

ALGORITMOS Y ESTRUCTURA DE DATOS:

Colas con Prioridad y Montículos

Guillermo Román Díez

`groman@fi.upm.es`

Universidad Politécnica de Madrid

Curso 2021/2022

Colas con Prioridad

- En una cola FIFO el primer elemento en entrar es el primero en salir
- En las *colas con prioridad* el orden de salida viene determinado por la **prioridad** del elemento
- Se puede ver una cola con prioridad como una estructura de datos en la que los elementos se almacenan en orden de prioridad
 - ▶ A nivel de implementación no es necesario que esto ocurra así, lo importante es que la cola devuelva primero el elemento con más prioridad (ojo!)
- Las **entradas** de una cola con prioridad tienen
 - ▶ Una **clave** que indica la prioridad del elemento
 - ▶ Un **valor** que indica el elemento a insertar
 - ▶ Al par *clave-valor* lo llamamos **entrada** (entry)

Interfaz `Entry<K,V>`

- En el interfaz `Entry<K,V>` tenemos:
 - ▶ **K** es la clave (key) de la entrada que vamos a insertar
 - ▶ **V** es el valor (value) que vamos a insertar
- Por convención, *la clave establece la prioridad inversamente: cuanto menor es la clave mayor es la prioridad*
 - ▶ También se conocen como *min-max queue* porque al desencolar se devuelve el elemento con la menor clave
 - ▶ Se utilizará el orden total entre las claves. Los objetos que se usen para la clave deben ser `Comparable` o disponer de un `Comparator`

Interfaz `Entry<K,V>`

- En el interfaz `Entry<K,V>` tenemos:
 - ▶ **K** es la clave (key) de la entrada que vamos a insertar
 - ▶ **V** es el valor (value) que vamos a insertar
- Por convención, *la clave establece la prioridad inversamente: cuanto menor es la clave mayor es la prioridad*
 - ▶ También se conocen como *min-max queue* porque al desencolar se devuelve el elemento con la menor clave
 - ▶ Se utilizará el orden total entre las claves. Los objetos que se usen para la clave deben ser `Comparable` o disponer de un `Comparator`
- Nos podemos encontrar con:
 - ▶ Dos o más entradas con la misma clave pero distintos valores
 - ▶ Dos o más entradas con el mismo valor pero distintas claves

```
public interface PriorityQueue<K,V>
    extends Iterable<Entry<K,V>> {

    int size(); public boolean isEmpty();

    Entry<K,V> enqueue(K key, V value) throws
        InvalidKeyException;

    Entry<K,V> first() throws EmptyPriorityQueueException;

    Entry<K,V> dequeue() throws EmptyPriorityQueueException;

    // -----

    void remove(Entry<K,V> entry) throws InvalidKeyException;

    void replaceKey(Entry<K,V> entry, K newKey) throws
        InvalidKeyException;

    void replaceValue(Entry<K,V> entry, V newValue) throws
        InvalidKeyException;
}
```

Interfaz `PriorityQueue<K,V>`

- Los métodos `first` y `dequeue` devuelven objetos que implementan el interfaz `Entry<K,V>`
- `enqueue`: recibe por separado una clave `key` y un valor `value`
- `first`: es un método observador para consultar el elemento con mayor prioridad (con la clave con menor valor)
- `dequeue`: devuelve el elemento con mayor prioridad (con la clave con menor valor) y lo borra de la cola
- `EmptyPriorityQueueException` se lanza cuando se intenta acceder la entrada de clave mínima en una cola vacía
- `InvalidKeyException` se lanza cuando la clave es `null` o no tiene definido un orden para los elementos de su clase

Interfaz `PriorityQueue<K,V>`: `remove`, `replaceKey` y `replaceValue`

- Los métodos `remove`, `replaceKey` y `replaceValue` requieren un `Entry<K,V>` como argumento **pero no vale cualquier** `Entry<K,V>`!
- Los `Entry<K,V>` que sirven como argumentos a estos métodos tienen que haber sido devuelto por uno de los métodos:
 - ▶ `first`
 - ▶ `enqueue`
 - ▶ o usando el iterador sobre la cola
- Un `Entry<K,V>` devuelto por estos métodos contiene una referencia (position) dentro la estructura de datos implementando la cola, para poder implementar eficientemente los métodos `remove`, `replaceKey` y `replaceValue`.

Ejemplo de Colas con Prioridad

```
PriorityQueue<Integer,String> cola = new  
    SortedListPriorityQueue<Integer,String>();  
  
cola.enqueue(1, "Programacion II");  
cola.enqueue(4, "Algoritmica Numerica");  
cola.enqueue(3, "Lenguajes y Automatas");  
cola.enqueue(0, "AED");  
  
while (!cola.isEmpty()) {  
    ...println(cola.dequeue());  
}  
  
Entry<Integer,String> entry =  
    cola.enqueue(7, "Programacion Funcional");  
  
// Cambiamos programacion funcional a 9  
cola.replaceKey(entry, 9);  
// Lo borramos  
cola.remove(entry);
```


Implementación de Colas con Prioridad

- Con una **lista de posiciones desordenada**
 - ▶ enqueue tiene complejidad $O(1)$
 - ▶ first tiene complejidad $O(n)$
 - ▶ dequeue tiene complejidad $O(n)$
- Con una **lista de posiciones desordenada con caché**
 - ▶ enqueue tiene complejidad $O(1)$
 - ▶ first tiene complejidad $O(1)$
 - ▶ dequeue tiene complejidad $O(n)$
- Con una **lista de posiciones ordenada**
(incluida en aedlib.jar como SortedListPriorityQueue)
 - ▶ enqueue tiene complejidad $O(n)$
 - ▶ first tiene complejidad $O(1)$
 - ▶ dequeue tiene complejidad $O(1)$

Motivación Montículos

- Como acabamos de ver, en las implementaciones de colas con prioridad alguno de los métodos tiene complejidad $O(n)$
- El **montículo** es una implementación de colas con prioridad que satisface las siguientes complejidades

insert	$O(\log(n))$
first	$O(1)$
dequeue	$O(\log(n))$

- Los métodos adicionales también tienen complejidad logarítmica o mejor:

replaceKey	$O(\log(n))$
replaceValue	$O(1)$
remove	$O(\log(n))$

- Incluida en aedlib.jar como HeapPriorityQueue

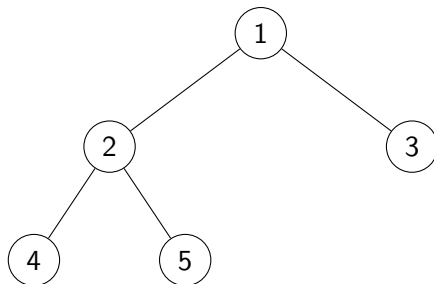
Motivación

- En la implementación de colas con prioridad mediante una lista usamos una *lista ordenada* usando el orden total de claves
- En una implementación con un **montículo** se utiliza un **árbol binario (casi)completo** para particionar la entrada
 - ▶ Reduciendo de esta forma las búsquedas a complejidad $O(\log(n))$
- Podemos decir que cambiamos una única fila de tamaño n por *múltiples filas* de tamaño $\log(n)$
- Aunque, en realidad un montículo se **describe** como un **árbol binario** pero se **implementa** mediante un **array**

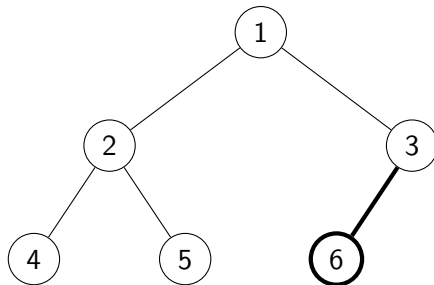
Terminología

- Un **árbol (casi)completo** es un árbol binario que puede estar completo o faltarle únicamente nodos “a la derecha” del último nivel
 - ▶ Todos los niveles (menos el último) deben estar completos
 - ▶ El último nivel se va llenando de izquierda a derecha
- Un nodo 'v' **está a la izquierda** de otro 'w' si 'v' aparece a la izquierda de 'w' en un recorrido en inorden
- El **último nodo de un árbol (casi)completo** es la hoja más a la derecha en el último nivel
- La **altura** de un árbol casi completo es $\log(n)$, donde n es el número de nodos almacenados en el árbol
- El **último nodo** de un árbol casi completo es el que se encuentra más a la derecha en el último nivel

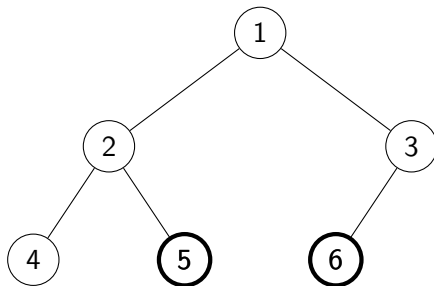
Árbol (casi)completo



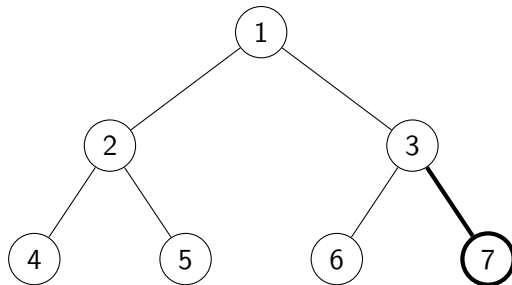
Añadimos el nodo 6



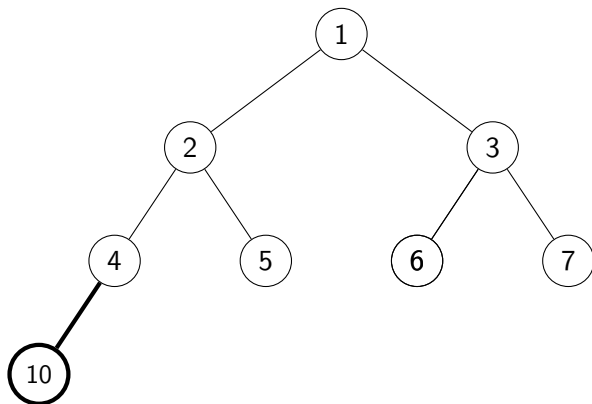
El nodo **5** está a la izquierda de **6**



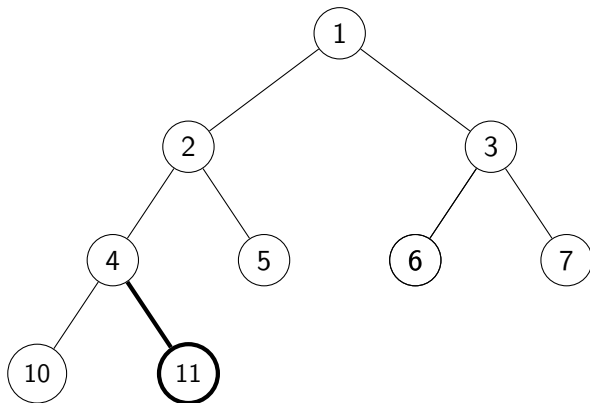
Añadimos el nodo 7



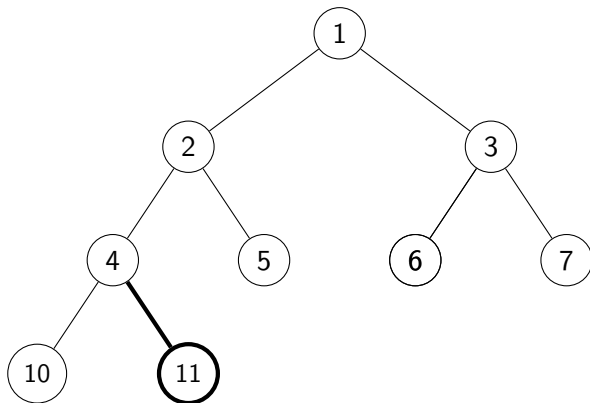
Al tener el nivel 2 completo, **cambiamos de nivel**



Añadimos más nodos en el nuevo nivel



El nodo **11** es el **último nodo**



Definición Montículo

Montículo

“Es un árbol binario (casi)completo que almacena entradas en los nodos tal que para todo nodo distinto de la raíz, su entrada es mayor o igual que la entrada almacenada en el nodo padre”

- El nombre “montículo” viene de “amontonar” claves en orden ascendente
- Cumple la **heap-order property**
 - ▶ Todos los caminos de la raíz a las hojas están ordenados ascendentemente
 - ▶ Si lo usamos para una *priority queue*, la entrada de menor clave (la más prioritaria) está almacenada en el nodo raíz

PriorityQueues con montículos

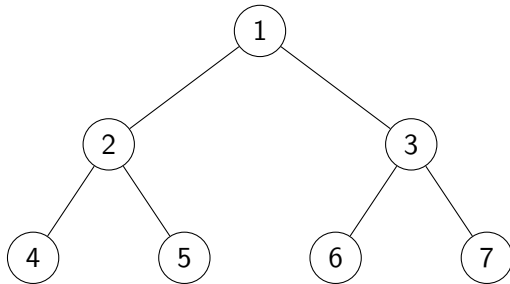
- Podemos implementar una **cola con prioridad** mediante un **montículo**
- El montículo se ordena considerando las claves de los elementos de la cola con prioridad
- Al igual que en la implementación con listas, necesitamos un **Comparator** o bien que las claves sean **Comparable**
- Podríamos conseguir que las operaciones de una cola con prioridad tengan las siguientes complejidades:

enqueue	$O(\log(n))$
first	$O(1)$
dequeue	$O(\log(n))$

Montículo y Heap-order Property

Pregunta

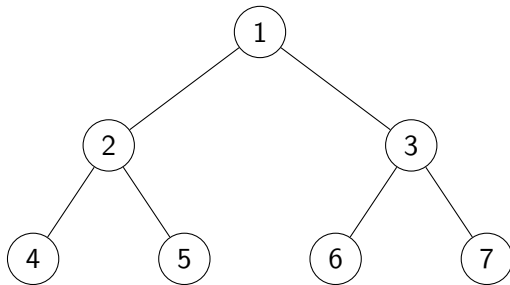
¿tiene el padre de cada nodo mayor o igual prioridad que el nodo?



Montículo y Heap-order Property

Pregunta

¿tiene el padre de cada nodo mayor o igual prioridad que el nodo?

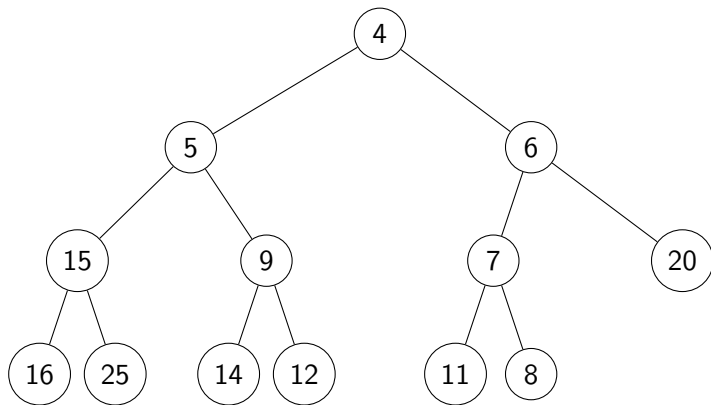


Si es un árbol casi completo y se cumple para todos los nodos, entonces, es un **montículo** que puede implementar una **cola con prioridad**

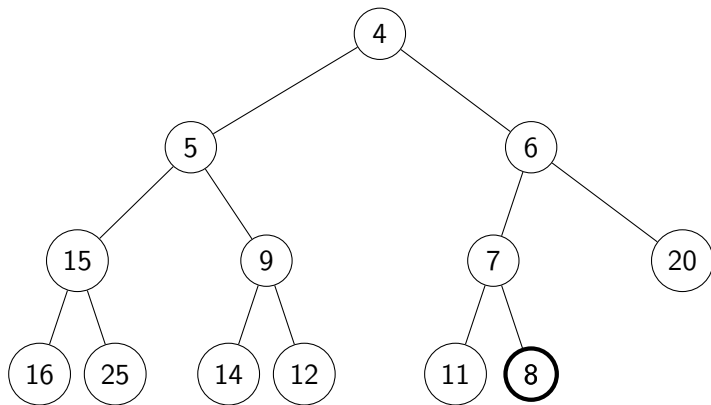
Operación enqueue

- La inserción de un nuevo nodo se hace incluyendo el nuevo nodo como el **último nodo del árbol**
- Como ésto puede *violar* la *heap-order property* puede ser necesario reajustar los nodos del árbol
 - ▶ Se comprueba la "heap-order property" entre el nuevo nodo y su padre
 - ★ Si se cumple hemos acabado
 - ★ Si no se cumple entonces se intercambian las entradas entre el nodo nuevo y el padre
 - ▶ Se repite la operación con el nodo intercambiado y el padre correspondiente hasta que se cumpla la "heap-order property" o hasta llegar a la raíz
- Esto se conoce como **up-heap bubbling**

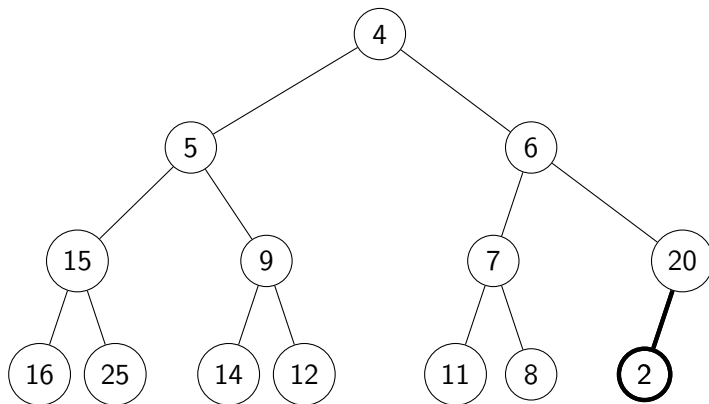
up-heap bubbling



up-heap bubbling

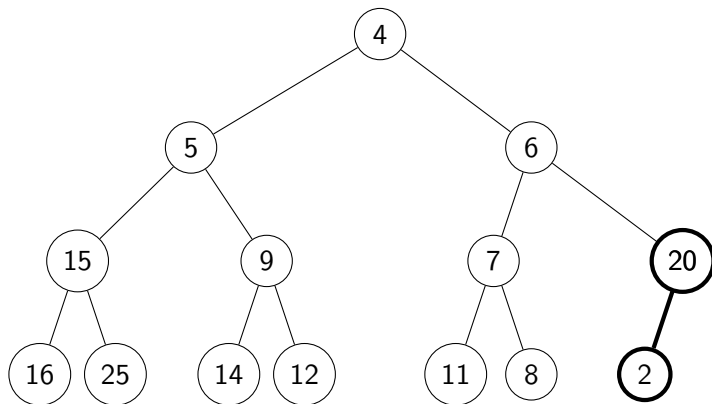


up-heap bubbling



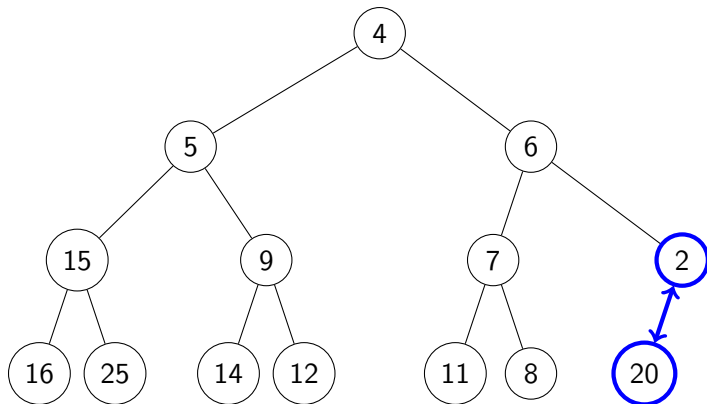
Insertamos el **nuevo** 2 nodo en la **último** posición libre

up-heap bubbling



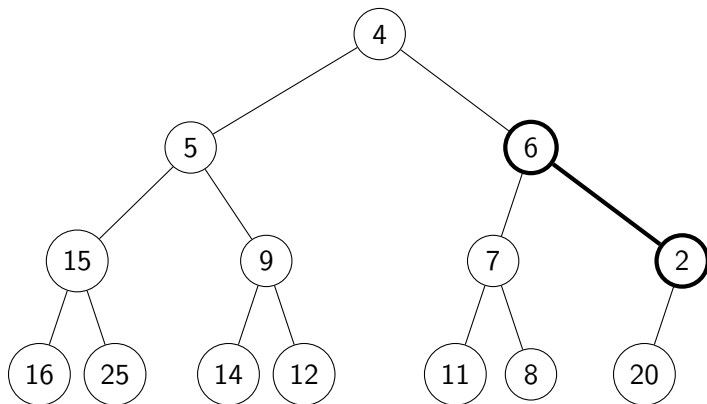
¿violamos la heap-order property?

up-heap bubbling



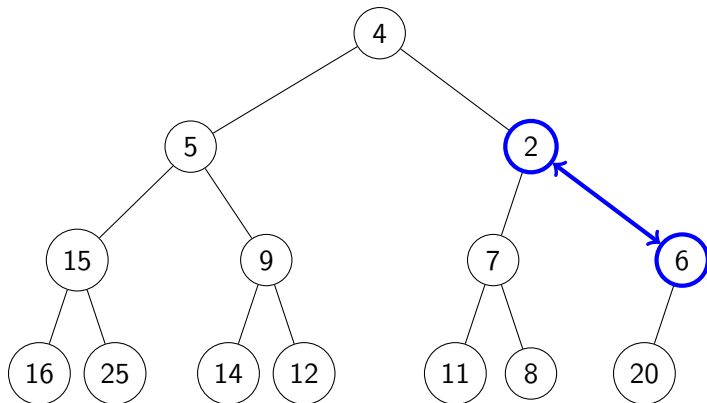
Intercambiamos los nodos

up-heap bubbling



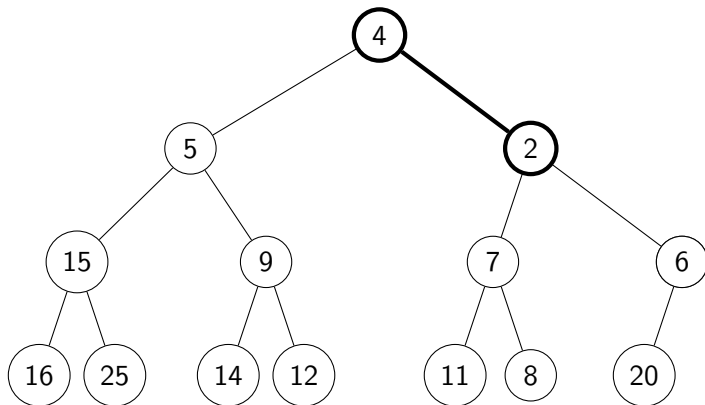
¿violamos la heap-order property?

up-heap bubbling



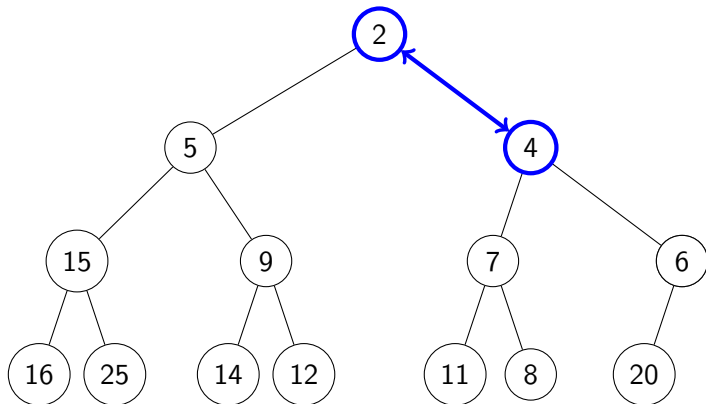
Intercambiamos los nodos

up-heap bubbling



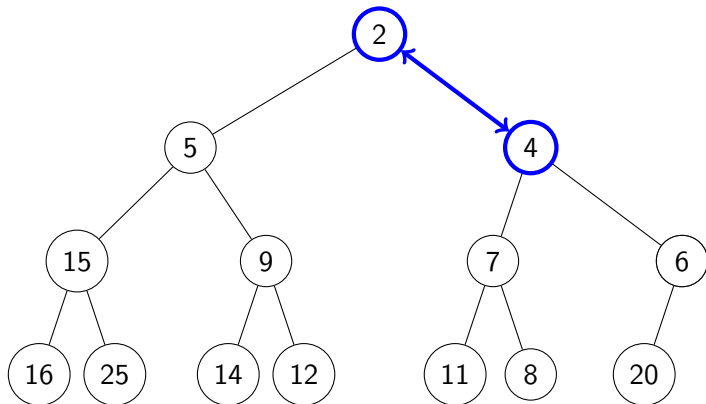
¿violamos la heap-order property?

up-heap bubbling



Intercambiamos los nodos

up-heap bubbling

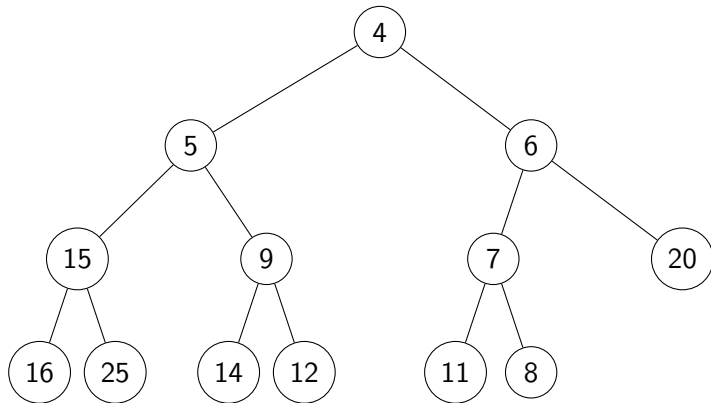


FIN

Operación de queue

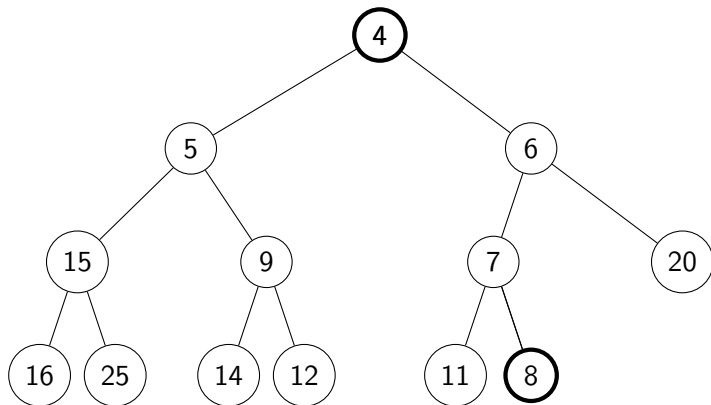
- La entrada de menor clave siempre es la raíz, pero la raíz no se puede borrar directamente
- **Intercambiamos** la **raíz** con el **último nodo** del árbol y borramos el último nodo
- Como ésto puede *violar* la *heap-order property* puede ser necesario reajustar los nodos del árbol
 - ▶ Se comprueba la *heap-order property* entre la raíz y sus hijos
 - ★ Si se cumple hemos acabado
 - ★ Si no se cumple entonces se intercambian las entradas entre la raíz y el nodo hijo de menor clave
 - ▶ Se repite la operación con el hijo intercambiado y sus hijos hasta que se cumpla la "heap-order property" o hasta llegar a una hoja
- Esto se conoce como **down-heap bubbling**

down-heap bubbling



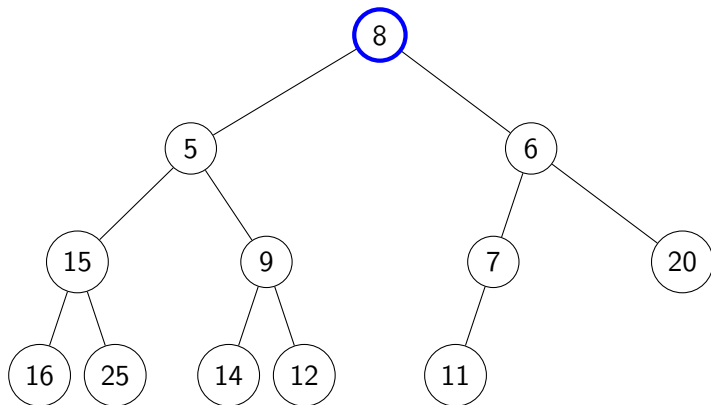
Ejecutamos dequeue

down-heap bubbling



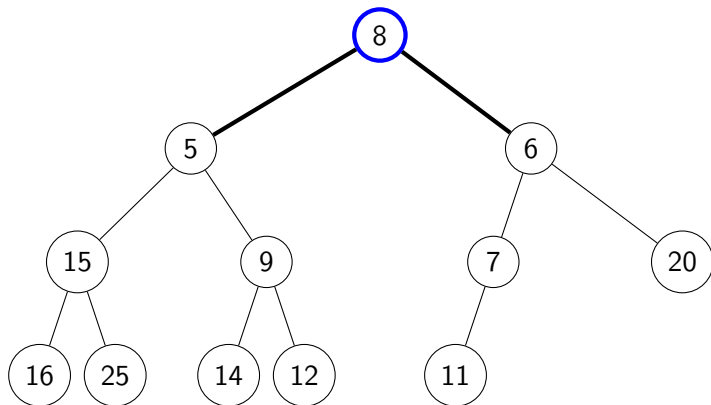
Quitamos la raíz y ponemos el último como raíz

down-heap bubbling



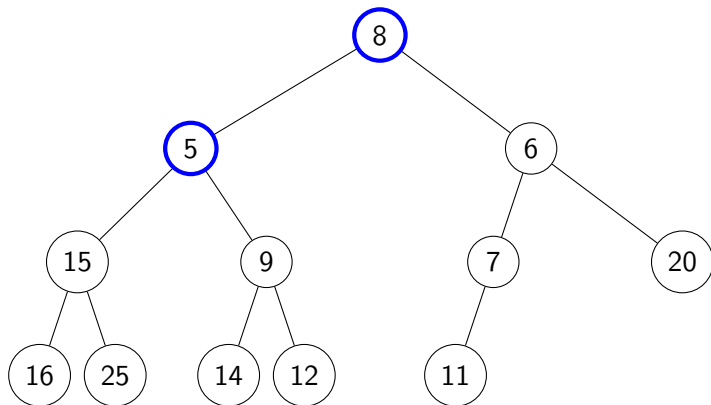
Quitamos la raíz y ponemos el último como raíz

down-heap bubbling



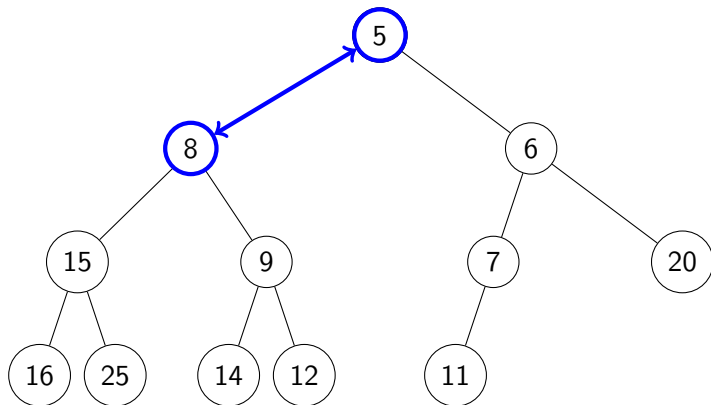
¿violamos la heap-order property?

down-heap bubbling



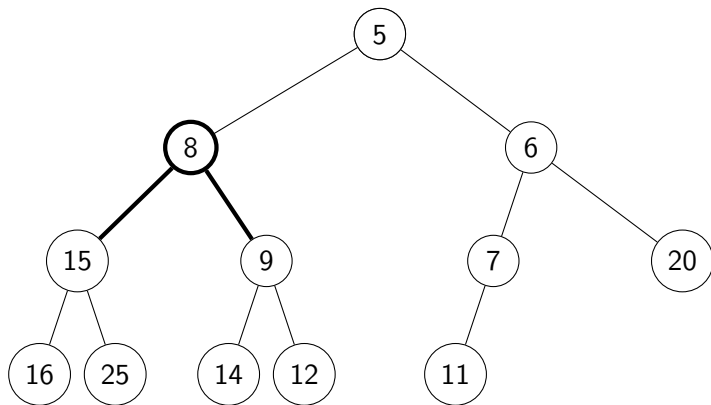
Intercambiamos con el hijo con mayor prioridad

down-heap bubbling



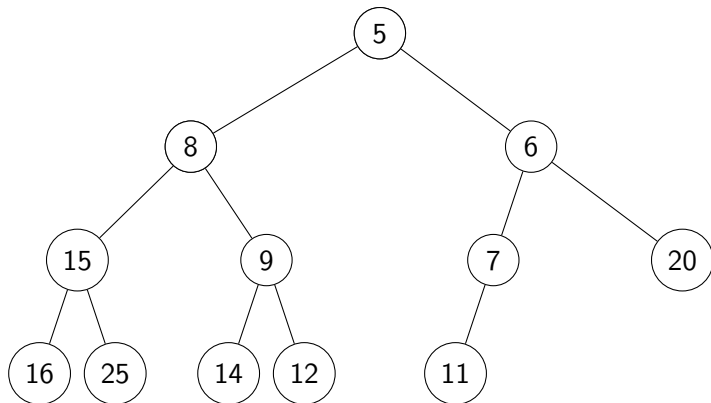
Intercambiamos con el hijo con mayor prioridad

down-heap bubbling



¿violamos la heap-order property?

down-heap bubbling

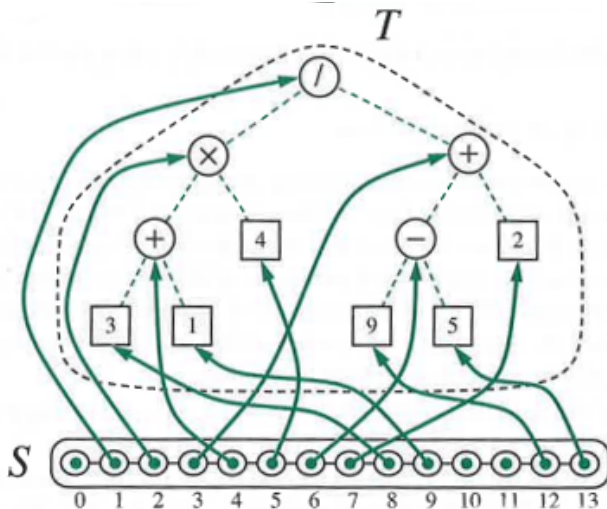


FIN

Implementando un árbol binario con un Array

- Un **árbol binario (casi)completo** se puede implementar mediante un **array**
- Como el árbol es perfecto es todos los niveles menos el último
- En el último nivel se meten de izquierda a derecha, es decir, **consecutivamente** en el array
- Dado un nodo en el índice i en el array
 - ▶ su padre *siempre* esta en el índice $(i - 2)/2$ (si existe)
 - ▶ su hijo izquierdo *siempre* esta en el índice $i * 2 + 1$ (si existe)
 - ▶ su hijo derecho *siempre* esta en el índice $i * 2 + 2$ (si existe)
- Esto nos permite tener complejidad $O(1)$ para
 - ▶ En las inserciones por el final
 - ▶ En el borrado del último elemento
 - ▶ Las operaciones de intercambio entre dos nodos
- Con esta implementación conseguimos que la inserción y el borrado tengan complejidad $O(\log(n))$

Implementando un árbol binario con un Array



Heap-sort

- Ya hemos visto que en una cola con prioridad implementada con un montículo, las inserciones se pueden hacer con $O(\log(n))$
- Podemos utilizar esto para implementar un algoritmo de ordenación eficiente
- El algoritmo **heap-sort** ordena los elementos de una lista con complejidad $O(n \cdot \log(n))$
 - ▶ Otros algoritmos de ordenación como *bubble-sort* o *quick-sort* tienen complejidad $O(n^2)$
 - ▶ *Merge-sort* o *shell-sort* también tienen complejidad $O(n \cdot \log(n))$