Actas XVIII JENUI 2012, Ciudad Real, 10-13 de julio 2012
I.S.B.N. 10: 84-615-7157-6 | I.S.B.N. 13:978-84-615-7157-4
Páginas 351-358

Extensión del Collections Framework de Java con una orientación docente

Gemma López Mínguez   Juan Gutiérrez Aguado
Departamento de Informática
Universidad de Valencia
Avda. de la Universidad s/n
46100 Burjassot (Valencia)
gemlomin@alumni.uv.es   juan.gutierrez@uv.es

Resumen

El *Collections Framework* de Java ofrece un conjunto amplio de tipos pero no está orientado a la docencia. Otras bibliotecas que incluyen algunos libros de texto sobre estructuras de datos y algoritmos o bien no están actualizadas (no usan tipos parametrizados por ejemplo) o no contienen algunos tipos que consideramos interesantes para implementar algoritmos sobre grafos (por ejemplo una cola con prioridad modificable). En este trabajo se presenta una biblioteca de tipos en Java que incluye algunas características que la hacen idónea para ser usada en cursos de estructuras de datos y algoritmos. En concreto: ofrece tipos adicionales (árbol general, grafo y cola con prioridad modificable); cada tipo tiene un método para obtener el coste (número de operaciones realizadas sobre los datos) del último método ejecutado; todos los tipos disponen de un método para mostrar la estructura interna; ofrece tipos para la generación gráficas de costes; y la obtención de las instancias se realiza a través de clases de factoría. Mediante el uso de esta biblioteca es posible diseñar actividades (presenciales o autónomas) para que los alumnos analicen las implicaciones que tienen diferentes implementaciones del tipo de datos sobre el coste de las operaciones o visualicen la evolución de la estructura al llamar a una serie de métodos.

Summary

The Java Collections Framework offers a considerable amount of types (possible too many for introductory courses on data structures and algorithms). Other libraries included in textbooks are not up to date (they do not offer parametrized types for instance) or do not contain some data types that are interesting to implement graph algorithms (as a modifiable priority queue). In this work we present a Java data type library that includes some valuable characteristics to teach data structures and algorithms. These are: it offers additional types such as general tree, graph, and modifiable priority queue; all the data types offer methods to obtain the cost of the last operation, and to show the internal structure; it offers a type that allows to create cost plots hidding details about Input/Output and gnuplot; and, only interfaces and factory clases are public thus hidding the implementation. With this library it is possible to design learning activities, so that students can study how the different implementations affect the cost or to see how the internal structure evolves as a result of calling some methods.

Palabras clave

Estructuras de datos, tipos parametrizados, cola de prioridad modificable, grafos.

1. Introducción

Este trabajo surge de la necesidad de abordar el aprendizaje de las estructuras de datos desde diferentes puntos de vista ofreciendo herramientas que faciliten su estudio. Los alumnos en muchas ocasiones se centran en los detalles de “bajo nivel” (aunque necesarios) de la implementación de los tipos. Por ello resulta necesario focalizar su atención en otro aspecto: la repercusión de la representación en el coste de las operaciones que ofrece. Para abordar esta cuestión puede ser interesante que la biblioteca de tipos ofrezca una serie de herramientas de forma nativa.

Existen diferentes bibliotecas de tipos disponibles en la web como código de soporte a libros de texto sobre estructuras de datos y algoritmos. Entre los más conocidos podríamos citar el de Goodrich y Tamassia [1] y el de Weiss [3]. Estas bibliotecas

son muy completas en cuanto al número de tipos que ofrecen [1]. Sin embargo, en ninguna de ellas se han encontrado las características orientadas a la docencia que ofrece el paquete que presentamos en este trabajo: no incluyen la posibilidad de visualizar la estructura interna del tipo de dato (de forma nativa), ni permiten la obtención de los costes de las operaciones que se realizan sobre los tipos de datos.

En concreto, la biblioteca de tipos que se presenta en este trabajo [2] tiene las siguientes características:

- Integra una serie de elementos que resultan interesantes para la docencia de asignaturas de estructuras de datos y algoritmos. Todos los tipos ofrecen: la presencia de un método que permite obtener el coste de la última operación realizada; y la presencia de un método que permite generar una representación gráfica de la estructura (para ello usamos la clase LJV [2]).
- Ofrece implementaciones adicionales de tipos que no se encuentran en otras bibliotecas: la cola con prioridad modificable.
- Constituye un ejemplo de uso de requisitos sobre el tipo parametrizado.
- Con el objeto de facilitar su uso se ofrece una clase con la que realizar experimentos de coste es sencillo ya que ofrece métodos para: recopilar datos, crear un fichero con formato gnuplot y generar la gráfica correspondiente.

En la sección 2 se presentan detalles sobre la implementación. A continuación, la sección 3 presenta ejemplos de resultados obtenidos. En la sección 4 se muestran ejemplos de tareas que se pueden plantear. Finalmente se esboza el trabajo futuro y se exponen las conclusiones.

2. Detalles sobre la implementación

Esta biblioteca consta de 7 paquetes, 37 clases y 8 interfaces. En esta sección se mostrarán los detalles sobre la implementación. Comenzaremos con una descripción de los tipos que ofrece y cómo están organizados. Posteriormente mostraremos un ejemplo de requisito impuesto sobre el tipo que almacena una de las estructuras proporcionadas. Seguiremos con una descripción de las clases de factoría que han sido utilizadas con el fin de ocultar la implementación. Finalizaremos la sección comentando los métodos y tipos que se ofrecen para mostrar la representación y obtener costes.

2.1. Tipos ofrecidos y organización

Los tipos ofrecidos están organizados en una serie de paquetes que describimos a continuación.

Hay una serie de paquetes que contienen las estructuras de datos elementales. En el paquete `datastructures.list` se ofrecen tres implementaciones del TAD `Lista`. Una de las implementaciones utiliza un array; otra usa una lista enlazada; y la tercera usa una lista doblemente enlazada. El tipo `Pila` se encuentra en el paquete `datastructures.stack`. En este caso hay dos implementaciones: una con un vector y otra con una lista doblemente enlazada. El tipo `Cola` está definido en el paquete `datastructures.queue` y las implementaciones realizadas son: una cola circular con vector y otra con una lista enlazada.

En el paquete `datastructures.tree` se encuentra una implementación de un árbol general. El API de DOM (`Document Object Model`) de XML ha servido como base para decidir las operaciones que ofrece este tipo. De este modo, cuando nos encontramos en un nodo del árbol es posible ir al hermano siguiente, ir al hermano anterior, ir al padre, obtener todos los hijos, etc. Además ofrece la interfaz `Action` que se puede utilizar para realizar alguna acción sobre los nodos del árbol en los recorridos DFS y BFS.

En el paquete `datastructures.mpq` se ofrece una implementación de una cola con prioridad modificable. Esta implementación utiliza dos estructuras: por un lado un montículo binario para almacenar los elementos y una tabla de dispersión (`HashMap`) para almacenar las posiciones de los elementos en el montículo. Para modificar la prioridad de un elemento que está en el montículo primero hay que localizarlo. Mediante el uso de la tabla de dispersión conseguimos que el coste de la localización sea bajo.

El paquete `datastructures.graph` proporciona una implementación del grafo utilizando listas de adyacencia. Al diseñar el grafo se separó la información que almacena el grafo (arcos entre vértices

[1]Estas bibliotecas están disponibles en:
`http://net3.datastructures.net/`
`http://users.cis.fiu.edu/~weiss/dsaajava3/code/`

[2]Esta biblioteca se encuentra en la página
`http://www.uv.es/jgutierr/datastructures/sp/`

representados como enteros) y la información que contiene cada vértice (que puede ser cualquier tipo `Serializable` para guardar y recuperar esta información).

Finalmente el paquete `datastructures.common` contiene dos clases: la clase `LJV`[3] y la clase `Experiment`. La primera de ellas se usa para implementar los métodos que permiten mostrar la estructura interna del tipo. La segunda clase sirve manejar un experimento de coste en función de la talla. En la última subsección de esta sección se proporcionan más detalles.

Como ejemplo, la figura 1 muestra el diagrama de clases del paquete `datastructures.tree`.

2.2. Uso de requisitos sobre los tipos parametrizados

En esta biblioteca de clases se han utilizado tipos parametrizados y en aquellas clases donde es necesario se han impuesto requisitos sobre el parámetro del tipo. Estos requisitos sobre el tipo se imponen indicando que debe implementar a una interfaz determinada.

Para mostrar un ejemplo usaremos la definición de la interfaz que define las operaciones que debe poseer toda cola con prioridad modificable.

```
public interface IMPQ<T extends
     ComparableModifiable<T, V>, V>{
  ...
  public T modifyPriority(T data, V value);
  ...
}
```

Como se puede ver en este código se impone un requisito sobre el tipo que va a contener. En concreto, se exige que implemente a la interface `ComparableModifiable<T,V>` que pertenece al mismo paquete.

La declaración de esta interfaz es la siguiente:

```
public interface ComparableModifiable<T,V> extends
     Comparable<T> {
        public void modifyValue(V value);

        public int compareWithValue(V value);
}
```

Esta interfaz extiende a la interfaz `java.util.Comparable<T>` que permite comparar elementos y esto es necesario para mantener la estructura de montículo. Además añade dos requisitos: que se pueda modificar su prioridad a partir del argumento de tipo *V* que se deja libre; y que se compare con otro valor pasado como argumento, este segundo método es necesario para saber si al modificar la prioridad tenemos que subir o bajar al elemento en el montículo. Aquí se puede observar otro de los criterios que han guiado el diseño de esta biblioteca: la flexibilidad. La idea es que se pueda pasar un objeto del tipo *V* que o bien represente la prioridad directamente (y por tanto será un tipo numérico) o un objeto a partir del cual se pueda obtener la prioridad.

Por tanto, estas clases se pueden utilizar como ejemplos de código donde es necesario imponer requisitos sobre el tipo que se pasa como parámetro.

2.3. Clases de factoría

Las clases e interfaces de esta biblioteca se han organizado en paquetes y en cada paquete sólo son visibles los tipos necesarios para el código cliente: las interfaces y clases de factoría que permiten obtener instancias de las diferentes implementaciones. El siguiente código muestra la clase de factoría que se encuentra en el paquete `datastructures.list`

```
package datastructures.list;

public class ListFactory{
   public  static <T> IList<T> newInstanceArray(){
      return new ListArray<T>();
   }

   public  static <T> IList<T> newInstanceArray(int
         cap){
      return new ListArray<T>(cap);
   }

   public static <T> IList<T> newInstanceLinkedList
        (){
      return new LinkedList<T>();
   }

   public static <T> IList<T>
        newInstanceDoublyLinkedList(){
      return new DoublyLinkedList<T>();
   }
}
```

Como se puede ver en este código, cada método crea una instancia de un determinado tipo, pero lo que se devuelve es una referencia del tipo de la interfaz. Esto hace que el código cliente sea independiente del objeto real creado.

[3] `http://www.cs.auckland.ac.nz/~j-hamer/LJV.html`

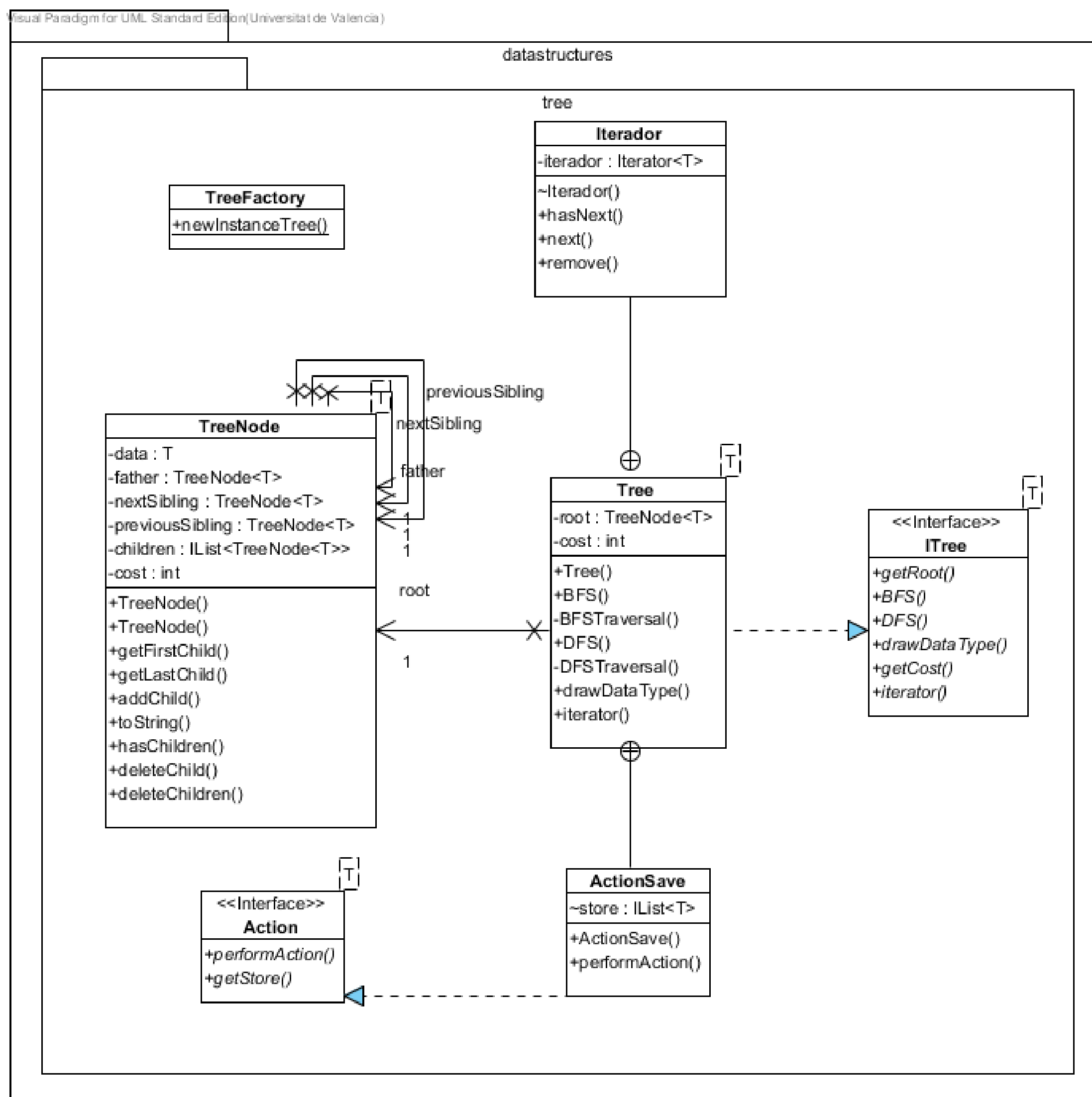

Figura 1: Diagrama de clases del paquete `datastructures.tree`

2.4. Obtención de costes y visualización de la estructura interna

Tal y como se ha comentado anteriormente, todos los tipos ofrecen el método `public int getCost()` que devuelve el coste de la última operación realizada. Para facilitar la realización de experimentos de coste en función de la talla se proporciona una clase llamada `Experiment`. Esta clase ofrece métodos para: almacenar series de datos de talla y coste; asignar las etiquetas del eje *x* y del eje *y*; establecer el título; y para generar el fichero gnuplot [4] y procesarlo para obtener una gráfica de costes. De este modo quien realiza el experimento no necesita conocer la Entrada/Salida en Java.

Además ofrecen el método `void`

[4] `http://www.gnuplot.info/`

`drawDataType(String f, Class c)` que almacena en el fichero `f` la estructura interna del tipo en formato gráfico, el segundo parámetro representa a una clase que queremos que sea mostrada usando `toString()` (es decir, no queremos que dibuje su estructura interna). Para la implementación de este método se ha utilizado la clase LJV. La clase `LJV` realiza llamadas a las herramientas de GraphViz[5] y

La información sobre las rutas donde están instaladas las aplicaciones graphviz y gnuplot se deben proporcionar en un fichero de propiedades.

3. Ejemplos de resultados obtenidos

En esta sección se presentan dos ejemplos que ilustran el tipo de resultados que se obtienen en cuanto a la generación de gráficas de costes y de diagramas de estructura interna.

La figura 2 muestra un ejemplo figura de coste en función de la talla para el caso peor de la inserción de elementos en la cola de prioridad. El caso peor consiste en insertar un elemento menor que los que existen en la cola hasta ese momento. Se trata del caso peor ya que ese elemento se inserta al final del montículo binario y debe subir hasta la raíz del mismo. Esta figura ha sido generada utilizando la clase `Experiment` comentada anteriormente.

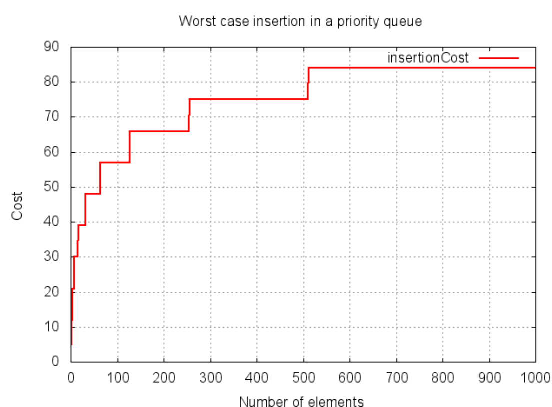

Figura 2: Gráfica de coste en el peor caso para la inserción de elementos en una cola de prioridad. Esta figura ha sido generada usando la clase `Experiment` incluida en la biblioteca.

[5]`http://www.graphviz.org/`

La figura 3 muestra el coste en función de la talla para el acceso a una posición aleatoria con tres implementaciones de una lista: una con un vector, otra con una lista enlazada y la tercera usando una lista doblemente enlazada. El coste de la lista doblemente enlazada es menor ya que en la implementación el recorrido se puede iniciar por el principio o por el final en función de dónde esté más próxima la posición a la que hay que ir. Esta figura también ha sido generada utilizando la clase `Experiment`.

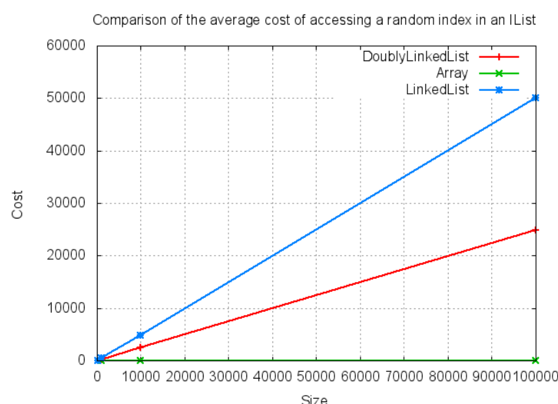

Figura 3: Gráfica comparativa de costes medios al acceder a una serie de posiciones aleatorias en una lista con tres implementaciones: con un vector, con una lista enlazada y con una lista doblemente enlazada. Esta figura ha sido generada usando la clase `Experiment` incluida en la biblioteca.

La figura 4 muestra dos ejemplos de diagramas con estructura interna de un `ITree` y de un `IQueue`.

4. Ejemplos de tareas que se pueden plantear

Una de las características principales que ha guiado este trabajo es que sea útil para el aprendizaje del alumno. En esta sección se presentan dos ejemplos de tareas que se pueden plantear a los alumnos. El primer ejemplo está relacionado con la visualización

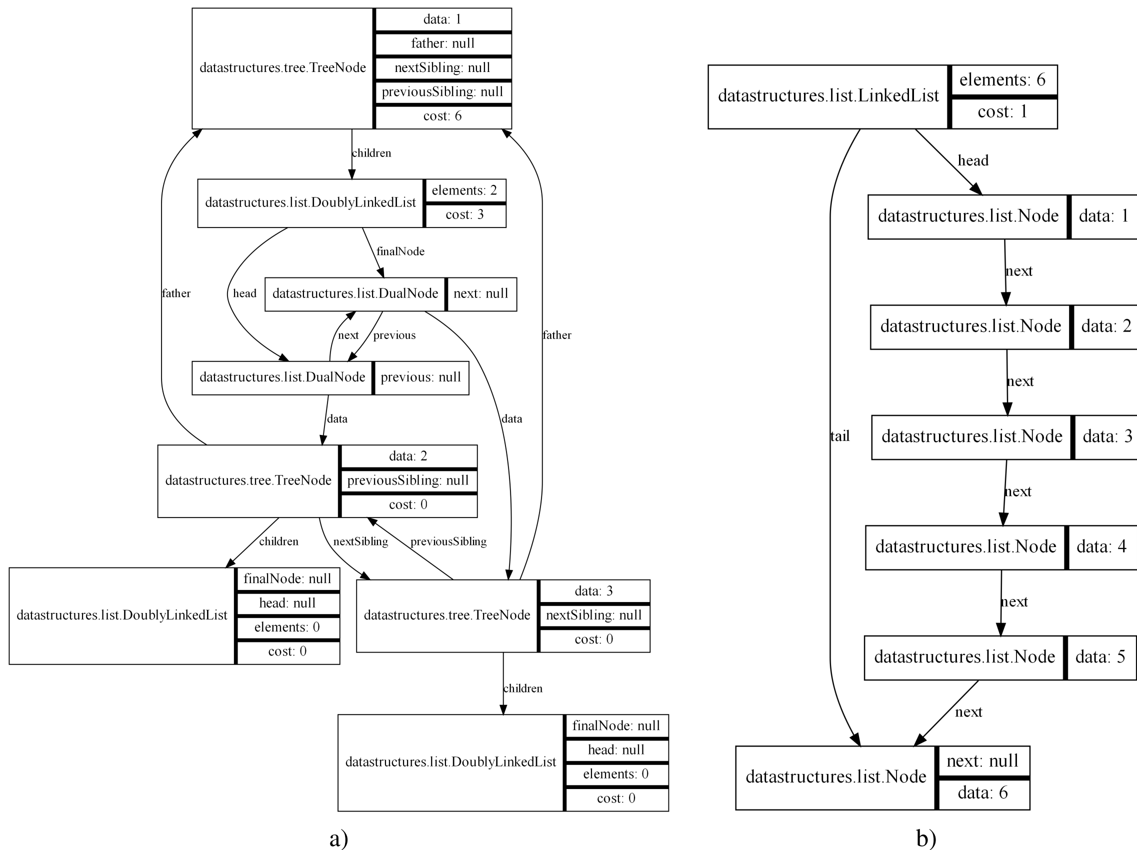

Figura 4: Ejemplos de diagramas generados enviando el mensaje `drawDataType(.)`. a) Un objeto del tipo `ITree` y b) un objeto del tipo `IQueue` implementado con una lista doblemente enlazada.

de la estructura interna y el segundo con la obtención de costes. Para la ejecución de las mismas se podría usar JUnit.

Como ejemplo de tarea se podría pedir a los alumnos que realicen la siguiente secuencia de pasos mostrando en cada uno de ellos la estructura interna para dos implementaciones diferentes:

1. Insertar *n* elementos en la cola.
2. Extraer un elemento de la cola.
3. Insertar un elemento en la cola.

Otro ejemplo de tarea consistiría en obtener el coste de realizar inserciones en una cola de prioridad (de mínimos) insertando un elemento menor que todos los que contiene (que constituye el peor caso del algoritmo). El código solución podría ser el siguiente (sólo se incluye el código del método de prueba usado en JUnit).

```
@Test
public final void testGetCost() {
   Experiment e = new Experiment();
   int n = 1000;
   IMPQ<Dato, Double> c = MPQFactory.
         newInstanceMPQM(n);
   for (int i = n - 1; i > 0; i--) {
      c.insert(new Dato(i));
      e.addData("insertionCost", n-i, c.getCost());
   }
   e.setTitle("Worst case insertion in a priority
         queue");
   e.setXLabel("Number of elements");
   e.setYLabel("Cost");
   e.generatePlot("figuras", "MPQM.txt", "
         insertMPQM.png");
}
```

Donde `Dato` es una clase que contiene un `double` con la prioridad y que implementa a la interfaz `ComparableModifiable<Dato,Double>`. La eje-

cución de esta prueba genera la figura 2. Como se puede observar es muy sencillo obtener gráficas de costes.

5. Trabajo futuro

En la actualidad estamos trabajando para desarrollar un *framework* de esquemas algorítmicos mediante un conjunto de clases abstractas que ofrezcan un esqueleto de los mismos. Este conjunto de esquemas usará los tipos de datos presentados en este trabajo y si algún problema concreto lo requiere se añadirán nuevos tipos.

Además, estamos utilizando la cola con prioridad modificable para desarrollar un *framework* similar a RMI (Remote Method Invocation). Nuestro desarrollo permite que se puedan registrar *stubs* asociados a diferentes servidores con el mismo nombre (es decir, tener un mismo servicio en diferentes máquinas) y que los servidores puedan informar al registro RMI sobre su carga (que será el valor de la prioridad). De este modo al cliente se le dará el *stub* que "mejor" pueda tratar su petición.

6. Conclusiones

En este trabajo se ha presentado una biblioteca con diferentes estructuras de datos concebida para la docencia. En concreto: se ha incorporado en cada tipo de dato un método que genera un diagrama con su estructura interna (para ello se ha utilizado la clase LVJ); además cada tipo de dato ofrece un método para obtener el coste de la última operación realizada. En relación con esto último se ofrece una clase que facilita la creación de gráficas de coste. Consideramos que estas dos características, que no están presentes en otras bibliotecas de tipos consultadas, pueden ayudar a los alumnos en el análisis de las diferentes implementaciones. Además se han mostrado ejemplos concretos de tareas que se pueden plantear a los alumnos en asignaturas relacionadas con las estructuras de datos y algoritmos.

7. Agradecimientos

Este trabajo ha sido parcialmente financiado con una Beca de Colaboración del Ministerio de Educación convocatoria 2011-2012 y ha sido desarrollado en el marco del proyecto 79/FO11/31 de Innovación Educativa del Vicerrectorado de Convergencia Europea y Calidad, Universidad de Valencia.

Referencias

[1] Goodrich, M. T. and Tamassia, R., *Data Structures and Algorithms in Java*, John Wiley and Sons, 2010.

[2] Hamer, J., *Visualising Java Data Structures as Graphs*, Sixth Australasian Conference on Computing Education, 2004.

[3] Weiss, M. A., *Data Structures and Algorithm Analysis in Java*, Addison-Wesley, 2007