# Project

**Data Structures**

**Fall 23**

**Submitted by:**

Gul Sher Khan Swati 22I-2637

Ahsan Faraz 22I-8791

**Submitted To:**

Maam Maheen Arshad

# Classes:

1. **Qnode:** template queue node
2. **Queue:** template queue class
3. **ListNode:** template linked list node
4. **List:** template linked list
5. **Vector:** template vector
6. **Node:** Graph node for pointers to adjacent nodes and key to store graph values
7. **Collectible (structure):** in game collectibles, like coins and trophies
8. **Player (structure):** main player with collectible list and x, y coordinates
9. **Maze:** main maze for the game with a x by x size
10. **Obstacle (structure):** in game obstacles
11. **Game:** main game class with a maze, maze size and queue of obstacles
12. **Score (structure):** score with value for name, score, coins and trophies
13. **treeNode:** node for BST of scores
14. **tree:** Binary search tree to sort the player scores

# Queue:

**Class Members:**

* **Qnode Structure:**
  + **data**: Template parameter representing the type of data stored in the node.
  + **next**: Pointer to the next node in the queue.
* **Queue Class:**
  + **front**: Pointer to the front (head) of the queue.
  + **rear**: Pointer to the rear (tail) of the queue.

**Member Functions:**

* **Queue Constructor:**
  + Initializes an empty queue by setting both the front and rear pointers to **NULL**.
* **isEmpty Function:**
  + Checks if the queue is empty.
  + Returns **true** if the queue is empty, **false** otherwise.
* **enqueue Function:**
  + Adds an element to the rear of the queue.
* **dequeue Function:**
  + Returns the element at the front of the queue and removes it.
  + Prints an error message and returns a default-initialized value if the queue is empty.
* **display Function:**
  + Displays the elements of the queue from front to rear.

# Vector

**Class Members**

* **data**: Pointer to the dynamic array storing elements.
* **size**: Current size of the vector.

**Member Functions**

**Constructors and Destructor**

* **vector(int size = 0)**: Default constructor. Creates an empty vector or a vector of a specified size.
* **vector(int size, const T &initial\_value)**: Constructor initializing the vector with a specified size and initial value.
* **~vector()**: Destructor. Releases the dynamically allocated memory.

**Accessors**

* **int getSize() const**: Returns the current size of the vector.
* **T \*getData() const**: Returns a pointer to the array of elements.

**Modifiers**

* **void push(const T &value)**: Adds an element to the end of the vector.
* **void pop()**: Removes the last element from the vector.

**Operator Overloads**

* **T &operator[](int index)**: Provides access to the elements of the vector using the subscript operator.

**Copy Constructor**

* **vector(const vector &other)**: Copy constructor that creates a deep copy of another vector.

**Copy Assignment Operator**

* **vector &operator=(const vector &other)**: Copy assignment operator for assigning the values of another vector to the current vector.

# List

**Class Members**

* **listNode Structure:**
  + **next**: Pointer to the next node in the list.
  + **prev**: Pointer to the previous node in the list.
  + **data**: The data stored in the node.
* **list Class:**
  + **head**: Pointer to the first node in the list.
  + **tail**: Pointer to the last node in the list.
  + **count**: The current count of nodes in the list.

**Member Functions**

**Constructors and Destructor**

* **list()**: Default constructor. Initializes an empty list.

**Operations**

* **void add(T data)**: Adds a new node with the provided data to the end of the list.
* **void remove(listNode<T> \*node)**: Removes the specified node from the list.
* **bool contains(T data)**: Checks if the list contains a node with the provided data.
* **void clear()**: Removes all nodes from the list, freeing memory.
* **listNode<T> \*get(int index)**: Retrieves the node at the specified index.

# Node Structure

Represents a node in the maze graph.

# collectible Structure

Represents a collectible item in the maze.

# Player Class

Represents the player in the game, including information such as name, position, score, lives, collected items, and game statistics.

**Member Functions**

* **Player(int x = 0, int y = 0, string n = "")**: Constructor for the Player class.

# Maze Class

Represents the maze in the game, including the maze graph and methods for updating the console display.

**Member Functions**

* **Maze(int s = 0)**: Constructor for the Maze class.
* **addEdge(int x1, int y1, int x2, int y2)**: Adds an edge between two nodes in the maze.
* **print(Player p)**: Prints the current state of the maze to the console.
* **getSize()**: Gets the size of the maze.
* **updateConsole(Player p)**: Updates the console display with the current maze state.

# Game Class

Encapsulates the game logic, including maze generation, player movement, obstacle and collectible interactions, and game state tracking.

**Member Functions**

* **Game(int s, string n = "")**: Constructor for the Game class.
* **generateWalls()**: Generates random walls in the maze.
* **generateObstacles()**: Generates random obstacles in the maze.
* **generateCollectibles()**: Generates random collectibles (coins and trophies) in the maze.
* **pause()**: Saves the game state to a file for later resumption.
* **resume(Player& p)**: Resumes the game from a previously saved state.
* **print()**: Prints the current state of the maze and player information to the console.
* **movePlayer(char dir)**: Moves the player in the specified direction.
* **removeObstacle(int x, int y, Queue<obstacle>& obstacles)**: Removes an obstacle from the list of obstacles.
* **playerCollision()**: Handles player collisions with maze elements.
* **isFinished()**: Checks if the game is finished.
* **gameOverScreen()**: Displays the game-over screen with player statistics.
* dijkstraShortestPath(): Calculates the shortest solvable path in the maze by Dijkstra algorithm

# scores Structure

Represents a structure to hold score-related information.

# 10. treeNode Class

Represents a node in the binary search tree.

# Tree Class

Represents a binary search tree to organize and sort scores.

**Member Functions**

* **insertbyscore(treeNode\*& root, scores d)**: Inserts a node into the BST based on the score.
* **insertbycoins(treeNode\*& root, scores d)**: Inserts a node into the BST based on the number of coins.
* **insertbytrophies(treeNode\*& root, scores d)**: Inserts a node into the BST based on the number of trophies.
* **transferToArray(treeNode\* r, scores\* arr, int& i)**: Transfers the BST data to an array.

# Solver

**Members:**

* **Node \*\*maze**: 2D array representing the maze.
* **int rows, cols**: Number of rows and columns in the maze.

**Member Functions:**

* **MazeSolver(Node \*\*maze, int s)**: Constructor that initializes the MazeSolver with the given maze and size.
* **bool isSolvable()**: Checks if the maze is solvable using Depth-First Search.
* **Node findStart()**: Finds the starting point in the maze.
* **bool dfs(int x, int y, vector<vector<bool>> &visited)**: Depth-First Search function for exploring the maze.

# Functions:

1. **void gotoxy(int x, int y):** Moves the cursor to the specified coordinates (x, y) on the console screen.
2. **void setcursor(bool visible, DWORD size):** Sets the visibility and size of the console cursor.
3. **char getInput():** Waits for a valid input character ('w', 'a', 's', 'd', or 'p') and returns it.
4. **void drawHeading():** Draws the game heading on the console screen.
5. **int menu(string &n):** Displays the game menu, takes user input, and returns an integer corresponding to the chosen option.
6. **int main():** The main function where the game loop and menu interactions occur. It initializes the game, handles menu options, and runs the game loop until the user chooses to exit.