HEAP Binaria

- Introducción
- Implementación en JAVA de la MaxHeap
- Inserción de un elemento
- Eliminación del elemento máximo
- Los métodos de filtrado ascendente y descendente
  - percolate_up()
  - percolate_down(indice)

HEAP Binaria
Introducción

Una **heap binaria** es un **árbol binario completo** que satisface la propiedad de orden de la heap. El orden puede ser de alguno de estos 2 tipos:

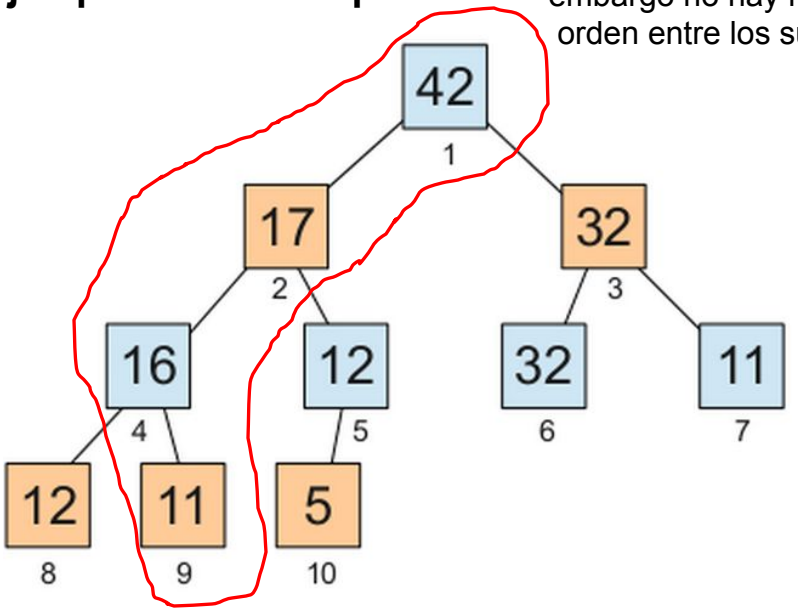
Propiedad de la MaxHeap: el valor de cada nodo es menor o igual que el de su padre/madre y el elemento máximo está en la raíz.

Propiedad de la MinHeap: el valor de cada nodo es mayor o igual que el de su padre/madre y el elemento mínimo está en la raíz.

La **implementación con arreglos** usa algunas propiedades para determinar dado un elemento, el índice donde están sus hijos o su padre/madre. Por conveniencia NO se usa la posición **0.**

Ejemplo de MaxHeap:

Cada camino está ordenado, sin embargo no hay relación de orden entre los subárboles

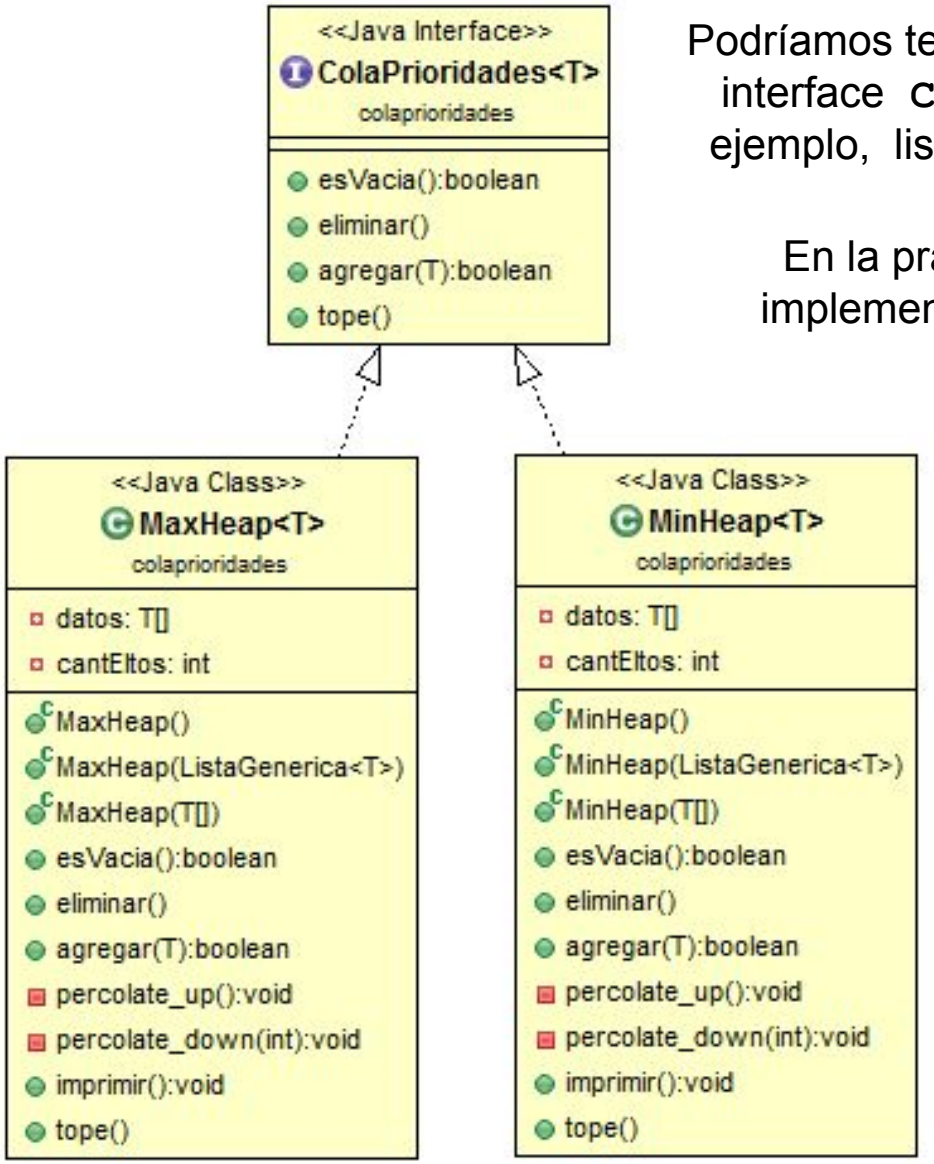


La raíz está almacenada en la posición 1

42	17	32	16	12	32	11	12	11	5
1	2	3	4	5	6	7	8	9	10

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]

Cola de Prioridades - HEAPs



Podríamos tener múltiples implementaciones de la interface **`ColaPrioridades<T>`** usando, por ejemplo, listas ordenadas, listas desordenadas, ABB, etc.

En la práctica usaremos un árbol binario implementado en un arreglo para Colas de Prioridades -> HEAPs

MaxHeap
Constructores

```
package heap;

public class MaxHeap<T extends Comparable<T>>
                  implemnets ColaPrioridades<T> {
 private T[] datos = (T[]) new Comparable[100];
 private int cantEltos = 0;

 public MaxHeap() {}

 public MaxHeap(ListaGenerica<T> lista){
 lista.comenzar();
   while(!lista.fin()){
     this.agregar(lista.proximo());
 }

 public MaxHeap(T[] elementos) {
  for (int i=0; i<elementos.length; i++) {
     cantEltos++;
     datos[cantEltos] = elementos[i];
  }
  for (int i=cantEltos/2; i>0; i--)
     this.percolate_down(i);
 }
 //Más Métodos
}
```

En java no se puede crear un arreglo de elementos T:

```
private T[] datos = new T[100];
```

Una opción es crear un arreglo de Comparable y castearlo:

```
private T[] datos=(T[]) new Comparable[100];
```

O(n.log n)

En este constructor se recibe una lista para inicializar la heap, se recorre y para cada elemento se agrega y se filtra. El **agregar()** es el método de la **HEAP.** El **agregar()** invoca al **`percolate_up()`**.

Orden lineal

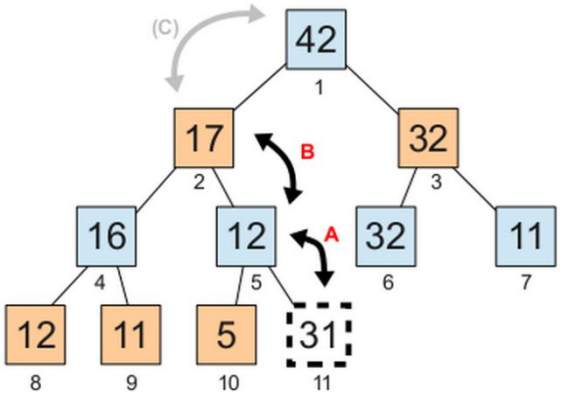
En este constructor, **después de agregar** todos los elementos en la heap en el orden en que vienen en el arreglo enviado por parámetro, se **restaura la propiedad de orden** intercambiando el dato de cada nodo hacia abajo a lo largo del camino que contiene los hijos máximos invocando al método **`percolate_down()`**.

HEAP

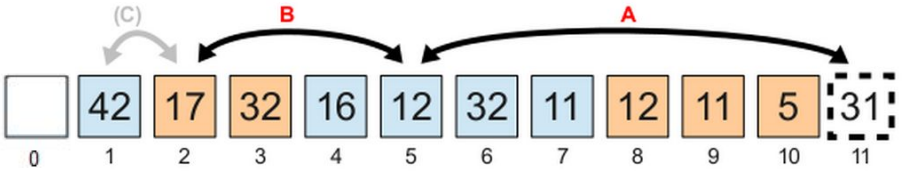
Filtrado hacia arriba: percolate_up() - Inserción

```
package heap;

public class Heap<T extends Comparable<T>>
                    implemnets ColaPrioridades<T> {
 private T[] datos = (T[]) new Comparable[100];
 private int cantEltos = 0;

 public boolean agregar(T elemento) {
  this.cantEltos++;
  this.datos[cantEltos] = elemento;
  this.percolate_up(cantEltos);
  return true;
 }

 private void percolate_up(int indice) {
  T temporal = datos[indice];
  while (indice/2>0
         && datos[indice/2].compareTo(temporal)<0){
         datos[indice] = datos[indice/2];
         indice = indice/2;
  }
  datos[indice] = temporal;
 }
}
```

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



Hay que intercambiar el 31 con el 12 y después con el 17 porque 31 es mayor que ambos. El último intercambio (c), no se realiza porque el 31 es menor que 42.



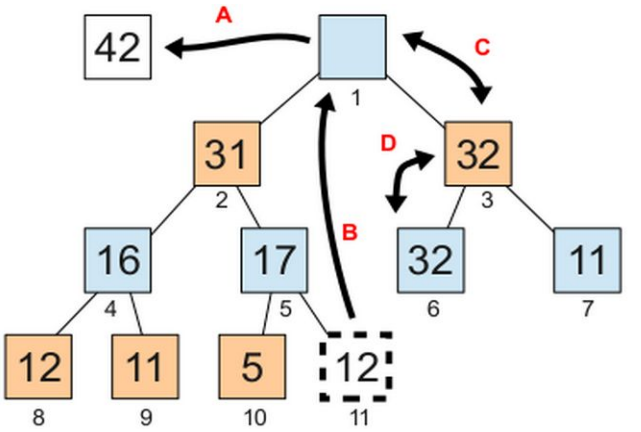
El filtrado hacia arriba restaura la propiedad de orden intercambiando el *elemento insertado* a lo largo del camino hacia arriba desde el lugar de inserción

HEAP

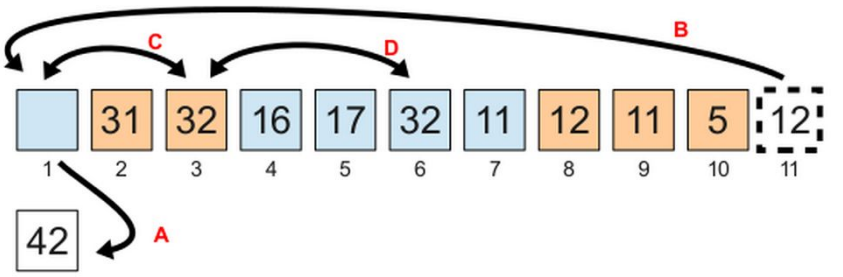
Eliminar máximo - percolate_down()

```
package heap;

public class Heap<T extends Comparable<T>>
                  implemnets ColaPrioridades<T>{
 private T[] datos = (T[]) new Comparable[100];
 private int cantEltos = 0;

public T eliminar() {
 if (this.cantElto > 0) {
  T elemento = this.datos[1];
  this.datos[1] = this.datos[this.cantEltos];
  this.cantEltos--;
  this.percolate_down(1);
  return elemento;
 }
 return null;
}

public T tope() {
 return this.datos[1];
}

public boolean esVacia() {
 if (this.cantEltos>0) {
   return false;
 }
 return true;
}
. . .
}
```

Para extraer un elemento de la heap hay que moverlo a una variable temporal, mover el último elemento de la heap al hueco, y hacerlo bajar mediante intercambios hasta restablecer la propiedad de orden (cada nodo debe ser mayor o igual que sus descendientes).



En el arreglo se ve así:



HEAP

Filtrado hacia abajo: *percolate_down*

Para filtrar: se elige el mayor de los hijos y se lo compara con el padre.

```
package heap;
public class MaxHeap<T extends Comparable<T>>
                        implemnets ColaPrioridades<T> {
 private T[] datos = (T[]) new Comparable[100];
 private int cantEltos = 0;

 private void percolate_down(int posicion) {
   T candidato = datos[posicion];
   boolean detener_percolate = false;
   while (2 * posicion <= cantEltos && !detener_percolate) {
      //buscar el hijo maximo de candidato (hijo_máximo es el indice)
      int hijo_maximo = 2 * posicion;
      if (hijo_maximo != this.cantEltos){ //hay mas eltos, tiene hderecho
        if (datos[hijo_maximo + 1].compareTo(datos[hijo_maximo]) > 0) {
            hijo_maximo++;
        }
      }
      if (candidato.compareTo(datos[hijo_maximo]) < 0) { //padre<hijo
            datos[posicion] = datos[hijo_maximo];
            posicion = hijo_maximo;
      } else {
           detener_percolate = true;
      }
   }
   this.datos[posicion] = candidato;
 }
 //Más métodos
}
```