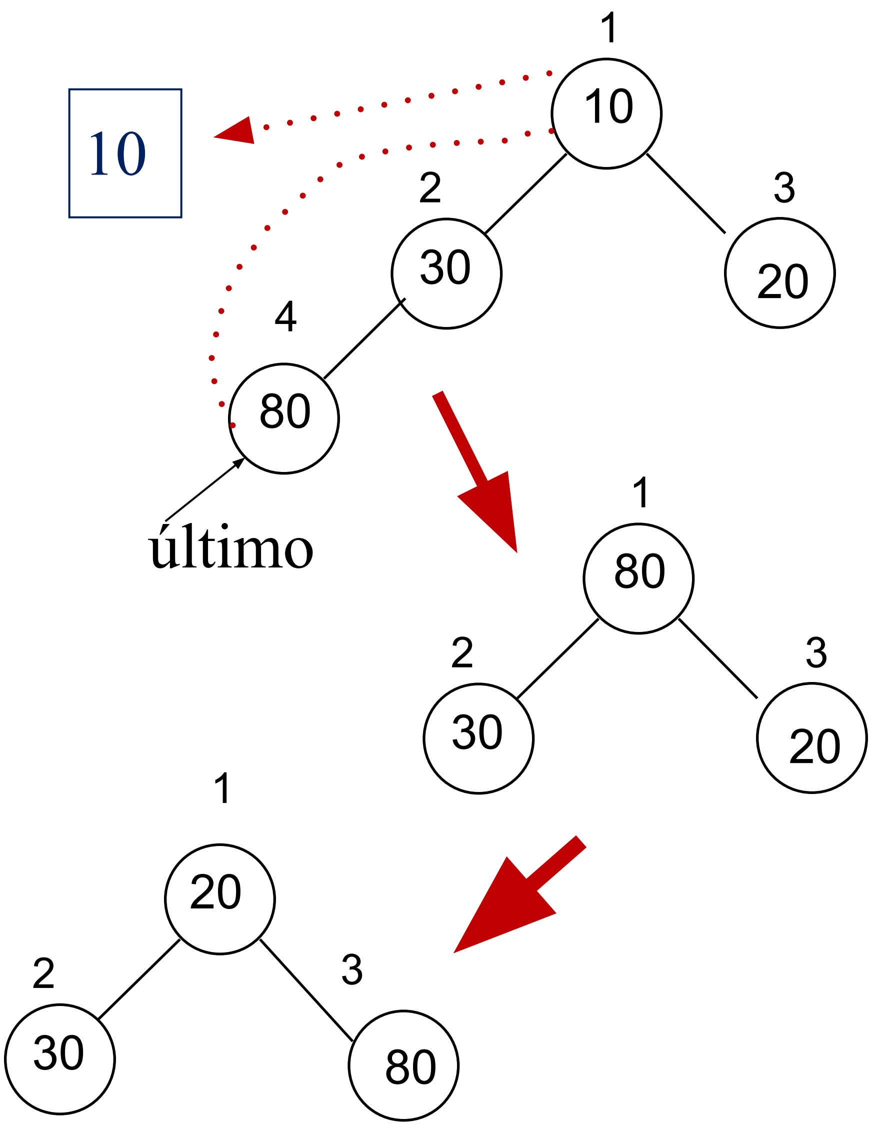
```
insert (Heap h, Comparable x)  {

    h.tamaño =  h.tamaño + 1;
    h.dato[h.tamaño] = x;
    percolate_up ( h , h.tamaño )

}  // end del insert
```

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



DeleteMin: 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.

Operación: delete_min (Versión 1)

```
delete_min ( Heap h, Comparable e)  {
 if (not esVacía(h) ) {
       e := h.dato[1];
       candidato := h.dato[ h.tamaño ];
       h.tamaño :=  h.tamaño  - 1;
       p := 1;
       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   //como existe el hijo derecho comparo a ambos
                      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;
       }
      h.dato[p] := candidato;
 }
}  // end del delete_min
```

Filtrado hacia abajo o Percolate_down

Operación: percolate_down

```
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 }
```

Operación: delete_min (Versión 2)

```
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
```

Algunos errores del ChatGPT

```
  8
 /  \
2   6
/
1
```

Paso 5: Insertar 10

Insertamos 10. Primero se compara con su padre (2). Como 10 es mayor que 2, hacemos un swap. Luego, el nuevo nodo 10 se compara con el padre 8. Como 10 es mayor que 8, se hace otro swap.

Heap hasta ahora:

```
   10
   /  \
  8    6
 / \
2   1
```

Algunos errores del ChatGPT

```
   10
  /  \
 8    6
/ \
2  1
```

Paso 6: Insertar 15

Insertamos 15. Primero se compara con su padre (8). Como 15 es mayor que 8, se hace un swap. Luego, el nuevo nodo 15 se compara con el padre 10. Como 15 es mayor que 10, se hace otro swap.

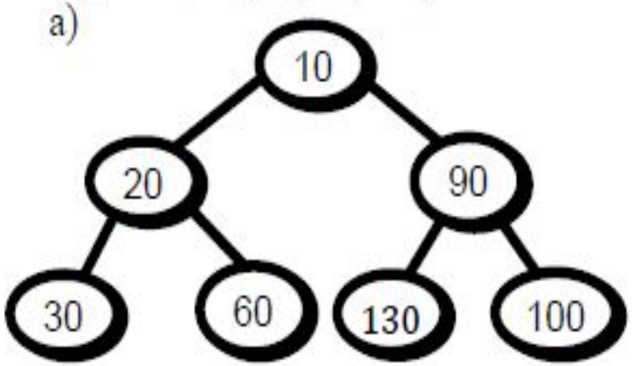
Heap hasta ahora:

```
   15
  /   \
 10    6
/  \    /
8  1 2
```

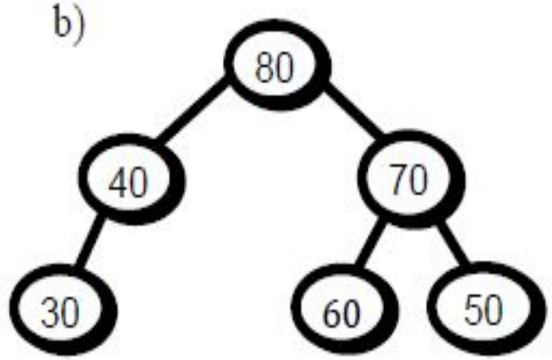
Ejercitación

1.- Indique para cada uno de los siguientes árboles binarios si son un árbol parcialmente ordenado. En caso negativo, explique por qué.

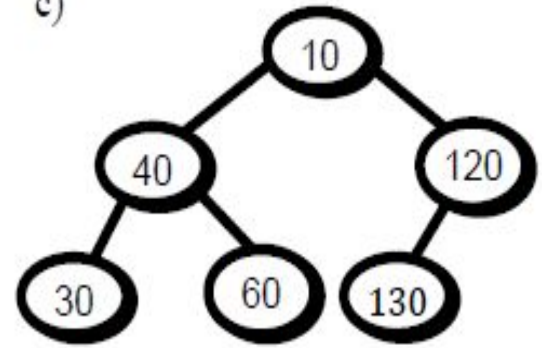
a)



b)

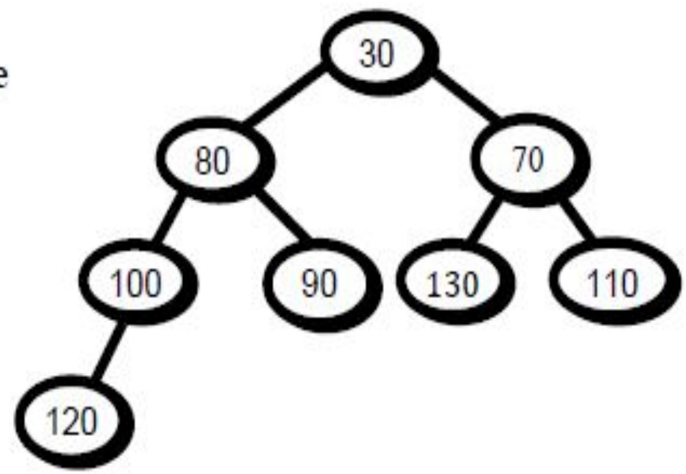


c)



2.- ¿Cuál de las siguientes opciones corresponde al almacenamiento lineal del siguiente árbol parcialmente ordenado o Heap binaria?

a) 30,70, 80 90,100, 110,120, 130
b) 30,80, 70,100, 90,130, 110, 120
c) 30,80, 100,120, 90,70, 130, 110
d) 120, 100, 90, 80, 130, 110, 70, 30



3.- Inserte los valores 60,75 y 10 a la heap anterior. Dibuje la heap resultante después de cada operación.

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)**