EDA (ETS Ingeniería Informática). Curso 2016-2017

Práctica 1. Elementos para el diseño y uso de una EDA en Java

Uso de una Cola de Prioridad en una aplicación de recomendación de páginas Web

Departamento de Sistemas Informáticos y Computación. UPV

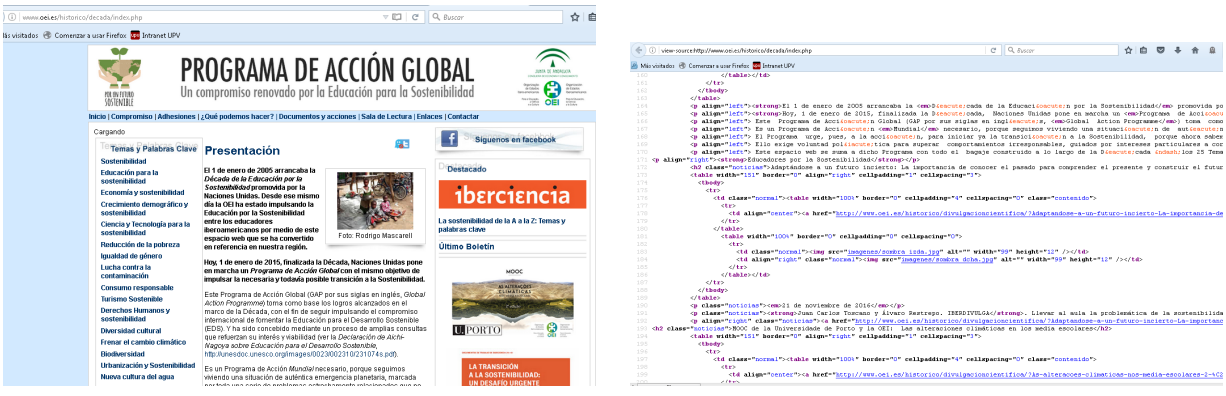
1. Descripción general, planificación y resultados de aprendizaje

El principal objetivo de esta práctica es que el alumno aplique los conceptos Java para Estructuras de Datos (EDAs) estudiados en el Tema 1 de la asignatura al diseño de una aplicación concreta. Específicamente, el alumno deberá ser capaz de implementar y utilizar eficazmente la jerarquía Java de la *Cola de Prioridad* que usa una aplicación de recomendación de páginas web.

Al mismo tiempo, como objetivo subsidiario, la realización de esta práctica permitirá al alumno crear y manejar la estructura básica de librerías de usuario *BlueJ* en la que, de forma incremental, irá ubicando las distintas clases Java que se desarrollen durante el curso. La práctica se desarrollará a lo largo de dos sesiones presenciales; en la primera sesión se podrían realizar las actividades 1 a 6 y en la segunda el resto.

1.1. Descripción del problema

Toda la información disponible en Internet se estructura como una serie de documentos que están localizados e identificados de forma única a través de su *url* (localizador uniforme de recursos). Una página Web es un fichero de texto escrito en *html* (lenguaje de marcado de hipertexto) y a través de diferentes marcas o etiquetas (*tags*) se describe la estructura de la página y la forma en la que se van a presentar los diferentes elementos cuando se carguen en el navegador (títulos, referencias, tablas, elementos gráficos, etc.) (ver el polimedia http://hdl.handle.net/10251/5809 del profesor M. Rebollo). En la figura 1 se muestra un ejemplo de página web (`http://www.oei.es/historico/decada`) y del código *html* asociado, que se puede obtener sin más que acceder a la página a través del navegador y seleccionar con el botón derecho *Ver código fuente de la página*. Una de las etiquetas es *href* que describe un hiperenlace; es decir, que pinchando en él se accede a otro documento. Si el hiperenlace contiene *http* indica que el hiperenlace es otra página web.



(a) (b)

Figura 1: Vista de la página web de la oei y de su código fuente

En esta práctica se va a desarrollar una aplicación que, previo análisis de diferentes páginas web, pueda ir recomendando o sugiriendo cuáles son las páginas que contienen más referencias a otras. El modelo de gestión de datos a utilizar será una *Cola de Prioridad* de la que se realizarán dos implementaciones diferentes sobre las que discutir la eficiencia en función de las operaciones más frecuentes.

1.2. Resultados de aprendizaje

Al acabar esta práctica el alumno debe ser capaz de:

- Reconocer la importancia de usar un modelo de datos para facilitar el uso de diferentes implementaciones que incorporen mejoras en la eficiencia.
- Argumentar sobre la adecuación de una implementación en función de las operaciones más frecuentes.
- Resolver un problema que requiera el uso del modelos de datos Cola de Prioridad.
- Diseñar una clase que implemente la interfaz Comparable para que pueda instanciar el parámetro de tipo de la clase Cola genérica.
- Usar e implementar eficientemente el modelo Lista Con Punto de Interés.
- Manejar con habilidad representaciones enlazadas.
- Crear una estructura de paquetes dada y ubicar las clases Java en los paquetes que se indican. Usar el entorno deprogramación para escribir código, compilar, ejecutar y depurar.

2. La aplicación de recomendación

A grandes rasgos, el funcionamiento del recomendador de páginas se podría describir como sigue: en primer lugar se crea el sistema de recomendación sin información o con la información extraída de una serie de páginas web cuyas urls se encuentran en un fichero de texto. En cualquier momento se puede solicitar información sobre la página recomendada o sobre las k páginas recomendadas. También se pueden añadir nuevas páginas para ser analizadas e incorporadas al sistema de recomendación o consultar el listado de páginas del sistema.

2.1. Las clases de la aplicación

Las principales clases que componen la aplicación son:

- `ReferenciasWeb`, que representa la información extraída del análisis de una página web. Una pagina Web `TIENE UNA` url que la identifica, `TIENE UN` número de líneas del fichero html y `TIENE UN` número de referencias externas. Es importante resaltar que el método `compareTo` de la clase `ReferenciasWeb` es el que permite establecer cuál de dos páginas dadas es más recomendable, tiene más prioridad: aquella que tenga un mayor número de referencias y, en el caso de que tengan el mismo número de referencias, la que tenga un menor número de líneas.
- `Recomendacion`, que representa el sistema de recomendación; de todas las páginas analizadas recomienda (extrae) aquella (s) que tiene (n) un mayor número de referencias externas (o a igualdad de referencias, menor número de líneas). Esta clase `TIENE UNA` *Cola de Prioridad* de `Referencia`; en la siguiente sección se recuerda la definición de esta EDA de Búsqueda Dinámica y se define su modelo Java.
- `GUIRecomendacion`, una sencilla interfaz gráfica para el sistema de recomendación. A través de ella, como se observa en la figura 2, se puede incluir una nueva página al sistema de recomendación, se puede consultar la página más recomendable (Top), las k más recomendables (Top k) o las páginas disponibles en el sistema (Listar páginas). Si en la llamada a su main se indica el nombre de un fichero de texto con los nombres de las url, una en cada línea, el sistema de recomendación se inicializa haciendo el análisis de todas ellas. Si se utiliza el fichero `pagsWebs.txt` que se deja disponible en poliformaT y se listan las páginas del sistema se obtiene lo que se ve en la parte derecha de la figura 2.

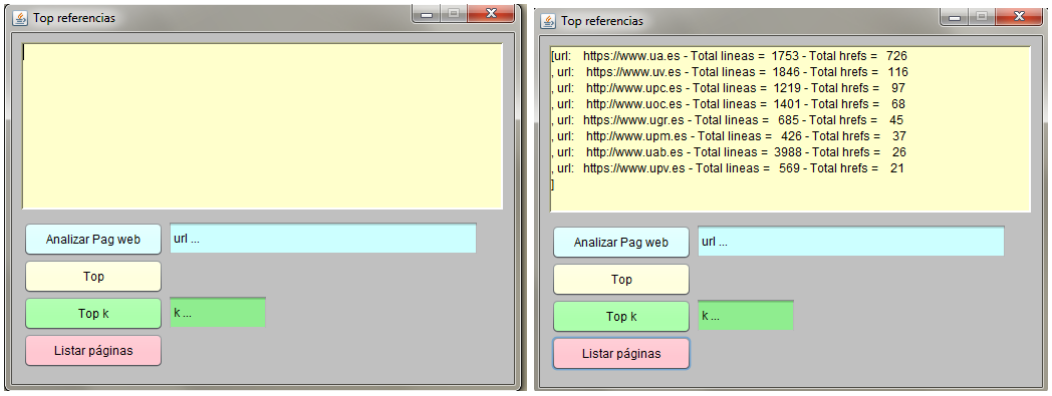


Figura 2: La interfaz gráfica `GUIRecomendacion`

2.2. *Cola de Prioridad*: modelo Java y su papel en la aplicación de recomendación

Una *Cola de Prioridad* es una colección homogénea de datos que solo se puede gestionar accediendo al dato que tiene la máxima prioridad. Para ello, nótese, es indispensable que cualquier dato de la colección, *d1*, sea *Comparable* con otro, *d2*, en base a su prioridad:

$$d1 < d2 \quad \texttt{SII} \quad \text{la prioridad de } d1 \text{ es mayor estricta que la de } d2$$

En base a este criterio de comparación resulta fácil deducir que el dato de mayor prioridad de la colección será el que mínimo tiempo de espera requiera para ser tratado o, equivalentemente, que acceder al dato más prioritario de una colección equivale a buscar su mínimo.

En el caso de datos con igual prioridad, en principio, se accede a ellos según un criterio *FIFO*, es decir, se accede al primero que se insertó en la *Cola de Prioridad*. Por todo ello, el comportamiento o modelo de gestión de una *Cola de Prioridad* se puede describir en Java como sigue:

```
public interface ColaPrioridad<E extends Comparable<E>> {
    /** Atendiendo a su prioridad, inserta el Elemento e en una Cola de Prioridad (CP) **/
    void insertar(E e);
    /** SII !esVacia(): obtiene y elimina el Elemento con máxima prioridad de una CP **/
    E eliminarMin();
    /** SII !esVacia(): obtiene el Elemento con máxima prioridad de una CP **/
    E recuperarMin();
    /** Comprueba si una Cola de Prioridad está vacía **/
    boolean esVacia();
}
```

La clase `Recomendacion` gestionará la información sobre las páginas web analizadas utilizando precisamente una `ColaPrioridad` de `ReferenciasWeb`; así se podrá añadir un nuevo análisis o se podrá consultar la página más recomendable según el criterio de la aplicación; es decir, la página más prioritaria.

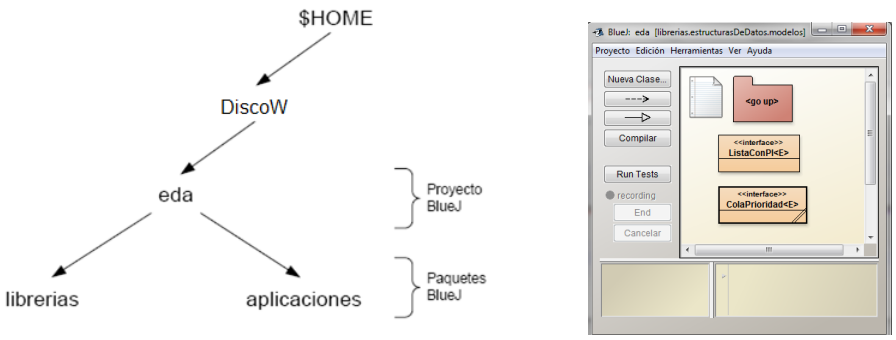
3. Actividades en el laboratorio

3.1. Actividad #1: Organización de paquetes y clases

Al tratarse de la primera sesión de prácticas, antes de llevar a cabo las actividades que se proponen en este apartado es necesario organizar el espacio de trabajo tal y como se muestra en la figura 3(a). Esto es, en primer lugar se debe crear un proyecto *BlueJ* denominado *eda* en el subdirectorio *DiscoW* del *HOME* del usuario. A su vez, dentro de este proyecto se deben crear dos nuevos paquetes *BlueJ*, *librerias* y *aplicaciones*, en los que se irán ubicando las clases asociadas a las distintas prácticas que se realicen este curso. Específicamente,

- en *librerias* se situarán los paquetes *BlueJ* con las clases de utilidades, excepciones y estructuras de datos;
- en *aplicaciones* se situarán los paquetes *BlueJ* con las clases de aplicaciones específicas.

En esta práctica se van a utilizar por primera vez dos modelos de EDAs: el de la *Cola de Prioridad*, que se usará para gestionar la información sobre las páginas analizadas del sistema de recomendación, y el de la *Lista Con Punto de Interés*, que se utilizará para diseñar una implementación lineal de una *Cola de Prioridad*. Estos modelos están definidos en Java mediante las interfaces `ColaPrioridad` y `ListaConPI` respectivamente, disponibles en *PoliformaT*. El alumno deberá crear el subpaquete *librerias.estructurasDeDatos* y dentro de él el paquete *modelos*. A continuación añadirá ambas interfaces a este paquete y las compilará (ver figura 3(b)).



(a) Los paquetes `librerias` y `aplicaciones` (b) El subpaquete `modelos`

Figura 3: Estructura del espacio de trabajo

3.2. Actividad #2: Implementación del modelo `ColaPrioridad` usando `ListaConPI`

En esta actividad se plantea el diseño de la primera implementación de la interfaz `ColaPrioridad`, la clase `LPIColaPrioridad` que implementa la interfaz haciendo uso, vía herencia, de los métodos de la clase `ListaConPI`; la idea es mantener todos los elementos de la *lista* ordenados de menor a mayor, de manera que el elemento más prioritario sea siempre el primero.

Para poder utilizar la interfaz `ListaConPI` es necesario tener disponible una implementación de la misma. Se puede conseguir una implementación eficiente de la interfaz `ListaConPI` utilizando una Lista Enlazada Genérica (LEG). Esta clase, que se diseñará en la actividad 7, recibe el nombre de `LEGListaConPI` y su correspondiente `.class` se encuentra disponible en *PoliformaT*. Para poder utilizarlo, dado que *BlueJ* solo permite añadir ficheros con extensión `.java`, el alumno deberá crear primero el paquete *BlueJ* que, en adelante, contendrá las implementaciones lineales de las EDAs que se diseñen: *librerias.estructurasDeDatos.lineales*; hecho esto, deberá salir de *BlueJ* y descargar el fichero `LEGListaConPI` en la carpeta o directorio del mismo nombre; finalmente, al entrar de nuevo en *BlueJ* y situarse en el paquete *librerias.estructurasDeDatos.lineales*, aparecerá el icono de la clase `LEGListaConPI` pero con el texto `(no source)` en su parte inferior para indicar que no corresponde a un fichero fuente y que, por tanto, no debe ser compilado sino solamente ejecutado. El alumno también deberá situar y compilar el fichero `NodoLEG.java` disponible en *PoliformaT* en el paquete *librerias.estructurasDeDatos.lineales*, pues contiene la clase de los nodos de una LEG que usa `LEGListaConPI`.

La clase `LPIColaPrioridad` se encuentra parcialmente implementada en *PoliformaT* y, al tratarse de una Implementación Lineal de `ColaPrioridad`, se añadirá al paquete *librerias.estructurasDeDatos.lineales*. El contenido del paquete `lineales` después de esta actividad puede verse en la figura 4.

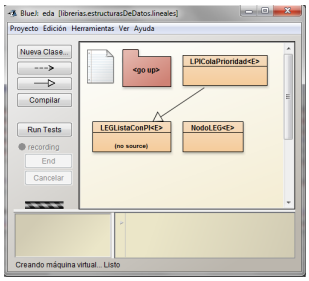


Figura 4: El paquete *librerias.estructurasDeDatos.lineales*

3.3. Actividad #3: Implementación del modelo `ColaPrioridad` usando la clase estándar `PriorityQueue`

Se trata ahora de realizar una segunda implementación de la interfaz `ColaPrioridad`, dejándose para la última actividad su comparación en base a criterios de eficiencia con la implementación anterior.

Está segunda implementación dará lugar a la clase `PriorityQColaPrioridad` que implementa la interfaz haciendo uso, vía Herencia, de los métodos de la clase `PriorityQueue` de la jerarquía `java.util.Collection` del estándar de Java. Como se puede leer en la documentación del API de Java, dicha clase representa una *Cola de Prioridad* mediante un Montículo Binario (*Heap*), un tipo particular de Árbol Binario que se estudiará en el Tema 5 de la asignatura. El esqueleto de la clase `PriorityQColaPrioridad` se encuentra en *PoliformaT* y deberá añadirse al nuevo paquete *librerias.estructurasDeDatos.jerarquicos*, que en adelante contendrá todas las Implementaciones Jerárquicas, basadas en un Árbol, de las EDAs que se diseñen. El contenido del paquete `jerarquicos` después de esta actividad puede verse en la figura 5.

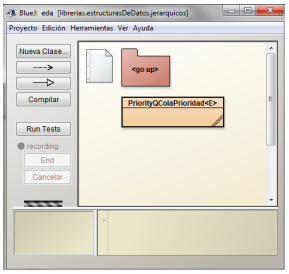


Figura 5: El paquete *librerias.estructurasDeDatos.jerarquicos*

3.4. Actividad #4: Implementación de la clase `ReferenciasWeb`

Las clases que se utilizan para modelar el sistema de recomendación (`ReferenciasWeb`, `Recomendacion` y `GUIRecomendacion`) se encuentran disponibles en *PoliformaT* y se deberán incluir en un nuevo paquete *Bluej* del proyecto *eda* denominado *aplicaciones.paginasWeb*.

El alumno deberá completar la clase `ReferenciasWeb` para que pueda instanciar los objetos de una Cola de Prioridad. Para ello deberá:

1. Indicar en su cabecera que esta clase implementa el interfaz `Comparable`.
2. Implementar el método `compareTo` de esta clase `Referencia` que, como se ha comentado anteriormente, se utiliza para establecer la prioridad de recomendación de una página: una página se recomendará antes que otra si tiene más enlaces externos o, a igualdad de enlacesm tiene menos líneas.

3.5. Actividad #5: Implementación de la clase `Recomendacion`

Esta clase permitirá gestionar la información procedente del análisis de un número cualquiera de páginas Web con el fin de servir como sistema de recomendación. Para ello mantendrá la información de las páginas analizadas en una Cola de Prioridad para permitir en todo momento un acceso eficiente a la página que debe ser recomendada. esta clase debe permitir la siguiente funcionalidad:

1. Crear el sistema de recomendación a partir del análisis de una lista de direcciones url que se proporcionan en un fichero de texto.
2. Crear el sistema de recomendación por defecto, esto es, sin información.
3. Recomendar la página que contenga más enlaces externos, y, a igualdad de enlaces, menor número de líneas.
4. Recomendar las k páginas que contengan más enlaces externos, y, a igualdad de enlaces, menor número de líneas.
5. Obtener una representación textual de las páginas analizadas y disponibles para ser recomendadas.

En esta actividad el alumno deberá completar la clase `Recomendacion`.

3.6. Actividad #6: Validación de las clases `ReferenciasWeb` y `Recomendacion`

El alumno dispone en *PoliformaT* del fichero `TestRecomendacion.class`, que le permitirá verificar la corrección de los distintos métodos que ha implementado durante la sesión: el `compareTo` de la clase `ReferenciasWeb` y los métodos de las clases `LPIColaPrioridad` y `PriorityQColaPrioridad`. Este fichero se debe descargar en la carpeta o directorio *aplicaciones.paginasWeb*, tras salir de *BlueJ* pues se trata de un `.class`; además, antes de ejecutarlo, el alumno debe comprobar que la estructura de paquetes y ficheros que ha creado en su proyecto *BlueJ eda* debe contener al menos las clases que se muestran en la figura 6; para ello puede hacer uso del comando `tree`.

```
.
├── aplicaciones
│   └── paginasWeb
│       ├── GUIRecomendacion.java
│       ├── Recomendacion.java
│       ├── ReferenciasWeb.java
│       └── TestRecomendacion.java
└── librerias
    └── estructurasDeDatos
        ├── jerarquicos
        │   └── PriorityQColaPrioridad.java
        ├── lineales
        │   ├── LPIColaPrioridad.java
        │   ├── NodoLEG.java
        │   └── LEGListaConPI.java
        └── modelos
            ├── ColaPrioridad.java
            └── ListaConPI.java
```

Figura 6: Estructura del proyecto *BlueJ eda* tras las 6 primeras actividades

Una vez sometidas las clases a los tests de prueba propuestos, es el momento de utilizar, si se desea, la aplicación gráfica; para ello se deberá ejecutar el método `main` de la clase `GUIRecomendacion`, si se desea puede utilizarse el fichero de urls `pagsWebs` que se descargará de polifomaT y se ubicará en la carpeta `paginasWeb`.

3.7. Actividad #7: Diseño de una implementación eficiente de `ListaConPI`: la clase `LEGListaConPI`

Antes de diseñar una implementación de Lista Con Punto de Interés (PI) es importante recordar que, al contrario de lo que ocurre en una Pila y una Cola, un elemento se puede insertar y eliminar de cualquier punto de su estructura. Así, la implementación de estas operaciones sobre un `array` supone siempre un desplazamiento de datos de coste directamente proporcional a la posición del PI de la Lista, lo que descarta *a-priori* esta representación. La única opción que resta entonces es la enlazada, mediante una LEG que, para poder implementar las operaciones `recuperar()`, `inicio()` y `fin()` de una Lista con PI en $\Theta(1)$, debe definir, al menos, tres referencias como atributos:

- `NodoLEG<E> pri`, la referencia al primer nodo de la LEG;
- `NodoLEG<E> ult`, la referencia al último nodo de la LEG;
- `NodoLEG<E> pI`, la referencia al nodo de la LEG cuyo dato ocupa el PI de la Lista.

Con esta representación, y siempre en $\Theta(1)$, ...

- se obtiene el dato que ocupa el PI de la Lista (`recuperar()`) accediendo a `pI.dato`;
- se sitúa el PI en inicio de Lista (`inicio()`) con la instrucción `pI = pri;`
- se sitúa el PI en fin de Lista (`fin()`) con la instrucción `pI = ult.siguiente;`
- se comprueba si el PI está en fin de Lista (`esFin()`) con la expresión `pI == ult.siguiente`.

Sin embargo, como tanto para borrar el nodo al que referencia `pI` como para insertar uno nuevo antes de él hay que disponer de un enlace al que le precede, esta implementación de Lista Con PI no permite hacerlo en tiempo constante; la figura 7 ilustra este problema para una `LEG` donde se quiere eliminar el último nodo:

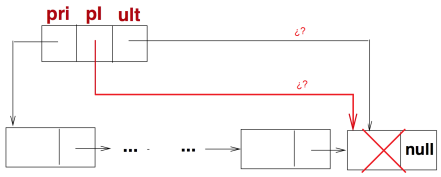


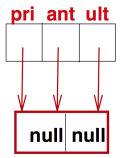
Figura 7: Implementación ineficiente de *Lista Con PI*

Para solucionar este problema de coste, i.e. para acceder al nodo anterior al que referencia `pI` en $\Theta(1)$, una sencilla solución es: substituir la referencia `pI` de la representación actual por la referencia al nodo `anterior` al que apunta `pI`; con ello, como se observa en la figura 8(a), la referencia `ant.siguiente` pasa a ser la que representa el PI de la Lista. Así, por ejemplo, usando la referencia `ant`, el código básico para `insertar(x)` es:

```
NodoLEG<E> nuevo = new NodoLEG<E>(x);
nuevo.siguiente = ant.siguiente;
ant.siguiente = nuevo;
```

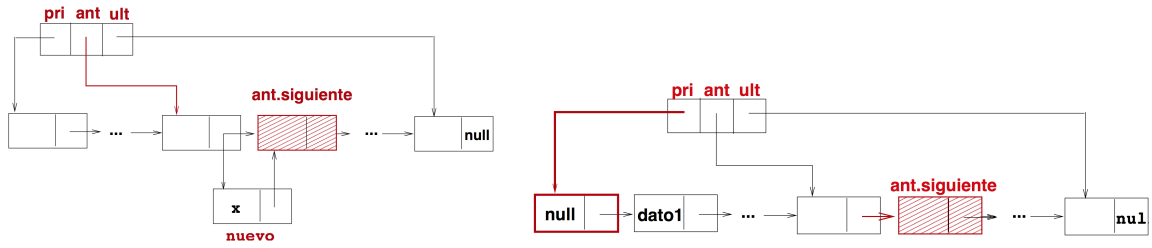
Ahora bien, y como quizás se haya observado, usar la referencia `ant` en lugar de `pI` presenta un inconveniente: `ant` no está definido en inicio de Lista. Para solventarlo caben dos opciones, ambas del mismo coste: bien desarrollar código específico para borrar e insertar en inicio de LEG, bien añadir a la LEG un primer nodo cabecera ficticio (i.e. un nodo al que siempre apunta `pri` y cuyo dato siempre es `null`). La implementación a realizar debe seguir la segunda opción y para ello se debe tener en cuenta que:

- Crear una Lista con PI vacía implica crear un nodo cabecera ficticio al que referencien `pri`, `ant` y `ult`, tal y como muestra la figura 8(b): `pri = ult = ant = new NodoLEG<E>(null);`.



- Considerando que las referencias `pri`, `ant` y `ult` pueden coincidir en el caso de Lista vacía, pero que `pri` y `ant` sólo coinciden cuando el PI de una Lista se sitúa en su inicio, las intrucciones que implementan las operaciones `esVacia()` e `inicio()` de una Lista Con PI son, respectivamente, `return pri == ult;` y `ant = pri;`

- Considerando que las únicas referencias que siempre coinciden en fin de Lista son `ult` y `ant`, las instrucciones de los métodos `fin()` y `esFin()` de una Lista Con PI serán, respectivamente, `ant = ult;` y `return ant == ult;`



(a) Utilizando como atributo el anterior al PI. (b) Utilizando además nodo cabecera ficticio

Figura 8: Implementación eficiente de *Lista Con PI*

Ahora es el momento de completar la clase `LEGListaConPI`, también disponible en *PoliformaT* y que se deberá añadir al paquete *librerias.estructurasDeDatos.lineales*. Tras conseguir compilar la clase `LEGListaConPI`, el alumno tiene que comprobar si el código que ha escrito en esta actividad funciona correctamente ejecutando el programa `PruebaLEGLPI` en el paquete *lineales* de su proyecto *BlueJ eda*. Para poder ejecutarlo se ha de descargar de *Poliformat* el fichero `PruebaLEGLPI.class` en esta misma carpeta.

Finalmente conviene comprobar que la clase `TestRecomendacion.class` (en *aplicaciones.paginasWeb*) sigue funcionando correctamente cuando use sa la clase `LEGListaConPI` desarrollada en esta actividad.

3.8. Actividad #8: Análisis y comparación de la eficiencia de los métodos de `LPIColaPrioridad` y `PriorityQColaPrioridad`

En *PoliformaT* se encuentra disponible una aplicación (`EficienciaCPGUI`) que, en modo gráfico, permite obtener y comparar el coste temporal promedio de los métodos `insertar` y `eliminarMin` implementados en las clases `LPIColaPrioridad` y `PriorityQColaPrioridad`. Esta aplicación utiliza la clase `Graph2D`) de la librería predefinida `graphLib.jar` mediante la que es posible realizar representaciones gráficas de puntos y líneas en un espacio bidimensional; en el fichero `Graph2D.html` se te proporciona la documentación de esta clase.

Para ejecutar la aplicación `EficienciaCPGUI` el alumno deberá previamente:

1. Crear un nuevo subpaquete, de nombre `util` en `librerias`.
2. Copiar en `$HOME/DiscoW/eda/librerias/util` los ficheros `EficienciaCPGUI.java` y `graphLib.jar`.
3. Reconfigurar el entorno BlueJ para, en `preferencias/librerias`, añadir la librería (el fichero `graphLib.jar`).

Al acabar, la estructura de carpetas y ficheros del directorio `eda` debe ser como se muestra en la figura 9.

```
.
├── aplicaciones
│   └── paginasWeb
│       ├── GUIRecomendacion.java
│       ├── Recomendacion.java
│       ├── ReferenciasWeb.java
│       └── TestRecomendacion.java
└── librerias
    ├── estructurasDeDatos
    │   ├── jerarquicos
    │   │   └── PriorityQColaPrioridad.java
    │   ├── lineales
    │   │   ├── LEGListaConPI.java
    │   │   ├── LPIColaPrioridad.java
    │   │   ├── NodoLEG.java
    │   │   └── PruebaLEGLPI.class
    │   └── modelos
    │       ├── ColaPrioridad.java
    │       └── ListaConPI.java
    └── util
        ├── EficienciaCPGUI.java
        └── graphLib.jar
```

Figura 9: Carpetas y ficheros del directorio `eda` al finalizar las actividades

Una vez comprobado que la estructura es correcta, el alumno ejecutará la aplicación `EficienciaCPGUI` y podrá ver los resultados del análisis de la eficiencia de los métodos `insertar` y `eliminarMin` implementados en las clases `LPIColaPrioridad` y `PriorityQColaPrioridad`: una ventana con los datos numéricos del análisis y tres representaciones gráficas, una con las 4 funciones y otras dos para el estudio de cada uno de los dos métodos (ver figura 10). En función de lo que observe, el alumno debe decidir cuál de las dos implementaciones de `ColaPrioridad` realizadas es la mejor en términos de eficiencia y, por tanto, la que debe reutilizar en la clase `Recomendacion` de la aplicación. hay que recordar que los resultados del estudio experimental del coste temporal de los métodos de `LPIColaPrioridad` refrendarán los proporcionados por su análisis teórico solo si el código desarrollado en la actividad `#7` es eficiente.

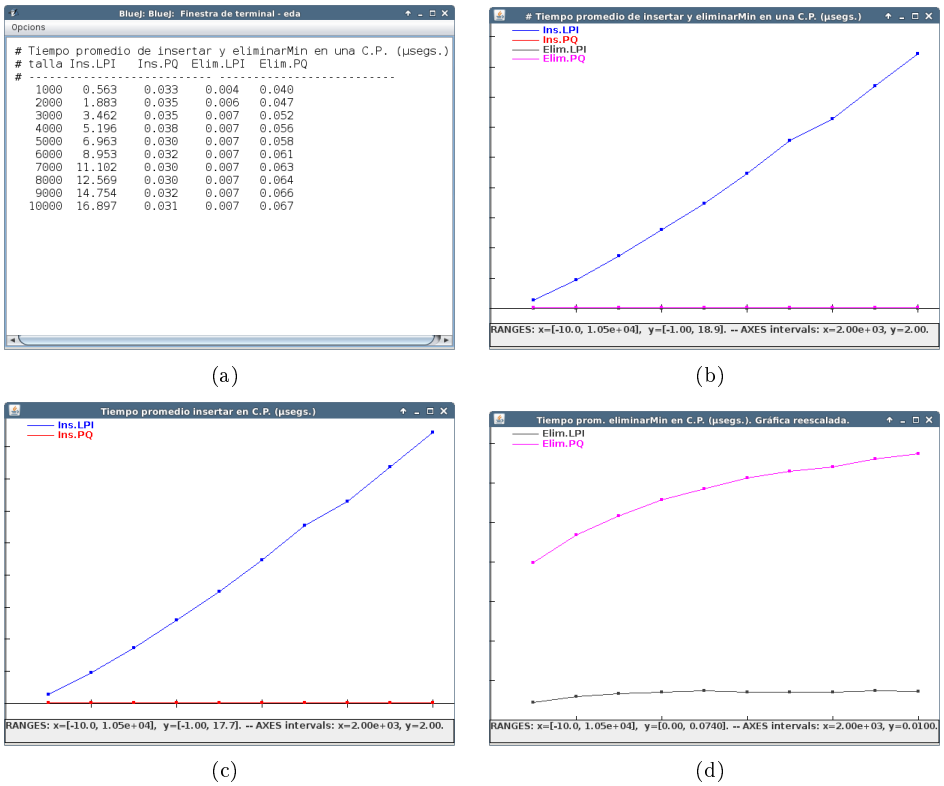


Figura 10: Ejemplo de ejecución de `EficienciaCPGUI`