DATASTRUCRE PROJE

NAME/SURNAME: HALA JABBAN

ST NO: 2121221350

DATA: 01/05/2025

TreasureHunterAdventure

Treasure Hunt Adventure is a Java Swing-based game that implements a treasure hunting board game using linked list data structure. The game features two levels of increasing difficulty, where players explore a map to find treasures while avoiding traps. This project demonstrates the practical application of data structures and objectoriented programming principles in game development .

The code defines the **Game Board** for the "Treasure Hunt Adventure" game.
It creates a path of 30 cells (Nodes), and assigns each cell a type (treasure chest, trap, move forward, move backward, or empty) based on the level (Level 1 or Level 2). In Level 2, it builds a **Doubly Linked List** to support backward movement, while Level 1 uses a **Singly Linked List**.

Main functions:

- **generateBoard()**: Creates the game board and distributes spot types.
- **movePlayerForward / movePlayerBack**: Moves the player forward or backward based on dice roll (with backward move allowed only in Level 2).
- **getCellValue()**: Returns the point value of a specific cell.
- **getNodeAtPosition()**: Returns the node at a specific position.

Spot type distribution by level

- **Level 1**:
  * Spots 1–5 → Treasures
  * Spots 6–10 → Traps
  * Spots 11–30 → Random (Treasure, Trap, or Empty)
- **Level 2**:
  * Spots 1–5 → Treasures
  * Spots 6–10 → Traps
  * Spots 11–15 → Move Forward

* Spots 16–20 → Move Backward
* Spots 21–30 → Random (Treasure, Trap, Move Forward, Move Backward, or Empty)

The code defines a **Node** in the game. Each node represents a **cell in the game board** and contains:

- **position**: The cell number.
- **type**: The type of the cell, such as "Treasure", "Trap", or "Empty".
- **next**: The reference to the next node in the list.
- **prev**: The reference to the previous node (used only in Level 2).

Main functions:

- **Constructor**: Initializes the node, setting its position and type (empty by default).
- **Getters and setters**: Functions to get and set position, type, next node, and previous node.

The code defines a **Player class** for the game. Each player has:

- **name**: The player's name.
- **position**: The player's current position on the game board.
- **score**: The points the player has accumulated.
- **currentLevel**: The level the player is currently playing.

Main functions:

- **Constructor**: Initializes the player with a name, starting position (1), score (0), and level (1).
- **Getters**: To retrieve the player's name, position, score, and level.
- **move()**: Moves the player forward by a specified number of steps.
- **addScore()**: Adds points to the player's score.
- **moveBack()**: Moves the player backward while ensuring the position stays within the range (1 to 30).

The code defines a **ScoreNode class** for a binary search tree (BST) of player scores. Each node in the tree represents a **score entry** for a player and contains:

- **score**: The score the player achieved.
- **username**: The player's name.
- **level**: The level the player reached.

- **left**: The reference to the left child node in the binary tree.
- **right**: The reference to the right child node in the binary tree.

Main function:

- **Constructor**: Initializes the node with score, username, and level, while setting both left and right references to null because the node is not yet connected to any other nodes.

The code defines a **Scoreboard class** for the game, which uses a **Binary Search Tree (BST)** to store player scores. This class performs several key functions:

- **addScore**: Adds a new score to the tree and saves it to the scores.txt file.
- **insert**: Inserts a score into the tree based on the score value (if the score is smaller than the current node, it goes to the left, otherwise to the right).
- **getScoresForUser**: Retrieves all scores for a given username by traversing the tree (using in-order traversal).
- **getBestScore**: Retrieves the best score for a player based on the highest score.
- **getWorstScore**: Retrieves the worst score for a player based on the lowest score.
- **saveToFile**: Saves each new score to the scores.txt file.
- **loadFromFile**: Loads all saved scores from the file into the tree when the game starts.

The code defines an **enum SpotType**, which defines the different types of spots on the game board, each with a specific effect:

- **TREASURE_CHEST**: Grants +10 points.
- **TRAP**: Deducts -5 points.
- **MOVE_FORWARD**: No point change but allows the player to move forward.
- **MOVE_BACKWARD**: No point change but allows the player to move backward.
- **EMPTY**: No effect on the game.

Each spot type has an associated **point value** that influences the game.

- **Constructor**: Sets the points for each spot type.
- **getPoints()**: Returns the point value associated with each spot type.

The code defines the **BoardPanel class**, which renders the game board graphically on the user interface using **Swing** in Java. This class represents the **game board** visually, displaying the **cells** that represent different types of spots (Treasure Chest, Trap, Move Forward, Move Backward, or Empty). The main functions of the class are:

1. **Constructor**:
    - Receives the **GameBoard** and **player position** and initializes properties like background color and preferred panel size.
2. **setPlayerPosition**: ○ Updates the player's position on the board and triggers a repaint of the panel.
3. **paintComponent**:
    - Renders the board each time it updates (e.g., when the player moves).
    - Draws each cell based on its type (Treasure, Trap, etc.) with color gradients and icons.
    - Draws glowing effects around the player when they are on a cell.

4. **getSymbolForType**:
    - Returns the appropriate symbol (like emojis) for each spot type (e.g., for Treasure, for Trap).

This class is used to provide a rich graphical experience for the player as they move across the board in the game.

This code defines a GamePanel class that represents the game interface for a treasure hunt game. It uses Java Swing to create a graphical user interface (GUI) where a player can play the game. Here's a summary of its functionality:

Key Components:

1. **Player and Game Board**:
    - player: Represents the user playing the game. ○ gameBoard: Represents the game board with cells that the player moves across.
2. **UI Components**:
    - JLabels (playerNameLabel, scoreLabel, levelLabel) display player information, score, and current level.
    - JButton (rollButton) allows the player to roll the dice.
    - BoardPanel displays the game board with the player's current position.
3. **Game Flow**:
    - **Rolling the Dice**: When the player clicks the "Roll Dice" button, a dice roll is simulated, and the player's position on the board is updated based on the dice result.
    - **Score and Level Management**: The player's score is updated based on the spot they land on. If the player finishes the level, they are prompted to continue to the next level or return to the main menu.
    - **End of Game**: If the player reaches the end of the board or the score threshold, the game ends, and their score is saved.
4. **Level Progression**:
    - **Level 1**: The game starts at level 1, and the player moves based on the dice roll. Once level 1 is completed, the player is given the option to continue to level 2.
    - **Level 2**: If the player progresses to level 2, the board and the player's position are reset, and the level label is updated.
5. **Saving Scores**: The player's score is saved to a file (score.txt) once the game ends.
6. **UI Styling**: The game uses customized styles for labels and buttons to match a dark theme with white text.

Game Actions:

- **Roll Dice**: Simulates a dice roll, moves the player accordingly, updates the score, and checks if the player has completed the current level.
- **Level Complete**: If the player finishes level 1, they are prompted to continue to level 2.
- **Game Over**: If the player reaches the final position or meets the score requirement, the game ends, and the final score is displayed.

Summary:

The GamePanel class is responsible for managing the player's interaction with the game board, handling dice rolls, updating scores, and progressing through levels. It also handles UI updates and game-over conditions

.

The code MainMenupanel mrepresents the main user interface for the **Treasure Hunt Adventure** game, with the following buttons:

- **Start Game**: To start the game after entering a username.
- **Scoreboard**: To view the scoreboard after entering a username.
- **Exit**: To exit the application.

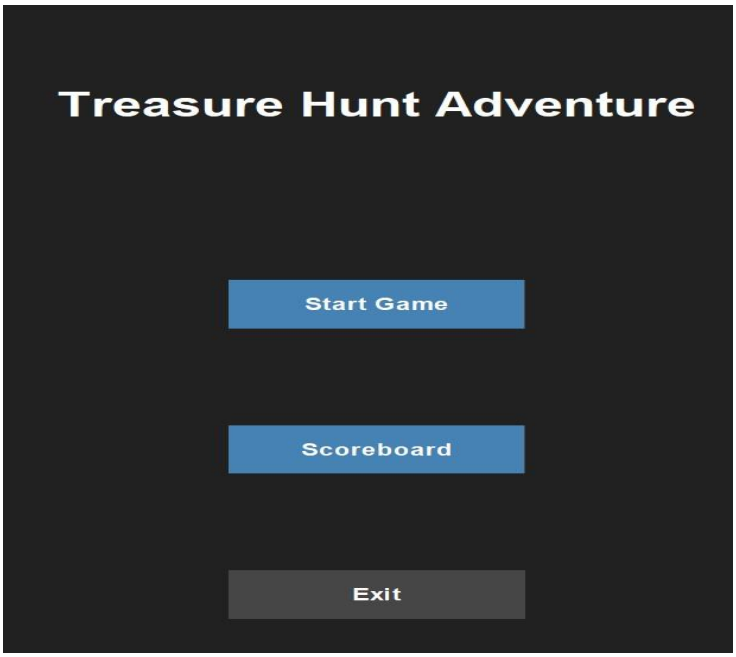
GridBagLayout is used to organize the elements in the user interface, with button styles that change colors when the mouse hovers over them. Event listeners are used to handle button clicks for actions such as starting the game, viewing the scoreboard, or exiting the application.

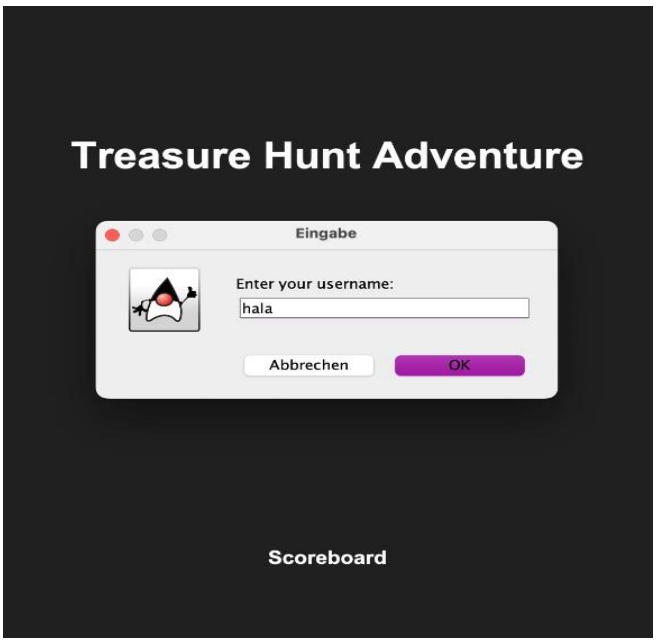
The code scoreboardpanel When displaying scores, it shows the best and worst scores of the player by level, along with a table listing all the scores the player has achieved. The code includes:

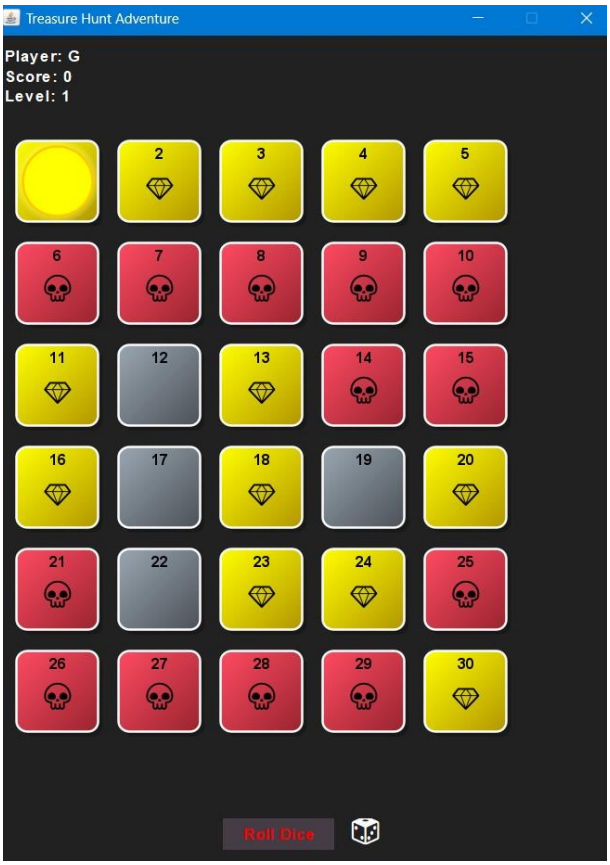
- **Title** at the top: Displays "High Scores".
- **Scores Panel**: Shows the best and worst scores along with level details.
- **Scores Table**: Displays the results for all levels the player has played.
- **Back Button**: Allows the user to return to the main menu.

All these components are added to the user interface using **BorderLayout**, with custom styles for colors and fonts.

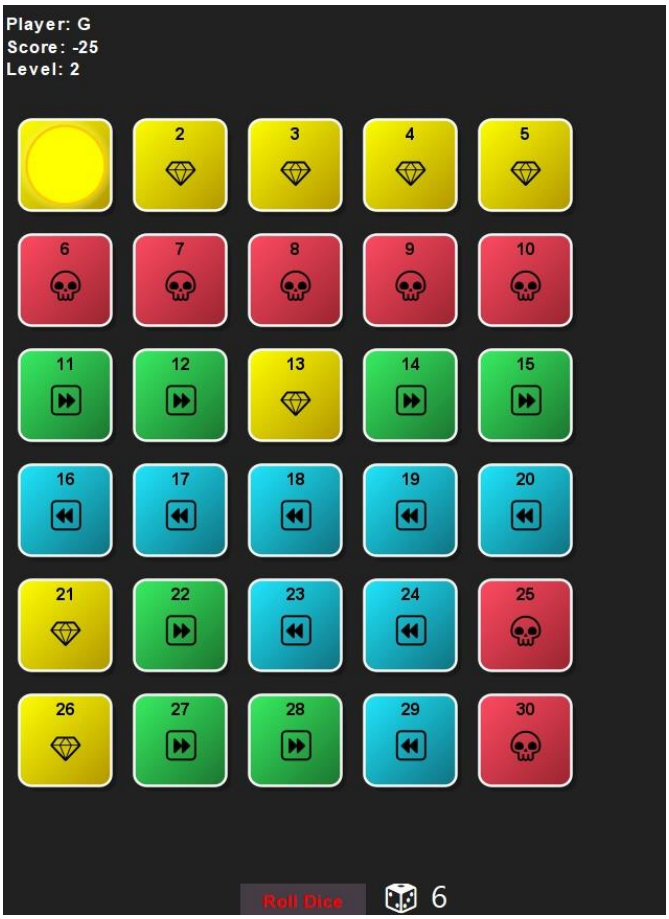
SAME PITCURE :

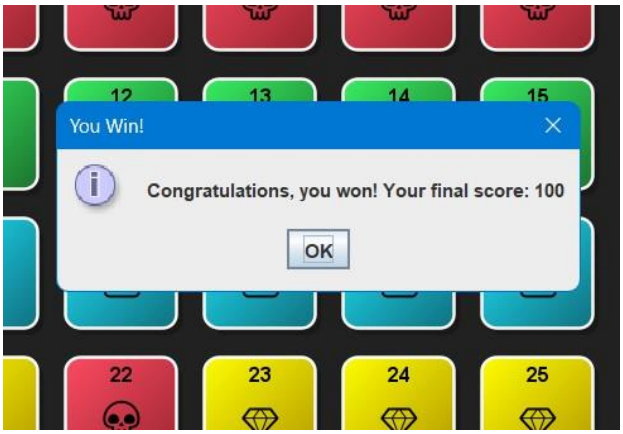

Treasure Hunt Adventure
Start Game
Scoreboard
Exit

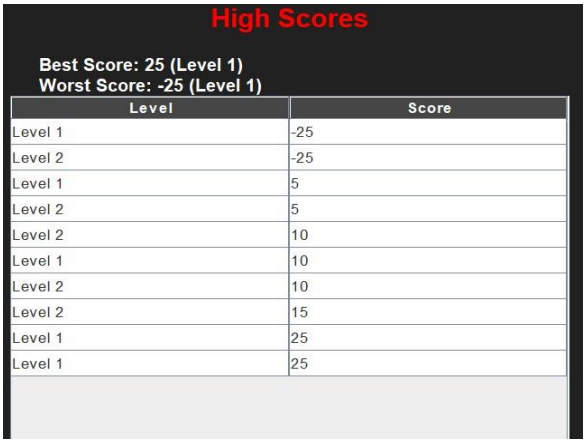

Treasure Hunt Adventure
Eingabe
Enter your username:
hala
Abbrechen
OK
Scoreboard

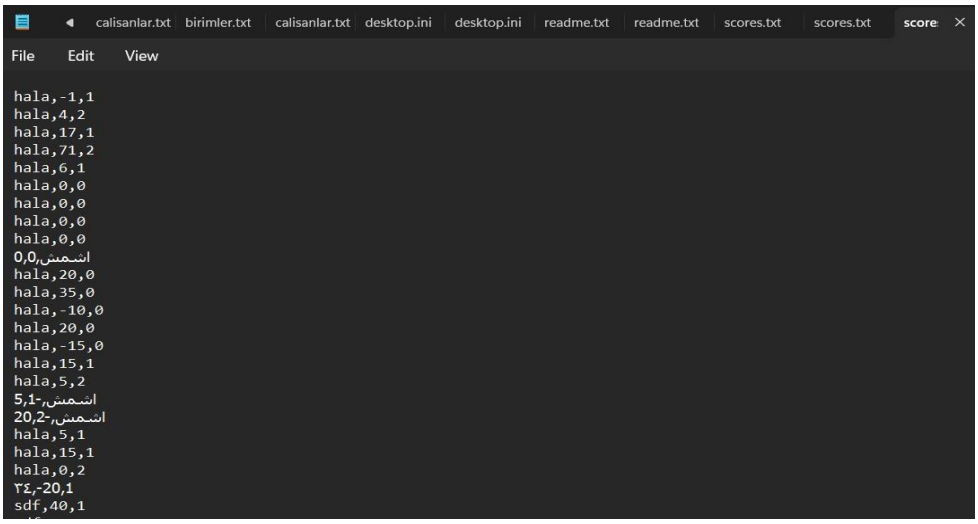

Treasure Hunt Adventure
Player: G
Score: 0
Level: 1
Roll Dice


Level 1 Complete
Level 1 complete! Score: -25
Continue to Level 2?
Yes
No









Data Structures Used

1.Linked List

Used for the game board implementation Each node represents a spot on the board Enables efficient traversal and spot management

2. Binary Search Tree Used

for score management Enables efficient score insertion and retrieval Maintains sorted order of scores

3. HashSet Used

for tracking visited spots Ensures O(1) lookup time for visited spots Prevents duplicate spot visits

Technical Details

Data Structure Implementation

The game uses several data structures to manage different aspects of the gameplay:

1.Linked List for Game Board

Efficient spot traversal Dynamic spot management Easy spot type assignment

2.Binary Search Tree for Scores

O(log n) insertion and retrieval Automatic score sorting Efficient score range queries

3. HashSet for Visited Spots

O(1) spot lookup Memory-efficient tracking Prevents duplicate visits File Management Scores are saved to scores.txt Automatic file creation if not exists Error handling for file operations Data persistence between sessions

Conclusion:

The **Treasure Hunt Adventure** project aims to develop an educational and entertaining game that combines fun and challenge through the use of data structure concepts. The project successfully implements linked lists and binary search trees effectively, along with building a user-friendly graphical interface using Java Swing. The game offers a clear scoring system and two levels of difficulty, providing players with a unique and motivating experience. This project serves as a practical example of how algorithms and data structures can be integrated into interactive applications.