Práctica 6

Listas Ligadas Simples

Matú Hernández Diana

Rojo Mata Daniel

Explicación de los algoritmos

- **Método constructor.**

```
1       public ListaLigadaSimple(){
2           cabeza = null;
3           rabo = null;
4           longitud = 0;
5       }
```

El método que se implemento es correcto ya que el *constructor* inicializa el objeto deseado con sus propiedades y valores, en este caso, una lista vacia, por lo tanto sus propiedades: cabeza, rabo y longitud, se representan con *null* y la longitud de la lista con 0.

- **Método insertar**

```
1       public void insertar(T elemento) throws IllegalArgumentException {
2           Nodo nuevoNodo = new Nodo(elemento);
3           if (cabeza == null) {
4               cabeza = nuevoNodo;
5               rabo = nuevoNodo;
6           }
7           else {
```

```
8               cabeza.modificaSiguiente(nuevoNodo);
9               cabeza = nuevoNodo;
10          }
11          longitud++;
12      }
```

Cuando la lista está vacía, crea un nuevo nodo que se convierte en tanto la cabeza como el rabo de la lista, esto se puede apreciar en las líneas $2-5$. En el caso de una lista no vacía, bloque *else* $7-10$, el nuevo nodo se conecta correctamente al nodo que antes era la cabeza, y luego se convierte en la nueva cabeza.

Además, se asegura el incrementar la longitud de la lista después de cada inserción por un elemento, esto es, se suma una unidad a la longitud de la lista, línea 11.

Estas operaciones garantizan que los elementos se inserten correctamente al principio de la lista enlazada, manteniendo la estructura y la longitud de la lista de manera precisa, así pues, el algoritmo es correcto.

- **Método eliminar**

```
1       @Override
2       public void eliminar(int i) {
3           if (i < 0 || i >= this.longitud ) {
4               throw new IllegalArgumentException("Índice inválido");
5           }
6
7           Nodo nodoActual = this.cabeza;
8           Nodo nodoAnterior = null;
9
10          for (int j = 0; j < i; j++) {
11              nodoAnterior = nodoActual;
12              nodoActual = nodoActual.siguiente;
13          }
```

```
14
15          if (nodoAnterior == null) {
16              this.cabeza = nodoActual.siguiente;
17          }
18          else {
19              nodoAnterior.siguiente = nodoActual.siguiente;
20              if (nodoActual.siguiente == null) {
21                  this.rabo = nodoAnterior;
22              }
23          }
24
25          this.longitud--;
26      }
```

El bloque if $3-5$ asegura la inserción de índices válidos, esto es, no se admiten índices con valores negativos o que estén fuera del rango de la longitud de la lista. En caso contrario se lanza una excepción del tipo *IllegalArgumentException*.

Se itera a través de un ciclo for, líneas $10-13$, la lista hasta llegar al nodo en la posición del índice proporcionado. *nodoActual* es el nodo que se eliminará, en palabras, es el *nodo i*, mientras que *nodoAnterior* mantiene la referencia al *nodo i-1*, éstos se definen en las líneas $7-8$.

El bloque $if-else$, líneas $15-23$ consta de lo siguiente:

Si el nodo a eliminar es el primero (cabeza), se actualiza *this.cabeza* para que apunte al siguiente nodo.

Si el nodo a eliminar está en una posición diferente, se actualiza *nodoAnterior.siguiente* para omitir el nodo que se eliminará. Si el nodo a eliminar es el último nodo (rabo), se actualiza *this.rabo* para que apunte al nodo anterior.

Por último, se decrementa en una unidad la longitud de la lista, línea 25.

El método es correcto porque en principio comprueba índices válidos, evita nodos desconectados y actualiza enlaces para preservar la estructura de la lista, además, ajusta la longitud y maneja nodos de inicio y fin.

- **Método acceder**

```
1       @Override
2       public T acceder(int i) throws IllegalArgumentException {
3           if(this.longitud < i || i < 0){
4           throw new IllegalArgumentException("Índice no válido");
5           }
6
7           Nodo nodoActual = this.cabeza;
8           for(int j=0 ; j < i; j++){
9               nodoActual = nodoActual.siguiente;
10          }
11
12          return nodoActual.elemento;
13      }
```

El bloque $if\ 3-5$ asegura la inserción de índices válidos, esto es, no se admiten índices con valores negativos o que estén fuera del rango de la longitud de la lista. En caso contrario se lanza una excepción del tipo *IllegalArgumentException*.

El algoritmo utiliza un bucle for, líneas $8-10$, para iterar a través de la lista enlazada hasta llegar al nodo en la posición del índice proporcionado. Se inicializa un nodo llamado *nodoActual* el cual hace referencia a la cabeza de la lista.

Cada iteración mueve *nodoActual* al siguiente nodo en la lista, asegurando que se alcance la posición deseada, esto es, el índice i. El algoritmo devuelve el elemento del nodo en la posición del índice especificado, línea 12, que es correcto ya que proporciona el valor del nodo correspondiente al índice solicitado.

Para este caso se puede dar un invariante de ciclo en forma de Lema:

LEMA: Antes de la iteración j nodoActual hace referencia a la posición del nodo con índice i

- **Inicialización:** Al comienzo del bucle, $j = 0$, *nodoActual* se establece en la cabeza de la lista, que es la posición del nodo 0 ($NodonodoActual = this.cabeza;$), así, el invariante se cumple.

- **Mantenimiento:** En cada iteración, *nodoActual* se mueve al siguiente nodo ($nodoActual = nodoActual.siguiente;$). Dado que *nodoActual* se actualiza en cada paso, sigue apuntando al nodo siguiente en la lista en cada iteración, es decir, hace referencia al nodo en la posición i, luego, el invariante se mantiene.
- **Finalización:** Cuando el bucle finaliza (j alcanza el valor de i), *nodoActual* está en el último nodo catalogado por j, esto es, está en la posición del índice i. Por lo tanto, el invariante se mantiene al finalizar el bucle.

Con base a lo argumentado anteriormente, el método es correcto.

- **Método buscar**

```
1       @Override
2       public boolean buscar(T elemento) {
3           Nodo nodoActual = this.cabeza;
4
5           while (nodoActual != null) {
6               if (nodoActual.elemento.equals(elemento)) {
7                   return true;
8               }
9               nodoActual = nodoActual.siguiente;
10          }
11          return false;
12      }
```

El *Nodo nodoActual* se inicializa en la cabeza de la lista, que es el primer nodo (con índice 0).

El algoritmo recorre la lista mientras *nodoActual* no sea nulo verificando en cada iteración si el elemento del nodo actual es igual al elemento que se está buscando, bloque if $6-8$. Si encuentra una coincidencia, el algoritmo devuelve *true*, línea 7, indicando que el elemento está presente en la lista. Si no encuentra coincidencias en toda la lista, devuelve $false$ (línea

11) después de recorrerla por completo, esto es, cuando $nodoActual.siguiente = null$.

Se expresa un invariante de ciclo en forma de Lema:

> *LEMA: En cada iteración del bucle while, si el elemento está en la lista, será encontrado eventualmente en ésta o en las iteraciones posteriores, lo que llevará a la devolución de true. Si el bucle ha revisado toda la lista sin encontrar el elemento, el método devolverá false.*

- **Inicialización:** Al iniciar el bucle, *nodoActual* se establece en la cabeza de la lista, asegurando que se esté apuntando a un nodo válido (distinto de *null*). Si es el caso que $nodoActual.elemento$ es igual a *elemento*, se retorna *true*, encontrando al elemento en esta iteración, manteniéndose el invariante. Si no se encuentra se retorna simplemente $false$, pues no hay más iteraciones, cumpliéndose el invariante.

- **Mantenimiento:** Mientras *nodoActual* no sea *null*, el bucle se ejecuta. Dentro del bucle, se verifica si $nodoActual.elemento$ es igual al elemento pasado como argumento. Si hay una coincidencia, el método retorna *true* en ésta iteración. Si no hay coincidencia, *nodoActual* se mueve al siguiente nodo en la lista ($nodoActual = nodoActual.siguiente$;) es decir, la iteración posterior, si ocurre que esta iteración existe la coincidencia buscada, se retorna *true*, de lo contrario se repite lo anterior; se hace el comparativo en la iteración posterior. Si en alguna iteración posterior se satisface el bloque $if\ 6-8$, entonces el invariante se cumple.

  Ahora, si se hizo la comparación del bloque $6-8$ para toda iteración posterior hasta que $nodoActual = null$ (es decir, se comparó con todos los nodos) entonces se retorna $false$, indicando que no se encontró el elemento correspondiente, así pues, el invariante se mantiene.

  Este paso asegura que el algoritmo se desplaza correctamente a través de la lista, manteniendo el invariante del ciclo.

- **Finalización:** La etapa de finalización se alcanza cuando el bucle *while* ha revisado toda la lista y *nodoActual* se ha convertido en *null*. En este punto, el algoritmo sabe que el elemento buscado no está en la lista y devuelve $false$ para indicar que el elemento no fue encontrado en ningún nodo de la lista. Así pues, se cumple el invariante.

Con base a lo argumentado anteriormente, el método es correcto.

- **Método darLongitud**

```
1       public int darLongitud() {
2           return this.longitud;
3       }
```

En la línea 2 se retorna el atributo *longitud* de la lista. Si la lista es vacía se regresaría 0, de lo contrario se retorna un valor natural.

El algoritmo es correcto porque devuelve el valor actual de longitud, el cual se mantiene correctamente y refleja el número actual de elementos en la lista enlazada

Complejidad en tiempo y espacio

- **Método constructor.**

```
1       public ListaLigadaSimple(){
2           cabeza = null; // 1
3           rabo = null;   // 1
4           longitud = 0;  // 1
5       }
```

$$T(n) = 3$$
$$M(n) = 0$$

- **Método insertar.**

```
1       public void insertar(T elemento) throws IllegalArgumentException {
2           Nodo nuevoNodo = new Nodo(elemento); // 1
3           if (cabeza == null) {  // 1
4               cabeza = nuevoNodo;  // 1
5               rabo = nuevoNodo;  // 1
6           }
```

```
7           else {
8               cabeza.modificaSiguiente(nuevoNodo);  // 1
9               cabeza = nuevoNodo; // 1
10          }
11          longitud++;  // 2
12      }
```

$$T(n) = 8$$

$$M(n) = 1$$

- **Método eliminar.**

```
1       @Override
2       public void eliminar(int i) {
3           if (i < 0 || i >= this.longitud ) {  // 5
4               throw new IllegalArgumentException("Índice inválido");
5           }
6
7           Nodo nodoActual = this.cabeza;  // 2
8           Nodo nodoAnterior = null;  // 1
9
10          for (int j = 0; j < i; j++) { // 4
11              nodoAnterior = nodoActual; // 1
12              nodoActual = nodoActual.siguiente;  // 2
13          }
14
15          if (nodoAnterior == null) {  // 1
16              this.cabeza = nodoActual.siguiente; // 3
17          }
18          else {
```

```
19              nodoAnterior.siguiente = nodoActual.siguiente; // 3
20              if (nodoActual.siguiente == null) {  // 2
21                  this.rabo = nodoAnterior;  // 2
22              }
23          }
24
25          this.longitud--;  // 3
26      }
```

$$T(n) = 3n + 23$$

$$M(n) = 1$$

- **Método acceder.**

```
1       @Override
2       public T acceder(int i) throws IllegalArgumentException {
3           if(this.longitud < i || i < 0){  // 4
4           throw new IllegalArgumentException("Índice no válido");
5           }
6
7           Nodo nodoActual = this.cabeza;
8           for(int j=0 ; j < i; j++){  // 4
9               nodoActual = nodoActual.siguiente;  // 2
10          }
11
12          return nodoActual.elemento;  // 1
13      }
```

$$T(n) = 2n + 9$$

$$M(n) = 1$$

- **Método buscar.**

```java
@Override
public boolean buscar(T elemento) {
    Nodo nodoActual = this.cabeza;  // 2

    while (nodoActual != null) {  // 2
        if (nodoActual.elemento.equals(elemento)) {  // 2
            return true;
        }
        nodoActual = nodoActual.siguiente;  // 2
    }
    return false;
}
```

$$T(n) = 2n + 6$$

$$M(n) = 1$$

- **Método darLongitud.**

```java
public int darLongitud() {
    return this.longitud;  // 1
}
```

$$T(n) = 1$$

$$M(n) = 0$$

Pseudocódigos

- **Método constructor.**

```
1       function ListaLigadaSimple
2           null -> cabeza
3           null -> rabo
4           0 -> longitud
5       end function
```

- **Método insertar**
- **Entrada:** Dato de tipo T a ser agregado al inicio de la lista.
- **Salida:** Lista actualizada con el nuevo elemento agregado al principio.

```
1       function insertar
2           new Nodo -> nodoNuevo
3           if (cabeza == null)
4               nuevoNodo -> cabeza
5               nuevoNodo -> rabo
6           end if
7           else
8               cabeza.modificaSiguiente
9               nuevoNodo -> cabeza
10          end else
11          longitud+1 -> longitud
12      end function
```

- **Método eliminar**
- **Entrada:** Índice i del nodo a ser eliminado, i número natural.
- **Salida:** Lista actualizada después de eliminar el nodo en la posición i, si el índice es válido. En caso de índice inválido, se lanza una excepción.

```
1     function eliminar(i)
2         if i < 0 OR i >= longitud entonces
3             throw "Indice no valido"
4         end if
5         Nodo nodoActual <- cabeza
6         Nodo nodoAnterior <- null
7         for j <- 0 to i - 1 do
8             nodoAnterior <- nodoActual
9             nodoActual <- nodoActual.siguiente
10        end for
11        if nodoAnterior = null entonces
12            cabeza <- nodoActual.siguiente
13        else
14            nodoAnterior.siguiente <- nodoActual.siguiente
15            if nodoActual.siguiente = null entonces
16                rabo <- nodoAnterior
17            end if
18        end if
19        longitud <- longitud - 1
20    end function
```

- **Método acceder.**
- **Entrada:** Índice i del nodo al que se desea acceder, i número natural.
- **Salida:** Elemento del nodo en la posición i, si el índice es válido. En caso de índice inválido, se lanza una excepción.

```
1     function acceder(i)
2         if i < 0 OR i >= longitud entonces
3             throw "Indice no valido"
4         end if
5         Nodo nodoActual <- cabeza
6         for j <- 0 to i - 1 do
7             nodoActual <- nodoActual.siguiente
8         end for
9         return nodoActual.elemento
10    end function
```

- **Método buscar.**
- **Entrada:** Elemento de tipo T que se busca en la lista.
- **Salida:** `true` si el elemento está presente en la lista, `false` si no lo está.

```
1     function buscar(T)
2         Nodo nodoActual <- cabeza
3             while nodoActual != null do
4                 if nodoActual.elemento = T entonces
5                     return true
6                 end if
7                 nodoActual <- nodoActual.siguiente
8             end while
9         return false
10    end function
```

- **Método darLongitud.**
- **Entrada:** La lista en sí misma, llamada L.
- **Salida:** Número de elementos en la lista, es decir, su longitud.

```
1       function darLongitud(L)
2           return L.longitud
3       end function
```