Laboratorio No. 4: Implementación de Listas Enlazadas

Nicolás Restrepo López
Universidad Eafit
Medellín, Colombia
nrestrepol@eafit.edu.co

Mateo Restrepo Sierra
Universidad Eafit
Medellín, Colombia
mrestrepos@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos.

1.1. Implementen los métodos insertar un elemento en una posición determinada, borrar un dato en una posición determinada y verificar si un dato está en la lista en la clase *LinkedListMauricio*, teniendo en cuenta que:

- **a)** Si desea un reto moderado, implemente una lista simplemente enlazada.
- **b)** Si desea un reto mayor, implemente una lista doblemente enlazada.
- **c)** Si desea un gran reto, implemente una lista circular doblemente enlazada.

Véase Anexos: Node, LinkedListMauricio, Prueba.

1.2. En el código entregado por el profesor está el método *get* y sus pruebas. Teniendo en cuenta esto, realicen tres pruebas unitarias para cada método. La idea es probar que su implementación de los métodos *insert* y *remove* funcionan correctamente por los menos en los siguientes casos:

- Cuando vamos a insertar/borrar y la lista está vacía.
- Cuando vamos insertar/borrar el primer el elemento
- Cuando vamos a insertar/borrar el último elemento.

Véase Anexo: LinkedListMauricioTest.

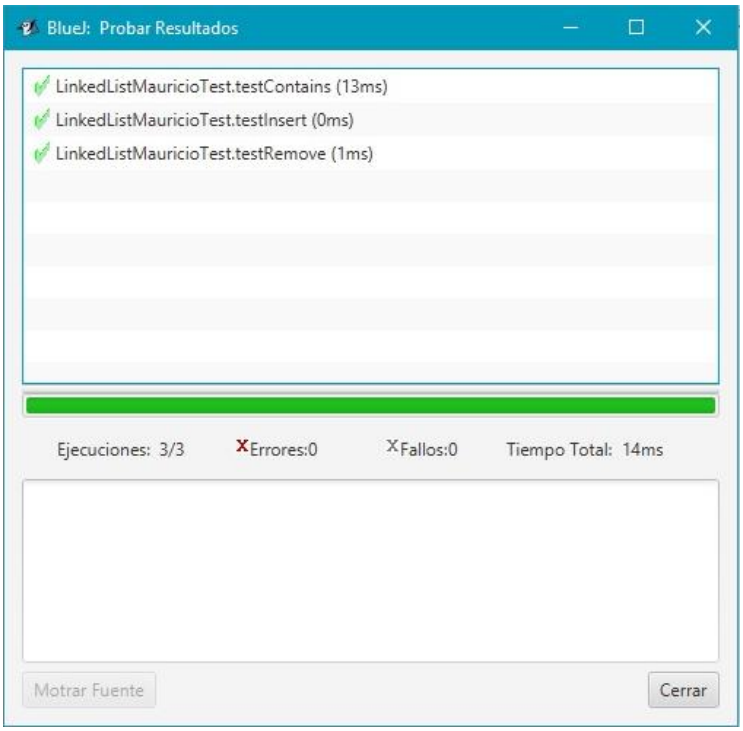
1.3. En un banco hay cuatro filas de clientes y solamente dos cajeros. Se necesita simular cómo son atendidos los clientes por los cajeros. Si lo desean, usen su implementación de listas enlazadas; de lo contrario, use la del API de Java.

Véase Anexo: Banco.

2.1. Resuelvan el ejercicio planteado usando pilas.

Véase Anexo: MundoBloques.

3.1. Verifiquen utilizando JUnit que todas las pruebas del numeral 1.2 pasan.



Véase Anexo: LinkedListMauricioTest.

3.2. Expliquen con sus propias palabras cómo funciona la implementación del ejercicio 2.1.

```
import java.util.ArrayList;
import java.util.Scanner;
import java.util.Stack;
public class MundoBloques
{
   public static Stack<Integer> search(ArrayList<Stack<Integer>> lista, int a)
   {
      Stack<Integer> aux = new Stack<Integer>();          //C1
      for (int i = 0; i < lista.size(); i++)              //C2 * n
      {
         if (lista.get(i).search(a) != -1)                //(C3 + O(1)) * n
            aux = lista.get(i);                           //C4
        }
        return aux;                                       //C5
    }
```

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

```
public static void moveOver(ArrayList<Stack<Integer>> lista, int a, int b)
{
   Stack<Integer> aux = new Stack<Integer>();          //C1
   Stack<Integer> aux1 = search(lista, a);             //C2 + O(n)
   Stack<Integer> aux2 = search(lista, b);             //C3 + O(n)
   if (aux1 != aux2)                                   //C4
   {
      while (aux1.peek() != a)                         //C5 * m
         aux.push(aux1.pop());                         //(C6 + C7) * m
      aux2.push(aux1.pop());                           //C8 + C9
      while (!aux.empty())                             //C10 * m
         aux1.push(aux.pop());                         //(C11 + C12) * m
   }
}
public static void moveOnto(ArrayList<Stack<Integer>> lista, int a, int b)
{
   Stack<Integer> aux = new Stack<Integer>();          //C1
   Stack<Integer> aux1 = new Stack<Integer>();         //C2
   Stack<Integer> aux2 = search(lista, a);             //C3 + O(n)
   Stack<Integer> aux3 = search(lista, b);             //C4 + O(n)
   if (aux2 != aux3)                                   //C5
   {
      while (aux2.peek() != a)                         //C6 * k
         aux.push(aux2.pop());                         //(C7 + C8) * k
      while (aux3.peek() != b)                         //C9 * i
         aux1.push(aux3.pop());                        //(C10 + C11) * i
      aux3.push(aux2.pop());                           //C12 + C13
      while (!aux.empty())                             //C14 * k
         aux2.push(aux.pop());                         //(C15 + C16) * k
      while (!aux1.empty())                            //C17 * i
         aux3.push(aux1.pop());                        //(C18 + C19) * i
   }
}
public static void pileOnto(ArrayList<Stack<Integer>> lista, int a, int b)
{
   Stack<Integer> aux = new Stack<Integer>();          //C1
   Stack<Integer> aux1 = new Stack<Integer>();         //C2
   Stack<Integer> aux2 = search(lista, a);             //C3 + O(n)
   Stack<Integer> aux3 = search(lista, b);             //C4 + O(n)
   if (aux2 != aux3)                                   //C5
   {
      while (aux2.peek() != a)                         //C6 * j
         aux.push(aux2.pop());                         //(C7 + C8) * j
      aux.push(aux2.pop());                            //C9 + C10
      while (aux3.peek() != b)                         //C11 * w
         aux1.push(aux3.pop());                        //(C12 + C13) * w
      while (!aux.empty())                             //C14 * j
         aux3.push(aux.pop());                         //(C15 + C16) * j
      aux3.push(aux2.pop());                           //C17 + C18
      while (!aux1.empty())                            //C19 * w
         aux3.push(aux1.pop());                        //(C20 + C21) * w
     }
 }
```

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

```
public static void pileOver(ArrayList<Stack<Integer>> lista, int a, int b)
{
   Stack<Integer> aux = new Stack<Integer>();          //C1
   Stack<Integer> aux1 = search(lista, a);             //C2 + O(n)
   Stack<Integer> aux2 = search(lista, b);             //C3 + O(n)
   if (aux1 != aux2)
   {
      while (aux1.peek() != a)                         //C4 * s
         aux.push(aux1.pop());                         //(C5 + C6) *s
      aux.push(aux1.pop());                            //C7 + C8
      while (!aux.empty())                             //C9 * s
         aux2.push(aux.pop());                         //(C10 + C11) * s
   }
}
public static void main (String [] args)
{
   Scanner sc = new Scanner(System.in);                //C1
   System.out.println("Ingrese el mundo de los bloques: ");
                                                       //C2
   int tam = sc.nextInt();                             //C3
   ArrayList<Stack<Integer>> lista = new ArrayList<Stack<Integer>>();
                                                       //C4
   for (int i = 0; i < tam; i++)                       //C5 * n
   {
      lista.add(new Stack<Integer>());                 //C6 * n * n
      lista.get(i).push(i);                            //(C7 + C8) * n
      System.out.println("Cola " + i + ": " + i);      //C9 * n
   }
   sc = new Scanner(System.in);                        //C10
   while(true)                                         //C11 * x
   {
      String command = sc.nextLine();                  //C12 * x
      if (command.equals("quit"))                      //C13 * x
      {
         sc.close();                                   //C14
         break;                                        //C15
      }
      String [] commands = command.split("\\s+");      //(C16 + C17) * x
      String verb = commands[0];                       //(C18 + C19) * x
      int a = Integer.parseInt(commands[1]);           //(C20 + C21) * x
      String prep = commands[2];                       //(C22 + C23) * x
      int b = Integer.parseInt(commands[3]);           //(C24 + C25) * x
      if (verb.equals("move"))                         //C26 * x
      {
         if (prep.equals("over"))                      //C27 * x
         {
            moveOver(lista, a, b);                     //O(m+n) * x
         }
         if (prep.equals("onto"))                      //C28 * x
         {
            moveOnto(lista, a, b);                     //O(k+i+n) * x
         }
      }
      else if (verb.equals("pile"))                    //C29 * x
```

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

```
            {
               if (prep.equals("over"))                    //C30 * x
               {
                  pileOver(lista, a, b);                   //O(s+n) * x
               }
               if (prep.equals("onto"))                    //C31 * x
               {
                  pileOnto(lista, a, b);                   //O(j+w+n) * x
               }
            }
         }
         for (int i = 0 ; i < lista.size(); i++)           //C32 * n
         {
            String acum = "";                              //C33 * n
            acum += "Cola " + i + ": ";                    //C34 * n
            Stack stack = lista.get(i);                    //(C35 + O(1)) * n
            Stack aux = new Stack();                       //C36 * n
            while(!stack.empty())                          //C37 * n * z
               aux.push(stack.pop());                      //(C38 + C39) * n * z
            while(!aux.empty())                            //C40 * n * z
            {
               int x = (int) aux.pop();                    //(C41 + C42) * n * z
               acum += x + " ";                            //C43 * n * z
            }
            System.out.println(acum);                      //C44 * n
         }
      }
   }
```

El Código consta de cuatro métodos principales que son *moveOver, moveOnto, pileOver* y *pileOnto,* los cuales establecen la manera cómo deben ser movidos los bloques. Pero antes de eso, se tiene un método auxiliar llamado *search,* y lo que hace básicamente es, mediante el ya establecido método *search* en pilas de Java, examina cada una de las pilas que conforman el arreglo. El método original debe retornar la posición de la pila en la cual está un elemento dado, y de no estar, retorna -1. Así que para el nuevo método se pasa como parámetro el arreglo de pilas y el elemento a buscar. Se abre un ciclo que examina cada pila, y de retornar algo diferente a -1, se entiende que el elemento está presente en dicha pila. El nuevo método *search* retornará la pila en la cual está presente el elemento.

Ahora, para moveOver se crean tres pilas auxiliares: una temporal y dos que serán las que contienen los números que se reciben como parámetro. Luego, de la pila donde está a, saco todos los elementos que estén encima de este mediante un ciclo, y luego agrego el elemento a en la pila donde está b. Finalmente vacío la pila temporal mediante un segundo ciclo, y los elementos que salen son agregados nuevamente a la pila donde originalmente estaba a.

Para moveOnto el proceso es similar, solo que ahora se tiene una cuarta pila auxiliar que será un temporal para agregar los elementos de la pila donde está b. Se inicia de la misma manera, solo que antes de agregar el elemento a en la pila donde está b, de esta deben vaciarse los elementos que estén sobre b mediante un ciclo. Los elementos se añaden al segundo temporal. Luego de agregar a, se vacía el segundo temporal en la pila donde está b, y se procede a hacer lo mismo con el primer temporal en la pila donde inicialmente estaba a.

Con pileOver y pileOnto es muy similar a moveOver y moveOnto, respectivamente. En el primer caso se procede de manera idéntica, solo que al final los elementos que estaban en la pila temporal no se añadirán a la pila donde inicialmente estaba a, sino donde está b. En el segundo caso ocurre lo mismo, solo que antes de vaciar el segundo temporal en la pila donde está b.

Cabe recordar que en caso de que los elementos a y b estén localizados en la misma pila, no se realiza nada. Claramente esto incluye al caso a = b, ya que obligatoriamente tendrían que estar en la misma pila.

El método principal crea un *Scanner* e imprime un texto indicando que debe escribirse el tamaño del arreglo. Luego se inicializa el arreglo del tamaño introducido, al cual se le insertan las pilas mediante un ciclo, y posteriormente se imprimen para que el usuario sepa cómo estaba organizado desde un principio. Después se crea un segundo *Scanner* que va a leer los comandos introducidos que controlan los movimientos de los bloques. Allí, se descomponen los comandos en verbo y preposición; el primero indica si se debe mover el bloque o la pila y el segundo indica si se coloca el bloque sobe la pila o sobre el segundo bloque. Dependiendo de los comandos se llaman los diferentes métodos. Adicionalmente en caso de que el comando sea *quit,* se rompe el ciclo y se cierra el *Scanner.* Finalmente hay un último ciclo que imprime el arreglo ya movido.

3.3. Calculen la complejidad del ejercicio realizado en el numeral 2.1 y agréguenla al informe PDF.

En el punto anterior tenemos ya el algoritmo con sus complejidades comentadas, así que simplemente sumaremos las complejidades del método principal. Obtenemos lo siguiente:

$$T(n) = C1 + C2 + C3 + C4 + C5 *n + C6 * n * n +(C7 + C8) * n + C9 * n + C10 + C11 * x + C12 * x + C13 * x + (C16 + C17) * x + (C18 + C19) * x + (C20 + C21) * x + (C22 + C23) * x + (C24 + C25) * x \ \ C26 * x + C29 * x + C30 * x + C31 * x + O(j+w+n) * x + C32 * n + C33 * n + C34 * n + (C35 + O(1)) * n + C36 * n + C37 * n * z + C40 * n * z + (C41 + C42) * n * z + C43 * n * z + C44 * n$$

$$T(n) = C_1 + C_2 * n + C_3 * x + C_4 * z * n + O(j+w+n) + C * n^2$$

$$T(n) = O(C_1 + C_2 * n + C_3 * x + C_4 * z * n + O(j+w+n) + C * n^2)$$

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

$T(n) = O(C_3 * x + C_4 * z * n + O(j+w+n) + C * n^2)$, por Regla de la Suma
$T(n) = O(x + z * n + j + w + n^2)$, por Regla del Producto

De ese modo, tenemos que la complejidad asintótica del algoritmo MundoBloques es $O(x + z * n + j + w + n^2)$. Podría resumirse diciendo que la complejidad asintótica del algoritmo es del orden $O(n^2)$.

3.4. Expliquen con sus palabras qué son las variables m y n del cálculo de complejidad del numeral 3.3.

En este algoritmo no solo existen variables m y n, sino muchas más (i, j, k, s, w, x, z). Principalmente la variable n representa el tamaño del arreglo de pilas, valor que no cambia a lo largo del algoritmo porque cada método recibe como parámetro el mismo arreglo. No obstante, las demás variables todas representan un mismo principio, y es la longitud de una pila en específico. Dado que no se tiene certeza con cuál método se comienzan los movimientos, se establecen diferentes valores para los tamaños de las pilas; esto además porque no se sabe si los tamaños de varias pilas puedan ser iguales.

4) Simulacro de Parcial.

1. El siguiente es un algoritmo invierte una lista usando como estructura auxiliar una pila:

```
01 public static LinkedList <String> invertir (LinkedList <String> lista){
02 Stack <String> auxiliar = new Stack <String>();
03 while(________ > 0){
04 auxiliar.push(lista.removeFirst());
05 }
06 while(auxiliar.size() > 0){
07 _____________
08 }
09 return lista;
10 }
```

De acuerdo a lo anterior, responda las siguientes preguntas:

a) ¿Qué condición colocaría en el ciclo *while* de la línea 3? (10%)

lista.size()

b) Complete la línea 7 de forma que el algoritmo tenga sentido (10%)

lista.add(auxiliar.pop())

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

2. En un banco, se desea dar prioridad a las personas de edad avanzada. El ingeniero ha propuesto un algoritmo para hacer que la persona de mayor edad en la fila quede en primer lugar.

Para implementar su algoritmo, el ingeniero utiliza colas. Las colas en Java se representan mediante la interfaz Queue. Una implementación de Queue es la clase LinkedList.

El método *offer* inserta en una cola y el método *poll* retira un elemento de una cola.

```
01 public Queue<Integer> organizar(Queue<Integer> personas){
02 int mayorEdad = 0;
03 int edad;
04 Queue<Integer> auxiliar1 =new LinkedList<Integer>();
05 Queue<Integer> auxiliar2 =new LinkedList<Integer>();
06 while(personas.size() > 0){
07 edad = personas.poll();
08 if(edad > mayorEdad) mayorEdad = edad;
09 auxiliar1.offer(edad);
10 auxiliar2.offer(edad);
11 }
12 while(_______){
13 edad = auxiliar1.poll();
14 if(edad == mayorEdad) personas.offer(edad);
15 }
16 while(_______){
17 edad = auxiliar2.poll();
18 if(edad != mayorEdad) ________;
19 }
20 return personas;
21 }
```

a) ¿Que condiciones colocaría en los 2 ciclos while de líneas 12 y 16, respectivamente?

12: auxiliar1.poll()>0

16: auxiliar2.poll()>0

b) Complete la línea 18 con el objeto y el llamado a un método, de forma que el algoritmo tenga sentido

personas.offer(auxiliar.pop());

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

3. ¿Cuál es la complejidad asintótica, para el peor de los casos, de la función procesarCola(q, n)?

```
01 public void procesarCola(Queue q, int n)
02 for (int i = 0; i < n; i++)     //C1 + C2*n
03 for(int j = 0; j < n; j++)      //(C3 + C4*n)*n
04 q.add(j);                       //C5*n*n
05 //Hacer algo en O(1)

//T(n) = C1 + C2*n + (C3 + C4*n)*n + C5*n*n
//T(n) = C' + C''*n + C*n^2
//O(C''*n + C*n^2)
//O(C*n^2)
//O(n^2)
```

a) O(n)
b) O (|q|), donde |q| es el número de elementos de q
c) $O(n^2)$
d) $O(2^n)$

En Java, el método add agrega un elemento al comienzo de una cola.

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co