TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE OAXACA

MATERIA:

ESTRUCTURA DE DATOS

CLAVE DE MATERIA:

SCD1007

CARRERA:

INGENIERÍA EN SISTEMAS COMPUTACIONALES

“PROYECTO TERCERA UNIDAD”.

PRESENTA:

ING. JOSE SEBASTIAN JAFET

NÚMERO DE CONTROL:

22161112

NOMBRE DEL CATEDRÁTICO:

MC. SILVA MARTINEZ DALIA

GRUPO: **3SB**

OAXACA DE JUÁREZ, OAXACA A 07 DE ABRIL DEL 2024

ÍNDICE

INTRODUCCIÓN

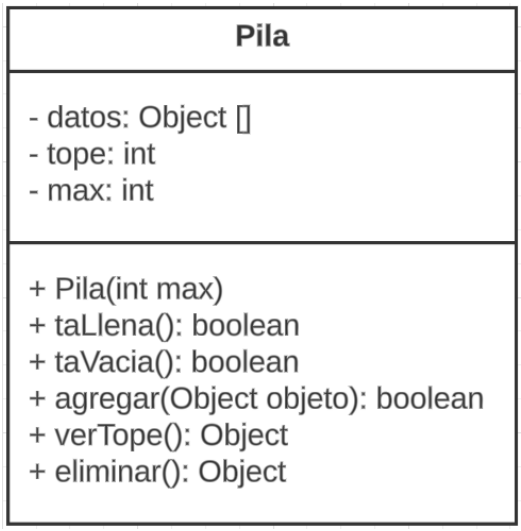
En este proyecto veremos como funcionan cada una de las estructuras de datos mas demandadas en nuestra época, debemos ser conscientes de la importancia y la relevancia de estos, ya que nuestra época esta vista principalmente en el avance de la tecnología y como esta interactúa con otras, el funcionamiento de cada uno de estos esta específicamente hecho para cada situación que lo requiera.

También veremos algunos ejemplos de como podremos implementar cada una de las estructuras de datos.

Sin más que decir, ¡vamos allá!

ESTÁTICOS O EN ARREGLOS:

- **Pila:**
  - **UML:**



  - **Código:**

```
public class Pila
{
    private Object datos[];
    private int tope, max;
    public Pila(int max){
        this.max=max;
        tope=-1;
        datos=new Object[this.max];
    }
    public boolean taLlena(){return tope==max-1;}
    public boolean taVacia(){return tope==-1;}
    public boolean agregar(Object objeto){
        if(!taLlena()){
            tope++;
            datos[tope]=objeto;
            return true;
        }
```

```
        return false;
    }
    public Object verTope(){
        if(!taVacia()){
            return datos[tope];
        }
        return null;
    }
    public Object eliminar(){
        if(!taVacia()){
            Object aux=datos[tope];
            tope--;
            return aux;
        }
        return null;
    }
}
```

- **API:**

  1. Constructor **Pila(int max)**:

  Este constructor inicializa una pila con un tamaño máximo dado. Inicializa el tope de la pila en -1 y crea un arreglo de objetos “datos” con el tamaño máximo especificado.

  2. Método **taLlena()**:

  Este método verifica si la pila está llena comparando si el tope de la pila es igual al índice máximo permitido “max – 1”. Devuelve “true” si la pila está llena, de lo contrario, devuelve “false”.

  3. Método **taVacia()**:

  Este método verifica si la pila está vacía comparando si el tope de la pila es igual a -1. Devuelve “true” si la pila está vacía, de lo contrario, devuelve “false”.

  4. Método **agregar(Object objeto)**:

Este método agrega un objeto a la pila. Verifica si la pila no está llena antes de agregar el objeto. Si la pila no está llena, incrementa el tope y agrega el objeto en la posición correspondiente en el arreglo de datos. Devuelve “true” si se agrega correctamente, de lo contrario, devuelve “false”.

5. Método **verTope()**:

Este método devuelve el objeto en la cima de la pila sin eliminarlo. Verifica si la pila no está vacía antes de devolver el objeto en la posición del tope del arreglo de datos. Devuelve el objeto en la cima de la pila o “null” si la pila está vacía.

6. Método **eliminar()**:

Este método elimina y devuelve el objeto en la cima de la pila. Verifica si la pila no está vacía antes de eliminar el objeto. Devuelve el objeto en la posición del tope del arreglo de datos y decrementa el tope en uno. Si la pila está vacía, devuelve “null”.

- **Ejemplo:**

Descripción

Escribe un programa que comience con una pila P de enteros inicialmente vacía y que ejecute una lista de las siguientes operaciones:
AGREGA v: Agregar v a P. Por ejemplo, si P = (3,1,4) y v=5 entonces P=(3,1,4,5).
CONSUME: Quitar los dos últimos elementos de P, calcular su suma y agregarla a P. Por ejemplo, si P=(3,1,4) entonces P=(3,5).
IMPRIME: Imprimir el último elemento de P. Por ejemplo, si P=(3,1,4) entonces se debe imprimir 4.

Entrada

Un entero N seguido de las N operaciones a realizar sobre P. Puedes suponer que 0<=N<=10^5 y que nunca se pedirá consumir o imprimir un elemento que no existe.

Salida

Cada para operación de impresión, el valor del entero correspondiente.

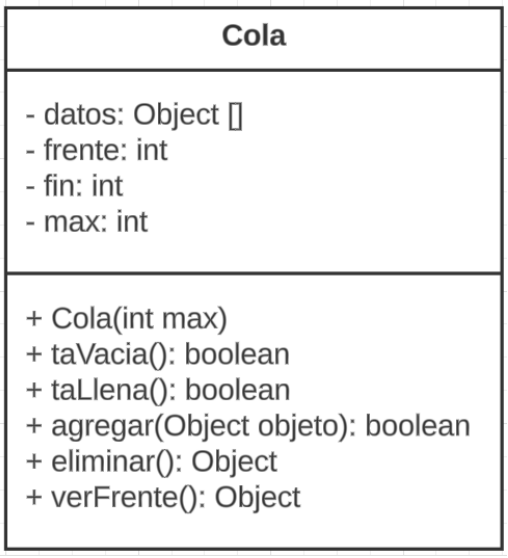
```
import java.util.Scanner;
public class Main {
    private int tope, max, pila[];
    public Main(int maximo) {
        max = maximo;
        tope = -1;
        pila = new int[max];
    }
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        String quevaser;
        int limite = entrada.nextInt();
        entrada.nextLine();
        int agregar = 0;
        Main objeto = new Main(limite);
        for(int c=0;c < limite;c++){
            quevaser = entrada.next();
            switch(quevaser){
                case "AGREGA":
                    agregar = entrada.nextInt();
                    objeto.agrega(agregar);
                break;
                case "CONSUME":
                    objeto.consume();
                break;
                case "IMPRIME":
                    System.out.println(objeto.imprime());
                break;
            }
        }
        entrada.close();
```

```
}
public boolean tavacia() {
    return tope == -1;
}
public boolean tallena() {
    return tope == max - 1;
}
public boolean agrega(int o){
    if (!tallena()) {
        tope++;
        pila[tope] = o;
        return true;
    }
    return false;
}
public int consume() {
    int o = 0;
    if (!tavacia() && tope >= 1) {
        o = pila[tope] + pila[tope - 1];
        tope--;
        pila[tope] = o;
    }
    return o;
}
public int imprime() {
    if (!tavacia()) {
        return pila[tope];
    } else {
        return 0;
    }
}
```

}

➢ Cola:

○ UML:



○ Código:

```
public class Cola
{
    private Object [] datos;
    int frente,fin,max;
    public Cola(int max)
    {
        this.max=max;
        datos = new Object [max];
        fin=-1;
        frente=0;
    }
    public boolean taVacia(){return fin==-1;}
    public boolean taLlena(){return fin==max-1;}
    public boolean agregar(Object objeto){
        if(!taLlena()){
            fin++;
            datos[fin]=objeto;
```

```
            return true;
        }
        return false;
    }
    public Object eliminar(){
        if(!taVacia()){
            Object aux=datos[frente];
            for(int i=0;i<fin;i++){
                datos[i]=datos[i+1];
            }
            fin--;
            return aux;
        }
        return null;
    }
    public Object verFrente(){
        if(!taVacia()){
            return datos[frente];
        }
        return null;
    }
}
```

- **API:**

  1. Constructor **Cola(int max)**:

  Este constructor inicializa una cola con un tamaño máximo dado. Inicializa el frente de la cola en 0, el final en -1 y crea un arreglo de objetos “datos” con el tamaño máximo especificado.

  2. Método **taVacia()**:

  Este método verifica si la cola está vacía comparando si el índice del final (“fin”) es -1. Devuelve `true` si la cola está vacía, de lo contrario, devuelve “false”.

3. Método **taLlena()**:

Este método verifica si la cola está llena comparando si el índice del final (“fin”) es igual al índice máximo permitido (“max – 1”). Devuelve `true` si la cola está llena, de lo contrario, devuelve “false”.

4. Método **agregar(Object objeto)**:

Este método agrega un objeto al final de la cola. Verifica si la cola no está llena antes de agregar el objeto. Si la cola no está llena, incrementa el índice del final (“fin”) y agrega el objeto en la posición correspondiente en el arreglo de datos. Devuelve `true` si se agrega correctamente, de lo contrario, devuelve “false”.

5. Método **eliminar()**:

Este método elimina y devuelve el objeto en el frente de la cola. Verifica si la cola no está vacía antes de eliminar el objeto. Desplaza todos los elementos en el arreglo de datos una posición hacia adelante para simular la eliminación del objeto en el frente. Luego, decrementa el índice del final (“fin”). Devuelve el objeto en el frente de la cola o “null” si la cola está vacía.

6. Método **verFrente()**:

Este método devuelve el objeto en el frente de la cola sin eliminarlo. Verifica si la cola no está vacía antes de devolver el objeto en la posición del frente del arreglo de datos. Devuelve el objeto en el frente de la cola o “null” si la cola está vacía.

- **Ejemplo:**

  Descripción

  Menganito tiene muy mala memoria, siempre que le presentan a una persona olvida facilmente los nombres, por lo que necesita crear un programa que sea capaz de recordar los nombres y los apellidos de cada persona que le presentan. A menganito quiere aprenderse primero los nombres y luego los apellidos de las personas, por lo que necesita una lista de nombres primero y luego una de apellidos.

  Entrada

Un entero n que representa cuantas personas le van apresentar a Menganito, seguido de n líneas con los nombres de las personas. Cada persona solo tiene un nombre y un apellido y estos aparecen separados por un espacio.No se aceptan letras acentuadas ni caracteres especiales.

Salida

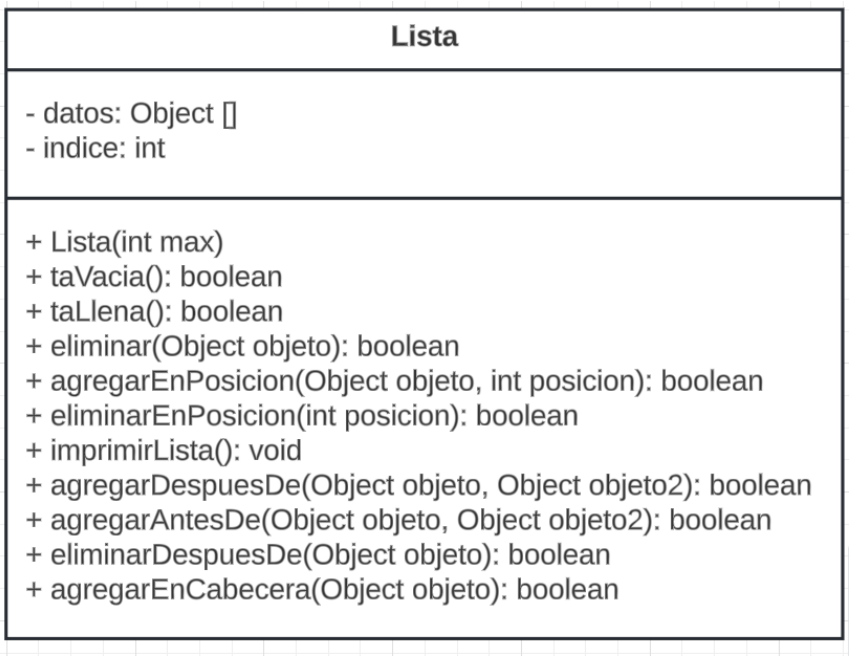
la lista de los nombres en n líneas, seguida por la lista de los apellidos también en n líneas.

```
import java.util.Scanner;
import java.util.ArrayList;
class Nombres1 {
    private ArrayList<String> objeto;
    private int f, fin, max;
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        int contador = entrada.nextInt();
        int almacenado = 0;
        String almacenar;
        Nombres1 nombre = new Nombres1(contador);
        Nombres1 apellido = new Nombres1(contador);
        entrada.nextLine();
        while (almacenado < contador) {
            almacenar = entrada.next();
            nombre.agregar(almacenar);
            almacenar = entrada.next();
            apellido.agregar(almacenar);
            almacenado++;
        }
        for (int i = contador; i >= 0; i--) {
            System.out.println(nombre.eliminar());
```

```
    }
    for (int i = contador; i >= 0; i--) {
        System.out.println(apellido.eliminar());
    }
    entrada.close();
}
public  Nombres1(int max) {
    this.max = max;
    objeto = new ArrayList<>(max);
    fin = -1;
    f = 0;
}
public boolean tavacia() {
    return fin == -1;
}
public boolean tallena() {
    return fin == max - 1;
}
public boolean agregar(String o) {
    if (!tallena()) {
        fin++;
        objeto.add(fin, o);
        return true;
    }
    return false;
}
public String eliminar() {
    if (!tavacia()) {
        String aux = objeto.get(f);
        f++;
        fin--;
```

```
            return aux;
        }
        return "";
    }
    public String verFrente() {
        if (!tavacia()) {
            return objeto.get(f);
        }
        return "";
    }
}
```

- **Lista:**
  - **UML:**



  - **Código:**

```
public class Lista {
    private Object datos[];
    private int indice;
    public Lista(int max) {
        indice = 0;
```

```
    datos = new Object[max];
}
public boolean taLlena() {
    return indice >= datos.length;
}
public boolean taVacia() {
    return indice == 0;
}
public boolean eliminar(Object objeto) {
    for (int i = 0; i < indice; i++) {
        if (datos[i].equals(objeto)) {
            for (int j = i; j < indice - 1; j++) {
                datos[j] = datos[j + 1];
            }
            datos[indice - 1] = null;
            indice--;
            return true;
        }
    }
    return false;
}
public boolean agregarEnPosicion(Object objeto, int posicion) {
    if (taLlena() || posicion < 0 || posicion >= datos.length) {
        return false;
    }
    for (int i = indice; i > posicion; i--) {
        datos[i] = datos[i - 1];
    }
    datos[posicion] = objeto;
    indice++;
    return true;
```

```
}
public boolean eliminarEnPosicion(int posicion) {
    if (taVacia() || posicion < 0 || posicion >= indice) {
        return false;
    }
    for (int i = posicion; i < indice - 1; i++) {
        datos[i] = datos[i + 1];
    }
    datos[indice - 1] = null;
    indice--;
    return true;
}
public void imprimirLista() {
    for (int i = 0; i < indice; i++) {
        System.out.println(datos[i]);
    }
}
public boolean agregarDespuesDe(Object objeto, Object objeto2) {
    for (int i = 0; i < indice; i++) {
        if (datos[i].equals(objeto)) {
            if (taLlena()) {
                return false;
            }
            for (int j = indice; j > i + 1; j--) {
                datos[j] = datos[j - 1];
            }
            datos[i + 1] = objeto2;
            indice++;
            return true;
        }
    }
```

```
    return false;
}
public boolean agregarAntesDe(Object objeto, Object objeto2) {
    for (int i = 0; i < indice; i++) {
        if (datos[i].equals(objeto)) {
            if (taLlena()) {
                return false;
            }
            for (int j = indice; j > i; j--) {
                datos[j] = datos[j - 1];
            }
            datos[i] = objeto2;
            indice++;
            return true;
        }
    }
    return false;
}
public boolean eliminarDespuesDe(Object objeto) {
    for (int i = 0; i < indice - 1; i++) {
        if (datos[i].equals(objeto)) {
            if (i == indice - 1) {
                return false;
            }
            for (int j = i + 1; j < indice - 1; j++) {
                datos[j] = datos[j + 1];
            }
            datos[indice - 1] = null;
            indice--;
            return true;
        }
```

```
        }
        return false;
    }
    public boolean agregarEnCabecera(Object objeto) {
        if (taLlena()) {
            return false;
        }
        for (int i = indice; i > 0; i--) {
            datos[i] = datos[i - 1];
        }
        datos[0] = objeto;
        indice++;
        return true;
    }
}
```

- **API:**

  1. Constructor **Lista(int max)**:

  Este constructor inicializa una lista con un tamaño máximo dado. Inicializa el índice en 0 y crea un arreglo de objetos “datos” con el tamaño máximo especificado.

  2. Método **taLlena()**:

  Este método verifica si la lista está llena comparando si el índice (“índice”) es mayor o igual al tamaño máximo del arreglo de datos. Devuelve “true” si la lista está llena, de lo contrario, devuelve “false”.

  3. Método **taVacia()**:

  Este método verifica si la lista está vacía comparando si el índice (“indice”) es igual a 0. Devuelve “true” si la lista está vacía, de lo contrario, devuelve “false”.

  4. Método **eliminar(Object objeto)**:

  Este método elimina la primera ocurrencia del objeto especificado en la lista. Itera sobre los elementos de la lista para encontrar la posición del

objeto. Desplaza los elementos hacia atrás para llenar el espacio dejado por el objeto eliminado. Reduce el índice (“índice”) y devuelve “true” si se elimina correctamente, de lo contrario, devuelve “false”.

5. Método **agregarEnPosicion(Object objeto, int posicion)**:

Este método agrega un objeto en una posición específica de la lista. Verifica si la lista está llena o si la posición es inválida. Desplaza los elementos hacia adelante para hacer espacio en la posición especificada. Inserta el objeto en la posición indicada y aumenta el índice (“índice”). Devuelve “true” si se agrega correctamente, de lo contrario, devuelve “false”.

6. Método **eliminarEnPosicion(int posicion)**:

Este método elimina el objeto en la posición especificada de la lista. Verifica si la lista está vacía o si la posición es inválida. Desplaza los elementos hacia atrás para llenar el espacio dejado por el objeto eliminado. Reduce el índice (“indice”) y devuelve “true” si se elimina correctamente, de lo contrario, devuelve “false”.

7. Método **imprimirLista()**:

Este método imprime todos los elementos de la lista en la consola.

8. Método **agregarDespuesDe(Object objeto, Object objeto2)**:

Este método agrega un objeto después de la primera ocurrencia del objeto especificado en la lista. Verifica si la lista está llena. Desplaza los elementos hacia adelante para hacer espacio. Inserta el nuevo objeto después del objeto especificado y aumenta el índice (“índice”). Devuelve “true” si se agrega correctamente, de lo contrario, devuelve “false”.

9. Método **agregarAntesDe(Object objeto, Object objeto2)**:

Este método agrega un objeto antes de la primera ocurrencia del objeto especificado en la lista. Verifica si la lista está llena. Desplaza los elementos hacia adelante para hacer espacio. Inserta el nuevo objeto antes del objeto especificado y aumenta el índice (“índice”). Devuelve “true” si se agrega correctamente, de lo contrario, devuelve “false”.

10. Método **eliminarDespuesDe(Object objeto)**:

Este método elimina el objeto que está después de la primera ocurrencia del objeto especificado en la lista. Devuelve “true” si se elimina correctamente, de lo contrario, devuelve “false”.

11. Método **agregarEnCabecera(Object objeto)**:

Este método agrega un objeto en la cabecera de la lista, desplazando todos los elementos hacia la derecha. Aumenta el índice (“índice”) y devuelve `true` si se agrega correctamente, de lo contrario, devuelve “false”.

- **Ejemplo:**

Escribe un programa que contenga un método que acepte como parámetro una lista de números enteros mayores que 0, pudiendo contener elementos duplicados. Este método debe sustituir cada valor repetido por 0. Para terminar, realiza un método muestre el array modificado. Nota: Necesitarás otro método para rellenar la lista de enteros. Le irá pidiendo números al usuario hasta que este introduzca un número negativo.

Ejemplo: 2 7 8 4 5 8 7 1 2 0 0 4 5 0 0 1

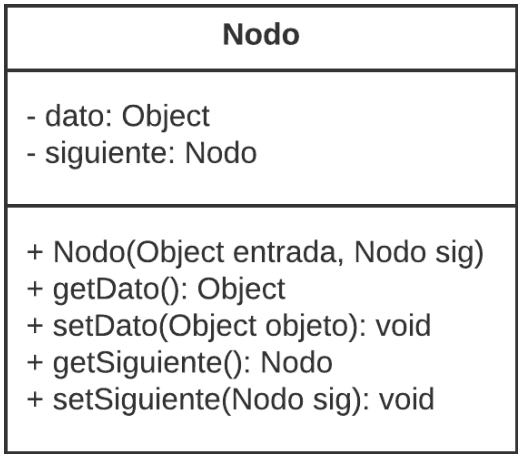
```
import java.util.ArrayList;
import java.util.Scanner;
public class SustituirPorCeros {
    public static void solicitarNumeros(ArrayList <Integer> listaNumeros) {
        Integer numero;
        boolean contador = true;
        String salir = "Si";
        Scanner teclado = new Scanner(System.in);
        System.out.println("------Lista de numeros Enteros----------");
        do {
            if (contador == true) {
                System.out.println("Introduce un numero mayor que 0 ");
                numero = teclado.nextInt();
```

```
                if (numero <=0) {
                    System.out.println("El numero no es correcto!!!");
                }else{
                    listaNumeros.add(numero);
                    contador = false;
                }
            }
            System.out.println("Introduce otro numero: ");
            numero = teclado.nextInt();
            if (numero <=0) {
                System.out.println("El numero no es correcto!!!");
            }else{
                listaNumeros.add(numero);
                contador = false;
            }
            System.out.println("Si quieres dejar de meter numeros escribe: \"Si\"");
            salir = teclado.next();
        } while (salir.equalsIgnoreCase("No"));
        //System.out.println(listaNumeros);
    }
    public static void rellenarDeCeros(ArrayList <Integer> listaNumeros) {
        ArrayList listaCopia = (ArrayList) listaNumeros.clone();
        for (int i = 0; i < listaNumeros.size(); i++) {
            for (int j = 0; j < listaCopia.size(); j++) {
                if ((i != j) && (listaNumeros.get(i) == listaCopia.get(j))) {
                    listaNumeros.set(i, 0);
                    listaNumeros.set(j, 0);
                }
            }
        }
```

```
        //System.out.println(listaNumeros);
    }
    public static void mostrarListas(ArrayList <Integer> listaNumeros) {
        System.out.println("------Lista de numeros Enteros----------");
        for (int i = 0; i < listaNumeros.size(); i++) {
            System.out.print(" " + listaNumeros.get(i) + " ");
        }
        System.out.println("");
    }
    public static void main(String[] args) {

        ArrayList <Integer> listaNumeros = new ArrayList <> ();
        solicitarNumeros(listaNumeros);
        mostrarListas(listaNumeros);
        rellenarDeCeros(listaNumeros);
        mostrarListas(listaNumeros);
    }

}
```

LIGADAS:

- **Nodo:**
  - **UML:**



- **Código:**

```
public class Nodo{
    private Object dato;
    private Nodo siguiente;
    public Nodo(Object entrada, Nodo sig){
        dato=entrada;
        siguiente=sig;
    }
    public Object getDato(){return dato;}
    public void setDato(Object objeto){dato=objeto;}
    public Nodo getSiguiente(){return siguiente;}
    public void setSiguiente(Nodo sig){this.siguiente=sig;}
}
```

- **API:**

1. Constructor **Nodo(Object entrada, Nodo sig)**:

Este constructor crea un nuevo nodo con un dato dado y una referencia al siguiente nodo especificado.

2. Método **getDato()**:

Este método devuelve el dato almacenado en el nodo.

3. Método **setDato(Object objeto)**:

Este método establece el dato almacenado en el nodo con el objeto especificado.

4. Método **getSiguiente()**:

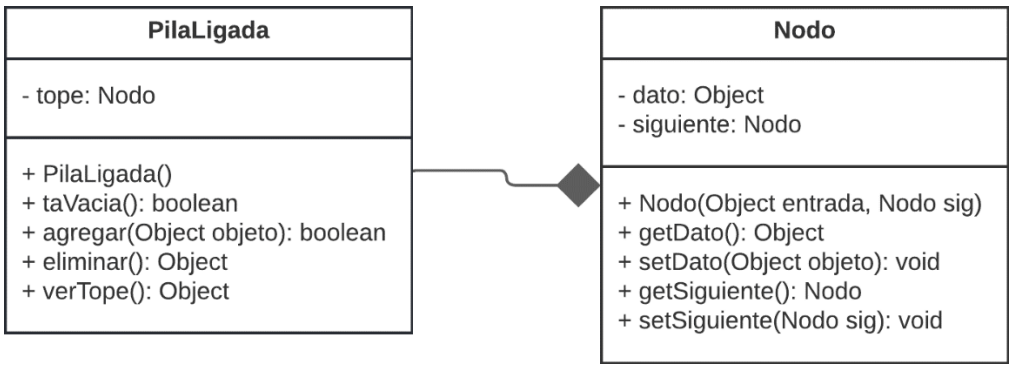
Este método devuelve la referencia al siguiente nodo en la lista.

5. Método **setSiguiente(Nodo sig)**:

Este método establece la referencia al siguiente nodo en la lista con el nodo especificado.

➢ Pila Ligada:

- **UML:**



- **Código:**

```
public class PilaLigada
{
    private Nodo tope;
    public PilaLigada(){
        tope=null;
    }
    public boolean taVacia(){
        return tope==null;
    }
    public boolean agregar(Object objeto){
        Nodo aux= new Nodo (objeto,tope);
        if(aux!=null){
            tope=aux;
            return true;
        }
        return false;
    }
    public Object eliminar(){
        Object aux=null;
        if(!taVacia()){
            aux=tope.getDato();
            tope=tope.getSiguiente();
        }
```

```
        return aux;
    }
    public Object verTope(){
        if(!taVacia()){
            return tope.getDato();
        }
        return null;
    }
}
```

- **API:**

  1. Constructor **PilaLigada()**:

  Este constructor inicializa una pila ligada vacía estableciendo el tope como “null”.

  2. Método **taVacia()**:

  Este método verifica si la pila está vacía comprobando si el tope es “null”. Retorna “true” si la pila está vacía, de lo contrario, retorna “false”.

  3. Método **agregar(Object objeto)**:

  Este método agrega un nuevo elemento a la pila. Crea un nuevo nodo con el objeto especificado y lo coloca en la cima de la pila. Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

  4. Método **eliminar()**:

  Este método elimina y devuelve el elemento en la cima de la pila. Primero verifica si la pila no está vacía. Si la pila no está vacía, guarda el dato del nodo en la cima, actualiza el tope de la pila al siguiente nodo y retorna el dato guardado. Si la pila está vacía, retorna “null”.

  5. Método **verTope()**:

  Este método devuelve el elemento en la cima de la pila sin eliminarlo. Verifica si la pila no está vacía y retorna el dato del nodo en la cima. Si la pila está vacía, retorna “null”.

- **Ejemplo:**

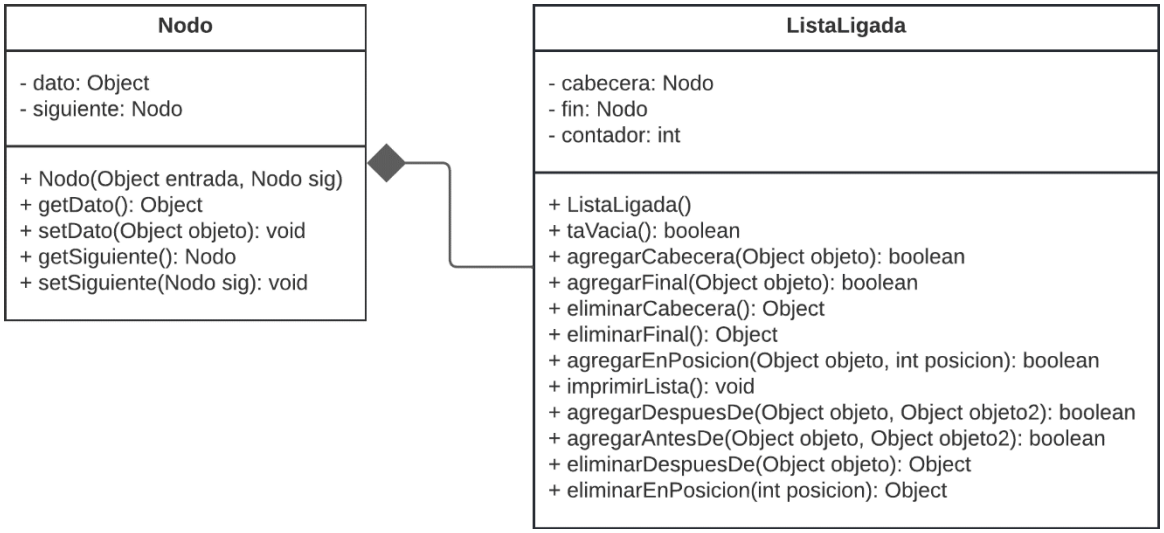
  Los últimos serán los primeros

Descripción

En una carrera en el pueblo de Yahualica todos los participantes creen que el que llegue primero será el ganador, pero no, las reglas son diferentes en Yahualica, el ganador es el que llega al último. Escribe un programa que dados los nombres de los concursantes como van cruzando la meta, imprima dichos nombres en orden de acuerdo al lugar que consiguieron en la competencia (El que llego al último obtiene el primer lugar, el penúltimo el segundo lugar, y así sucesivamente).

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Stack<String> pila = new Stack<>();
        String a="";
        while(!a.equalsIgnoreCase("#")){
            pila.push(a);
            a=sc.next();
        }
        while(!pila.isEmpty()){
            a=pila.peek();
            System.out.println(a);
            pila.pop();

        }

    }

}
```

- **Lista Ligada:**
  - **UML:**



- **Código:**

```
public class ListaLigada
{
    private Nodo cabecera, fin;
    private int contador;
    public ListaLigada(){
        cabecera=fin=null;
        contador=0;
    }
    public boolean taVacia(){
        return contador==0;
    }
    public boolean agregarCabecera(Object objeto){
        if(objeto!=null){
            if(cabecera==null){
                cabecera=fin=new Nodo(objeto, null);
                cabecera.setSiguiente(cabecera);
                contador++;
            }
            else{
                cabecera=new Nodo(objeto, cabecera);
```

```
            contador++;
        }
        return true;
    }
    else {return false;}
}
public boolean agregarFinal(Object objeto){
    if(objeto==null){return false;}
    if(cabecera==null){
        fin=cabecera=new Nodo(objeto, null);
        contador++;
        return true;
    }
    else{
        Nodo n=new Nodo(objeto, null);
        fin.setSiguiente(n);
        fin=n;
        contador++;
        return true;
    }
}
public Object eliminarCabecera(){
    Object objeto=null;
    if(cabecera!=null){
        objeto=cabecera.getDato();
        cabecera=cabecera.getSiguiente();
        if(cabecera==null){fin=null;}
        contador--;
    }
    return objeto;
}
```

```
public Object eliminarFinal(){
    Object objeto=null;
    if(cabecera==null){return null;}
    if(cabecera.getSiguiente()==null){
        objeto=cabecera.getDato();
        cabecera=fin=null;
    }
    else{
        Nodo auxiliar=cabecera;
        while(auxiliar.getSiguiente()!=fin){
            auxiliar=auxiliar.getSiguiente();
        }
        objeto=fin.getDato();
        fin=auxiliar;
        fin.setSiguiente(null);
    }
    contador--;
    return objeto;
}
public boolean agregarEnPosicion(Object objeto, int posicion){
    int cont=0;
    boolean proceso=false;
    if(objeto!=null){
        for(Nodo aux=cabecera; aux!=null; aux.getSiguiente()){
            if(cont==posicion){
                Nodo nuevo=new Nodo(objeto, aux.getSiguiente());
                aux.setSiguiente(nuevo);
                if(aux==fin){fin=nuevo;}
                proceso=true;
            }
            cont++;
```

```
            }
            contador++;
        }
        return proceso;
    }
    public void imprimirLista(){
        Nodo aux=cabecera;
        while(aux!=null){
            System.out.println(aux.getDato());
            aux=aux.getSiguiente();
        }
    }
    public boolean agregarDespuesde(Object objeto, Object objeto2){
        for(Nodo aux=cabecera; aux!=null; aux.getSiguiente()){
            if(aux.getDato().equals(objeto)){
                Nodo aux2=aux.getSiguiente();
                aux.setSiguiente(new Nodo(objeto2, aux2));
                contador++;
                return true;
            }
        }
        return false;
    }
    public boolean agregarAntesde(Object objeto, Object objeto2){
        for(Nodo aux=cabecera; aux!=null; aux=aux.getSiguiente()){

if(aux.getSiguiente().getDato().toString().compareTo(objeto2.toString())
==0){
                Nodo aux2=aux.getSiguiente();
                aux.setSiguiente(new Nodo(objeto, aux2));
                contador++;
```

```
                return true;
            }
        }
        return false;
    }
    public Object eliminarDespuesde(Object objeto){
        Object aux=null;
        for(Nodo aux2=cabecera; aux2!=null; aux2=aux2.getSiguiente()){

if(aux2.getDato().toString().equalsIgnoreCase(objeto.toString())){
                if(aux2.getSiguiente()==null){return null;}
                else{
                    aux=aux2.getDato();
                    aux2.setSiguiente(aux2.getSiguiente().getSiguiente());
                    contador--;
                }
            }
        }
        return aux;
    }
    public Object eliminarEnPosicion(int posicion){
        if(posicion<1){return null;}
        if(posicion==1){return eliminarCabecera();}
        int cont=1;
        boolean proceso=false;
        Nodo aux;
        for(aux=cabecera; aux!=null; aux=aux.getSiguiente()){
            if(cont==posicion-1){
                Object aux2=aux.getDato();
                if(aux.getSiguiente()!=null){
                    aux.setSiguiente(aux.getSiguiente().getSiguiente());
```

```
            }
            else{return aux;}
          }
          cont++;
        }
        return aux;
    }
}
```

- **API:**

  1. Constructor **ListaLigada()**:

  Este constructor inicializa una lista ligada vacía estableciendo tanto la cabecera como el fin como “null”, y el contador en 0.

  2. Método **taVacia()**:

  Verifica si la lista está vacía comparando el contador con 0. Retorna “true” si la lista está vacía, de lo contrario, retorna “false”.

  3. Método **agregarCabecera(Object objeto)**:

  Agrega un nuevo nodo con el objeto especificado al principio de la lista (cabecera). Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

  4. Método **agregarFinal(Object objeto)**:

  Agrega un nuevo nodo con el objeto especificado al final de la lista. Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

  5. Método **eliminarCabecera()**:

  Elimina y devuelve el objeto almacenado en el nodo de la cabecera. Retorna el objeto eliminado si la lista no está vacía, de lo contrario, retorna “null”.

  6. Método **eliminarFinal()**:

  Elimina y devuelve el objeto almacenado en el nodo del final de la lista. Retorna el objeto eliminado si la lista no está vacía, de lo contrario, retorna “null”.

  7. Método **agregarEnPosicion(Object objeto, int posicion)**:

Agrega un nuevo nodo con el objeto especificado en la posición indicada en la lista. Retorna `true` si la operación fue exitosa, de lo contrario, retorna “false”.

8. Método **imprimirLista()**:

Imprime los objetos almacenados en los nodos de la lista, empezando desde la cabecera y avanzando a través de los enlaces de cada nodo.

9. Método **agregarDespuesde(Object objeto, Object objeto2)**:

Agrega un nuevo nodo con el objeto2 después del nodo que contiene el objeto especificado. Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

10. Método **agregarAntesde(Object objeto, Object objeto2)**:

Agrega un nuevo nodo con el objeto especificado antes del nodo que contiene el objeto2. Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

11. Método **eliminarDespuesde(Object objeto)**:

Elimina el nodo siguiente al nodo que contiene el objeto especificado. Retorna el objeto eliminado si la operación fue exitosa, de lo contrario, retorna “null”.

12. Método **eliminarEnPosicion(int posicion)**:

Elimina el nodo en la posición especificada en la lista. Retorna el objeto eliminado si la operación fue exitosa, de lo contrario, retorna “null”.

- **Ejemplo:**

  Implementación de un gestor de tareas o lista de pendientes. En esta aplicación, los usuarios pueden agregar, eliminar y marcar tareas como completadas. Aquí tienes un ejemplo de cómo se puede utilizar una lista ligada para implementar un gestor de tareas:

  Supongamos que deseas desarrollar una aplicación de gestión de tareas donde los usuarios pueden agregar nuevas tareas, marcar las tareas como completadas y eliminar tareas de la lista.

  Agregar Tarea: Los usuarios pueden agregar una nueva tarea a la lista de pendientes. Cada tarea tiene un título y una descripción.

Marcar Tarea como Completada: Los usuarios pueden marcar una tarea como completada una vez que la hayan terminado.

Eliminar Tarea: Los usuarios pueden eliminar una tarea de la lista de pendientes si ya no es relevante.

```
public class GestorTareas {
    private ListaLigada listaTareas;

    public GestorTareas() {
        listaTareas = new ListaLigada();
    }

    public void agregarTarea(String titulo, String descripcion) {
        Tarea nuevaTarea = new Tarea(titulo, descripcion);
        listaTareas.agregarFinal(nuevaTarea);
    }

    public void marcarTareaComoCompletada(int indice) {
        Tarea tarea = (Tarea) listaTareas.obtenerElementoEnPosicion(indice);
        if (tarea != null) {
            tarea.setCompletada(true);
        }
    }

    public void eliminarTarea(int indice) {
        listaTareas.eliminarEnPosicion(indice);
    }

    public void imprimirListaTareas() {
        listaTareas.imprimirLista();
    }
```

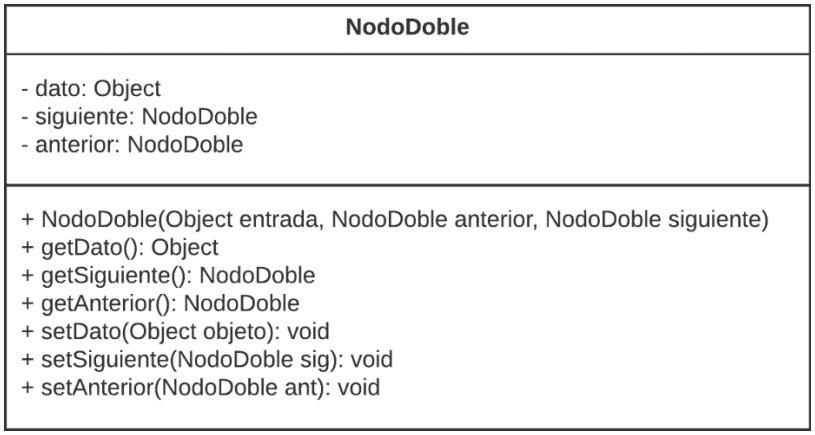
```
// Clase interna para representar una tarea
private class Tarea {
    private String titulo;
    private String descripcion;
    private boolean completada;

    public Tarea(String titulo, String descripcion) {
        this.titulo = titulo;
        this.descripcion = descripcion;
        this.completada = false;
    }

    public String getTitulo() {
        return titulo;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public boolean isCompletada() {
        return completada;
    }

    public void setCompletada(boolean completada) {
        this.completada = completada;
    }

    @Override
    public String toString() {
```

```
            String estado = completada ? "Completada" : "Pendiente";
            return titulo + " - " + descripcion + " (" + estado + ")";
        }
    }
}
```

DOBLES Y CIRCULARES:

➢ Nodo Doble:

- **UML:**



- **Código:**

```
public class NodoDoble
{
    private Object dato;
    private NodoDoble siguiente, anterior;
    public NodoDoble(Object entrada, NodoDoble anterior, NodoDoble siguiente){
        dato=entrada;
        this.siguiente=siguiente;
        this.anterior=anterior;
    }
    public Object getDato(){return dato;}
    public NodoDoble getSiguiente(){return siguiente;}
    public NodoDoble getAnterior(){return anterior;}
```

```
    public void setDato(Object objeto){dato=objeto;}
    public void setSiguiente(NodoDoble sig){this.siguiente=sig;}
    public void setAnterior(NodoDoble ant){this.anterior=ant;}
}
```

- **API:**

1. Constructor **NodoDoble(Object entrada, NodoDoble anterior, NodoDoble siguiente)**:

Crea un nuevo nodo con el dato especificado y establece sus referencias al nodo anterior y al siguiente nodo.

2. Método **getDato()**:

Devuelve el dato almacenado en este nodo.

3. Método **getSiguiente()**:

Devuelve una referencia al siguiente nodo en la lista.

4. Método **getAnterior()**:

Devuelve una referencia al nodo anterior en la lista.

5. Método **setDato(Object objeto)**:

Establece el dato almacenado en este nodo con el objeto especificado.

6. Método **setSiguiente(NodoDoble sig)**:

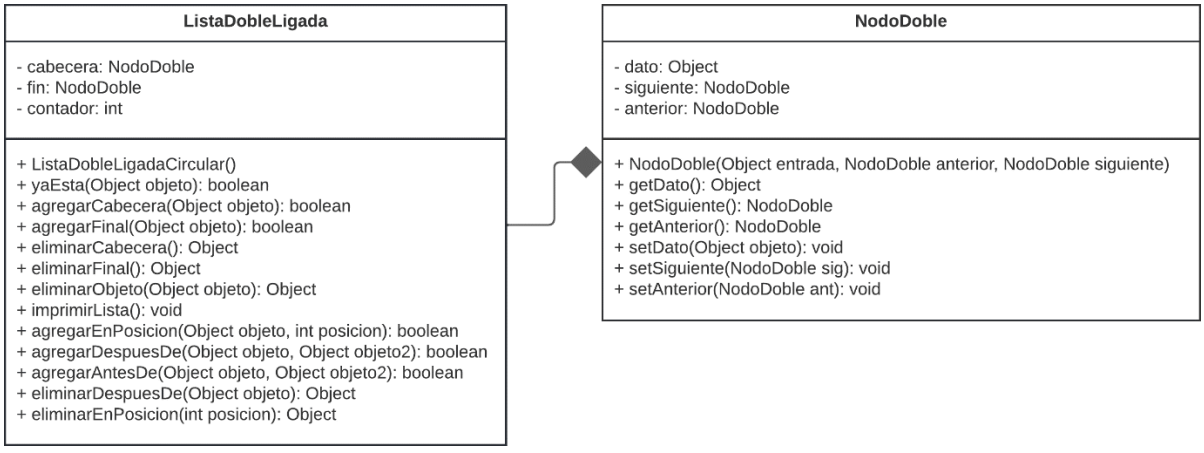
Establece la referencia al siguiente nodo con el nodo especificado.

7. Método **setAnterior(NodoDoble ant)**:

Establece la referencia al nodo anterior con el nodo especificado.

➢ Lista Doble Ligada:

- **UML:**



- **Código:**

```
public class ListaDobleLigada
{
    private NodoDoble cabecera, fin;
    private int contador;
    public ListaDobleLigada(){
        cabecera=fin=null;
        contador=0;
    }
    public boolean yaEsta(Object objeto){
        if(cabecera==null || objeto==null){return false;}
        NodoDoble aux=cabecera;
        while(aux!=null && !aux.getDato().equals(objeto)){
            aux=aux.getSiguiente();
        }
        if(aux==null){return false;}
        return true;
    }
    public boolean agregarCabecera(Object objeto){
        if(objeto==null){return false;}
        if(yaEsta(objeto)){return false;}
        NodoDoble nuevo=new NodoDoble(objeto, null, cabecera);
        if(cabecera!=null){cabecera.setAnterior(nuevo);}
        else{fin=nuevo;}
        cabecera=nuevo;
        contador++;
        return true;
    }
    public boolean agregarFinal(Object objeto){
        if(objeto==null)return false;
        if(cabecera==null){return agregarCabecera(objeto);}
```

```
    NodoDoble nuevo=new NodoDoble(objeto, fin, null);
    if(nuevo==null){return false;}
    fin.setSiguiente(nuevo);
    fin=nuevo;
    contador++;
    return true;
}
public Object eliminarCabecera(){
    if(cabecera==null){return null;}
    Object aux=cabecera.getDato();
    cabecera=cabecera.getSiguiente();
    if(cabecera!=null){cabecera.setAnterior(null);}
    else{fin=cabecera;}
    contador--;
    return aux;
}
public Object eliminarFinal(){
    if(cabecera==null){return null;}
    Object aux=fin.getDato();
    if(cabecera==fin){cabecera=fin=null;}
    else{
        fin=fin.getAnterior();
        fin.setSiguiente(null);
    }
    contador--;
    return aux;
}
public Object eliminarObjeto(Object objeto){
    if(cabecera==null){return null;}
    if(cabecera.getDato().equals(objeto)){
        return eliminarCabecera();
```

```
    }
    NodoDoble aux=cabecera.getSiguiente();
    while(aux!=null && !aux.getDato().equals(objeto)){
        aux=aux.getSiguiente();
    }
    if(aux==null){return null;}
    if(aux==fin){
        contador--;
        return eliminarFinal();
    }
    aux.getAnterior().setSiguiente(aux.getSiguiente());
    aux.getSiguiente().setAnterior(aux.getAnterior());
    contador--;
    return aux.getDato();
}
public void imprimirLista() {
    NodoDoble aux = cabecera;
    while (aux != null) {
        System.out.println(aux.getDato());
        aux = aux.getSiguiente();
    }
}
public boolean agregarEnPosicion(Object objeto, int posicion) {
    if (objeto == null || posicion < 0 || posicion > contador) {
        return false;
    }
    if (posicion == 0) {
        return agregarCabecera(objeto);
    } else if (posicion == contador) {
        return agregarFinal(objeto);
    } else {
```

```
            NodoDoble nuevo = new NodoDoble(objeto, null, null);
            NodoDoble aux = cabecera;
            for (int i = 0; i < posicion - 1; i++) {
                aux = aux.getSiguiente();
            }
            nuevo.setSiguiente(aux.getSiguiente());
            nuevo.setAnterior(aux);
            aux.getSiguiente().setAnterior(nuevo);
            aux.setSiguiente(nuevo);
            contador++;
            return true;
        }
    }
    public boolean agregarDespuesDe(Object objeto, Object objeto2) {
        if (objeto == null || objeto2 == null || cabecera == null) {
            return false;
        }
        NodoDoble aux = cabecera;
        while (aux != null && !aux.getDato().equals(objeto)) {
            aux = aux.getSiguiente();
        }
        if (aux == null) {
            return false;
        }
        NodoDoble nuevo = new NodoDoble(objeto2, aux,
aux.getSiguiente());
        if (aux == fin) {
            fin = nuevo;
        } else {
            aux.getSiguiente().setAnterior(nuevo);
        }
```

```
        aux.setSiguiente(nuevo);
        contador++;
        return true;
    }
    public boolean agregarAntesDe(Object objeto, Object objeto2) {
        if (objeto == null || objeto2 == null || cabecera == null) {
            return false;
        }
        NodoDoble aux = cabecera;
        while (aux != null && !aux.getDato().equals(objeto)) {
            aux = aux.getSiguiente();
        }
        if (aux == null) {
            return false;
        }
        NodoDoble nuevo = new NodoDoble(objeto2, aux.getAnterior(),
aux);
        if (aux == cabecera) {
            cabecera = nuevo;
        } else {
            aux.getAnterior().setSiguiente(nuevo);
        }
        aux.setAnterior(nuevo);
        contador++;
        return true;
    }
    public Object eliminarDespuesDe(Object objeto) {
        if (objeto == null || cabecera == null) {
            return null;
        }
        NodoDoble aux = cabecera;
```

```
    while (aux != null && !aux.getDato().equals(objeto)) {
        aux = aux.getSiguiente();
    }
    if (aux == null || aux == fin) {
        return null;
    }
    NodoDoble nodoAEliminar = aux.getSiguiente();
    aux.setSiguiente(nodoAEliminar.getSiguiente());
    if (nodoAEliminar == fin) {
        fin = aux;
    } else {
        nodoAEliminar.getSiguiente().setAnterior(aux);
    }
    contador--;
    return nodoAEliminar.getDato();
}
public Object eliminarEnPosicion(int posicion) {
    if (posicion < 0 || posicion >= contador) {
        return null;
    }
    if (posicion == 0) {
        return eliminarCabecera();
    } else if (posicion == contador - 1) {
        return eliminarFinal();
    }
    NodoDoble aux = cabecera;
    for (int i = 0; i < posicion; i++) {
        aux = aux.getSiguiente();
    }
    aux.getAnterior().setSiguiente(aux.getSiguiente());
    aux.getSiguiente().setAnterior(aux.getAnterior());
```

```
            contador--;
            return aux.getDato();
        }
    }
```

- **API:**

  1. Constructor **ListaDobleLigada()**:

  Inicializa la lista doblemente ligada estableciendo la cabecera y el final como nulos, y el contador en cero.

  2. Método **yaEsta(Object objeto)**:

  Verifica si un objeto dado ya está presente en la lista. Recorre la lista para buscar el objeto. Devuelve “true” si lo encuentra, de lo contrario, devuelve “false”.

  3. Método **agregarCabecera(Object objeto)**:

  Agrega un nuevo nodo con el objeto especificado al principio de la lista. Si el objeto ya está en la lista, no lo agrega y devuelve “false”. Actualiza la cabecera y el final si es necesario.

  4. Método **agregarFinal(Object objeto)**:

  Agrega un nuevo nodo con el objeto especificado al final de la lista. Si el objeto ya está en la lista, no lo agrega y devuelve “false”. Actualiza el final de la lista si es necesario.

  5. Método **eliminarCabecera()**:

  Elimina el primer nodo de la lista y devuelve su dato. Actualiza la cabecera y el final si es necesario.

  6. Método **eliminarFinal()**:

  Elimina el último nodo de la lista y devuelve su dato. Actualiza el final de la lista si es necesario.

  7. Método **eliminarObjeto(Object objeto)**:

  Elimina el nodo que contiene el objeto especificado de la lista y devuelve su dato. Si el objeto no está presente en la lista, devuelve “null”. Actualiza la cabecera y el final si es necesario.

  8. Método **imprimirLista()**:

Imprime los datos de los nodos en la lista, recorriéndola desde la cabecera hasta el final.

9. Método **agregarEnPosicion(Object objeto, int posicion)**:

Agrega un nuevo nodo con el objeto especificado en la posición indicada de la lista. Si la posición está fuera de rango o el objeto ya está en la lista, devuelve “false”. Actualiza la cabecera y el final si es necesario.

10. Método **agregarDespuesDe(Object objeto, Object objeto2)**:

Agrega un nuevo nodo con el objeto2 después del nodo que contiene el objeto especificado. Si el objeto especificado no está presente en la lista, no agrega el nodo y devuelve “false”. Actualiza la cabecera y el final si es necesario.

11. Método **agregarAntesDe(Object objeto, Object objeto2)**:

Agrega un nuevo nodo con el objeto2 antes del nodo que contiene el objeto especificado. Si el objeto especificado no está presente en la lista, no agrega el nodo y devuelve “false”. Actualiza la cabecera y el final si es necesario.

12. Método **eliminarDespuesDe(Object objeto)**:

Elimina el nodo que sigue al nodo que contiene el objeto especificado. Si el objeto especificado no está presente en la lista o el nodo siguiente es el final de la lista, devuelve “null”. Actualiza la cabecera y el final si es necesario.

13. Método **eliminarEnPosicion(int posicion)**:

Elimina el nodo en la posición especificada de la lista. Si la posición está fuera de rango, devuelve “null”. Actualiza la cabecera y el final si es necesario.

- **Ejemplo:**

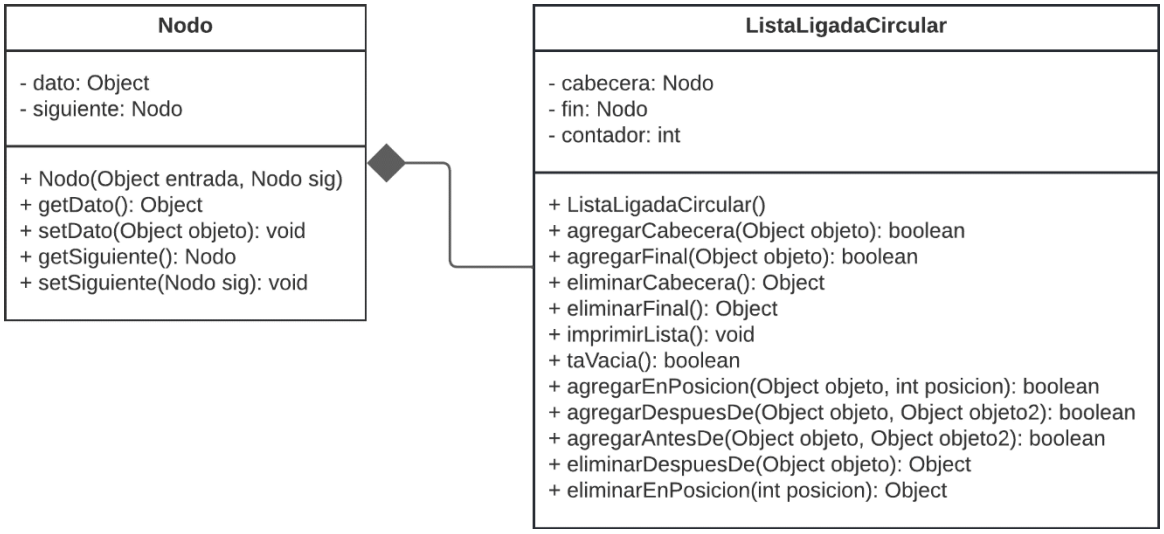
  Implementación de un editor de texto simple. En este editor de texto, los usuarios pueden realizar operaciones como insertar texto en cualquier posición, eliminar texto, deshacer y rehacer cambios, y mover el cursor hacia adelante y hacia atrás en el texto.

Supongamos que estás desarrollando un editor de texto simple en el que los usuarios pueden escribir, editar y formatear texto.
Insertar Texto: Los usuarios pueden insertar texto en cualquier posición del documento.
Eliminar Texto: Los usuarios pueden eliminar texto seleccionado o caracteres específicos del documento.
Deshacer y Rehacer Cambios: Los usuarios pueden deshacer o rehacer cambios realizados en el documento.
Mover Cursor: Los usuarios pueden mover el cursor hacia adelante y hacia atrás en el documento para realizar ediciones precisas.

```
public class EditorTexto {
    private ListaDobleLigada texto;

    public EditorTexto() {
        texto = new ListaDobleLigada();
    }

    public void insertarTexto(int posicion, String textoInsertar) {
        texto.insertarEnPosicion(posicion, textoInsertar);
    }

    public void eliminarTexto(int posicionInicio, int posicionFin) {
        texto.eliminarEnRango(posicionInicio, posicionFin);
    }

    public void deshacer() {
        texto.deshacer();
    }

    public void rehacer() {
```

```
        texto.rehacer();
    }

    public void moverCursorAdelante() {
        texto.moverCursorAdelante();
    }

    public void moverCursorAtras() {
        texto.moverCursorAtras();
    }

    public void imprimirTexto() {
        texto.imprimir();
    }
}
```

- **Lista Ligada Circular:**
  - **UML:**



  - **Código:**

```
public class ListaLigadaCircular {
    private Nodo cabecera, fin;
    private int contador;
```

```
public ListaLigadaCircular() {
    cabecera = fin = null;
    contador = 0;
}
public boolean agregarCabecera(Object objeto) {
    if (objeto == null) {
        return false;
    }
    if (cabecera == null) {
        cabecera = fin = new Nodo(objeto, null);
        cabecera.setSiguiente(cabecera);
    } else {
        Nodo aux = new Nodo(objeto, cabecera);
        if (aux == null) {
            return false;
        }
        cabecera = aux;
        fin.setSiguiente(cabecera);
    }
    contador++;
    return true;
}
public boolean agregarFinal(Object objeto) {
    if (objeto == null) {
        return false;
    }
    if (cabecera == null) {
        return agregarCabecera(objeto);
    } else {
        Nodo aux = new Nodo(objeto, cabecera);
        if (aux == null) {
```

```
            return false;
        }
        fin.setSiguiente(aux);
        fin = aux;
    }
    contador++;
    return true;
}
public Object eliminarCabecera() {
    Object objeto = null;
    if (cabecera != null) {
        objeto = cabecera.getDato();
        if (cabecera == fin) {
            cabecera = fin = null;
        } else {
            cabecera = cabecera.getSiguiente();
            fin.setSiguiente(cabecera);
        }
        contador--;
    }
    return objeto;
}
public Object eliminarFinal() {
    Object objeto = null;
    if (cabecera != null) {
        objeto = fin.getDato();
        if (cabecera == fin) {
            cabecera = fin = null;
        } else {
            Nodo aux = cabecera;
            while (aux.getSiguiente() != fin) {
```

```
                aux = aux.getSiguiente();
            }
            aux.setSiguiente(cabecera);
            fin = aux;
        }
        contador--;
    }
    return objeto;
}
public void imprimirLista() {
    if (taVacia()) {
        System.out.println("La lista está vacía");
        return;
    }
    Nodo aux = cabecera;
    do {
        System.out.println(aux.getDato());
        aux = aux.getSiguiente();
    } while (aux != cabecera);
}
public boolean taVacia() {
    return contador == 0;
}
public boolean agregarEnPosicion(Object objeto, int posicion) {
    if (objeto == null || posicion < 0 || posicion > contador) {
        return false;
    }
    if (posicion == 0) {
        return agregarCabecera(objeto);
    } else if (posicion == contador) {
        return agregarFinal(objeto);
```

```
    } else {
        Nodo nuevo = new Nodo(objeto, null);
        Nodo aux = cabecera;
        for (int i = 0; i < posicion - 1; i++) {
            aux = aux.getSiguiente();
        }
        nuevo.setSiguiente(aux.getSiguiente());
        aux.setSiguiente(nuevo);
        contador++;
        return true;
    }
}
public boolean agregarDespuesDe(Object objeto, Object objeto2) {
    if (objeto == null) {
        return false;
    }
    Nodo aux = cabecera;
    do {
        if (aux.getDato().equals(objeto)) {
            Nodo nuevo = new Nodo(objeto2, aux.getSiguiente());
            aux.setSiguiente(nuevo);
            if (aux == fin) {
                fin = nuevo;
            }
            contador++;
            return true;
        }
        aux = aux.getSiguiente();
    } while (aux != cabecera);
    return false;
}
```

```
public boolean agregarAntesDe(Object objeto, Object objeto2) {
    if (objeto == null) {
        return false;
    }
    Nodo anterior = null;
    Nodo actual = cabecera;
    do {
        if (actual.getDato().equals(objeto)) {
            Nodo nuevo = new Nodo(objeto2, actual);
            if (anterior != null) {
                anterior.setSiguiente(nuevo);
            } else {
                cabecera = nuevo;
                fin.setSiguiente(cabecera);
            }
            contador++;
            return true;
        }
        anterior = actual;
        actual = actual.getSiguiente();
    } while (actual != cabecera);
    return false;
}
public Object eliminarDespuesde(Object objeto) {
    if (cabecera == null) {
        return null;
    }
    Nodo aux = cabecera;
    do {
        if (aux.getDato().equals(objeto)) {
```

```
            if (aux.getSiguiente() != cabecera) {
                Object objetoEliminado = aux.getSiguiente().getDato();
                aux.setSiguiente(aux.getSiguiente().getSiguiente());
                contador--;
                return objetoEliminado;
            }
        }
        aux = aux.getSiguiente();
    } while (aux != cabecera);
    return null;
}
public Object eliminarEnPosicion(int posicion) {
    if (posicion < 0 || posicion >= contador) {
        return null;
    }
    if (posicion == 0) {
        return eliminarCabecera();
    }
    Nodo anterior = null;
    Nodo actual = cabecera;
    for (int i = 0; i < posicion; i++) {
        anterior = actual;
        actual = actual.getSiguiente();
    }
    Object objetoEliminado = actual.getDato();
    if (actual == fin) {
        fin = anterior;
    }
    anterior.setSiguiente(actual.getSiguiente());
    contador--;
    return objetoEliminado;
```

```
    }
}
```

- **API:**

1. Constructor **ListaLigadaCircular()**:

Inicializa una lista ligada circular vacía estableciendo tanto la cabecera como el fin como “null”, y el contador en 0.

2. Método **agregarCabecera(Object objeto)**:

Agrega un nuevo nodo con el objeto especificado al principio de la lista (cabecera). Si la lista está vacía, crea un nuevo nodo y lo establece como la cabecera y el fin, si no, inserta un nuevo nodo antes de la cabecera actual. Asegura que el último nodo apunte a la cabecera. Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

3. Método **agregarFinal(Object objeto)**:

Agrega un nuevo nodo con el objeto especificado al final de la lista. Si la lista está vacía, llama al método “agregarCabecera(objeto)”. Si no está vacía, crea un nuevo nodo y lo establece como el siguiente del último nodo, actualizando el puntero de fin y asegurando que el último nodo apunte a la cabecera. Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

4. Método **eliminarCabecera()**:

Elimina y devuelve el objeto almacenado en el nodo de la cabecera. Si la lista tiene un solo nodo, establece la cabecera y el fin como “null”. Si no, mueve la cabecera al siguiente nodo y actualiza el puntero del último nodo para que apunte a la nueva cabecera. Retorna el objeto eliminado si la lista no está vacía, de lo contrario, retorna “null”.

5. Método **eliminarFinal()**:

Elimina y devuelve el objeto almacenado en el nodo del final de la lista. Si la lista tiene un solo nodo, establece la cabecera y el fin como “null”. Si no, recorre la lista hasta el penúltimo nodo, actualiza el puntero del último nodo para que apunte a la nueva cabecera y elimina el último

nodo. Retorna el objeto eliminado si la lista no está vacía, de lo contrario, retorna “null”.

6. Método **imprimirLista()**:

Imprime los objetos almacenados en los nodos de la lista en un bucle do-while, comenzando desde la cabecera y avanzando hasta que el nodo actual sea igual a la cabecera.

7. Método **taVacia()**:

Verifica si la lista está vacía comparando el contador con 0. Retorna “true” si la lista está vacía, de lo contrario, retorna “false”.

8. Método **agregarEnPosicion(Object objeto, int posicion)**:

Agrega un nuevo nodo con el objeto especificado en la posición indicada en la lista. Si la posición es 0, llama al método “agregarCabecera(objeto)”. Si la posición es igual al contador, llama al método “agregarFinal(objeto)”. En otro caso, inserta el nuevo nodo después del nodo en la posición anterior a la especificada. Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

9. Método **agregarDespuesDe(Object objeto, Object objeto2)**:

Agrega un nuevo nodo con el objeto2 después del nodo que contiene el objeto especificado. Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

10. Método **agregarAntesDe(Object objeto, Object objeto2)**:

Agrega un nuevo nodo con el objeto especificado antes del nodo que contiene el objeto2. Retorna “true” si la operación fue exitosa, de lo contrario, retorna “false”.

11. Método **eliminarDespuesde(Object objeto)**:

Elimina el nodo siguiente al nodo que contiene el objeto especificado. Retorna el objeto eliminado si la operación fue exitosa, de lo contrario, retorna “null”.

12. Método **eliminarEnPosicion(int posicion)**:

Elimina el nodo en la posición especificada en la lista. Retorna el objeto eliminado si la operación fue exitosa, de lo contrario, retorna “null”.

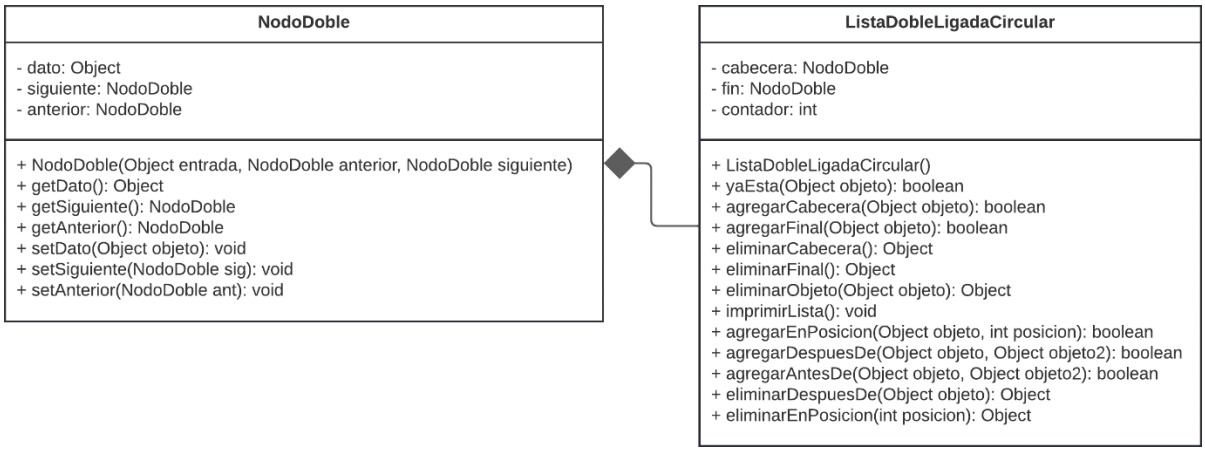
- **Ejemplo:**

```
public class Main {
    public static void main(String[] args) throws Exception{
        ListaCircular listaCircular = new ListaCircular();

        System.out.println("<<-- Ejemplo de lista circular simple -->>\n");

        // Agregar al final de la lista circular
        listaCircular.agregarAlFinal(12);
        listaCircular.agregarAlFinal(15);
        listaCircular.agregarAlFinal(9);
        // Agregar in inicio de la lista circular
        listaCircular.agregarAlInicio(41);
        listaCircular.agregarAlInicio(6);

        System.out.println("<<-- Lista Circular -->>");
        listaCircular.listar();

        System.out.println("\n\n<<-- Tamaño -->");
        System.out.println(listaCircular.getTamanio());

        System.out.println("\n<<-- Obtener el valor del nodo en la posicion
3 -->>");
        System.out.println(listaCircular.getValor(3));

        System.out.println("\nInsrta un nodo con valor 16 despues del 15");
        listaCircular.insertarPorReferencia(15, 16);
        listaCircular.listar();
        System.out.print(" | Tamaño: ");
        System.out.println(listaCircular.getTamanio());
```

```
        System.out.println("\n\nInsrta un nodo con valor 44 en la posición 5");
        listaCircular.insrtarPorPosicion(5, 44);
        listaCircular.listar();
        System.out.print(" | Tamaño: ");
        System.out.println(listaCircular.getTamanio());

        System.out.println("\nActualiza el valor 12 del tercer nodo por 13");
        listaCircular.editarPorReferencia(12, 13);
        listaCircular.listar();
        System.out.print(" | Tamaño: ");
        System.out.println(listaCircular.getTamanio());

        System.out.println("\nActualiza el valor nodo en la posición 0 por 17");
        listaCircular.editarPorPosicion(0, 17);
        listaCircular.listar();
        System.out.print(" | Tamaño: ");
        System.out.println(listaCircular.getTamanio());

        System.out.println("\nElimina el nodo con el valor 41");
        listaCircular.removerPorReferencia(41);
        listaCircular.listar();
        System.out.print(" | Tamaño: ");
        System.out.println(listaCircular.getTamanio());

        System.out.println("\nElimina el nodo en la posición 4");
        listaCircular.removerPorPosicion(4);
        listaCircular.listar();
        System.out.print(" | Tamaño: ");
        System.out.println(listaCircular.getTamanio());
```

```
        System.out.println("\nConsulta si existe el valor 30");
        System.out.println(listaCircular.buscar(30));

        System.out.println("\nConsulta la posicion del valor 16");
        System.out.println(listaCircular.getPosicion(16));

        System.out.println("\nElimina la lista");
        listaCircular.eliminar();

        System.out.println("\nConsulta si la lista está vacia");
        System.out.println(listaCircular.esVacia());

        System.out.println("\n\n<<-- Fin de ejemplo lista simple -->>");
    }
}
```

- **Lista Doble Ligada Circular:**
  - **UML:**



  - **Código:**

```
public class ListaDobleLigadaCircular
{
    private NodoDoble cabecera, fin;
    private int contador;
```

```
public ListaDobleLigadaCircular() {
    cabecera = fin = null;
    contador = 0;
}
public boolean yaEsta(Object objeto) {
    if (cabecera == null || objeto == null) {
        return false;
    }
    NodoDoble aux = cabecera;
    do {
        if (aux.getDato().equals(objeto)) {
            return true;
        }
        aux = aux.getSiguiente();
    } while (aux != cabecera);
    return false;
}
public boolean agregarCabecera(Object objeto) {
    if (objeto == null || yaEsta(objeto)) {
        return false;
    }
    if (cabecera == null) {
        cabecera = fin = new NodoDoble(objeto, null, null);
        cabecera.setSiguiente(cabecera);
        cabecera.setAnterior(cabecera);
    } else {
        NodoDoble nuevo = new NodoDoble(objeto, fin, cabecera);
        cabecera.setAnterior(nuevo);
        fin.setSiguiente(nuevo);
        cabecera = nuevo;
    }
```

```
    contador++;
    return true;
}
public boolean agregarFinal(Object objeto) {
    if (objeto == null || yaEsta(objeto)) {
        return false;
    }
    if (cabecera == null) {
        return agregarCabecera(objeto);
    } else {
        NodoDoble nuevo = new NodoDoble(objeto, fin, cabecera);
        cabecera.setAnterior(nuevo);
        fin.setSiguiente(nuevo);
        fin = nuevo;
    }
    contador++;
    return true;
}
public Object eliminarCabecera() {
    if (cabecera == null) {
        return null;
    }
    Object dato = cabecera.getDato();
    if (cabecera == fin) {
        cabecera = fin = null;
    } else {
        cabecera = cabecera.getSiguiente();
        fin.setSiguiente(cabecera);
        cabecera.setAnterior(fin);
    }
    contador--;
```

```
    return dato;
}
public Object eliminarFinal() {
    if (fin == null) {
        return null;
    }
    Object dato = fin.getDato();
    if (cabecera == fin) {
        cabecera = fin = null;
    } else {
        fin = fin.getAnterior();
        fin.setSiguiente(cabecera);
        cabecera.setAnterior(fin);
    }
    contador--;
    return dato;
}
public Object eliminarObjeto(Object objeto) {
    if (objeto == null || cabecera == null) {
        return null;
    }
    NodoDoble aux = cabecera;
    do {
        if (aux.getDato().equals(objeto)) {
            if (aux == cabecera) {
                return eliminarCabecera();
            } else if (aux == fin) {
                return eliminarFinal();
            } else {
                aux.getAnterior().setSiguiente(aux.getSiguiente());
                aux.getSiguiente().setAnterior(aux.getAnterior());
```

```
                contador--;
                return aux.getDato();
            }
        }
        aux = aux.getSiguiente();
    } while (aux != cabecera);
    return null;
}
public void imprimirLista() {
    if (cabecera == null) {
        return;
    }
    NodoDoble aux = cabecera;
    do {
        System.out.println(aux.getDato());
        aux = aux.getSiguiente();
    } while (aux != cabecera);
}
public boolean agregarEnPosicion(Object objeto, int posicion) {
    if (objeto == null || posicion < 0 || posicion > contador) {
        return false;
    }
    if (posicion == 0) {
        return agregarCabecera(objeto);
    } else if (posicion == contador) {
        return agregarFinal(objeto);
    } else {
        NodoDoble aux = cabecera;
        for (int i = 0; i < posicion - 1; i++) {
            aux = aux.getSiguiente();
        }
```

```
            NodoDoble nuevo = new NodoDoble(objeto, aux,
aux.getSiguiente());
            aux.getSiguiente().setAnterior(nuevo);
            aux.setSiguiente(nuevo);
            contador++;
            return true;
        }
    }
    public boolean agregarDespuesDe(Object objeto, Object objeto2) {
        if (objeto == null || objeto2 == null || cabecera == null) {
            return false;
        }
        NodoDoble aux = cabecera;
        do {
            if (aux.getDato().equals(objeto)) {
                NodoDoble nuevo = new NodoDoble(objeto2, aux,
aux.getSiguiente());
                aux.getSiguiente().setAnterior(nuevo);
                aux.setSiguiente(nuevo);
                contador++;
                return true;
            }
            aux = aux.getSiguiente();
        } while (aux != cabecera);
        return false;
    }
    public boolean agregarAntesDe(Object objeto, Object objeto2) {
        if (objeto == null || objeto2 == null || cabecera == null) {
            return false;
        }
        NodoDoble aux = cabecera;
```

```
        do {
            if (aux.getDato().equals(objeto)) {
                NodoDoble nuevo = new NodoDoble(objeto2,
aux.getAnterior(), aux);
                aux.getAnterior().setSiguiente(nuevo);
                aux.setAnterior(nuevo);
                contador++;
                return true;
            }
            aux = aux.getSiguiente();
        } while (aux != cabecera);
        return false;
    }
    public Object eliminarDespuesDe(Object objeto) {
        if (objeto == null || cabecera == null) {
            return null;
        }
        NodoDoble aux = cabecera;
        do {
            if (aux.getDato().equals(objeto)) {
                NodoDoble nodoAEliminar = aux.getSiguiente();
                aux.setSiguiente(nodoAEliminar.getSiguiente());
                nodoAEliminar.getSiguiente().setAnterior(aux);
                contador--;
                return nodoAEliminar.getDato();
            }
            aux = aux.getSiguiente();
        } while (aux != cabecera);
        return null;
    }
    public Object eliminarEnPosicion(int posicion) {
```

```
        if (posicion < 0 || posicion >= contador) {
            return null;
        }
        if (posicion == 0) {
            return eliminarCabecera();
        } else if (posicion == contador - 1) {
            return eliminarFinal();
        }
        NodoDoble aux = cabecera;
        for (int i = 0; i < posicion; i++) {
            aux = aux.getSiguiente();
        }
        aux.getAnterior().setSiguiente(aux.getSiguiente());
        aux.getSiguiente().setAnterior(aux.getAnterior());
        contador--;
        return aux.getDato();
    }
}
```

- **API:**

  1. Constructor **ListaDobleLigadaCircular()**:

  Inicializa la lista doblemente ligada circular estableciendo la cabecera y el final como nulos, y el contador en cero.

  2. Método **yaEsta(Object objeto)**:

  Verifica si un objeto dado ya está presente en la lista. Recorre la lista circular para buscar el objeto. Devuelve “true” si lo encuentra, de lo contrario, devuelve “false”.

  3. Método **agregarCabecera(Object objeto)**:

  Agrega un nuevo nodo con el objeto especificado al principio de la lista. Si el objeto ya está en la lista o es nulo, no lo agrega y devuelve “false”. Actualiza la cabecera y el final si es necesario.

  4. Método **agregarFinal(Object objeto)**:

Agrega un nuevo nodo con el objeto especificado al final de la lista. Si el objeto ya está en la lista o es nulo, no lo agrega y devuelve “false”. Actualiza la cabecera y el final si es necesario.

5. Método **eliminarCabecera()**:

Elimina el primer nodo de la lista y devuelve su dato. Actualiza la cabecera y el final si es necesario.

6. Método **eliminarFinal()**:

Elimina el último nodo de la lista y devuelve su dato. Actualiza la cabecera y el final si es necesario.

7. Método **eliminarObjeto(Object objeto)**:

Elimina el nodo que contiene el objeto especificado de la lista y devuelve su dato. Si el objeto no está presente en la lista, devuelve “null”. Actualiza la cabecera y el final si es necesario.

8. Método **imprimirLista()**:

Imprime los datos de los nodos en la lista, recorriéndola desde la cabecera hasta el final en un bucle circular.

9. Método **agregarEnPosicion(Object objeto, int posicion)**:

Agrega un nuevo nodo con el objeto especificado en la posición indicada de la lista. Si la posición está fuera de rango o el objeto ya está en la lista, devuelve “false”. Actualiza la cabecera y el final si es necesario.

10. Método **agregarDespuesDe(Object objeto, Object objeto2)**:

Agrega un nuevo nodo con el objeto2 después del nodo que contiene el objeto especificado. Si el objeto especificado no está presente en la lista, no agrega el nodo y devuelve “false”. Actualiza la cabecera y el final si es necesario.

11. Método **agregarAntesDe(Object objeto, Object objeto2)**:

Agrega un nuevo nodo con el objeto2 antes del nodo que contiene el objeto especificado. Si el objeto especificado no está presente en la lista, no agrega el nodo y devuelve “false”. Actualiza la cabecera y el final si es necesario.

12. Método **eliminarDespuesDe(Object objeto)**:

Elimina el nodo que sigue al nodo que contiene el objeto especificado. Si el objeto especificado no está presente en la lista o el nodo siguiente es el final de la lista, devuelve “null”. Actualiza la cabecera y el final si es necesario.

13. Método **eliminarEnPosicion(int posicion)**:

Elimina el nodo en la posición especificada de la lista. Si la posición está fuera de rango, devuelve “null”. Actualiza la cabecera y el final si es necesario.

- **Ejemplo:**

```
import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
/**
 * @author Diego Fernando Echeverry
 * @author Carlos Augusto Barrera
 */
public class ListaCircularDoble {
    private  int contador;
    class Nodo {

        Object dato;
        ListaCircularDoble.Nodo anterior, siguiente;
    }
    ListaCircularDoble.Nodo inicio;

    public void insertarInicio(Object v) {
        if (inicio == null) { // cuando no se ha creado ningun nodo
            inicio = new ListaCircularDoble.Nodo();//Se crea un espacio en memoria para el nodo inicio
            inicio.dato = v; // se inserta el dato
            inicio.siguiente = inicio;
            inicio.anterior = inicio;
```

```
            contador++;
        } else {
            //ingresar al final
            ListaCircularDoble.Nodo nuevo = new ListaCircularDoble.Nodo();//Se crea un espacio en memoria para el nodo inicio
            nuevo.dato = v;// se inserta el dato
            nuevo.siguiente = inicio; //nuevo en su .next apunta hacia nuestro nodo inicio
            nuevo.anterior = inicio.anterior; // nuevo en su .ant apunta al nodo que es el final
            inicio.anterior = nuevo; //nuevo en su .ant apunta hacia nuestro nodo nuevo
            nuevo.anterior.siguiente = nuevo; //nuevo en su nodo anterioir en su .next apunta hacia nuestro nodo nuevo
            inicio = nuevo; //ahora inicio apunta a nuestro nodo Nuevo
            contador++;
        }
    }

    public void insertarDatoInicio(Object v) {
        if (inicio != null) {
            ListaCircularDoble.Nodo nuevo = new ListaCircularDoble.Nodo();//Se crea un espacio en memoria para el nodo inicio
            nuevo.dato = v;// se inserta el dato
            ListaCircularDoble.Nodo aux = inicio.anterior;// creamos temportal para reposicionar la direccion de los nodos
            nuevo.anterior = aux;// enlazo el nuevo nodo con el ultimo nodo de la lista
```

```
            aux.siguiente = nuevo; //el ultimo nodo lo apunto en su siguiente al nuevo nodo
            inicio.anterior=nuevo;// el primer nodo lo enlazon en su anterio con el nuevo nodo
            nuevo.siguiente = inicio;//el nuevo nodo en su siguiente lo enlazo con el primero
            inicio = nuevo;//ahora el nuevo nodo apunta al inicio
            contador++;
        }
    }

    public void insertarEnmedio(Object v) {

        int cont = 1;
        ListaCircularDoble.Nodo auxiliar = inicio;
        while (auxiliar.siguiente != inicio) {
            int medio = (int) (contador / 2);
            if (cont == medio) {
                ListaCircularDoble.Nodo nuevo = new ListaCircularDoble.Nodo();//Se crea un espacio en memoria para el nodo inicio
                nuevo.dato = v;// se inserta el dato
                ListaCircularDoble.Nodo auxiliar2 = auxiliar.siguiente;
                auxiliar.siguiente = nuevo;
                nuevo.anterior = auxiliar;
                nuevo.siguiente = auxiliar2;
                auxiliar2.anterior = nuevo;
                contador++;
                break;
            }
```

```
            cont++;
            auxiliar = auxiliar.siguiente;
        }
    }
    public void insertarEnPosicion(Object v,int pos) {

        int cont = 1;
        ListaCircularDoble.Nodo auxiliar = inicio;
        while (auxiliar.siguiente != inicio) {
            if (cont == pos -1) {
                ListaCircularDoble.Nodo nuevo = new ListaCircularDoble.Nodo();//Se crea un espacio en memoria para el nodo inicio
                nuevo.dato = v;// se inserta el dato
                ListaCircularDoble.Nodo auxiliar2 = auxiliar.siguiente;
                auxiliar.siguiente = nuevo;
                nuevo.anterior = auxiliar;
                nuevo.siguiente = auxiliar2;
                auxiliar2.anterior = nuevo;
                contador++;
                break;
            }

            cont++;
            auxiliar = auxiliar.siguiente;
        }
    }

    public Object extraerInicio() {
        if (inicio != null) {
            Object v;
```

```
            if (inicio.siguiente != inicio) {
                v = inicio.dato; // se inserta el dato
                inicio.siguiente.anterior = inicio.anterior; // inicio en su nodo siguiente, en su .ant apunta al nodo anterior de inicio
                inicio.anterior.siguiente = inicio.siguiente; // inicio en su nodo anterior, en su .next apunta al nodo siguiente de inicio
                inicio = inicio.siguiente;
                contador--;
            } else {
                v = inicio.dato;
                inicio = null;
            }
            return v;
        }
        return -1;
    }

    public Object extraerFinal() {
        if (inicio != null) {
            Object v;
            if (inicio.siguiente != inicio) {
                v = inicio.anterior.dato;
                inicio.anterior.anterior.siguiente = inicio;//inicio en su nodo anterior, anterior, en su .next apunta a inicio
                inicio.anterior = inicio.anterior.anterior; //inicio en su .next apunta a inicio en su nodo anterior en su .ant
                contador--;
                return v;//se retorna el valo
            } else {
                return extraerInicio();//se llama a este metodo para extraer inicio, ya que solo tenemos un nodo
```

```
                //el cual es inicio y final a la vez, y a traves de exte metodo se
extrae :D
            }
        }
        return -1;
    }

    public void buscar(String palabra) {
        boolean estado = false;
        ListaCircularDoble.Nodo aux = inicio;
        if (aux != null) {
            while (aux.siguiente != inicio) {
                if (aux.dato.equals(palabra)) {
                    estado = true;
                    break;
                }
                aux = aux.siguiente;
            }
            if (estado) {
                JOptionPane.showMessageDialog(null, "si esta!");
            } else {
                JOptionPane.showMessageDialog(null,  "No  esta!",  "Erorr",
JOptionPane.ERROR_MESSAGE);
            }

        }
    }

    public void eliminar(String palabra) {
        boolean estado = false;
        ListaCircularDoble.Nodo aux = inicio;
```

```
        if (aux != null) {
            while (aux.siguiente != inicio) {
                if (aux.dato.equals(palabra)) {
                    inicio.siguiente.anterior = aux.anterior; // inicio en su nodo siguiente, en su .ant apunta al nodo anterior de inicio
                    inicio.anterior.siguiente = aux.siguiente; // inicio en su nodo anterior, en su .next apunta al nodo siguiente de inicio
                    inicio = inicio.siguiente;
                    estado = true;
                }
                aux = aux.siguiente;
            }
        }
    }
    /**
     * metodo para imprimir en jlist
     * @return
     */
    public DefaultListModel<String> mostrarL() {
        DefaultListModel<String> l1 = new DefaultListModel<>();

        ListaCircularDoble.Nodo aux = inicio;
        if (aux != null) {
            l1.addElement(" dato | siguiente | anterior");
            while (aux.siguiente != inicio) {
                l1.addElement(
                        aux.dato + " | "
                        + aux.anterior.dato + " | "
                        + aux.siguiente.dato);
                aux = aux.siguiente;
            }
```

```
            l1.addElement(
                    aux.dato + " | "
                    + aux.anterior.dato + " | "
                    + aux.siguiente.dato);
        }
        return l1;
    }
}
```

CONCLUSIÓN

Bien, durante el transcurso o en esta travesía como le quieran o quieras decir, hemos visto mas que nada como se lleva a cabo la codificación o la realización del código para el funcionamiento de cada una de las estructuras de datos.

Cabe recalcar que, como ya había mencionado anteriormente, la tecnología es una de las cosas primordiales en nuestro tiempo o nuestra época, por lo que tenemos que familiarizarnos con ella para poder desarrollar exponencialmente nuestras habilidades y llevar a cabo un correcto uso de la misma.

Solo me queda decir que, espero les haya gustado mi trabajo y por supuesto, les haya ayudado para tener una mayor comprensión de este tema, nos vemos.