Proyecto de Gestor de Almacenes

I. Introduction

En el presente artículo, se explican los algoritmos y estructuras de datos utilizados en el programa, incluyendo listas enlazadas, grafos y el Árbol Binario de Búsqueda (BST). Estas herramientas permiten almacenar y organizar datos eficientemente, facilitando búsquedas y optimizando la experiencia del usuario en la aplicación. La utilización de listas enlazadas y estructuras de grafos permite resolver problemas de manera eficiente y mejorar la gestión de datos. En conjunto, estos elementos contribuyen a la funcionalidad y usabilidad del programa de manera efectiva.

II. MARCO CONCEPTUAL

En la presente sección, se procederá a explicar los algoritmos y estructuras de datos utilizados en el programa.

A. Listas:

Las listas enlazadas son estructuras de datos que consisten en una secuencia de elementos llamados nodos. Cada nodo contiene un valor o dato y un puntero (enlace) que apunta al siguiente nodo en la lista. En el caso de las listas simplemente enlazadas, cada nodo sólo tiene un enlace que apunta al siguiente nodo en la secuencia, mientras que en las listas doblemente enlazadas, cada nodo tiene dos enlaces: uno que apunta al nodo anterior y otro que apunta al siguiente nodo.

La lista enlazada básica es la lista enlazada simple la cual tiene un enlace por nodo. Este enlace apunta al siguiente nodo en la lista, o al valor NULL o a la lista vacía, si es el último nodo.[1]Figura 1

Lista doblemente enlazadas

Un tipo de lista enlazada más sofisticado es la lista doblemente enlazada o lista enlazadas de dos vías. Cada nodo tiene dos enlaces: uno apunta al nodo anterior, o apunta al valor NULL si es el primer nodo; y otro que apunta al nodo siguiente, o apunta al valor NULL si es el último nodo. Figura 2

Operaciones con listas

El tipo de datos Lista debe tener las siguientes propiedades:
• Determinar si está vacía la lista.
• Limpiar la lista.
• Agregar un elemento.
• Encontrar un elemento. (por valor).
•Actualizar un elemento.
• Eliminar un elemento.

B. Grafos:

Los grafos son estructuras matemáticas y abstractas que consisten en un conjunto de nodos (vértices) conectados entre sí por medio de aristas (arcos). Un grafo G se representa mediante un par ordenado G = (V, E), donde V es el conjunto de vértices o nodos y E es el conjunto de aristas que representan las conexiones entre los vértices.

Estructura de datos matemática y abstracta que consiste en un conjunto de nodos o vértices conectados entre sí mediante aristas o arcos. Cuenta con un par ordenado G = (V, E), donde V es el conjunto de vértices o nodos, y E es el conjunto de aristas que representan las conexiones entre los vértices. Cada arista en E puede estar asociada con uno o más vértices y representa una relación entre ellos.[8]

Cuenta con algoritmos como el de búsqueda en anchura, búsqueda en profundidad y el Dijkstra.

También cuenta 3 con tipos:

- Dirigido: Las aristas tienen una dirección específica
- No dirigido: La relación entre dos vértices es bidireccional.
- Ponderado: Las aristas tienen un valor numérico asociado, llamado peso o costo.

Los grafos son ampliamente utilizados en la modelación de diversas situaciones y problemas en ciencias de la computación, como la representación de redes de computadoras, relaciones sociales, rutas de transporte, entre otros. Existen algoritmos y técnicas específicas para trabajar con grafos, como el recorrido en profundidad (DFS) y el recorrido en amplitud (BFS), que permiten explorar sus elementos y propiedades de manera eficiente.

La clase llamada GraphLink<E> es como la médula espinal de esta sección de grafos. Este mismo representa un grafo dirigido ponderado utilizando una lista enlazada para almacenar los vértices y sus aristas. Su estructura se puede explicar definiendo sus métodos:

. public ListLinked<Vertex<E>> listVertex: Un atributo que representa la lista enlazada de vértices del grafo. Cada nodo de esta lista contiene un objeto de tipo Vertex<E>, que representa un vértice en el grafo.

```
11  public class Vertex<E> implements Comparable<Vertex<E>> {
12      protected E data;
13      protected ListLinked<Edge<E>> listAdj;
14      protected int dist;
15      protected boolean visited;
16      private Vertex<E> prev;
17
18      public Vertex(E data) {
19          this.data = data;
            this.listAdj = new ListLinked<Edge<E>>();
21          this.dist = Integer.MAX_VALUE;
22          this.visited = false;
23      }
```

. **public GraphLink():** Constructor de la clase GraphLink. Inicializa la lista enlaz

```java
public class GraphLink<E> {

    protected ListLinked<Vertex<E>> listVertex;

    public GraphLink() {
        this.listVertex = new ListLinked<Vertex<E>>();
    }

    public void insertVertex(E data) {
        Vertex<E> v = new Vertex<E>(data);
        if (this.listVertex.search(v)) {
            System.out.println("Vertice con " + data + " ya fue insertado");
        } else {
            this.listVertex.insertFirst(v);
        }
    }
```

ada de vértices del grafo.

. **public void insertVertex(E data):** Método que permite insertar un nuevo vértice en el grafo. Crea un nuevo objeto Vertex<E> con el dato proporcionado y lo inserta al principio de la lista enlazada de vértices, siempre que el vértice con el mismo dato no exista previamente.

```java
    public void insertVertex(E data) {
        Vertex<E> v = new Vertex<E>(data);
        if (this.listVertex.search(v)) {
            System.out.println("Vertice con " + data + " ya fue insertado");
        } else {
            this.listVertex.insertFirst(v);
        }
    }
```

. **public void insertEdge(E dataOri, E dataDes, int weight):** Método que permite insertar una arista (con peso) entre dos vértices existentes en el grafo. Primero busca los vértices de origen y destino en la lista enlazada de vértices. Si ambos vértices existen, crea una nueva arista y la inserta al principio de la lista de adyacencia del vértice de origen.

```java
    public void insertEdge(E dataOri, E dataDes, int weight) {
        Vertex<E> vOri = this.listVertex.searchData(new Vertex<E>(dataOri));
        Vertex<E> vDes = this.listVertex.searchData(new Vertex<E>(dataDes));

        if (vOri == null || vDes == null) {
            System.out.println(dataOri + " o " + dataDes + " no existen....");
        } else {
            Edge<E> e = new Edge<E>(vDes, weight);
            if (vOri.listAdj.search(e)) {
                System.out.println("Arista (" + dataOri + "," + dataDes + ") ya fue insertada...");
            } else {
                vOri.listAdj.insertFirst(e);
                // No necesitas insertar una arista en vDes, ya que vOri es la referencia de destino
            }
        }
    }
```

. **public void removeEdge(E dataOri, E dataDes):** Método que permite eliminar una arista entre dos vértices existentes en el grafo. Busca los vértices de origen y destino en la lista enlazada de vértices y elimina la arista de la lista de adyacencia del vértice de origen y del vértice de destino.

```java
    public void removeEdge(E dataOri, E dataDes) {
        Vertex<E> vOri = this.listVertex.searchData(new Vertex<E>(dataOri));
        Vertex<E> vDes = this.listVertex.searchData(new Vertex<E>(dataDes));

        if (vOri == null || vDes == null) {
            System.out.println(dataOri + " o " + dataDes + " no existen....");
        } else {
            vOri.listAdj.remove(new Edge<E>(vDes));
            vDes.listAdj.remove(new Edge<E>(vOri));
        }
    }
```

. **public void removeVertex(E data):** Método que permite eliminar un vértice y todas sus aristas del grafo. Busca el vértice en la lista enlazada de vértices y elimina todas las aristas que lo conectan con otros vértices.

```java
    public void removeVertex(E data) {
        Vertex<E> v = this.listVertex.searchData(new Vertex<E>(data));
        if (v == null) {
            System.out.println(data + " no existe....");
        } else {
            for (Node<Vertex<E>> aux = this.listVertex.getHead(); aux != null;
                    aux = aux.getNext()) {
                aux.getData().listAdj.remove(new Edge<E>(v));
            }
            this.listVertex.remove(v);
        }
    }
```

. **public boolean searchEdge(E dataOri, E dataDes):** Método que busca una arista entre dos vértices existentes en el grafo. Retorna true si la arista existe y false en caso contrario.

```java
    public boolean searchEdge(E dataOri, E dataDes) {
        Vertex<E> vOri = this.listVertex.searchData(new Vertex<E>(dataOri));
        Vertex<E> vDes = this.listVertex.searchData(new Vertex<E>(dataDes));

        if (vOri == null || vDes == null) {
            return false;
        } else {
            return vOri.listAdj.search(new Edge<E>(vDes));
        }
    }
```

. **public boolean searchVertex(E data):** Método que busca un vértice en el grafo. Retorna true si el vértice existe y false en caso contrario.

```java
    public void removeVertex(E data) {
        Vertex<E> v = this.listVertex.searchData(new Vertex<E>(data));
        if (v == null) {
            System.out.println(data + " no existe....");
        } else {
            for (Node<Vertex<E>> aux = this.listVertex.getHead(); aux != null;
                    aux = aux.getNext()) {
                aux.getData().listAdj.remove(new Edge<E>(v));
            }
            this.listVertex.remove(v);
        }
    }
```

. **public ResultDistr dijkstra(E initialVertex, E finalVertex):** Método que implementa el algoritmo de Dijkstra para encontrar el camino más corto entre dos vértices en el grafo. Utiliza una cola de prioridad para explorar los vértices en el orden de menor distancia desde el vértice inicial hasta el vértice final. Devuelve un objeto ResultDistr que contiene el camino más corto como una lista de vértices y la distancia total del camino.

```java
public ResultDistr dijkstra(E initialVertex, E finalVertex) {
    Vertex<E> s = this.listVertex.searchData(new Vertex<E>(initialVertex));
    Vertex<E> f = this.listVertex.searchData(new Vertex<E>(finalVertex));

    s.setDist(0);

    PriorityQueue<Vertex<E>> queue = new PriorityQueue<>();
    queue.offer(s);

    while (!queue.isEmpty()) {
        Vertex<E> u = queue.poll();
        if (u.equals(f)) {
            break;
        }
        u.setVisited(true);
        for (Node<Edge<E>> aux = u.listAdj.getHead(); aux != null; aux = aux.getNext()) {
            Vertex<E> v = aux.getData().refdest;

            if (!v.isVisited() && u.getDist() != Integer.MAX_VALUE && u.getDist()
                    + aux.getData().weight < v.getDist()) {
                v.setDist(u.getDist() + aux.getData().weight);

                v.setPrev(u);

                queue.offer(v);
            }
        }
    }

    List<Almacen> shortestPath = new ArrayList<>();

    int totalDistance = 0;

    Vertex<E> current = f;

    while (current != null) {
        shortestPath.add(0, (Almacen) current.getData());
        totalDistance += current.getDist();
        current = current.getPrev();
    }
    return new ResultDistr(shortestPath, totalDistance);
}
```

Básicamente, esta clase GraphLink<E> proporciona una implementación de un grafo dirigido ponderado utilizando una lista enlazada para almacenar los vértices y sus aristas. Además, incluye el algoritmo de Dijkstra para encontrar el camino más corto entre dos vértices en el grafo.

C. Binary Search Tree (BST):

Estructura de datos de árbol en la que cada nodo tiene un valor y cumple con la propiedad de que el valor de cada nodo en el subárbol izquierdo es menor que el valor del nodo raíz, y el valor de cada nodo en el subárbol derecho es mayor que el valor del nodo raíz.[7]

```java
private NodoArbol<E> eliminarNodo(NodoArbol<E> actual, E dato, NodoArbol<E> nuevoNodoRaiz) {
    if (actual == null) {
        return null;
    }

    int comparacion = dato.compareTo(actual.getData());

    if (comparacion < 0) {
        actual.setIzquierda(eliminarNodo(actual.getIzquierda(), dato, nuevoNodoRaiz));
    } else if (comparacion > 0) {
        actual.setDerecha(eliminarNodo(actual.getDerecha(), dato, nuevoNodoRaiz));
    } else {
        // El nodo a eliminar es el actual

        // Caso 1: El nodo tiene 0 o 1 hijo
        if (actual.getIzquierda() == null) {
            if (actual.getDerecha() == null) {
                nuevoNodoRaiz.setData(actual.getData());  // Guardar el nodo que será eliminado
            }
            return actual.getDerecha();
        } else if (actual.getDerecha() == null) {
            nuevoNodoRaiz.setData(actual.getData());  // Guardar el nodo que será eliminado
            return actual.getIzquierda();
        }

        // Caso 2: El nodo tiene 2 hijos
        nuevoNodoRaiz.setData(actual.getData());  // Guardar el nodo que será eliminado
        actual.setData(encontrarMinimo(actual.getDerecha()));
        actual.setDerecha(eliminarNodo(actual.getDerecha(), actual.getData(), nuevoNodoRaiz));
    }

    return actual;
}
```

Esta propiedad permite realizar búsquedas, inserciones y eliminaciones de manera eficiente, ya que al seguir el camino adecuado en el árbol según el valor buscado o a insertar, se reduce significativamente el espacio de búsqueda. En un BST bien construido, el tiempo de búsqueda, inserción y eliminación es O(log n) en promedio, donde "n" es el número de nodos en el árbol.

Esta estructura posee métodos que varían en funcionalidad según se determinó durante la creación, estos pueden explicarse así:

1. ArbolBinario(): Constructor de la clase ArbolBinario. Inicializa el árbol estableciendo la raíz como null.

```java
public class ArbolBinario<E extends Comparable<E>> {

    private NodoArbol<E> raiz;

    public ArbolBinario() {
        this.raiz = null;
    }
```

2. insertar(E data): Método público que permite insertar un nuevo elemento data en el árbol. La inserción se realiza de manera recursiva respetando las propiedades del árbol binario de búsqueda, donde los elementos menores se colocan a la izquierda y los elementos mayores a la derecha.

```java
    public void insertar(E data) {
        this.raiz = insertarRecursivo(this.raiz, data);
    }
```

3. insertarRecursivo(NodoArbol<E> nodo, E data): Método privado que realiza la inserción de manera recursiva. Si el nodo es nulo, crea un nuevo nodo con el valor data. Si data es menor que el valor del nodo actual, la inserción se realiza en el subárbol izquierdo; si es mayor, en el subárbol derecho. Retorna el nodo actualizado.

```java
    private NodoArbol<E> insertarRecursivo(NodoArbol<E> nodo, E data) {
        if (nodo == null) {
            return new NodoArbol<>(data);
        }

        if (data.compareTo(nodo.getData()) < 0) {
            nodo.setIzquierda(insertarRecursivo(nodo.getIzquierda(), data));
        } else if (data.compareTo(nodo.getData()) > 0) {
            nodo.setDerecha(insertarRecursivo(nodo.getDerecha(), data));
        }

        return nodo;
    }
```

4. buscar(E data): Método público que permite buscar un elemento data en el árbol. La búsqueda se realiza de manera recursiva comparando el valor del nodo actual con data. Si son iguales, se devuelve el valor del nodo; si data es menor, se busca en el subárbol izquierdo; si es mayor, en el subárbol derecho.

```java
    private E buscarRecursivo(NodoArbol<E> nodo, E data) {
        if (nodo == null) {
            return null;
        }

        if (data.compareTo(nodo.getData()) == 0) {
            return nodo.getData();
        } else if (data.compareTo(nodo.getData()) < 0) {
            return buscarRecursivo(nodo.getIzquierda(), data);
        } else {
            return buscarRecursivo(nodo.getDerecha(), data);
        }
    }
```

5. buscarRecursivo(NodoArbol<E> nodo, E data): Método privado que realiza la búsqueda de manera recursiva.

```java
private E buscarRecursivo(NodoArbol<E> nodo, E data) {
    if (nodo == null) {
        return null;
    }

    if (data.compareTo(nodo.getData()) == 0) {
        return nodo.getData();
    } else if (data.compareTo(nodo.getData()) < 0) {
        return buscarRecursivo(nodo.getIzquierda(), data);
    } else {
        return buscarRecursivo(nodo.getDerecha(), data);
    }
}
```

6. eliminar(E dato): Método público que permite eliminar un elemento dato del árbol. La eliminación se realiza de manera recursiva y se manejan diferentes casos según el número de hijos que tenga el nodo a eliminar.

```java
public E eliminar(E dato) {
    NodoArbol<E> nuevoNodoRaiz = new NodoArbol<>(null);
    eliminarNodo(this.raiz, dato, nuevoNodoRaiz);
    return nuevoNodoRaiz.getData();
}
```

7. eliminarNodo(NodoArbol<E> actual, E dato, NodoArbol<E> nuevoNodoRaiz): Método privado que realiza la eliminación de manera recursiva.

8. encontrarMinimo(NodoArbol<E> actual): Método privado que encuentra el valor mínimo en el subárbol dado. Se utiliza en el caso de eliminación de un nodo con dos hijos.

```java
private E encontrarMinimo(NodoArbol<E> actual) {
    while (actual.getIzquierda() != null) {
        actual = actual.getIzquierda();
    }
    return actual.getData();
}
```

9. recorridoInorden(): Método público que realiza un recorrido inorden del árbol, que imprime los elementos en orden ascendente.

```java
public void recorridoInorden() {
    recorridoInorden(raiz);
}
```

10. recorridoInorden(NodoArbol<E> actual): Método privado que realiza el recorrido inorden de manera recursiva.

```java
private void recorridoInorden(NodoArbol<E> actual) {
    if (actual != null) {
        recorridoInorden(actual.getIzquierda());
        System.out.println(actual.getData());
        recorridoInorden(actual.getDerecha());
    }
}
```

11. getStockTotal(): Método público que devuelve la suma del stock de todos los elementos del árbol. Esta función asume que los elementos del árbol son del tipo Producto, ya que llama al método getStock() en cada nodo.

```java
public int getStockTotal() {
    return getStockTotalRecursivo(this.getRaiz());
}
```

12. getStockTotalRecursivo(NodoArbol<E> nodo): Método privado que calcula el stock total de manera recursiva.

```java
private int getStockTotalRecursivo(NodoArbol<E> nodo) {
    if (nodo == null) {
        return 0;
    }
    Producto producto = (Producto) nodo.getData();
    return producto.getStock() + getStockTotalRecursivo(nodo.getIzquierda())
            + getStockTotalRecursivo(nodo.getDerecha());
}
```

13. getProductos(): Método público que devuelve un ArrayList con todos los elementos del árbol. También asume que los elementos del árbol son del tipo Producto.

```java
public ArrayList<E> getProductos() {
    ArrayList<E> productos = new ArrayList<>();
    getProductosRecursivo(this.raiz, productos);
    return productos;
}
```

14. getProductosRecursivo(NodoArbol<E> nodo, ArrayList<E> productos): Método privado que obtiene todos los productos del árbol de manera recursiva.

```java
private void getProductosRecursivo(NodoArbol<E> nodo, ArrayList<E> productos) {
    if (nodo != null) {
        productos.add(nodo.getData());
        getProductosRecursivo(nodo.getIzquierda(), productos);
        getProductosRecursivo(nodo.getDerecha(), productos);
    }
}
```

15. getRaiz(): Método público que devuelve el nodo raíz del árbol.

```java
public NodoArbol<E> getRaiz() {
    return raiz;
}
}
```

D. PriorityQueue:

Estructura de datos en la que los elementos se organizan en función de su prioridad. Cada elemento tiene una prioridad asociada, y los elementos con mayor prioridad son atendidos antes que los de menor prioridad.

```java
public class PriorityQueue<E extends Comparable<E>> {
    private Node<E> head;
    private Comparator<E> comparator;

    public PriorityQueue() {
        this.head = null;
    }

    public PriorityQueue(Comparator<E> comparator) {
        this.head = null;
        this.comparator = comparator; // Asigna el comparador proporcionado
    }
```

Cuenta con operaciones de inserción, extracción de elementos según su prioridad, modificar prioridades y acceder a elementos.[6]

```java
public E removeMin() {
    if (isEmpty()) {
        return null;
    } else {
        Node<E> minNode = head;
        head = head.getNext();
        minNode.setNext(null);
        return minNode.getData();
    }
}

public boolean isEmpty() {
    return head == null;
}
public void offer(E data) {
    if (isEmpty() || data.compareTo(head.getData()) < 0) {
        head = new Node<E>(data, head);
    } else {
        Node<E> tmp = head;
        while (tmp.getNext() != null && data.compareTo(tmp.getNext().getData()) >= 0) {
            tmp = tmp.getNext();
        }
        Node<E> newNode = new Node<E>(data, tmp.getNext());
        tmp.setNext(newNode);
    }
}
```

```java
public E poll() {
    if (isEmpty()) {
        return null;
    } else {
        Node<E> minNode = head;
        head = head.getNext();
        minNode.setNext(null);
        return minNode.getData();
    }
}
@Override
public String toString() {
    String str = "";
    for(Node<E> aux = this.head; aux != null; aux = aux.getNext()) {
        str = str + aux.toString() + ", ";
    }
    return str;
}
}
```

E. Algoritmo Dijkstra:

El algoritmo de Dijkstra es un algoritmo eficiente (de complejidad O(n2) donde n es el número de vértices) que sirve para encontrar el camino de coste mínimo desde un nodo origen a todos los demás nodos del grafo.[4] Figura 3

III. PROGRAMA

En esta sección se explicará a detalle el funcionamiento teórico del programa.

A) GESTOR DE ALMACENES:

En la primera parte del proyecto tenemos el gestor de almacenes. Para su implementación utilizamos la estructura de datos denominada grafo ponderado, en el cada nodo representa un almacén, las aristas representan las rutas que unen los almacenes y cuentan con un peso que indica la distancia entre los almacenes.

Se utilizó esta estructura porque el problema plantea la distribución de productos entre los almacenes y la empresa DOTA SAC. requiere de las rutas más cortas para el ahorro de combustible.

Los métodos que se requieren para el gestor de almacenes son los siguientes:

- Agregar almacén: Requiere de 3 atributos: código, nombre y dirección.

```java
public class GrafoAlmacenes extends GraphLink<Almacen> {
    private static final String PATH_ALMACENES = "archivos/almacenes.csv";
    private static final String PATH_PRODUCTOS = "archivos/productos.csv";
    private static final String PATH_RUTAS = "archivos/rutas.csv";
    public GrafoAlmacenes() {
        super();
    }
```

- Agregar almacén desde archivo: Utiliza de la coma ‘,’ como separador de campos y usa un solo registro por línea.
- Dar de baja a almacén: Al eliminar el almacén del grafo, este transportará sus productos a los almacenes cuyo stock de dichos productos sea menor que el resto de almacenes.

```java
public void eliminarAlmacen(int codigoAlmacen) {
    Almacen almacenAEliminar = buscarAlmacenPorCodigo(codigoAlmacen);
    if (almacenAEliminar != null) {
        // Buscar el almacén con menos productos
        Almacen almacenConMenosProductos = getAlmacenConMenosProductos();

        // Trasladar todos los productos del almacén eliminado al almacén con menos productos
        for (Producto producto : almacenAEliminar.getInventario().getProductos()) {
            almacenConMenosProductos.agregarProducto(producto);
        }

        // Eliminar el almacén
        super.removeVertex(almacenAEliminar);

        // Actualizar los archivos CSV
        actualizarAlmacenesCSV();
        actualizarProductosCSV();
        actualizarRutasCSV();
    } else {
        System.out.println("El almacén con código " + codigoAlmacen + " no existe.");
    }
}
```

- Buscar un almacén: Recorre los vértices del grafo según el código del almacén buscando coincidencias.

```java
public Almacen buscarAlmacenPorCodigo(int codigo) {
    for (Vertex<Almacen> vertex : this.listVertex) {
        Almacen almacen = vertex.getData();
        if (almacen.getCodigo() == codigo) {
            return almacen;
        }
    }
    return null;
}
```

- Buscar un producto en todos los almacenes: Cada almacén cuenta con métodos para mostrar los productos que tiene. Se utilizan estos métodos para encontrar un producto según su código buscando coincidencias.

```java
public Producto buscarProducto(int codigoAlmacen, int codigoProducto) {
    Almacen almacen = buscarAlmacenPorCodigo(codigoAlmacen);
    if (almacen != null) {
        return almacen.buscarProducto(codigoProducto);
    } else {
        System.out.println("El almacén con código " + codigoAlmacen + " no existe.");
        return null;
    }
}
```

- Mostrar todos los almacenes: Muestra todos los nodos del grafo.

```java
public Almacen getAlmacenConMenosProductos() {
    Almacen almacenConMenosProductos = null;
    int minStock = Integer.MAX_VALUE;

    for (Vertex<Almacen> vertex : this.listVertex) {
        Almacen almacen = vertex.getData();
        int stockTotal = almacen.getInventario().getStockTotal();
        if (stockTotal < minStock) {
            minStock = stockTotal;
            almacenConMenosProductos = almacen;
        }
    }

    return almacenConMenosProductos;
}
```

- Todos los métodos anteriores cuentan con un condicional que verifica si existe un almacén.

B) GESTOR DE PRODUCTOS

En la segunda parte del proyecto, tenemos el gestor de productos. Para su implementación utilizamos la estructura

denominada Binary Search Tree (BST), en el que cada nodo almacena un producto junto a sus atributos que son el código, la descripción y el stock. El criterio de ordenación es según el valor numérico del código, los códigos menores en valor irán a la izquierda y los mayores en valor a la derecha.

```java
public class Producto implements Comparable<Producto> {
    private int codigo;
    private String descripcion;
    private int stock;
    private int codigoAlmacen; // Nuevo campo

    public Producto(int codigo, String descripcion, int stock, int codigoAlmacen) {
        this.codigo = codigo;
        this.descripcion = descripcion;
        this.stock = stock;
        this.codigoAlmacen = codigoAlmacen; // Inicialización del nuevo campo
    }

```

Se utilizó esta estructura porque las operaciones de búsqueda, inserción y eliminación tienen una complejidad de orden logarítmica O(Log n), disminuyendo el tiempo de ejecución.

Los métodos que se requieren para el gestor de productos son los siguientes:

- Agregar productos a un almacén: Requiere como parámetros el código del producto y el código del almacén al cual será insertado.

```java
public void agregarProducto(Producto producto) {
    Producto productoExistente = inventario.buscar(producto);
    if (productoExistente != null) {
        // Si el producto ya existe en el inventario, aumentamos su stock
        productoExistente.setStock(productoExistente.getStock() + producto.getStock());
    } else {
        // Si el producto no existe en el inventario, lo insertamos
        producto.setCodigoAlmacen(this.codigo);  // Asigna el código del almacén al producto
        inventario.insertar(producto);
    }
}
```

- Agregar productos desde un archivo: Utiliza de la coma ‘,’ como separador de campos y usa un solo registro por línea.

```java
public void extraerProducto(int codigoProducto, int cantidad) {
    Producto producto = buscarProducto(codigoProducto);
    if (producto != null) {
        if (producto.getStock() >= cantidad) {
            producto.setStock(producto.getStock() - cantidad);
            System.out.println("Se extrajeron " + cantidad +
                    " unidades del producto " + producto.getDescri
        } else {
            System.out.println("No hay suficiente stock del produ
                    + producto.getDescripcion() + ".");
        }
    } else {
        System.out.println("El producto con código " + codigoProd
                + " no existe en este almacén.");
    }
}
```

- Dar de baja un producto de un almacén: Usa como parámetros el código del producto y el código del almacén. Si existe un producto reducirá su stock; si llega a cero se eliminará el nodo del producto.

```java
public Producto eliminarProducto(int codigoProducto) {
    Producto productoAEliminar = new Producto(codigoProducto, "", 0, this.codigo);
    return inventario.eliminar(productoAEliminar);
}
```

- Buscar un producto en un almacén: Recibe el código del producto y devuelve sus atributos.

```java
public Producto buscarProducto(int codigoProducto) {
    Producto productoABuscar = new Producto(codigoProducto, "", 0, this.codigo);
    return inventario.buscar(productoABuscar);
}
```

- Mostrar los productos de un almacén: Muestra los productos del almacén utilizando un recorrido inorden.

```java
public void mostrarProductos() {
    System.out.println("Productos en el almacén " + nombre + ":");
    inventario.recorridoInorden();
}
```

- Todos los métodos anteriores cuentan con un condicional que verifica si existe algún producto.

C) ESTABLECIMIENTO DE RUTAS DE DISTRIBUCIÓN

En la tercera parte del proyecto, tenemos el establecimiento de rutas de distribución. Para su implementación utilizamos el algoritmo de Dijkstra. Este algoritmo calcula la ruta más corta posible entre dos nodos y agiliza el proceso de distribución de productos.

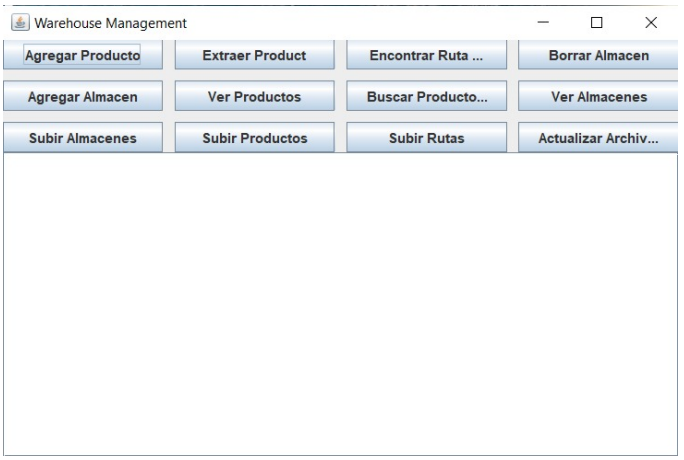
Se utilizó dicho algoritmo porque cuenta con una complejidad O((V + E) * log(V)) donde V representa el número de vértices (nodos) en el grafo y E representa el número de aristas (arcos) en el grafo y porque es el algoritmo visto en clase.

Los métodos que se requieren para el establecimiento de rutas de distribución son:

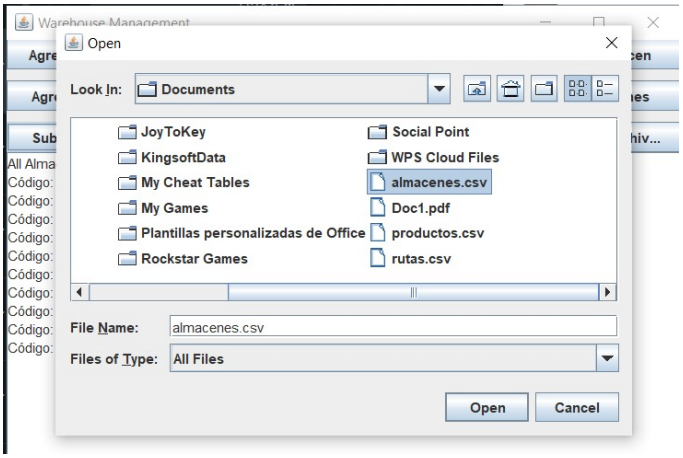
```java
public class ResultDistr {
    private List<Almacen> result;
    private int totalDistance;

    public ResultDistr(){

    }
    public ResultDistr(List<Almacen> result, int totalDistance) {
        this.result = result;
        this.totalDistance = totalDistance;
    }
```

- Agregar vías desde archivo: De ser necesario modificar el grafo añadiendo nuevas vías, se sube un archivo con las aristas y sus pesos
- Obtener las rutas de distribución: Recibe como parámetros dos nodos, uno de inicio y otro de destino. Usa el algoritmo de Dijkstra para calcular el camino más corto.

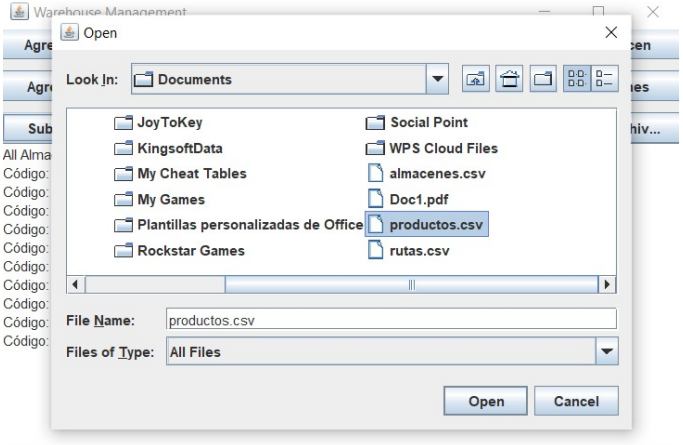
INTERFAZ GRAFICA:



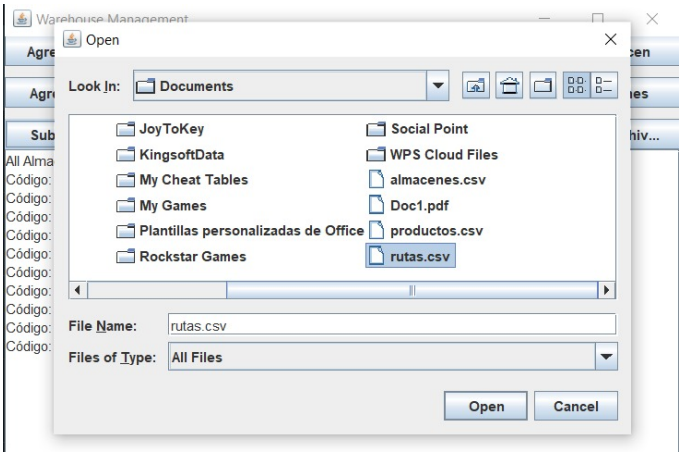
-Subir almacenes desde archivo



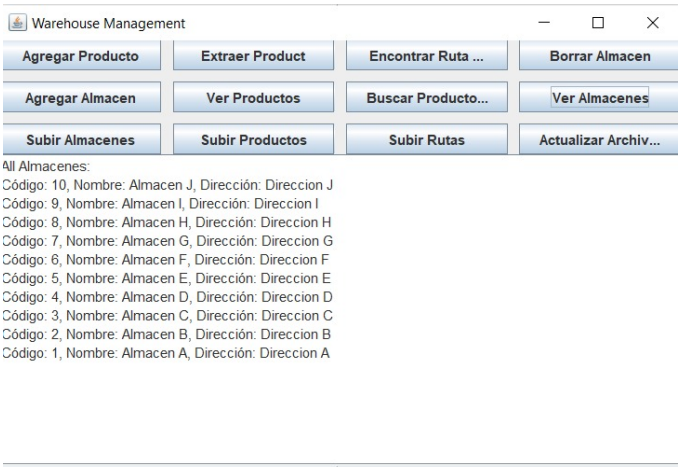
-Subir productos desde archivo



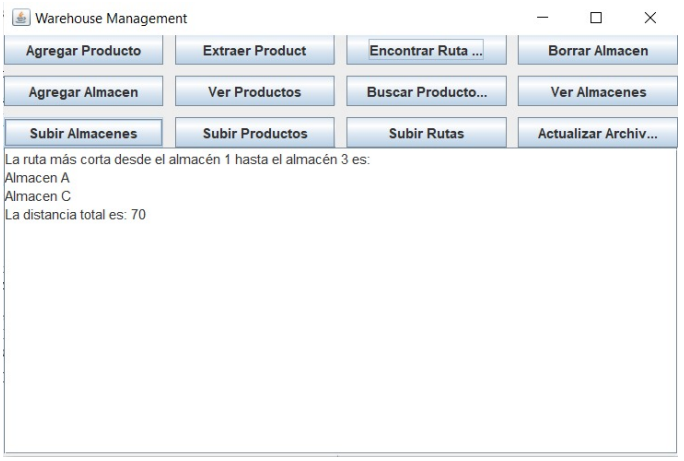
-Subir rutas desde archivo



-Ver Almacenes subidos



-Encontrar ruta más corta



IV. CONCLUSIONES

En definitiva se han presentado diferentes algoritmos y estructuras de datos utilizados en programación. Se ha explicado el funcionamiento de las listas simplemente enlazadas y doblemente enlazadas, así como las operaciones básicas que se pueden realizar con ellas, como agregar, eliminar y actualizar elementos, y encontrar un elemento por valor. También se ha mencionado la estructura de datos matemática y abstracta que es el grafo, y se han presentado algunos de sus algoritmos, como el de búsqueda en anchura, búsqueda en profundidad y el Dijkstra. Además, se ha mencionado la estructura de datos de árbol Binary Search Tree (BST) y la estructura de datos PriorityQueue. En resumen, esta sección ha proporcionado una visión general de algunas de las estructuras de datos y algoritmos más comunes utilizados en programación.

REFERENCIAS

[1] Oracle. (23 de octubre de 2021). Java Documentation.Oracle: https://docs.oracle.com/javase/tutorial/java/generics/why.html

[2] Joyanes, L., & Zahonero, I. (2002). Programación en Java 2. Algoritmos, Estructura de datos y Programación Orientada a Objetos.

[3] J. Chase y J. Lewis, *Estructura de Datos Con Java*. Pearson Educacion, 2006.

[4] Torrubia, G., & Terrazas, V. (2012). Algoritmo de Dijkstra. Un tutorial interactivo. VII Jornadas de Enseñanza Universitaria de la Informática (JENUI 2001).

[5] Gómez, Leopoldo Sebastián M. "Diseño de Interfaces de Usuario Principios, Prototipos y Heurísticas para Evaluación."

[6] D. Zhang and D. Dechev, "A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists," in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 3, pp. 613-626, 1 March 2016, doi: 10.1109/TPDS.2015.2419651.

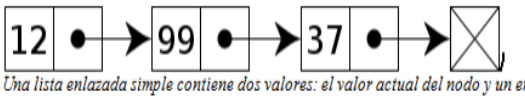
[7] Estructura de datos, algoritmos y programación orientada a objetos. Gregory L. Heileman. McGraw-Hill.

[8] Aho A.V., Hopcroft J.E., Ullman J.E. Estructuras de datos y Algoritmos. Addison-Wesley, 1988.

[9] McMillan, M. (2007). Data Structures and Algorithms. Nueva york: Cambridge University Press.

[10] Libro de M.A. Weiss, “Estructuras de Datos en Java” (AdissonWesley, 2000).

Anexos



Una lista enlazada simple contiene dos valores: el valor actual del nodo y un enlace al siguiente nodo

Figura 1

Una lista doblemente enlazada contiene tres valores: el valor, el link al nodo siguiente, y el link al anterior

Figura 2

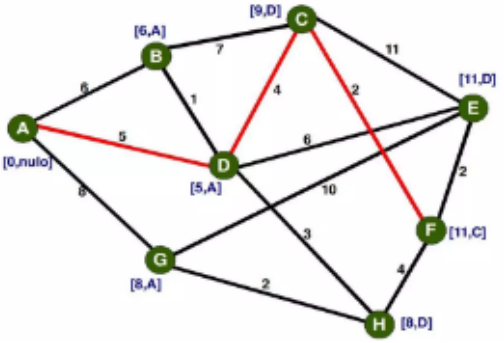


Figura 3