Universidad Diego Portales
escuela de informática & telecomunicaciones

ESTRUCTURAS DE DATOS & ANÁLISIS DE ALGORITMOS

Laboratorio 5 Conecta 4

Autores:
Roberto Ibaceta Cisternas
Cristóbal Castro Peña

Profesor: *Marcos Fantoval*

28 de junio de 2025

Índice

1. Introducción

En presente laboratorio se detalla el diseño, implementación y posterior análisis del juego `Conecta 4` como una aplicación orientada a objetos, integrando estructuras de datos fundamentales como árboles binarios de búsqueda y tablas hash. Este proyecto tiene como objetivo principal reforzar el uso y la comprensión de dichas estructuras al aplicarlas en un sistema dinámico que requiere una organización eficiente de la información, como el manejo de jugadores y el registro de partidas. A través de este laboratorio se abordarán conceptos como el manejo de colisiones, búsqueda eficiente, ordenamiento y la implementación de lógica de juego mediante estructuras de datos.

El juego `Conecta 4` sirve como base para implementar un sistema capaz de simular múltiples partidas entre jugadores, obtener estadísticas individuales y generar análisis relacionados con el desempeño y la organización de datos.

2. Implementación y Métodos

En esta sección se describe la arquitectura del software y las decisiones de diseño tomadas para construir el sistema de juego de `Conecta 4`. La implementación se llevó a cabo siguiendo el diseño planteado por la actividad, utilizando un desarrollo modular basado en cuatro clases principales: **Player**, **Scoreboard**, **ConnectFour**, **Game**, además de las estructuras de datos **BST** y **HashST**, las cuales fueron estudiadas en clases.

Para respaldar nuestro trabajo, se ha dispuesto todo el código en el siguiente repositorio de GitHub: https://github.com/camioner/lab5

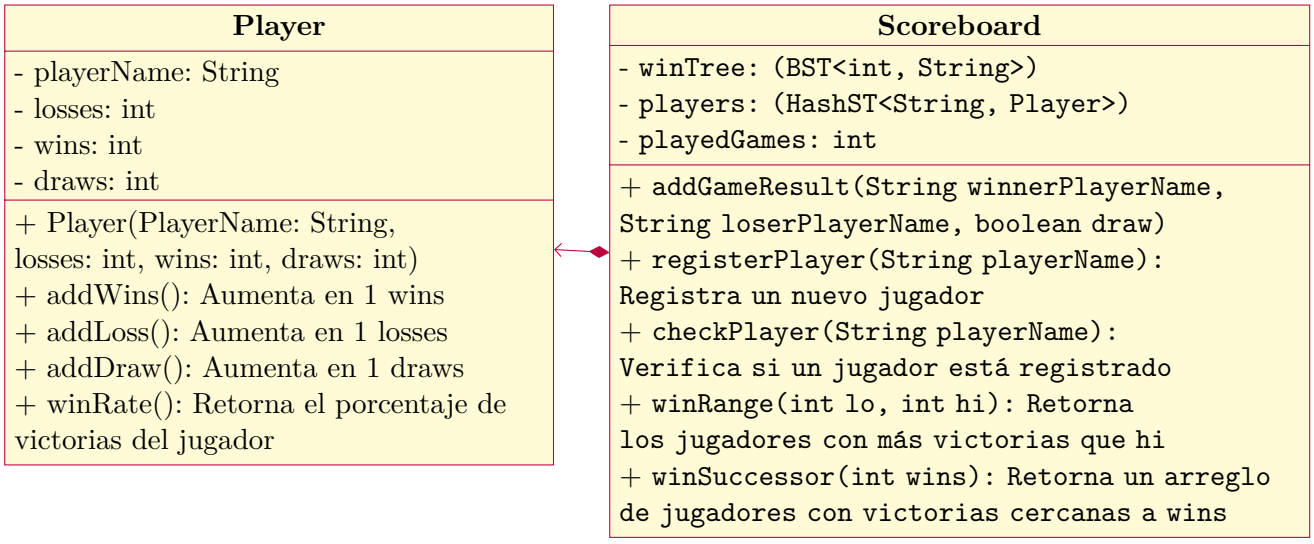

Figura 1: Diagrama UML de las clases `Player` y `Scoreboard`.

En este diagrama se observa la relación de composición entre dos clases: una encargada de crear los objetos `Player` y otra destinada a manejar operaciones básicas en torno a estos objetos, como por ejemplo verificar si el jugador ya está registrado. A continuación, se profundiza en cada una de las clases.

2.1. Clase Player

Descripción:
La clase `Player` representa a un jugador del juego Conecta 4. Está definida por atributos que almacenan su historial de partidas, junto con métodos para modificar y consultar dicha información.

Atributos:

- `playerName (String)`: Representa el nombre del jugador, usado como identificador único.
- `wins (int)`: Cantidad de partidas ganadas por el jugador.
- `draws (int)`: Cantidad de partidas empatadas.
- `losses (int)`: Cantidad de partidas perdidas.

Métodos:

- `addWin()`: Incrementa en 1 el contador de partidas ganadas.
- `addDraw()`: Incrementa en 1 el contador de partidas empatadas.
- `addLoss()`: Incrementa en 1 el contador de partidas perdidas.
- `winRate()`: Retorna el porcentaje de victorias sobre el total de partidas jugadas, como un valor decimal entre 0 y 1. Si no ha jugado ninguna partida, retorna 0.

```
public void addWin() {
    victoria += 1;
}

public void addDraw() {
    empates += 1;
}

public void addLoss() {
    perdidas += 1;
}

public double WinRate() {
    int total = victoria + perdidas + empates;
    WinRate = victoria / total;
    return WinRate;
}
```

Listing 1: Métodos de la clase Player

2.2. Clase Scoreboard

Descripción:
La clase `Scoreboard` se encarga de gestionar a los jugadores registrados y los resultados de las partidas jugadas. Utiliza estructuras de datos como árboles de búsqueda y tablas hash para organizar la información.

Atributos:

- `winTree (BST<int, String>)`: Árbol binario de búsqueda que ordena a los jugadores por su cantidad de victorias. La clave es el número de victorias y el valor el nombre del jugador.

- `players (HashST<String, Player>)`: Tabla hash que asocia el nombre de un jugador con su instancia de la clase `Player`.
- `playedGames (int)`: Número total de partidas registradas.

Métodos:

- `addGameResult(winnerPlayerName: String, looserPlayerName: String, draw: boolean)`: Registra el resultado de una partida, actualiza las estadísticas de los jugadores y modifica el árbol de victorias.
- `registerPlayer(playerName: String)`: Registra un nuevo jugador si no existe ya en el sistema.
- `checkPlayer(playerName: String)`: Verifica si un jugador con el nombre dado está registrado.
- `winRange(lo: int, hi: int)`: Retorna un arreglo con los jugadores cuyo número de victorias se encuentra entre los valores dados.
- `winSuccessor(wins: int)`: Retorna un arreglo con los jugadores que tienen el número de victorias mínimo mayor al valor especificado.

```
public void addgameresult(String Winname, String losername, boolean draw) {
    if (!players.contains(Winname) || !players.contains(losername)) return;
    Jugador ganador = (Jugador) players.get(Winname);
    Jugador perdedor = (Jugador) players.get(losername);

    if (draw) {
        ganador.addDraw();
        perdedor.addDraw();
    } else {
        if (ganador.victoria > 0) {
            wintree.root = BST.removeJugador(wintree.root, ganador.victoria, ganador.
    nombre);
        }
        if (perdedor.victoria > 0) {
            wintree.root = BST.removeJugador(wintree.root, perdedor.victoria,
    perdedor.nombre);
        }
        ganador.addWin();
        perdedor.addLoss();
        wintree.root = BST.push(wintree.root, ganador.victoria, ganador.nombre);
        wintree.root = BST.push(wintree.root, perdedor.victoria, perdedor.nombre);
    }
    playedgames++;
}
public void registPlayer(String playername) {
    if (!players.contains(playername)) {
        Jugador j = new Jugador(playername);
        players.add(playername, j);
    }
}
public Jugador[] winSuccessor(int wins) {
    BST.Node nodo = BST.successor(wintree.root, wins);
    if (nodo == null) return new Jugador[0];

    Jugador[] jugadores = new Jugador[nodo.jugadores.size()];
    for (int i = 0; i < nodo.jugadores.size(); i++) {
        jugadores[i] = (Jugador) players.get(nodo.jugadores.get(i));
    }
    return jugadores;
```

```
}
public boolean check4player(String playername) {
    return players.contains(playername);
}
public void guardarDatos() {
    try (PrintWriter writer = new PrintWriter(new FileWriter("jugadores.txt"))) {
        writer.println("playedGames=" + playedgames);
        ArrayList<Jugador> lista = players.getJugadores();
        for (Jugador j : lista) {
            writer.println(j.nombre + "," + j.victoria + "," + j.empates + "," + j.
    perdidas);
        }
    } catch (IOException e) {
        System.out.println("Error al guardar los datos: " + e.getMessage());
    }
}

public void cargarDatos() {
    try (BufferedReader reader = new BufferedReader(new FileReader("jugadores.txt")))
     {
        String linea = reader.readLine();
        if (linea.startsWith("playedGames=")) {
            playedgames = Integer.parseInt(linea.split("=")[1]);
        }
        String datos;
        while ((datos = reader.readLine()) != null) {
            String[] partes = datos.split(",");
            String nombre = partes[0];
            int wins = Integer.parseInt(partes[1]);
            int draws = Integer.parseInt(partes[2]);
            int losses = Integer.parseInt(partes[3]);

            Jugador j = new Jugador(nombre);
            j.victoria = wins;
            j.empates = draws;
            j.perdidas = losses;

            players.add(nombre, j);
            wintree.root = BST.push(wintree.root, wins, nombre);
        }
    } catch (IOException e) {
        System.out.println("No se pudo cargar el archivo: " + e.getMessage());
    }
}
public Jugador[] winRange(int lo, int hi) {
    ArrayList<String> nombres = new ArrayList<>();
    BST.winRange(wintree.root, lo, hi, nombres);

    Jugador[] jugadores = new Jugador[nombres.size()];
    for (int i = 0; i < nombres.size(); i++) {
        jugadores[i] = (Jugador) players.get(nombres.get(i));
    }
    return jugadores;
}
```

Listing 2: Métodos de la clase Scoreboard

2.3. Clase ConnectFour

Descripción:
La clase `ConnectFour` representa el tablero del juego y su lógica interna. Permite realizar movimientos y verificar si el juego ha terminado.

Atributos:

- `grid (char[7][6])`: Matriz de caracteres que representa el estado del tablero. Las celdas vacías contienen un espacio (`' '`), y las ocupadas contienen `'X'` o `'O'`.
- `currentSymbol (char)`: Representa el símbolo del jugador actual (`'X'` o `'O'`), que se alterna tras cada turno.

Métodos:

- `ConnectFour()`: Constructor que inicializa el tablero vacío y establece `'X'` como símbolo inicial.
- `makeMove(int z)`: Intenta insertar el símbolo actual en la columna `z`. Si la columna está llena o es inválida, retorna `false`; de lo contrario, actualiza el tablero y el símbolo, y retorna `true`.
- `isGameOver()`: Verifica si el juego ha terminado por victoria o empate. Debe retornar o registrar el estado final del juego según se defina.

```
public ConnectFour() {
    grid = new char[7][6];
    for (int fila = 0; fila < 7; fila++) {
        for (int columna = 0; columna < 6; columna++) {
            grid[fila][columna] = ' ';
        }
    }
    currentSymbol = 'X';
}

public boolean makeamove(int move) {
    if (move < 0 || move > 6 || grid[0][move] != ' ') {
        return false;
    }

    for (int fila = 6; fila >= 0; fila--) {
        if (grid[fila][move] == ' ') {
            grid[fila][move] = currentSymbol;
            lastfil = fila;
            lastCol = move;

            if (currentSymbol == 'X') {
                currentSymbol = 'O';
            } else {
                currentSymbol = 'X';
            }

            return true;
        }
    }

    return false;
}
```

Listing 3: Métodos de la clase ConnectFour

```java
public String isGameOver(int fila, int col) {
    char symbol = grid[fila][col];
    if (symbol == ' ') return "";

    // Horizontal
    String line = buildLine(fila, col, 0, -1) + symbol + buildLine(fila, col, 0, 1);
    if (line.contains(repeat(symbol, 4))) return String.valueOf(symbol);

    // Vertical
    line = symbol + buildLine(fila, col, 1, 0);
    if (line.contains(repeat(symbol, 4))) return String.valueOf(symbol);

    // Diagonal
    line = buildLine(fila, col, -1, -1) + symbol + buildLine(fila, col, 1, 1);
    if (line.contains(repeat(symbol, 4))) return String.valueOf(symbol);

    // Diagonal
    line = buildLine(fila, col, -1, 1) + symbol + buildLine(fila, col, 1, -1);
    if (line.contains(repeat(symbol, 4))) return String.valueOf(symbol);

    for (int c = 0; c < 6; c++) {
        if (grid[0][c] == ' ') {
            return ""; // a n hay espacios: no hay empate
        }
    }

    return "DRAW"; // si termin  el for, entonces est  lleno
}

private String buildLine(int fila, int col, int dFila, int dCol) {
    StringBuilder sb = new StringBuilder();
    int f = fila + dFila;
    int c = col + dCol;

    while (f >= 0 && f < 7 && c >= 0 && c < 6) {
        sb.append(grid[f][c]);
        f += dFila;
        c += dCol;
    }

    return sb.toString();
}

private String repeat(char c, int times) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < times; i++) sb.append(c);
    return sb.toString();
}
```

Listing 4: Métodos auxiliar isGameOver de la clase ConnectFour

2.4. Clase Game

Descripción:

La clase `Game` representa una partida específica de Conecta 4 entre dos jugadores. Encapsula el tablero, los nombres de los jugadores y el estado de la partida.

Atributos:

- `status (String)`: Estado actual de la partida: puede ser `IN-PROGRESS`, `VICTORY` o `DRAW`.
- `winnerPlayerName (String)`: Nombre del jugador ganador, o cadena vacía en caso de empate.
- `playerNameA (String)`: Nombre del primer jugador.
- `playerNameB (String)`: Nombre del segundo jugador.
- `connectFour (ConnectFour)`: Instancia de la clase `ConnectFour` que representa el tablero y la lógica del juego.

Métodos:

- `Game(String playerNameA, String playerNameB)`: Constructor que inicializa una partida entre los dos jugadores, con estado inicial `IN-PROGRESS`.
- `play()`: Ejecuta la partida mediante turnos alternados. Usa `makeMove` e `isGameOver` de `ConnectFour` para controlar el flujo del juego. Actualiza el estado final y registra el resultado.

```
public String play() {
    Scanner scanner = new Scanner(System.in);
    while (true) {
        printBoard();

        String currentPlayer = (connectFour.currentSymbol == 'X') ? playerNameA :
    playerNameB;
        System.out.print(currentPlayer + " (" + connectFour.currentSymbol + "), elige
     columna (0 a 5): ");

        int columna;
        try {
            columna = scanner.nextInt();
        } catch (Exception e) {
            System.out.println("Entrada inv lida. Intenta de nuevo.");
            scanner.nextLine(); // Limpiar buffer
            continue;
        }
        if (!connectFour.makeamove(columna)) {
            System.out.println("Movimiento inv lido. Intenta otra columna.");
            continue;
        }
        String resultado = connectFour.isGameOver(connectFour.lastfil, connectFour.
    lastCol);
        if (resultado.equals("X") || resultado.equals("O")) {
            printBoard();
            status = "VICTORY";
            winnerPlayerName = resultado.equals("X") ? playerNameA : playerNameB;
            System.out.println("  Victoria   de " + winnerPlayerName + "!");

            // Actualizar scoreboard
            String loser = (winnerPlayerName.equals(playerNameA)) ? playerNameB :
    playerNameA;
            scoreboard.addgameresult(winnerPlayerName, loser, false);
            return winnerPlayerName;
        } else if (resultado.equals("DRAW")) {
            printBoard();
            status = "DRAW";
            System.out.println("  Empate  !");
```

```
            // Actualizar scoreboard para empate
            scoreboard.addgameresult(playerNameA, playerNameB, true);
            return "";
        }
    }
}
```

Listing 5: Método play de la clase Game

```
private void printBoard() {
    System.out.println("\nTablero:");
    for (int fila = 0; fila < 7; fila++) {
        for (int col = 0; col < 6; col++) {
            System.out.print("| " + connectFour.grid[fila][col] + " ");
        }
        System.out.println("|");
    }
    System.out.println("  0   1   2   3   4   5\n");
}

// main para probar el juego
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Scoreboard scoreboard = new Scoreboard();
    scoreboard.cargarDatos(); // carga datos guardados al iniciar

    System.out.print("Nombre jugador A (X): ");
    String nameA = scanner.nextLine();

    System.out.print("Nombre jugador B (O): ");
    String nameB = scanner.nextLine();

    Game partida = new Game(nameA, nameB, scoreboard);
    partida.play();

    // Guardar datos al terminar
    scoreboard.guardarDatos();
}
```

Listing 6: Método printBoard y método main de la clase Game

2.5. Clase BST

Descripción:

La siguiente clase la añadimos nosotros para ordenar mas el código, a continuación su función:

La clase añadida `BST` implementa un árbol binario de búsqueda (Binary Search Tree) utilizado para almacenar y ordenar a los jugadores según su número de victorias. Cada nodo del árbol agrupa a los jugadores que tienen la misma cantidad de victorias, permitiendo realizar operaciones como inserción, eliminación, búsqueda de sucesores y consulta de rangos.

Estructura:

- `root (Node)`: Nodo raíz del árbol. Es el punto de entrada para todas las operaciones.

Clase interna Node:

- `key (int)`: Representa la cantidad de victorias.
- `jugadores (ArrayList<String>)`: Lista de nombres de jugadores que tienen la misma cantidad de victorias.
- `izq, der (Node)`: Punteros a los hijos izquierdo y derecho del nodo.

Funciones principales:

- `push(Node root, int key, String nombre)`: Inserta un jugador con `key` victorias en el árbol. Si ya existe un nodo con esa cantidad de victorias, se agrega a la lista de jugadores.
- `removeJugador(Node root, int key, String nombre)`: Elimina un jugador de su nodo correspondiente. Si no quedan más jugadores en el nodo, este se elimina completamente.
- `successor(Node root, int wins)`: Retorna el nodo con la menor cantidad de victorias mayor al valor entregado. Es útil para encontrar el sucesor en rendimiento.
- `winRange(Node root, int lo, int hi, ArrayList<String>resultado)`: Agrega a la lista `resultado` todos los jugadores que tengan entre `lo` y `hi` victorias.
- `preOrder, inOrder, postOrder(Node root)`: Métodos clásicos de recorrido del árbol, útiles para depuración o visualización.

Uso en el sistema:
Esta clase es utilizada dentro de la clase `Scoreboard` para:

- Mantener ordenados a los jugadores según sus victorias.
- Consultar rangos de rendimiento con eficiencia.
- Encontrar rápidamente al jugador con más victorias posterior a un valor dado.
- Insertar o eliminar jugadores del árbol conforme ganan o pierden.

Esta implementación personalizada permite manejar múltiples jugadores con la misma cantidad de victorias y realizar operaciones clave para el sistema de ranking del juego Conecta 4. a continuación el código de esta:

```
1  import java.io.*;
2  import java.util.*;
3
4  public class BST {
5      public Node root;
6
7      public static class Node {
8          int key; // cantidad de victorias
9          ArrayList<String> jugadores; // nombres de jugadores con esas victorias
10         Node izq, der;
11
12         public Node(int key) {
13             this.key = key;
14             this.jugadores = new ArrayList<>();
15             this.izq = null;
16             this.der = null;
17         }
18     }
19
20     static Node getSuccessor(Node curr) {
21         curr = curr.der;
```

```
        while (curr != null && curr.izq != null) {
            curr = curr.izq;
        }
        return curr;
    }

    public static Node successor(Node root, int wins) {
        Node succ = null;
        while (root != null) {
            if (wins < root.key) {
                succ = root;
                root = root.izq;
            } else {
                root = root.der;
            }
        }
        return succ;
    }

    static Node delNode(Node root, int x) {
        if (root == null) {
            return root;
        }
        if (root.key > x) {
            root.izq = delNode(root.izq, x);
        } else if (root.key < x) {
            root.der = delNode(root.der, x);
        } else {
            if (root.izq == null) return root.der;
            if (root.der == null) return root.izq;

            Node succ = getSuccessor(root);
            root.key = succ.key;
            root.jugadores = new ArrayList<>(succ.jugadores);
            root.der = delNode(root.der, succ.key);
        }
        return root;
    }

    public static Node removeJugador(Node root, int key, String nombre) {
        if (root == null) return null;

        if (key < root.key) {
            root.izq = removeJugador(root.izq, key, nombre);
        } else if (key > root.key) {
            root.der = removeJugador(root.der, key, nombre);
        } else {
            root.jugadores.remove(nombre);
            if (root.jugadores.isEmpty()) {
                return delNode(root, key);
            }
        }
        return root;
    }

    public static Node push(Node root, int key, String nombre) {
        if (root == null) {
            Node nuevo = new Node(key);
            nuevo.jugadores.add(nombre);
```

```
            return nuevo;
        } else if (key == root.key) {
            if (!root.jugadores.contains(nombre))
                root.jugadores.add(nombre);
        } else if (key < root.key) {
            root.izq = push(root.izq, key, nombre);
        } else {
            root.der = push(root.der, key, nombre);
        }
        return root;
    }

    public static void preOrder(Node root) {
        if (root != null) {
            System.out.print(root.key + " ");
            preOrder(root.izq);
            preOrder(root.der);
        }
    }

    public static void inOrder(Node root) {
        if (root != null) {
            inOrder(root.izq);
            System.out.print(root.key + " ");
            inOrder(root.der);
        }
    }

    public static void postOrder(Node root) {
        if (root != null) {
            postOrder(root.izq);
            postOrder(root.der);
            System.out.print(root.key + " ");
        }
    }
    public static void winRange(Node root, int lo, int hi, ArrayList<String>
    resultado) {
        if (root == null) return;

        if (root.key > lo) {
            winRange(root.izq, lo, hi, resultado);
        }

        if (root.key >= lo && root.key <= hi) {
            resultado.addAll(root.jugadores);
        }

        if (root.key < hi) {
            winRange(root.der, lo, hi, resultado);
        }
    }
}
```

Listing 7: Clase BST: Árbol binario de búsqueda para registrar jugadores por victorias

3. Análisis

3.1. Análisis asintótico

- **addGameResult**: Opera principalmente con inserciones y eliminaciones en el BST y consultas en la tabla hash. La complejidad promedio es $O(\log n)$, siendo n el número de nodos (niveles de victorias distintas) en el BST.
- **registerPlayer**: Inserta un jugador en la tabla hash con complejidad promedio $O(1)$.
- **checkPlayer**: Verifica la existencia de un jugador con búsqueda en la tabla hash, complejidad promedio $O(1)$.
- **winRange**: Realiza un recorrido del BST para devolver jugadores en un rango dado. El tiempo depende del tamaño del rango y es $O(k + \log n)$, con k jugadores en el rango.
- **winSuccessor**: Busca el sucesor en el BST con complejidad promedio $O(\log n)$.

3.2. Ventajas y Desventajas

Ventajas:

- **Claridad en la separación de responsabilidades:** La implementación está estructurada en clases bien definidas (`Player`, `Scoreboard`, `ConnectFour`, `Game`), lo que mejora la organización del código y facilita su comprensión y mantenimiento.
- **Uso de estructuras eficientes personalizadas:** La implementación del árbol binario de búsqueda (`BST`) y la tabla hash (`hash`) permite operaciones rápidas para registrar, buscar y organizar jugadores según sus victorias, lo cual es adecuado para manejar estadísticas y rankings.

Desventajas:

- **Escalabilidad limitada:** Aunque las estructuras personalizadas funcionan correctamente en este contexto académico, no están optimizadas para grandes volúmenes de datos ni utilizan balanceo automático (por ejemplo, un `AVL` o `TreeMap` como los que ofrece Java), lo cual podría afectar el rendimiento en sistemas con muchos jugadores.
- **Dependencia de implementaciones externas:** La implementación del árbol BST y la tabla hash no utiliza las colecciones estándar de Java, lo que obliga a los desarrolladores a mantener su propio código para funcionalidades que podrían estar mejor probadas y optimizadas en las librerías estándar.

3.3. Desafíos

Durante el desarrollo del laboratorio, uno de los principales desafíos se relacionó con la implementación del método `isGameOver`, encargado de determinar el fin de una partida. Inicialmente, se evaluaron alternativas que consideraban revisar la totalidad del tablero tras cada movimiento. Sin embargo, dicha estrategia resultó poco eficiente y propensa a errores. Posteriormente, se adoptó una solución más localizada y eficiente: registrar la última jugada realizada y verificar a partir de esa posición si se habían conectado cuatro fichas consecutivas en alguna dirección válida (horizontal, vertical o diagonal). Esta decisión simplificó la lógica del algoritmo y redujo el costo computacional de la verificación.

En cuanto a la estructura de tabla de dispersión utilizada para gestionar los jugadores, se presentaron dificultades al momento de comprender su uso práctico en la implementación. En una etapa temprana del desarrollo, no resultaba claro cómo almacenar y acceder correctamente a los objetos de tipo `Jugador`

mediante sus nombres como claves. Esta problemática fue resuelta tras consultar directamente al docente, lo que permitió entender el funcionamiento interno de la tabla y aplicarla con mayor precisión en el sistema.

Por otra parte, el diseño del árbol binario de búsqueda (BST) también presentó ciertas complejidades, particularmente en lo relativo a la incorporación de listas de jugadores dentro de cada nodo. Al comienzo, resultaba poco intuitiva la representación de múltiples jugadores con la misma cantidad de victorias mediante un arreglo dinámico asociado a una sola clave. Fue necesario ajustar la lógica de inserción y eliminación de jugadores en estos nodos para asegurar una gestión correcta del árbol. Una vez superado ese obstáculo, se logró una integración satisfactoria del BST al sistema, permitiendo la implementación de consultas eficientes por rango o por sucesor de victorias.

3.4. Extensión: Soporte para torneos de Conecta 4

Para extender el sistema y permitir la organización de torneos de Conecta 4 entre los jugadores, se propone una solución basada en la gestión de múltiples partidas organizadas por rondas, acumulación de puntajes y definición de un campeón. A continuación, se describen los principales requerimientos y modificaciones necesarias:

Requerimientos funcionales:

- Implementar una nueva clase `Tournament` encargada de gestionar el desarrollo del torneo completo.
- Permitir la inscripción de múltiples jugadores registrados en el `Scoreboard`.
- Generar emparejamientos automáticos por ronda (por ejemplo, tipo eliminación directa o todos contra todos).
- Llevar un registro de las partidas jugadas y los resultados obtenidos por cada jugador.
- Definir un sistema de puntaje (por ejemplo: 3 puntos por victoria, 1 por empate, 0 por derrota).
- Determinar el jugador campeón con base en los puntajes acumulados o mediante rondas eliminatorias.

Modificaciones al diseño:

- **Clase `Scoreboard`**: Se podría extender para incluir métodos que manejen el puntaje acumulado y los jugadores clasificados por torneo.
- **Clase `Player`**: Agregar atributos como `tournamentPoints` o `matchesPlayed` para almacenar información adicional temporal durante el torneo.
- **Nueva clase `Tournament`**: Encargada de:
  - Registrar participantes.
  - Organizar rondas y emparejamientos.
  - Ejecutar partidas mediante la clase `Game`.
  - Llevar el seguimiento del avance del torneo y declarar al campeón.
- Incorporar visualizaciones o resúmenes del torneo al finalizar, como tablas de posiciones o brackets eliminatorios.

Esta extensión permitiría llevar el sistema a un nivel más competitivo y dinámico, agregando funcionalidad para competencias organizadas. Además, ofrecería una aplicación más realista del sistema.

3.5. Revisión de estructura de datos

Reemplazo de BST:
Una estructura de datos estándar de Java que puede reemplazar al árbol binario de búsqueda (`BST`) es `TreeMap<Integer, ArrayList<String>`. Esta estructura mantiene los elementos ordenados por clave y permite acceso eficiente a rangos y sucesores.

Ventajas de TreeMap:

- Ordenamiento automático de las claves (cantidad de victorias) en orden ascendente.
- Métodos nativos como `higherKey()`, `subMap()`, y `firstEntry()` facilitan la implementación de funciones como `winSuccessor` y `winRange`.
- Código más conciso y mantenible al no requerir la implementación manual de nodos y recorrido del árbol.

Desventajas de TreeMap:

- Menor control sobre la lógica interna del árbol (no es posible personalizar el comportamiento del balanceo o la estructura del nodo).
- Al ser una estructura general, puede tener un uso de memoria y rendimiento ligeramente mayor si no se requiere todo su potencial.

Reemplazo de HashST:
La tabla de hash personalizada (`hash`) utilizada para registrar jugadores puede ser reemplazada por `HashMap<String, Player>`.

Ventajas de HashMap:

- Ofrece acceso y búsqueda en tiempo constante promedio (`O(1)`).
- Amplia documentación, soporte de la comunidad y compatibilidad con otras colecciones Java.
- Reduce significativamente la complejidad del código al delegar el manejo de colisiones y redimensionamiento al lenguaje.

Desventajas de HashMap:

- Requiere que la clase `Player` implemente correctamente los métodos `equals()` y `hashCode()` si se desean búsquedas avanzadas por objeto.
- Menor control didáctico: no permite al estudiante comprender el funcionamiento interno de una tabla de hash como lo haría al implementarla manualmente.

3.6. Pruebas

Con el objetivo de validar el correcto funcionamiento del sistema, se realizaron pruebas interactivas mediante la ejecución del programa principal. Durante las pruebas, se simularon partidas reales entre distintos jugadores, ingresando sus nombres y seleccionando columnas a través de la entrada estándar.

En la segunda prueba documentada, los jugadores `cristian` y `pablo` participaron en una partida. Cada jugador alternó sus movimientos de manera correcta y el sistema fue capaz de mostrar el estado

actualizado del tablero tras cada jugada. La partida finalizó con una victoria para `cristian`, tras lograr conectar cuatro fichas consecutivas. El sistema detectó correctamente la condición de victoria y finalizó la partida, desplegando el mensaje correspondiente.

Además, se verificó que el archivo `jugadores.txt` se actualizara correctamente, reflejando las estadísticas acumuladas de los jugadores registrados. El contenido del archivo tras las partidas fue el siguiente:

```
playedGames=2
pablo,0,0,1
anais,0,0,1
cristian,2,0,0
```

Esto confirma que el sistema logró registrar el número total de partidas jugadas y actualizar adecuadamente los conteos de victorias y derrotas de cada jugador.

En general, se observó un comportamiento estable y coherente del programa durante la ejecución, con una interacción clara y fluida entre los distintos componentes del sistema (tablero, jugadores, lógica de juego y sistema de almacenamiento de datos).

4. Conclusión

Durante el desarrollo de este laboratorio se reforzaron conceptos fundamentales sobre estructuras de datos y programación orientada a objetos, aplicados en un proyecto práctico y estructurado como el juego Conecta 4. La implementación de clases independientes para modelar jugadores, el tablero, el sistema de partidas y el control estadístico permitió desarrollar un sistema modular, reutilizable y extensible.

Uno de los principales aprendizajes fue el diseño y uso de estructuras de datos personalizadas. En particular, el árbol binario de búsqueda (`BST`) y la tabla de hash propia (`hash`) obligaron a comprender a fondo cómo se comportan internamente estas estructuras, cómo optimizar operaciones de inserción, búsqueda y eliminación, y cómo adaptarlas a las necesidades específicas del sistema.

Además, esta experiencia permitió identificar claramente las diferencias entre usar estructuras propias y utilizar las implementaciones estándar del lenguaje, como `TreeMap` o `HashMap`. Aunque estas últimas ofrecen mayor eficiencia y simplicidad de uso, las estructuras personalizadas permiten un control total sobre la lógica y pueden ser necesarias en contextos específicos.

Por ejemplo, en sistemas embebidos o aplicaciones críticas con restricciones de memoria o procesamiento, puede ser más eficiente implementar una estructura a medida que evite el overhead de una clase genérica. Otro caso concreto es en competiciones de programación o videojuegos en línea donde se necesite una lógica de ordenamiento o búsqueda que no esté contemplada directamente por las librerías estándar, como el almacenamiento de múltiples elementos por clave en orden específico y su manipulación directa.

En resumen, este laboratorio no solo fortaleció los conocimientos teóricos, sino que también entregó herramientas prácticas para diseñar, analizar y justificar la elección de estructuras de datos según el problema planteado. La posibilidad de extender el sistema a funcionalidades como torneos muestra que el diseño propuesto es flexible, escalable y aplicable en escenarios reales.