12 | TDA Lista

Definición

Definición 12.1

Una lista es:

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

Alternativamente:

Definición 12.2

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 12.1

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

Actividad 12.2

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.

Entregable: Entrega estos diagramas al menos para dos métodos. Se te sugiere utilizar software para dibujo de diagramas como Dia o dibujo vectorizado como Inkscape.

13 | Lista doblemente ligada

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

Estructura

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 con datos.
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.

En particular, un forma de mantener las referencias al primer y último nodo es utilizar un nodo *centinela*. La clase lista tendrá un atributo que siempre hace referencia al centinela.

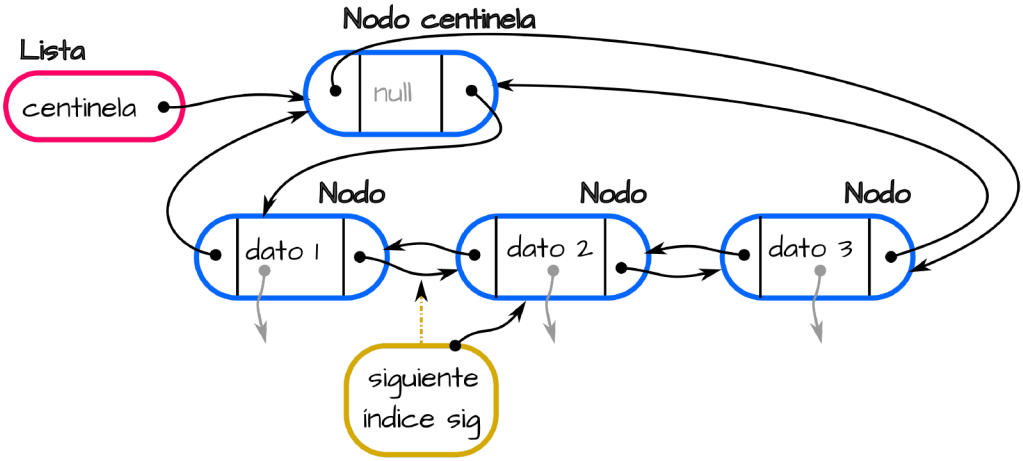

Figura 13.1 Lista doblemente ligada con nodos, centinela e iterador.

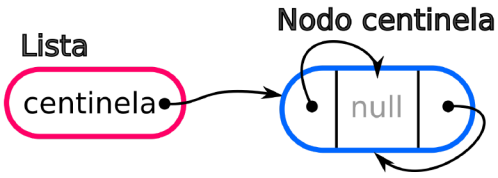

Figura 13.2 Lista con centinela vacía. El centinela apunta a si mismo como primer y último elemento.

Este nodo no guarda ningún dato en su interior, pero su nodo siguiente será el primer nodo de la lista y su nodo anterior será el último, como muestra la Figura 13.1; si la lista está vacía el centinela apunta a sí mismo en ambas direcciones Figura 13.2.

Con esta implementación los atributos `anterior` y `siguiente` de los nodos nunca es `null`, lo que facilitará mucho la programación de los métodos para agregar y remover elementos de la lista, pues agregar o remover de los extremos se ve exactamente igual que si se estuviera en medio de la lista.

Iterador

Conceptualmente, el iterador para una lista se encuentra colocado entre dos elementos sucesivos, de modo que siempre tiene un elemento siguiente y uno anterior, excepto en los extremos, por ello permite recorrerla de principio a fin y de regreso.

Actividad 13.1

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

Como la definición de la lista permite acceder a todos sus elementos, el iterador también tendrá la capacidad de insertar y remover datos conforme se desplaza a través de la estructura. Para ello, el objeto iterador debe contar con atributos que le permitan acceder a los nodos que serían devueltos por llamadas a los métodos `next` y `previous` y recordar qué movimiento fue el último que realizó. Además, como se vió en la definición de lista, cada elemento en ella tiene asociado un índice que indica su posición en la estructura; por ello el iterador también debe llevar la cuenta del número de elemento que devolverían cualquiera de las llamadas a los métodos anteriores y devolverlo en los métodos `nextIndex` y `previousIndex`.

La Figura 13.1 muestra cómo podemos visiualizar la colocación del objeto iterador. Observese que, a través de las diferentes referencias tanto en el iterador, como en los nodos de la lista, es posible acceder directa o indirectamente a todas las piezas de información que requieren los métodos anteriores y reasignar variables acordemente cuando el iterador sea desplazado.

Desarrollo

Para implementar el TDA *Lista* se deberá extender la clase:

`ColecciónAbstracta<E>`

programada anteriormente e implementar la interfaz `List<E>` del paquete `java.util`. Esto se deberá hacer en una clase llamada

`ListaDoblementeLigada<E>`.

dentro del paquete

`ed.estructuras.lineales`

Observa que, en esta ocasión, sí permitiremos el almacenamiento de datos `null` en la estructura, ten especial cuidado en los métodos donde operarás con los datos almacenados.

1. Dentro del paquete correspondiente programa una clase con acceso de paquete para representar al nodo con el dato a almacenar, de tipo genérico, y las referencias a los nodos anteriores y siguientes. Agrega los constructures que consideres necesarios, así como métodos de lectura y escritura.

   Otra opción es implementar esta clase como una clase interna estática de la clase ListaDoblementeLigada<E>, en este caso podrías no necesitar los métodos de

lectura y escritura pues tendrás permiso de acceder los atributos directamente mientras estés dentro de la clase que implementa la lista. Elige la opción que prefieras.

2. Ahora programa `ListaDoblementeLigada<E>` según lo indicado, comienza por el constructor para una lista vacía; en él deberás crear al centinela y asignar sus referencias como se indicó anteriormente.

3. A continuación programa el iterador. Observa que en esta ocasión se debe implementar `ListIterator<E>`, todos sus métodos son obligatorios. Prueba que los métodos del iterador funcionen, si programas de una vez el método `add(E e)` de la lista, las pruebas unitarias te ayudarán a ver si vas bien.

4. Programa ahora los demás métodos de la lista, observa que varios de los métodos ya los implementaste en `ColecciónAbstracta<E>`, para los que faltan usa el iterador cada vez que puedas, te ahorrará mucho trabajo y hará más fácil que tu código sea correcto.

   Sólamente el método `sublist()` es opcional, pues su correcta implementación implica programar por doble vez casi todos los métodos de lista, los demás métodos son obligatorios.

   Los métodos a programar son:

   - `public Iterator<E> iterator()`
   - `public ListIterator<E> listIterator()`
   - `public ListIterator<E> listIterator(int index)`
     Este método en particular te ayudará a programar varios de los métodos siguientes.
     Una optimización opcional para tu iterador es agregar un constructor que lo coloque al final de la lista. Si necesitas colocarlo en en alguna posición que está más cerca del final que del principio, colócalo al final y retrocede tantos lugares como necesites.
   - `public boolean addAll(int index, Collection<? extends E> c)`
     Ten cuidado con que `c` no sea `this`, si fuera el caso lanza una

     `IllegalArgumentException`
   - `public E get(int index)`
   - `public E set(int index, E element)`
   - `public void add(int index, E element)`
   - `public E remove(int index)`
   - `public int indexOf(Object o)`
   - `public int lastIndexOf(Object o)`

5. Sobreescribe el método `clear()` de manera que su complejidad sea $\mathcal{O}(\mathbf{1})$.

Preguntas

1. Explica la diferencia conceptual entre los tipos `Nodo<E>` y `E`.
2. ¿Por qué `ListIterator` sólo permite remover 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?