Práctica 7
Lista doblemente ligada

Estructuras de datos

Meta

Que el alumno domine el manejo de información almacenada en una *Lista*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Visualizar cómo se almacena una lista en la memoria de la computadora mediante el uso de nodos con referencias a su elemento anterior y su elemento siguiente.
- Programar dicha representación en un lenguaje orientado a objetos.

Antecedentes

Definición 1

Una lista es:

$$\text{Lista} = \begin{cases} \text{Lista vacía} \\ \text{Dato seguido de otra lista} \end{cases}$$

Alternativamente:

Definición 2

Una **lista** es una secuencia de cero a más elementos **de un tipo determinado** (que por lo general se denominará tipo-elemento). Se representa como una sucesión de elementos separados por comas:

$$a_0, a_1, ..., a_{n-1}$$

donde $n \geqslant 0$ y cada a_i es del tipo **tipo-elemento**.

- Al número n de elementos se le llama *longitud* de la lista.
- a_0 es el *primer elemento* y a_{n-1} es el *último elemento*.

- Si $n = 0$, se tiene una **lista vacía**, es decir, que no tiene elementos. Aho, Hopcroft y Ullman 1983, pp. 427

En este caso utilizaremos como definición del tipo de datos abstracto, la interfaz definida por Oracle:

http://docs.oracle.com/javase/8/docs/api/java/util/List.html.

Actividad 1

Lee la definición de la interfaz `List<E>`. ¿Te queda claro lo qué debe hacer cada método? Si no, pregunta a tu ayudante.

Actividad 2

Elige los métodos que concideres más importantes y dibuja cómo te imaginas que se ve la lista antes de mandar llamar un método y qué le sucede cuando éste es invocado.

Una **lista doblemente ligada** es una implementación de la estructura de datos lista, que se caracteriza por:

1. Guardar los datos de la lista dentro de nodos que hacen referencia al nodo con el dato anterior y al nodo con el siguiente dato.
2. Tener una referencia al primer y último nodo.
3. Tener un tamaño dinámico, pues el número de datos que se puede almacenar está limitado únicamente por la memoria de la computadora y el tamaño de la lista se incrementa y decrementa conforme se insertan o eliminan datos de ella.
4. Es fácil recorrerla de inicio a fin o de fin a inicio.

Desarrollo

Para implementar el TDA *Lista* se deberá extender la clase abstracta `ColeccionAbstracta<E>` e implementar la interfaz `List<E>`. Esto se deberá hacer en una clase `ListaDoblementeLigada<E>`.

1. Crea la clase `ListaDoblementeLigada<E>`, que extiende `ColeccionAbstracta<E>` e implementa la interfaz `List<E>`.
2. Programa los métodos faltantes. Sólamente `sublist()` es opcional, los demás son obligatorios.

Preguntas

1. Explica la diferencia conceptual entre los tipos `Nodo<E>` y `E`.
2. ¿Por qué `ListIterator` sólo permite remover, agregar o cambiar datos después de llamar `previous` o `next`?
3. Si mantenemos los elementos ordenados alfabéticamente, por ejemplo, ¿cuándo sería más eficiente agregar un elemento desde el inicio o el final de la lista?
4. En qué casos sería más eficiente obtener un elemento desde el inicio de la lista o desde el final de la lista.

Forma de entrega

- Asegúrense de borrar todos los archivos generados, incluyendo archivos de respaldo usando `ant clean`.
- Copien el directorio `Practica7` dentro de un directorio llamado `<apellido1_apellido2>` donde `apellido1` es el primer apellido de un miembro del equipo y `apellido2` es el primer apellido del otro miembro. Por ejemplo: `marquez_vazquez`.
- Borren el directorio libs en la copia.
- Agreguen un archivo `reporte.pdf` dentro del directorio `Practica7` de la copia, con el nombre completo de los integrantes del equipo, las repuestas a las preguntas de la sección anterior y cualquier comentario que quieran hacer sobre la práctica.
- Compriman el directorio `<apellido1_apellido2>` creando

  `<apellido1_apellido2>.tar.gz`.

  Por ejemplo: `marquez_vazquez.tar.gz`.
- Suban este archivo en la sección correspondiente del aula virtual. Basta una entrega por equipo.

Forma de evaluación

Para su calificación final se tomarán en cuenta los aspectos siguientes:

70 %	Calificación generada por las pruebas automáticas. Usaremos nuestros archivos, por lo que si realizan modificaciones a sus copias, éstas no tendrán efecto al momento de calificar.
10 %	Reporte con respuestas a las preguntas.
10 %	Documentación completa y adecuada. Entrega en el formato requerido, sin archivos `.class`, respaldos, bibliotecas de `JUnit` u otros no requeridos.
10 %	Revisión manual del código, para verificar que se cumpla con las especificaciones, que no se haya copiado, etcétera.

Referencias

Aho, Alfred V., John E. Hopcroft y Jeffrey D. Ullman (1983). *Data Structures and Algorithms*. Addison-Wesley.