Algoritmos
y
Estructuras de Datos

Cursada 2011

Prof. Catalina Mostaccio
Prof. Alejandra Schiavoni

Facultad de Informática - UNLP

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
  - Tiempo de Ejecución
- 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)
  - DeleteMin tiene O(1)
- Lista no ordenada
  - Insert tiene O(1)
  - DeleteMin tiene O(N)
- Árbol Binario de Búsqueda
  - Insert y DeleteMin tienen en promedio O(log N)

Heap Binaria

- Es una implementación de colas de prioridad que no usa punteros y permite implementar ambas operaciones con un tiempo de O(log N) en el peor caso

- Cumple con dos propiedades:
  - ✓ Propiedad estructural
  - ✓ Propiedad de orden

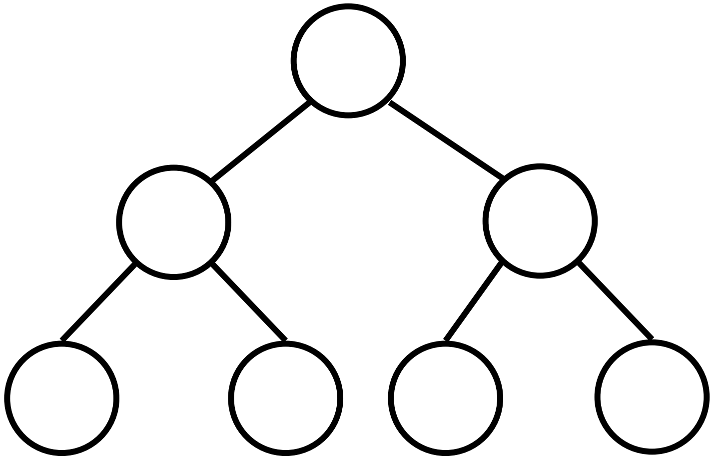
Propiedad estructural

Una heap es un árbol binario completo

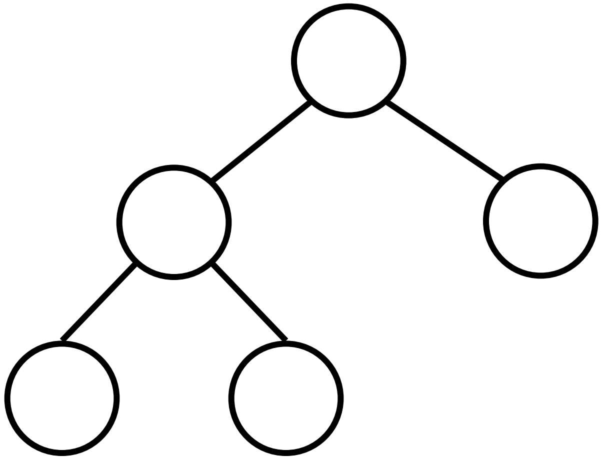
- ✓ En un árbol binario lleno de altura *h*, los nodos internos tienen exactamente 2 hijos y las hojas tienen la misma profundidad
- ✓ Un árbol binario completo de altura *h* es un árbol binario lleno de altura *h-1* y en el nivel *h*, los nodos se completan de izquierda a derecha

Propiedad estructural (cont.)

Árbol binario lleno

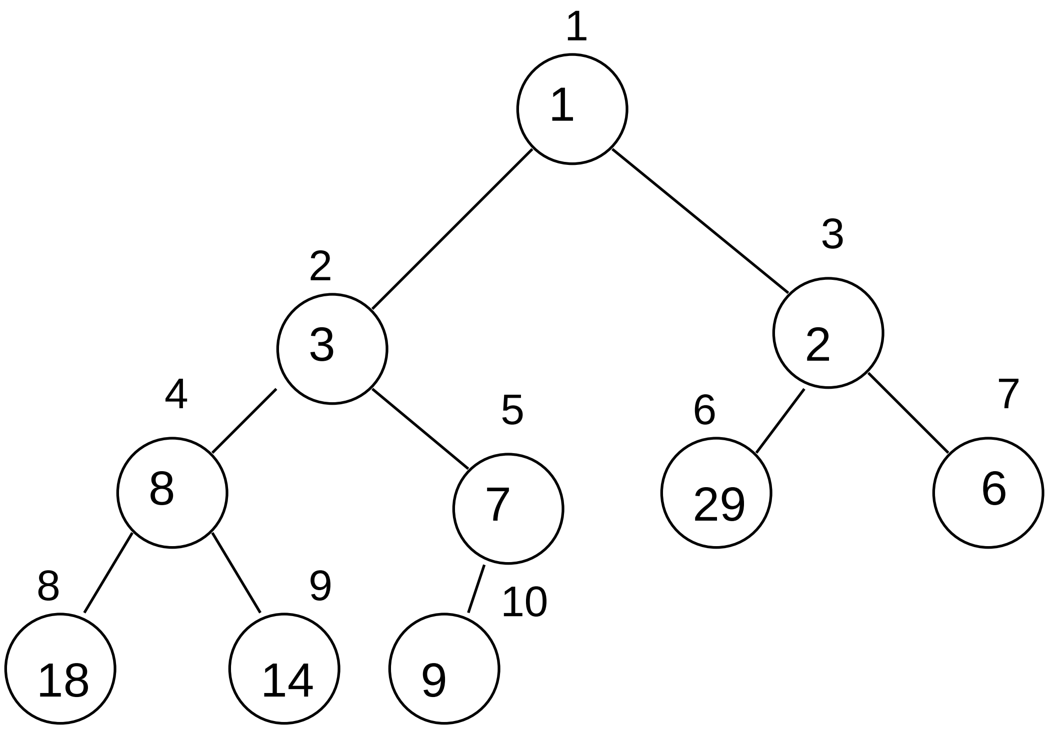


Árbol binario completo



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:*

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

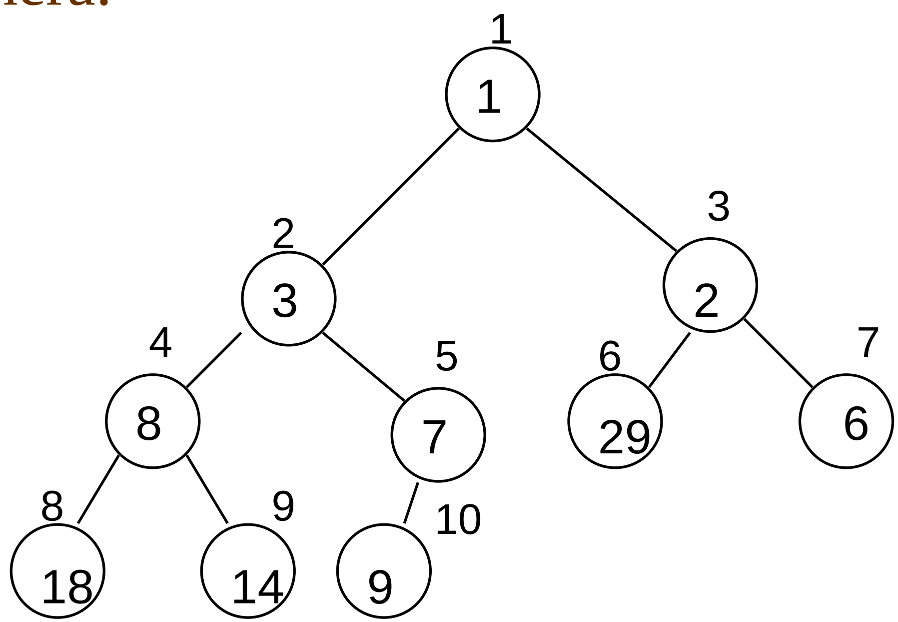
La altura *h* del árbol es de **O(log n)**

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:

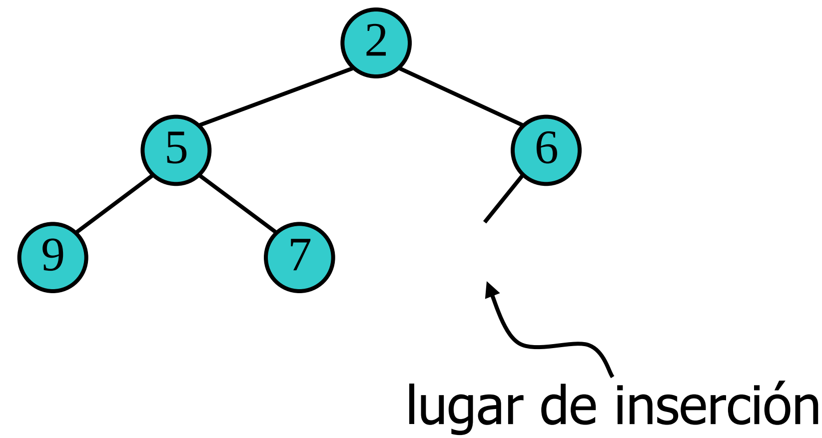
- *Un arreglo que contiene los datos*
- *Un valor que me indica el número de elementos almacenados*

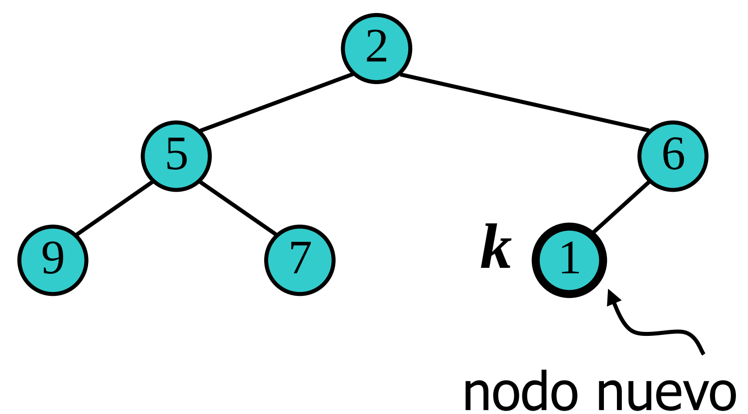
Ventaja:

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

Operación: Insert

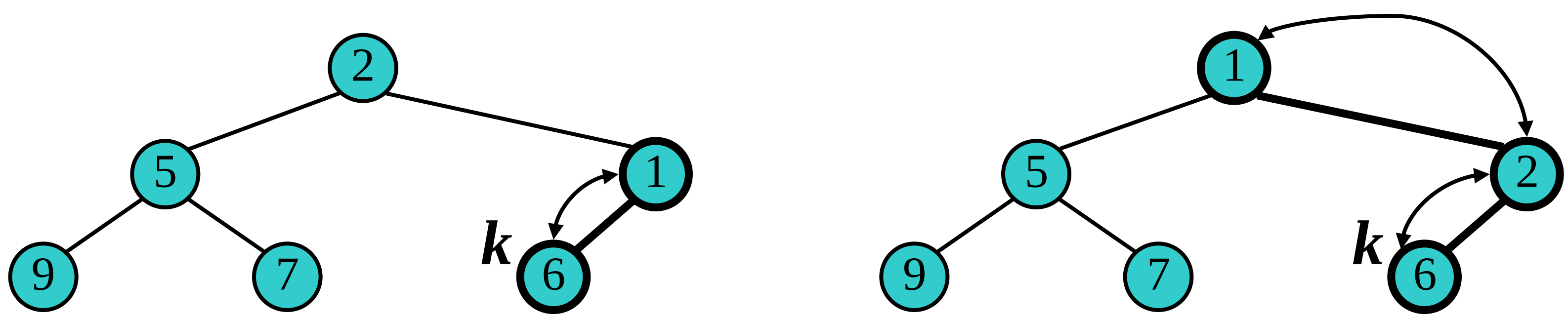
- El dato se inserta como último ítem en la heap
  - La propiedad de la heap puede ser violada
- Se debe hacer un filtrado hacia arriba para restaurar la propiedad de orden





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 tiempo de ejecución de *O(log n)*



Operación: insert <u>Version 1</u>

```
insert ( var H : heap;  x : elemtype)  {

        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 ( var H : heap;  i : integer)  {

        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
```

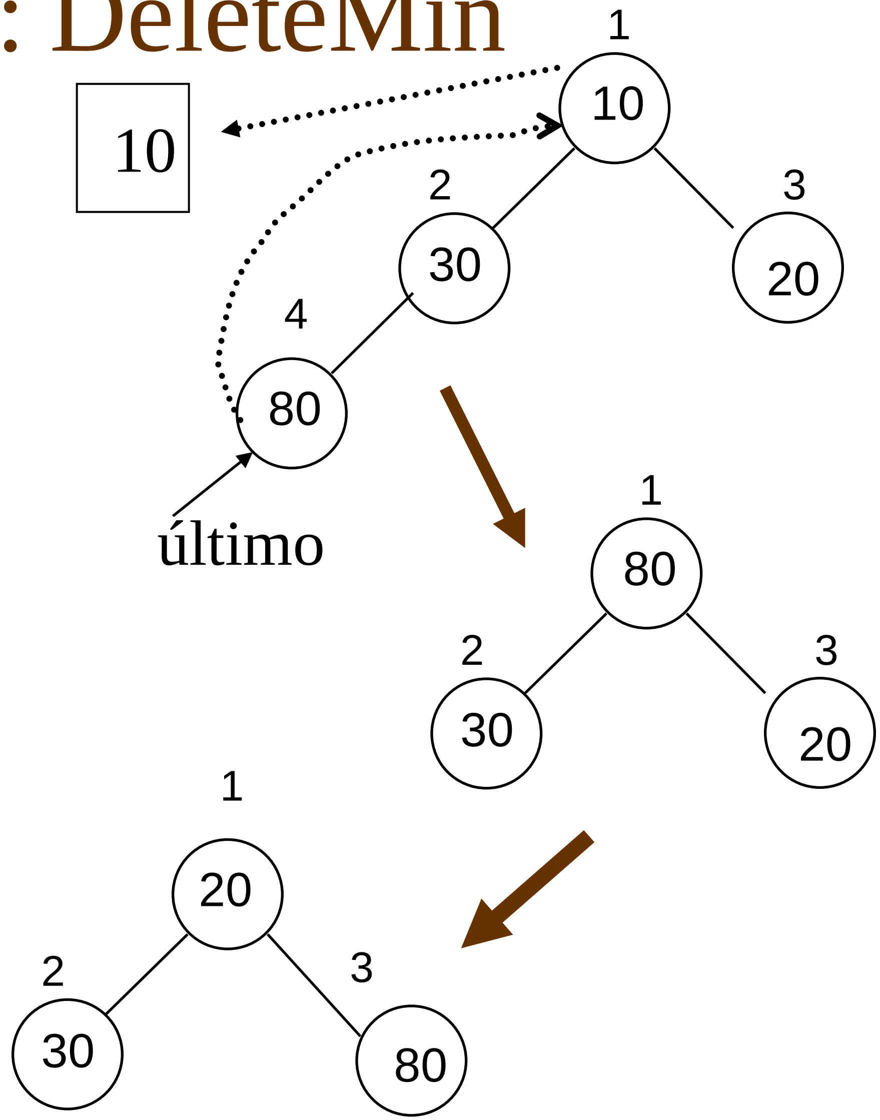
Operación: insert Version 2

```
insert ( var H : heap;  x : elemtype)  {

        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 tiempo de ejecución de *O(log n)*

Operación: delete_min <u>Version 1</u>

Filtrado hacia abajo o Percolate_down

```
Delete_min ( var H : heap; var e : elemtype)  {
 if (not is_empty (H) ) {
          e := H.dato[1]
          candidato := H.dato[ H.tamaño ]
          H.tamaño :=  H.tamaño  - 1;
          p := 1;
          stop_perc := false;
          while (  2* p <= H.size )  and ( not stop_perc) {
               h_min := 2 * p;   // buscar el hijo con clave menor
               if h_min <> H.size   then  //como existe el hijo derecho comparo a ambos
                      if ( H.dato[h_min +1] < H.dato[h_min] ) // then
                                                 h_min := h_min + 1
               if  candidato  > H.dato [h_min] { //  then percolate_down
                                                 H.dato [p] := H.dato[ h_min ];
                                                 p := h_min;
                                                 }
                                               else    stop_perc := true;
          }
     H.dato[ p ] := candidato;
 }
}  // end del delete_min
```

Operación: percolate_down

```
percolate_down ( var H : heap; var p : integer) {

    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] ) then
                                                    h_min := h_min + 1
        if  candidato  > H.dato [h_min] { // then
                                                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 Version 2

```
Delete_min ( var H : heap; var e : elemtype) {

  if (   H.tamaño > 0  ) then   {     //  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
```

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)**

Operación: BuildHeap

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

- Se pueden insertar los elementos de a uno
  - El tiempo de ejecución es de O(n logn)
- Usar un algoritmo de orden lineal:
  - Insertar los elementos desordenados en un árbol binario completo
  - Filtrar hacia abajo cada uno de elementos

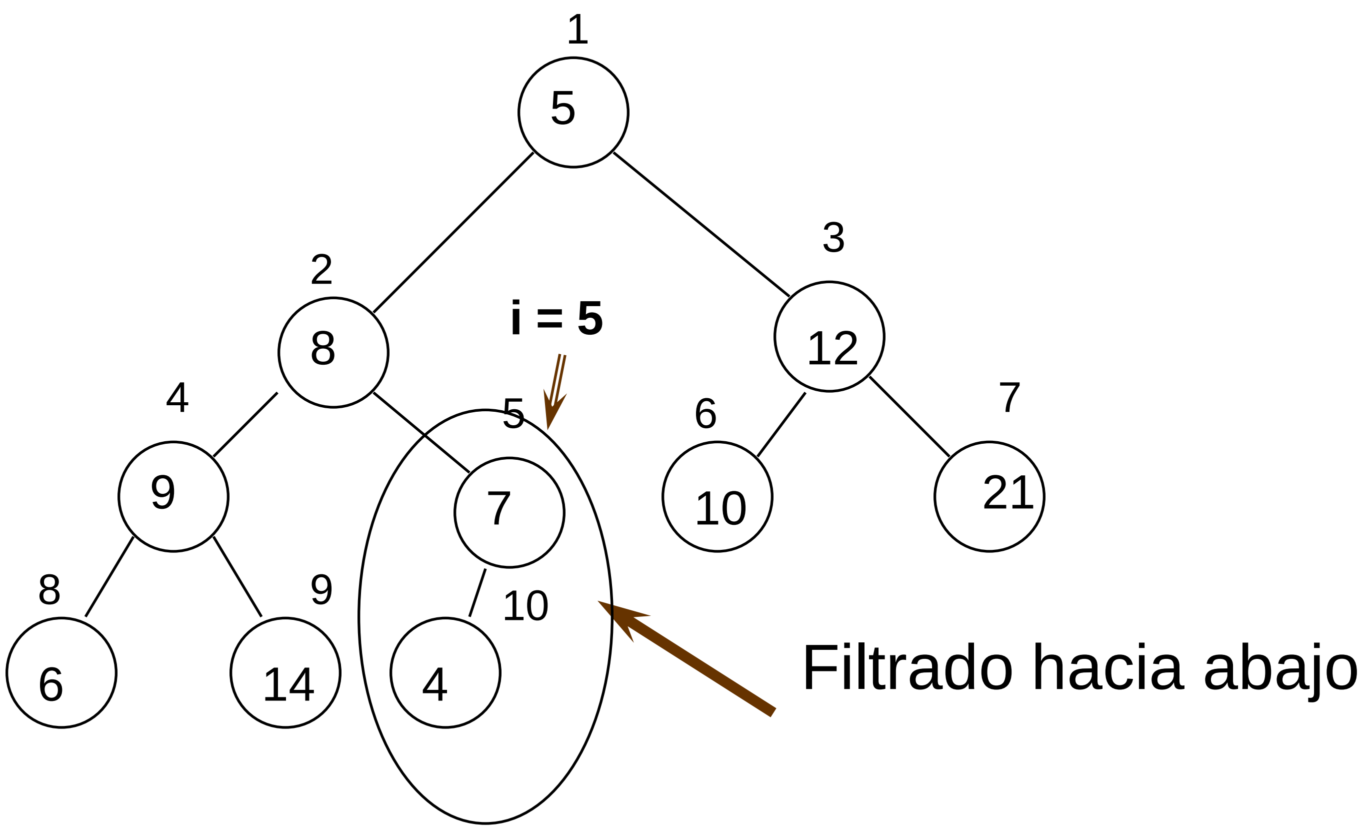
BuildHeap

- 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

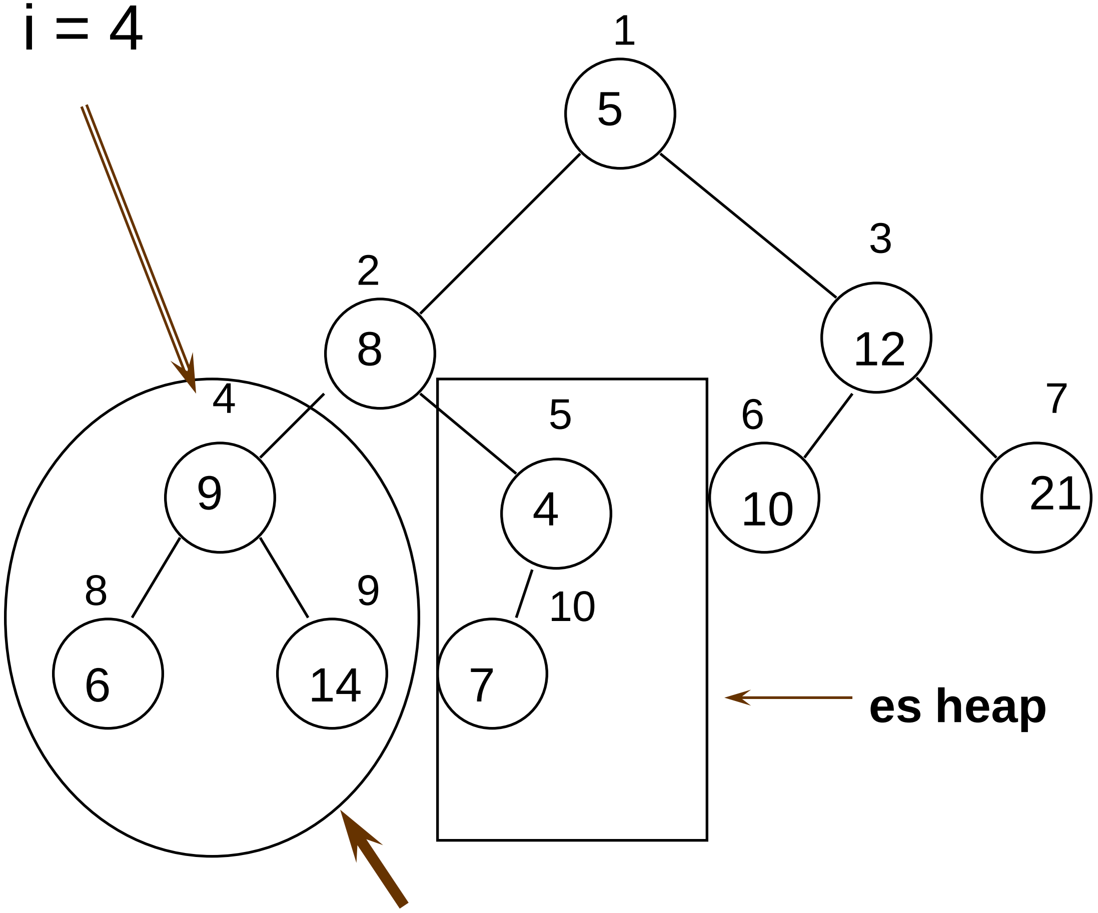
BuildHeap

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



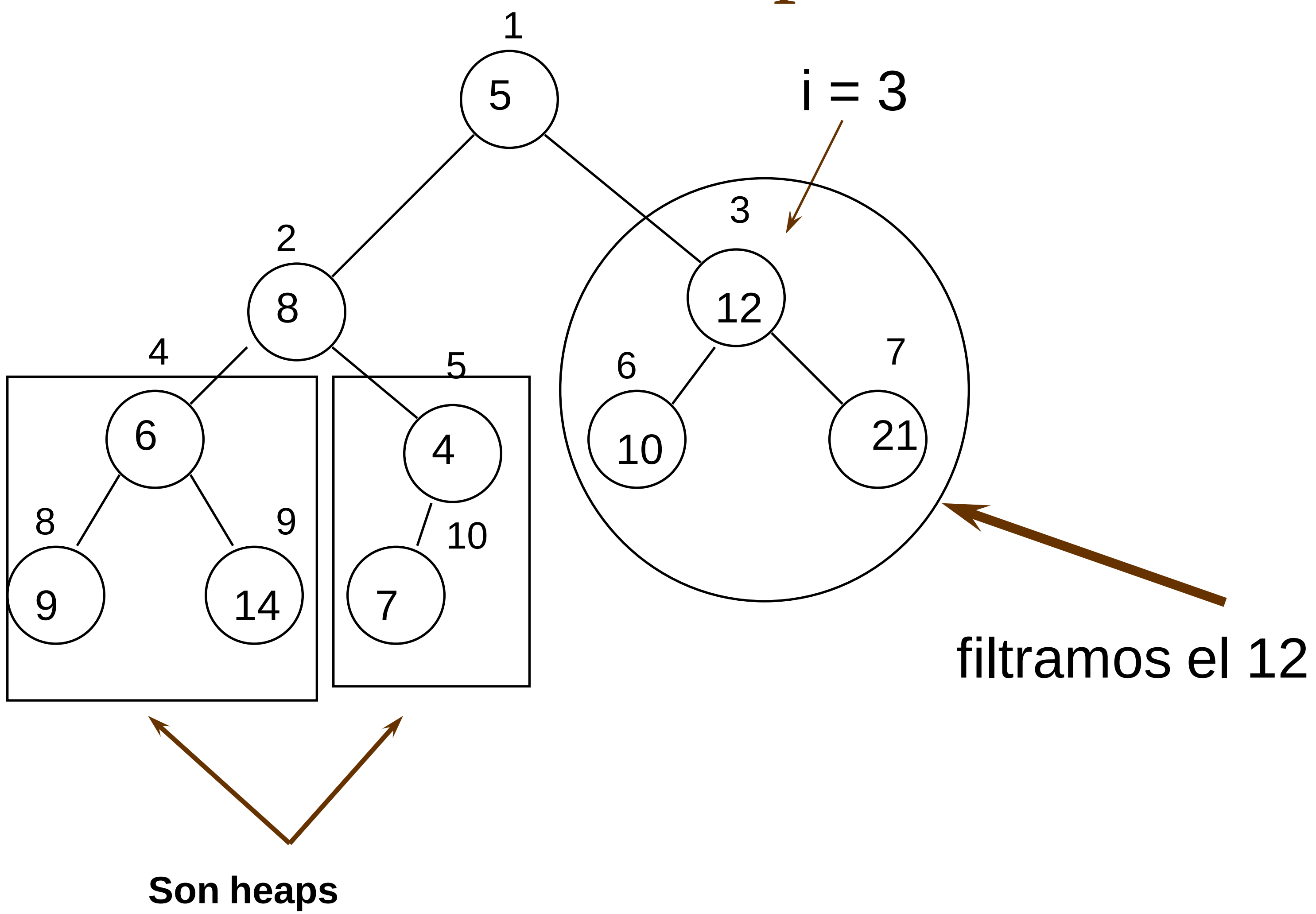
BuilHeap

i = 4



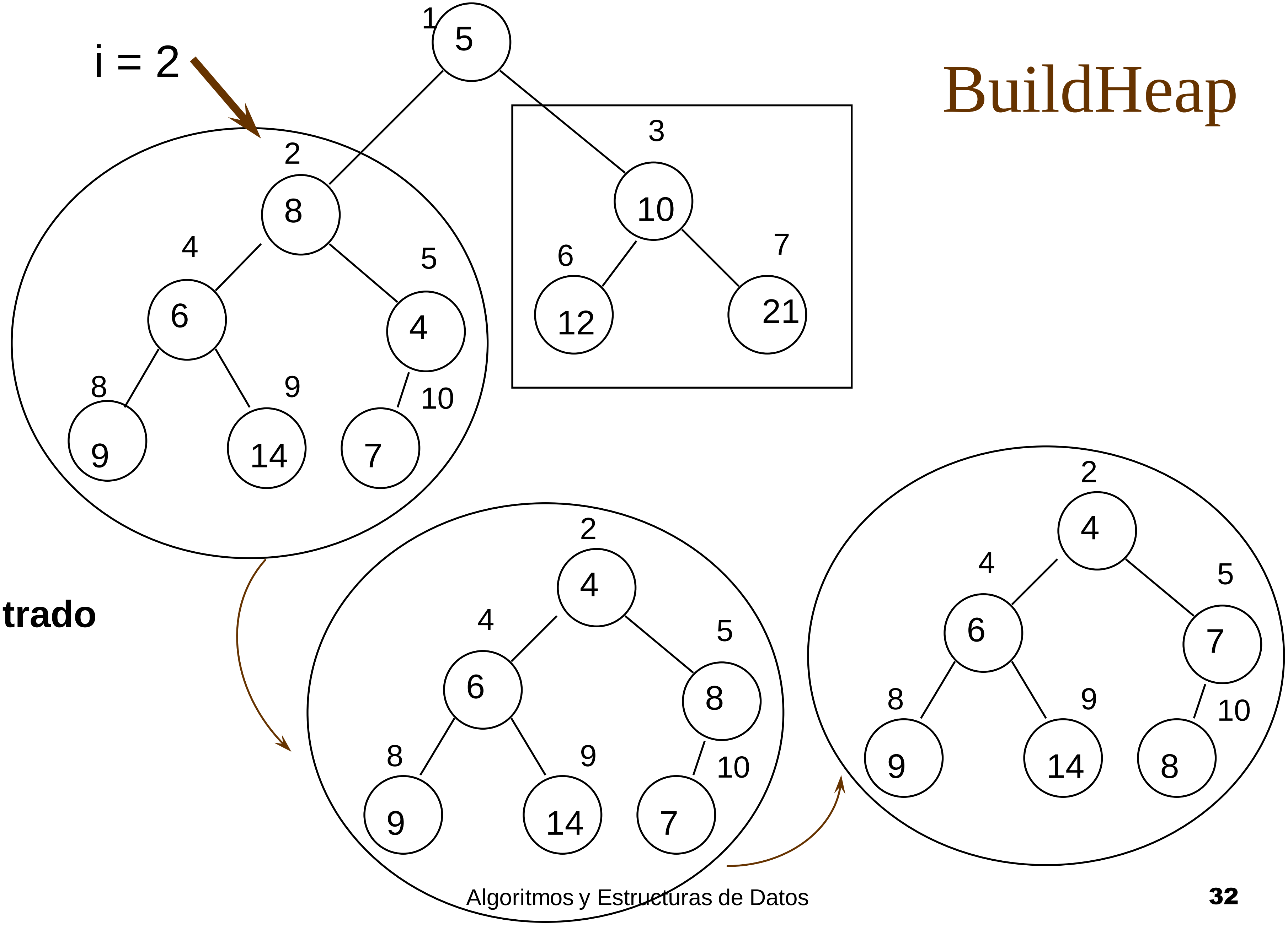
filtramos el 9

BuildHeap



BuildHeap

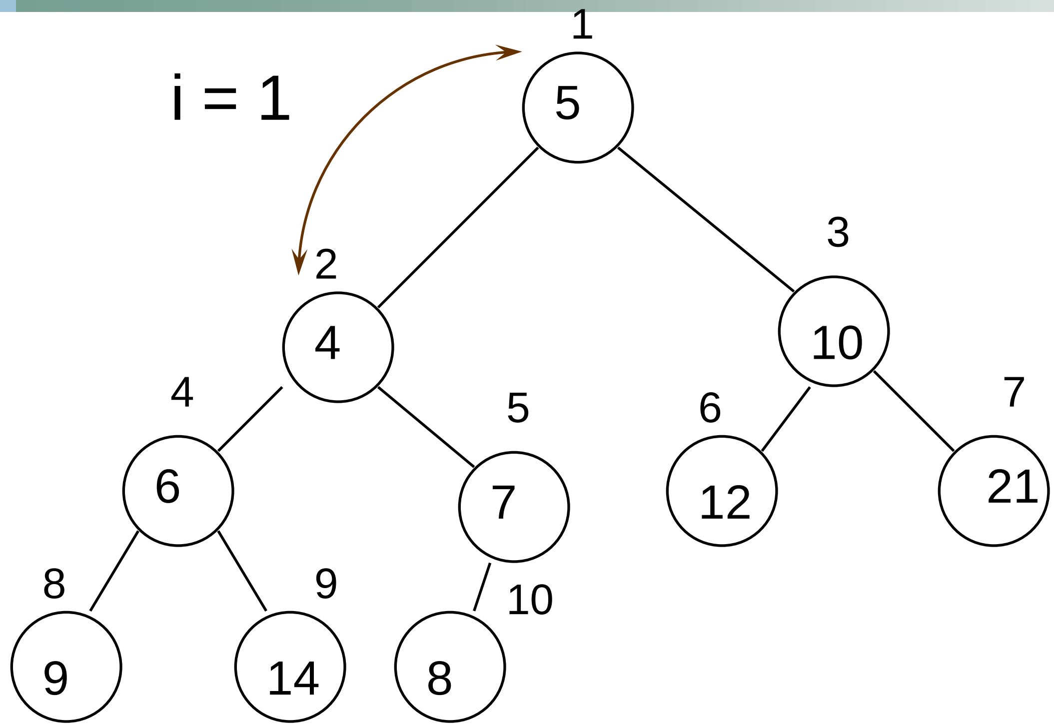
i = 2

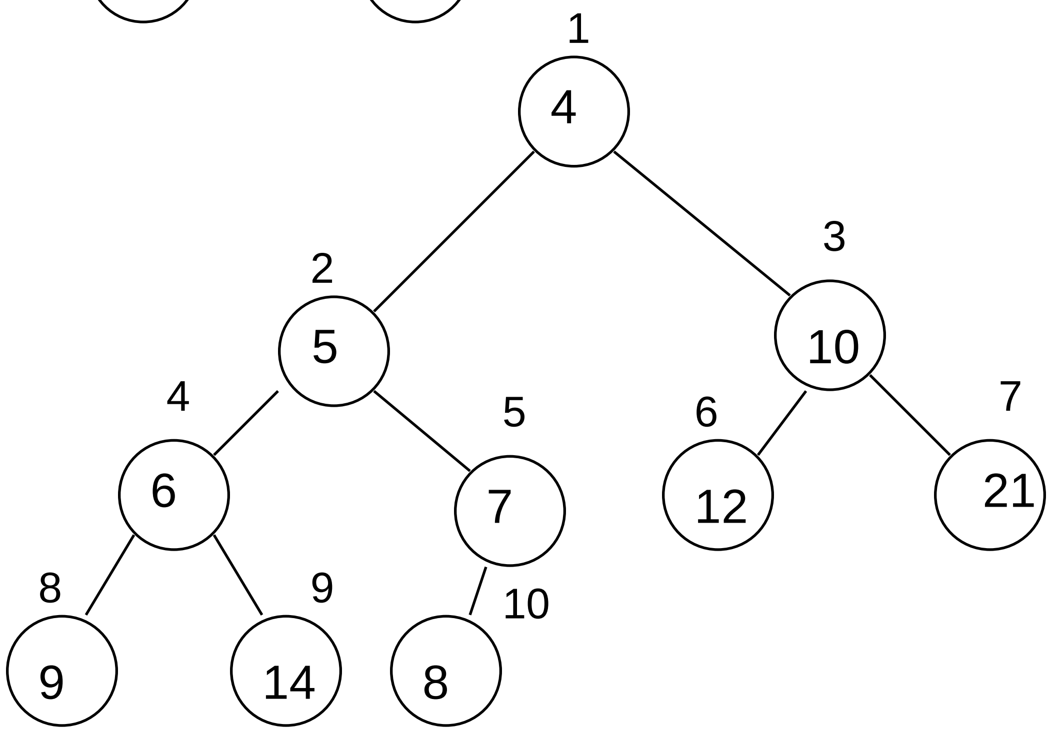


filtrado

BuildHeap

i = 1





Tiempo de ejecución

- En el filtrado de cada nodo recorremos su altura

- Para acotar el tiempo de ejecución de la operación BuilHeap, debemos calcular la suma de las alturas de todos los nodos

Tiempo de ejecución

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 $h – i$

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

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

Tiempo de ejecución (cont.)

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

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

Restando las dos igualdades (B) – (A)

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

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

$S + 1 = -h + (2^{h+1} – 1)$

$S = (2^{h+1} – 1) – (h + 1)$

Tiempo de ejecución (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 BuilHeap es lineal

Ordenación de vectores usando Heap

Dado un conjunto de N elementos se los quiere ordenar en forma creciente.

Existen dos alternativas:

a) *Algoritmo que usa una heap y es de O(n log n)*

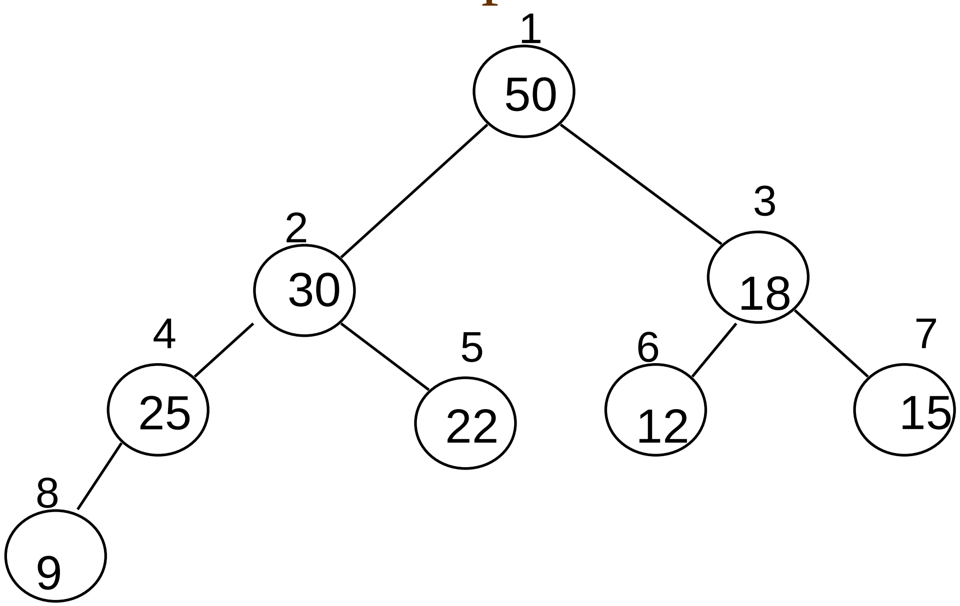
- Construir una MinHeap, realizar N DeleteMin operaciones e ir guardando los elementos extraídos en otro arreglo.
  - Desventaja: requiere el doble de espacio

Ordenación de vectores usando Heap

***b)** Algoritmo HeapSort de O(n log n)*

➢Construir una MaxHeap, 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:

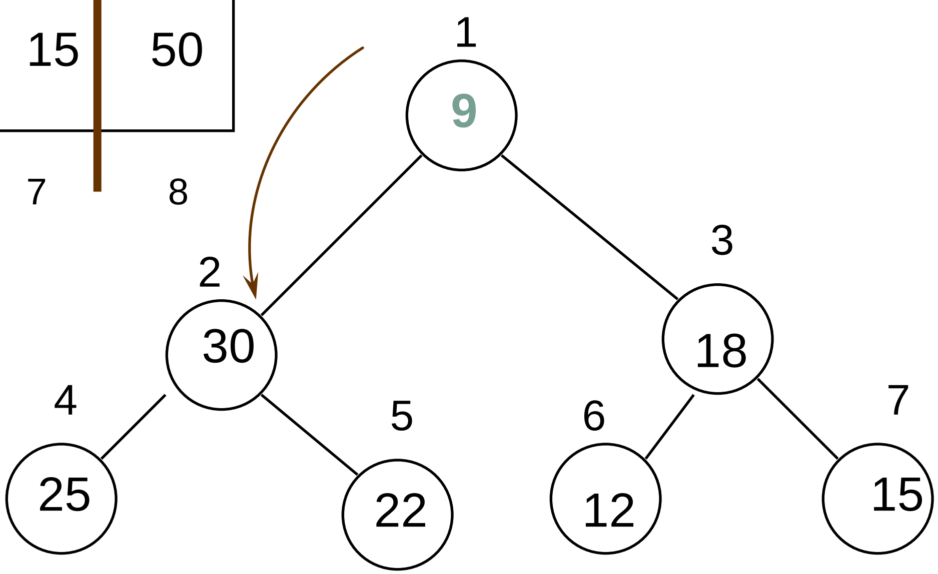


HeapSort (cont.)

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

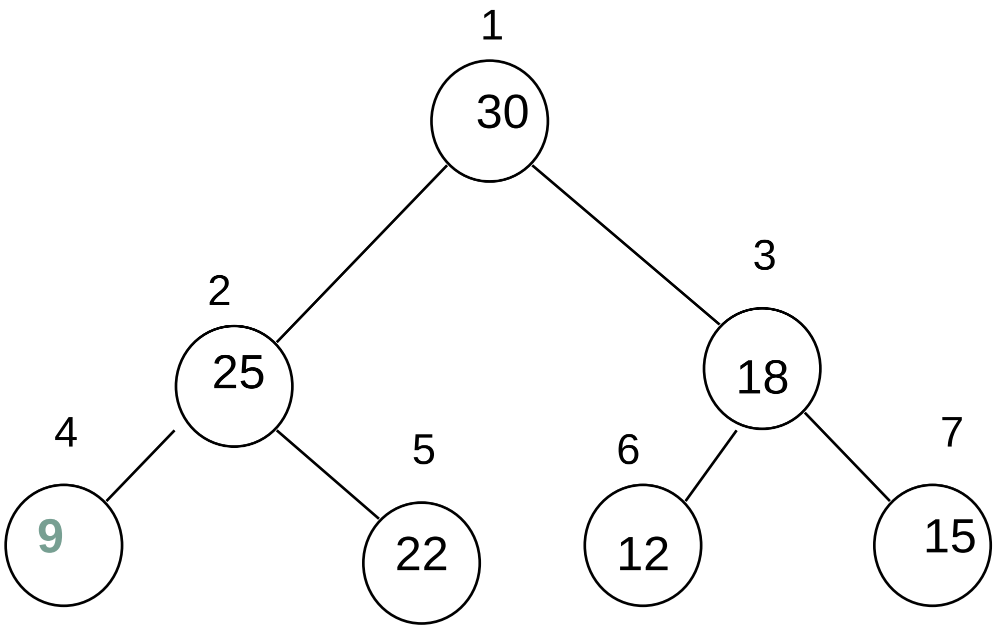
- Intercambio el primero con el último y
- decremento el tamaño

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



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

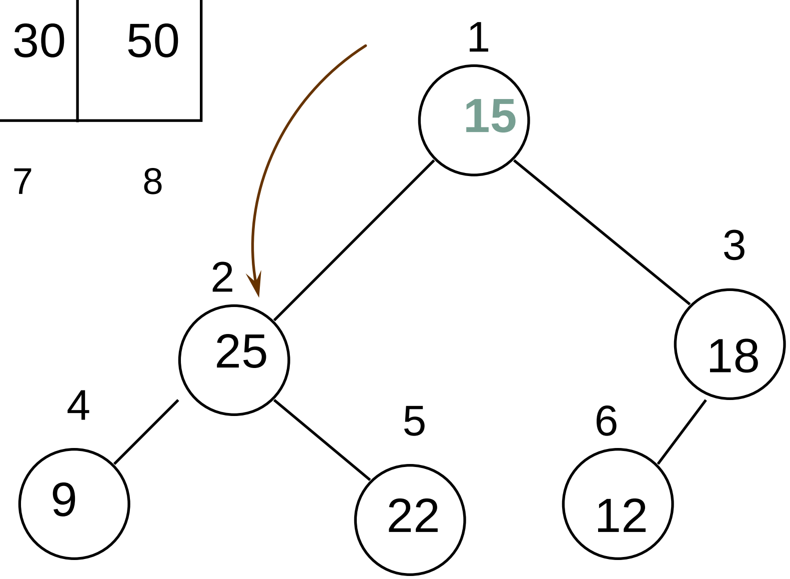
HeapSort (cont.)

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

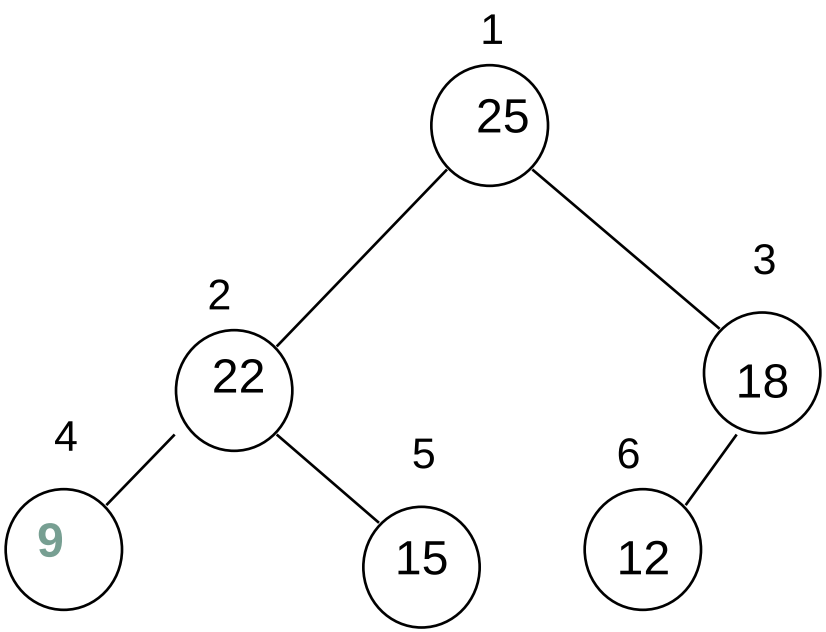
15	25	18	9	22	12	30	50
1	2	3	4	5	6	7	8

- 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

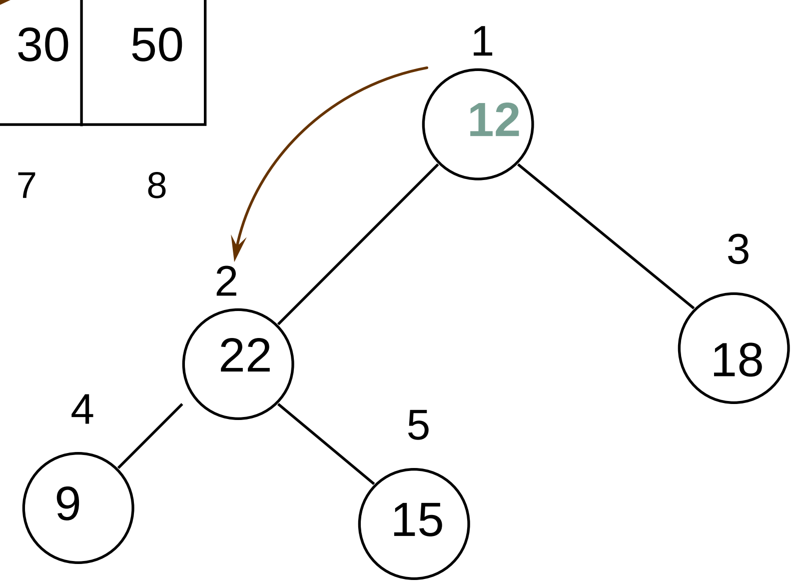
HeapSort (cont.)

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

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

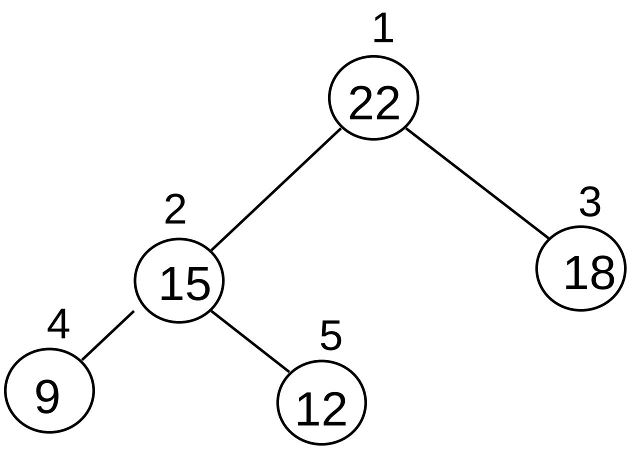
12	22	18	9	15	25	30	50
1	2	3	4	5	6	7	8

Filtrar el **12**



HeapSort (cont.)

Después de filtrar el **12** hacia abajo

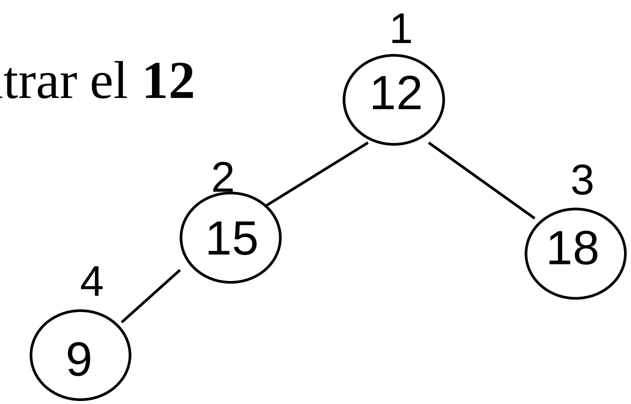


22	15	18	9	12	25	30	50
1	2	3	4	5	6	7	8

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

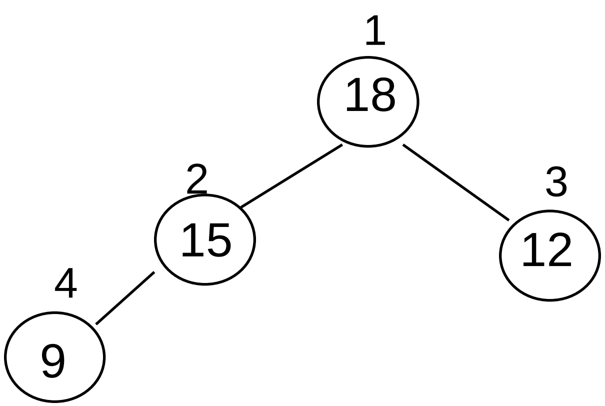
12	15	18	9	22	25	30	50
1	2	3	4	5	6	7	8

Filtrar el **12**



HeapSort (cont.)

Después de filtrar el **12** hacia abajo

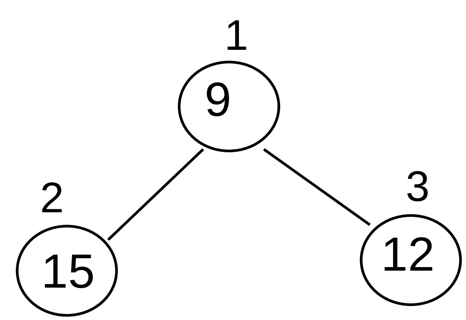


18	15	12	9	22	25	30	50
1	2	3	4	5	6	7	8

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

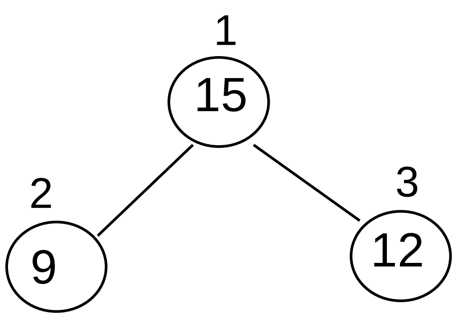
9	15	12	18	22	25	30	50
1	2	3	4	5	6	7	8

Filtrar el **9**



HeapSort (cont.)

Después de filtrar el **9** hacia abajo

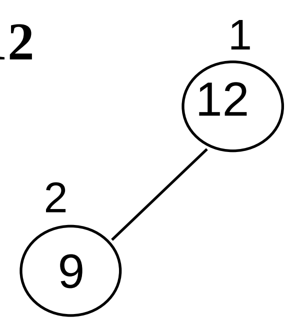


15	9	12	18	22	25	30	50
1	2	3	4	5	6	7	8

15	9	12	18	12	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**

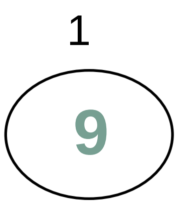


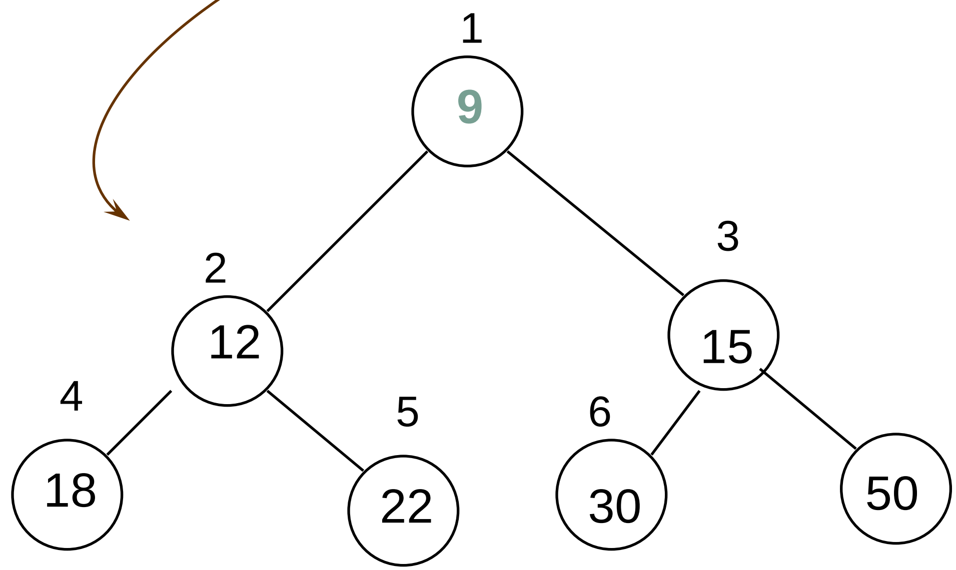
HeapSort (cont.)

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

Datos almacenados internamente

Heap conceptual





HeapSort (fin)