TEMA 8: COLAS CON PRIORIDAD Y MONTÍCULOS (HEAPS)

En una **cola FIFO (First In First Out)**, el primer elemento en entrar es el primero en salir; mientras que en las **colas con prioridad** el orden de salida viene determinado por la **prioridad** del elemento. Se puede ver una cola como una estructura de datos en la que los elementos se almacenan en orden de prioridad. Las entradas de una cola con prioridad tienen:
(1) Una **clave** que indica la prioridad del elemento
(2) Un **valor** que indica el elemento a insertar
El conjunto de ambos (*clave-valor*) se denomina **entrada** (**entry**)

En el interfaz `Entry<K,V>` tenemos:
- `K` es la clave (**key**) de la entrada que vamos a insertar
- `V` es el valor (**value**) que vamos a insertar

Por convención, la clave establece la prioridad inversamente (cuanto menor es la clave mayor es la prioridad). También se conocen como **min-max queue** porque al desencolar se devuelve el elemento con la menor clave. Se utilizará el orden total entre las claves. Los objetos que se usen para la clave deben ser `Comparable` o disponer de un `Comparator`. Nos podemos encontrar dos o más entradas con la misma clave pero distinto valor, con el mismo valor pero distintas claves o con la misma clave y los mismos valores.

```
public interface PriorityQueue<K,V> extends Iterable<Entry<K,V>> {
  int size();
  boolean isEmpty();
  Entry<K,V> enqueue(K key, V value) throws InvalidKeyException;
  Entry<K,V> first() throws EmptyPriorityQueueException ;
  Entry<K,V> dequeue() throws EmptyPriorityQueueException ;
  void remove(Entry<K,V> entry ) throws InvalidKeyException ;
  void replaceKey(Entry<K,V> entry, K newKey) throws InvalidKeyException;
  void replaceValue(Entry<K,V> entry, V newValue) throws InvalidKeyException;
}
```

- `first` y `dequeue` devuelven objetos que implementan el interfaz `Entry<K,V>`
- `enqueue` recibe por separado una clave key y un valor value
- `first` es un método observador para consultar el elemento con mayor prioridad (con la clave con menor valor) y lo borra de la cola
- `EmptyPriorityQueueException` se lanza cuando se intenta acceder a la entrada de clave mínima en una cola vacía
- `InvalidKeyException` se lanza cuando la clave es **null** o no tiene definido un orden para los elementos de su clase
- `remove`, `replaceKey` y `replaceValue` requieren un `Entry<K,V>` como argumento, **¡pero no vale cualquiera! ¡¡OJO!!** Deben haber sido devueltos por uno de los métodos `first, enqueue` o usando el iterador sobre la cola. Un `Entry<K,V>` devuelto por estos métodos contiene una referencia (`position`) dentro de la estructura de datos implementando la cola

Complejidad de las distintas implementaciones de una **cola con prioridad**:

PositionList Desordenada	**PositionList Desordenada con caché**	**PositionList ordenada (SortedListPriorityQueue)**
`enqueue` → O(1)	`enqueue` → O(1)	`enqueue` → O(n)
`dequeue` → O(n)	`dequeue` → O(n)	`dequeue` → O(1)
`first` → O(n)	`first` → O(1)	`first` → O(1)

Montículos:
Un **montículo** (`HeapPriorityQueue`) es una implementación de **colas con prioridad** que satisface las siguientes complejidades:

`enqueue` → O(log(n)) `first` → O(1) `replaceKey` → O(log(n))

`dequeue` → O(log(n)) `remove` → O(log(n)) `replaceValue` → O(1)

En una implementación de una **cola con prioridad** mediante una lista usamos una lista ordenada usando el orden total de claves; mientras que en una implementación con **montículos**, se utiliza un *árbol binario (casi)completo* para particionar la entrada (reduciendo de esta forma las búsquedas a complejidad O(log(n))), donde todo nodo distinto de la raíz tiene una entrada mayor o igual que la entrada almacenada en el nodo padre.

Cumplen la **propiedad heap-order**. Es decir, todos los caminos de la raíz a las hojas están ordenados ascendentemente y la entrada de menor clave (la más prioritaria) está almacenada en el nodo raíz. Por ello, es necesario un `Comparator` o bien que las claves sean `Comparable`.

Operación enqueue:
La inserción de un nuevo nodo se hace incluyendo el nuevo nodo como el último nodo del árbol y reajustando los nodos en caso de necesidad, es decir, si se viola la prioridad heap-order.

- Se comprueba la heap-order priority entre el nuevo nodo y su padre
  - Si se cumple, hemos acabado
  - Si no se cumple, entonces se intercambian las entradas entre el nodo nuevo y el padre
- Se repite la operación con el nodo intercambiado y el padre correspondiente hasta que se cumpla la heap-order priority o hasta que se llegue a la raíz
  A esto se le conoce como **up-heap bubbling**

Operación dequeue:
La entrada de menor clave siempre es la raíz, pero no se puede borrar directamente. Por lo que intercambiamos la raíz con el último nodo del árbol y lo borramos reajustando los nodos en caso de necesidad, es decir, si se viola la prioridad heap-order.

- Se comprueba la heap-order priority entre la raíz y sus hijos
  - Si se cumple, hemos acabado
  - Si no se cumple, entonces se intercambian las entradas entre la raíz y el nodo hijo de menor clave
- Se repite la operación con el nodo intercambiado y sus hijos hasta que se cumpla la heap-order priority o hasta llegar a una hoja
  A esto se le conoce como **down-heap bubbling**