# 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:

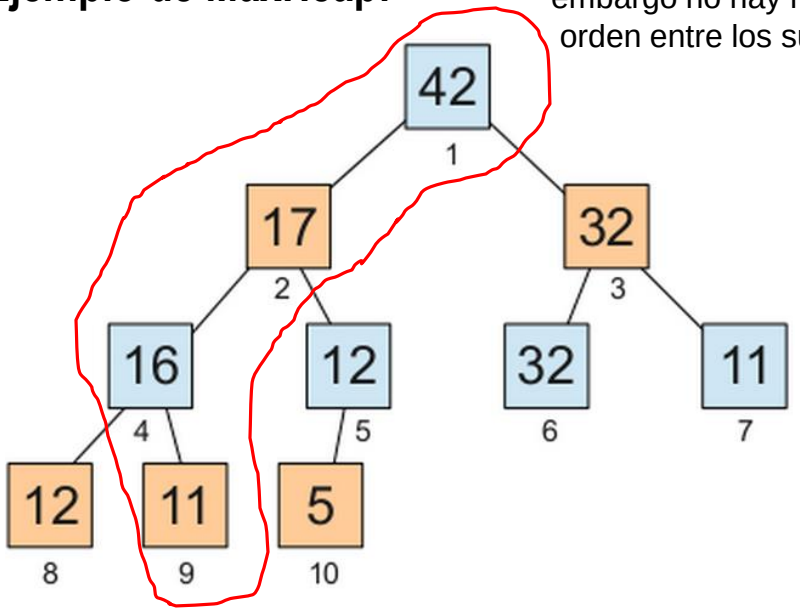
***Propiedad de la MinHeap***: el valor de cada nodo es mayor o igual que el de su padre/madre y el elemento mínimo está en la raíz.

***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.

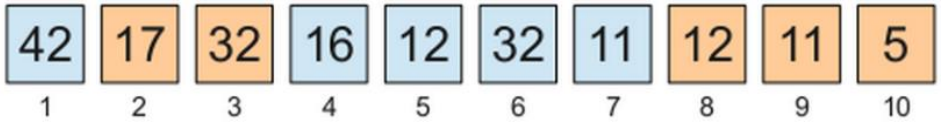
La **implementación con arreglos** usa algunas propiedades para determinar dado un elemento, el índice donde están sus hijos o su padre/madre. Por conveniencia NO se usa la posición **0.**

**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

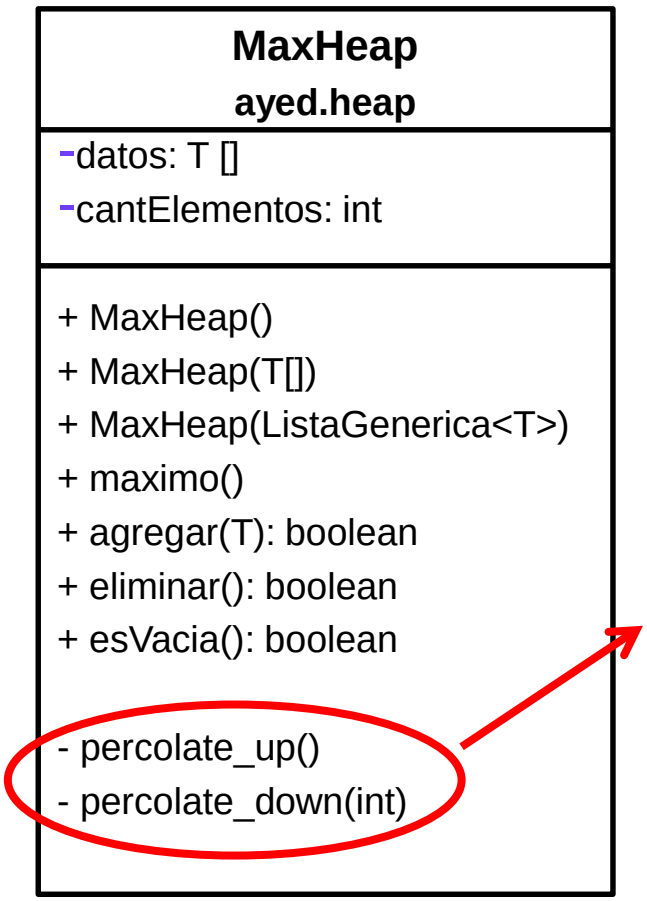



El hijo izquierdo está en la posición 2*i

El hijo derecho está en la posición 2*i + 1

El padre está en la posición [i/2]

# HEAP - MaxHeap
## Implementación en JAVA




Estos métodos acomodan los elementos insertados y/o eliminados de la MaxHeap . Se usan solamente en la clase MaxHeap.

percolate_up(): filtra hacia arriba. Se usa en la inserción.

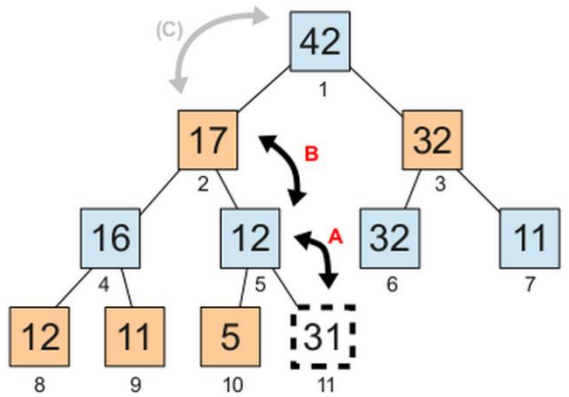
percolate_down(): filtra hacia abajo . Se usa en la eliminación.

# MaxHEAP

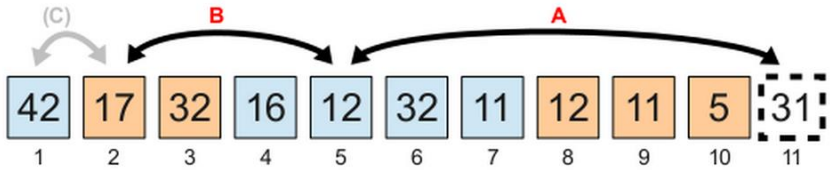
## Insertar un elemento

```
package ayed.heap;
import tp03.*;
public class MaxHeap<T extends Comparable<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();
  return true;
 }
 private void percolate_up() {
  // filtrar hacia arriba
  T temporal = datos[cantEltos];
  int indice = cantEltos;
  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 el arreglo. La propiedad de orden se puede romper. En ese caso, se hace 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 no se realiza porque 31 es menor que 42.



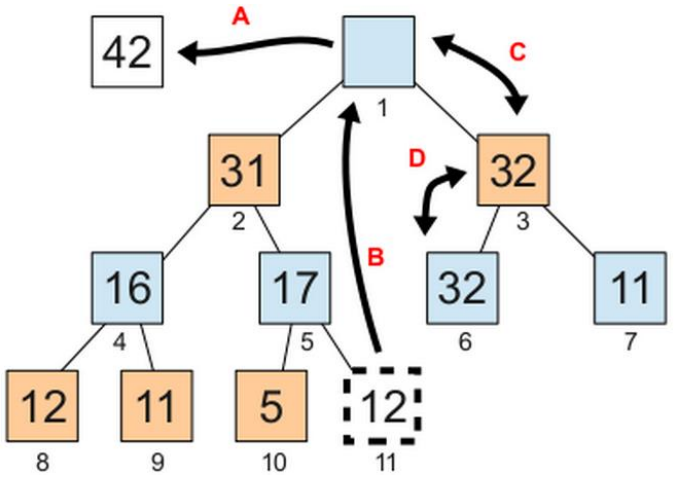

El filtrado hacia arriba restaura la propiedad de orden intercambiando el nuevo elemento a lo largo del camino hacia arriba desde el lugar de inserción.

# HEAP

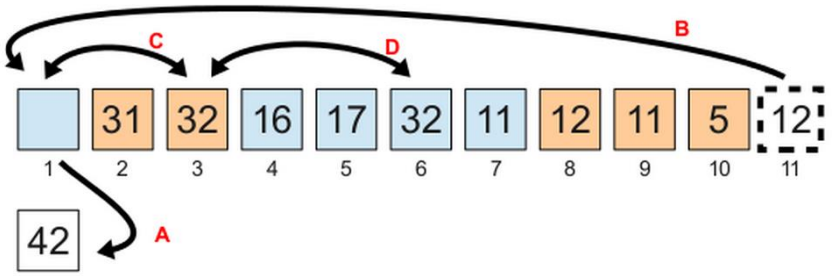
## Eliminar máximo

```
package heap;
import tp03.*;
public class Heap<T extends Comparable<T>> {
 private T[] datos = (T[]) new Comparable[100];
 private int cantEltos = 0;

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

public T maximo() {
 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 (el valor de cada nodo es menor o igual que el de su padre/madre ).




En el arreglo se ve así:



# MaxHEAP

## Filtrado hacia abajo: *percolate_down*

```
package ayed.heap;
import tp03.*;
public class MaxHeap<T extends Comparable<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 índice del hijo máximo (será nuevo candidato)
      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){ //hijo>que padre
           // FILTRO HACIA ABAJO
          datos[posicion] = datos[hijo_maximo];
           posicion = hijo_maximo;
      } else
            detener_percolate = true;
   }
   this.datos[posicion] = candidato;
 }
 //continúa
}
```

# HEAP - MaxHeap

## Algo sobre los constructores

```
package ayed.heap;
import tp03.*;
public class MaxHeap<T extends Comparable<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);
 }
 //continúa
}
```

**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];
```

Este constructor recibe una lista de elementos, la recorre, agrega de a un elemento a la vez y para elemento elemento agregado realiza un filtrado hacia arriba que garantiza la propiedad de orden.

Este constructor agregar todos los elementos del arreglo en la MaxHeap en el orden en que los recibe y luego restaura la propiedad de orden haciendo un filtrado hacia abajo intercambiando, cuando es necesario, el dato de cada nodo hacia abajo a lo largo del camino de los hijos máximos.