"The Quest for the Crystal Kingdom"

Scenario:

The player is on a quest to retrieve a legendary crystal from a dangerous and enchanted forest. The forest is filled with enemies and obstacles, and the player must navigate through it to find the crystal. The player has a map of the forest, which shows the location of the crystal and the player's current location (always (0,0)). However, the map is not detailed, and the player must explore the forest to find the best route to the crystal.

In the forest, the player must defeat enemies and overcome obstacles to progress through the journey. The player can also collect reward items that will help them on their journey.

Tasks:

- Implement an AVL tree to store the player's inventory of item scores. The AVL tree should allow the player to add and remove item scores (based on rewards). The first generated id should be 100. The rest of ids will be generated randomly (0-200). All nodes beside the root should be inserted based on the value of id.

    o The node class/structure should be maintained as follows:

    class Node
    {
        int id; //generated randomly int reward_score; //score of a particular reward you obtained int count; //to avoid duplicate id nodes (maintain count of the id) Node *next;

    }

- Implement Floyd's algorithm to calculate. the shortest path between any two areas in the forest. The algorithm should take into account the locations of enemies and obstacles

- Implement Prim's algorithm to find the minimum spanning tree of the forest. The algorithm should take into account the locations of enemies and obstacles.

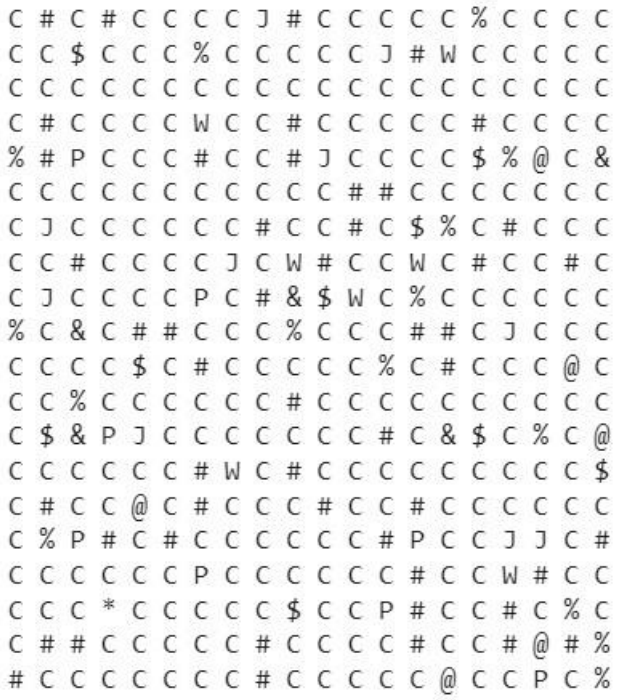
- Implement Kruskal's algorithm to find the minimum spanning tree of the forest. The algorithm should take into account the locations of enemies and obstacles.

- Implement Dijkstra's algorithm to find the shortest path from the player's current location to the crystal. The algorithm should take into account the locations of enemies and obstacles.

Note: You can specify the weights of the edges towards the obstacles as 100, all other edges have a weight of 1.

Details of the Game Map:

The game map is built with alphabets and special characters specified in the key given below. A sample map is given in figure 1.



C: Clear path
J: Jewel
P: Potion
W: Weapon
%: Death point
#: Obstacle
&: Dragon
$: Goblin
@: Werewolf
*: Crystal (goal point)

Figure 1: A sample Game Map

The map has the following entities:

i. Safe paths: These are the nodes where you (the character) can move without any restriction.
ii. Obstacles: These are the nodes from where you should not move any further since the weights of their edges are costly. The player should choose an alternate path on encountering an obstacle.
iii. Rewards: These are the nodes on which you visit to collect items such as Jewels, Weapons, and Potions. On collecting a jewel, weapon, or a potion, you will get +50, +30, and +70, respectively in your score.
iv. Enemies: These are the nodes that if you visit them, one of your reward item will be removed from the item inventory based on the following three enemies:
    a. If you visit the node with a Werewolf, you will lose a weapon.
    b. If you visit the node with a Goblin, you will lose a potion.
    c. If you visit the node with a Dragon, you will lose a Jewel.

    Note: Losing a reward will decrement the score for that reward as well.
v. Death points: These are the nodes that if you visit on them, you will die. You will have to restart the game from the starting point.

Expected Outcomes:

1. The shortest path between any two areas in the forest.
2. The shortest path from current location.
3. The minimum spanning tree of the forest by both algorithms.
4. The score increased/decreased at every step, and the final score at the end.
5. The menu must have these features but is not limited to

    - Shortest path using default location (0,0) by Floyd/ Dijkstra.
    - Shortest path using custom location (coordinates entered by the user) by Floyd/ Dijkstra.
    - Minimum Spanning Tree by Prims and Kruskal.

Restrictions:

- The algorithms should be implemented from scratch, without using any pre-built libraries or APIs.
- The program should be written in C++ and should be well-documented and easy to understand.