La interface Comparable<T> en
Colas de prioridades

El código de esta presentación es a nivel informativo.
No se implementarán estas estructuras en la práctica.

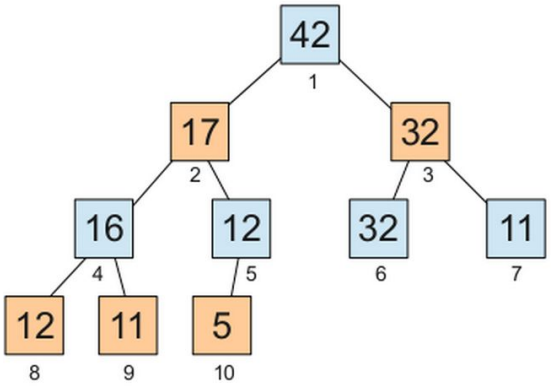
Colas de Prioridad

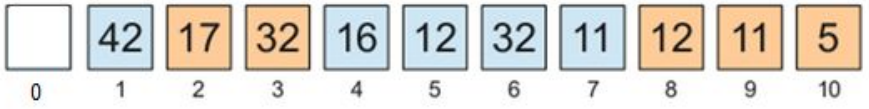
En las *colas de prioridad*, el *orden lógico* de sus elementos está determinado por la prioridad de los mismos. Los elementos de mayor prioridad están en el frente de la cola y los de menor prioridad están al final. De esta manera, cuando se encola un elemento, puede suceder que éste se mueva hasta el comienzo de la cola.

Hay varias implementaciones de cola de prioridad (listas ordenadas, listas desordenadas, ABB, etc.) pero la manera clásica es utilizar una ***heap binaria***. La heap binaria implementa la cola "usando" un árbol binario que permite encolar y desencolar con un O(log n) y debe cumplir dos propiedades:

- *propiedad estructural:* ser un árbol binario completo de altura h, es decir, un árbol binario lleno de altura h-1 y en el nivel h, los nodos se completan de izquierda a derecha.

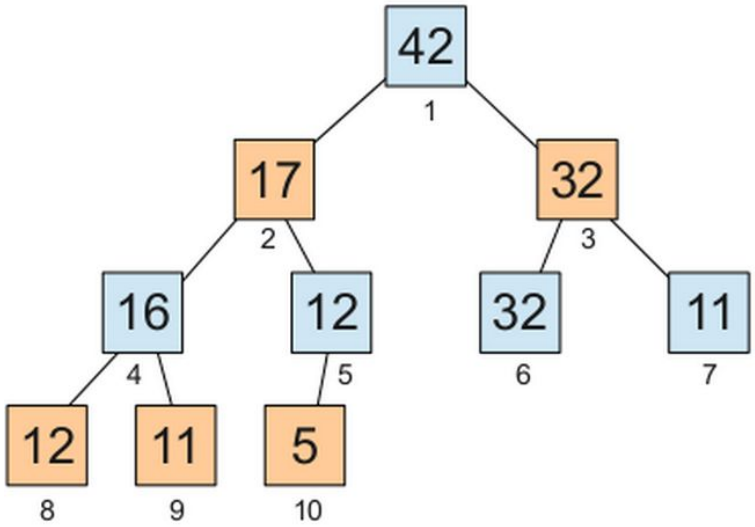
- *propiedad de orden:* El elemento máximo (en MaxHeap) está almacenado en la raíz y para cada nodo, el valor almacenado es mayor o igual al de sus hijos (en MinHeap es inverso).



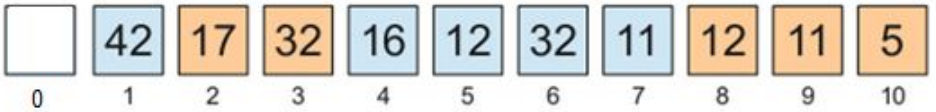


Colas de Prioridad

Usando un arreglo para almacenar los elementos, podemos usar algunas propiedades para determinar, dado un elemento, el lugar donde están sus hijos o su padre.



La raíz está almacenada en la posición 1



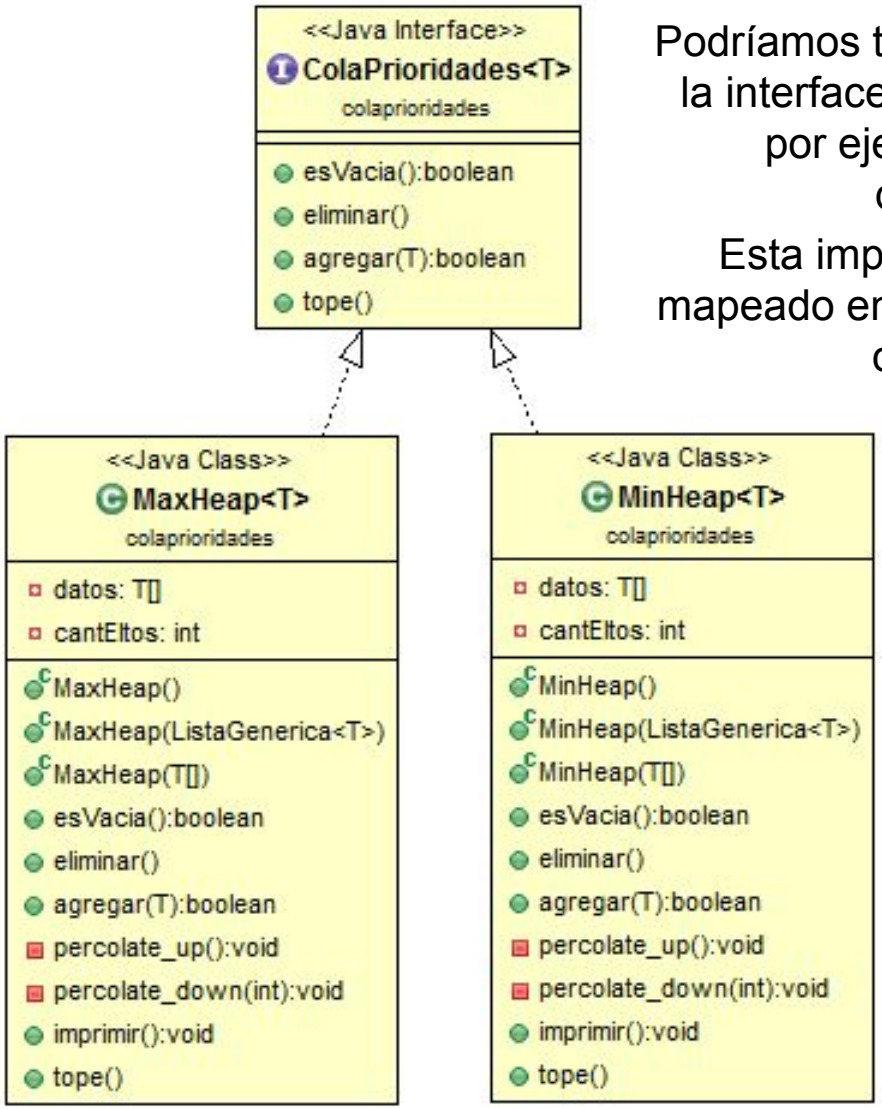
El hijo izquierdo está en la posición 2*i
El hijo derecho está en la posición 2*i + 1
El padre está en la posición [i/2]

Hay dos tipo de heaps: ***MinHeap*** y ***MaxHeap***.

MinHeap: el valor de la raíz es menor que el valor de sus hijos y éstos son raíces de minHeaps

MaxHeap: el valor de la raíz es mayor que el valor de sus hijos y éstos son raíces de maxHeaps.

Cola de Prioridades - HEAPs



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

Esta implementación usa un árbol binario mapeado en un arreglo para implementar Colas de Prioridades -> HEAPs

MaxHeap
Constructores

```
package heap;

public class MaxHeap<T extends Comparable<T>>
                  implements 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);
 }
 . . .
}
```

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 la lista, 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 (conocido como BuildHeap)

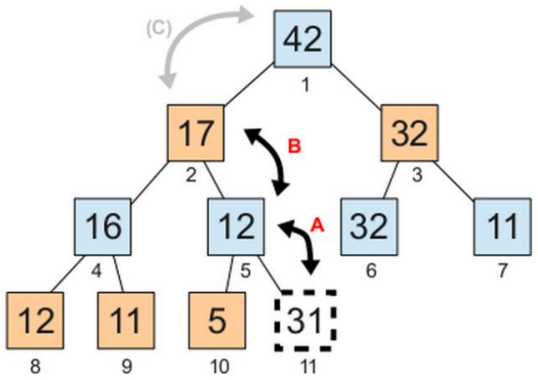
En este constructor, después de agregar todos los elementos en la heap, en el orden en que vienen en el arreglo enviado por parámetro, restaura la propiedad de orden intercambiando el dato de cada nodo hacia abajo a lo largo del camino que contiene los hijos máximos invocando al método **`percolate_down()`**.

HEAP

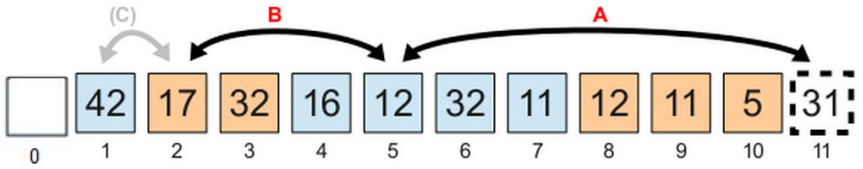
Insertar/Agregar un elemento

```
package heap;

public class MaxHeap<T extends Comparable<T>>
                  implements 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 31 es menor que 42.



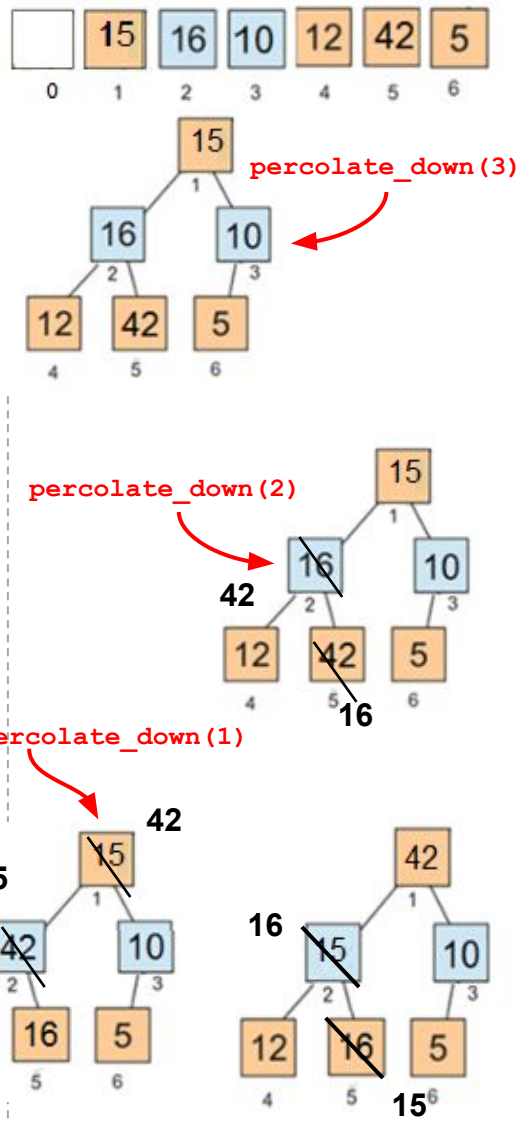
El filtrado hacia arriba restaura la propiedad de orden intercambiando el *elemento insertado* a lo largo del camino hacia arriba desde el lugar de inserción

HEAP

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>>
                        implements 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 índice)
    int hijo_maximo = 2 * posicion;
    if (hijo_maximo != this.cantEltos){//hay+ eltos,tiene hijo derecho
       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;
 }
 . . .
}
```

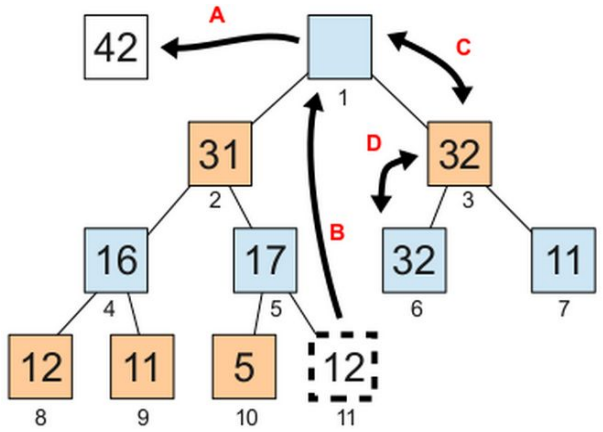


HEAP

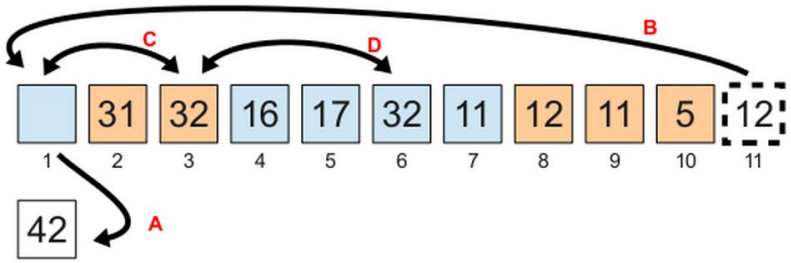
Eliminar máximo – tope()

```
package heap;

public class Heap<T extends Comparable<T>>
                implements 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:

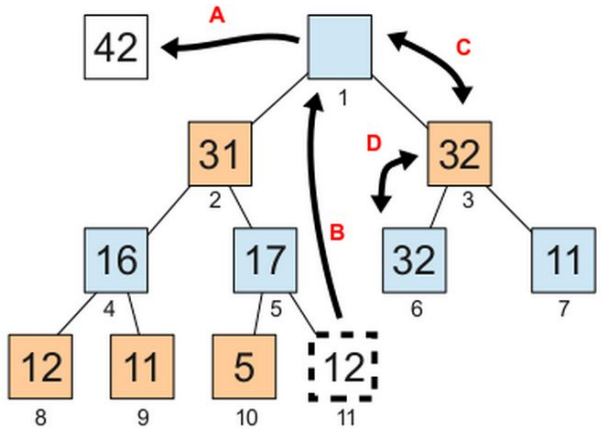


HEAP

Eliminar máximo – tope()

```
package heap;

public class Heap<T extends Comparable<T>>
                implements 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: