**`**

**Data Structures – Project**

**Presented By**

Table of Contents

[Classes Documentation: 3](#_Toc152706719)

[**Class: NodeForVector** 3](#_Toc152706720)

[Class: Vector 3](#_Toc152706721)

[Class: NodeForQueue 4](#_Toc152706722)

[Class: Queue 4](#_Toc152706723)

[Class: NodeForStack 5](#_Toc152706724)

[Class: Stack 5](#_Toc152706725)

[Class: Maze 5](#_Toc152706726)

[Class: NodeForGraph 6](#_Toc152706727)

[Class: Graph 6](#_Toc152706728)

[Class: Player 7](#_Toc152706729)

[Class: NodeForBinarySearchTree 7](#_Toc152706730)

[Class: BinarySearchTree 7](#_Toc152706731)

[Class: File 8](#_Toc152706732)

[Class: NodeForGraphSolve 8](#_Toc152706733)

[Class: AdjacentMatrix 9](#_Toc152706734)

[Class: GraphSolve 9](#_Toc152706735)

[Class Diagram 10](#_Toc152706736)

Classes Documentation:

## **Class: NodeForVector**

**Description:** This is a template class representing a node in a Vector, which is essentially a linked list. It stores a value and a pointer to the next node.

**Public Methods:**

* NodeForVector(): Constructor initializing the next pointer to NULL.

## Class: Vector

**Description:** A template class implementing a dynamic array-like data structure using a linked list. It provides functionalities to manipulate a list of elements.

**Public Methods:**

* Vector(): Constructor initializing the Vector.
* push(T value): Adds a new element at the end of the Vector.
* pop(): Removes the last element from the Vector.
* insert(int index, T value): Inserts an element at a specified index.
* remove(int index): Removes an element at a specified index.
* clear(): Clears all elements from the Vector.
* print(): Displays the elements of the Vector.
* isEmpty(): Checks if the Vector is empty.
* getSize(): Returns the number of elements in the Vector.
* setSize(int s): Sets the size of the Vector.
* \*getHead() / setHead(NodeForVector<T> h)\*\*: Get/Set the head of the Vector.
* \*getTail() / setTail(NodeForVector<T> t)\*\*: Get/Set the tail of the Vector.
* ~Vector(): Destructor for the Vector.

## Class: NodeForQueue

**Description:** A template class representing a node in a Queue, similar to NodeForVector, used for implementing a queue with a linked list.

**Public Methods:**

* NodeForQueue(): Constructor initializing the next pointer to NULL.

## Class: Queue

**Description:** A template class for a Queue data structure implemented using a linked list. It allows for standard queue operations.

**Public Methods:**

* Queue(): Constructor initializing the Queue.
* enqueue(T value): Adds an element to the end of the Queue.
* dequeue(): Removes the element from the front of the Queue.
* clear(): Clears the Queue of all elements.
* print(): Displays the elements of the Queue.
* isEmpty(): Checks if the Queue is empty.
* find(T value): Searches for an element in the Queue.
* getSize(): Returns the number of elements in the Queue.
* setSize(int s): Sets the size of the Queue.
* \*getHead() / setHead(NodeForQueue<T> h)\*\*: Get/Set the head of the Queue.
* \*getTail() / setTail(NodeForQueue<T> t)\*\*: Get/Set the tail of the Queue.
* ~Queue(): Destructor for the Queue.

## Class: NodeForStack

**Description:** This template class represents a node in a Stack. It stores a value and a pointer to the next node, forming a linked list structure.

**Public Methods:**

* NodeForStack(): Constructor that initializes the next pointer to NULL.

## Class: Stack

**Description:** A template class that implements a Stack data structure using a linked list. It supports typical stack operations and is used for storing and managing data in a last-in, first-out manner.

**Public Methods:**

* Stack(): Constructor that initializes the Stack.
* push(T value): Adds an element to the top of the Stack.
* pop(): Removes the top element from the Stack.
* clear(): Clears all elements from the Stack.
* print(): Displays the elements of the Stack.
* isEmpty(): Checks if the Stack is empty.
* getSize(): Returns the number of elements in the Stack.
* setSize(int s): Sets the size of the Stack.
* \*getHead() / setHead(NodeForStack<T> h)\*\*: Get/Set the head of the Stack.
* ~Stack(): Destructor for the Stack.

## Class: Maze

**Description:** This class represents a maze structure, providing functionalities to create and manage mazes of varying difficulties. It includes methods for maze generation, printing, and management of maze attributes.

**Public Methods:**

* Maze(): Constructor that initializes the Maze.
* createEasyMaze(): Generates an easy maze layout.
* createMediumMaze(): Generates a medium difficulty maze layout.
* createHardMaze(): Generates a hard maze layout.
* printMaze(): Prints the current maze layout.
* getRows() / setRows(int r): Get/Set the number of rows in the Maze.
* getColumns() / setColumns(int c): Get/Set the number of columns in the Maze.
* deleteMaze(): Deletes the current maze, freeing memory.
* getMaze() / setMaze(int m)\*\*: Get/Set the maze data structure.
* getMode() / setMode(char m): Get/Set the mode of the Maze.
* getObstacles() / setObstacles(int o): Get/Set the number of obstacles in the Maze.
* getPowerUps() / setPowerUps(int p): Get/Set the number of power-ups in the Maze.
* getCoins() / setCoins(int c): Get/Set the number of coins in the Maze.
* ~Maze(): Destructor for the Maze.

## Class: NodeForGraph

**Description:** Represents a node in a Graph. It includes attributes to mark various properties such as paths, walls, power-ups, obstacles, coins, and specific start and end points, along with its position and connections to adjacent nodes.

**Public Methods:**

* NodeForGraph(): Constructor initializing the node's attributes.
* ~NodeForGraph(): Destructor for the NodeForGraph.

## Class: Graph

**Description:** This class represents a graph data structure, tailored for the game's maze mechanics. It includes functionalities for creating and managing the graph based on the maze, placing obstacles, power-ups, coins, and handling player movements.

**Public Methods:**

* Graph(): Constructor that initializes the Graph.
* ~Graph(): Destructor for the Graph.
* createGraphFromMaze(Maze mazeObj): Generates a graph from a given Maze object.
* deletePreviousGraphAndCopyNew(Maze mazeObj): Deletes the current graph and copies a new one from a Maze object.
* placeRandomObstacles(): Randomly places obstacles in the graph.
* placeRandomPowerUps(): Randomly places power-ups in the graph.
* placeRandomCoins(): Randomly places coins in the graph.
* moveUp() / moveDown() / moveLeft() / moveRight(): Moves the player in the specified direction, if possible.
* checkIfCompleted(): Checks if the game is completed.
* playGame(): Main method to play the game, handling game logic.
* printGraph(): Prints the current state of the graph.
* Getters and Setters for various attributes: Such as total nodes, array of nodes, dimensions (rows and columns), mode, completion status, obstacles, power-ups, coins, queues for obstacles and power-ups, stack for coins, score, and lives.

## Class: Player

**Description:** This class represents a player in the game. It stores the player's name, date, time of play, score, and their current position in the maze.

**Public Methods:**

* Player(): Constructor that initializes a Player object.
* updateDateAndTime(): Updates the date and time for the player.
* ~Player(): Destructor for the Player class.
* Getters and Setters: Methods to get and set the player's name, date, time, score, and current maze.

## Class: NodeForBinarySearchTree

**Description:** A node class for a binary search tree (BST), containing a Player object and pointers to left and right child nodes. This structure is used to organize players, typically by their scores or other criteria.

**Public Methods:**

* NodeForBinarySearchTree(): Constructor initializing the node with NULL for left and right pointers.
* ~NodeForBinarySearchTree(): Destructor for the NodeForBinarySearchTree.

## Class: BinarySearchTree

**Description:** Implements a binary search tree data structure, which is used to store and manage Player objects, likely for organizing and retrieving player data such as high scores.

**Public Methods:**

* BinarySearchTree(): Constructor that initializes the BST.
* ~BinarySearchTree(): Destructor for the BinarySearchTree.
* insert(Player player): Inserts a new player into the BST.
* \*printInOrder(NodeForBinarySearchTree node)\*\*: Prints the BST in an in-order traversal.
* \*printPreOrder(NodeForBinarySearchTree node)\*\*: Prints the BST in a pre-order traversal.
* \*printPostOrder(NodeForBinarySearchTree node)\*\*: Prints the BST in a post-order traversal.
* \*displayHighestScoreDescending(NodeForBinarySearchTree node)\*\*: Displays players in descending order of their scores.
* writeToFileForLeaderBoard(BinarySearchTree tree): Writes the leaderboard information to a file.
* readFromFileForLeaderBoard(BinarySearchTree& tree): Reads leaderboard information from a file.
* Getters and Setters: Methods to get and set the root of the BST.

## Class: File

**Description:** This class handles file operations related to the game. It provides functionalities for reading and writing game data such as leaderboard information and game states, facilitating game resume and score tracking.

**Public Methods:**

* File(): Constructor for the File class.
* ~File(): Destructor for the File class.
* writeToFileForLeaderBoard(BinarySearchTree tree): Writes leaderboard information to a file.
* readFromFileForLeaderBoard(BinarySearchTree& tree): Reads leaderboard information from a file.
* writeToFileForResumeGame(Player player, Graph graph): Writes the current game state to a file for resuming later.
* readFromFileForResumeGame(Player& playerObj, Graph& graphObj, Maze& mazeObj): Reads the game state from a file to resume a previously saved game.

## Class: NodeForGraphSolve

**Description:** Represents a node in the GraphSolve data structure. It contains properties such as path, wall, power-up, obstacle, coin, and others, relevant to the game's maze-solving aspect.

**Public Methods:**

* NodeForGraphSolve(...): Constructor initializing the node with various attributes including position, value, and weight.

## Class: AdjacentMatrix

**Description:** A helper class used in the GraphSolve class, representing a node in an adjacency matrix.

**Public Attributes:**

* head: A pointer to the head NodeForGraphSolve.

## Class: GraphSolve

**Description:** Represents a graph data structure specifically designed for solving the maze in the game. It includes methods for creating the graph from the maze, placing game elements, and implementing maze-solving algorithms.

**Public Methods:**

* GraphSolve(): Constructor that initializes the GraphSolve.
* ~GraphSolve(): Destructor for the GraphSolve.
* createGraph(Maze mazeObj): Creates a graph from a Maze object.
* placeRandomObstacles() / placeRandomPowerUps() / placeRandomCoins(): Places obstacles, power-ups, and coins randomly in the graph.
* createNode(int vertex, int input\_weight): Creates a new node in the graph.
* printGraph(): Prints the current graph.
* addEdge(int source, int destination, int weight): Adds an edge between two nodes in the graph.
* createAdjacentMatrix(): Creates an adjacency matrix for the graph.
* printEdges(): Prints the edges of the graph.
* dijkstra(int source, int destination): Implements Dijkstra's algorithm for pathfinding in the graph.
* Getters and Setters for various attributes: Such as rows, columns, maze object, etc.

# Class Diagram

