Knock Out Tournament Bracket Manager

Modeling and Simulating Competition Brackets Using Binary Tree Data Structures

Tournament Structure: A Binary Tree Approach

Core Architecture

Our tournament system leverages a **complete binary tree** to represent the entire bracket structure. This elegant data structure naturally mirrors the elimination process.

- **Leaf Nodes:** Individual players (Alice, Bob, Charlie, etc.)
- **Internal Nodes:** Match outcomes (QF1, SF1, Final)
- **Root Node:** Championship match and ultimate winner

Team Contributions

- **Setup & Randomization:** Chay SengHap
- **Match Simulation:** Seng Dina
- **Search Operations:** LySrongchhay
- **Tree Traversal:** Yann Sokmeng
- **Matchup Prediction:** Yun Eychean
- **Player Statistics:**Sor Channorakpitou
- **Bracket Display:** Roth Sorayuth

Setup and Randomization

1

buildTournament()

Constructs the fixed 8-player bracket topology with proper hierarchy: Final → 2 Semifinals → 4 Quarterfinals. Implements **std::unique_ptr** for automatic memory management, eliminating manual cleanup.

2

shuffleVector()

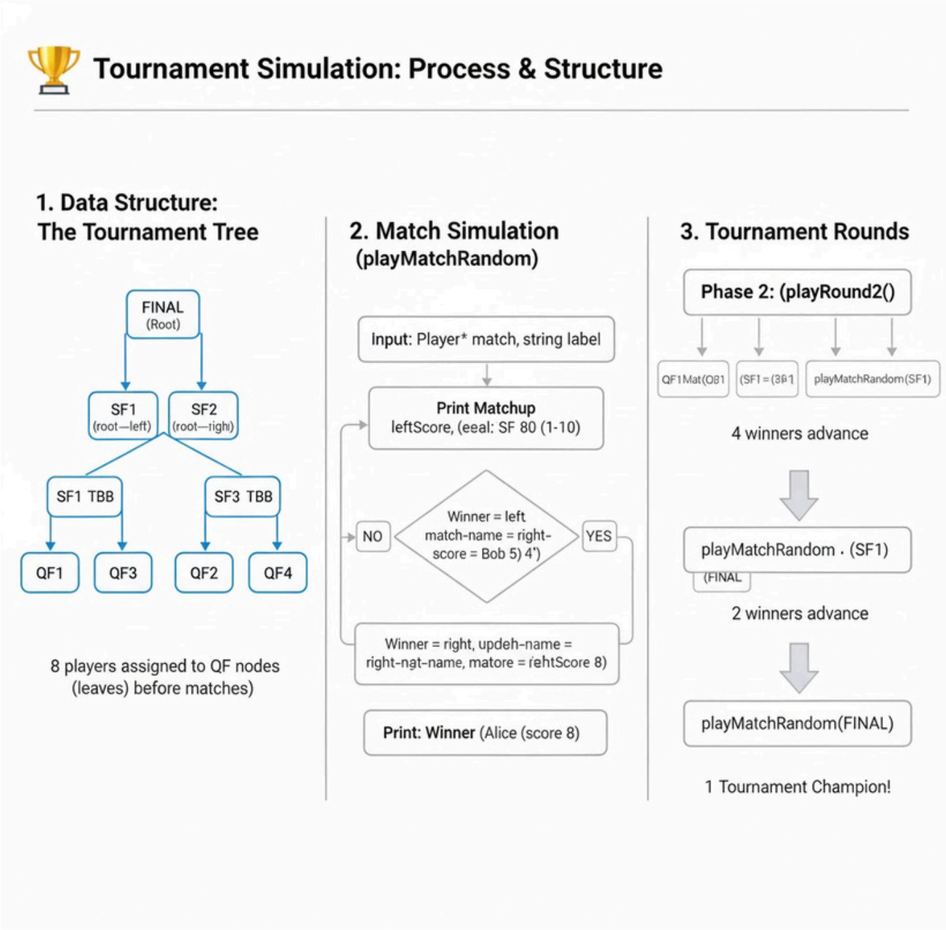
Applies the **Fisher-Yates Shuffle** algorithm to randomly order the initial 8 players. This ensures fair and unbiased seeding, preventing any predetermined advantages in bracket positioning.

3

randomScore()

Leverages **std::mt19937** Mersenne Twister engine combined with **std::uniform_int_distribution** to generate realistic, unpredictable match scores for tournament simulation.

Match Simulation Engine



Core Mechanism

The **playMatchRandom()** function drives match outcomes by generating random scores for competing players, determining the winner, and propagating results upward through the tree structure.

Round Progression

01

playRound1() - Quarterfinals

Simulates four QF matches using direct pointer access (root→left→left.get()) to locate and process specific match nodes.

02

playRound2() - Semifinals

Advances winners from QFs to two SF matches, updating internal nodes with victorious players and their scores.

03

playFinal() - Championship

Determines the tournament champion by simulating the final match between the two semifinal winners.

Search Operations

findLeaf(name)

Goal: Locate a specific player in the bracket

Method: Depth-First Search (DFS) traversal checking only leaf nodes where left and right pointers are null

Use Case: Quickly find any player's starting position

findMatchById(id)

Goal: Locate any match by its unique identifier

Method: Complete DFS over entire tree structure, comparing each node's matchId against target

Use Case: Direct access to specific rounds (QF1, SF2, etc.)

findMatchByName(name)

Goal: Find matches won by a particular player

Method: DFS searching internal nodes (matchId ≠ 0) where winner's name matches input

Use Case: Track player progression through tournament

Tree Traversal & Output Display

height(node)

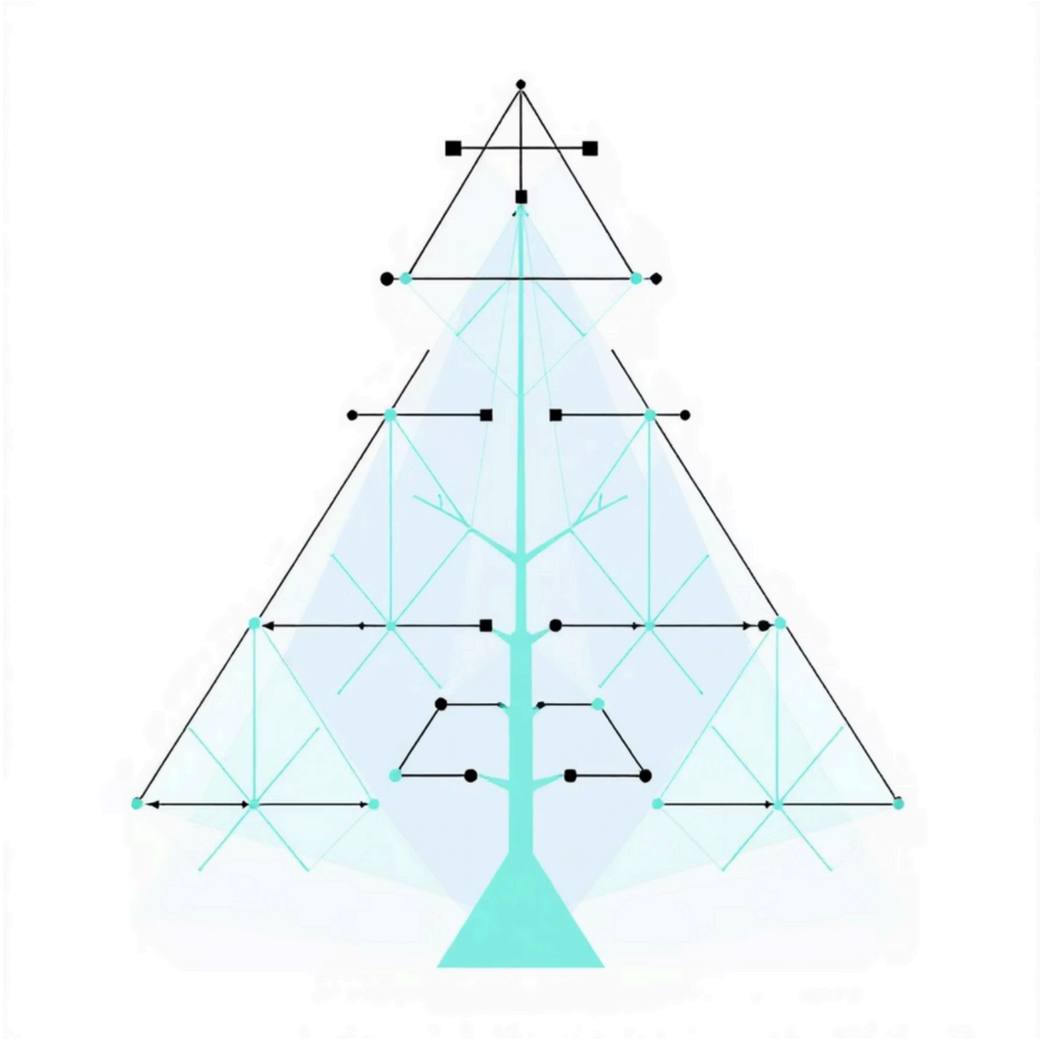
This function figures out which round a match belongs to.
Players are round 0, quarterfinals round 1, semifinals round 2, and the final is round 3.

printPlayers()

This function goes through the tournament and prints all the players at the bottom of the tree, basically showing the full list of participants.

printMatchesAtRound(round)

This function prints only the matches from a specific round.
If you give it round 1, it shows all quarterfinals; round 2 shows semifinals; round 3 shows the final, including each match's details.



Round-Height Mapping

- **Height 0:** 8 Players (Leaves)
- **Height 1:** 4 Quarterfinals
- **Height 2:** 2 Semifinals
- **Height 3:** 1 Final Winner

Matchup Prediction: Lowest Common Ancestor

lca(root, a, b)

Implements the classic **Lowest Common Ancestor** algorithm for binary trees. Identifies the lowest node in the tree containing both player A and player B as descendants.

Key Insight: The LCA node represents the first match where their bracket paths would intersect if both players continue winning.

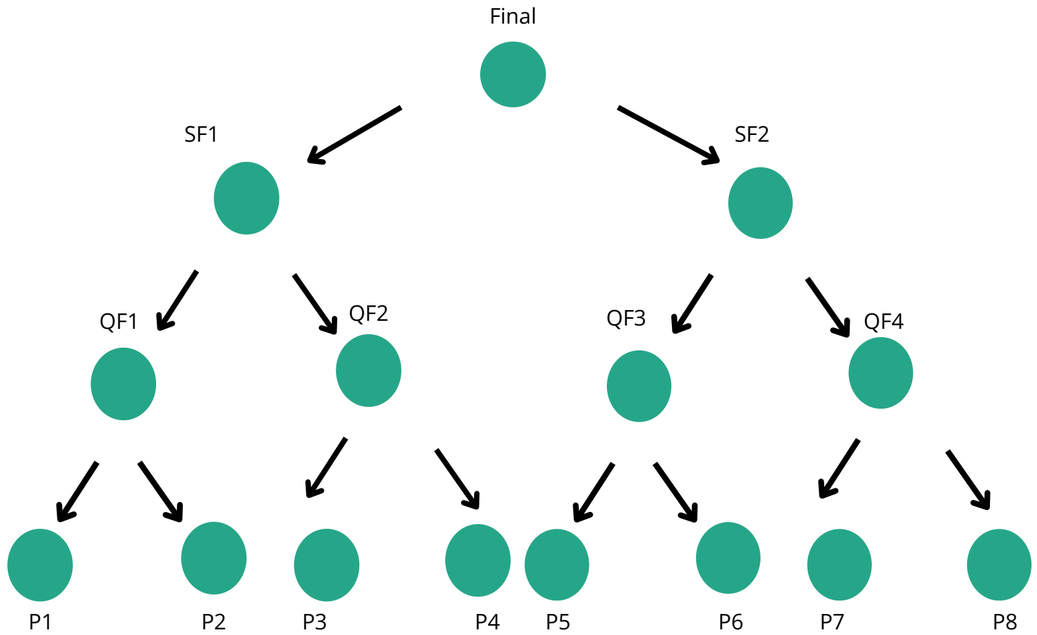
wouldMeet(p1, p2)

Step 1: Locate leaf nodes for both players

Step 2: Execute lca() to find meeting match node M

Step 3: Return M's matchId and height() (round number)

Output Example: "Alice and Bob would meet in SF1 (Round 2)"



Player Progression & Statistics

1

collectTheoreticalPath()

Performs recursive **upward traversal** from player's leaf node to root, collecting all matchIds in their potential bracket ladder. Represents the complete path if the player wins every match.

2

pathToFinal() - Actual Run

Obtains theoretical path, then iterates through matches sequentially. **Stops and returns** at the first match where the player's name doesn't match the recorded winner, indicating elimination point.

3

getTotalScoreByName()

Executes DFS traversal across entire tree structure, accumulating scores only from internal match nodes where the player's name appears as winner. Provides comprehensive performance metric.

Bracket Display

printBracket() Architecture

This master function shows the complete tournament output, serving as the primary driver for result presentation.

01

Initialize with Players

Calls printPlayers() to display Round 0 - all 8 tournament participants

02

Display Each Round

Loops from r=1 to maxRound, calling printMatchesAtRound(r) for progressive bracket revelation

03

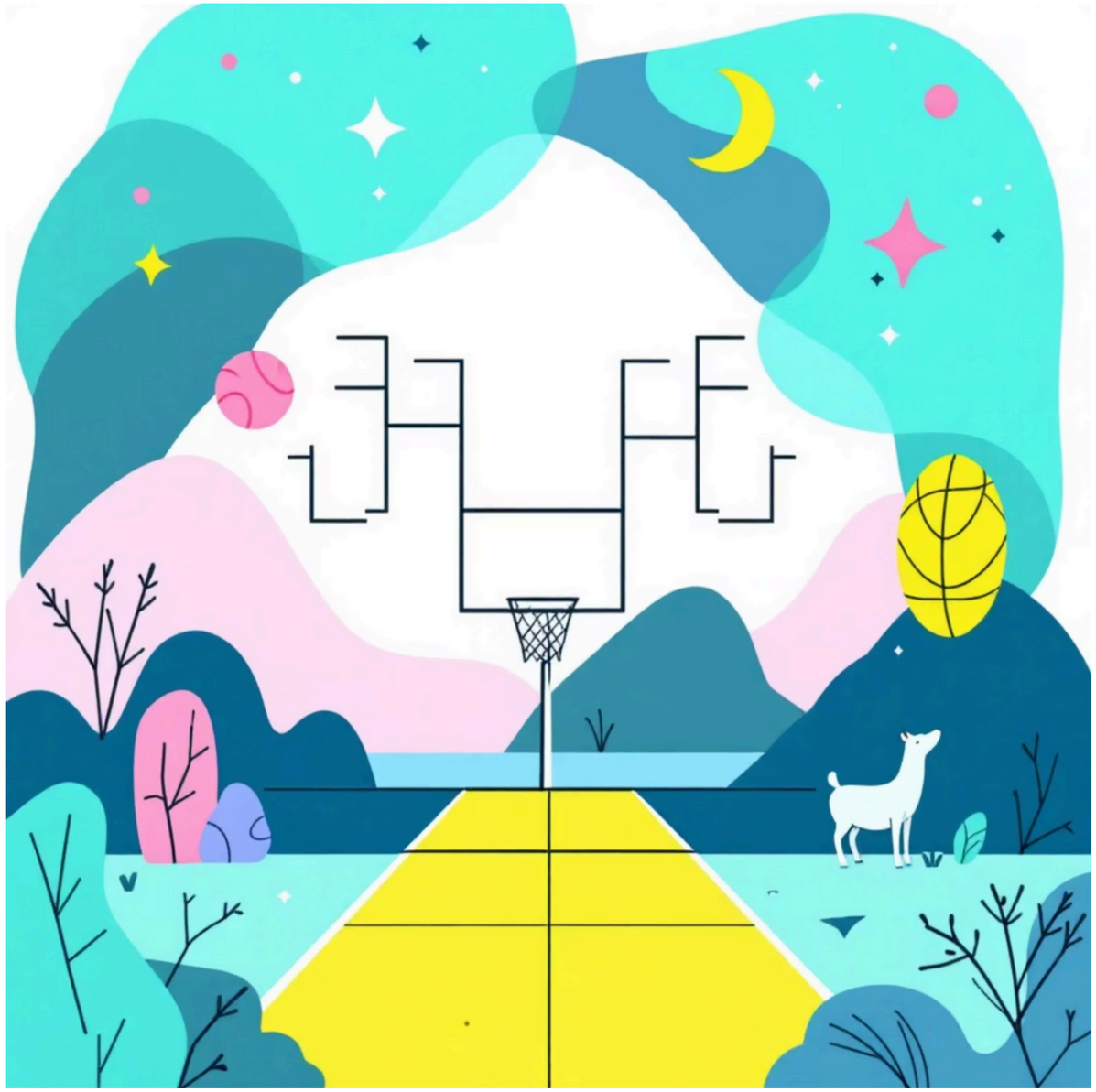
Announce Champion

Directly accesses root node after playFinal() completes to display ultimate winner

```
void printBracket() const {
        int maxRound = height(root.get()); // FINAL level = 3 in this 8-player tree

        cout << "\n=== ROUND 0: PLAYERS ===\n";
        printPlayers(root.get());

        for (int r = 1; r <= maxRound; ++r) {
            if (r == maxRound)
                cout << "\n=== FINAL (ROUND " << r << ") ===\n";
            else
                cout << "\n=== ROUND " << r << " ===\n";
            printMatchesAtRound(root.get(), r);
        }
    }
```



Project Summary & Demonstration

Achievement

Successfully modeled a complete single-elimination tournament system using **Binary Tree** data structures in C++, demonstrating elegant mapping between competition brackets and tree topology.

Key Features

Implemented sophisticated functionality including randomization, round progression, path analysis using LCA algorithm, and comprehensive player tracking throughout the tournament lifecycle.

Live Demonstration

Watch as the system generates random player seeding, simulates match outcomes, and demonstrates advanced features like wouldMeet() predictions and pathToFinal() progression tracking.

View Code Demo

Questions & Answers

Thank You!