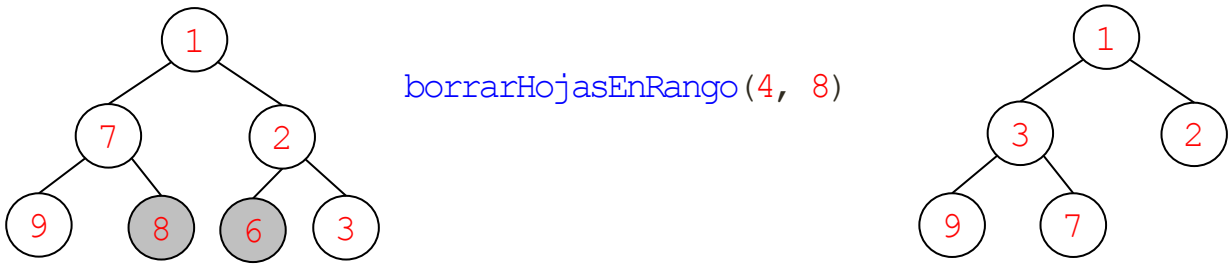
Ejercicios de clase
TEMA 5 (B)– Cola de Prioridad y Montículo Binario

Ejercicio 9

Diseña en la clase `MonticuloBinario` un método `borrarHojasEnRango` que, de la manera más eficiente posible, borre aquellas Hojas de un Heap Minimal que estén dentro del rango genérico y no vacío `[x,y]`. Ten en cuenta que el orden de los Nodos del Heap resultante no es importante.

SOLUCIÓN:

El método debe borrar las hojas que caigan dentro de un determinado rango. Por ejemplo:



Haremos una solución en dos pasos:

- En primer lugar, se borran todas aquellas hojas que caigan dentro del rango. Para ello haremos un recorrido secuencial desde la primera hoja (que ocupa la posición talla/2 +1 del array) hasta la última. Si la hoja está en el rango, se borra de la forma habitual: su contenido se "machaca" con el contenido del último elemento y se decrementa la talla. Antes de pasar a la hoja que está en la posición i+1, hay que comprobar que el elemento que ahora ocupa la posición no hay que borrarlo.
- Una vez se han borrado todas las hojas que caen el rango, se realiza una reconstrucción del heap completo.

La operación viene dominada por el coste de construir el heap, por tanto, coste lineal con el número del elementos.

```
public void borrarHojasEnRango(E x, E y) {
	// PASO 1: Recorrer SOLO las Hojas del Heap, borrando "por niveles"
	//         aquellas que estén en [x, y]
	int i = talla / 2 + 1; // primera Hoja
	while (i <= talla) {   // Recorrido de las Hojas, hasta talla
		if (elArray[i].compareTo(x) >= 0 && elArray[i].compareTo(y) <= 0) {
			elArray[i] = elArray[talla--]; // Borrado "PEREZOSO"
		}
		else { i++; }
	}
	// PASO 2: ¿Sigue siendo Heap tras borrar las Hojas en [x,y]?
	//         Si no lo es, "arreglar" elArray
	this.arreglar();
}
```

Ejercicio 10

Diseña en la clase `MonticuloBinario` un método `igualesAlMinimo` que, de la manera más eficiente posible, devuelve el número de elementos de un Heap Minimal iguales al mínimo (este incluido).

SOLUCIÓN:

El mínimo en un minHeap es el primer elemento. En la solcuión se hace un recorrido de todos los nodos del heap siempre que el nodo padre sea igual al mínimo.

```
public int igualesAlMinimo() {
	int res = 0;
	if (talla != 0) { res = igualesAlMinimo(1); }
	return res;
}
protected int igualesAlMinimo(int i) {
	int res = 0; // Caso base IMPLÍCITO Hoja
	if (elArray[i].compareTo(elArray[1]) == 0) {
		res++;
		if (2*i <= talla)   { res += igualesAlMinimo(2*i); }
		if (2*i+1 <= talla) { res += igualesAlMinimo(2*i+1); }
	}
	return res;
}
```

Ejercicio 11

Se pide diseñar un método genérico, estático e iterativo `fusion` que dadas `qP1` y `qP2`, dos Colas de Prioridad implementadas mediante Min-Heaps, devuelva una Lista Con PI que contiene los datos de `qP1` y `qP2` ordenados ascendentemente; dicho método no puede hacer uso de ninguna estructura de datos auxiliar para calcular su resultado y, además, `qP1` y `qP2` deben estar vacías al concluir su ejecución. El coste temporal del método diseñado deberá ser Θ(x·logx), siendo x la suma de las tallas de `qP1` y `qP2` o, equivalentemente, la talla de la Lista resultado.

SOLUCIÓN:

En estos problemas tened en cuenta que sobre el Heap sólo se puede obtener el Mínimo y eliminar el Mínimo (además de insertar y ver si está vacío). Al ser sólo 4 operaciones no hay muchas opciones complicadas para resolver los problemas. Aquí la solución es simplemente ir comparando el Mínimo de cada heap y extraer el menor de ellos metiéndolo en la LPI.

Cuando se vacía uno de los Heaps hay que meter en la lista el resto de elementos que quedan en el otro heap. Fijaos que los dos "while" que hay al final no están puestos como una alternativa ("si uno se ha vaciado entonces meter el resto del otro"). Está escrito como una secuencia: Añadir lo que resta de qP1 y luego añadir lo que resta de qP2. Lógicamente sólo una de esas dos instrucciones llega a ejecutarse (es una opción tan buena como preguntar primero cuál es la que aún tiene elementos).

El coste es O(n.log n) siendo n el conjunto total de elementos, porque para cada uno de esos n elementos hay que hacer la operación EliminarMin que tiene un coste O(log n).

```
public static <E extends Comparable<E>> ListaConPI<E> fusion(ColaPrioridad<E> qP1, ColaPrioridad<E> qP2) {
	ListaConPI<E> res = new LEGListaConPI<E>();
	while (!qP1.esVacia() && !qP2.esVacia()) {
		if ((qP1.recuperarMin()).compareTo(qP2.recuperarMin()) < 0)
			res.insertar(qP1.eliminarMin());
		else res.insertar(qP2.eliminarMin());
	}
	while (!qP1.esVacia()) res.insertar(qP1.eliminarMin());
	while (!qP2.esVacia()) res.insertar(qP2.eliminarMin());
	return res;
}
```

Ejercicio 12

Se pide diseñar un método genérico y estático `sucesor` que devuelva el sucesor de un dato `k` en una Cola de Prioridad `cP`, implementada con un Montículo Binario Minimal; si el sucesor de `k` no está en `cP` el método devuelve `null` para advertirlo. Además, dicho método tiene que usar una Pila como estructura de datos auxiliar, pues al terminar su ejecución `cP` debe contener los mismos elementos que tenía antes de invocarlo y se quiere que su coste sea el mejor posible ($\Omega(1)$ y $O(x \cdot \log x)$, siendo x la talla de `cP`); queda a criterio del programador que dicha Pila sea la Pila de la Recursión o una implementada mediante un `ArrayPila`.

SOLUCIÓN:

En este problema se pide encontrar el sucesor de un valor k, es decir, el primer elemento del heap cuyo valor sea superior a k. De nuevo nos encontramos con la ventaja o el inconveniente de que sólo disponemos de cuatro métodos (insertar, recuperarMin, eliminarMin, esVacio). Fijaos que la única manera de ir accediendo a los elementos del Heap es a través de la secuencia de operaciones recuperarMin y después eliminarMin (para poder acceder al siguiente). Por eso hace falta almacenar los elementos que se van extrayendo.

La primera solución es un algoritmo recursivo parecido a alguno que hicimos en el tema 1. Se va sacando el mínimo del Heap hasta que es mayor que el valor k (ese es el sucesor). Fijaos que la llamada recursiva almacena en una variable local llamada min el valor del mínimo en esa "copia" del algoritmo recursivo. Por tanto cuando se ha completado la llamada recursiva ese min lo volvemos a insertar en el Heap, de manera que se reconstruye el Heap (ojo, puede quedar de forma diferente, aunque tenga los mismos valores).

La segunda solución es un algoritmo iterativo en el que se van extrayendo los mínimos sucesivos hasta encontrar el elemento buscado. Los valores que se van extrayendo se almacenan en una pila, y luego los valores de la pila se vuelven a insertar en el Heap. El coste en el mejor caso es $\Omega(1)$ ya que en la comparación podría terminarse, y en el peor caso es O(n.log n) cuando el número buscado es mayor que todos y hay que extraer los n elementos y volverlos a insertar.

Si se utiliza la Pila de la Recursión:

```
public static <E extends Comparable <E>> E sucesor(ColaPrioridad<E> cP, E k) {
	if (cP.esVacia()) return null;
	E min = cP.recuperarMin();
	if (min.compareTo(k) > 0) res = min;
	else
		min = cP.eliminarMin();
		res = succesor(cP, k);
		cP.insertar(min);
	}
	return res;
}
```

Si se utiliza una Pila auxiliar:

```
public static <E extends Comparable <E>> E sucesor(ColaPrioridad<E> cP, E k) {
	E res = null;
	Pila<E> aux = new ArrayPila<E>();
	while (!cP.esVacia() && cP.recuperarMin().compareTo(k) <= 0) {
		E min = cP.eliminarMin();
		aux.apilar(min);
	}
	if (!cP.esVacia()) res = cP.recuperarMin();
	while (!aux.esVacia()) cP.insertar(aux.desapilar()); return res;
}
```