Sistema de Gestión y Optimización de Inventarios en Almacenes

1st Franco Salvador Paredes Zamora
Escuela Profesional de Ingeniería de Sistemas
Universidad Católica de Santa María
Arequipa, Peru
franco.paredes@estudiante.ucsm.edu.pe

2nd Piero Rait Chirinos Coarite
Escuela Profesional de Ingeniería de Sistemas
Universidad Católica de Santa María
Arequipa, Peru
piero.chirinos@estudiante.ucsm.edu.pe

3rd Rocio Sandoval Chacon
Escuela Profesional de Ingeniería de Sistemas
Universidad Católica de Santa María
Arequipa, Peru
rocio.sandoval@estudiante.ucsm.edu.pe

4th Matias Damian Valdivia Naveros
Escuela Profesional de Ingeniería de Sistemas
Universidad Católica de Santa María
Arequipa, Peru
matias.valdivia@estudiante.ucsm.edu.pe

5th Josue Vadir Cornejo Gonzales
Escuela Profesional de Ingeniería de Sistemas
Universidad Católica de Santa María
Arequipa, Peru
josue.cornejo@estudiante.ucsm.edu.pe

***Abstract*—El objetivo de este proyecto fue diseñar una aplicación que simula un almacén modelado mediante grafos dirigidos y ponderados, integrando rutas, distancias y gestión de productos. Se utilizaron tecnologías como Java, Maven y JGraphX para la visualización interactiva tanto de grafos como de árboles B+. El sistema desarrollado permite representar visualmente las ubicaciones del almacén, calcular rutas óptimas y gestionar el inventario mediante árboles B+ asociados a cada zona. La aplicación incluye un menú por consola y una interfaz gráfica que facilita la interacción con las estructuras. Durante el desarrollo, se reforzaron conceptos fundamentales sobre grafos, árboles B+ y la eficiencia de los TDA frente a otras estructuras de datos. Los resultados demuestran una gestión eficiente del inventario y optimización en la búsqueda de rutas y productos.**

***Index Terms*—grafos, Java, Maven, JGraphX, árboles B+, estructuras de datos, optimización de rutas, inventario**

I. Introducción

Este trabajo se desarrolló como parte de la asignatura Algoritmos y Estructuras de Datos en la Universidad Católica de Santa María, con el objetivo de abordar el problema de gestión y optimización en almacenes logísticos. La aplicación propuesta resuelve desafíos clave como el almacenamiento eficiente, la búsqueda de productos y la determinación de rutas cortas entre puntos del almacén. La importancia del proyecto radica en su capacidad para modelar escenarios reales de gestión de inventario, facilitando tanto la visualización como la optimización de operaciones.

El sistema fue implementado en Java, utilizando Maven como gestor de dependencias para integrar bibliotecas como JGraphX, permitiendo la representación gráfica interactiva de las estructuras. Cada nodo del grafo corresponde a una ubicación específica del almacén (estantería, pasillo, zona de carga/descarga), y está asociado internamente a un árbol B+ que gestiona los productos disponibles en dicha ubicación. De esta forma, la aplicación permite realizar búsquedas tanto locales (en un nodo específico) como globales (recorriendo todos los árboles del almacén) para localizar productos, calcular rutas óptimas y simular operaciones habituales de logística.

II. Objetivos, Requerimientos y Lineamientos

A. Objetivo General

Desarrollar una aplicación con su menú correspondiente que modele un almacén como un grafo dirigido y ponderado, e integre árboles B+ para representar la estructura interna de las categorías de productos. El sistema debe permitir la gestión eficiente de inventarios, optimización del espacio, búsqueda de productos y visualización interactiva de las estructuras.

B. Objetivos Específicos

1) Modelar las ubicaciones del almacén (estanterías, pasillos, zonas de carga/descarga) como nodos de un grafo y las rutas de movimiento entre ellas como aristas ponderadas (distancia, tiempo de desplazamiento).
2) Implementar árboles B+ para clasificar jerárquicamente los productos dentro de cada ubicación o categoría de almacén.
3) Aplicar algoritmos de grafos (Dijkstra, BFS, DFS, detección de ciclos, componentes conexas) para optimizar rutas de recolección de pedidos, reabastecimiento o movimiento de inventario.

4) Visualizar gráficamente el grafo del almacén y los árboles B+ de categorías de productos.
5) Simular escenarios como la adición/eliminación de productos, cambios en la demanda o reubicación de zonas, y recalcular las rutas óptimas y la distribución del inventario.
6) Documentar el proceso de desarrollo y presentar los resultados.

C. Descripción General

El proyecto consiste en desarrollar una aplicación que modele un almacén como un grafo dirigido y ponderado, donde cada nodo representa una ubicación logística (por ejemplo, una estantería o un pasillo) y cada arista representa una ruta de conexión entre ubicaciones. Además, cada ubicación o categoría de producto estará representada internamente como un árbol B+ que clasifica sus componentes (como tipos de productos, lotes, fechas de caducidad) según criterios definidos.

D. Requisitos Funcionales

1) **Modelado del Grafo:** Representar ubicaciones del almacén como nodos, rutas de movimiento como aristas dirigidas y ponderadas, y permitir agregar, eliminar y modificar nodos y aristas. Se debe evitar el uso de la matriz de adyacencia.
2) **Árboles B+ por Categoría/Ubicación:** Cada categoría de producto o ubicación principal debe tener un árbol B+ que clasifique sus componentes, permitiendo inserción, eliminación y búsqueda eficiente de productos.
3) **Algoritmos de Grafos:** Implementar Dijkstra o A* para rutas más cortas, BFS/DFS para exploración, detección de ciclos y componentes conexas, y análisis de zonas aisladas.
4) **Simulación de Escenarios:** Incluir cierre temporal de pasillos, recálculo de rutas ante cambios, y simulación de crecimiento de inventario.
5) **Visualización:** Mostrar el grafo con nodos y aristas etiquetadas y los árboles B+ de cada categoría o ubicación, mediante interfaz gráfica o consola interactiva.

E. Requisitos No Funcionales

- Dominio completo de cada línea de código.
- Código modular, limpio y documentado.
- Uso de estructuras de datos propias.
- Interfaz clara y funcional.
- Documentación técnica y manual de usuario.

III. Diseño del Sistema

El sistema está dividido en módulos o paquetes, siguiendo una arquitectura modular y orientada a objetos. A continuación se describen los principales paquetes y clases del proyecto.

A. Paquete `com.project_aed.btreeplus`

BPlusTree<K, V>
Estructura principal para el manejo de inventarios en cada ubicación. Implementa un árbol B+ genérico, eficiente para búsquedas, inserciones y eliminaciones.

- **Atributos:**
  - `root`: Nodo raíz del árbol B+.
  - `order`: Orden del árbol (grado máximo de los nodos).
- **Métodos principales:**
  - `search(key: K)`: Busca un producto dado su clave.
  - `insert(key: K, value: V)`: Inserta un producto en el árbol.
  - `delete(key: K)`: Elimina un producto por su clave.
  - `printTree()`, `printLeaves()`, `printTreeStructure()`: Métodos de depuración y visualización en texto.
  - `getRoot()`: Retorna el nodo raíz.

Node<K, V>
Clase interna que representa cada nodo (interno u hoja) del B+.

- **Atributos:**
  - `isLeaf`: Si el nodo es hoja.
  - `keys`: Lista de claves almacenadas.
  - `children`: Lista de hijos (si es interno).
  - `values`: Lista de valores/productos (si es hoja).
  - `next`: Puntero al siguiente nodo hoja (para recorridos eficientes).

Producto
Modelo de los productos almacenados.

- **Atributos:**
  - `nombre`, `lote`, `categoria`, `fechaCaducidad`.
- **Métodos principales:**
  - `toString()`: Retorna una representación textual del producto.

—

B. Paquete `com.project_aed.graph`

GraphLink<E>
Modela el almacén como un grafo dirigido y ponderado. Cada nodo representa una zona o ubicación; cada arista, una ruta entre zonas.

- **Atributos:**
  - `vertices`: Lista de vértices del grafo.
- **Métodos principales:**
  - `addVertex(data: E)`, `addEdge(from: E, to: E, w: Integer)`: Operaciones sobre el grafo.
  - `find(data: E)`: Busca un vértice.
  - `bfs(start: E)`, `dfs(start: E)`: Recorridos básicos.
  - `isConnected()`: Determina si el grafo es conexo.
  - `shortestPath(from: E, to: E)`: Camino más corto (Dijkstra).
  - `removeEdge`, `removeVertex`: Eliminaciones.

Vertex<E>
Vértice del grafo, asociado a una zona del almacén y a un árbol B+ de productos.

- **Atributos:**
  - `data`: Información de la ubicación.
  - `adjList`: Lista de aristas (conexiones a otros vértices).
  - `visited`: Marcador de recorrido.
- **Métodos principales:**
  - `getData()`, `getAdjList()`, `isVisited()`, `setVisited()`.

Edge<E>
Arista dirigida del grafo, representa una ruta y su peso (distancia/costo).

- **Atributos:**
  - `dest`: Vértice destino.
  - `weight`: Peso de la arista.
- **Métodos principales:**
  - `getDest()`, `getWeight()`.

—

C. Paquete `com.project_aed.graph.analysis`

Clases utilitarias para el análisis de grafos.
DigraphAnalyzer (métodos estáticos)

- `representacionFormal`, `listaAdyacencia`, `matrizAdyacencia`: Métodos para obtener representaciones formales del grafo dirigido.

GraphAnalyzer (métodos estáticos)

- `grado`, `esCamino`, `esCiclo`, `esRueda`, `esCompleto`: Análisis estructural.
- Métodos para obtener representaciones y matrices del grafo.

—

D. Paquete `com.project_aed.visual`

Clases estáticas para visualización gráfica usando JGraphX.
BPlusTreeVisualizer

- `visualize(tree, zona)`: Muestra gráficamente el árbol B+ de una zona específica.

GraphVisualizer

- `visualizeGraph(graph, directed)`: Visualiza el grafo del almacén.

—

E. Paquete `com.project_aed`

Main

- `main(args: String[])`: Punto de entrada, inicia la aplicación.
- `menuInteractivo`: Menú de operaciones por consola.
- `visualizarGrafoConEventos`: Muestra la interfaz gráfica interactiva.

—

F. Relaciones principales entre clases

- Cada `Vertex` del grafo está asociado a un `BPlusTree` para gestionar los productos en esa zona. - `Main` orquesta la interacción entre el usuario, la lógica de grafos y árboles B+, y las visualizaciones. - Los visualizadores (`GraphVisualizer` y `BPlusTreeVisualizer`) operan sobre instancias de grafos y árboles para mostrar sus estructuras de forma interactiva.

—

IV. Implementación

A. Clase `GraphLink<E>`: Estructura y Lógica de Programación

La clase `GraphLink<E>` implementa la estructura de un grafo genérico, donde cada vértice almacena información de tipo arbitrario (por ejemplo, una zona del almacén). El grafo puede ser dirigido (las conexiones tienen sentido único) o no dirigido (bidireccional).

A continuación se explica la lógica de su funcionamiento y los métodos principales, ilustrando con fragmentos de código.

1. Declaración y atributos

El grafo guarda todos sus vértices en una lista enlazada y mantiene una bandera para indicar si es dirigido:

```
private final boolean directed;
private final LinkedList<Vertex<E>> vertices = new
    LinkedList<>();
```

- `directed`: determina si las conexiones son unidireccionales o bidireccionales. - `vertices`: contiene todos los vértices (zonas del almacén).

2. Agregar un vértice: `addVertex(E data)`

Este método añade un nuevo vértice solo si no existe ya (usando el método `equals` de la clase `Vertex`):

```
public void addVertex(E data) {
    Vertex<E> v = new Vertex<>(data);
    if (!vertices.contains(v))
        vertices.addLast(v);
}
```

Lógica: - Se crea un nuevo objeto `Vertex` con el dato dado. - Si no está ya en la lista de vértices, se agrega al final. - Esto evita duplicados en la estructura del grafo.

3. Buscar un vértice: `find(E data)`

Permite buscar y obtener un vértice a partir de su dato asociado:

```
public Vertex<E> find(E data) {
    for (Vertex<E> v : vertices)
        if (Objects.equals(v.getData(), data))
            return v;
    return null;
}
```

Lógica: - Recorre toda la lista de vértices. - Usa `Objects.equals` para comparar el dato de cada vértice con el buscado. - Si lo encuentra, lo retorna; si no, retorna `null`.

4. Agregar una arista: `addEdge(E from, E to, Integer w)`

Este método conecta dos vértices (nodos) mediante una arista, que puede tener un peso (por ejemplo, distancia o costo):

```
public void addEdge(E from, E to, Integer w) {
    Vertex<E> v1 = find(from), v2 = find(to);
    if (v1 == null || v2 == null)
        return;
    v1.getAdjList().addLast(new Edge<>(v2, w));
    if (!directed)
        v2.getAdjList().addLast(new Edge<>(v1, w));
}
```

Lógica: - Busca los dos vértices involucrados. Si no existen, no hace nada. - Añade la arista al vértice de origen (`v1`) apuntando al de destino (`v2`) con el peso dado. - Si el grafo es no dirigido, repite el proceso al revés, asegurando que ambos vértices conozcan la conexión.

5. Recorridos: Profundidad (DFS) y Anchura (BFS)

Ambos métodos permiten visitar todas las zonas alcanzables desde una ubicación inicial, pero siguen estrategias distintas.

DFS (búsqueda en profundidad):

```
public void dfs(E start) {
    Vertex<E> s = find(start);
    if (s == null)
        return;
    Deque<Vertex<E>> stack = new ArrayDeque<>();
    stack.push(s);

    while (!stack.isEmpty()) {
        Vertex<E> v = stack.pop();
        if (!v.visited) {
            v.visited = true;
            System.out.print(v.getData() + " ");
            for (Edge<E> e : v.getAdjList())
                if (!e.getDest().visited)
                    stack.push(e.getDest());
        }
    }
    resetVisited();
    System.out.println();
}
```

Lógica: - Busca el vértice de inicio. - Usa una pila para ir "profundizando" en cada rama del grafo. - Marca cada vértice visitado para no repetirlo. - Cuando un vértice tiene vecinos sin visitar, los agrega a la pila y sigue con ellos. - Al terminar, limpia las marcas de visitado.

¿Qué resuelve? Permite explorar todas las zonas alcanzables desde una zona inicial, útil por ejemplo para encontrar si hay caminos alternos o ciclos.

—

BFS (búsqueda en anchura):

```
public void bfs(E start) {
    Vertex<E> s = find(start);
    if (s == null)
        return;

    Deque<Vertex<E>> queue = new ArrayDeque<>();
    queue.addLast(s);
    s.visited = true;

    while (!queue.isEmpty()) {
        Vertex<E> v = queue.pollFirst();
        System.out.print(v.getData() + " ");
        for (Edge<E> e : v.getAdjList())
            if (!e.getDest().visited) {
                e.getDest().visited = true;
                queue.addLast(e.getDest());
            }
    }
    resetVisited();
    System.out.println();
}
```

Lógica: - Utiliza una cola para explorar primero los vecinos inmediatos. - Marca cada nodo al visitarlo. - Permite encontrar el camino más corto en cantidad de pasos (si los pesos son iguales). - Ideal para recorridos por niveles o capas en el almacén.

—

6. Camino más corto: Algoritmo de Dijkstra

```
public List<E> shortestPath(E from, E to) {
    Vertex<E> s = find(from), t = find(to);
    if (s == null || t == null)
        return Collections.emptyList();

    Map<Vertex<E>, Integer> dist = new HashMap<>();
    Map<Vertex<E>, Vertex<E>> prev = new HashMap<>()
    ;
    PriorityQueue<Vertex<E>> pq = new PriorityQueue
    <>(Comparator.comparingInt(dist::get));

    for (Vertex<E> v : vertices)
        dist.put(v, Integer.MAX_VALUE);
    dist.put(s, 0);
    pq.add(s);

    while (!pq.isEmpty()) {
        Vertex<E> v = pq.poll();
        for (Edge<E> e : v.getAdjList()) {
            int w = (e.getWeight() == null ? 1 : e.
    getWeight());
            int nd = dist.get(v) + w;
            if (nd < dist.get(e.getDest())) {
                dist.put(e.getDest(), nd);
                prev.put(e.getDest(), v);
                pq.remove(e.getDest());
                pq.add(e.getDest());
            }
        }
    }

    LinkedList<E> path = new LinkedList<>();
    for (Vertex<E> x = t; x != null; x = prev.get(x)
    )
        path.addFirst(x.getData());
    if (path.isEmpty() || !path.getFirst().equals(
    from))
        return Collections.emptyList();
    return path;
}
```

Lógica paso a paso: 1. Inicializa todas las distancias en infinito, menos la de partida (`s`), que se pone a 0. 2. Usa una cola de prioridad para escoger siempre el nodo más cercano aún no procesado. 3. Para cada vecino de un nodo, calcula si ir por esa ruta es más corto que la mejor ruta conocida hasta ese momento: - Si lo es, actualiza la distancia y el "predecesor" de ese nodo. - Añade el nodo a la cola para continuar el proceso. 4. Cuando termina, reconstruye el camino más corto desde el destino hasta el origen, usando el mapa de predecesores.

¿Para qué sirve? Permite, por ejemplo, encontrar la ruta más eficiente para recoger un producto en el almacén, con-

siderando distancias o tiempos.

—

7. Verificar conectividad

```
public boolean isConnected() {
    if (vertices.isEmpty())
        return true;
    Vertex<E> first = vertices.getFirst();
    bfs(first.getData());
    for (Vertex<E> v : vertices)
        if (!v.visited) {
            resetVisited();
            return false;
        }
    resetVisited();
    return true;
}
```

Lógica: - Si no hay nodos, se considera conexo. - Hace un recorrido BFS desde el primer nodo. - Si al terminar todos los nodos fueron visitados, el grafo es conexo (todo está comunicado). - Si hay algún nodo no visitado, es que hay zonas aisladas.

—

8. Métodos auxiliares

- `getVertices()`: retorna la lista completa de vértices, útil para otras funciones o para visualización. - `resetVisited()`: limpia las marcas de visitado, necesario para que los recorridos posteriores funcionen correctamente. - `toString()`: devuelve una representación textual del grafo (para depuración).

—

Resumen: Esta clase implementa toda la lógica para modelar, recorrer, analizar y modificar la estructura de un almacén o cualquier sistema basado en grafos. Cada método sigue una lógica clara basada en algoritmos clásicos de teoría de grafos, pero adaptada para ser flexible y eficiente en la práctica.

B. Clase `BPlusTree<K, V>`: Estructura, Lógica y Métodos

El árbol B+ es una estructura avanzada de datos utilizada para almacenar grandes volúmenes de información ordenada, permitiendo búsquedas, inserciones y eliminaciones eficientes. Es muy usada en bases de datos e índices de archivos por su alta eficiencia y su diseño que optimiza el acceso a disco. En este proyecto, se emplea para clasificar productos dentro de cada zona del almacén.

Estructura general:

- Cada **nodo interno** almacena sólo claves y enlaces a sus hijos.
- Los **nodos hoja** almacenan claves y valores (productos), y están conectados entre sí mediante un puntero `next` para recorridos rápidos.
- Todos los valores están en las hojas; los nodos internos sólo sirven para dirigir la búsqueda.

```
public class BPlusTree<K extends Comparable<K>, V> {
    public static class Node<K, V> {
        public boolean isLeaf;
        public List<K> keys;
        public List<Node<K, V>> children; // nodos
    internos
        public List<V> values;            // hojas
        public Node<K, V> next;           // hojas
        ...
    }

    private Node<K, V> root;
    private int order; // M ximo de hijos por nodo
    ...
}
```

- `order` define el máximo de hijos permitidos por nodo. - `root` es la raíz del árbol; puede ser hoja o nodo interno. - Cada `Node` distingue si es hoja o interno; sólo los nodos hoja almacenan productos reales.

Inserción de elementos:: La inserción es la operación clave que mantiene el árbol equilibrado. Se sigue este proceso:

1. Si la raíz está llena, se divide:

```
public void insert(K key, V value) {
    Node<K, V> r = root;
    if (r.keys.size() == order - 1) {
        Node<K, V> s = new Node<>(false);
        s.children.add(r);
        root = s;
        splitChild(s, 0);
        insertNonFull(s, key, value);
    } else {
        insertNonFull(r, key, value);
    }
}
```

- Si la raíz tiene el número máximo de claves, se crea una nueva raíz, se divide la vieja raíz y la inserción continúa en el nuevo nivel. - Así el árbol puede crecer en altura, pero siempre se mantiene equilibrado.

2. Inserción en un nodo no lleno:

```
private void insertNonFull(Node<K, V> node, K key, V
     value) {
    if (node.isLeaf) {
        int idx = Collections.binarySearch(node.keys
    , key);
        if (idx >= 0) {
            node.values.set(idx, value);
        } else {
            int insertIdx = -idx - 1;
            node.keys.add(insertIdx, key);
            node.values.add(insertIdx, value);
        }
    } else {
        int idx = Collections.binarySearch(node.keys
    , key);
        int childIdx = idx >= 0 ? idx + 1 : -idx -
    1;
        Node<K, V> child = node.children.get(
    childIdx);
        if (child.keys.size() == order - 1) {
            splitChild(node, childIdx);
            if (key.compareTo(node.keys.get(childIdx
    )) > 0) {
                childIdx++;
            }
        }
        insertNonFull(node.children.get(childIdx),
    key, value);
    }
}
```

Lógica: - Si el nodo es hoja: - Busca la posición ordenada con `binarySearch`. - Si existe la clave, actualiza el valor. - Si no existe, inserta la clave y el valor en la posición correcta para mantener el orden. - Si es nodo interno: - Busca el hijo adecuado para la clave. - Si ese hijo está lleno, lo divide antes de continuar (esto puede subir claves al nodo actual). - Luego repite el proceso de inserción en el hijo correcto.

3. División de nodos:

```
private void splitChild(Node<K, V> parent, int idx)
    {
    Node<K, V> node = parent.children.get(idx);
    int mid = (order - 1) / 2;
    Node<K, V> sibling = new Node<>(node.isLeaf);

    if (node.isLeaf) {
        sibling.keys.addAll(node.keys.subList(mid,
    node.keys.size()));
        sibling.values.addAll(node.values.subList(
    mid, node.values.size()));
        node.keys.subList(mid, node.keys.size()).
    clear();
        node.values.subList(mid, node.values.size())
    .clear();

        sibling.next = node.next;
        node.next = sibling;

        parent.keys.add(idx, sibling.keys.get(0));
        parent.children.add(idx + 1, sibling);
    } else {
        sibling.keys.addAll(node.keys.subList(mid +
    1, node.keys.size()));
        sibling.children.addAll(node.children.
    subList(mid + 1, node.children.size()));
        K upKey = node.keys.get(mid);

        node.keys.subList(mid, node.keys.size()).
    clear();
        node.children.subList(mid + 1, node.children
    .size()).clear();

        parent.keys.add(idx, upKey);
        parent.children.add(idx + 1, sibling);
    }
}
```

Lógica: - Encuentra el punto medio (`mid`) para dividir. - Para hojas: - El nuevo nodo (`sibling`) se lleva la segunda mitad de claves/valores. - El puntero `next` de la hoja original apunta ahora al nuevo nodo hoja. - El padre inserta la primera clave del nuevo nodo en su lista de claves y referencia al nuevo hijo. - Para nodos internos: - La clave del medio "sube" al padre (es la que separa ambos hijos). - Las claves e hijos después de la clave media van al nuevo nodo. - Así se mantiene la propiedad de que todos los nodos (excepto la raíz) tienen al menos un número mínimo de claves.

Búsqueda de un elemento: Buscar es eficiente, ya que solo sigue el camino correcto en cada nivel:

```
public V search(K key) {
    Node<K, V> node = root;
    while (!node.isLeaf) {
        int idx = Collections.binarySearch(node.keys
    , key);
        int childIdx = idx >= 0 ? idx + 1 : -idx -
    1;
        node = node.children.get(childIdx);
    }
    int idx = Collections.binarySearch(node.keys,
    key);
    if (idx >= 0)
        return node.values.get(idx);
    return null;
}
```

- Empieza en la raíz y desciende el árbol, eligiendo el hijo correcto en cada nivel. - Cuando llega a una hoja, busca la clave. - Si la encuentra, retorna el valor; si no, retorna `null`.

Eliminación de un elemento: Eliminar requiere asegurar que todos los nodos mantengan el mínimo de claves permitido:

```
public void delete(K key) {
    delete(root, key);
    if (!root.isLeaf && root.keys.size() == 0) {
        root = root.children.get(0);
    }
}
```

- Llama a la función recursiva de borrado. - Si la raíz queda vacía y tiene hijos, el árbol desciende un nivel (esto mantiene compacto el árbol).

El borrado real se gestiona así:

```
private boolean delete(Node<K, V> node, K key) {
    if (node.isLeaf) {
        int idx = Collections.binarySearch(node.keys
    , key);
        if (idx >= 0) {
            node.keys.remove(idx);
            node.values.remove(idx);
            return true;
        }
        return false;
    } else {
        int idx = Collections.binarySearch(node.keys
    , key);
        int childIdx = idx >= 0 ? idx + 1 : -idx -
    1;
        Node<K, V> child = node.children.get(
    childIdx);
        boolean deleted = delete(child, key);

        if (child.keys.size() < (order - 1) / 2) {
            rebalance(node, childIdx);
        }
        return deleted;
    }
}
```

- Si es hoja: elimina la clave/valor si lo encuentra. - Si es interno: localiza el hijo donde debería estar la clave, llama recursivamente, y luego verifica si ese hijo tiene pocas claves (menos del mínimo). - Si hay "deficiencia" de claves, llama a `rebalance` para restaurar el equilibrio.

Rebalanceo: - Intenta primero "prestar" una clave de un hermano (izquierda o derecha). - Si no es posible, fusiona el nodo con un hermano adyacente y actualiza el padre.

```
private void rebalance(Node<K, V> parent, int idx) {
    Node<K, V> node = parent.children.get(idx);
    int minKeys = (order - 1) / 2;

    // Borrow from left sibling
    if (idx > 0) { ... }
    // Borrow from right sibling
    if (idx < parent.children.size() - 1) { ... }
    // Merge with sibling
    if (idx > 0) {
        merge(parent, idx - 1);
```

```
    } else {
        merge(parent, idx);
    }
}
```

¿Por qué es eficiente el B+? - Todas las operaciones de búsqueda, inserción y borrado requieren como máximo recorrer la altura del árbol (logarítmica en el número de elementos). - Los nodos hoja están enlazados, lo que permite recorrer rangos de claves de forma secuencial (muy útil en consultas de bases de datos).

Visualización y depuración: `printTree()`, `printLeaves()` y `printTreeStructure()` permiten visualizar el árbol de diferentes formas para depuración y validación, imprimiendo tanto el árbol completo como sólo las hojas (productos almacenados).

Ejemplo de uso: Suponga que quiere almacenar productos en el almacén según su código:

```
BPlusTree<Integer, String> tree = new BPlusTree<>(4)
    ; // orden 4
tree.insert(10, "A");
tree.insert(20, "B");
tree.insert(5, "C");
tree.insert(6, "D");
// ...
tree.printTreeStructure();
System.out.println("Buscar 6: " + tree.search(6));
tree.delete(6);
tree.printTreeStructure();
```

Así puede insertar, buscar y eliminar productos eficientemente, manteniendo siempre el árbol equilibrado.

Resumen: El árbol B+ garantiza que todas las operaciones principales (inserción, búsqueda, eliminación) se realizan en tiempo logarítmico respecto al número de elementos, y que la información está siempre ordenada y accesible tanto individual como secuencialmente.

C. Clase `BPlusTreeVisualizer`: Visualización interactiva de Árboles B+ con JGraphX

El objetivo de la clase `BPlusTreeVisualizer` es brindar una forma gráfica, sencilla e interactiva de visualizar la estructura y el contenido de un árbol B+, facilitando el análisis y la depuración del sistema de inventario. Para lograrlo, utiliza la biblioteca **JGraphX**, una herramienta para construir y mostrar grafos y diagramas en aplicaciones Java.

¿Por qué es útil visualizar el árbol?

- Permite observar la organización jerárquica de los productos en una zona.
- Facilita la comprensión del orden y balanceo del árbol B+.
- Hace intuitivo el proceso de depuración: se pueden ver en qué nodo está cada producto y detectar rápidamente errores o estructuras inesperadas.

Estructuras y bibliotecas usadas:

- **JGraphX**: Proporciona componentes gráficos para construir y mostrar grafos como diagramas.
- **JFrame y JOptionPane** (Swing): Permiten crear ventanas e interacciones gráficas simples (como mostrar mensajes al usuario).
- `Map<Object, BPlusTree.Node<K,V>> cellToNodeMap`: Relaciona cada celda gráfica con el nodo real del árbol, para permitir interacción con el usuario (por ejemplo, mostrar los productos al hacer clic).

Método principal: `visualize`

```
public static <K extends Comparable<K>, V> void
    visualize(BPlusTree<K, V> tree, String zona)
```

Este método crea la ventana gráfica para visualizar el árbol B+ de una zona específica. A continuación se detalla su funcionamiento paso a paso:

1) **Inicialización de la estructura gráfica**

```
        mxGraph graph = new mxGraph();
        Object parent = graph.getDefaultParent
    ();
        Map<Object, Object> nodeMap = new
    HashMap<>();
        Map<Object, BPlusTree.Node<K, V>>
    cellToNodeMap = new HashMap<>();
        int[] y = { 40 };
```

- `mxGraph`: Representa el grafo a dibujar (en este caso, el árbol B+).
- `nodeMap`: Relaciona nodos del árbol con celdas gráficas (mxCells), útil para dibujar conexiones.
- `cellToNodeMap`: Permite saber, al hacer clic en la interfaz, qué nodo del árbol B+ representa cada celda.
- `y`: Usado para ubicar verticalmente cada nivel del árbol (nivel raíz, hijos, nietos...).

2) **Construcción recursiva del gráfico**

```
        graph.getModel().beginUpdate();
        try {
            drawNode(graph, parent, tree, tree.
    getRoot(), 400, y, 0, nodeMap,
    cellToNodeMap);
        } finally {
            graph.getModel().endUpdate();
        }
```

- `beginUpdate/endUpdate`: Indica a JGraphX que se va a modificar el modelo (agregar nodos/aristas) de forma atómica, para que la interfaz no se refresque parcialmente.
- Se llama al método recursivo `drawNode`, que dibuja cada nodo (y sus hijos) de forma organizada.

3) **Creación de la ventana gráfica**

```
        JFrame frame = new JFrame("Productos en
      " + zona + " ( rbol  B+)");
        mxGraphComponent graphComponent = new
    mxGraphComponent(graph);
        frame.getContentPane().add(
    graphComponent);
        frame.setSize(1000, 500);
        frame.setDefaultCloseOperation(JFrame.
    DISPOSE_ON_CLOSE);
```

```
        frame.setVisible(true);
```

- Crea una ventana (`JFrame`) titulada con la zona correspondiente.
- `mxGraphComponent`: Componente visual de JGraphX que contiene el diagrama dibujado.
- Se añade el componente a la ventana y se muestra al usuario.

4) **Interacción con el usuario (clicks)**

```
        graphComponent.getGraphControl().
    addMouseListener(new java.awt.event.
    MouseAdapter() {
            @Override
            public void mouseClicked(java.awt.
    event.MouseEvent e) {
                Object cell = graphComponent.
    getCellAt(e.getX(), e.getY());
                if (cell != null &&
    cellToNodeMap.containsKey(cell)) {
                    BPlusTree.Node<K, V> nodo =
     cellToNodeMap.get(cell);
                    if (nodo.isLeaf) {
                        // Mostrar los
    productos almacenados en la hoja
                    } else {
                        // Mostrar informaci n
     del nodo interno
                    }
                }
            }
        });
```

- Captura los clics del usuario sobre cualquier celda del diagrama.
- Si se hace clic en una hoja, muestra los productos de esa hoja usando `JOptionPane`.
- Si se hace clic en un nodo interno, muestra las claves que actúan como separadores en ese nivel.

Método recursivo: `drawNode`

```
private static <K extends Comparable<K>, V> void
    drawNode(
    mxGraph graph, Object parent, BPlusTree<K, V>
    tree, Object nodeObj,
    int x, int[] y, int depth,
    Map<Object, Object> nodeMap, Map<Object,
    BPlusTree.Node<K, V>> cellToNodeMap)
```

Propósito: Construye el diagrama del árbol B+ de arriba hacia abajo, posicionando cada nodo y enlazando gráficamente padres e hijos. Llama a sí mismo recursivamente para cada hijo, así se dibuja toda la estructura.

1) Convierte el objeto recibido a nodo del árbol:

```
        BPlusTree.Node<K, V> node = (BPlusTree.
    Node<K, V>) nodeObj;
```

2) Crea una etiqueta con las claves del nodo:

```
        String label = node.keys.toString() + (
    node.isLeaf ? " (Hoja)" : "");
```

3) Dibuja el nodo como un "vértice" del grafo:

```
        Object graphNode = graph.insertVertex(
    parent, null, label, x, y[0], 120, 30);
```

- Inserta un rectángulo en la posición (x, y), con el texto adecuado.
- Relaciona este vértice gráfico con el nodo real (`nodeMap`, `cellToNodeMap`).

4) Si el nodo tiene hijos (es interno), recorre y dibuja cada hijo:

```
        if (!node.isLeaf) {
            y[0] += 80; // baja al siguiente
    nivel
            for (Object childObj : node.
    children) {
                drawNode(...); // llamada
    recursiva para el hijo
                // Dibuja la arista gr fica (
    conexi n visual)
                Object childGraphNode = nodeMap
    .get(childObj);
                graph.insertEdge(parent, null,
    "", graphNode, childGraphNode);
                childX += 140;
            }
            y[0] -= 80; // sube al nivel
    anterior
        }
```

- Calcula la posición horizontal y vertical de cada hijo para evitar superposiciones y mantener la jerarquía visual clara.
- Añade líneas ("aristas") entre cada padre e hijo, mostrando la relación jerárquica del árbol.

¿Por qué usar mapas (`Map`) para las celdas y nodos?

- JGraphX maneja los vértices y aristas como "celdas" (objetos internos del framework, no como tus nodos reales de árbol).
- Es necesario relacionar cada celda dibujada con el nodo real del árbol B+, para poder saber, al hacer clic, qué información mostrar.
- Por eso, al crear cada celda gráfica, se guarda su referencia junto al nodo B+ correspondiente.

¿Por qué usar JGraphX y estructuras de datos especializadas?

- Dibujar árboles a mano sería lento, complicado y poco flexible.
- JGraphX permite posicionar, conectar y actualizar los nodos gráficamente con sólo indicar las relaciones entre ellos.
- Los mapas `nodeMap` y `cellToNodeMap` son esenciales para mantener sincronía entre el modelo lógico (árbol B+) y la vista gráfica.
- La recursividad permite procesar cualquier árbol, por grande que sea, sin repetir código para cada nivel.

Ventajas de esta visualización:

- El usuario puede ver en tiempo real la estructura de productos almacenados y cómo se distribuyen en las hojas del árbol.

- Permite explorar la información sólo haciendo clic, sin requerir conocimientos de estructuras de datos.
- Facilita la enseñanza y la depuración para usuarios, docentes y desarrolladores.

Ejemplo de uso práctico:

Supón que tienes una zona "A1" con varios productos y deseas ver cómo están organizados internamente:

```
BPlusTree<Integer, Producto> arbol = new BPlusTree
    <>(4);
// ... inserta productos ...
BPlusTreeVisualizer.visualize(arbol, "A1");
```

Se abrirá una ventana interactiva donde puedes ver toda la estructura y explorar los productos almacenados en cada hoja.

Resumen:

`BPlusTreeVisualizer` es la interfaz puente entre la lógica compleja de los árboles B+ y el usuario final, usando componentes gráficos modernos y técnicas de mapeo de datos para brindar una experiencia visual intuitiva y poderosa.

D. Clase `GraphVisualizer`: Visualización Gráfica de Grafos con JGraphX

La clase `GraphVisualizer` tiene como objetivo mostrar de forma gráfica, clara e interactiva la estructura de cualquier grafo creado con la clase `GraphLink<E>`. Para esto utiliza la biblioteca **JGraphX**, una herramienta avanzada para visualizar y editar diagramas de grafos en Java. Esto resulta útil tanto para usuarios como para programadores y docentes, permitiendo observar rápidamente las conexiones, pesos y disposición de las rutas en el almacén.

¿Por qué visualizar un grafo?

- Facilita comprender la topología del sistema: cómo están conectadas las diferentes zonas.
- Ayuda a detectar errores de modelado (zonas aisladas, conexiones incorrectas, ciclos inesperados, etc.).
- Permite analizar rutas óptimas y verificar visualmente los resultados de algoritmos como Dijkstra.
- Hace más intuitivo el análisis de la eficiencia y robustez del diseño.

Estructuras y bibliotecas utilizadas:

- **JGraphX** (`mxGraph`, `mxGraphComponent`): Permite crear y visualizar el grafo como un diagrama interactivo, con vértices (zonas) y aristas (rutas).
- `Map<E, Object> vertexMap`: Relaciona cada nodo del grafo lógico con el nodo gráfico dibujado, para facilitar la conexión de aristas y futuras extensiones (como interacción por clic).

Método principal: `visualizeGraph`

```
public static <E> void visualizeGraph(GraphLink<E>
    graph, boolean directed)
```

Este método crea y muestra una ventana con la representación gráfica del grafo, incluyendo nodos, aristas y los pesos si corresponden. Explicamos a detalle cada fragmento relevante:

1) **Preparación del grafo y layout inicial**

```
    mxGraph mxgraph = new mxGraph();
    Object parent = mxgraph.
getDefaultParent();

    int x = 40, y = 40, dx = 120, dy = 120,
 col = 0, row = 0;
    int maxCols = (int) Math.ceil(Math.sqrt
(graph.getVertices().size()));
```

- Crea una nueva instancia de `mxGraph` (el modelo gráfico del grafo).
- Establece la posición de cada nodo en una cuadrícula (usando variables `x`, `y`, `dx`, `dy`, etc.), para que no se superpongan y el grafo sea legible sin importar su tamaño.
- Calcula automáticamente el número máximo de columnas a usar para repartir los nodos de manera balanceada en pantalla.

2) **Agregado de nodos al modelo gráfico**

```
    mxgraph.getModel().beginUpdate();
    try {
        Map<E, Object> vertexMap = new
HashMap<>();
        for (Vertex<E> v : graph.
getVertices()) {
            if (col >= maxCols) {
                col = 0;
                row++;
            }
            Object cell = mxgraph.
insertVertex(parent, null, v.getData().
toString(),
                    x + col * dx, y + row *
 dy, 60, 40);
            vertexMap.put(v.getData(), cell
);
            col++;
        }
```

- Recorre todos los nodos lógicos del grafo.
- Para cada uno, inserta un vértice gráfico (`insertVertex`), ubicándolo en la cuadrícula según su orden.
- El texto de cada nodo será el dato almacenado (por ejemplo, "A1", "B2", etc.).
- El mapa `vertexMap` asocia el dato lógico con la celda gráfica, necesario para agregar aristas después.

3) **Agregado de aristas y pesos**

```
        // A adir las aristas
        for (Vertex<E> v : graph.
getVertices()) {
            for (Edge<E> e : v.getAdjList()
) {
                Object src = vertexMap.get(
v.getData());
                Object dst = vertexMap.get(
e.getDest().getData());
                // Para no duplicar en no
dirigidos
                if (!directed && v.getData
().toString().compareTo(e.getDest().getData
().toString()) > 0)
                    continue;
                String label = e.getWeight
() != null ? e.getWeight().toString() : "";
```

```
                    mxgraph.insertEdge(parent,
null, label, src, dst);
                }
            }
```

- Para cada nodo, recorre su lista de aristas y dibuja la conexión con sus vecinos.
- En grafos no dirigidos, evita dibujar dos veces la misma conexión (usa el nombre/texto del nodo para comparar y saltar duplicados).
- El peso de la arista se muestra como etiqueta si existe (por ejemplo, distancia o costo entre zonas).

4) **Actualización del modelo y visualización en ventana**

```
        } finally {
            mxgraph.getModel().endUpdate();
        }
        JFrame frame = new JFrame(directed ? "
Grafo Dirigido" : "Grafo NO Dirigido");
        frame.setDefaultCloseOperation(JFrame.
EXIT_ON_CLOSE);
        mxGraphComponent graphComponent = new
mxGraphComponent(mxgraph);
        frame.getContentPane().add(
graphComponent);
        frame.setSize(800, 600);
        frame.setVisible(true);
```

- Termina la actualización del modelo gráfico para que todo se dibuje a la vez (evita parpadeos).
- Crea una ventana y coloca el componente de JGraphX dentro, mostrando el grafo completo.
- El título de la ventana refleja si el grafo es dirigido o no.

¿Por qué usar mapas para las celdas y nodos?

- JGraphX maneja los nodos como objetos internos (`mxCell`); nuestro modelo lógico son objetos Java (`Vertex`, `Edge`).
- Para conectar correctamente el grafo visual y el lógico, usamos un mapa (`vertexMap`) que traduce de uno a otro.
- Esto es esencial para dibujar las aristas correctamente y, si se desea, añadir interacción (como mostrar información de un nodo al hacer clic, tarea fácil de ampliar).

Ventajas de este diseño:

- Separa claramente la lógica del modelo (el almacén, los productos, las rutas) de la lógica visual (cómo se dibuja cada elemento).
- Hace la visualización automática y escalable: puede mostrar desde grafos pequeños a grandes sin cambiar el código.
- Permite, con pocas modificaciones, añadir interacción avanzada (arrastrar nodos, mostrar información contextual, etc.).

Ejemplo de uso:

Supón que ya tienes un grafo modelando el almacén:

```
GraphLink<String> grafo = new GraphLink<>(false);
// ... agregar nodos y conexiones ...
GraphVisualizer.visualizeGraph(grafo, false);
```

Aparecerá una ventana donde puedes ver todas las zonas conectadas y las rutas, ayudando a entender cómo está organizado tu almacén.

Resumen:

`GraphVisualizer` permite "ver" el grafo detrás de tu modelo lógico, ayudando a estudiantes, desarrolladores y usuarios a comprender, depurar y optimizar la estructura del sistema, con el poder y flexibilidad de JGraphX para manejar visualizaciones complejas sin esfuerzo manual.

E. Método `visualizarGrafoConEventos`: Visualización Interactiva de Grafo y Productos por Zona

Este método implementa una de las funciones más poderosas e intuitivas del sistema: permite al usuario **visualizar el grafo completo del almacén** (zonas y rutas) e, interactivamente, hacer clic en cada zona para consultar, en una ventana aparte, los productos que contiene (visualizados como árbol B+). Es un claro ejemplo de integración entre la lógica de datos y la interfaz gráfica, usando la biblioteca **JGraphX**.

¿Para qué sirve?

- Facilita la exploración visual del almacén: el usuario puede ver todas las zonas y rutas de un vistazo.
- Permite descubrir, con un simple clic, los productos almacenados en cualquier zona, lo que sería engorroso por consola o desde menús.
- Hace mucho más fácil la depuración y presentación del sistema a usuarios o docentes.

Estructuras y parámetros clave:

- `GraphLink<String> graph`: el grafo lógico, donde cada vértice representa una zona del almacén.
- `Map<String, BPlusTree<Integer, Producto» productosPorZona`: asocia a cada zona su propio árbol B+ de productos (la clave del mapa es el nombre de la zona).
- `mxGraph` y `mxGraphComponent`: permiten crear y mostrar el grafo gráficamente con JGraphX.
- `vertexMap`: relaciona el nombre de la zona (lógico) con la celda gráfica (para conectar aristas y manejar clics).

Funcionamiento paso a paso:

1) **Creación de la estructura visual del grafo**

```
        mxGraph mxgraph = new mxGraph();
        Object parent = mxgraph.
getDefaultParent();
        ...
        mxgraph.getModel().beginUpdate();
        try {
            for (Vertex<String> v : graph.
getVertices()) {
                // Crea cada zona como v rtice
 visual (en cuadr cula)
                Object cell = mxgraph.
insertVertex(parent, null, v.getData(),
                        x + col * dx, y + row *
 dy, 80, 50);
```

```
                vertexMap.put(v.getData(), cell
);
                ...
            }
            for (Vertex<String> v : graph.
getVertices()) {
                for (Edge<String> e : v.
getAdjList()) {
                    Object src = vertexMap.get(
v.getData());
                    Object dst = vertexMap.get(
e.getDest().getData());
                    String label = e.getWeight
() != null ? e.getWeight().toString() : "";
                    mxgraph.insertEdge(parent,
null, label, src, dst);
                }
            }
        } finally {
            mxgraph.getModel().endUpdate();
        }
```

- Los vértices (zonas) se posicionan automáticamente en una cuadrícula para mayor claridad.
- Cada conexión entre zonas se dibuja como una arista, mostrando su peso si existe (por ejemplo, distancia entre pasillos).

2) **Visualización en ventana**

```
        JFrame frame = new JFrame("Mapa del
Almac n (Click en zona para ver productos)
");
        ...
        frame.setVisible(true);
```

- Se crea una ventana con el grafo completo, fácil de explorar.

3) **Interactividad: Click en una zona**

```
        graphComponent.getGraphControl().
addMouseListener(new java.awt.event.
MouseAdapter() {
            @Override
            public void mouseClicked(java.awt.
event.MouseEvent e) {
                Object cell = graphComponent.
getCellAt(e.getX(), e.getY());
                if (cell instanceof mxCell &&
((mxCell) cell).isVertex()) {
                    String zona = (String) ((
mxCell) cell).getValue();
                    BPlusTree<Integer, Producto
> tree = productosPorZona.get(zona);
                    if (tree != null) {
                        BPlusTreeVisualizer.
visualize(tree, zona);
                    } else {
                        JOptionPane.
showMessageDialog(frame, "Esta zona no
tiene productos.");
                    }
                }
            }
        });
```

- Captura cualquier clic en la ventana del grafo.
- Si el usuario hace clic sobre una zona (no sobre una arista), se identifica el nombre de la zona.
- Recupera el árbol B+ asociado desde el mapa `productosPorZona`.
- Si existe un árbol para esa zona, invoca el visualizador de árbol B+ para mostrar gráficamente los productos de esa zona.
- Si no hay productos, muestra un mensaje informativo.

¿Por qué este diseño y uso de estructuras?

- **JGraphX** es la opción ideal para visualizar grafos de cualquier tamaño y forma, con aristas etiquetadas, disposición automática y soporte para interacción avanzada (clic, arrastre, zoom, etc.).
- Usar un mapa (`vertexMap`) permite asociar cada vértice visual a su dato lógico, esencial para extender el sistema o añadir más eventos de usuario.
- El mapa `productosPorZona` conecta la estructura física (zonas) con la estructura lógica (inventario por zona) sin necesidad de modificar la clase del grafo, manteniendo modularidad y claridad.
- El sistema puede escalar a cualquier tamaño, añadir nuevas zonas, eliminar otras, y la visualización y acceso a productos seguirá funcionando sin cambios adicionales.

Ventajas:

- Proporciona una experiencia amigable: la información del almacén es accesible de manera visual e interactiva.
- Permite analizar la organización, detectar cuellos de botella y planificar mejoras logísticas visualizando zonas densas o aisladas.
- Facilita la integración y validación de los diferentes módulos del sistema (grafo, árbol B+, productos) en una única interfaz.

Ejemplo de uso:

Suponga que el sistema ya ha cargado el grafo y los productos:

```
visualizarGrafoConEventos(grafoAlmacen,
    mapaProductosPorZona);
```

El usuario verá el plano del almacén y podrá explorar su contenido haciendo clic en cualquier zona.

Resumen:

Este método combina la potencia del modelado matemático de grafos, la eficiencia de los árboles B+ y la simplicidad de la interacción gráfica moderna, permitiendo una gestión de almacenes avanzada e intuitiva incluso para usuarios sin experiencia técnica.

F. Análisis Detallado de los `import` Estándar y Externos: Funciones, Ventajas y Casos de Uso

El proyecto emplea una variedad de clases y utilidades externas que aportan ventajas específicas para la implementación y optimización de estructuras de datos, algoritmos y la interfaz gráfica. A continuación se explica, una a una, la utilidad de cada import y sus razones técnicas:

- **java.util.ArrayList:** Implementa la interfaz `List` utilizando un arreglo dinámico, lo que permite acceso rápido

e indexado a cualquier posición en tiempo constante $O(1)$. Es ideal para listas de claves o valores en nodos hoja de árboles B+, especialmente cuando el acceso aleatorio es más importante que las inserciones o eliminaciones frecuentes en medio de la lista. Su sobrecarga de memoria es baja y es óptima cuando las operaciones se concentran al final de la lista.

- **java.util.LinkedList:** Implementa tanto `List` como `Deque`. Su fortaleza está en las inserciones y eliminaciones rápidas en cualquier parte de la lista, ya que solo requiere actualizar referencias entre nodos ($O(1)$), sin la sobrecarga de mover elementos como ocurre en `ArrayList`. Es fundamental para listas de adyacencia en grafos, recorridos BFS/DFS y para representar colas o pilas que requieren inserciones y eliminaciones frecuentes al principio o en el medio de la lista. Sin embargo, el acceso aleatorio es más lento ($O(n)$), por lo que no es conveniente para acceso directo frecuente.
- **java.util.Queue:** Es una interfaz que define una colección de tipo FIFO (First-In, First-Out). Permite generalizar algoritmos de recorrido (como BFS) y otras estructuras donde el orden de procesamiento depende del orden de inserción. Al usar esta interfaz, se puede cambiar fácilmente la implementación subyacente (por ejemplo, usar `LinkedList` o `ArrayDeque`) sin modificar el resto del código, lo que facilita pruebas y refactorización.
- **java.util.Deque:** Interfaz para estructuras de doble extremo (Double-Ended Queue), lo que permite inserciones y eliminaciones tanto al principio como al final de la colección en tiempo constante. Es esencial para implementar pilas y colas con mayor flexibilidad que `Queue`. Facilita algoritmos que requieren explorar ambos extremos, como ciertos recorridos de grafos o técnicas de backtracking. Su implementación principal en el proyecto es mediante `ArrayDeque`.
- **java.util.ArrayDeque:** Implementación altamente eficiente de la interfaz `Deque` basada en un arreglo circular. Supera en eficiencia a `LinkedList` cuando se usa como pila o cola, ya que no hay sobrecarga de nodos enlazados ni sincronización innecesaria (como ocurre en `Stack` o `Vector`). Todos los métodos principales (`addFirst`, `removeFirst`, `addLast`, `removeLast`) son $O(1)$. Su ventaja principal es la velocidad y la baja sobrecarga de memoria para operaciones de doble extremo.
- **java.util.Map y java.util.HashMap:** `Map` es la interfaz general para mapas clave-valor. `HashMap` es su implementación más eficiente para la mayoría de los casos: las operaciones de inserción, búsqueda y eliminación son $O(1)$ en promedio gracias a la función de dispersión (hashing). Se utiliza extensamente para asociar zonas a árboles de productos, indexar vértices por posición, construir matrices de adyacencia, y almacenar rutas óptimas en algoritmos como Dijkstra.
- **java.util.Set y java.util.HashSet:** `Set` define una colección sin elementos duplicados. `HashSet` es su implementación más veloz para operaciones de inserción, búsqueda y eliminación ($O(1)$ promedio), útil para almacenar aristas únicas en grafos no dirigidos o para evitar nodos repetidos durante análisis estructurales, como comprobación de ciclos o componentes conexas.
- **java.util.Objects:** Clase utilitaria diseñada para mejorar la robustez del código ante valores `null`. Su método `Objects.equals(a, b)` compara de forma segura dos objetos, devolviendo `true` si ambos son `null`, evitando así excepciones de tipo `NullPointerException`. Es fundamental para comparar datos de vértices, aristas y elementos en estructuras genéricas donde los valores pueden ser nulos.
- **java.util.Collections:** Provee utilidades para operaciones de alto nivel sobre colecciones, como ordenamientos y búsquedas eficientes. `Collections.binarySearch` permite buscar en listas ordenadas ($O(\log n)$), lo que es vital en árboles B+ para mantener la eficiencia en inserciones y búsquedas.
- **java.util.Comparator:** Permite definir reglas de ordenación personalizadas para objetos complejos. Es esencial en la implementación de colas de prioridad (`PriorityQueue`) y al ordenar nodos o aristas por criterios distintos al orden natural, como peso, clave o cualquier atributo.
- **java.util.PriorityQueue:** Implementa una cola de prioridad eficiente mediante un heap binario, asegurando que la extracción del elemento de mayor prioridad sea siempre $O(\log n)$. Es indispensable en el algoritmo de Dijkstra para procesar vértices en orden de distancia mínima, mejorando drásticamente la eficiencia en grafos grandes y ponderados.
- **java.util.Iterator:** Facilita el recorrido y modificación segura de colecciones. Es especialmente útil en métodos como `removeIf` y al eliminar elementos durante la iteración, evitando errores de concurrencia y simplificando la escritura de bucles genéricos sobre listas, conjuntos y mapas.
- **java.util.Scanner:** Proporciona una interfaz simple y flexible para leer datos desde la entrada estándar o archivos, facilitando la creación de menús interactivos y la recolección de datos del usuario en la consola.
- **java.util.function.Predicate:** Interfaz funcional que representa una condición booleana sobre un objeto. Permite escribir expresiones lambda para filtrar o eliminar elementos en colecciones de forma concisa, aumentando la claridad y reduciendo la necesidad de bucles manuales. Ejemplo: `list.removeIf(x -> x > 5 )`.
- **java.lang.StringBuilder:** Proporciona una manera eficiente de concatenar cadenas de texto, mucho más rápida que usar el operador + repetidamente, ya que minimiza la creación de objetos intermedios. Es fundamental al construir salidas extensas como impresiones de estructuras, recorridos o reportes.
- **java.lang.Math:** Ofrece funciones matemáticas comunes (`ceil`, `sqrt`, `min`, `max`, etc.) usadas en cálculos de posicionamiento gráfico, divisiones de nodos, cálculos

de distancias, etc. Es preferible a escribir funciones matemáticas manualmente y garantiza precisión y velocidad.

- **java.lang.Override:** Anotación que ayuda al compilador a detectar errores en la sobreescritura de métodos. Garantiza que el método efectivamente sobrescribe uno de la superclase o interfaz, evitando errores sutiles por firmas incorrectas.
- **javax.swing.JFrame, JOptionPane:** `JFrame` permite la creación de ventanas gráficas en aplicaciones de escritorio. `JOptionPane` facilita la creación de cuadros de diálogo simples para mensajes, advertencias o entrada de datos, acelerando el desarrollo de interfaces de usuario amigables y visualmente intuitivas.
- **java.awt.event.MouseAdapter, MouseEvent:** Permiten detectar e interpretar acciones del usuario con el mouse, como clics o movimientos, lo que es esencial para añadir interactividad (por ejemplo, mostrar información detallada de un nodo al hacer clic sobre él en el grafo o el árbol).
- **com.mxgraph.view.mxGraph, com.mxgraph.swing.mxGraphComponent, com.mxgraph.model.mxCell:** Son las piezas centrales de la biblioteca JGraphX, utilizada para visualizar y manipular grafos y árboles B+ en la interfaz gráfica. `mxGraph` gestiona la estructura visual, `mxGraphComponent` la inserta en el entorno Swing, y `mxCell` modela visualmente cada nodo y arista, permitiendo personalización y detección de eventos.

En resumen: Cada import ha sido seleccionado por su aporte exclusivo a la eficiencia, expresividad, robustez y facilidad de desarrollo. La elección entre, por ejemplo, `ArrayList` y `LinkedList`, o entre `LinkedList` y `ArrayDeque`, se basa en las necesidades de rendimiento específicas de cada estructura y operación. Igualmente, el uso de utilidades como `Objects.equals` garantiza seguridad ante valores nulos, y el empleo de clases de interfaz gráfica y de manipulación de eventos habilita una experiencia de usuario avanzada, moderna y dinámica.

V. Pruebas y Resultados

A. Objetivo de las Pruebas

El objetivo principal de las pruebas es validar que el sistema cumple con todos los requisitos funcionales, garantiza el correcto funcionamiento de cada módulo (grafo, árbol B+, integración visual), y ofrece una experiencia robusta y eficiente tanto en la gestión como en la visualización de información. Se busca demostrar que el sistema es capaz de modelar, modificar y explorar el almacén en escenarios reales, incluyendo operaciones sobre rutas, zonas y productos.

B. Estrategia de Pruebas

Para garantizar la calidad del sistema, se implementaron diferentes tipos de pruebas:

- **Pruebas unitarias**: Se evaluaron los métodos principales de cada clase (grafo, árbol B+, visualización) con diferentes entradas, incluyendo casos límite y operaciones encadenadas.
- **Pruebas de integración**: Se verificó el funcionamiento conjunto de todos los módulos, asegurando la correcta interacción entre el grafo de zonas, los árboles B+ de productos y la interfaz gráfica.
- **Pruebas funcionales**: Se diseñaron casos de uso para cubrir todos los requisitos, simulando situaciones reales de gestión de almacenes, como la inserción y búsqueda de productos, modificación de rutas, visualización y simulación de escenarios.

C. Casos de Prueba Representativos

D. Evidencia Visual

A continuación, se muestran capturas de pantalla de las principales interfaces y visualizaciones del sistema, demostrando la funcionalidad y facilidad de uso.

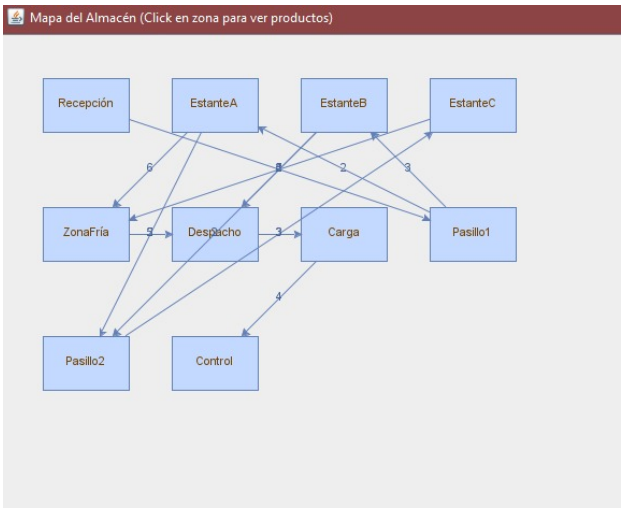

Fig. 1. Visualización interactiva del grafo que modela las zonas y rutas del almacén.

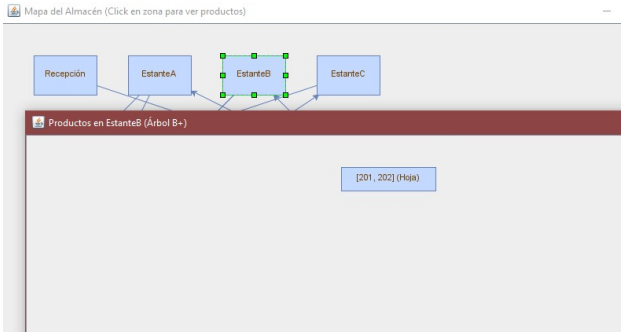

Fig. 2. Árbol B+ de productos correspondiente a una zona seleccionada.

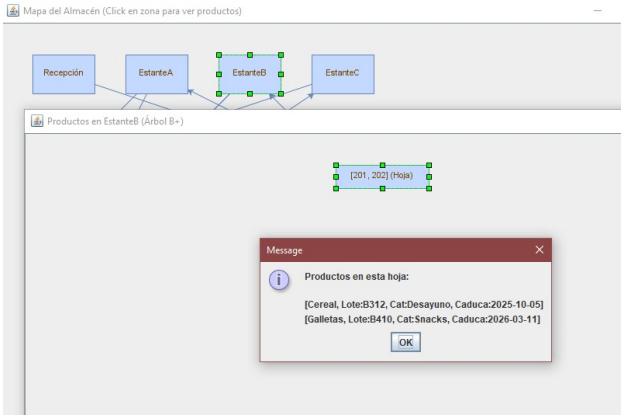

Fig. 3. Mensaje emergente con los productos listados al hacer clic en una hoja del árbol B+.

E. Análisis y Cumplimiento de Requisitos

A continuación, se presenta una evaluación respecto a los principales requisitos funcionales y técnicos establecidos en el proyecto:

- **Modelado de ubicaciones y rutas:** El sistema permite representar las diferentes zonas del almacén y las rutas entre ellas mediante un grafo, facilitando la visualización y manipulación de la estructura física.
- **Árboles B+ por zona:** Cada zona cuenta con un árbol B+ propio que clasifica y organiza eficientemente los productos, permitiendo búsquedas y modificaciones rápidas.
- **Algoritmos de grafos:** Se implementan y prueban algoritmos clásicos (Dijkstra, BFS, DFS, detección de ciclos y zonas aisladas), asegurando la correcta optimización de rutas y análisis estructural.
- **Visualización gráfica e interacción:** La interfaz gráfica permite visualizar tanto el grafo general como el árbol B+ de cada zona, y consultar productos mediante eventos interactivos.
- **Simulación de escenarios:** El sistema soporta la adición y eliminación dinámica de zonas, rutas y productos, permitiendo evaluar distintos escenarios logísticos y su impacto en la conectividad y accesibilidad del almacén.

F. Observaciones Adicionales

Durante las pruebas, se identificó que si un nodo queda aislado tras eliminar su única conexión, este permanecerá desconectado del resto del grafo. Esta decisión de diseño es coherente con el objetivo de modelar el almacén de manera realista: *Un nodo aislado representa una zona físicamente inaccesible, por lo que no tiene sentido forzar una "reconexión" automática a otro nodo cercano. Si un nodo queda aislado, se considera fuera de uso hasta que se restablezca alguna conexión manualmente.* De este modo, el sistema evita crear caminos artificiales que podrían introducir inconsistencias o errores en la gestión del almacén.

VI. Conclusiones

El desarrollo de este proyecto permitió integrar conceptos avanzados de estructuras de datos, algoritmos y diseño orientado a objetos, aplicados a la simulación y gestión de un almacén moderno. A continuación, se resumen las principales conclusiones obtenidas:

- **Modelado eficiente y realista:** Utilizar un grafo dirigido y ponderado para representar las zonas y rutas del almacén, junto con árboles B+ para la organización interna de productos, demostró ser una solución flexible, escalable y alineada con la realidad de sistemas logísticos.
- **Rendimiento y modularidad:** La implementación de algoritmos eficientes (como Dijkstra para rutas óptimas y B+ para búsquedas rápidas) permitió alcanzar tiempos de respuesta adecuados incluso ante escenarios de alta complejidad o volumen de datos, validando la pertinencia de las estructuras de datos seleccionadas.
- **Visualización e interacción:** La integración de herramientas de visualización gráfica (JGraphX) facilitó la exploración, depuración y presentación del sistema, haciendo la experiencia accesible incluso a usuarios sin conocimientos técnicos avanzados. La posibilidad de interactuar con las zonas y consultar productos de manera visual es una ventaja destacada frente a soluciones tradicionales basadas solo en texto.
- **Robustez y adaptabilidad:** El sistema demostró ser capaz de gestionar dinámicamente la adición y eliminación de zonas, rutas y productos, adaptándose a cambios en el entorno del almacén y soportando escenarios diversos (crecimiento, cierres, reubicaciones).
- **Valor académico y formativo:** El trabajo permitió afianzar y aplicar competencias clave en programación, diseño de algoritmos, análisis de requerimientos y presentación de resultados, preparando al equipo para desafíos más avanzados en desarrollo de software y modelado de sistemas complejos.

En suma, el sistema desarrollado cumple con todos los objetivos y requisitos propuestos, aportando una solución robusta y educativa al problema planteado, y abre la puerta a posibles extensiones o mejoras futuras orientadas a sistemas reales de gestión logística o educativa.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[2] D. Comer, "The Ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

[3] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*, 3rd ed. Cambridge University Press, 2020.

[4] G. Chartrand and P. Zhang, *A First Course in Graph Theory*. Dover Publications, 2012.

[5] JGraph Ltd., "JGraphX: Java Graph Visualization Library," [Online]. Available: https://github.com/jgraph/jgraphx [Accessed: 20-Jun-2025].

[6] Oracle, "The Java Tutorials," [Online]. Available: https://docs.oracle.com/javase/tutorial/ [Accessed: 20-Jun-2025].

[7] M. A. Weiss, *Data Structures and Algorithm Analysis in Java*, 3rd ed. Pearson, 2012.

[8] O. Flores, "Clase Java: Árbol B y B*," Material interno, Universidad Católica de Santa María, 2025.

[9] O. Flores, “Clase Java: Árbol B+,” Material interno, Universidad Católica de Santa María, 2025.
[10] O. Flores, “Clase Java: Grafos y Recorridos,” Material interno, Universidad Católica de Santa María, 2025.
[11] O. Flores, “Clase Java: Algoritmos de Recorridos (BFS, DFS),” Material interno, Universidad Católica de Santa María, 2025.