Informe de Laboratorio 04
Tema: Sort y Listas Enlazadas

Nota

Estudiantes	Escuela	Asignatura
Condorios Yllapuma Jorge Cusilayme García Jose Mamani Mamani Alexis Valdivia Luna Carlo	Escuela Profesional de Ingeniería de Sistemas	Estructura de Datos Semestre: III Código: 1702124

Laboratorio	Tema	Duración
04	Sort y Listas Enlazadas	04 horas

Semestre académico	Fecha de inicio	Fecha de entrega
2023 - A	Del 7 Junio 2023	Al 14 Junio 2023

1. Tarea

- Utilizar el tipo generico de Lista Enlazada para generar los peores casos y ejecutar el algoritmo de ordenamiento.
- Utilizar el tipo generico de Doble Lista Enlazada para generar los peores casos y ejecutar el algoritmo de ordenamiento.

2. Equipos, materiales y temas utilizados

- Windows 10 Home Single Language 64 bits (10.0, compilación 19045)
- VIM 9.0.
- Git 2.40.1.
- Cuenta en GitHub con el correo institucional.
- Java

3. URL de Repositorio Github

- URL del Repositorio GitHub para clonar o recuperar.
- `https://github.com/JorgeCY21/EDA_LAB_D`
- URL para el laboratorio 04 en el Repositorio GitHub.
- `https://github.com/JorgeCY21/EDA_LAB_D/tree/main/Lab_04`

4. Resolucion de ejercicios propuestos

4.1. Ejercicio 1

- Utilizar el tipo generico de Lista Enlazada para generar los peores casos y ejecutar el algoritmo de ordenamiento.

Listing 1: Node.java

```java
class Node<T> {
    private T element; // Elemento almacenado en el nodo
    private Node<T> nextNode; // Referencia al siguiente nodo

    public Node(T element) {
        this.element = element; // Inicializa el nodo con el elemento especificado
        nextNode = null; // Establece el siguiente nodo como null
    }

    public T getElement() {
        return element; // Retorna el elemento almacenado en el nodo
    }

    public void setElement(T element) {
        this.element = element; // Establece el elemento del nodo
    }

    public Node<T> getNextNode() {
        return nextNode; // Retorna el siguiente nodo
    }

    public void setNextNode(Node<T> nextNode) {
        this.nextNode = nextNode; // Establece el siguiente nodo
    }
}
```

- Este código define la clase Node que representa un nodo en una estructura de datos enlazada. Los nodos en una estructura de datos enlazada se utilizan para almacenar valores y mantener la conexión entre los elementos de la estructura.
- La clase Node utiliza un parámetro de tipo genérico T para permitir la flexibilidad de almacenar diferentes tipos de elementos en los nodos.
- El constructor Node se utiliza para inicializar un nodo con el elemento especificado. Establece el atributo element con el valor proporcionado y establece nextNode como null.
- En resumen, esta clase proporciona los elementos básicos para crear y manipular nodos en una estructura de datos enlazada.

Listing 2: LinkedList.java

```java
public class LinkedList<T extends Comparable<T>> {
   private Node<T> raiz; // Raíz o primer nodo de la lista enlazada
   private int tamano; // Tamaño actual de la lista enlazada

   public LinkedList() {
```

```
6       raiz = null; // Inicializa la raíz como null al crear una nueva lista enlazada
7       tamano = 0; // Inicializa el tamaño como 0 al crear una nueva lista enlazada
8    }
9
10   public void insert(T elemento) {
11      Node<T> nuevoNodo = new Node<>(elemento); // Crea un nuevo nodo con el elemento
            especificado
12      if (raiz == null) {
13         raiz = nuevoNodo; // Si la lista enlazada está vacía, el nuevo nodo se convierte en
               la raíz
14      } else {
15         Node<T> nodoActual = raiz;
16         while (nodoActual.getNextNode() != null) {
17            nodoActual = nodoActual.getNextNode(); // Itera hasta llegar al último nodo de la
                  lista
18         }
19         nodoActual.setNextNode(nuevoNodo); // Establece el siguiente nodo del último nodo
               como el nuevo nodo
20      }
21      tamano++; // Incrementa el tamaño de la lista enlazada después de la inserción
22   }
23
24   public Node<T> get(int indice) {
25      Node<T> aux = raiz;
26      for (int i = 0; i < indice; i++) {
27         aux = aux.getNextNode(); // Itera a través de la lista hasta llegar al nodo en el í
               ndice especificado
28      }
29      return aux; // Retorna el nodo en el índice especificado
30   }
31
32   public void remove(int indice) {
33      if (indice < tamano) {
34         if (indice == 0)
35            raiz = raiz.getNextNode(); // Si el índice es 0, se elimina el primer nodo (la
                  raíz)
36         else {
37            Node<T> anterior = this.get(indice - 1);
38            anterior.setNextNode(this.get(indice + 1)); // Establece el siguiente nodo del
                  nodo anterior como el siguiente nodo del nodo a eliminar
39         }
40         tamano--; // Reduce el tamaño de la lista enlazada después de la eliminación
41      }
42   }
43
44   public void selectionSort() {
45      Node<T> nodoActual = raiz;
46      while (nodoActual != null) {
47         Node<T> nodoMinimo = nodoActual;
48         Node<T> nodoComparador = nodoActual.getNextNode();
49         while (nodoComparador != null) {
50            if (nodoComparador.getElement().compareTo(nodoMinimo.getElement()) < 0) {
51               nodoMinimo = nodoComparador; // Encuentra el nodo mínimo comparando los
                     elementos y lo asigna como el nuevo nodo mínimo
52            }
53            nodoComparador = nodoComparador.getNextNode(); // Avanza al siguiente nodo en la
```

```
                comparación
            }
            swapNodes(nodoActual, nodoMinimo); // Intercambia los elementos de los nodos actual
                y mínimo
            nodoActual = nodoActual.getNextNode(); // Avanza al siguiente nodo en la lista
        }
    }

    private void swapNodes(Node<T> nodoA, Node<T> nodoB) {
        T temp = nodoA.getElement();
        nodoA.setElement(nodoB.getElement()); // Intercambia los elementos de los nodos A y B
        nodoB.setElement(temp);
    }

    public void printLinkedList() {
        Node<T> nodoActual = raiz;
        while (nodoActual != null) {
            System.out.print(nodoActual.getElement() + " "); // Imprime el elemento del nodo
                actual
            nodoActual = nodoActual.getNextNode(); // Avanza al siguiente nodo en la lista
        }
        System.out.println();
    }
}
```

- La clase LinkedList utiliza un parámetro de tipo genérico T que establece el tipo de elementos que se pueden almacenar en la lista enlazada.
- El constructor LinkedList se utiliza para inicializar una nueva lista enlazada estableciendo la raíz como null y el tamaño como 0.
- El método insert se utiliza para insertar un nuevo elemento al final de la lista enlazada. Crea un nuevo nodo con el elemento especificado y lo agrega al final de la lista ajustando las referencias de los nodos existentes.
- El método get se utiliza para obtener el nodo en un índice especificado. Itera a través de la lista enlazada hasta llegar al nodo en el índice especificado y lo devuelve.
- El método remove se utiliza para eliminar un nodo en un índice especificado. Si el índice es válido, ajusta las referencias de los nodos adyacentes para eliminar el nodo de la lista y reduce el tamaño de la lista enlazada.
- El método selectionSort se utiliza para ordenar la lista enlazada utilizando el algoritmo de ordenamiento de selección. Compara los elementos de los nodos y realiza intercambios para ordenar los nodos en orden ascendente.
- El método swapNodes se utiliza para intercambiar los elementos de dos nodos dados.
- El método printLinkedList se utiliza para imprimir los elementos de la lista enlazada en la consola.

Listing 3: Test.java

```
public class Test {

  public static void main(String[] args) {
    LinkedList<Integer> linkedList = new LinkedList<>();
```

```
5
6       int size = 10000;
7       for (int i = size; i > 0; i--) {
8         linkedList.insert(i);
9       }
10
11      System.out.println("Lista original:");
12      linkedList.printLinkedList();
13
14      long startTime = System.currentTimeMillis();
15
16      linkedList.selectionSort();
17
18      long endTime = System.currentTimeMillis();
19
20      System.out.println("Lista ordenada:");
21      linkedList.printLinkedList();
22
23      long elapsedTime = endTime - startTime;
24      System.out.println("Tiempo transcurrido: " + elapsedTime + " milisegundos");
25    }
26  }
```

- El método main crea una instancia de la clase LinkedList utilizando LinkedList¡Integer¿linkedList = new LinkedList¡¿(), lo que crea una lista enlazada que almacenará números enteros.
- Se declara y se inicializa la variable size con el valor 10000, que representa el tamaño de la lista enlazada.
- En resumen, este código crea una lista enlazada de números enteros en orden descendente, la ordena utilizando el algoritmo de ordenamiento de selección y luego imprime la lista original y la lista ordenada junto con el tiempo de ejecución del algoritmo de ordenamiento.
- Ejecucion
- Tamaño: 25

```
run:
Lista original:
25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Lista ordenada:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Tiempo transcurrido: 0 milisegundos
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Tamaño: 10000

```
run:
Lista original:
10000 9999 9998 9997 9996 9995 9994 9993 9992 9991 9990 9989 9988 9987 9986 9985 9984 9983 9982 9981 9980 9979 9978 9977 9976 9975 9974 9973
Lista ordenada:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
Tiempo transcurrido: 94 milisegundos
BUILD SUCCESSFUL (total time: 0 seconds)
```

4.1.1. Ejercicio 2

- Utilizar el tipo generico de Doble Lista Enlazada para generar los peores casos y ejecutar el algoritmo de ordenamiento.

Listing 4: Node.java

```java
package lab4;

public class Node<E> {
    private E data;           // Almacena el dato del nodo
    private Node<E> nextNode; // Referencia al siguiente nodo
    private Node<E> previousNode;// Referencia al nodo anterior

    // Constructores

    Node() {
        this.data = null;
        this.nextNode = null;
        this.previousNode = null;
    }

    Node(E data) {
        this.data = data;
        this.nextNode = null;
        this.previousNode = null;
    }

    Node(E data, Node<E> nextNode) {
        this.data = data;
        this.nextNode = nextNode;
        this.previousNode = null;
    }

    Node(E data, Node<E> nextNode, Node<E> previousNode) {
        this.data = data;
        this.nextNode = nextNode;
        this.previousNode = previousNode;
    }

    // Métodos para acceder y modificar los datos y referencias del nodo

    public E getData() {
        return data;
    }

    public void setData(E data) {
        this.data = data;
    }

    public Node<E> getNextNode() {
        return nextNode;
    }

    public void setNextNode(Node<E> nextNode) {
        this.nextNode = nextNode;
    }
```

```
51
52      public Node<E> getPreviousNode() {
53          return previousNode;
54      }
55
56      public void setPreviousNode(Node<E> previousNode) {
57          this.previousNode = previousNode;
58      }
59  }
```

- La clase Node tiene un parámetro de tipo genérico E que representa el tipo de dato que se almacenará en el nodo.
- Hay varios constructores disponibles para crear un objeto Node. Estos constructores permiten especificar el dato del nodo, así como las referencias al siguiente nodo y al nodo anterior.
- En resumen, la clase Node proporciona la estructura básica de un nodo en una estructura de datos enlazada, con métodos para acceder y modificar los datos y referencias del nodo.

Listing 5: DobleList.java

```
1   package lab4;
2
3   public class DobleList<E extends Comparable<E>> {
4       private Node<E> head; // Referencia al primer nodo
5       private Node<E> tail; // Referencia al último nodo
6       private int size;    // Tamaño de la lista
7
8       public DobleList() {
9           this.head = null; // Inicializar la lista vacía
10      }
11
12      public boolean isEmpty() {
13          return head == null && tail == null; // Verificar si la lista está vacía
14      }
15
16      public void insertAlInicio(E data) {
17          Node<E> newnode = new Node<E>(data); // Crear un nuevo nodo con el dato proporcionado
18          if (isEmpty()) {
19              head = newnode; // Si la lista está vacía, el nuevo nodo será el primer y único
                    nodo
20          } else {
21              newnode.setNextNode(head); // Establecer el siguiente nodo del nuevo nodo al
                    actual primer nodo
22              head.setPreviousNode(newnode); // Establecer el nodo anterior del actual primer
                    nodo al nuevo nodo
23              head = newnode; // Establecer el nuevo nodo como el nuevo primer nodo
24          }
25      }
26
27      public void insertFinal(E data) {
28          Node<E> newnode = new Node<E>(data); // Crear un nuevo nodo con el dato proporcionado
29          if (isEmpty()) {
30              head = newnode; // Si la lista está vacía, el nuevo nodo será el primer y último
                    nodo
```

```
31          tail = newnode;
32      } else {
33          tail.setNextNode(newnode); // Establecer el siguiente nodo del último nodo actual
                al nuevo nodo
34          tail = newnode; // Establecer el nuevo nodo como el nuevo último nodo
35      }
36  }
37
38  public int size() {
39      return size; // Obtener el tamaño de la lista
40  }
41
42  public void generarPeoresCasos() {
43      if (isEmpty() || size == 1) {
44          System.out.println("La lista no tiene nada que ordenar");
45      } else {
46          Node<E> nodoActual = head;
47          while (nodoActual != null) {
48              Node<E> minNode = nodoActual;
49              Node<E> comparerNode = nodoActual.getNextNode();
50              while (comparerNode != null) {
51                  if (comparerNode.getData().compareTo(minNode.getData()) > 0) {
52                      minNode = comparerNode;
53                  }
54                  comparerNode = comparerNode.getNextNode();
55              }
56              swapNodes(nodoActual, minNode);
57              nodoActual = nodoActual.getNextNode();
58          }
59      }
60  }
61
62  public void insertionSort() {
63      if (head == null || head.getNextNode() == null) {
64          return; // La lista está vacía o tiene solo un elemento, no es necesario ordenar
65      }
66
67      Node<E> sortedTail = head; // Puntero al final de la sección ordenada de la lista
68
69      // Iterar desde el segundo nodo hasta el final de la lista
70      Node<E> currentNode = head.getNextNode();
71      while (currentNode != null) {
72          E currentData = currentNode.getData();
73
74          // Buscar la posición correcta para insertar el nodo actual en la sección ordenada
75          Node<E> insertionPoint = sortedTail;
76          while (insertionPoint != null && insertionPoint.getData().compareTo(currentData) >
                0) {
77              insertionPoint = insertionPoint.getPreviousNode();
78          }
79
80          // Mover los nodos para insertar el nodo actual en la posición correcta
81          if (insertionPoint == null) {
82              // El nodo actual debe ser el nuevo inicio de la lista
83              sortedTail.setNextNode(currentNode.getNextNode());
84              if (currentNode.getNextNode() != null) {
```

```
                    currentNode.getNextNode().setPreviousNode(null);
                }
                currentNode.setNextNode(head);
                head.setPreviousNode(currentNode);
                head = currentNode;
            } else {
                // Insertar el nodo actual después del punto de inserción
                sortedTail.setNextNode(currentNode.getNextNode());
                if (currentNode.getNextNode() != null) {
                    currentNode.getNextNode().setPreviousNode(sortedTail);
                }
                currentNode.setNextNode(insertionPoint.getNextNode());
                if (insertionPoint.getNextNode() != null) {
                    insertionPoint.getNextNode().setPreviousNode(currentNode);
                }
                insertionPoint.setNextNode(currentNode);
                currentNode.setPreviousNode(insertionPoint);
            }

            // Actualizar el puntero al final de la sección ordenada
            if (sortedTail.getNextNode() != null) {
                sortedTail = sortedTail.getNextNode();
            }

            // Mover al siguiente nodo para su comparación e inserción
            currentNode = sortedTail.getNextNode();
        }
    }

    private void swapNodes(Node<E> nodeA, Node<E> nodeB) {
        if (nodeA == nodeB) {
            return; // Los nodos son iguales, no es necesario intercambiar
        }

        // Intercambio de elementos
        E temp = nodeA.getData();
        nodeA.setData(nodeB.getData());
        nodeB.setData(temp);
    }

    @Override
    public String toString() {
        Node<E> currentNode = head;
        StringBuilder cadena = new StringBuilder();
        while (currentNode != null) {
            cadena.append(currentNode.getData().toString()).append(" ");
            currentNode = currentNode.getNextNode();
        }
        return cadena.toString(); // Obtener una representación en cadena de la lista
    }
}
```

- La clase DobleList tiene un parámetro de tipo genérico E que representa el tipo de dato que se almacenará en la lista.
- La lista doblemente enlazada contiene referencias a dos nodos, head (cabeza) que apunta al

primer nodo de la lista y tail (cola) que apunta al último nodo de la lista. Además, se mantiene un contador size para realizar un seguimiento del tamaño de la lista.

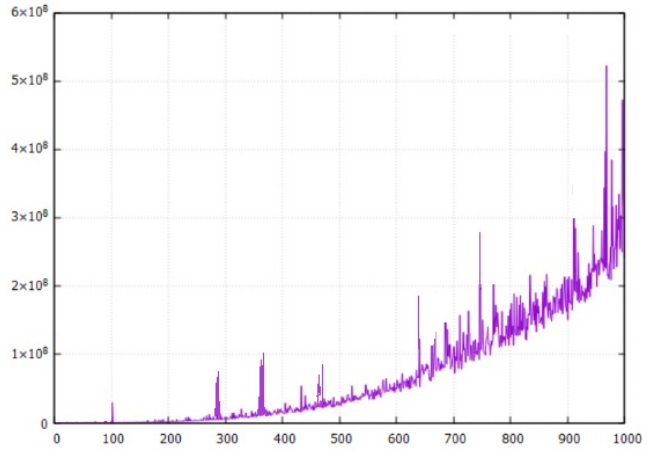
- El constructor de la clase inicializa la lista vacía estableciendo la cabeza (head) como null.
- El método isEmpty verifica si la lista está vacía verificando si tanto la cabeza (head) como la cola (tail) son null.
- Los métodos insertAlInicio e insertFinal permiten insertar un nuevo nodo al inicio y al final de la lista, respectivamente.
- El método size devuelve el tamaño actual de la lista.
- El método generarPeoresCasos implementa un algoritmo de ordenación de selección para ordenar la lista en orden descendente.
- El método insertionSort implementa un algoritmo de ordenación por inserción para ordenar la lista en orden ascendente.
- El método swapNodes intercambia los elementos de dos nodos dados.
- El método toString crea una representación de cadena de la lista concatenando los elementos de los nodos.

Listing 6: Lab4.java

```java
package lab4;

import java.io.FileWriter;
import java.io.IOException;
import com.panayotis.gnuplot.JavaPlot;

import java.util.Scanner;

public class Lab4 {
    public static void main(String[] args) {
        int tamano;
        Scanner teclado = new Scanner(System.in);
        System.out.print("Introduzca el tamaño del arreglo: ");
        tamano = teclado.nextInt();

        DobleList<Integer>[] array = new DobleList[tamano]; // Arreglo de listas doblemente
            enlazadas
        for (int i = 0; i < tamano; i++) {
            array[i] = new DobleList<>(); // Crear una nueva lista doblemente enlazada
            for (int y = 1; y <= i + 1; y++) {
                array[i].insertFinal(y); // Insertar elementos en la lista actual
            }
            array[i].generarPeoresCasos(); // Generar peores casos para la lista actual
        }

        try {
            times(array, tamano); // Calcular los tiempos de ejecución y escribir en un archivo
        } catch (IOException ex) {
            // Manejo de excepciones en caso de error de escritura en archivo
        }

```

```
31          JavaPlot p = new JavaPlot();
32          p.addPlot(" 'D:/EDA_LAB_D/Lab_04/Ejercicio02/insercion.txt' with lines"); // Agregar
                un gráfico desde un archivo de datos
33          p.plot(); // Mostrar el gráfico
34
35      }
36
37      public static void times(DobleList array[], int tamano) throws IOException {
38
39          String archivoInsersion = "insercion.txt"; // Nombre del archivo de salida
40          FileWriter escritor = new FileWriter(archivoInsersion); // Crear un escritor de
                archivos
41          // escritor.write("Numero\tTiempo\n");
42
43          for (int i = 0; i < tamano; i++) {
44              long nano_start = System.nanoTime(); // Tiempo inicial en nanosegundos
45              array[i].insertionSort(); // Ordenar la lista actual
46              long nano_end = System.nanoTime(); // Tiempo final en nanosegundos
47              long time = nano_end - nano_start; // Calcular el tiempo de ejecución
48              escritor.write(i + 1 + "\t" + time + "\n"); // Escribir el número de elementos y
                    el tiempo en el archivo
49          }
50          escritor.close(); // Cerrar el escritor de archivos
51      }
52
53  }
```

- El programa solicita al usuario que ingrese el tamaño de un arreglo mediante la clase Scanner.
- Se crea un arreglo de tipo DobleList¡Integer¿llamado array, que contendrá tamano listas doblemente enlazadas.
- Se itera sobre el arreglo array y se crea una nueva lista doblemente enlazada en cada posición. Luego se insertan elementos en cada lista de acuerdo a su posición en el arreglo.
- Se genera peores casos para cada lista del arreglo utilizando el método generarPeoresCasos de la clase DobleList.
- Se llama al método times para calcular los tiempos de ejecución y escribirlos en un archivo.
- Se crea una instancia de la clase JavaPlot para mostrar un gráfico.
- Se agrega un gráfico desde un archivo de datos utilizando el método addPlot de la clase JavaPlot.
- Se muestra el gráfico utilizando el método plot de la clase JavaPlot.
- Ejecucion:



5. Preguntas

5.1. ¿Como se ejecutaría sus implementaciones desde terminal(consola)?

- Descarga la biblioteca JavaPlot desde el siguiente enlace https://sourceforge.net/projects/gnujavaplot/
- Extrae el archivo ZIP descargado y encuentra el archivo JAR de la biblioteca JavaPlot
- Abre la consola y navega hasta el directorio donde se encuentra el archivo JAR de JavaPlot.
- Compila tu archivo Java que utiliza JavaPlot utilizando el siguiente comando:

```
javac -cp gnuplot-java.jar TuArchivoJava.java
```

- Ejecuta tu archivo Java utilizando el siguiente comando:

```
java -cp .:gnuplot-java.jar TuArchivoJava
```

6. Conclusiones

- La implementación de una cola de prioridad utilizando un heap proporciona un tiempo de ejecución eficiente para las operaciones de inserción y eliminación de elementos con mayor prioridad.
- El uso de estructuras de datos genéricas y programación orientada a objetos permite crear implementaciones flexibles y reutilizables.
- El diseño de clases y métodos bien encapsulados promueve un código limpio y mantenible.
- La comprensión de las propiedades y operaciones del heap es fundamental para implementar correctamente una cola de prioridad.
- En este ejercicio, se ha implementado una cola de prioridad utilizando un heap como estructura de datos subyacente. La implementación del heap permite realizar las operaciones de inserción, eliminación y acceso a los elementos con mayor y menor prioridad de manera eficiente. El uso de

una clase genérica y la implementación de la interfaz Comparable proporcionan flexibilidad para trabajar con diferentes tipos de elementos y prioridades.

- La implementación de una cola de prioridad basada en un heap es una estrategia poderosa y eficiente para resolver problemas que implican la gestión de elementos con diferentes niveles de prioridad. Al comprender los conceptos y técnicas presentadas en este ejercicio, se puede aplicar este conocimiento a una amplia gama de problemas en los que se requiere un manejo eficiente de la prioridad.
- Informe a detalle en:
- `https://docs.google.com/document/d/1jAOoSbbO_PYWXzPXPUOXgS0euRT48e6vSWjYnETln9Y/edit?usp=sharing`

7. Referencias

- `https://aulavirtual.unsa.edu.pe/2023A/mod/page/view.php?id=44926`
- `https://drive.google.com/file/d/1Ma9WERoSytydCbQc9S3XOsiUbim1XXQq/view`