

Asignatura	Datos del alumno	Fecha
Estructura de datos	Apellidos: López Morales	30/11/2024
	Nombre: Álvaro	

Laboratorio: Implementación de pilas y colas.

Memoria del proyecto.

1. Introducción

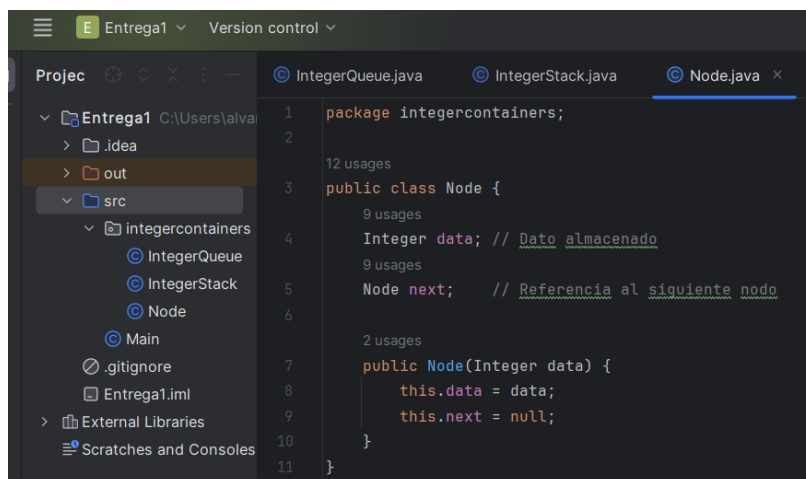
En este proyecto se ha implementado dos estructuras de datos fundamentales: pila (LIFO) y cola (FIFO), usando nodos enlazados. Las estructuras han sido implementadas desde cero sin el uso de clases predefinidas para ello como ArrayList o LinkedList, para poder entender cómo funcionan las listas a un nivel más bajo y cómo se manipulan los nodos para construir estas estructuras.

2. Diseño e Implementación

Estructura de los Nodos

Para ambas estructuras, se ha creado una clase Node, que es la base de cada elemento en la pila o cola. Cada nodo contiene dos campos:

- data (Integer): El valor almacenado en el nodo.
- next (Node): Una referencia al siguiente nodo en la estructura, que nos permite enlazar los nodos entre sí.



```

1 package integercontainers;
2
3 12 usages
4 public class Node {
5     9 usages
6     Integer data; // Dato almacenado
7     9 usages
8     Node next; // Referencia al siguiente nodo
9
10    2 usages
11    public Node(Integer data) {
12        this.data = data;
13        this.next = null;
14    }
15 }

```

Implementación de la Pila (LIFO)

La pila es una estructura de datos que sigue el principio de “El último en entrar será el primero en salir” (LIFO). En esta implementación:

- Se mantiene una referencia al nodo superior (top) de la pila.

Asignatura	Datos del alumno	Fecha
Estructura de datos	Apellidos: López Morales	30/11/2024
	Nombre: Álvaro	

- `push(Integer i)`: Se inserta un nuevo nodo al principio de la pila, convirtiéndolo en el nuevo nodo superior.
- `pop()`: Elimina el nodo superior y devuelve su valor.
- `top()`: Devuelve el valor del nodo superior sin eliminarlo.
- `size()`: Devuelve el número de nodos en la pila.
- `search(Integer i)`: Recorre la pila y verifica si el valor existe.

Código de la Pila:

```
package integercontainers;

3 usages
public class IntegerStack {
    12 usages
    private Node top; // Nodo en la cima de la pila
    4 usages
    private int size; // Número de elementos en la pila

    //constructor de la clase
    1 usage
    public IntegerStack() {
        this.top = null;
        this.size = 0;
    }

    // Apilar un elemento
    3 usages
    public void push(Integer i) {
        Node newNode = new Node(i);
        newNode.next = top; // Apunta al antiguo "top"
        top = newNode;     // Ahora este es el nuevo "top"
        size++;
    }

    // Desapilar un elemento
    1 usage
    public Integer pop() {
        if (top == null) {
            return null; // Pila vacía
        }
        Integer value = top.data; // Valor del nodo en la cima
        top = top.next;          // Mueve la cima al siguiente nodo
        size--;
        return value;
    }

    // Obtener el elemento en la cima
    no usages
    public Integer top() {
        return (top == null) ? null : top.data;
    }

    // Contar número de elementos
    no usages
    public int size() {
        return size;
    }
}
```

Asignatura	Datos del alumno	Fecha
Estructura de datos	Apellidos: López Morales	30/11/2024
	Nombre: Álvaro	

```

// Comprobar si un elemento está en la pila
1 usage
public boolean search(Integer i) {
    Node current = top;
    while (current != null) {
        if (current.data.equals(i)) {
            return true;
        }
        current = current.next;
    }
    return false;
}

// Mostrar contenido
@Override
public String toString() {
    if (top == null) {
        return "Empty stack";
    }
    StringBuilder sb = new StringBuilder();
    Node current = top;
    while (current != null) {
        sb.append(current.data).append(" ");
        current = current.next;
    }
    return sb.toString();
}
}

```

Implementación de la Cola (FIFO)

La cola, al contrario de la pila, sigue el principio de “primero en entrar, primero en salir” (FIFO). Este principio, por ejemplo, es el que se sigue en industria que trabaje con materia prima que tenga una vida útil determinada. En esta implementación:

- Se mantienen dos referencias: head (cabeza) y tail (cola).
- insert(Integer i): Inserta un nuevo nodo al final de la cola.
- remove(): Elimina el nodo en la cabeza de la cola y devuelve su valor.
- seek(): Devuelve el valor del nodo en la cabeza sin eliminarlo.
- size(): Devuelve el número de nodos en la cola.
- search(Integer i): Recorre la cola y verifica si el valor existe.

La inserción y eliminación en la cola son eficientes ($O(1)$) debido a las referencias directas a los nodos de cabeza y cola.

Código de la Cola:

Asignatura	Datos del alumno	Fecha
Estructura de datos	Apellidos: López Morales	30/11/2024
	Nombre: Álvaro	

```

1 package integercontainers;
2
3 3 usages
4 public class IntegerQueue {
5     12 usages
6     private Node head; // Nodo al inicio de la cola
7     6 usages
8     private Node tail; // Nodo al final de la cola
9     4 usages
10    private int size; // Número de elementos en la cola
11
12    1 usage
13    public IntegerQueue() {
14        this.head = null;
15        this.tail = null;
16        this.size = 0;
17    }
18
19    // Insertar un elemento
20    3 usages
21    public void insert(Integer i) {
22        Node newNode = new Node(i);
23        if (tail == null) { // La cola está vacía
24            head = newNode;
25            tail = newNode;
26        } else {
27            tail.next = newNode; // Añade al final
28            tail = newNode;    // Actualiza el final
29        }
30        size++;
31    }

```

```

32    // Sacar un elemento
33    1 usage
34    public Integer remove() {
35        if (head == null) {
36            return null; // Cola vacía
37        }
38        Integer value = head.data; // Valor en la cabecera
39        head = head.next;        // Mueve la cabecera al siguiente nodo
40        if (head == null) {      // Si la cola quedó vacía
41            tail = null;
42        }
43        size--;
44        return value;
45    }
46
47    // Obtener el elemento en la cabecera
48    1 usage
49    public Integer seek() {
50        return (head == null) ? null : head.data;
51    }
52
53    // Contar elementos
54    1 usage
55    public int size() {
56        return size;
57    }

```

Asignatura	Datos del alumno	Fecha
Estructura de datos	Apellidos: López Morales	30/11/2024
	Nombre: Álvaro	

```

// Comprobar si un elemento está en la cola
1 usage
public boolean search(Integer i) {
    Node current = head;
    while (current != null) {
        if (current.data.equals(i)) {
            return true;
        }
        current = current.next;
    }
    return false;
}

// Mostrar contenido
@Override
public String toString() {
    if (head == null) {
        return "Empty queue";
    }
    StringBuilder sb = new StringBuilder();
    Node current = head;
    while (current != null) {
        sb.append(current.data).append(" ");
        current = current.next;
    }
    return sb.toString();
}
}

```

En ambas clases, tanto en IntegerStack como en IntegerQueue, se ha realizado un override del método ToString, con el objetivo de que éste nos devuelva que están vacías en caso de que lo estén y que vaya acumulando los valores de cada nodo en un objeto de tipo StringBuilder, acumulándolos en una cadena y mostrándolos por pantalla separados por espacios.

Finalmente, adjunto la imagen del archivo Main, donde se pueden ver las pruebas que he realizado para verificar que estaba todo correcto:

Asignatura	Datos del alumno	Fecha
Estructura de datos	Apellidos: López Morales	30/11/2024
	Nombre: Álvaro	

```

1  import integercontainers.IntegerQueue;
2  import integercontainers.IntegerStack;
3
4  public class Main {
5      public static void main(String[] args) {
6
7          System.out.println("-----Probando la estructura IntegerStack-----");
8
9          IntegerStack stack = new IntegerStack();
10         stack.push(10);
11         stack.push(20);
12         stack.push(30);
13         System.out.println("Pila después de apilar 10, 20, 30: " + stack);
14         System.out.println("Desapilando elemento: " + stack.pop());
15         System.out.println("Pila tras desapilar elemento de la cabecera: " + stack);
16         System.out.println("Elemento en la cima: " + stack.top());
17         System.out.println("Tamaño de la pila: " + stack.size());
18         System.out.println("¿Contiene 20? " + stack.search(20));
19         System.out.println("Mostrando contenido de la pila: " + stack);
20
21         System.out.println("\n-----Probando la estructura IntegerQueue -----");
22         IntegerQueue queue = new IntegerQueue();
23         queue.insert(50);
24         queue.insert(60);
25         queue.insert(70);
26         System.out.println("Cola después de insertar 10, 20, 30: " + queue);
27         System.out.println("Sacando elemento: " + queue.remove());
28         System.out.println("Cola tras extraer elemento de la cabecera: " + queue);
29         System.out.println("Elemento en la cabecera: " + queue.seek());
30         System.out.println("Tamaño de la cola: " + queue.size());
31         System.out.println("¿Contiene 60? " + queue.search(60));
32         System.out.println("Mostrando contenido de la cola: " + queue);

```

5. Conclusiones

El uso de nodos enlazados para implementar estructuras de datos como pilas y colas permite una comprensión más profunda de cómo funciona la memoria dinámica y cómo se gestionan las referencias entre los elementos. Aunque las clases estándar como ArrayList y LinkedList son sí las había utilizado brevemente, he visto que trabajar directamente con nodos proporciona una base sólida para entender el funcionamiento interno de las listas enlazadas y las estructuras dinámicas en general.