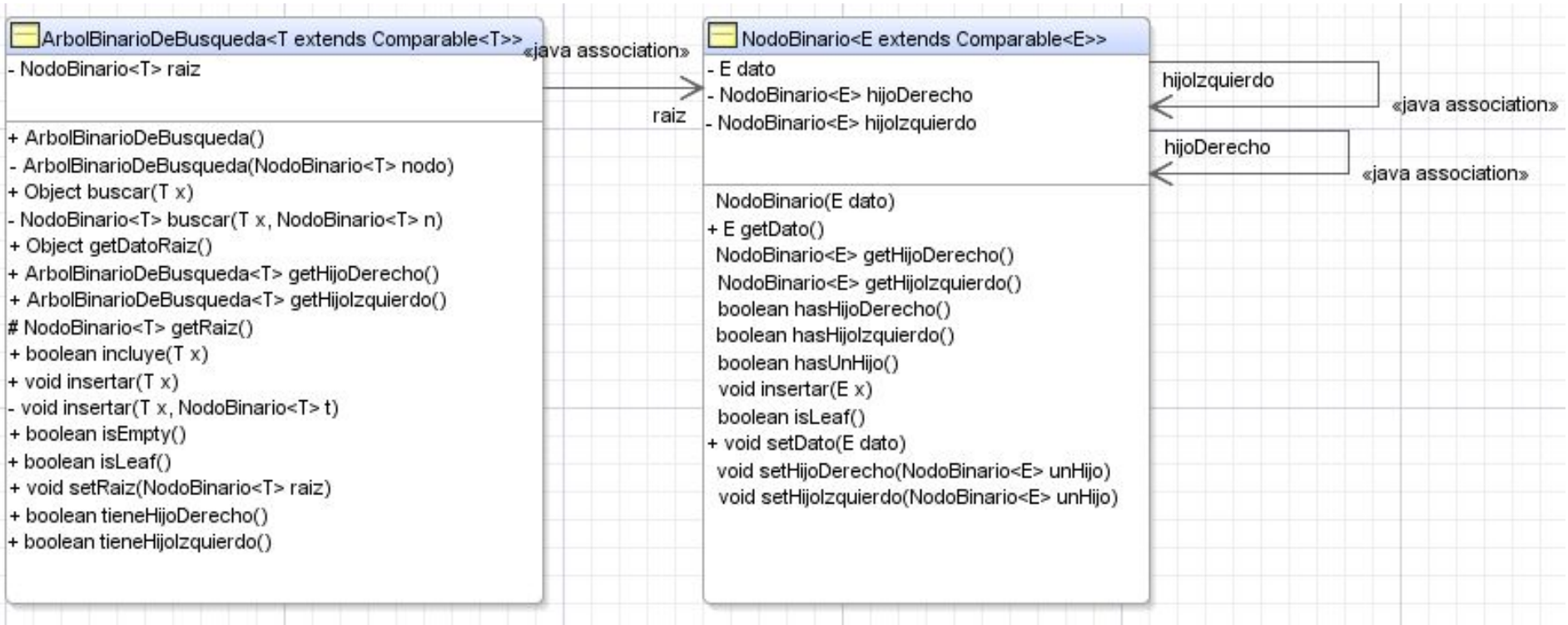
La interface Comparable<T> en Árboles Binarios de Búsqueda y HEAP

Sólo a nivel informativo. No se implementarán estas estructuras en la práctica

Arboles Binarios de Búsqueda
Estructura



<T extends Comparable<T>> un objeto de tipo **T** que implemente la interface **Comparable<T>**
<E extends Comparable<E>> un objeto de tipo **E** que implemente la interface **Comparable<E>**

Se utilizó E para mostrar que son definiciones independientes pero podria haberse usado el mismo nombre de variable en ambas definiciones

Árboles Binarios de Búsqueda- Estructura

```
package ayed2010;
public class ArbolBinarioDeBusqueda<T extends Comparable<T>>{

  private NodoBinario<T> raiz;

  public ArbolBinarioDeBusqueda() {
     this.setRaiz(null);
  }
  private ArbolBinarioDeBusqueda(NodoBinario<T> nodo) {
     this.setRaiz(nodo);
  }
  NodoBinario<T> getRaiz() {
     return this.raiz;
  }
  public void setRaiz(NodoBinario<T> raiz) {
     this.raiz = raiz;
  }
  public T getDatoRaiz() {
     return (this.getRaiz().getDato());
  }

  . . .
 }
```

```
package ayed2010;
public class NodoBinario<E extends Comparable<E>>{

 private E dato;
 private NodoBinario<E> hijoIzquierdo;
 private NodoBinario<E> hijoDerecho;
 NodoBinario(E dato) {
   this.dato = dato;
 }
 public E getDato() {
   return dato;
 }
 public void setDato(E dato) {
   this.dato = dato;
 }
 NodoBinario<E> getHijoIzquierdo() {
    return hijoIzquierdo;
 }
 NodoBinario<E> getHijoDerecho() {
    return hijoDerecho;
 }
 void setHijoIzquierdo(NodoBinario<E> unHijo) {
   hijoIzquierdo = unHijo;
 }
 void setHijoDerecho(NodoBinario<E> unHijo) {
   hijoDerecho = unHijo;
 }
  . . .
}
```

Árboles Binarios de Búsqueda

Ejemplos de instanciación y uso de árboles binarios de búsqueda con diferentes tipos:

```
ArbolBinarioDeBusqueda<Integer> abb = new ArbolBinarioDeBusqueda<Integer>();
abb.insertar(new Integer(2));
abb.insertar(6);
abb.insertar(1);
abb.insertar(13);
abb.insertar(5);

ArbolBinarioDeBusqueda<Persona> abb = new ArbolBinarioDeBusqueda<Persona>();
Persona p = new Persona("Paula","Gomez",16);
abb.insertar(p);
p = new Persona("Ana","Rios",6);
abb.insertar(p);
p = new Persona("Maria","Ferrer",55);
abb.insertar(p);
Alumno a = new Alumno("Luis", "Rios", 18);   // Alumno subclase de Persona
Abb.insertar(a);
```

Clases que implementan la interface Comparabe

Árboles Binarios de Búsqueda

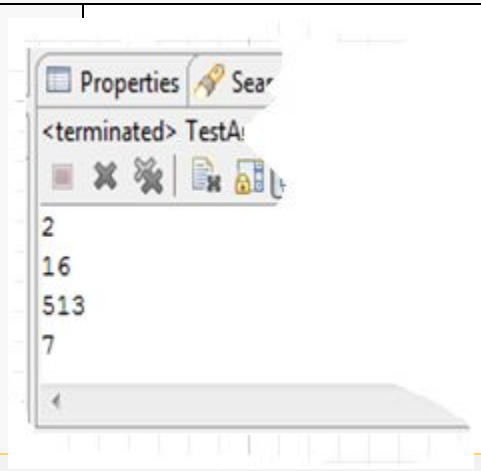
Inserción de un dato (Weiss)

```
package ayed2010;

public class ArbolBinarioDeBusqueda
   <T extends Comparable<T>> {
 private NodoBinario<T> raiz;
 . . .

 public void insertar(T x) {
    raiz = this.insertar(x, raiz);
 }
 private NodoBinario<T> insertar(T x, NodoBinario<T> t){
  if (t == null) {
    t = new NodoBinario<T>(x);
  }
  else
  if (x.compareTo(t.getDato()) < 0)
     t.setHijoIzquierdo(this.insertar(x, t.getHijoIzquierdo()));
    else
     if (x.compareTo(t.getDato()) > 0)
     t.setHijoDerecho(this.insertar(x, t.getHijoDerecho()));
  return t;
 }
}
```

```
package ayed2010;
public class NodoBinario<E> extends Comparable<E>> {
 private E dato;
 private NodoBinario<E> hijoIzquierdo;
 private NodoBinario<E> hijoDerecho;
 . . .
}
```




```
ArbolBinarioDeBusqueda<Integer> abb =
   new ArbolBinarioDeBusqueda<Integer>();
abb.insertar(2);
abb.insertar(6);
abb.insertar(1);
abb.insertar(13);
abb.insertar(5);
abb.insertar(7);
abb.insertar(5);
abb.mostrar();
```

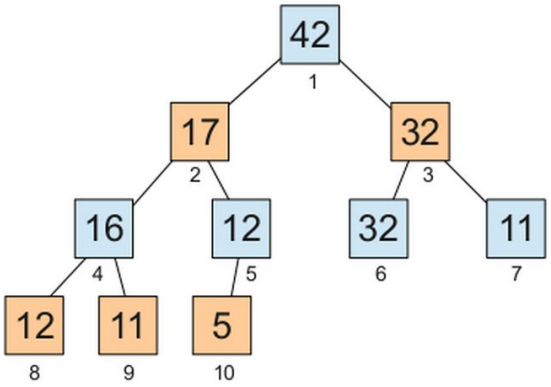
Colas de Prioridad

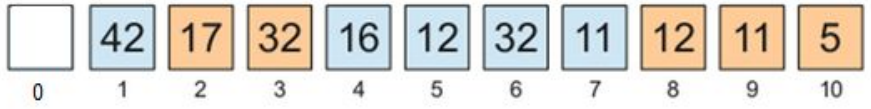
En las *colas de prioridad*, el *orden lógico* de sus elementos está determinado por la prioridad de los mismos. Los elementos de mayor prioridad están en el frente de la cola y los de menor prioridad están al final. De esta manera, cuando se encola un elemento, puede suceder que éste se mueva hasta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

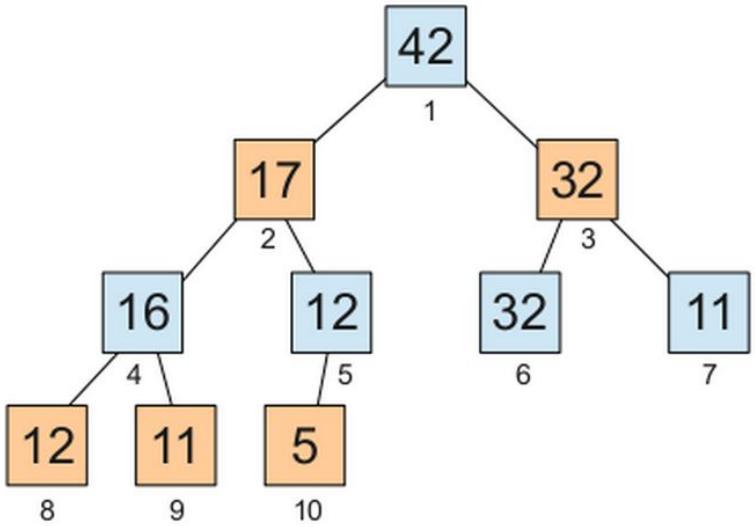
- propiedad de orden: El elemento máximo (en MaxHeap) está almacenado en la raíz y el dato almacenado en cada nodo es mayor o igual al de sus hijos (en MinHeap es inverso).



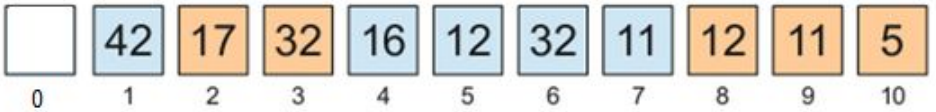


Colas de Prioridad

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



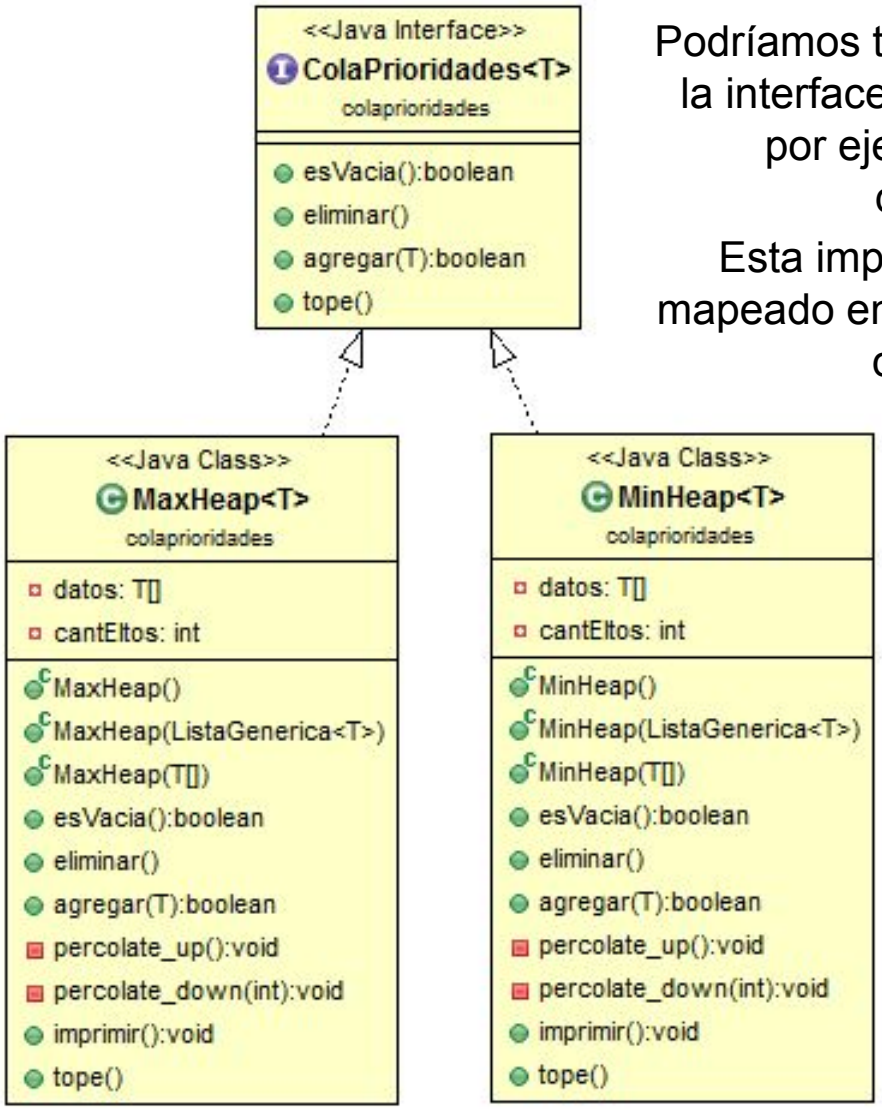
El hijo izquierdo está en la posición 2*i
El hijo derecho está en la posición 2*i + 1
El padre está en la posición [i/2]

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>>
                    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);
 }
 . . .
}
```

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

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()`**.