Programación 3

Cursada 2022

Prof. Alejandra Schiavoni (ales@info.unlp.edu.ar)

Prof. Pablo Iuliano (piuliano@info.unlp.edu.ar)

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

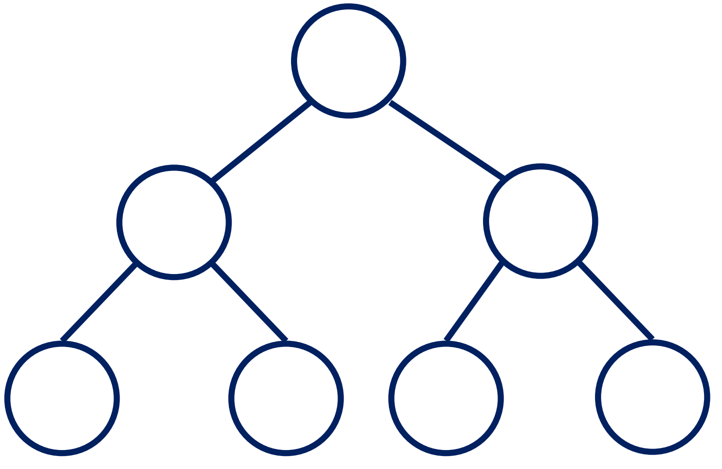
Una heap es un árbol binario completo

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

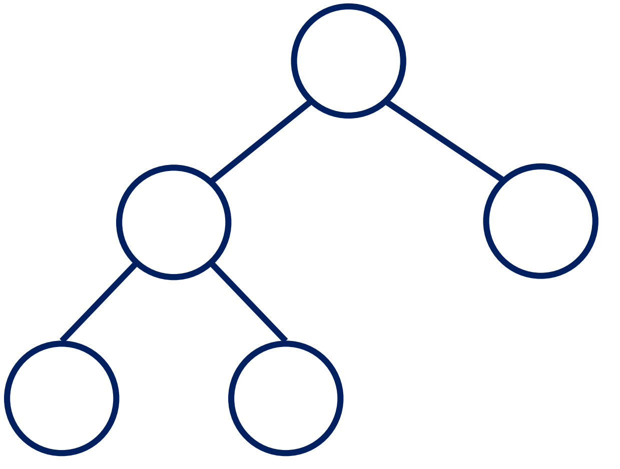
- ✓ Un árbol binario completo de altura *h* es un árbol binario lleno de altura *h-1* y en el nivel *h*, los nodos se completan de izquierda a derecha

Propiedad estructural (cont.)

Árbol binario lleno

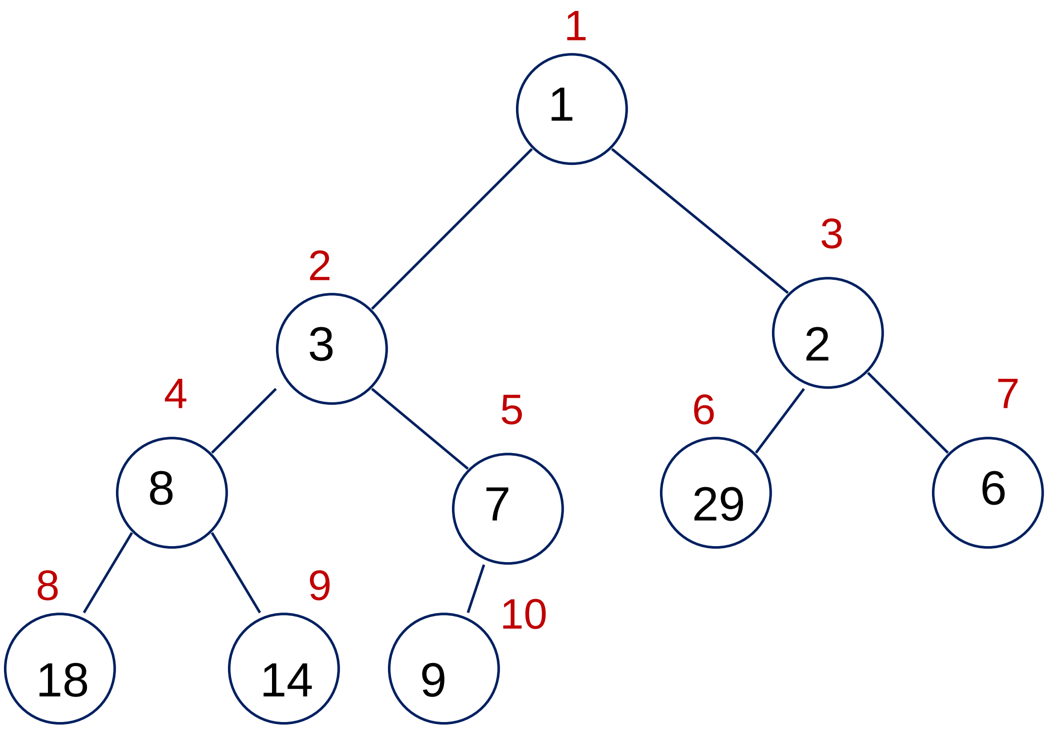


Árbol binario completo



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:*

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

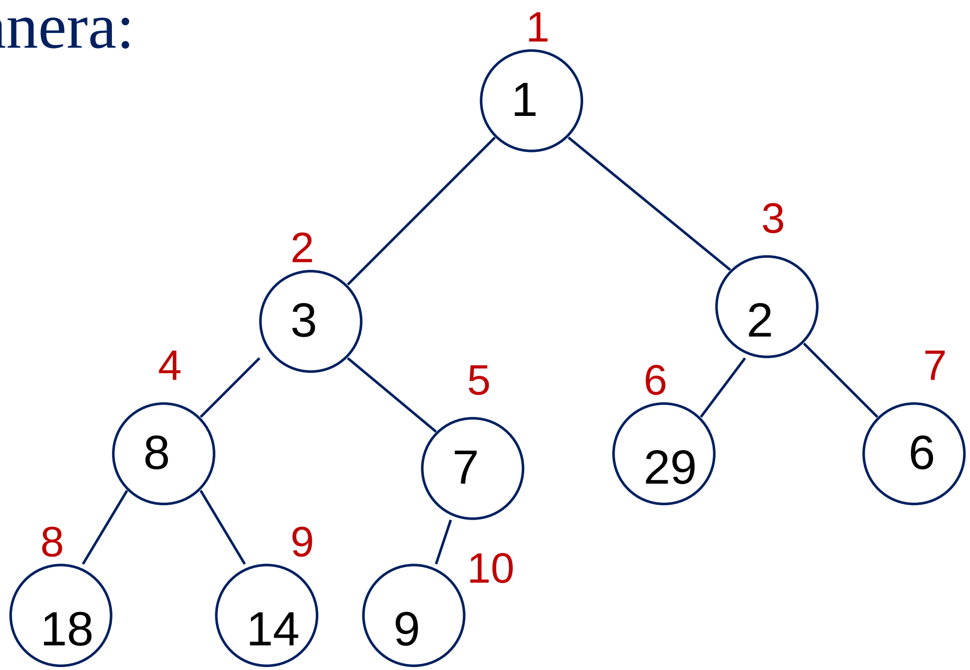
La altura *h* del árbol es de **O(log n)**

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:

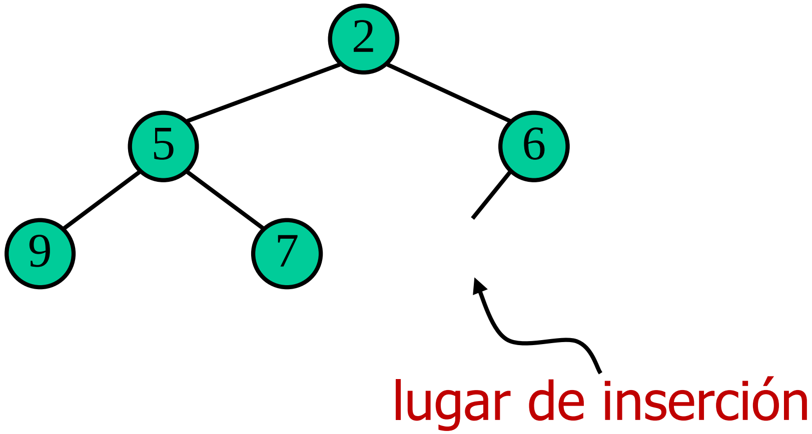
- *Un arreglo que contiene los datos*
- *Un valor que me indica el número de elementos almacenados*

Ventaja:

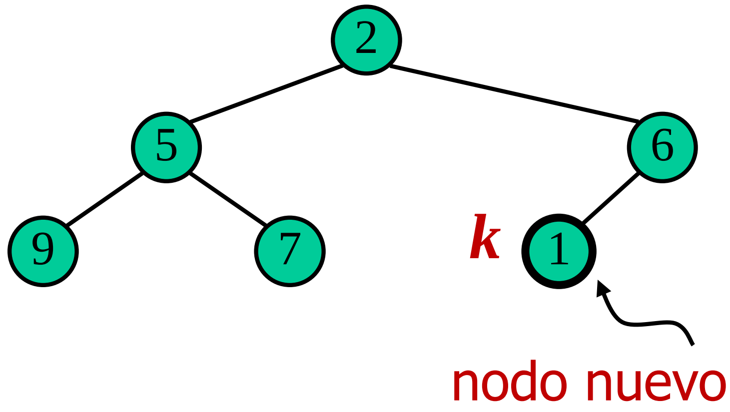
- ✓ No se necesita usar punteros
- ✓ Fácil implementación de las operaciones

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

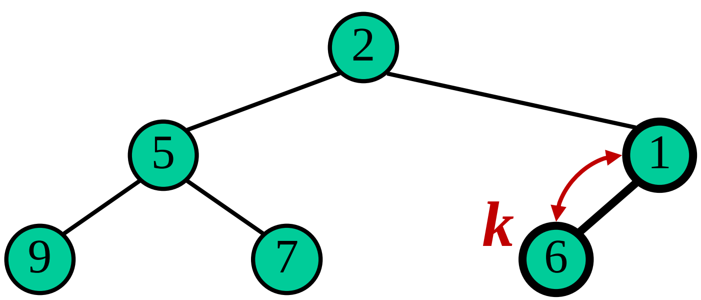


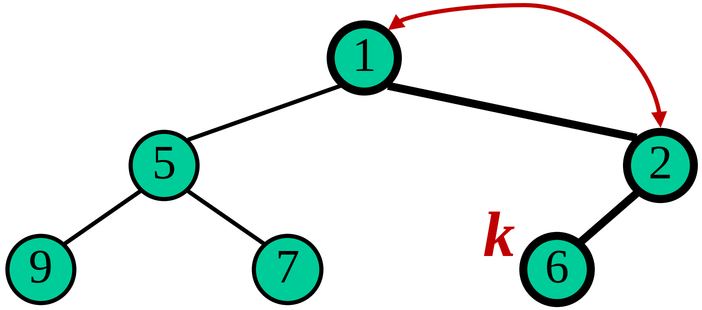
Inserto el 1



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)

```
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
```

Filtrado hacia arriba o Percolate_up

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)

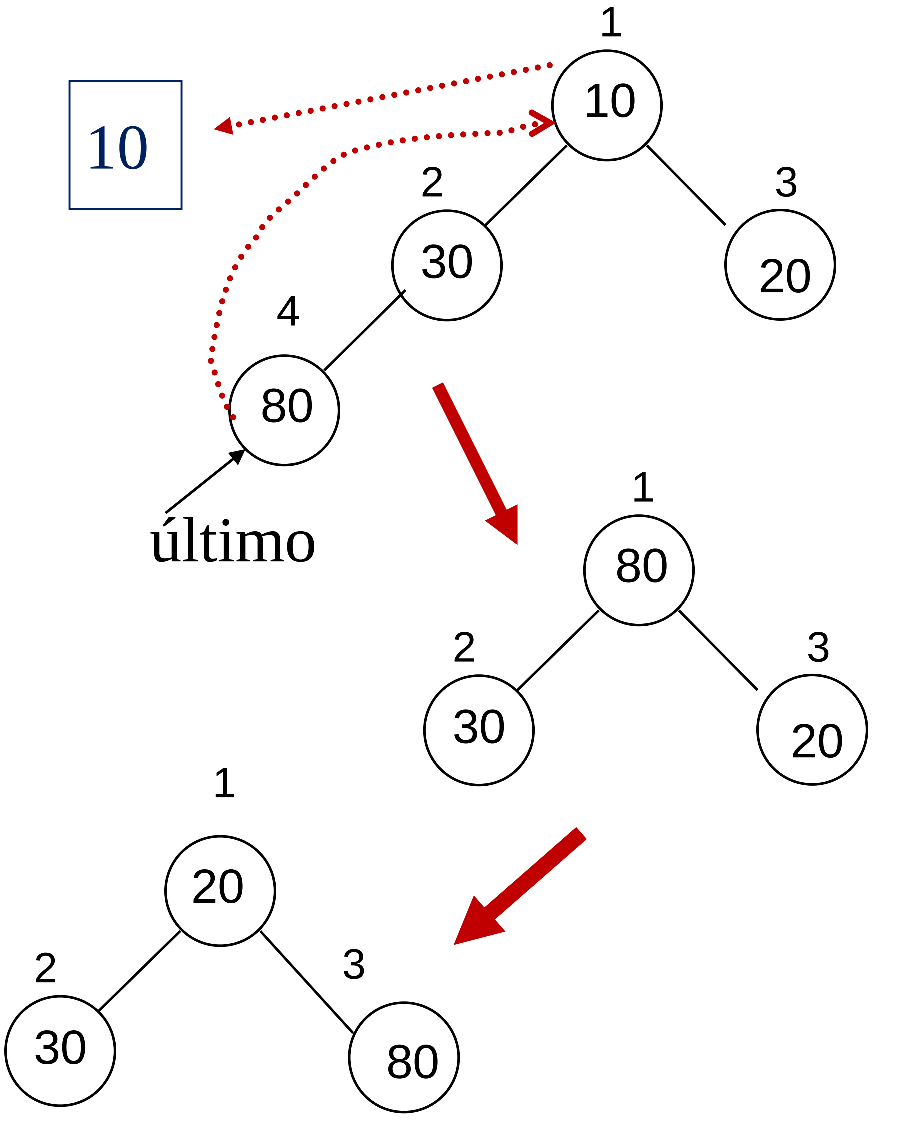
```
insert (Heap h, Comparable x)  {

        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 }
```

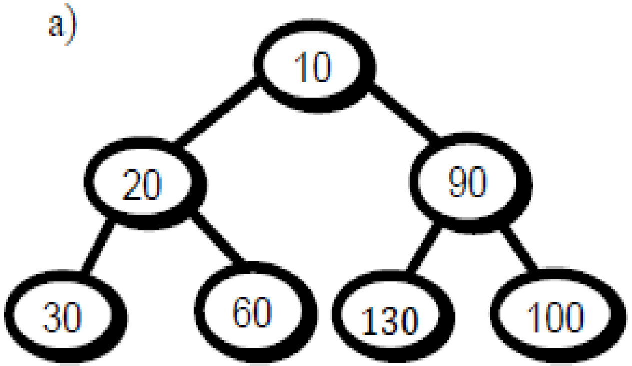
Operación: delete_min (Versión 2)

```
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
```

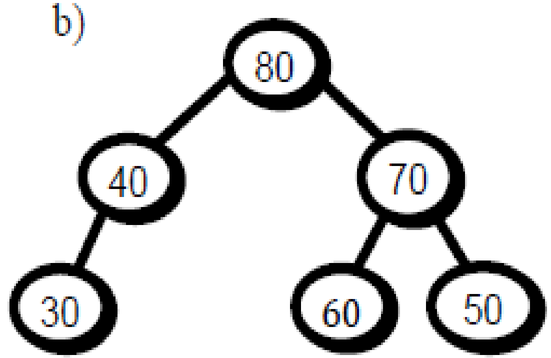
Ejercitación

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

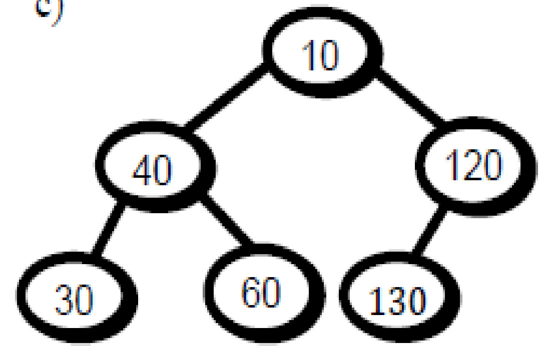
a)



b)

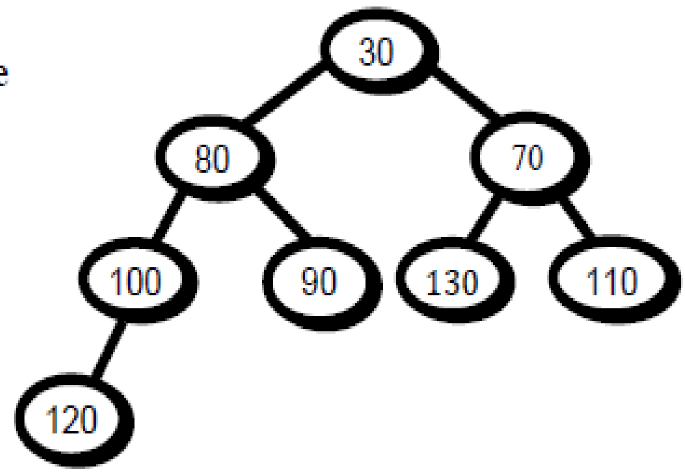


c)



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)

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

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

✓ Se pueden insertar los elementos de a uno

⇒ se realizan (n log n) operaciones en total

✓ Se puede usar un algoritmo de orden lineal, es decir, proporcional a los n elementos ⇒ **BuildHeap**

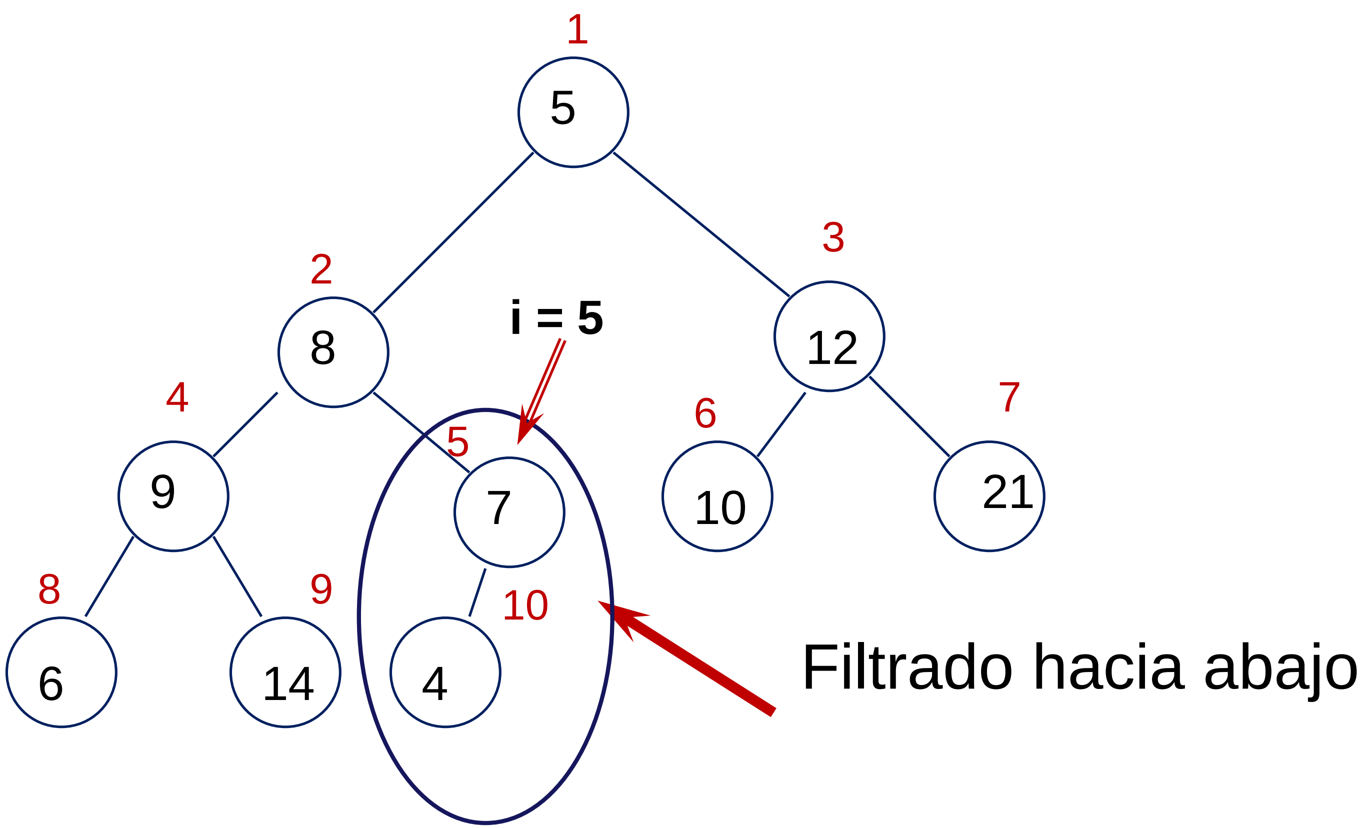
- Insertar los elementos desordenados en un árbol binario completo
- Filtrar hacia abajo cada uno de elementos

Algoritmo BuildHeap

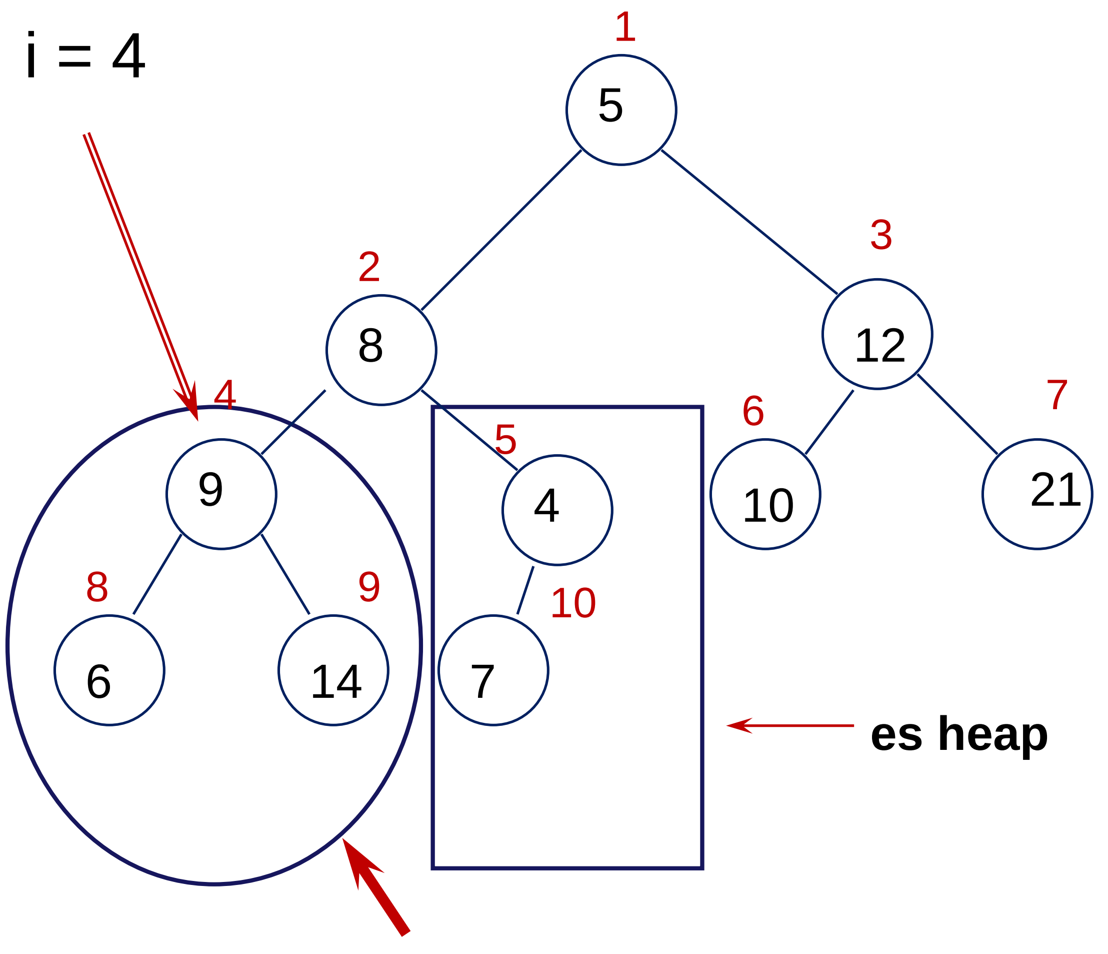
- Para filtrar:
  - se elige el menor de los hijos
  - se compara el menor de los hijos con el padre
- Se empieza filtrando desde el elemento que está en la posición (tamaño/2):
  - se filtran los nodos que tienen hijos
  - el resto de los nodos son hojas

BuildHeap

5	8	12	9	7	10	21	6	14	4
1	2	3	4	5	6	7	8	9	10

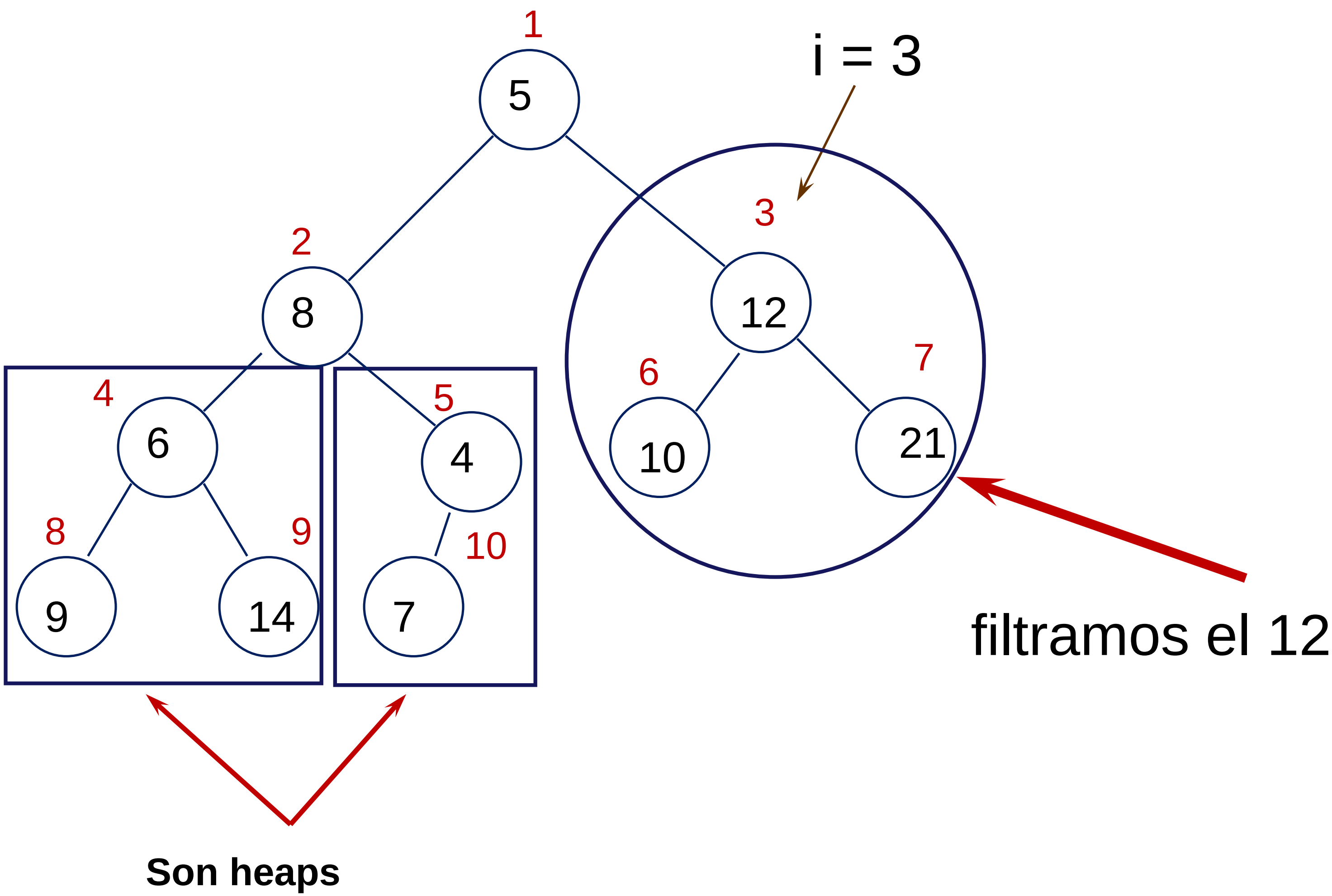


BuildHeap

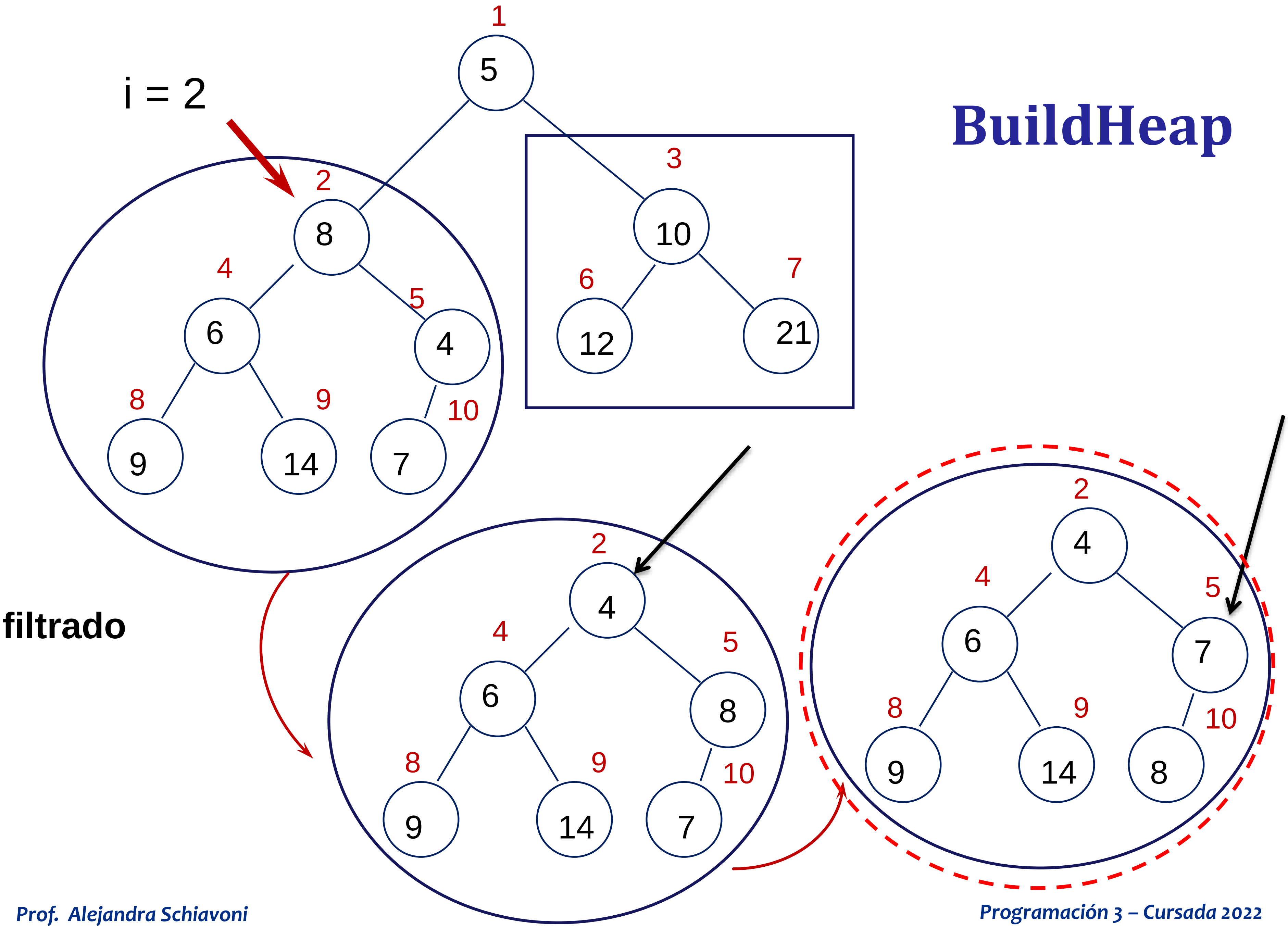


filtramos el 9

BuildHeap

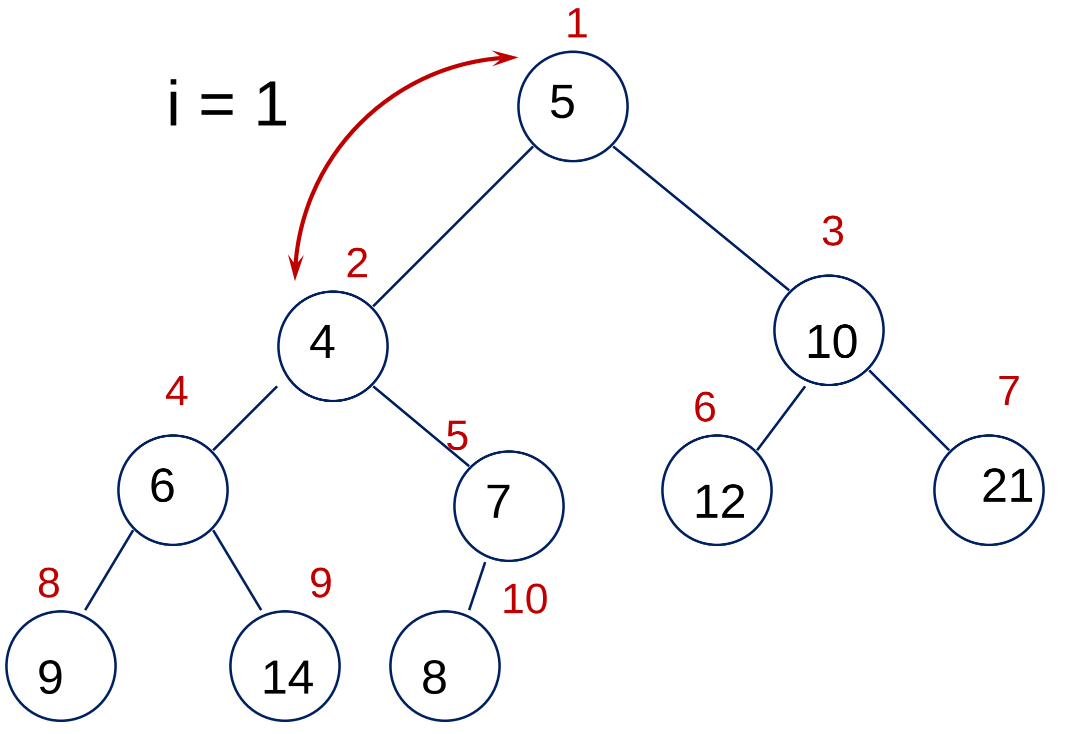


BuildHeap

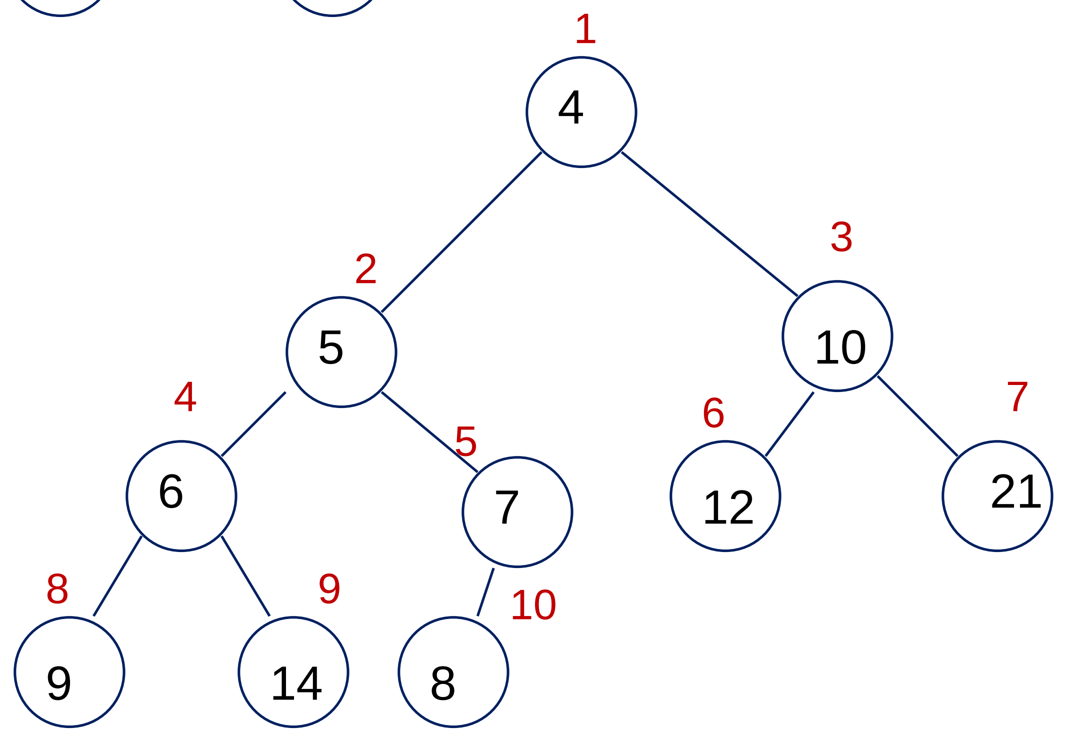


i = 2
1
5
2
8
3
10
4
6
5
4
6
12
7
21
8
9
9
14
10
7
filtrado
2
4
4
6
5
8
8
9
9
14
10
7
2
4
4
6
5
7
8
9
9
14
10
8

BuildHeap



i = 1
1
5
2
4
3
10
4
6
5
7
6
12
7
21
8
9
9
14
10
8



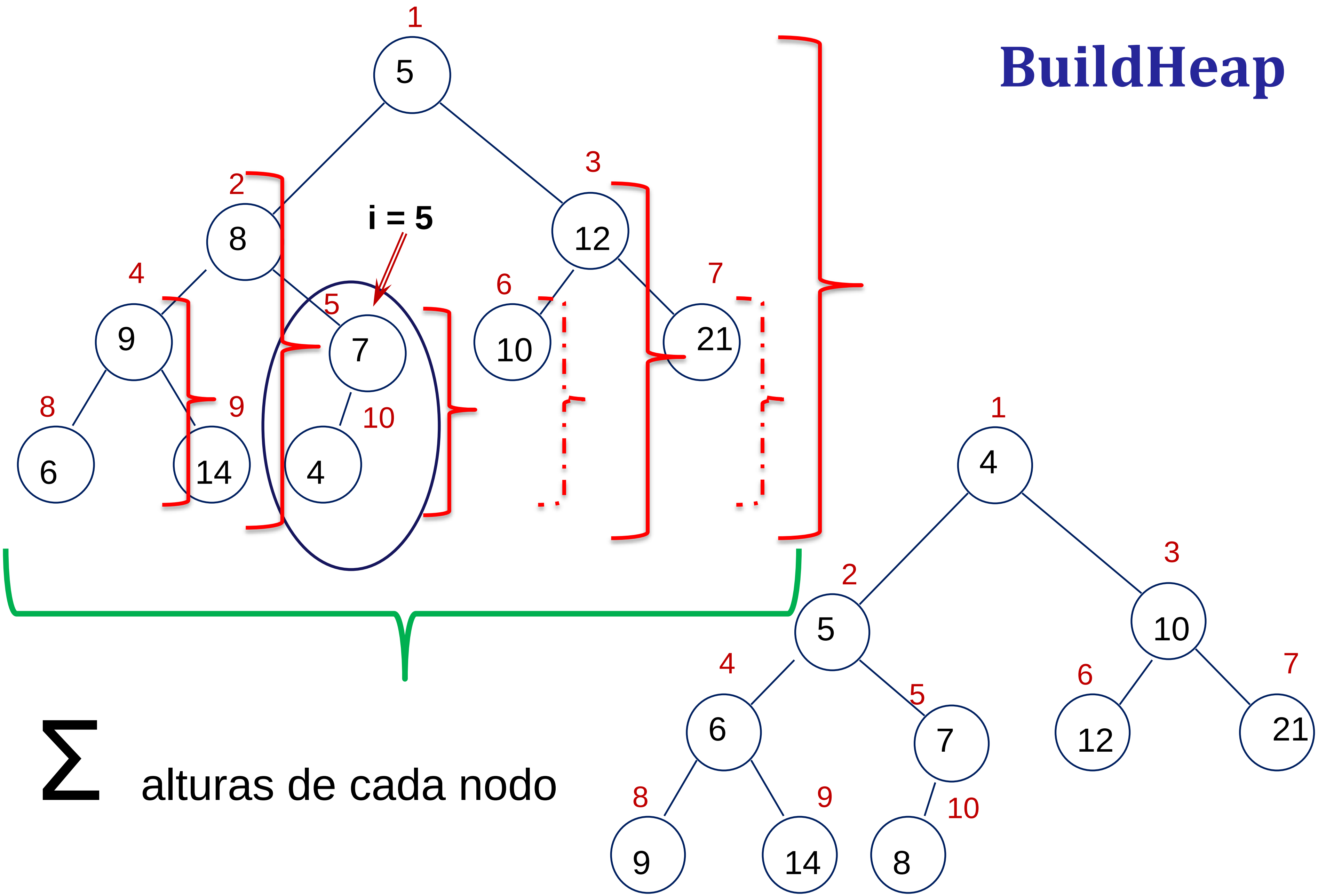
1
4
2
5
3
10
4
6
5
7
6
12
7
21
8
9
9
14
10
8

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

BuildHeap



$\sum$ alturas de cada nodo

Cantidad de operaciones requeridas (cont.)

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 $h – i$

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

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

Cantidad de operaciones requeridas (cont.)

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

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

Restando las dos igualdades (B) – (A)

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

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

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

$$S + 1 = -h + (2^{h+1} - 1)$$

$$S = (2^{h+1} - 1) - (h + 1)$$

Cantidad de operaciones requeridas (cont.)

- 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**

Ordenación de vectores usando Heap

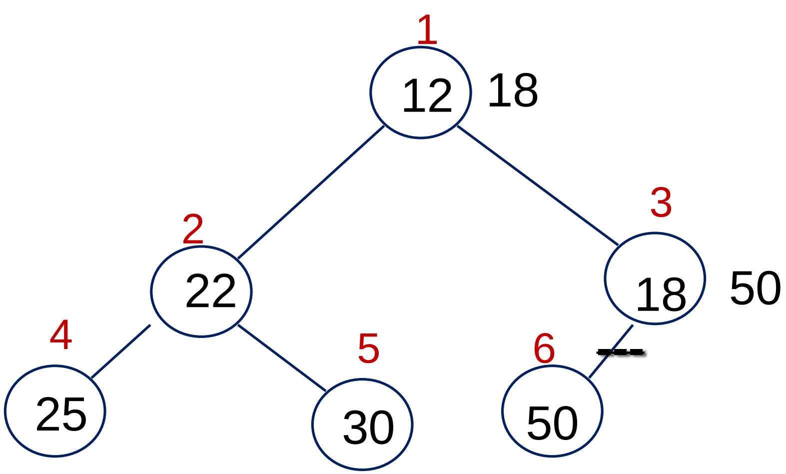
*Ejemplo: Construir una MinHeap, realizar **6** DeleteMin operaciones e ir guardando los elementos extraídos en otro arreglo.*

50	30	18	25	22	12

entrada

12					
1	2	3	4	5	6

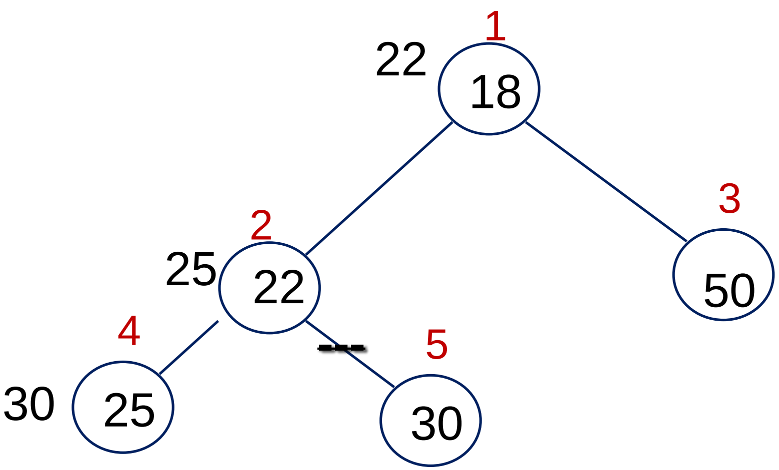
salida



12	22	18	25	30	50
1	2	3	4	5	6

18	22	50	25	30	~~50~~
1	2	3	4	5	6

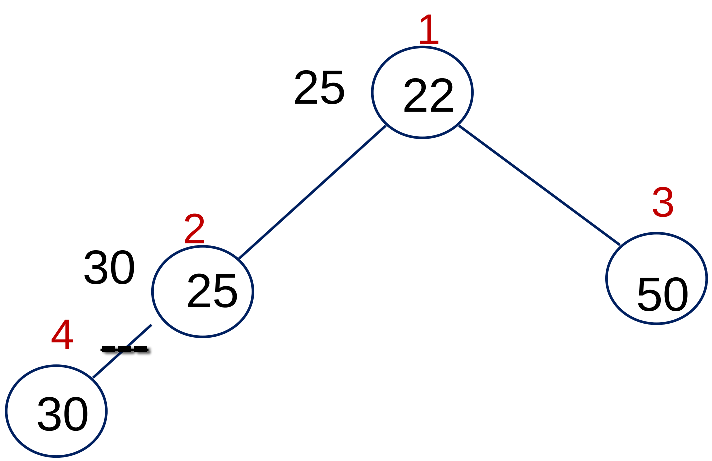
Ordenación de vectores usando Heap



salida

12	18	22			
1	2	3	4	5	6

18	22	50	25	30	~~50~~
1	2	3	4	5	6



22	25	50	30	~~30~~	~~50~~
1	2	3	4	5	6

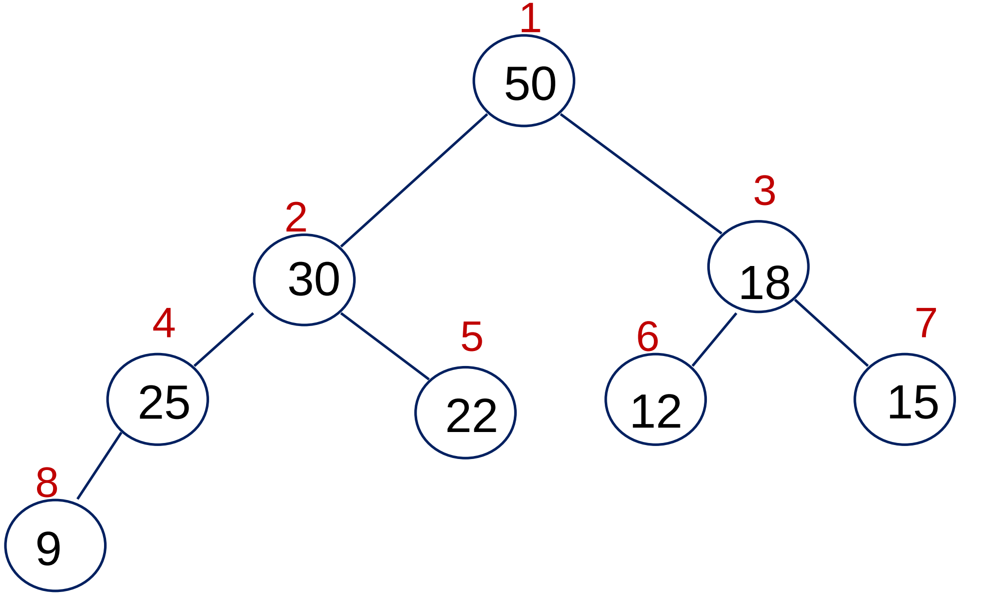
25	30	50	~~30~~	~~30~~	~~50~~
1	2	3	4	5	6

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:

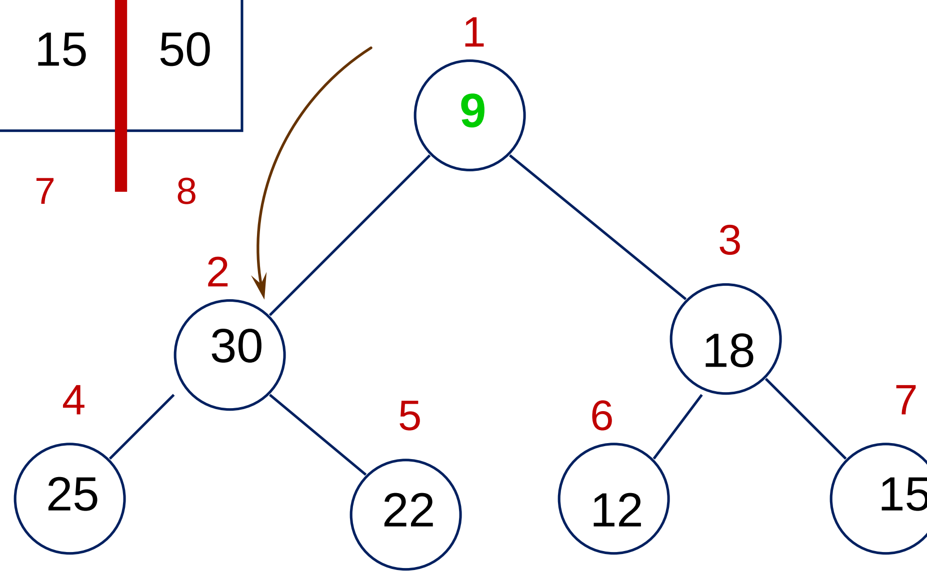


HeapSort (cont.)

1	2	3	4	5	6	7	8
50	30	18	25	22	12	15	9

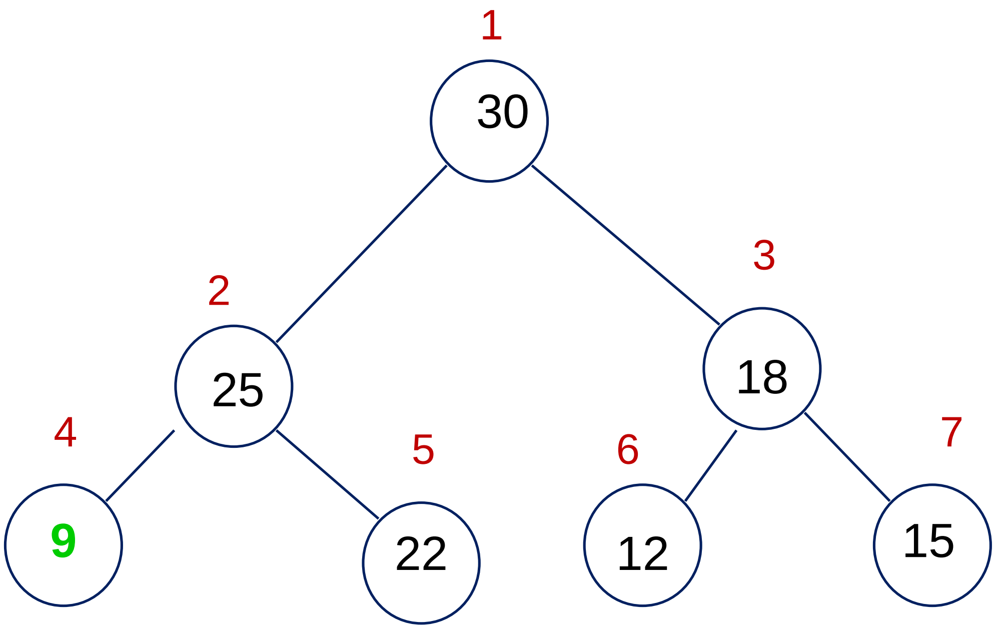
- Intercambio el primero con el último y
- decremento el tamaño

1	2	3	4	5	6	7	8
9	30	18	25	22	12	15	50



Filtrar el **9**

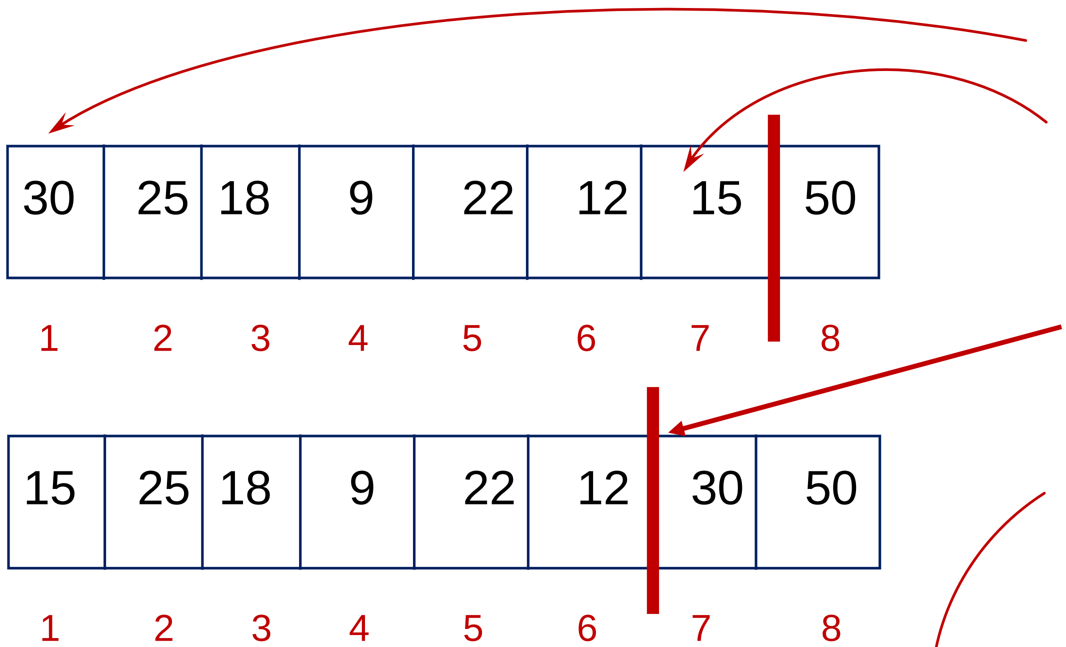
HeapSort (cont.)



Después de filtrar el **9** hacia abajo

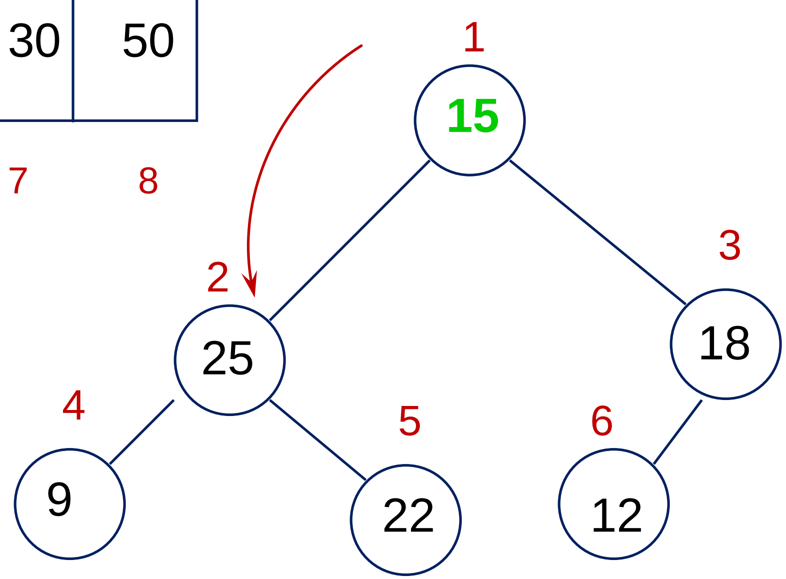
30	25	18	9	22	12	15	50
1	2	3	4	5	6	7	8

HeapSort (cont.)

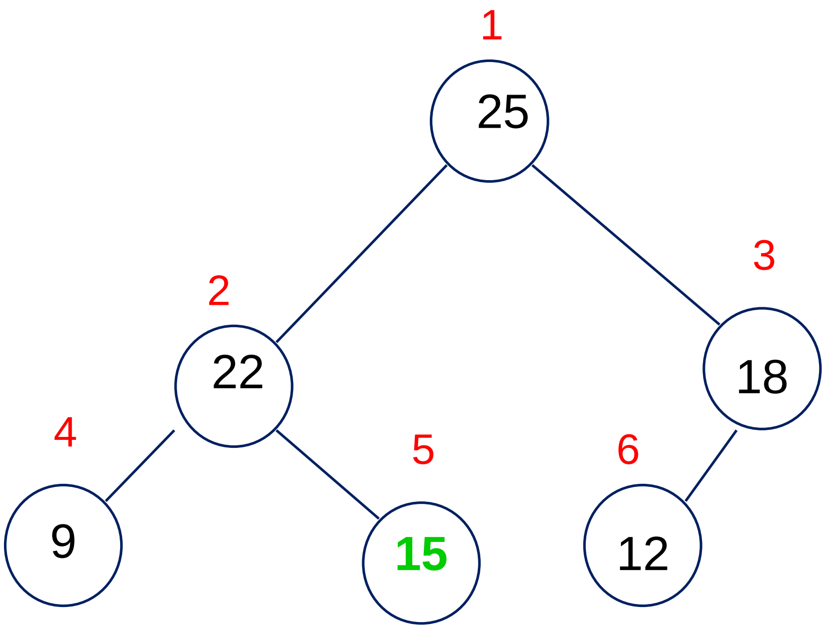


- Intercambio el primero con el último y
- decremento el tamaño

Filtrar el **15**



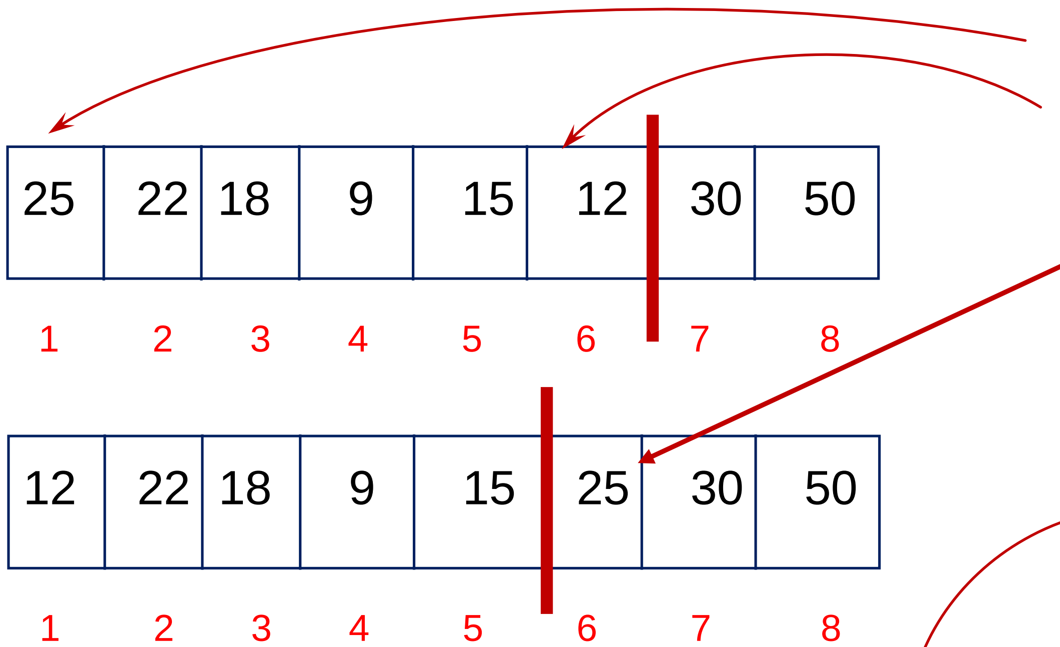
HeapSort (cont.)



Después de filtrar el **15** hacia abajo

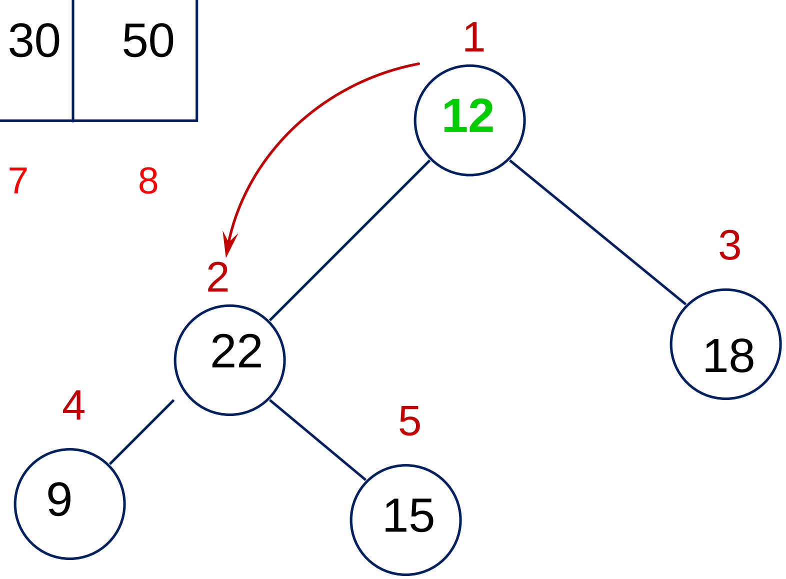
25	22	18	9	15	12	30	50
1	2	3	4	5	6	7	8

HeapSort (cont.)



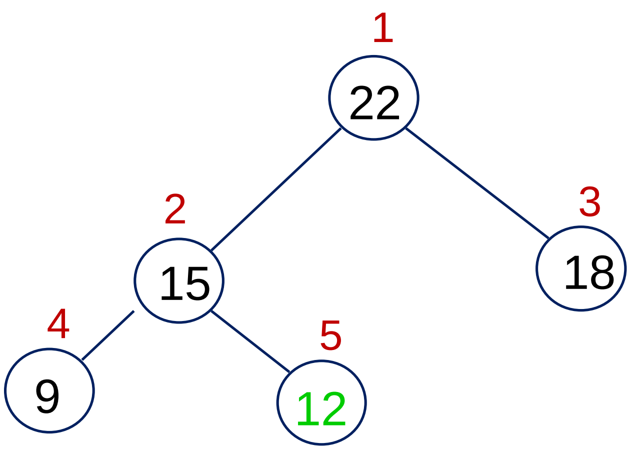
- Intercambio el primero con el último y
- decremento el tamaño

Filtrar el **12**



HeapSort (cont.)

Después de filtrar el **12** hacia abajo

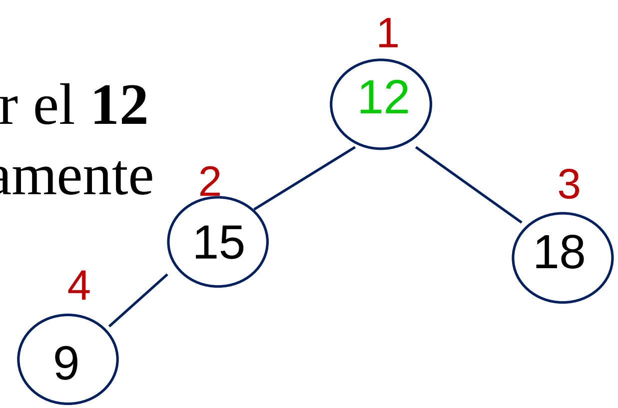


22	15	18	9	12	25	30	50
1	2	3	4	5	6	7	8

22	15	18	9	12	25	30	50
1	2	3	4	5	6	7	8

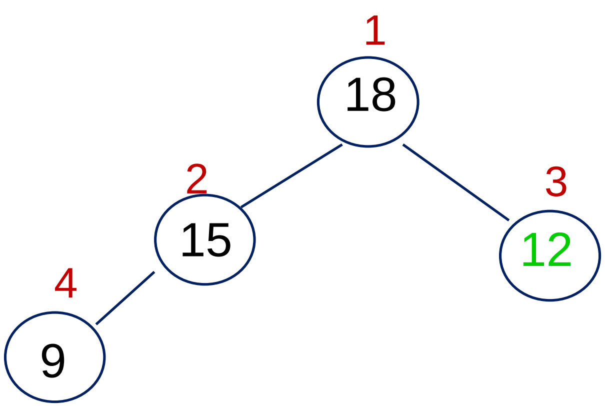
12	15	18	9	22	25	30	50
1	2	3	4	5	6	7	8

Filtrar el **12** nuevamente



HeapSort (cont.)

Después de filtrar el **12** hacia abajo

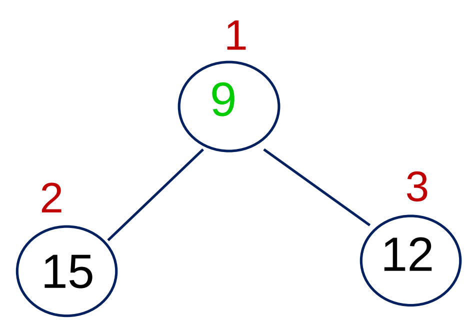


1	2	3	4	5	6	7	8
18	15	12	9	22	25	30	50

1	2	3	4	5	6	7	8
18	12	15	9	22	25	30	50

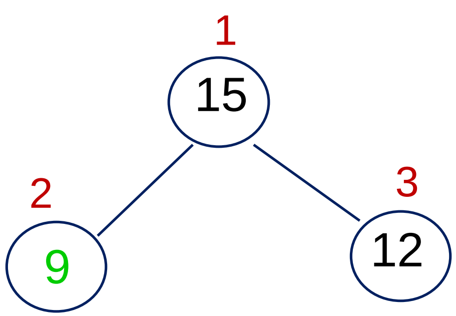
1	2	3	4	5	6	7	8
9	15	12	18	22	25	30	50

Filtrar el **9**



HeapSort (cont.)

Después de filtrar el **9** hacia abajo

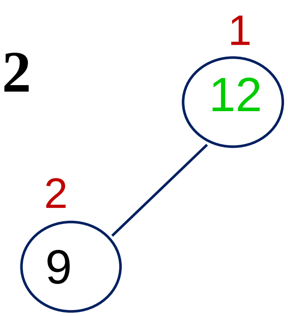


15	9	12	18	22	25	30	50
1	2	3	4	5	6	7	8

15	9	12	18	22	25	30	50
1	2	3	4	5	6	7	8

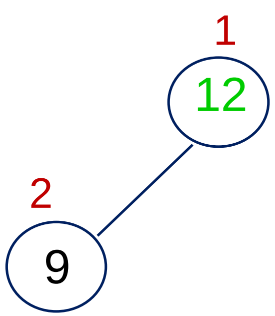
12	9	15	18	22	25	30	50
1	2	3	4	5	6	7	8

Filtrar el **12**



HeapSort (cont.)

Después de filtrar el **12** hacia abajo

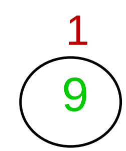


12	9	15	18	22	25	30	50
1	2	3	4	5	6	7	8

12	9	15	18	12	25	30	50
1	2	3	4	5	6	7	8

9	12	15	18	22	25	30	50
1	2	3	4	5	6	7	8

Filtrar el **9**



HeapSort (cont.)

9	12	15	18	22	25	30	50
1	2	3	4	5	6	7	8

Datos almacenados internamente

Heap conceptual