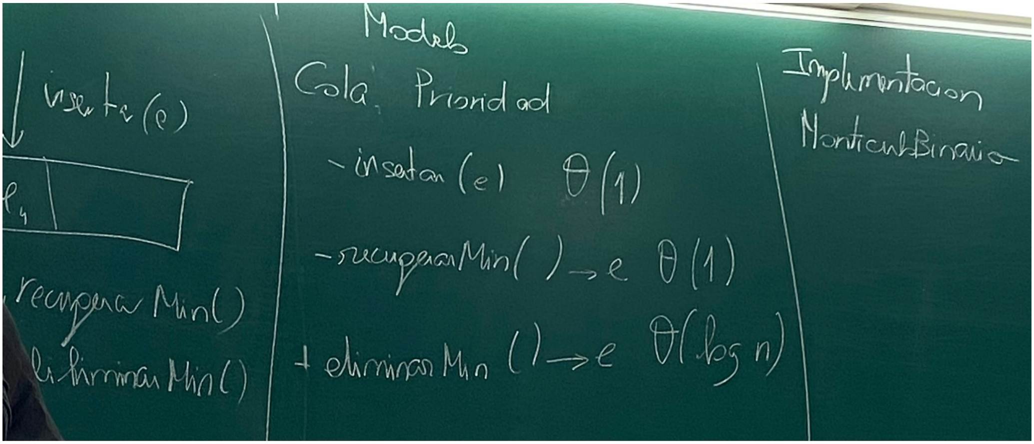

esVacia() $\Theta(1)$

Montículo binario

-Árbol binario(2 hijos máximo)
-Completo(niveles completo excepto último)
-orden padre<hijos

n nodos
log(n) altura

representación vector elArray talla n+1

- **Problema 4. Selección de los K mejores**

Si la **Cola de Prioridad** se implementara como una **ListaConPI** ...

En **O(N · K)**, y usando memoria adicional, los K mejores de N están en la Lista con PI, ordenados ascendentemente.

O(N · K) **NO** es necesariamente mejor que O(N · log N)

Si la **Cola de Prioridad** se implementa como un **MonticuloBinario** ...

En **O(N · log K)**, los K mejores de N están, **SIN ORDENAR**, en el Montículo Binario.

O(N · log K) **SÍ** es mejor que O(N · log N) cuando K << N

Coste Promedio / Representación	insertar	recuperarMin	eliminarMin
Lineal	Θ(1)	Θ(n)	Θ(n)
Lineal Ordenada	**Θ(n)**	**Θ(1)**	**Θ(1)**
¿Montículo Binario?	**Θ(1)**	**Θ(1)**	**Θ(log n)**

AB Equilibrado & Parcialmente Ordenado

- **elArray[1]** representa a su Nodo Raíz
- si **elArray[i]** representa a su i-ésimo Nodo (Por Niveles)
  - Su Hijo Izquierdo es **elArray[2i]**, si 2i ≤ talla
  - Su Hijo Derecho es **elArray[2i+1]**, si 2i + 1 ≤ talla
  - Su Padre es **elArray[i/2]**, excepto para i = 1

¿Por qué no usar la posición 0 del array?

Caminos ascendentes.
min-> raíz elArray[i]

- **Operaciones *kernel***: las operaciones de una Cola de Prioridad
  - Insertar un nuevo elemento *e* en un *Heap* (*add*): insertar(e)
  - Comprobar si un *Heap* está vacío (*isEmpty*): esVacia()
  - Devolver, **SIN** eliminar, el mínimo de un *Heap* (*peek*): recuperarMin()
  - Devolver **Y** eliminar el mínimo de un *Heap* (*poll*): eliminarMin()
- **Especificación y Esquema algorítmico de las operaciones modificadoras:**

  PreCondición: el árbol es AB Completo y cumple propiedad de orden del *Heap*

  1. Realizar la operación sobre un AB Completo **Y** comprobar que el Árbol resultante es también un AB Completo.
  2. Comprobar si el Árbol resultante cumple la propiedad de orden del *Heap*. Si NO lo cumple, restaurarla mediante las operaciones pertinentes.

  PostCondición: el árbol es AB Completo y cumple propiedad de orden del *Heap*
- **Coste promedio estimado:** para una gran mayoría de las operaciones, al menos las que requieren el acceso a uno de sus datos, varía entre O(1) y O(log talla), **por lo que en cualquier caso es sublineal**.

```java
/** insertar e en un Heap */
public void insertar(E e) {
    if (talla == elArray.length - 1) duplicarArray();
    // PASO 1: Buscar la posición de inserción ordenada de e
    // (a) Preservar la Propiedad Estructural
    int posIns = ++talla;
    // (b) Preservar la Propiedad de Orden
    posIns = reflotar(e, posIns);
    // PASO 2: Insertar e en su posición de inserción ordenada
    elArray[posIns] = e;
}
```

```java
protected int reflotar(E e, int posIns) {
    while (posIns > 1 && e.compareTo(elArray[posIns / 2]) < 0) {
        elArray[posIns] = elArray[posIns / 2];
        posIns = posIns / 2;
    }
    return posIns;
}
```

3. La clase Java **MonticuloBinario**

Método eliminarMin(): código

```java
/** recuperar y eliminar el mínimo de un Heap */
public E eliminarMin() {
    E elMinimo = elArray[1];
    // PASO 1: Borrar el mínimo del Heap
    // (a) Preservar la Propiedad Estructural:
    //     borrar "Por Niveles" el mínimo
    elArray[1] = elArray[talla--];
    // (b) Preservar la Propiedad de Orden:
    //     buscar posición de inserción ordenada de elArray[1]
    // PASO 2: Insertar elArray[1] en su posición ordenada
    hundir(1);
    return elMinimo;
}
```

```java
protected void hundir(int pos) {
    int posActual = pos;
    E aHundir = elArray[posActual];
    int hijo = posActual * 2;
    boolean esHeap = false;
    while (hijo <= talla && !esHeap) {
        if (hijo < talla
                && elArray[hijo + 1].compareTo(elArray[hijo]) < 0) {
            hijo++;
        }
        if (elArray[hijo].compareTo(aHundir) < 0) {
            elArray[posActual] = elArray[hijo];
            posActual = hijo;
            hijo = posActual * 2;
        }
        else { esHeap = true; }
    }
    elArray[posActual] = aHundir;
```

(a)Talla del problema: **n** = talla (número de elementos), o **h** = $\lfloor \log_2 talla \rfloor$, **la altura** del *Heap* del que se borra el mínimo (equivalentemente, la longitud de su camino más largo).

(b) Como un nodo de un AB Completo se borra en $\Theta(1)$, *eliminarMin* y *hundir(1)* tienen el mismo coste. Por tanto, basta con analizar el coste de *hundir(1)*:

- Caso Mejor: El dato en raíz es el nuevo mínimo del *Heap*, por lo que NO se hunde

$$T^{M}_{hundir(1)/eliminarMin}(n) \in \Theta(1) \Rightarrow T_{hundir(1)/eliminarMin}(n) \in \Omega(1)$$

- Caso Peor: El dato en raíz es un nuevo máximo del *Heap*, por lo que hay que hundir *posActual* hasta una hoja, i.e. H = $\lfloor \log_2 talla \rfloor$ veces

$$T^{P}_{hundir(1)/eliminarMin}(n) \in \Theta(h=\log_2 n) \Rightarrow T_{hundir(1)/eliminarMin}(n) \in O(\log_2 n)$$

Caso promedio: $T^{\mu}_{eliminarMin}(n) \in \Theta(h=\log_2 n)$

3. La clase Java `MonticuloBinario`

Otros métodos: construir un Heap con N datos

<u>Solución 1</u> (directa pero costosa!!): insertar en un Heap vacío, uno por uno, los N datos:

```
public class Solucion1 {
   public static void main(String[] args) {
      int N = … ;
      Random r = new Random();
      MonticuloBinario<Integer> h
         = new MonticuloBinario<Integer>();
      for (int i = 1; i <= N; i++) {
         Integer e = new Integer((r.nextInt(10) + 1) * i);
         h.insertar(e);
      }
   }
   ...
}
```

$T_{Solucion1}(n=N) \in \Omega(n)$

$T_{Solucion1}(n=N) \in O(n \cdot \log n)$

<u>Solución 2</u> (óptima!!): introducir en **elArray** los N datos, en el orden en el que aparezcan, y después "arreglar"-lo.

```
public class Solucion2 {
   public static void main(String[] args) {
      int N = … ;
      Random r = new Random();
      MonticuloBinario<Integer> h
         = new MonticuloBinario<Integer>();
      for (int i = 1; i <= N; i++) {
         Integer e = new Integer((r.nextInt(10) + 1) * i);
         h.introducir(e);
      }
      ¿h.arreglar();?
   }
```

$T_{for}(n=N) \in \Theta(n)$

```
public void introducir(E e) {
   if (talla == elArray.length - 1) duplicarArray();
   elArray[++talla] = e;
}
```

3. La clase Java MonticuloBinario

Método arreglar(): coste

(a) Talla del problema: **n** = talla (número de elementos), o **h** = $\lfloor \log_2 talla \rfloor$, **la altura** del AB Completo a "arreglar"

(b) Relación de Recurrencia para el caso general, i.e. para un AB que no es hoja (n>0): $T_{arreglar}(h) = 2 * T_{arreglar}(h - 1) + T_{hundir}(x)$

- Caso Mejor: Caso Mejor de *hundir*

$$T^M_{arreglar}(h) = 2 * T^M_{arreglar}(h - 1) + k * 1$$

$$T^M_{arreglar}(h = \log_2 n) \in \Theta(2^h = 2^{\log_2 n} = n) \Rightarrow T_{arreglar}(n) \in \Omega(n)$$

- Caso Peor: Caso Peor de *hundir*

$$T^P_{arreglar}(h) = 2 * T^P_{arreglar}(h - 1) + k * h$$

$$T^P_{arreglar}(h = \log_2 n) \in \Theta(2^h = 2^{\log_2 n} = n) \Rightarrow T_{arreglar}(n) \in O(n)$$

$$T_{arreglar}(h = \log_2 n) \in \Theta(n)$$

Hojas empiezan en talla/2 + 1