COLAS CON PRIORIDAD

Jesús Pérez Melero

Índice

AVISO: El código se puede optimizar mucho más. Está escrito así para que sea más comprensible.

Implementación mediante cursores

Interfaz Entry<E,E>

Durante toda la implementación se tomará como ejemplo un matadero de cerdos, donde queremos matarlos a todos empezando por los más jóvenes. Así que, de momento, lo único que nos importa es la **edad** del cerdo, pero además como somos buenos porqueros le vamos a poner un **nombre** a cada cerdito.

Lo primero que hay que declarar para implementar una cola es de qué cosas va a ser la cola. La cola será de objetos *Entry <K,V>* donde **K representa la clave**, la prioridad, y **V el valor de esa pareja**.

Para declarar el Entry que usaremos en nuestra implementación, podemos hacerlo así:

```
public class EntryCerdo<K,V> implements Entry <Integer,String>{

	private Integer k;
	private String v;

	public EntryCerdo (Integer edad, String nombre) {

		this.k = edad;
		this.v = nombre;
	}
```

Donde dejamos claro que nuestra *Entry EntryCerdo<K,V>* **implementa el interfaz** *Entry<Integer,String>* del paquete net.datastructures (previa importación).

¿Por qué es *Entry<Integer,String>*?

Porque nosotros queremos representar cerdos, como hemos dicho antes, mediante su edad y su nombre, pero **lo único que nos interesa es la edad**, por eso en el campo de K hemos puesto *Integer*, porque ahí irá una edad, y en el otro campo un *String* porque ahí irá el nombre del cerdo.

Al ser Entry un interfaz, nos obliga a implementar varios métodos, los cuales son:

```
	public Integer getKey() { return this.k; }

	public String getValue() { return this.v; }

	public String toString () {

		return "[ (0_0)~ : " + v + " " + k + " anios ]";
	}
```

GetKey() y *getValue()* son getters para acceder a los campos K y V de nuestra *Entry*.

El método *toString()* simplemente representa objetos de tipo EntryCerdo<K,V>.

Además es útil definir un método equals que se usará más adelante:

```
public boolean equals (EntryCerdo<K,V> o) {

		return this.getKey().equals(o.getKey()) &&
				this.getValue().equals(o.getValue());
	}
```

Interfaz Comparator <E>

Como hemos dicho, queremos implementar una cola **con prioridad**, es decir, habrá objetos que tengan más prioridad que otros. Es por ello que es necesario definir **un orden total** que sea capaz de cubrir cualquier situación, de forma que todos los elementos que haya o pueda haber en la cola estén regidos por ese orden.

Esto en Java se consigue implementando, por ejemplo, el interfaz *Comparator*, dentro del paquete java.Util. En nuestro caso, queremos que los cerdos más jóvenes mueran primero, luego necesitamos implementar un comparador que resuma esto:

Si i = 0 → Tienen la misma edad

Si i > 0 → El primero es más viejo

Si i < 0 → El primero es más joven

En este caso es muy fácil de implementar ya que esto se puede identificar **como una resta entre las edades de los dos cerdos**. Si ambas edades son iguales, su resta será 0, si la del primero es mayor que la del segundo, el resultado será mayor que 0 implicando que el primero es más viejo e igual para el último caso.

Su aspecto Java sería este:

```
public class AgeComparator <K> implements Comparator <Integer> {


	public int compare(Integer age1, Integer age2) {

		return age1 - age2;
	}

}
```

Implementar el método *compare* **es obligatorio** ya que es un método del interfaz Comparator<E>.

Una vez más la E se transforma en *Integer*, porque **lo que queremos comparar son edades**, es decir, números.

Implementación de la cola con prioridad

Una vez tenemos todo lo anteriormente dicho, podemos proceder a implementar la cola con prioridad (en adelante PQ). Las PQ's implementan el interfaz *PriorityQueue<E,E>.*

Cuando creemos nuestra PQ parece obvio que **tendremos que pasarle información sobre cómo comparar los elementos** para establecer la prioridad. De esto se encargará el Comparador que habremos definido antes. Por lo tanto, uno de los **atributos** de nuestra PQ será un objeto que implemente el interfaz *Comparator*.

Nuestro segundo atributo puede no ser siempre igual, y es la **estructura de datos** donde vamos a ir guardando lo que entre a la cola, es decir, objetos *EntryCerdo<K,V>* en este caso.

Podríamos implementar la cola con un *array*, si ésta fuera finita, pero como en este caso no lo es, es preferible usar una lista, concretamente una **PositionList**, pero se podría usar cualquier otra estructura de datos, ya que **el orden en que llegan los objetos a la cola nos da igual a la hora de elegir al próximo que va a salir de ella**.

De esta forma podemos declarar ya las primeras líneas de nuestra PQ:

```
public class PositionListPQ implements PriorityQueue <Integer,String>{

	/** Lista de objetos entry */
	private PositionList <EntryCerdo<Integer,String>> list;
	/** Orden total definido */
	private AgeComparator <Integer> comp;
```

Que son básicamente los dos atriutos anteriormente descritos, el comparador y la lista.

Lo siguiente que debemos implementar es el constructor de la lista:

```
/**
	 * Constructor
	 * @param comp - orden total
	 */
	public PositionListPQ (AgeComparator <Integer> comp) {

		this.list = new NodePositionList <EntryCerdo<Integer,String>> ();
		this.comp = comp;
	}
```

Que se encarga de crear una *PositionList* vacía, que albergará objetos *EntryCerdo<Integer,String>.* El argumento que le pasaremos será el comparador, el orden total que regirá nuestra PQ.

El interfaz *PriorityQueue* del paquete net.datastructures nos obliga a implementar varios métodos:

- **Insert**: Para insertar un objeto en la PQ.
- **isEmpty**: Para comprobar si la PQ está vacía.
- **min**: Para ver qué elemento es el siguiente que saldrá de la cola.
- **removeMin**: Para ver qué elemento es el siguiente que saldrá de la cola, y sacarlo de la cola.
- **Size**: Ver el tamaño de nuestra PQ.

Método insert

Este método se encarga de insertar un objeto en la PQ. Puede lanzar la excepción *InvalidKeyException* cuando la clave del objeto que le pasemos sea inválida, por cualquier motivo.

Si no se dan situaciones anómalas, habrá que añadir a la lista del atributo un nuevo objeto, con los datos que nos han pasado.

```
/**
	* insert : inserta en la cola una nueva entrada formada por una dupla de objetos
entry
	*/

public EntryCerdo<Integer,String> insert(Integer edad, String nombre) throws
InvalidKeyException {

		if (edad == null || !(edad instanceof Integer) || edad < 0)
			throw new InvalidKeyException ("Clave introducida invalida");

		//Se crea la entry y se añade al atributo lista
		EntryCerdo<Integer,String> nueva = new EntryCerdo<Integer,String>
(edad,nombre);
		list.addLast(nueva);

		return nueva;
	}
```

Este método además devuelve al que lo invoca una copia de lo que se ha introducido en la PQ. En nuestro caso los objetos que recibiremos serán una edad y un nombre, y este método se encargara de crear el objeto *EntryCerdo<Integer,String>* que se depositará en la PQ.

Método isEmpty

Este método se encarga de decir si nuestra PQ está vacía. Resulta obvio ver que la PQ estará vacia si y sólo si la lista del atributo está vacía, por tanto, nuestro método será:

```
/**
	* isEmpty : Indica si la cola esta o no vacia
	*/
	public boolean isEmpty() {

		return list.isEmpty();
	}
```

Método size

Este método se encarga de mostrarnos el tamaño de nuestra PQ. Análogamente al método anterior, recurriremos a la lista del atributo:

```
public int size() {

		return list.size();
	}
```

Método min

Este método se encarga de mostrar cuál es el siguiente elemento que saldrá de la cola. Este método puede lanzar la excepción *EmptyPriorityQueueException* si la lista atributo es null o está vacía.

En otro caso, debemos ir recorriendo la lista atributo, **utilizando el criterio del comparador** para seleccionar al objeto que tendrá la prioridad, en este caso, el cerdo más joven.

```
/**
	 * min : Muestra el siguiente elemento que va a salir de la cola
	 */
	public EntryCerdo<Integer,String> min() throws EmptyPriorityQueueException {

		if(list == null || list.isEmpty())
			throw new EmptyPriorityQueueException ("La cola esta vacia");

		Position <EntryCerdo<Integer,String>>minum = list.first();
		int min = minum.element().getKey();
		Position <EntryCerdo<Integer,String>> cursor = list.next(list.first());

		while (cursor != null)
		{
			int compar = comp.compare(min, cursor.element().getKey());
			if (compar > 0)
			{
				minum = cursor;
				min = minum.element().getKey();
				cursor = list.next(cursor);
			}
			else
				cursor = list.next(cursor);
		}
		return minum.element();
	}
```

Funcionamiento del while: Mientras no hayamos recorrido toda la lista, tendremos una variable, comprar, que guardará el resultado de comparar el minimo parcial actual, con el de otro cursor en el que se esté buscando (inicialmente el segundo, ya que el primero se considera el minimo parcial inicial).

Si esta variable tiene un valor mayor que 0, siginifica que hemos encontrado un cerdo más joven, por lo que ahora el mínimo parcial será el susodicho, y se avanza el cursor para comprobar en la siguiente posición.

Si la variable no es mayor que 0, entonces se avanza a la siguiente posición.

Al final, se devuelve minum.element() porque este método devuelve un objeto EntryCerdo<Integer,String>, por lo que lo que hay que devolver no es el cursor, sino el elemento del cursor que es el objeto EntryCerdo<Integer,String>.

Método removeMin

Este método se encarga de mostrar cuál es el siguiente elemento que saldrá de la cola, y sacarlo de la cola. Este método puede lanzar la excepción *EmptyPriorityQueueException* si la lista atributo es null o está vacía.

En otro caso, debemos ir recorriendo la lista hasta encontrarnos con el *min* de la PQ. Una vez lo encontremos, debemos eliminarlo de la PQ (eliminándolo de la lista). Quedaría más o menos así:

```
/**
	 * removeMin : Muestra el elemento que va a salir y lo borra, lo saca de la cola
	 */
	public Entry<Integer, String> removeMin() throws EmptyPriorityQueueException {

		if(list == null || list.isEmpty())
			throw new EmptyPriorityQueueException ("La cola esta vacia");

		EntryCerdo<Integer,String> min = min();

		Position<EntryCerdo<Integer,String>> cursor = list.first();

		boolean found = false;
		while(cursor != null && !found)
		{
			if (min.equals(cursor.element()))
			{
				found = true;
				list.remove(cursor);
			}
			else
				cursor = list.next(cursor);
		}
		return min;
	}
```

Obviamente se recurre al método *min* anterior para identificar al elemento que debe salir de la PQ.

Otros métodos

Es posible e incluso recomendable declarar más métodos adicionales, como por ejemplo un *toString* para ver el contenido de nuestra PQ. Puede implementarse de la siguiente forma:

```
public String toString () {

		String res = "";
		if (!list.isEmpty()) {
			Position<EntryCerdo<Integer,String>> cursor = list.first();

			while (cursor != null)
			{
				res += cursor.element().toString() + "\n";
				cursor = list.next(cursor);
			}
		}
		else
			res = "Corral vacio";
		return res;
	}
```

Pruebas

```java
public static void main(String[] args) {

        AgeComparator <Integer> comp = new AgeComparator<Integer>();

        PositionListPQ pqlist = new PositionListPQ (comp); //Creacion PQ

        pqlist.insert(18, "El yayo");
        pqlist.insert(15, "josu");
        pqlist.insert(9, "porqui");
        pqlist.insert(14, "pinky");
        pqlist.insert(3, "guarro");

        System.out.println("COMO ESTA EL CORRAL");
        System.out.println(pqlist.toString());
        System.out.println("Tamaño del corral: " + pqlist.size());

        System.out.println("");
        System.out.println("Matamos al mas joven");
        System.out.println(pqlist.removeMin());
        System.out.println("");

        System.out.println("COMO ESTA EL CORRAL");
        System.out.println(pqlist.toString());
        System.out.println("Tamaño del corral: " + pqlist.size());

        System.out.println("");
        System.out.println("Matamos al mas joven");
        System.out.println(pqlist.removeMin());
        System.out.println("");

        System.out.println("COMO ESTA EL CORRAL");
        System.out.println(pqlist.toString());
        System.out.println("Tamaño del corral: " + pqlist.size());

        System.out.println("");
        System.out.println("Matamos al mas joven");
        System.out.println(pqlist.removeMin());
        System.out.println("");

        System.out.println("COMO ESTA EL CORRAL");
        System.out.println(pqlist.toString());
        System.out.println("Tamaño del corral: " + pqlist.size());

        System.out.println("");
        System.out.println("Matamos al mas joven");
        System.out.println(pqlist.removeMin());
        System.out.println("");

        System.out.println("COMO ESTA EL CORRAL");
        System.out.println(pqlist.toString());
        System.out.println("Tamaño del corral: " + pqlist.size());

        System.out.println("");
        System.out.println("Matamos al mas joven");
        System.out.println(pqlist.removeMin());
        System.out.println("");

        System.out.println("COMO ESTA EL CORRAL");
        System.out.println(pqlist.toString());
        System.out.println("Tamaño del corral: " + pqlist.size());
}
```