Estructuras de datos

Pilas y colas para solución de problemática

Resultado de aprendizaje

Walter González Zúñiga

Aldo Barrera González

TID42M

Docente Milton Batres

UTCH | Universidad Tecnológica de Chihuahua

Contenido

REPORTE DE PRÁCTICA

Objetivo:

El objetivo de esta tarea es poner en práctica los conocimientos adquiridos a lo largo de la unidad 4, que comprende el uso de colas y pilas implementadas con listas doblemente enlazadas a través de diferentes problemas.

Este reporte busca documentar el proceso para resolver cada uno de los ejercicios.

Material:

- Formato de Reporte de Práctica
- Internet
- Computador
- NeoVim (lo desinstalé)

Desarrollo:

Problema 1

Planteamiento del problema

Utilizando la clase Pilas implementada con listas dobles se debe resolver lo siguiente:

> En un establo se guarda un rebaño de hipopótamos.
>
> Se desea ordenar los hipopótamos según su peso, usando dos establos adicionales. Los establos son tan angostos que los hipopótamos deben estar formados en fila (pila). Cada establo tiene una abertura que es la entrada y salida al mismo tiempo. Mostrar la pila original y la pila de hipopótamos ordenada por su peso de mayor a menor.
>
> Nota el usuario deberá introducir la pila original con los pesos de los hipopótamos (10 hipos)

Solución de problemática

Primero, se desarrolló la clase StackLinked, que hereda de DLinkedList e implementa la interfaz de Stack e iterable, esta implementación mantiene un numero limite de elementos que puede almacenar la estructura.

```
public class StackLinked<T extends Comparable<T>> extends DLinkedList<T>
implements Stack<T>, Iterable<T> {
    private long _top, _maxLength;
    private final static int SIZE = 10;
```

Se implementaron los metodos necesarios para poder obtener información y agregar siguiendo los axiomas de las pilas.

```
@Override
public boolean push(T element) {
    try {
        if (!isFull()) {
            this.add(element);
            _top++;
            return true;}
        return false;
    } catch (FullException e) {
        System.out.println("Stack is full: " + e.getMessage());
        return false;}
}
```

Ambos metodos fueron muy sencillos de implementar, ya que **push** simplemente hacía una comprobación de isFull y luego usaba el metodo add() heredado de DlinkedList.

```
      @Override
public T pop() throws EmptyStackException {
    if (isEmpty()) {
        throw new EmptyStackException();
    }

    T element = this.getLastElement().getValue();

    if (getLength() == 1) {
        root.setRight(null);
        tail.setLeft(null);
    } else {
        Nodo<T> newLast = tail.getLeft().getLeft();
        if (newLast != null) {
            newLast.setRight(null);
            tail.setLeft(newLast); }
    }
    _top--;
    setLength(getLength() - 1);
    return element; }
```

Pop hace una comprobación isEmpty() y usa el método getLastElement() heredado tambien de DlinkedList, obteniendo el ultimo valor añadido a la lista y removiendolo mediante el cambio de apuntadores de tail y del penultimo valor.

Peek hace lo mismo que pop pero sin remover el elemento.

```
@Override
public T peek() throws EmptyStackException {
    T element = this.getLastElement().getValue();
    return element;
}
```

El metodo **Sort** es el pastor en esta abstracción de hipopotamos y corrales. Se encarga de devolver una nueva pila con los hipopotamos ordenados de mayor a menor por su peso.

```
public StackLinked<T> sort() {
    StackLinked<T> tmpSort = new StackLinked<T>();
    while (!this.isEmpty()) {
        T element = this.pop();
        while (!tmpSort.isEmpty() && tmpSort.peek().compareTo(element) <
0) {
            this.push(tmpSort.pop());
        }
        tmpSort.push(element);
    }
    return tmpSort;
}
```

Primero, se crea una nueva pila con el nombre de tmpSort y comenzamos a recorrer **this** hasta vaciarlo.

Guardamos el elemento superior the **this** en una variable **element**.

Un bucle while anidado procede a verificar dos condiciones:

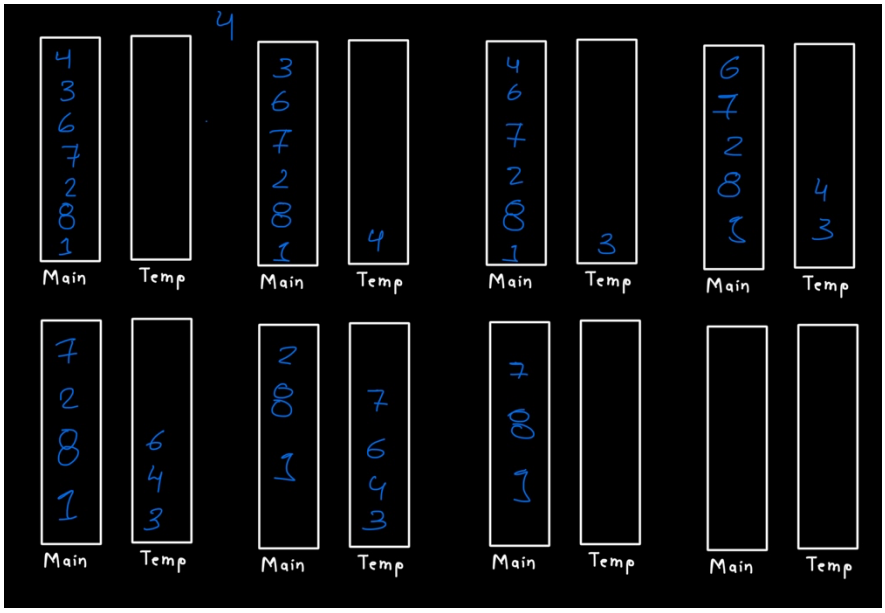
- ✅ La pila temporal (**tmpSort**) no está vacía.
- ✅ El elemento superior de **tmpSort** es menor que el **element**

Si se cumplen las condiciones del bucle anidado, el elemento superior de **tmpSort** se extrae y se coloca de nuevo en la pila original. Esto mueve temporalmente los elementos más pequeños de vuelta a la pila original hasta encontrar la posición correcta para **element**.

Una vez que se encuentra la posición correcta para **element** (es decir, cuando **tmpSort** está vacía o el elemento superior de **tmpSort** es mayor o igual a **element**), **element** se inserta en **tmpSort**.

Después de que todos los elementos de la pila original han sido procesados e insertados en **tmpSort**, el método devuelve **tmpSort**, que ahora contiene todos los elementos ordenados en orden descendente.

Aunque resuelve el problema, este algoritmo no resulta eficiente, alcanzando una complejidad $O(n^2)$.



Este fue el boceto con el cual se pudo obtener el algoritmo de ordenamiento, se cambió la implementación para ordenar de más grande a chico.

Este ejercicio se procesa en main, donde se crea una pila establo y pide al usuario ingresar los pesos de 10 hipopotamos, posteriormente, imprime la pila, la ordena y vuelve a imprimir el establo ordenado.

```
public static void EstabloTest() {
    BufferedReader reader = new BufferedReader(new InputStreamReader
(System.in));
    StackLinked<Double> establo = new StackLinked<Double>();

    System.out.println("Ingrese el peso de 10 hipopótamos:");

    try {
        for (int i = 1; i <= 10; i++) {
            System.out.print("Peso del hipopótamo " + i + ": ");
            Double peso = Double.parseDouble(reader.readLine());
            establo.push(peso);
        }

        System.out.println("Establo original: " + establo);
        StackLinked<Double> establoOrdenado = establo.sort();
        System.out.println("Establo ordenado: " +
establoOrdenado.toString());

    } catch (IOException e) {
        System.out.println("Error de lectura: " + e.getMessage());
    } catch (NumberFormatException e) {
        System.out.println("Error: Por favor ingrese un número válido");
    }
}
```

Problema 2

Planteamiento del problema

Desarrollar un programa de Agenda que almacene las actividades diarias en una cola implementada con listas dobles. El programa debe incluir las siguientes funciones:

1. Gestión de actividades:
   - Agregar nueva actividad (estado: activa)
   - Eliminar actividad específica
   - Eliminar todas las actividades
   - Cambiar estado de actividad a pendiente
   - Trasladar actividad actual al final de la cola
   - Mostrar actividades activas
   - Mostrar actividades pendientes

2. Cada actividad debe contener:
   - Número identificador (generado automáticamente)
   - Nombre de la actividad
   - Hora programada
   - Estado (activa/pendiente)

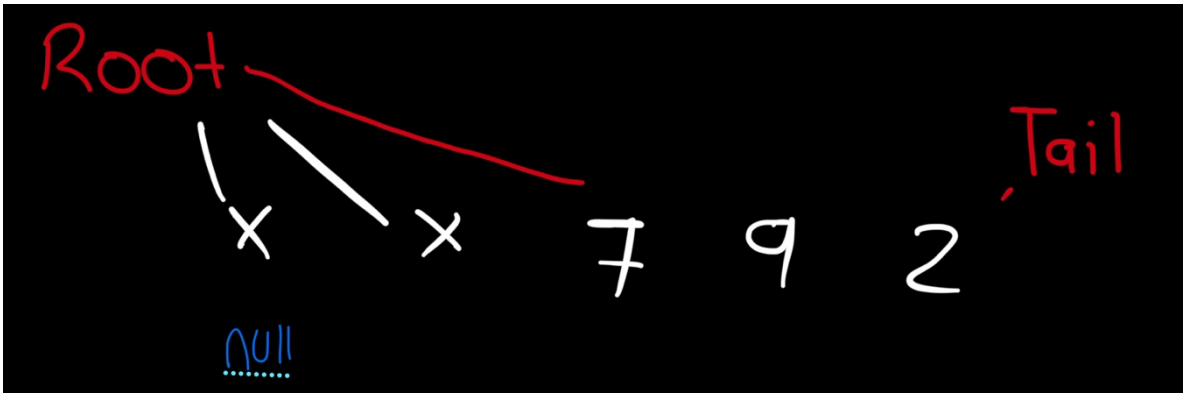
Implement un menú interactivo que permita acceder a todas estas funcionalidades.

Solución de problemática

Una cola hecha a partir de listas dobles es sumamente sencilla y no requiere de muchos cambios al momento de heredar los métodos y la estructura desde la clase DLinkedList. Hubo un momento de mirar al vacío después de escribir estas líneas:

```
public QueueLinked() {
    super();
    // aqui fue donde me di cuenta, despues de ver la pantalla fijamente
por 5 minutos
    // que las listas ya son colas. }
```

Al momento de realizar que no necesitaba muchos cambios y que la lista doble ya es practicamente una cola, se eliminó la longitud máxima que podría tener para aprovechar la versatilidad de las listas dobles.



Momento exacto donde se comprendió que ya se trataba de una cola.

El método **enqueue**, son simples llamdas al metodo de la superclase **add()**, no es necesario realizar cambios, siempre se agregan al final de la cola.

```
@Override
public void enqueue(T element){
    super.add(element);
}
```

El metodo **dequeue** almacena el primer elemento de la cola y se encarga de redirigir a root hacia el segundo elemento y retornamos el primero.

```
@Override
public T dequeue() {
    T element = root.getRight().getValue();
    root.setRight(root.getRight().getRight());
    return element;
}
```

Así, nuestra estructura de datos está lista para funcionar como cola a través de listas dobles.

Después, se creó la clase tareas para almacenar información de cada una.

```
public class Tareas {
    private static int idCounter = 0;
    private int numero;
    private String nombre;
    private String hora;
    private Boolean status;
```

El atributo idCounter es estático para poder asignar id´s unicos facilmente y los constructores asignan reciben los datos para llenar estos atributos.

Se creó una clase Agenda, la cual se encarga de manejar la logica de este ejercicio.

Un objeto agenda posee dos colas, una de actividades pendientes y otra de actividades activas, de igual forma, se inicializa un lector de teclado con Buffered Reader.

```
public class Agenda {
    private QueueLinked<Tareas> activas;
    private QueueLinked<Tareas> pendientes;
    private BufferedReader reader;

public Agenda() {
    activas = new QueueLinked<>();
    pendientes = new QueueLinked<>();
    reader = new BufferedReader(new InputStreamReader(System.in));
}
```

Aqui, integramos un metodo **test()** que lleva al usuario a un menú donde puede interactuar con todos los metodos de la agenda.

InsertarActividad recibe los datos de la tarea, crea una nueva tarea con la información y la agrega a la cola de tareas activas.

```
public void insertarActividad(String nombre, String hora) {
    Tareas nueva = new Tareas(nombre, hora);
    activas.offer(nueva);
}
```

EliminarActividad extrae y devuelve la primera tarea de la cola de tareas activas, eliminándola de la misma.

```
public Tareas eliminarActividad() {
    return activas.dequeue();
}
```

EliminarTodasActividades vacía por completo tanto la cola de tareas activas como la cola de tareas pendientes, eliminando todas las tareas existentes en ambas colas.

```
public void eliminarTodasActividades() {
    while (!activas.isEmpty()) {
        activas.dequeue();
    }
    while (!pendientes.isEmpty()) {
        pendientes.dequeue();
    }
}
(Aunque ahora viendolo, sería mas eficiente crear nuevas instancias)
```

CancelarActividad toma la primera tarea de la cola de tareas activas, cambia su estado a inactivo (false) y la mueve a la cola de tareas pendientes.

```
public void cancelarActividad() {
    if (!activas.isEmpty()) {
        Tareas cancelada = activas.pull();
        cancelada.setStatus(false);
        pendientes.offer(cancelada);
    }
}
```

MoverAlFinal extrae la primera tarea de la cola de tareas activas y la coloca al final de la misma cola, reubicándola como última tarea.

```
public void moverAlFinal() {
    if (!activas.isEmpty()) {
        Tareas actual = activas.pull();
        activas.offer(actual);
    }
}
```

Estos métodos devuelven una representación en texto del contenido de las colas: **verActivas** muestra todas las tareas en la cola de activas, mientras que **verPendientes** muestra todas las tareas en la cola de pendientes.

```
public String verActivas () {
    return activas.toString();
}

public String verPendientes() {
    return pendientes.toString();
}
```

Implementación en main

Se desarrolló la clase principal App que contiene el menú para escoger el ejercicio a procesar. Esta implementación sirve como punto de entrada para probar las funcionalidades de la Agenda y el Establo de hipopótamos.

```
public static void main(String[] args) throws Exception {
    BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
    int opcion;

    do {
        System.out.println("\nLe menu principal");
        System.out.println("1. Test Agenda");
        System.out.println("2. Test Establo");
        System.out.println("0. Salir");
        System.out.print("Seleccione una opción: ");

        try {
            opcion = Integer.parseInt(reader.readLine());
            switch (opcion) {
                case 1:
                    Agenda agenda = new Agenda();
                    agenda.test();
                    break;
                case 2:
                    EstabloTest();
                    break;
                case 0:
                    System.out.println("Au revoir beau");
                    break;
                default:
                    System.out.println("je suis fatigué je ne comprends
pas");
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            opcion = -1;
        }
    } while (opcion != 0);

    reader.close();
}
```

En la clase main, Tambien, se incluye el metodo **EstabloTest**(), que corre el ejercicio de los hipopótamos, comentada anteriormente.

Conclusiones:

Este resultado de aprendizaje ha sido el más sencillo de los que hemos realizado para esta materia.

La herencia de DLinkedList resultó ser, otra vez, eficaz y directa, especialmente cuando nos dimos cuenta de que las listas ya eran prácticamente colas. Estos momentos de realización son los que hacen que la programación sea tan gratificante.

El ejercicio del establo de hipopótamos fue particularmente interesante, aunque nos hubiera gustado implementar un algoritmo de ordenamiento más eficiente que $O(n^2)$. Quizás en un futuro podríamos revisitar este problema e implementar una solución utilizando otros métodos adaptados a la estructura de pila.

La implementación de la agenda demostró cómo las estructuras de datos básicas pueden utilizarse para crear aplicaciones prácticas y funcionales. El uso de dos colas separadas para actividades activas y pendientes resultó ser una solución elegante que cumplió perfectamente con los requerimientos.

Y rompiendo la tradición, este ha sido el trabajo más tranquilo hasta ahora, aunque como la selección mexicana, las cosas siempre se pueden poner peor; listos para árboles.

Referencias

CS Dojo. (2020, October 16). Introduction to Stacks and Queues (Data Structures & Algorithms #12) [Video]. YouTube. https://www.youtube.com/watch?v=A3ZUpyrnCbM

ProgressiveCoder. (2020, January 17). Sort a stack using a temporary stack [Video]. YouTube. https://www.youtube.com/watch?v=K0XXVSL4wUo