UNIVERSIDAD DIEGO PORTALES

Laboratorio 4: Árboles de búsqueda binarios

ESTRUCTURA DE DATOS Y ALGORITMOS: 2024-02

EZEQUIEL MORALES
PROFESORES: KAROL SUCHAN, YERKO ORTIZ, RODRIGO ALVAREZ.
AYUDANTE: DIEGO BANDA.

1. Introducción:

En el presente informe se detalla el desarrollo del juego TicTacToe, que permite registrar jugadores, gestionar partidas y realizar un seguimiento de estadísticas como victorias, empates y derrotas. El sistema emplea diversas estructuras de datos, como árboles binarios de búsqueda (BST) y tablas hash (HashST), para asegurar un manejo eficiente de los datos.

El uso de un HashST permite registrar jugadores de manera rápida y acceder a su información mediante búsquedas constantes, mientras que el BST organiza a los jugadores según su número de victorias, facilitando la consulta de rangos y sucesores.

Además, se han implementado funcionalidades para jugar partidas interactivas, determinar resultados con precisión, y visualizar las estadísticas actualizadas de los jugadores de forma dinámica. El desarrollo incluye un análisis del rendimiento de las estructuras utilizadas, la detección de desafíos en la lógica de juego y propuestas para extender el sistema, como el soporte para torneos. A lo largo del informe, se reflexiona sobre la capacidad del sistema para manejar diferentes escenarios y cómo se optimizó para garantizar su correcta operación.

udp

2. Implementación:

Para dar inicio a este programa se deben definir las estructuras de datos principales, como un BST (BinarySearchTree) y HashST, los cuales se tomó como referencia las implementaciones del texto guía.

Así mismo para la implementación del Árbol se crea la clase BST y la clase Node, donde sus argumentos son: root, key, val, left, right y size. Además de inicializar sus debidos constructores.

```java
class BST<Key extends Comparable<Key>, Value> {
     Node root;
    //Clase nodo
    class Node {
        Key key;
        Value val;
        Node left, right;
        int size;
    //Constructor de nodo
        public Node(Key key, Value val, int size) {
            this.key = key;
            this.val = val;
            this.size = size;
        }

    }
    //Constructor de BST
    public BST() {
    }
```

Listing 1: Clase, constructor y de BST-Node.

Luego se tienen que crear los métodos principales para la clase BST, los cuales son:

Métodos de búsqueda:

get(Key key)]: Busca un nodo en el árbol, recibe una key y retorna el valor asociado a la key.
inOrder(Node root): Realiza un recorrido en orden del árbol, imprimiendo las key y sus valores asociados en orden ascendente.

```java
//Metodos de busqueda.
  public Value get(Key key) {
      if (key == null) throw new IllegalArgumentException("La key no puede ser nula");
      return get(root, key);
  }

  private Value get(Node x, Key key) {
      if (key == null) throw new IllegalArgumentException("La key no puede ser nula");
      if (x == null) return null;
      int cmp = key.compareTo(x.key);
      if      (cmp < 0) return get(x.left, key);
      else if (cmp > 0) return get(x.right, key);
      else              return x.val;
  }

  public void inOrder(Node root){
      if (root == null){
          return;
      }

      inOrder(root.left);
      System.out.println("Victorias: " + root.key + " ; Nombre: " + root.val);
      inOrder(root.right);

  }
```

Listing 2: Métodos de búsqueda de BST.

Método de inserción:

put(Key key, Value val): Inserta un nuevo nodo en el árbol cumpliendo los parámetros de un BST.

```java
//Metodo de inserción.

    public void put(Key key, Value val) {
        if (key == null) throw new IllegalArgumentException("La key no puede ser nula");
        root = put(root, key, val);
    }

    private Node put(Node x, Key key, Value val) {
        if (x == null) return new Node(key, val, 1);
        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;
        x.size = 1 + size(x.left) + size(x.right);
        return x;
    }
```

Listing 3: Método de inserción de BST.

Métodos adicionales:

Estos métodos fueron de utilidad para la implementación de las clases principales del programa.

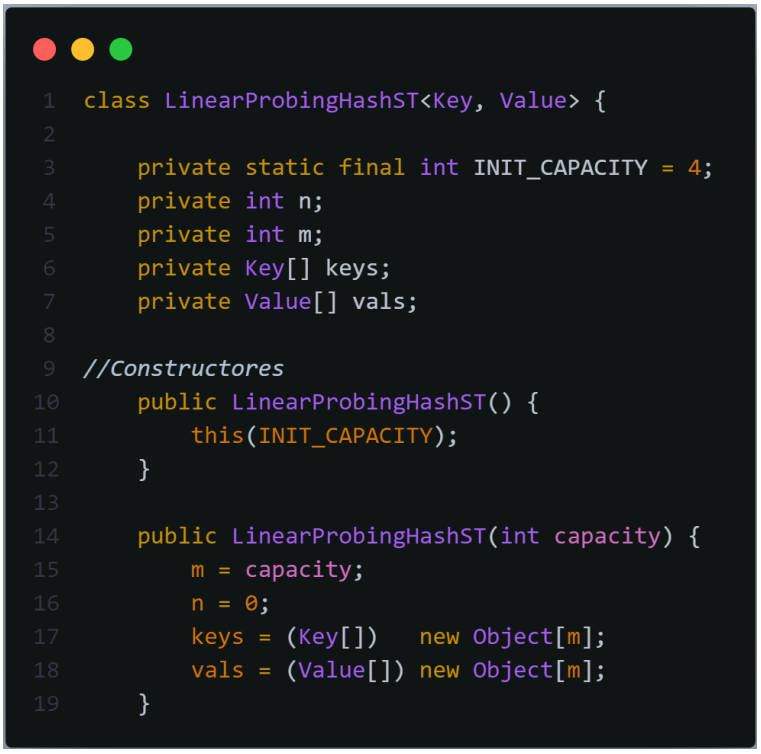
contains(Key key): Verifica si una key está presente en árbol.
size(): Retorna el número total de nodos en el árbol.

delete(Key key): Elimina un nodo con la key especificada, manteniendo las propiedades del BST.
findSuccesor(Node x): Encuentra el sucesor de un nodo, que es el nodo con la clave más pequeña en el subárbol derecho.

```java
public boolean contains(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to contains() is null");
        return get(key) != null;
    }

    public int size() {return size(root);}

    private int size(Node x) {
        if (x == null) return 0;
        else return x.size;
    }

    public void delete(Key key) {
        if (key == null) throw new IllegalArgumentException("calls delete() with a null key");
        root = delete(root, key);
    }

    private Node delete(Node x, Key key) {
        if (x == null) return null;

        int cmp = key.compareTo(x.key);
        if (cmp < 0) {
            x.left = delete(x.left, key);
        } else if (cmp > 0) {
            x.right = delete(x.right, key);
        } else {

            if (x.right == null) return x.left;
            if (x.left == null) return x.right;


            Node successor = findSuccessor(x.right);
            successor.right = delete(x.right, successor.key);
            successor.left = x.left;
            x = successor;
        }

        x.size = size(x.left) + size(x.right) + 1;
        return x;
    }

    private Node findSuccessor(Node x) {
        while (x.left != null) {
            x = x.left;
        }
        return x;
    }
```

Listing 4: Métodos adicionales de BST.

Una vez creada y estructurada la clase BST, se puede seguir con la implementación del HashST, con la clase LinearProbingHashST se hizo elección de este tipo de HashST debido a su simplicidad y eficiencia para manejar un programa con pocos datos (TicTacToe). La clase LinearProbingHashST tiene como argumentos: INIT_CAPACITY, n (número de key), m (size de la tabla), keys, values. Y se inicializa su debido constructor.



```java
class LinearProbingHashST<Key, Value> {

    private static final int INIT_CAPACITY = 4;
    private int n;
    private int m;
    private Key[] keys;
    private Value[] vals;

//Constructores
    public LinearProbingHashST() {
        this(INIT_CAPACITY);
    }

    public LinearProbingHashST(int capacity) {
        m = capacity;
        n = 0;
        keys = (Key[])   new Object[m];
        vals = (Value[]) new Object[m];
    }
```

Listing 5: Clase y constructor LinearProbingHashST.

Luego se crean los métodos principales de la clase LinearProbingHashST, los cuales son:

Método de búsqueda:

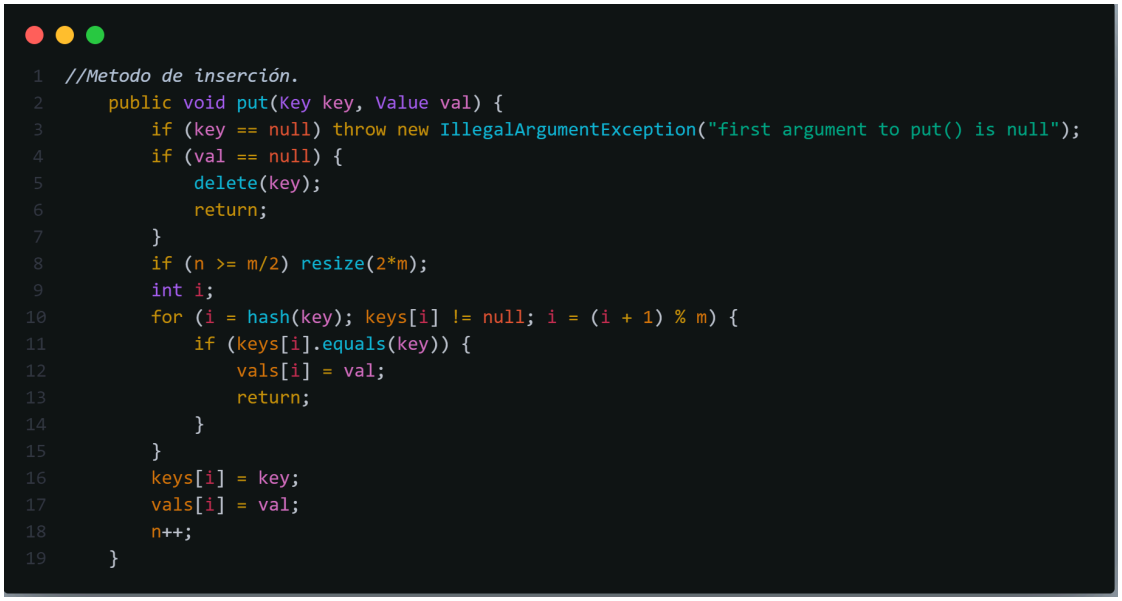
get(Key key): Busca un valor asociado a una key en la tabla hash y retorna el valor asociado.

```java
//Metodo de busqueda.
    public Value get(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to get() is null");
        for (int i = hash(key); keys[i] != null; i = (i + 1) % m)
            if (keys[i].equals(key))
                return vals[i];
        return null;
    }
```

Listing 6: Método de búsqueda de LinearProbingHashST.

Método de inserción:

put(Key key, Value val): Inserta una key y value en la tabla, si la clave existe actualiza el valor. Redimensiona la tabla para mantener la eficiencia. Utiliza hashing lineal para manejar colisiones.



```java
//Metodo de inserción.
    public void put(Key key, Value val) {
        if (key == null) throw new IllegalArgumentException("first argument to put() is null");
        if (val == null) {
            delete(key);
            return;
        }
        if (n >= m/2) resize(2*m);
        int i;
        for (i = hash(key); keys[i] != null; i = (i + 1) % m) {
            if (keys[i].equals(key)) {
                vals[i] = val;
                return;
            }
        }
        keys[i] = key;
        vals[i] = val;
        n++;
    }
```

Listing 7: Método de inserción de LinearProbingHashST.

Métodos adicionales:

Estos métodos fueron de utilidad para la implementación de las clases principales del programa.

size(): Retorna el número de elementos en la tabla hash.
isEmpty(): Verifica si la tabla hash está vacía.
contains(Key key): Verifica si una key está presente en la tabla hash.
hash(Key key): Calcula el valor hash de una clave utilizando su código hash
delete(Key key): Elimina una key y su valor asociado a la tabla hash.
resize(Key key): Redimensiona la tabla hash a una nueva capacidad especificada.
check(): Verifica la integridad interna de la tabla hash. Asegura que el tamaño sea apropiado y que los valores asociados a las claves estén correctos.

```java
public int size() {return n;}
    public boolean isEmpty() {return size() == 0;}

    public boolean contains(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to contains() is null");
        return get(key) != null;
    }

    private int hash(Key key) {
        int h = key.hashCode();
        h ^= (h >>> 20) ^ (h >>> 12) ^ (h >>> 7) ^ (h >>> 4);
        return h & (m-1);
    }

    public void delete(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to delete() is null");
        if (!contains(key)) return;
        int i = hash(key);
        while (!key.equals(keys[i])) {
            i = (i + 1) % m;
        }
        keys[i] = null;
        vals[i] = null;
        i = (i + 1) % m;
        while (keys[i] != null) {
            Key   keyToRehash = keys[i];
            Value valToRehash = vals[i];
            keys[i] = null;
            vals[i] = null;
            n--;
            put(keyToRehash, valToRehash);
            i = (i + 1) % m;
        }
        n--;
        if (n > 0 && n <= m/8) resize(m/2);
        assert check();
    }

    private void resize(int capacity) {
        LinearProbingHashST<Key, Value> temp = new LinearProbingHashST<Key, Value>(capacity);
        for (int i = 0; i < m; i++) {
            if (keys[i] != null) {
                temp.put(keys[i], vals[i]);
            }
        }
        keys = temp.keys;
        vals = temp.vals;
        m    = temp.m;
    }

    private boolean check() {
        if (m < 2*n) {
            System.err.println("Hash table size m = " + m + "; array size n = " + n);
            return false;
        }
        for (int i = 0; i < m; i++) {
            if (keys[i] == null) continue;
            else if (get(keys[i]) != vals[i]) {
                System.err.println("get[" + keys[i] + "] = " + get(keys[i]) + "; vals[i] = " + vals[i]);
                return false;
            }
        }
        return true;
    }
```

Listing 8: Métodos adicionales de LinearProbingHashST.

2.1. Clase Player.

Tras haber implementado el BST-HashST, ahora se puede comenzar a crear la clase Player, esta clase representa a un jugador, tiene como argumentos: playerName, wins, draws, losses. Además se inicializa su constructor.

```java
class Player{
    String playerName;
    int wins;
    int draws;
    int losses;
//Constructor de player
Player(String playerName){
    this.playerName = playerName;
    this.wins = 0;
    this.draws = 0;
    this.losses = 0;
}
```

Listing 9: Clase y constructor Player.

Luego se crean los métodos principales de la clase Player, los cuales son:

addWin(): Añade una victoria al jugador.
addDraw(): Añade un empate al jugador.
addLoss(): Añade una derrota al jugador.
winRate(): Retorna el porcentaje de victorias en decimales entre 0 y 1 del jugador.

Métodos adicionales:
getDraws(): Retorna el número de empates.
getLosses(): Retorna el número de derrotas.
getWins(): Retorna el número de victorias.

```java
//Metodos de player
void addWin(){this.wins++;}
void addDraw(){this.draws++;}
void addLoss(){this.losses++;}
double winRate(){
int totaldepartidas = (wins + draws + losses);
     if(totaldepartidas == 0) {
       return 0;
    }

double porcetajeVictorias = (double) wins / totaldepartidas;
return porcetajeVictorias;
}
//Metodos adicionales
public int getDraws(){return draws;}
public int getLosses(){return losses;}
public int getWins(){return wins;}
```

Listing 10: Métodos de Player.

2.2. Clase Scoreboard.

Para añadir un tablero en el juego, se debe crear la clase Scoreboard esta gestiona el registro de jugadores y sus resultados, esta clase tiene como argumentos: winTree (BST), players (HashST) y playedGames. Donde winTree tiene como key el número de victorias y value el nombre del jugador, players es una tabla de hash que lleva el registro de los jugadores. Así también se inicializa su constructor.

```java
class Scoreboard{
    BST<Integer,String> winTree; //key: numero de victorias, value: playername.
    LinearProbingHashST<String, Player> players; //registro de jugadores.
    int playedGames; //total de partidas en el sistema.
    //Contructor
    Scoreboard(){
        this.winTree = new BST<Integer, String>();
        this.players = new LinearProbingHashST<String, Player>();
        this.playedGames = 0;
    }
```

Listing 11: Clase y constructor Scoreboard.

Luego se crean los métodos principales de la clase Scoreboard, los cuales son:
addGameResult(String winnerPlayerName, String looserPlayerName, boolean draw): Registra el resultado de una partida según el caso, actualiza el árbol de victorias (winTree) y finalmente incrementa el total de partidas jugadas.
registerPlayer(String playerName): Registra un nuevo jugador, si antes no fue registrado.
checkPlayer(String playerName): Recibe un jugador y verifica si existe en el mapa.

```java
void addGameResult(String winnerPlayerName, String looserPlayerName, boolean draw) {
    Player ganador = players.get(winnerPlayerName);
    Player perdedor = players.get(looserPlayerName);
    if (draw == true) {
        ganador.addDraw();
        perdedor.addDraw();
    } else {
        int oldWins = ganador.getWins();
        ganador.addWin();
        int newWins = ganador.getWins();

        winTree.delete(oldWins);
        winTree.put(newWins, winnerPlayerName);
        perdedor.addLoss();
    }
    playedGames++;
}

void registerPlayer(String playerName){
Player nuevoPlayer = new Player(playerName);
if (players.contains(playerName)){
    System.out.println("Jugador ya registrado");
    return;
}
players.put(playerName, nuevoPlayer);
}

boolean checkPlayer(String playerName){
if (players.contains(playerName)){
    System.out.println("Jugador registrado");
    return true;
}
System.out.println("Jugador no encontrado");
return false;
}
```

Listing 12: Métodos de Scoreboard.

winrange(int lo, int hi): Devuelve un arreglo de jugadores cuyo número de victorias está dentro del rango [lo, hi]. Se utiliza un recorrido parcial de inorder. Mientras recorre el árbol, si encuentra un nodo cuya clave está dentro del rango, obtiene el jugador correspondiente del mapa y lo agrega a la lista . Si la key del nodo supera el valor hi, se detiene el recorrido, ya que los valores restantes no están en el rango. Finalmente, convierte la lista de jugadores en un arreglo y lo retorna.

```java
Player[] winRange(int lo, int hi) {
    ArrayList<Player> playersInRange = new ArrayList<>();
    ArrayList<BST<Integer, String>.Node> stack = new ArrayList<>();
    BST<Integer, String>.Node current = winTree.root;

    while (current != null || !stack.isEmpty()) {
        while (current != null) {
            stack.add(current);
            current = current.left;
        }
        current = stack.remove(stack.size() - 1);
        int key = current.key;
        if (key >= lo && key <= hi) {
            String playerName = current.val;
            Player player = players.get(playerName);
            if (player != null) {
                playersInRange.add(player);
            }
        }
        if (key > hi) break;
        current = current.right;
    }
    return playersInRange.toArray(new Player[0]);
}
```

Listing 13: Método de Scoreboard.

winSuccesor(int wins): Recorre el árbol buscando un nodo cuya key sea mayor que la ingresada (wins), Si lo encuentra lo guarda como posible sucesor y sigue recorriendo el árbol para buscar una clave más cercana, cuando valida el sucesor se agrega a una lista que se convierte en un arreglo que se retorna.

```java
Player[] winSuccessor(int wins) {
    BST<Integer, String>.Node current = winTree.root;
    BST<Integer, String>.Node successor = null;

    while (current != null) {
        if (wins < current.key) {
            successor = current;
            current = current.left;
        } else {
            current = current.right;
        }
    }
    if (successor != null) {
        ArrayList<Player> result = new ArrayList<>();
        String playerName = successor.val;
        Player player = players.get(playerName);
        if (player != null) {
            result.add(player);
        }
        return result.toArray(new Player[0]);
    }
    return new Player[0];
}
```

Listing 14: Método de Scoreboard.

2.3. Clase Game.

Para la creación de la clase Game, se tiene como argumentos: status, winnePlayerName, playerNameA, playerNameB, ticTacToe. Dónde status puede ser: IN_PROGRESS, VICTORY O DRAW, también se inicializa un constructor que empieza una nueva partida entre dos jugadores, crea una nueva instancia de ticTacToe y establece el estado inicial en *IN_PROGRESS*.

```
class Game{
String status; //status: IN_PROGRESS, VICTORY, DRAW.
String winnerPlayerName; //String vacio si es empate.
String playerNameA;
String playerNameB;
TicTacToe ticTacToe;

//Metodos de Game / constructor.

Game(String playerNameA, String playerNameB) {
    this.playerNameA = playerNameA;
    this.playerNameB = playerNameB;
    this.ticTacToe = new TicTacToe();
    this.status = "IN_PROGRESS";
    this.winnerPlayerName = "";
}
```

Listing 15: Clase y constructor Game.

Luego se debe crear el método principal de Game, para esta clase se modificó el método principal de la clase para que no retorne el nombre del jugador ganador al finalizar . En su lugar, todas las operaciones necesarias, incluyendo el registro del resultado y la impresión de estadísticas, se realizan dentro del método. Esto permite simplificar el flujo de ejecución al manejarlo como un void, facilitando su implementación y uso.

play(): ejecuta una partida de TicTacToe entre dos jugadores en un ciclo que continúa hasta que se declare un ganador o empate. Solicita los nombres de los jugadores, los registra en el scoreboard y prepara el tablero. Durante el juego, muestra el estado del tablero y solicita al jugador en turno que ingrese coordenadas para su movimiento. Usa makemove para validar los movimientos e IsGameOver para verificar si la partida terminó, actualizando el status a VICTORY o DRAW. Al finalizar, registra el resultado, muestra estadísticas como rangos de victorias y el árbol de victorias, y pregunta si desean jugar otra partida.

```java
void play() {
    Scanner scanner = new Scanner(System.in);
    Scoreboard scoreboard = new Scoreboard();

    System.out.println("Bienvenido a TicTacToe.");

    System.out.print("Ingrese el nombre del Jugador 1 (X): ");
    String player1 = scanner.nextLine();
    scoreboard.registerPlayer(player1);

    System.out.print("Ingrese el nombre del Jugador 2 (O): ");
    String player2 = scanner.nextLine();
    scoreboard.registerPlayer(player2);

    System.out.println("\nIniciando el juego entre " + player1 + " y " + player2 + ".");

    boolean jugarDeNuevo = true;
    while (jugarDeNuevo) {
        Game game = new Game(player1, player2);

        System.out.println("\nEl tablero está vacío y el jugador " + player1 + " comenzará (X).\n");

        while (game.status.equals("IN_PROGRESS")) {
            System.out.println("Estado actual del tablero:");
            game.ticTacToe.printBoard();
            String currentPlayer = game.ticTacToe.currentSymbol == 'X' ? player1 : player2;
            System.out.println("Turno de " + currentPlayer + " (" + game.ticTacToe.currentSymbol + ")");
            int x, y;
            while (true) {
                System.out.print("Ingrese la fila (1-3): ");
                x = scanner.nextInt();
                System.out.print("Ingrese la columna (1-3): ");
                y = scanner.nextInt();

                if (game.ticTacToe.makeMove(x - 1, y - 1)) {
                    break;
                } else {
                    System.out.println("Movimiento inválido. Inténtelo de nuevo.");
                }
            }
            if (game.ticTacToe.isGameOver()) {
                if (game.ticTacToe.checkWinner()) {
                    game.status = "VICTORY";
                    game.winnerPlayerName = currentPlayer;
                } else {
                    game.status = "DRAW";
                }
            }
        }
        System.out.println("\nEstado final del tablero:");
        game.ticTacToe.printBoard();

        if (game.status.equals("VICTORY")) {
            System.out.println("El ganador es: " + game.winnerPlayerName);
            scoreboard.addGameResult(game.winnerPlayerName, game.winnerPlayerName.equals(player1) ? player2 : player1, false);
            System.out.println("\nEstadísticas:");
            System.out.println("\nRango de victorias a buscar: ");
            System.out.print("Ingrese el rango mínimo de victorias: ");
            int minVictorias = scanner.nextInt();
            System.out.print("Ingrese el rango máximo de victorias: ");
            int maxVictorias = scanner.nextInt();
            scanner.nextLine(); // Limpiar buffer
            System.out.println("Jugadores con victorias entre " + minVictorias + " y " + maxVictorias + ":");
            Player[] playersInRange = scoreboard.winRange(minVictorias, maxVictorias);
            for (Player player : playersInRange) {
                System.out.println(player.playerName + " - Victorias: " + player.getWins());
            }
            System.out.println("\nBuscar sucesores: ");
            System.out.print("Ingrese la cantidad de victorias para buscar sucesores: ");
            int victorias = scanner.nextInt();
            Player[] successorPlayers = scoreboard.winSuccessor(victorias);
            for (Player player : successorPlayers) {
                System.out.println(player.playerName + " - Victorias: " + player.getWins());
            }
            scanner.nextLine(); // Limpiar buffer
            System.out.println("\nÁrbol de victorias (in-order):");
            scoreboard.winTree.inOrder(scoreboard.winTree.root);
        } else if (game.status.equals("DRAW")) {
            System.out.println("Es un empate.");
            scoreboard.addGameResult(player1, player2, true);
        }
        System.out.print("\n¿Quieres jugar de nuevo? (s/n): ");
        jugarDeNuevo = scanner.next().equalsIgnoreCase("s");
        scanner.nextLine(); // Limpiar buffer
    }
    System.out.println("\nGame over.");
    scanner.close();
}
```

Listing 16: Método de Game.

2.4. Clase TicTacToe.

Para implementar el tablero y la lógica del juego se debe crear la clase TicTacToe, la cual tiene como argumentos: grid, currentSymbol (X o 0). Donde grid es el tablero 3x3 del juego y se inicializa en el constructor de la clase.

```java
public class TicTacToe {
char[][] grid;
char currentSymbol;

TicTacToe() {
    this.grid = new char[3][3];
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            grid[i][j] = ' ';
        }
    }
    this.currentSymbol = 'X';
}

```

Listing 17: Clase y constructor TicTacToe.

Luego se crean los métodos principales de la clase TicTacToe, los cuales son:

makeMove(int x, int y): Realiza un movimiento en el tablero si (x,y) es válido coloca el símbolo del jugador actual y alterna al siguiente.
isGameOver(): Verifica si hay un ganador (líneas completas en filas, columnas o diagonales) o si el tablero está lleno. Retorna true si el juego terminó, sino false.

```java
boolean makeMove(int x, int y) {
    if (x < 0 || x >= 3 || y < 0 || y >= 3 || grid[x][y] != ' ') {
        return false;
    }
    grid[x][y] = currentSymbol;
    currentSymbol = (currentSymbol == 'X') ? 'O' : 'X';
    return true;
}

boolean isGameOver() {
    for (int i = 0; i < 3; i++) {
        if (grid[i][0] != ' ' && grid[i][0] == grid[i][1] && grid[i][1] == grid[i][2]) return true;
        if (grid[0][i] != ' ' && grid[0][i] == grid[1][i] && grid[1][i] == grid[2][i]) return true;
    }
    if (grid[0][0] != ' ' && grid[0][0] == grid[1][1] && grid[1][1] == grid[2][2]) return true;
    if (grid[0][2] != ' ' && grid[0][2] == grid[1][1] && grid[1][1] == grid[2][0]) return true;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (grid[i][j] == ' ') return false;
        }
    }
    return true;
}
```

Listing 18: Métodos de TicTacToe.

Métodos adicionales (Importantes para la lógica y funcionamiento):

`checkWinner()`: Verifica si hay un ganador revisando filas, columnas y diagonales. Retorna true si hay tres símbolos iguales no vacíos en línea, sino false. *Esta separación entre `IsGameOver` y `CheckWinner` asegura que cada método se enfoque en una tarea específica (tal se ve en la implementación de `play()`).*
`printBoard()`: Imprime el estado actual del tablero.

```java
boolean checkWinner() {

    for (int i = 0; i < 3; i++) {
        if (grid[i][0] != ' ' && grid[i][0] == grid[i][1] && grid[i][1] == grid[i][2]) return true;
        if (grid[0][i] != ' ' && grid[0][i] == grid[1][i] && grid[1][i] == grid[2][i]) return true;
    }
    if (grid[0][0] != ' ' && grid[0][0] == grid[1][1] && grid[1][1] == grid[2][2]) return true;
    if (grid[0][2] != ' ' && grid[0][2] == grid[1][1] && grid[1][1] == grid[2][0]) return true;


    return false;
}

void printBoard() {
    System.out.println();
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            System.out.print(grid[i][j] == ' ' ? "-" : grid[i][j]);
            if (j < 2) System.out.print(" | ");
        }
        System.out.println();
        if (i < 2) System.out.println("---------");
    }
    System.out.println();
}
```

Listing 19: Métodos adicionales de `TicTacToe`.

3. Análisis.

3.1. Tiempo de ejecución (Scoreboard).

addGameResult: Obtener los players del map es **O(1)** por la tabla de hash, luego al eliminar en el árbol ambas búsquedas son **O(log(n))**, el término con mayor complejidad es el árbol, por lo tanto es **O(log(n))**.

registerPlayer: El ingresar un nodo y verificar si existe una key en una tabla de hash es **O(1)**.

checkPlayer: El verificar si existe una key en una tabla de hash es **O(1)**

winRange: Al realizar un recorrido inorder en el arbol, visitando nodos en un rango [lo,hi] dependemos del número de nodos en este rango, por lo tanto es **O(k + log(n))** donde **k** es el número de nodos.

winSuccessor: El buscar un sucesor en un árbol es **O(log(n))**

Método de Scoreboard	Tiempo de ejecución: BigO
addGameResult	O(log(n))
registerPlayer	O(1)
checkPlayer	O(1)
winRange	O(k + log(n))
winSuccessor	O(log(n))

3.2. Mayores desafíos.

Método Play y IsGameOver: Uno de los principales retos fue asegurar que las partidas se registraran correctamente como empate cuando no había un ganador. Inicialmente, no se manejaba bien esta condición, ya que dependía de un flujo de ejecución que no verificaba el estado del tablero de forma precisa. Para solucionar esto, implementé el método checkWinner, que analiza todas las filas, columnas y diagonales para determinar si existe un ganador o si el tablero está lleno sin un ganador, clasificándolo como empate. Esta verificación adicional resultó clave para garantizar que los resultados fueran precisos y reflejaran correctamente el estado del juego.

Método winSuccesor: La lógica para encontrar el sucesor más cercano en el número de victorias resultó compleja debido a la naturaleza del BST. Era necesario identificar el nodo con la clave más pequeña mayor al valor dado, y al principio, la solución no funcionaba correctamente en árboles con ramas muy desbalanceadas. La clave fue implementar una búsqueda iterativa que aprovechara las propiedades del BST, almacenando temporalmente posibles sucesores en un nodo auxiliar. Esto aseguró que el método retornara siempre el jugador correcto, incluso en estructuras desbalanceadas.

3.3. Escalabilidad de TicTacToe: clase Tournament.

Para agregar soporte a torneos, el sistema debería gestionar múltiples rondas, clasificaciones más dinámicas, reglas para determinar el campeón, etc. Esto se puede implementar en una nueva clase llamada Tournament y que tenga atributos como playerScores, games, Leaderboard, etc. Así también con sus debidos métodos que hagan funcionar esta extensión para un torneo.

3.4. Reemplazo de BST y HashST.

Para usar otras estructura de datos en vez de BST y HashST, implementaría un TreeMap y un HashMap de la biblioteca estandar de Java, estos tienen la misma estructura en términos generales, pero con operaciones con un tiempo de ejecución garantizado gracias al balanceo (TreeMap) y manejo de colisiones (HashMap) automáticos además de no necesitar realizar modificaciones a la implementación. Las desventajas posibles pueden ser consumo adicional por el balanceo interno y un menor control sobre la gestión de colisiones.

3.5. Observaciones.

Tras haber jugado varias partidas con amigos en discord y revisar la estadísticas de cada uno, tras haber ganado el mismo número de victorias se sobreescribe el nuevo ganador, pero luego de haber ganado más de una victoria y diferentes de ellas, nuevamente se imprime perfectamente los jugadores. Esto es debido a que el BST se actualiza según una key única, por lo tanto si se repiten se sobreescribe más no borra el anterior jugador. Al usar la implementación del texto guía sin hacer modificaciones drásticas a la estructura esto ocurre más no es un gran problema para el buen funcionamiento del juego.

4. Conclusión:

En conclusión, el sistema de gestión para el juego Tic Tac Toe ha demostrado ser eficaz para registrar y administrar partidas, así como para realizar un seguimiento detallado de las estadísticas de los jugadores. Esto se logró mediante la implementación de estructuras de datos como árboles binarios de búsqueda (BST) y tablas hash (HashST), que permitieron un manejo eficiente de los datos, desde la rápida consulta y registro de jugadores hasta la organización de estadísticas según su rendimiento.

La integración de funcionalidades como la verificación de resultados, el cálculo dinámico de estadísticas y la posibilidad de buscar rangos y sucesores en las victorias asegura una experiencia interactiva robusta y precisa. Además, el análisis de los resultados de las partidas, junto con la optimización en la gestión de datos, contribuye al buen rendimiento y escalabilidad del sistema.

En general, el uso de estas estructuras de datos y métodos proporciona una base sólida para el sistema, abriendo la posibilidad de incorporar mejoras futuras, como la gestión de torneos o funcionalidades adicionales para enriquecer la experiencia de los jugadores.