UNIVERSIDAD DIEGO PORTALES
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES

Laboratorio 4: Conecta 4

Autores:
Valentina Mella
Valentina Diaz

Profesor:
Marcos Fantoval

16 de Junio 2025

Índice

1. Introducción

En el presente informe, se desarrollará una implementación del juego Conecta 4 utilizando estructuras de datos avanzadas para gestionar jugadores, partidas y estadísticas de manera eficiente. El objetivo principal es demostrar cómo un árbol binario de búsqueda (BST) y una Tabla de Hash (HashST) pueden optimizar el almacenamiento y recuperación de datos en un entorno interactivo.

Para lograrlo, el sistema se compone de varias clases, entre las cuales las clases que se pueden mencionar como clave son las siguientes: la clase Player modela a cada jugador, registrando su nombre, victorias y empates. La clase Scoreboard utiliza un BST para mantener un ranking ordenado de jugadores según sus victorias, mientras que la HashST permite acceder rápidamente a los datos de un jugador específico. Por otro lado, la clase ConnectFour implementa la lógica del juego, incluyendo el tablero, la validación de movimientos y la detección de victorias. Finalmente, la clase Game actúa como intermediario, coordinando la interacción entre los jugadores y el tablero, y mostrando los resultados en tiempo real.

La elección de estas estructuras de datos no es arbitraria. El BST garantiza que las operaciones de inserción, eliminación y búsqueda se realicen en tiempo logarítmico (O(log n)), lo que es ideal para mantener un ranking ordenado. Por su parte, la HashST ofrece acceso constante (O(1)) a los datos de cualquier jugador, acelerando las actualizaciones de estadísticas durante las partidas. Esta combinación permite que el sistema sea escalable y eficiente, incluso con un gran número de jugadores registrados.

La relevancia de este proyecto radica en su enfoque práctico. No solo aplica conceptos teóricos de estructuras de datos, sino que también ilustra cómo estás pueden impactar en el rendimiento de una aplicación real. Además, el uso de un juego como Conecta 4 hace que el aprendizaje sea más tangible. En definitiva, este trabajo sirve como un modelo para el desarrollo de sistemas que requieren gestión dinámica de datos y respuestas en tiempo real, destacando la importancia de elegir las herramientas adecuadas en el diseño de software.

2. Implementación

El código implementado se encuentra en el siguiente enlace: InformeLab5

La implementación se realizó en Java, las clases principales son:

```java
class BST<Key extends Comparable<Key>, Value>{
    private Node root;
    private class Node{
        Key key;
        Value val;
        Node left, right;

        Node(Key key, Value val) {
            this.key = key;
            this.val = val;}
    }
    public void put(Key key, Value val) {
        root = put(root, key, val);
    }

    private Node put(Node x, Key key, Value val){
        if (x == null) return new Node(key, val);
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x.left = put(x.left, key, val);
        else if (cmp > 0) x.right = put(x.right, key, val);
        else x.val = val;

        return x;
    }
```

```java
class HashST<Key, Value> {
    private static final int INIT_CAPACITY = 16;
    private Node[] st;
    private int n;

    private static class Node {
        Object key, val;
        Node next;

        Node(Object key, Object val, Node next) {
            this.key = key;
            this.val = val;
            this.next = next;
        }
    }

    public HashST() {
        this(INIT_CAPACITY);
    }
```

- **BST y HashST**: Se implementaron desde cero para gestionar a los jugadores y sus estadísticas. BST se utilizó para ordenar a los jugadores por número de victorias, mientras que HashST permitió un acceso rápido a los datos de cada jugador mediante su nombre.

```java
class Player implements Comparable<Player> {
    private String playerName;
    private int wins, draws, losses;

    public Player(String playerName) {
        this.playerName = playerName;
        this.wins = 0;
        this.draws = 0;
        this.losses = 0;
    }

    public void addWin() { wins++; }
    public void addDraw() { draws++; }
    public void addLoss() { losses++; }
```

Figura 1: Class Player

- **Player**: Representa a un jugador con atributos entre los cuales se hayan nombre, victorias, empates y derrotas. Los métodos incluyen actualización de estadísticas y cálculo del porcentaje de victorias.

```
class Scoreboard {
    private BST<Integer, List<String>> winTree = new BST<>();
    private HashST<String, Player> players = new HashST<>();
    private int playedGames = 0;

    public void registerPlayer(String playerName) {
        if (!players.contains(playerName)) {
            Player p = new Player(playerName);
            players.put(playerName, p);
            addToWinTree(p);
        }
    }
```

Figura 2: Class Scorboard

- **Scoreboard**: Gestiona el registro de jugadores y los resultados de las partidas. Utiliza BST para ordenar jugadores por victorias y HashST para almacenar los objetos *Player*. Los métodos incluyen registro de jugadores, actualización de resultados y consultas por rangos de victorias.

```
class ConnectFour {
    private char[][] grid = new char[6][7];
    private char currentSymbol = 'X';
    private boolean gameOver = false;
    private char winner = ' ';

    public ConnectFour() {
        for (int fila = 0; fila < 6; fila++) {
            for (int col = 0; col < 7; col++) {
                grid[fila][col] = ' ';
            }
        }
    }
```

Figura 3: Class ConnectFour

- **ConnectFour**: Implementa la lógica del juego, incluyendo el tablero, los turnos y detecta las victorias o empates. El método *makeMove* coloca fichas en el tablero y verifica si el juego ha terminado.

```
class Game{
    private String playerA, playerB;
    private ConnectFour cf = new ConnectFour();
    private Scanner sc = new Scanner(System.in);

    public Game(String playerA, String playerB) {
        this.playerA = playerA;
        this.playerB = playerB;
    }
```

Figura 4: Class Game

- **Game**: Controla el flujo de una partida, alternando turnos entre los jugadores y mostrando el estado del tablero. Utiliza *ConnectFour* para la lógica del juego y *Scoreboard* para registrar los resultados.
- **Main**: Se aplica para verificar que el código funciona y a partir de este se muestran las partidas en la consola.

3. Análisis

3.1. Métodos de la clase Scoreboard

- **addGameResult**:

  Complejidad: $O(\log n + m)$

  Explicación: Operación en BST ($O(\log n)$) + actualización de listas ($O(m)$)

- **registerPlayer**:

  Complejidad: $O(1)$ promedio, $O(n)$ peor caso

  Explicación: Inserción en tabla hash ($O(1)$) + BST ($O(\log n)$)

- **checkPlayer**:

  Complejidad: $O(1)$ promedio, $O(n)$ peor caso

  Explicación: Búsqueda en tabla hash

- **winRange**:

  Complejidad: $O(\log n + k)$

  Explicación: Búsqueda en BST ($O(\log n)$) + recorrido de k elementos

- **winSuccessor**:

  Complejidad: $O(\log n)$

  Explicación: Búsqueda de sucesor en BST balanceado

3.2. Ventajas

La implementación propuesta destaca por su eficiencia en el manejo de operaciones clave, gracias a la combinación estratégica de un árbol binario de búsqueda (BST) y una tabla de hash (HashST). El BST permite organizar a los jugadores según su número de victorias, facilitando consultas rápidas para generar rankings o identificar sucesores; todas estas son operaciones esenciales en un sistema de puntuaciones. Por otro lado, la tabla de hash agiliza el acceso directo a los datos de cada jugador mediante su nombre, optimizando las búsquedas individuales. Esta dualidad mejora significativamente el rendimiento general, contribuyendo desde la visualización de estadísticas hasta la gestión de torneos. Además, el diseño modular del código, con clases bien definidas como `Player`, `Scoreboard` y `ConnectFour`, promueve la reutilización y el mantenimiento sencillo, aspectos clave para futuras extensiones del sistema.

3.3. Desventajas

Como desventajas no se puede evitar mencionar, la sincronización entre el BST y la tabla de hash introduce complejidad adicional, ya que cada actualización en las estadísticas de un jugador requiere modificar ambas estructuras de datos. Esto no solo aumenta el riesgo de errores, sino que también puede ralentizar operaciones que, en teoría, deberían ser rápidas. Otro punto crítico a mencionar es el rendimiento de la tabla de hash en escenarios adversos: si las colisiones son frecuentes debido a una función de hash poco eficiente o a una capacidad inicial inadecuada, el tiempo de acceso podría degradarse significativamente. Estas desventajas no invalidan el diseño, pero señalan áreas de mejora para adaptar el sistema a contextos más demandantes o de mayor escala.

3.4. Desafíos

A lo largo de la ardua programación del código, donde se presentó mayor desafío, fue en la clase ConnectFour, ya que, el equipo termino agregando más métodos de los solicitados para la fiel y correcta compilación del código, la mayor complicación presentada fue a la hora de presentar las victorias, ya que se debía tomar en cuenta cuando se formaran 4 fichas seguidas ya sea en línea recta, vertical, horizontal o diagonal, esto presento un desafío para el equipo al ser la parte más complicada de diseñar por parte del equipo; sin embargo, con la correcta ejecución y búsqueda de información, el equipo logro solucionar este problema, dando paso a la actual y mejorada versión del código, la cual fue utilizada a lo largo de todo este informe.

3.5. Extensión

Para que el código generado pueda ser utilizado en torneos, se proponen las siguientes mejoras:

- Nueva clase `Tournament` para gestionar múltiples rondas
  (Permitiría organizar partidas en fases eliminatorias o por grupos)
- Atributos adicionales en `Player`:
  - Puntos acumulados
    (Necesario para llevar registro del desempeño global en el torneo)
  - Comparadores para clasificación
    (Facilitaría ordenar jugadores por distintos criterios de desempate)
- Nuevas operaciones en `Scoreboard`:
  - Generación de rankings por torneo
    (Mostraría posiciones actualizadas según el avance del torneo)

3.6. Revisión de Estructuras de Datos

Como alternativa a las implementaciones propias de estructuras de datos, se pueden considerar las siguientes opciones de la librería estándar de Java, cada una con sus particulares métodos:

Reemplazo del BST implementado:

- `TreeMap<K,V>` de Java Collections Framework

- **Ventajas**:

  La clase `TreeMap` proporciona una implementación altamente optimizada y probada en entornos productivos, incluyendo métodos especializados como `subMap()` para manejo de rangos y `higherKey()` para búsqueda de sucesores, junto con el beneficio del balanceo automático mediante su implementación como árbol Red-Black.

- **Desventajas**:

  Entre sus limitaciones se encuentra el control reducido sobre la implementación interna del árbol, un mayor consumo de memoria debido a la estructura de objetos de Java, y un rendimiento ligeramente inferior cuando se trabaja con conjuntos pequeños de datos (n ¡1000 elementos).

Reemplazo de la Tabla Hash implementado:

- `HashMap<K,V>` de Java Collections Framework

- **Ventajas**:

  La implementación estándar de `HashMap` ofrece importantes ventajas como el redimensionamiento automático al alcanzar el factor de carga y/o iteración más eficiente mediante iteradores optimizados.

- **Desventajas**:

  Como contraparte, esta estructura requiere implementaciones robustas de `hashCode()` y `equals()`, presenta comportamiento no determinista en el orden de iteración y genera mayor overhead por el manejo de concurrencia en su versión básica.

3.7. Pruebas

```
Vale (X), elige columna (0-6): 3
 0 1 2 3 4 5 6
---------------
| | | | | | | |
---------------
| | | | | | | |
---------------
| | | | | | | |
---------------
| | | | | | | |
---------------
|O|O|O| | | | |
---------------
|X|X|X|X| | | |
---------------
Vale gana
```

Figura 5: Victoria Horizontal

```
Javi (O), elige columna (0-6): 0
 0 1 2 3 4 5 6
---------------
| | | | | | | |
---------------
| | | | | | | |
---------------
|O| | | | | | |
---------------
|O|X| | | | | |
---------------
|O|X| | | | | |
---------------
|O|X|X| | | | |
---------------
Javi gana
```

Figura 6: Victoria Vertical

```
vale (X), elige columna (0-6): 3
 0 1 2 3 4 5 6
---------------
| | | | | | | |
---------------
| | | | | | | |
---------------
| | | |X| | | |
---------------
| | |X|O| | | |
---------------
| |X|O|O| | | |
---------------
|X|O|O|X|X| | |
---------------
vale gana
```

Figura 7: Victoria Diagonal

```
Javi (O), elige columna (0-6): 6
 0 1 2 3 4 5 6
---------------
|O|O|O|X|O|O|O|
---------------
|X|X|X|O|X|X|X|
---------------
|O|O|O|X|O|O|O|
---------------
|X|X|X|O|X|X|X|
---------------
|O|O|O|X|O|O|O|
---------------
|X|X|X|O|X|X|X|
---------------
?Empate!
```

Figura 8: Empate

Observaciones durante pruebas interactivas:

- Visualización correcta del tablero
- Detección adecuada de victorias en todas direcciones
- Manejo correcto de empates

El código se desempeña de manera correcta sin sufrir errores o complicaciones a lo largo de las partidas realizadas, las partidas fueron puestas a prueba en la página de Programiz Online Java Compiler, por lo cual se destaca que el código se ejecuta de manera excepcionalmente útil en un programa compilador de java online.

4. Conclusión

Finalmente, se puede determinar que el código realizado fue ejecutado con éxito, logrando demostrar que se implementó de manera adecuada y efectiva el juego Conecta 4, creando las clases BST, HashST, Player, Scoreboard, ConnectFour y Game en Java, las cuales permitieron gestionar jugadores, partidas y la lógica del juego de manera eficiente, todo esto se realizó con el fin de generar el juego conecta 4 para la clase de estructura de datos y algoritmos de la universidad diego portales, donde las estudiantes se encargaron de aplicar los atributos y métodos solicitados para la ejecución correcta del código.

Evidenciando a través del análisis de complejidad la eficacia del sistema implementado, donde las operaciones clave mostraron un comportamiento predecible. O(log n) para las consultas en el BST y O(1) para los accesos en la HashST, demostrando que al aumentar el número de jugadores y partidas, el sistema mantiene su rendimiento gracias a las estructuras de datos seleccionadas.

Esta experiencia permitió comprender la importancia de elegir estructuras de datos adecuadas para problemas específicos. En casos reales, como sistemas de ranking para competencias deportivas o plataformas de juegos online, donde la cantidad de usuarios suele variar constantemente, siendo necesario implementar estructuras personalizadas, como lo fue en el caso de este laboratorio cuando:

- Se requiera un control preciso sobre el almacenamiento y recuperación de datos para optimizar el rendimiento en operaciones críticas.
- Las estructuras estándar no ofrezcan la flexibilidad necesaria para manejar relaciones complejas entre los datos, como el mantenimiento simultáneo de un índice ordenado y accesos rápidos por clave.

El laboratorio demostró que, aunque las librerías estándar son útiles en muchos casos, existen escenarios donde implementaciones personalizadas pueden ofrecer ventajas significativas en términos de eficiencia y adaptación a requisitos específicos.