Data Structures and Algorithms Project Report: Knockout Tournament Bracket

1. Introduction and Data Structure Design

In this project, we built a single-elimination knockout tournament system using full binary tree (BT). A knockout bracket fits naturally with a binary tree because each match involves two competitors, and the winner continues up to the next round until reaching the final match, which become the of the tree.

We use a Player structure to represent both real players and match nodes:
+ Leaf Nodes (Actual Player) | This contains only player names. Their matchId and score are irrelevant.
+ Internal Nodes store |

i. matchId > the unique match number (1-7)
ii. Name > winner name

iii. score > winner score for that match.

We rely on std::unique_ptr<Player> for left and right child nodes. This makes memory handling automatic and follows t=modern C++ conventions.

2. Bracket Construction & Traversal

Building the Tournament (buildTournament)

Our bracket has **8 players**, so the binary tree is perfectly balanced with:

- 4 Quarterfinal matches
- 2 Semifinal matches
- 1 Final match

This gives the tree a height of 3.

i. Create all internal match nodes first Assign matchId from Bottom to Top)
ii. Randomize player positions (sufflePlayer)
iii. Attach players to leaf nodes

Tree Traversal:

Function	Purpose	Algorithm
Height ()	Finds match round level	Standard recursive height calculation
findLeaf ()	Finds a specific player	DFS until matching leaf is found
printPlayers ()	Displays all players	DFS filtering leaf nodes
printMatchesAtRound ()	Shows matches in a specific round	Uses height comparisons

3. Advanced Features & Algorithms

A. Path to Final (pathToFinal)

This feature shows **how far a player actually progressed** in the tournament.

The process:

1.Collect the theoretical path: Starting at the root, we use DFS to find the target player in the leaves and then gather all matchId along the path to the root.

2.Check all actual results: For each match along that path, if the winner's name matches the player mean they advanced and if not mean they got eliminated.

B. Would Two Players Meet? (wouldMeet)

This uses the concept of **Lowest Common Ancestor (LCA)**.

In our tournament tree:

-if two players keep winning

-they will meet at the first match node that is ancestor of both players.

Then we apply a simple LCA algorithm:
- The returned node gives the matchId and height.

-The height corresponds to the round mean height 1 (Quarter Final), height 2 (semifinal) and height 3 (FINAL).

4. Time Complexity Summary

Because the bracket is balanced and small, all operations run efficiently.

Operation	Time Complexity	Explanation
buildTournament ()	**O(N)**	Building nodes + shuffling players
playRound	**O(1)**	Direct access to match nodes
Height ()	**O(N)**	Standard DFS height calculation
findLeaf ()	**O(N)**	Worst-case search for the target player

Operation	Time Complexity	Explanation
pathToFinal ()	**O(N)**	Searching + checking each match
wouldMeet () (LCA)	**O(N)**	Balanced tree makes it close to O(log N)

5. Conclusion

This project allowed us to apply several DSA concepts in a practical and interesting way by modeling a **knockout tournament** with a **full binary tree**.

Key concept we used:

- Recursion: for building and navigating the tree.

- DFS Traversal: For searching, printing and analyzing data.
- LCA: to check where two player would eventually meet.
- Modern C++ feature: especially unique_ptr for memory safety.

The system is also easy to expand. If we wanted to support 16 or 32 players, only the build logic needs adjustment-everything else scales automatically thank to the recursive algorithms.