INFORME NEXPAY: Estructuras de datos

Laura Milena Cárdenas Saavedra
Bryan Bernardo Cárdenas Saavedra
Cesar Camilo Túlcan Erira

Resumen: Este proyecto presenta el desarrollo de Nexpay, un simulador de monedero virtual enfocado en la implementación de estructuras de datos propias en Java. El sistema permite gestionar múltiples monederos, cuentas, transacciones reversibles y programadas, además de ofrecer análisis financiero y clasificación por puntos. Se utilizaron listas enlazadas, pilas, colas de prioridad, árboles binarios y grafos dirigidos desarrollados manualmente, sin utilizar colecciones estándar. El enfoque principal fue demostrar la aplicabilidad de estas estructuras en un entorno funcional, eficiente y escalable.

1. INTRODUCCIÓN:

La gestión financiera digital ha impulsado la creación de plataformas cada vez más inteligentes y personalizadas. En este contexto, el proyecto Nexpay simula una billetera virtual con soporte para múltiples monederos y funcionalidades avanzadas, haciendo énfasis en la aplicación de estructuras de datos eficientes construidas manualmente.

A diferencia de soluciones tradicionales que se apoyan en librerías estándar, Nexpay fue desarrollado con el objetivo académico de implementar desde cero estructuras de datos fundamentales, y demostrar cómo estas pueden aplicarse a un sistema financiero realista. Cada módulo del sistema—desde la administración de cuentas y transacciones, hasta la programación de operaciones y la generación de alertas—está respaldado por una estructura diseñada y optimizada a mano.

Este informe detalla la arquitectura del sistema, las decisiones de diseño, y cómo cada estructura de datos fue seleccionada, desarrollada e integrada para resolver una necesidad específica del sistema. Así, Nexpay no solo representa una herramienta funcional, sino también un ejercicio riguroso de diseño algorítmico orientado a la práctica profesional de la ingeniería de software.

2. OBJETIVOS:

- Implementar estructuras de datos personalizadas como listas enlazadas, pilas, colas de prioridad y árboles binarios para el manejo de cuentas, transacciones y usuarios.
- Diseñar un sistema de transacciones reversibles y programadas que aproveche estructuras como pilas y colas de prioridad propias.
- Integrar un sistema de multi monederos utilizando grafos dirigidos para representar las relaciones internas y calcular el saldo global del usuario.
- Validar el correcto funcionamiento de las estructuras de datos mediante su aplicación en funcionalidades reales como el historial, clasificación por puntos y análisis de transacciones.

3. DESARROLLO

El desarrollo del sistema Nexpay se llevó a cabo utilizando Java con el framework Spring Boot para la gestión del backend, y Thymeleaf como motor de plantillas para la generación dinámica de las vistas HTML. El sistema se enfocó en modelar una billetera virtual con gestión de cuentas, transacciones (incluyendo programadas y reversibles), puntos, y análisis por monederos — haciendo especial énfasis en la implementación de estructuras de datos propias.

3.1. Proyecto.nexpay.web.controller

Contiene todos los controladores del sistema. Manejan las rutas HTTP, procesan formularios y coordinan la lógica entre los modelos y las vistas. Algunos ejemplos:

- AuthController gestiona el inicio de sesión y registro.
- AdminDashboardController permite al administrador gestionar usuarios, cuentas, monederos y transacciones.
- UserDashboardController permite a los usuarios visualizar y administrar su actividad financiera.

3.2. Proyecto.nexpay.web.datastructures

Este paquete implementa todas las estructuras de datos personalizadas, creadas desde cero para reemplazar las colecciones estándar de Java. Incluye:

- SimpleList, DoubleLinkedList, Stack, PriorityQueue.
- WalletGraph para la relación entre monederos y sus nodos.

Estas estructuras soportan operaciones fundamentales como inserción, eliminación, recorrido y búsqueda.

3.3. Proyecto.nexpay.web.model

Incluye todas las clases de dominio que representan las entidades principales del sistema:

- User, Administrator, Account, Transaction, Wallet, entre otras.
- Cada entidad encapsula sus atributos y funcionalidades específicas.

3.4. Proyecto.nexpay.web.persistence

Encargado de la persistencia de los datos usando archivos de texto planos. Se almacenan en la carpeta resources, y cada entidad tiene su clase dedicada, como:

- UserPersistence, WalletPersistence, AccountPersistence, TransactionPersistence.

3.5. Proyecto.nexpay.web.service

Aquí reside la lógica de negocio principal del sistema:

- TransactionManager: ejecuta y revierte transacciones.
- ScheduleTransactionManager: administra transacciones programadas mediante colas de prioridad.
- NotificationManager: gestiona alertas automáticas.

3.6. Proyecto.nexpay.web.fileUtil

Contiene utilidades de lectura y escritura en archivos. Soporta la persistencia general del sistema de forma modular y segura.

3.7. Proyecto.nexpay.web.Exception

Agrupa clases de excepción personalizadas para un manejo más claro y controlado de errores en el sistema.

3.8. Resources

Contiene los archivos de persistencia de los datos (por ejemplo, users.txt, transactions.txt, etc.) y las vistas HTML desarrolladas con Thymeleaf. Estas vistas

están enlazadas dinámicamente con los datos enviados desde los controladores.

3.9. NexpayWebApplication.java

Es la clase principal de arranque del sistema, donde se inicializa Spring Boot y se cargan las instancias de Nexpay.

Esta estructura modular favorece la mantenibilidad, escalabilidad y reutilización de componentes. La separación entre lógica de negocio, persistencia, controladores y vistas garantiza un desarrollo ordenado y una clara asignación de responsabilidades.

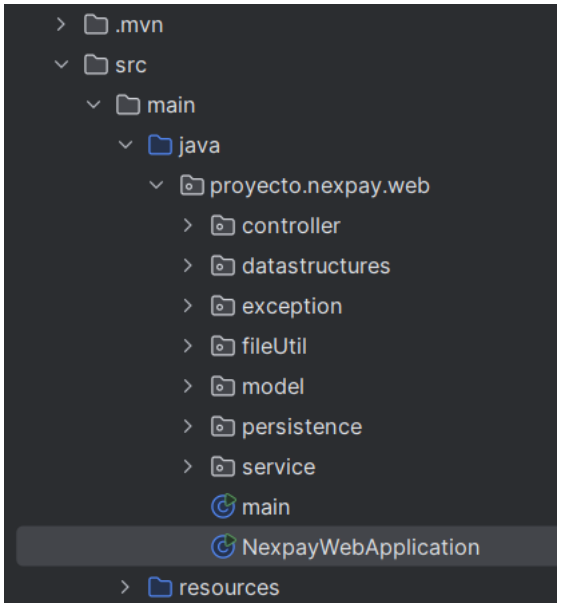


Imagen 1. Estructura del proyecto Nexpay.
Fuente: propia.

4. IMPLEMENTACIÓN DE ESTRUCTURAS DE DATOS PROPIAS

El desarrollo del sistema Nexpay tuvo como eje central la utilización de estructuras de datos implementadas manualmente, sin recurrir a las colecciones de la biblioteca estándar de Java. Esto no solo responde a una exigencia técnica del proyecto, sino que también permitió demostrar el dominio de conceptos fundamentales en estructuras de datos y su aplicación en un sistema funcional completo.

Para ello, se implementaron desde cero varias estructuras, ubicadas en el paquete proyecto.nexpay.web.datastructures. Entre ellas se encuentran:

SimpleList<T>, una lista enlazada simple genérica utilizada en la gestión de usuarios, cuentas y monederos. Esta estructura permite almacenar y recorrer elementos de forma secuencial, y se encuentra instanciada en la clase Nexpay como lista de User, Account y Wallet. A continuación, en la *Imagen 2* se presenta la clase *SimpleList* ubicada en datastructures, para una mejor visualización de esta, y en la *Imagen 3* su implementación en Nexpay para administrar dichas entidades.

```java
package proyecto.nexpay.web.datastructures;

import java.util.Iterator;
import java.util.stream.Stream;
import java.util.stream.StreamSupport;
import java.util.Spliterators;
import java.util.Spliterator;

public class SimpleList<T> implements Iterable<T> {

    private int size;
    private Node<T> firstNode;
    private Node<T> lastNode;

    public SimpleList() {
        this.size = 0;
        this.firstNode = null;
        this.lastNode = null;
    }

    public int getSize() { return size; }

    public void setSize(int size) { this.size = size; }

    public Node<T> getFirstNode() { return firstNode; }

    public void setFirstNode(Node<T> firstNode) { this.firstNode = firstNode; }

    public Node<T> getLastNode() { return lastNode; }

    public void setLastNode(Node<T> lastNode) { this.lastNode = lastNode; }
```

Imagen 2. Clase SimpleList.
Fuente: propia.

```java
package proyecto.nexpay.web.model;

import ...

public class Nexpay implements Serializable {

    private static final long serialVersionUID = 1L;
    private static Nexpay instance;

    private SimpleList<User> users;
    private SimpleList<Account> accounts;
    private DoubleLinkedList<Transaction> transactions;
    private SimpleList<Wallet> wallets;

    private UserCRUD userCRUD;
    private AccountCRUD accountCRUD;
    private TransactionCRUD transactionCRUD;
    private WalletCRUD walletCRUD;

    private TransactionManager TManager;
    private ScheduledTransactionManager SManager;
    private ScheduledTransactionExecutor executor;

    private Thread backupThread;

    private Nexpay(NotificationManager notificationManager) {
        this.users = new SimpleList<>();
        this.accounts = new SimpleList<>();
        this.wallets = new SimpleList<>();
        this.transactions = new DoubleLinkedList<>();
```

Imagen 3. Implementación de SimpleList en Nexpay.
Fuente: propia.

DoubleLinkedList<T>, una lista doblemente enlazada que permite recorrer elementos tanto hacia adelante como hacia atrás. Esta estructura es utilizada específicamente para almacenar las transacciones (Transaction) del sistema, permitiendo operaciones más flexibles y eficientes sobre el historial de movimientos. A continuación, en la *Imagen 3* se presenta la clase *DoubleLinkedList* ubicada en datastructures, para una mejor visualización de esta, y en la *Imagen 3* también se puede visualizar su implementación en Nexpay para administrar el listado de transacciones.

```java
package proyecto.nexpay.web.datastructures;

import java.util.Iterator;

public class DoubleLinkedList<T> implements Iterable<T>{
    private int size;
    private DoubleNode<T> firstNode;
    private DoubleNode<T> lastNode;

    public DoubleLinkedList() {
        this.size = 0;
        this.firstNode = null;
        this.lastNode = null;
    }

    public int getSize() { return size; }

    public void setSize(int size) { this.size = size; }

    public DoubleNode<T> getFirstNode() { return firstNode; }

    public void setFirstNode(DoubleNode<T> firstNode) { this.firstNode = firstNode; }

    public DoubleNode<T> getLastNode() { return lastNode; }

    public void setLastNode(DoubleNode<T> lastNode) { this.lastNode = lastNode; }

    public void addFirst(T nodeValue) {
        DoubleNode<T> newNode = new DoubleNode<>(nodeValue);
```

Imagen 4. Clase DoubleLinkedList.
Fuente: propia.

Stack<T>, una estructura tipo pila basada en nodos, utilizada para el manejo de transacciones reversibles. Cada vez que se ejecuta una transacción, esta se almacena en la pila, lo que permite implementar la funcionalidad de "deshacer" la última transacción. Esta pila es utilizada en la clase TransactionManager. A continuación, en la *Imagen 5* se presenta la clase *Stack* ubicada en datastructures, para una mejor visualización de esta, y en la *Imagen 6* también se puede visualizar su implementación en *TransactionManager* para administrar la pila de transacciones.

```java
package proyecto.nexpay.web.datastructures;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class Stack<T> implements Iterable<T> {
    private Node<T> top;
    private int size;

    public Stack() {
        this.top = null;
        this.size = 0;
    }

    public void push(T data) {
        Node<T> newNode = new Node<>(data);
        newNode.setNextNode(top);
        top = newNode;
        size++;
    }

    public T pop() {
        if (isEmpty()) {
            throw new NoSuchElementException("Stack is empty.");
        }
        T data = top.getData();
        top = top.getNextNode();
        size--;
        return data;
    }
```

Imagen 5. Clase Stack.
Fuente: propia.

```java
package proyecto.nexpay.web.service;

import proyecto.nexpay.web.datastructures.Stack;
import proyecto.nexpay.web.model.*;

public class TransactionManager {

    private final Nexpay nexpay;
    private final TransactionCRUD transactionCRUD;
    private final AccountCRUD accountCRUD;
    private final Stack<Transaction> transactionStack;
    private final NotificationManager notificationManager;
    private boolean undoTransaction;
    private final Stack<Transaction> revertedTransactions;

    public TransactionManager(Nexpay nexpay) {
        this.nexpay = nexpay;
        this.transactionCRUD = nexpay.getTransactionCRUD();
        this.accountCRUD = nexpay.getAccountCRUD();
        this.transactionStack = new Stack<>();
        this.notificationManager = new NotificationManager(nexpay);
        this.undoTransaction = false;
        this.revertedTransactions = new Stack<>();
    }

    public boolean isUndoTransaction() {
        return undoTransaction;
    }
```

Imagen 6. Implementación de Stack en TransactionManager.
Fuente: propia.

PriorityQueue<T>, una cola de prioridad implementada para gestionar las transacciones programadas, es decir, operaciones financieras que deben ejecutarse en una fecha futura. Esta estructura es manejada desde la clase ScheduledTransactionManager, donde se ordenan automáticamente las transacciones por fecha programada de ejecución. A continuación, en la *Imagen 7* se presenta la clase *PriorityQueue* ubicada en datastructures, para una mejor visualización de esta, y en la *Imagen 8* también se puede visualizar su implementación en *ScheduledTransactionManager* para ser implementada al momento de realizar un transacción, en TransactionManager.

```java
package proyecto.nexpay.web.datastructures;

import java.util.Iterator;

public class PriorityQueue<T extends Comparable<T>> implements Iterable<T> {

    private Node<T> head;
    private Node<T> tail;
    private int size;
    private boolean shutdown = false;


    public Node<T> getHead() { return head; }

    public void setHead(Node<T> head) { this.head = head; }

    public Node<T> getTail() { return tail; }

    public void setTail(Node<T> tail) { this.tail = tail; }

    public int getSize() { return size; }

    public void setSize(int size) { this.size = size; }

    public synchronized void enqueue(T data) {
        Node<T> newNode = new Node<>(data);

        if (head == null) {
            head = tail = newNode;
        } else if (data.compareTo(head.getData()) < 0) {
            newNode.setNextNode(head);
```

Imagen 7. Clase Priority Queue.
Fuente: propia.

```java
package proyecto.nexpay.web.service;

import proyecto.nexpay.web.datastructures.DoubleLinkedList;
import proyecto.nexpay.web.datastructures.PriorityQueue;
import proyecto.nexpay.web.model.Transaction;
import proyecto.nexpay.web.persistence.TransactionPersistence;

import java.io.IOException;
import java.time.LocalDateTime;

public class ScheduledTransactionManager {
    private TransactionManager transactionManager;
    private PriorityQueue<ScheduledTransaction> scheduledTransactions;
    TransactionPersistence Pt = TransactionPersistence.getInstance();

    public PriorityQueue<ScheduledTransaction> getScheduledTransactions() {
        return scheduledTransactions;
    }

    public ScheduledTransactionManager(TransactionManager transactionManager) {
        this.transactionManager = transactionManager;
        this.scheduledTransactions = new PriorityQueue<>();
        loadScheduledTransactions();
    }

    public void scheduleTransaction(Transaction transaction, LocalDateTime date) {
        ScheduledTransaction scheduledTransaction = new ScheduledTransaction(transaction, date);
        scheduledTransactions.enqueue(scheduledTransaction);
        Pt.saveScheduledTransaction(scheduledTransaction);
        System.out.println("Transaction scheduled for: " + date);
    }
```

Imagen 8. Implementación de PriorityQueue en ScheduledTransactionManager.
Fuente: propia.

BinarySearchTree<T>, un árbol binario de búsqueda(BST) fue implementado para gestionar la información de los puntos acumulados por cada cliente de manera eficiente. Esta estructura fue clave para mantener un registro ordenado de los puntos de forma que se pueda acceder rápidamente a la información necesaria para la actualización y la consulta. La implementación de esta estructura se da desde la clase PointManager, clase donde se encuentran los métodos para manejar los puntos acumulados de un usuario. En la *imagen 9* se observa la implementación de la clase BST y en la *imagen 10* se observa la aplicación en la clase *PointManager.*

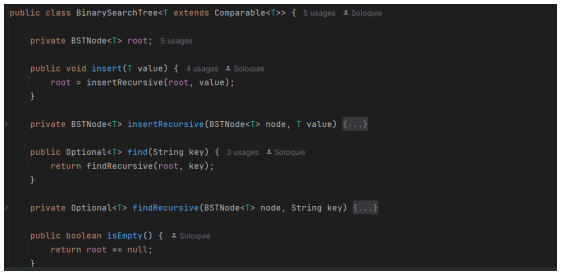


Imagen 9. Class BinarySearchTree<T>.
Fuente: propia.

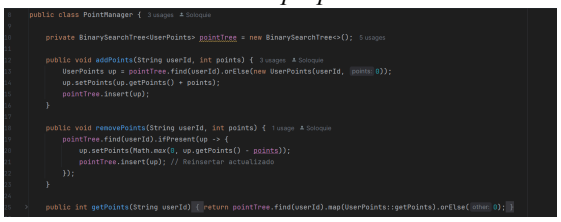


Imagen 10. Implementación de BinarySearchTree en PointManager.
Fuente: propia.

AVLTree<T>, un árbol binario auto-balanceado se usa para gestionar la eficiencia y balance de los datos relacionados de los rangos de los usuarios. Esta estructura se utilizó específicamente para mantener actualizados los rangos de los usuarios a medida que sus puntos cambien, lo cual también garantiza que se asignen correctamente los beneficios y privilegios dentro del sistema. La implementación de este árbol se da en la clase RankManager en la cual se maneja el rango de los usuarios según sus puntos. En la *imagen 11* se ve la implementación del *ArbolAVL* en su clase y en la *imagen 12* se ve su uso en la clase *RankManager.*

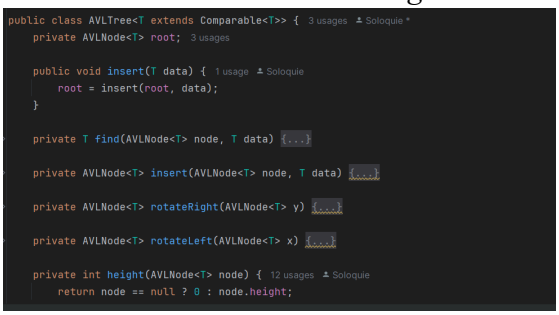


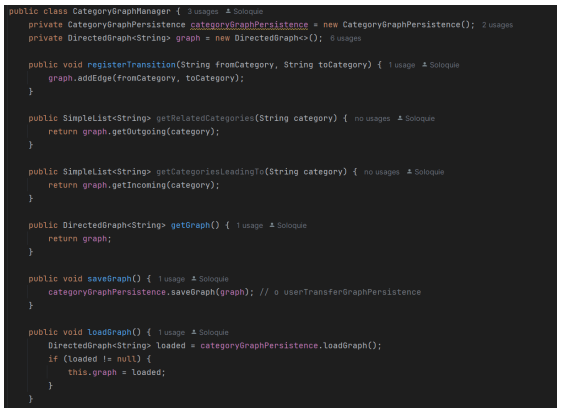
Imagen 11. Clase AVLTree.
Fuente: propia.

Imagen 12. Implementación de AVLTree en RankManager.
Fuente: propia.

DirectedGraph<T>, un grafo dirigido es utilizado para el análisis avanzado de patrones de gasto y relaciones de transferencia entre usuarios, para esto se usó un grafo dirigido implementado con nodos y listas de adyacencia. El grafo permite representar relaciones dirigidas como las transacciones entre categorías de gasto. El uso de esta estructura de datos se da en la clase CategoryGraphManager en la cual se guardan las relaciones de las categorías de gastos de los usuarios. En la *imagen 13* se observa la clase *DirectedGraph* y en la *imagen 14* se ve la implementación de la estructura.

```
public class DirectedGraph<T> implements Serializable {  23 usages  Soloquin

    private final SimpleList<GraphNode<T>> nodes = new SimpleList<>();  10 usages

    public void addNode(T value) {  no usages  Soloquin
        if (findNode(value) == null) {
            nodes.addLast(new GraphNode<>(value));
        }
    }

    public void addEdge(T from, T to) {  4 usages  Soloquin
        GraphNode<T> fromNode = findNode(from);
        GraphNode<T> toNode = findNode(to);

        if (fromNode == null) {
            fromNode = new GraphNode<>(from);
            nodes.addLast(fromNode);
        }

        if (toNode == null) {
            toNode = new GraphNode<>(to);
            nodes.addLast(toNode);
        }

        if (!fromNode.getAdjacency().contains(to)) {
            fromNode.getAdjacency().addLast(to);
        }
    }
```

Imagen 13. Clase DirectedGraph<T>.
Fuente: propia.



```
public class CategoryGraphManager {  3 usages  Soloquin
    private CategoryGraphPersistence categoryGraphPersistence = new CategoryGraphPersistence();  2 usages
    private DirectedGraph<String> graph = new DirectedGraph<>();  6 usages

    public void registerTransition(String fromCategory, String toCategory) {  1 usage  Soloquin
        graph.addEdge(fromCategory, toCategory);
    }

    public SimpleList<String> getRelatedCategories(String category) {  no usages  Soloquin
        return graph.getOutgoing(category);
    }

    public SimpleList<String> getCategoriesLeadingTo(String category) {  no usages  Soloquin
        return graph.getIncoming(category);
    }

    public DirectedGraph<String> getGraph() {  1 usage  Soloquin
        return graph;
    }

    public void saveGraph() {  1 usage  Soloquin
        categoryGraphPersistence.saveGraph(graph); // o userTransferGraphPersistence
    }

    public void loadGraph() {  1 usage  Soloquin
        DirectedGraph<String> loaded = categoryGraphPersistence.loadGraph();
        if (loaded != null) {
            this.graph = loaded;
        }
    }
```

Imagen 12. Implementación de DirectedGraph en CategoryGraphManager.
Fuente: propia.

Todas estas estructuras fueron integradas en el núcleo del sistema a través de la clase Nexpay, que actúa como fachada y orquestador de la lógica interna. Esta clase implementa el patrón Singleton y concentra las colecciones principales que representan el estado del sistema. Específicamente, en ella se encuentran los siguientes atributos:

SimpleList<User> users: almacena los usuarios registrados.

SimpleList<Account> accounts: almacena las cuentas bancarias asociadas a los monederos.

SimpleList<Wallet> wallets: contiene los monederos virtuales creados por cada usuario.

DoubleLinkedList<Transaction> transactions: almacena el historial completo de transacciones.

Stack<Transaction> transactionStack: guarda las transacciones ejecutadas recientemente para permitir su reversión.

PriorityQueue<ScheduledTransaction> scheduledTransactions: organiza las transacciones programadas en función de su fecha de ejecución.

5. IMPLEMENTACIÓN SISTEMA DE PUNTOS Y GESTION DE RANGOS

El sistema de puntos en Nexpay fue diseñado para incentivar la participación y fidelidad de los usuarios mediante la acumulación de puntos por sus transacciones financieras y la asignación de rangos que determinan los beneficios y privilegios dentro de la plataforma.

Para gestionar los puntos acumulados, se implementó una estructura propia de un Árbol Binario de Búsqueda (Binary Search Tree BST) *(Imagen 9)*. Dicho árbol permite almacenar y organizar los puntos asociados a cada usuario de manera eficiente, facilitando las operaciones de inserción, búsqueda y actualización con una complejidad promedio logarítmica.

Cada vez que un usuario realiza una transacción válida, el sistema calcula los puntos correspondientes a dicha operación y los suma al total acumulado del usuario, tanto en la estructura BST como en la entidad de usuario persistida, esto se lo ve aplicado en la clase *TransaccionManager* en el método *executeTransaccion* mediante *PointManager* la cual es

encargada de manejar los puntos del usuario, *Imagen 13*.

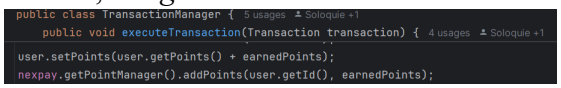


Imagen 13. Asignación de puntos mediante PointManager.
Fuente: propia.

Además, se mantiene un historial detallado de las transacciones relacionadas con los puntos, lo que permite ofrecer un seguimiento transparente y ordenado de las actividades que generan puntos, esto se da mediante la clase *PointHistoryPersistence* y se puede observar en la *imagen 14* .

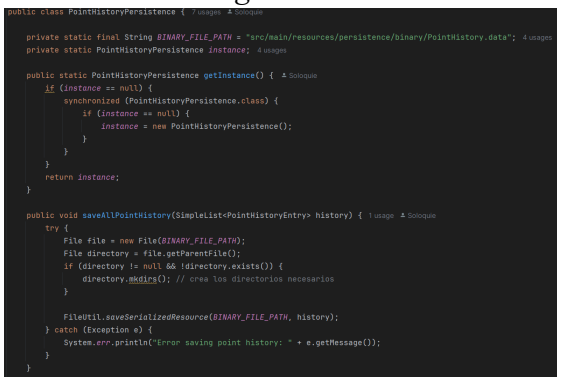


Imagen 14. Persistencia de los puntos acumulados de usuarios.
Fuente: propia.

Para mantener actualizados los rangos de los usuarios conforme evolucionan sus puntos acumulados, se implementó un árbol AVL. Esta estructura garantiza que las operaciones de inserción, eliminación y búsqueda se realicen en tiempo logarítmico incluso en el peor caso, manteniendo la eficiencia del sistema.

El Árbol AVL se utiliza para reflejar el nivel de cada usuario basado en sus puntos, categorizándolos en rangos como Bronce, Plata, Oro y Platino. Estos rangos determinan el acceso a diversos beneficios, tales como descuentos, bonificaciones y promociones exclusivas. La actualización dinámica del árbol permite que el sistema responda en tiempo real a los cambios en los puntos, asegurando que los usuarios reciban el rango adecuado a su perfil de consumo y participación. Esta funcionalidad se la ve manejada por la clase *RankManager* la cual tiene un árbol AVL y se encarga de calcular automáticamente el rango del cliente, esto se lo ve en la *imagen 15*.

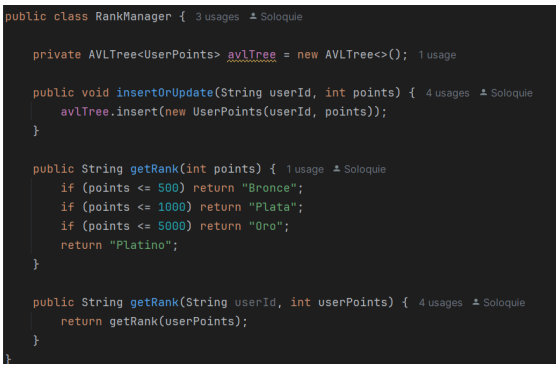


Imagen 15. Manejo del rango de clientes mediante RankManager.
Fuente: propia.

5.1. Ejemplos prácticos del sistema de puntos y su impacto en el monedero virtual

1. Acumulación de puntos por transacciones frecuentes

- Cada vez que un usuario realiza un depósito, retiro o transferencia, acumula puntos automáticamente según el monto y tipo de operación.
- Por ejemplo, un depósito de $1000 podría otorgar 10 puntos, mientras que una transferencia de $500 podría otorgar 5 puntos.

2. Asignación de rangos según puntos acumulados

- Los usuarios con más puntos alcanzan rangos superiores como Plata, Oro o Platino.
- Por ejemplo, un usuario que supera los 1000 puntos obtiene rango Oro

y accede a beneficios exclusivos como menores comisiones o promociones especiales.

Las dos funciones mostradas anteriormente se las puede observar en la siguiente imagen:

Imagen 16. Acumulación de puntos y rango de usuario .
Fuente: propia.

3. Canje de puntos por beneficios económicos

- Los puntos pueden canjearse dentro del monedero virtual por descuentos en comisiones, retiros gratuitos o bonos en dinero real.
- Un usuario que canjea 500 puntos puede obtener un mes sin cargos por retiros, reduciendo sus costos.
- Otro puede usar 1000 puntos para recibir un bono de $50 directo en su cuenta del monedero, incrementando su saldo disponible.

Imagen 17. Canje de puntos por beneficios económicos .
Fuente: propia.

5. Análisis de comportamiento y mejora continua

- El sistema registra y analiza el historial de puntos y transacciones para identificar patrones de gasto.
- Esto permite al monedero virtual anticipar necesidades, recomendar productos y ajustar las estrategias de incentivos para maximizar el uso y la satisfacción.

Imagen 18. Análisis de comportamiento.
Fuente: propia.

La interfaz visual del sistema, construida con Spring Boot y Thymeleaf, permite interactuar con estas estructuras mediante formularios HTML. Las vistas están ubicadas en la carpeta resources/templates, y las operaciones CRUD que se realizan desde la interfaz afectan directamente las estructuras mencionadas. Además, el sistema implementa persistencia de datos mediante archivos .txt ubicados en resources, de forma que todas las estructuras pueden cargarse y guardarse correctamente entre sesiones.

La implementación manual de estas estructuras no solo permitió un mejor control sobre el comportamiento del sistema, sino que además facilitó la integración de funcionalidades avanzadas como el sistema de transacciones reversibles, el sistema de puntos por transacción, y la gestión de monederos independientes para cada usuario mediante un grafo personalizado.

En resumen, el uso de estas estructuras implementadas desde cero representa uno de los pilares más importantes del

proyecto, ya que sustenta toda la lógica de almacenamiento, manipulación y visualización de los datos sin depender de herramientas externas, consolidando así un enfoque didáctico y profesional para la solución desarrollada.

6. CONCLUSIONES

- La implementación de estructuras de datos personalizadas en el sistema Nexpay permitió un control total sobre la gestión y manipulación de la información, asegurando eficiencia y adecuación a las necesidades específicas de cada módulo funcional.

- El uso de listas enlazadas (SimpleList y DoubleLinkedList) resultó fundamental para representar entidades dinámicas como usuarios, cuentas, transacciones y monederos, facilitando operaciones de inserción, eliminación y recorrido sin recurrir a colecciones estándar.

- La pila (Stack) utilizada en el manejo de transacciones reversibles y la cola de prioridad (PriorityQueue) para las transacciones programadas, demostraron la capacidad del sistema para incorporar lógica avanzada de control, programación y retroceso de operaciones, mejorando así la robustez y trazabilidad de las acciones del usuario.

- La arquitectura modular del proyecto, combinada con la integración de Spring Boot y Thymeleaf, permitió una separación clara entre la lógica de negocio, la persistencia y las vistas, lo cual favorece la mantenibilidad y escalabilidad del sistema a futuro.

- La implementación de estructuras de datos propias como el árbol binario de búsqueda (BinarySearchTree), el árbol AVL (AVLTree) y el grafo dirigido (DirectedGraph) permitió una gestión optimizada y especializada de los puntos de usuario, rangos y patrones de gasto, respectivamente. Estas estructuras garantizan eficiencia en las operaciones de búsqueda, actualización y análisis dentro del sistema.

- El sistema de puntos, apoyado en el árbol binario de búsqueda, ofrece un mecanismo dinámico de incentivos que favorece la fidelización y el compromiso del usuario. La integración del árbol AVL para la gestión de rangos asegura que la asignación de beneficios sea precisa y adaptable al comportamiento del usuario en tiempo real.

- El uso del grafo dirigido para analizar las relaciones entre categorías de gasto y transferencias entre usuarios proporciona una herramienta avanzada para la identificación de patrones financieros, posibilitando la generación de informes detallados que enriquecen la experiencia y el control financiero de los usuarios.