Tournament Bracket

Members:

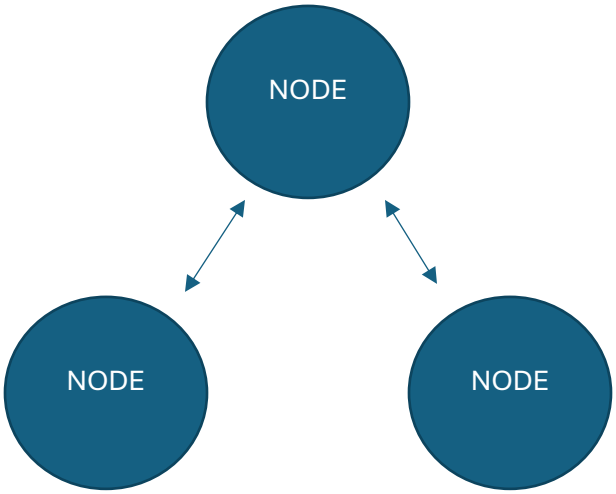
Kong Sophea IDTB110245

Lim Vinchay IDTB110036

Chey Naryvety IDTB110199

Vong Sovanpanha IDTB110332

1. Chosen representatives: Pointers



We chose to represent the Knock-Out Tournament Bracket Manager with pointers because we don't know the number of players that will be participating in the tournament, so using array is not suitable.

```
10  struct Node {
11      string name;
12      int matchId;
13      int round;
14      string winner;
15      Node* left;
16      Node* right;
17      Node* parent;
18      bool isLeaf;
19
20      Node(string n = "BYE") : name(n), matchId(0), round(0), winner("?"),
21                               left(nullptr), right(nullptr), parent(nullptr),
22                               isLeaf(true) {}
23  };
```

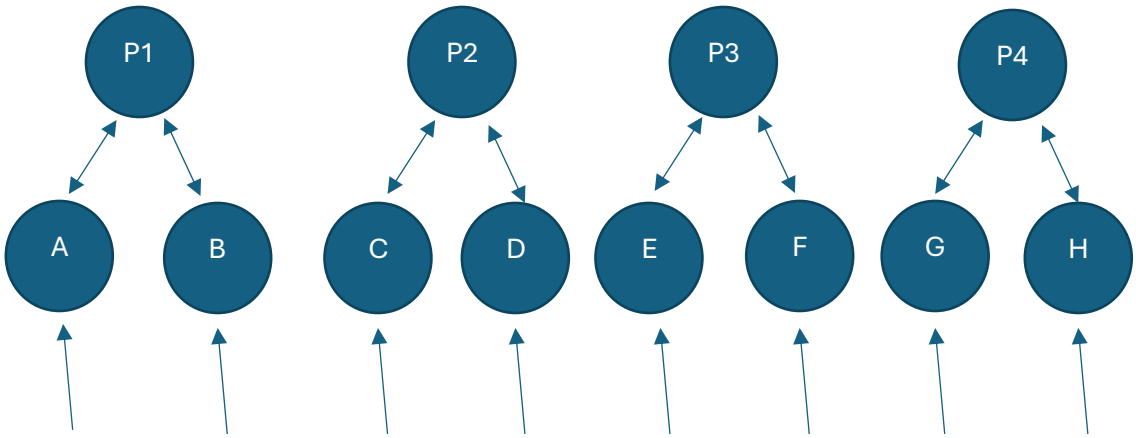
2. Core implementation

a. Build bracket:
    i. Complexity: O(N)
    ii. Explanation:
        1. Take the string list Build list of Nodes with that string(Name)

A , B , C , D , E , F , G , H

NODE(A) , NODE(B) , NODE(C) , NODE(D) , NODE(E) , NODE(F) , NODE(G) , NODE(H)

        2. On the list of Nodes we pick 2 of them at a time and make a root node for them.

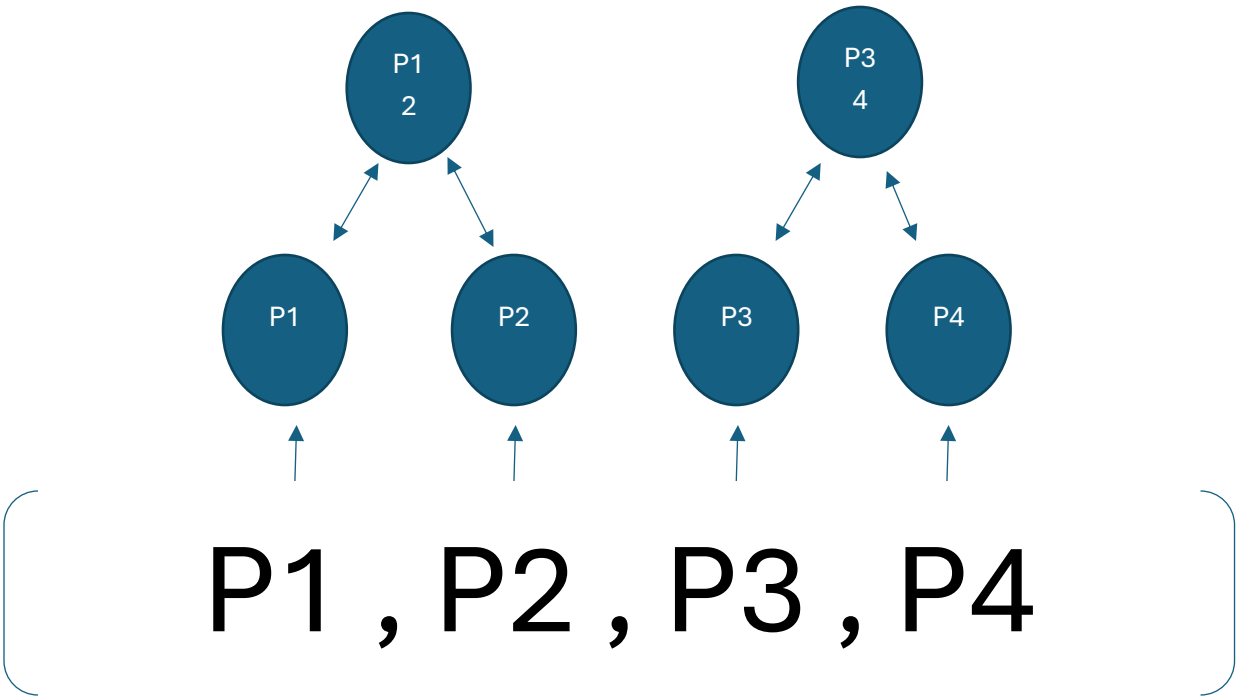


NODE(A) , NODE(B) , NODE(C) , NODE(D) , NODE(E) , NODE(F) , NODE(G) , NODE(H)

        3. Push the root node to a list.

P1 , P2 , P3 , P4

        4. Work on the root list to create root for them.



5. Repeat the process till only one node left

b. Record result
    i. Explanation:
        1. Validate the match ID:

            If the ID is out of range or the match already has a winner, return false.

        2. Identify competitors:

            Get the left and right players from the match node.

            If either side hasn't been resolved yet (ex. "?"), return false.

        3. Determine the winner:

            If one side is "BYE", the other side automatically wins.

            If both sides are valid, check if winnerName matches either player. If it does, assign the winner. Otherwise, return false.

        4. Propagate automatic wins upward:

            Traverse up the tree from the current match.

            If a parent match has one "BYE" side and no winner yet, assign the other side as winner.

            Repeat until no more automatic wins can be assigned.

c. Path query
    i. Time Complexity: O(N)
    ii. Explanation
        1. Find each player's leaf node
            - if leaf, compare name; otherwise search left, then right.
            - If it finds a leaf with name then it returns that Node*, otherwise nullptr

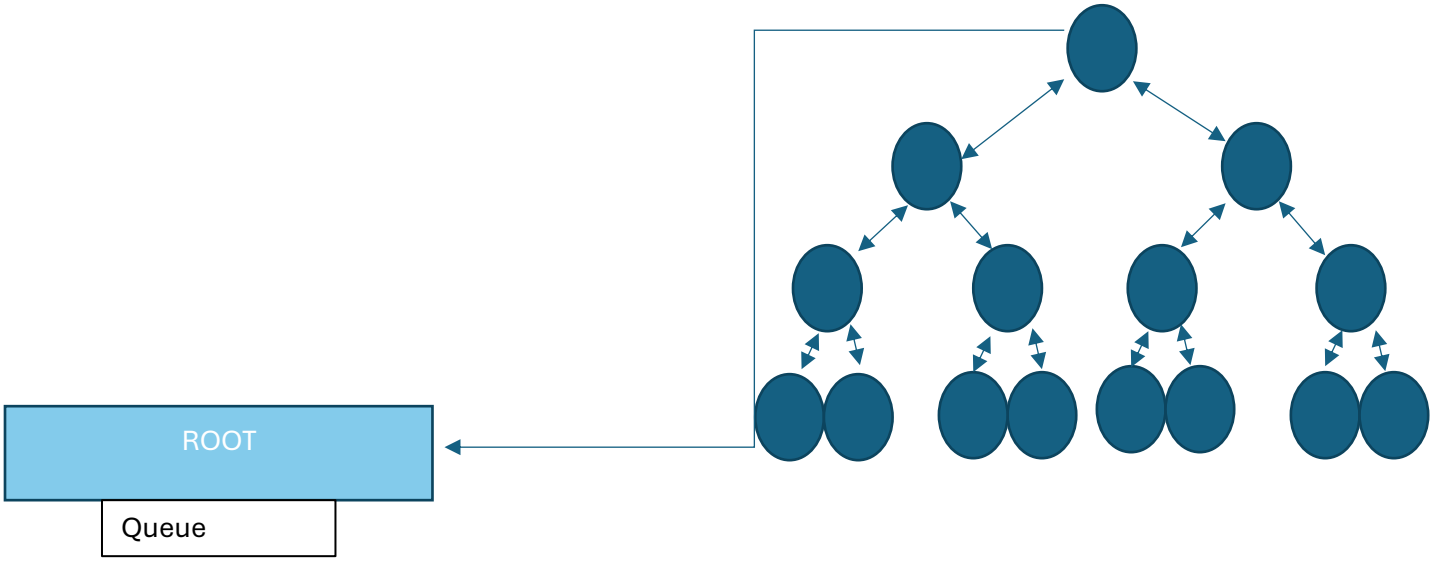
        2. Start from the player's fist match
            - Move to the leaf's parent, which represents the player's first match in the bracket

3. Move upward through the bracket
   - For every ancestor match: Add cur->matchId to the path.
4. Stop early if the player is eliminated
   - If the match already has a winner
     and the winner is not the current player,
     stop immediately because the player cannot advance further.
5. Continue until the final if the player has not been eleminated
   - Move upward using cur = cur->parent
     until reaching the root or breaking due to elimination.
6. Return the match path
   - Return the collected match IDs, which represent the player's full path toward the final (ending early if they lose).

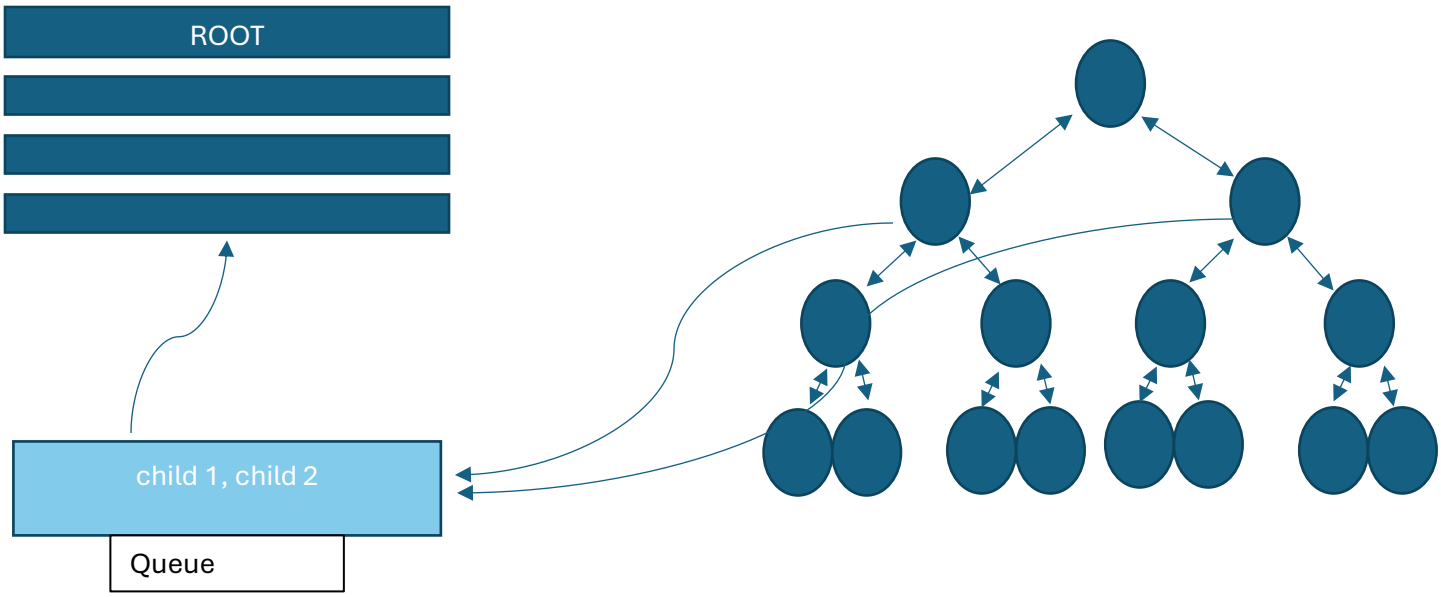
d. Would Meet
   i. Time Complexity: O(N)
   ii. Explanation
      1. Find each player's leaf node
         - if leaf, compare name; otherwise search left, then right.
         - If it finds a leaf with name then it returns that Node*, otherwise nullptr
      2. Check both found
         - If either a or b is nullptr (not found), wouldMeet returns {0,0} immediately
      3. Build ancestor/chain for a
         - Starting from a, follow .parent up to the root, pushing each node into chainA array(vector)
      4. Check for common node that is not a leaf
         - Compare t to every node x in chainA. If x == t (they are the same node) and ->isLeaf is false, that node is the nearest common non-leaf ancestor. Return {t->matchId, t->round}.
      5. If no suitable common ancestor
         - If the loops finish without finding one, return {0,0}

e. Print bracket
   i. Complexity: O(N)
   ii. Explanation:
      1. Create 2d dynamic array size = levels + 1 to store node on each level

2. Create a queue and push root into it



3. Pop the node out of the queue and push the two children into the queue



4. Push to node into the 2d array by their level
5. Traverse the 2d array and print it out

Demo

```
Bracket built for 8 players, 3 rounds

Match Results:
Match 1 (Round 1): Anna [43] vs Ben [54] -> Winner: Ben
Match 2 (Round 1): Chou [16] vs Dara [74] -> Winner: Dara
Match 5 (Round 2): Ben [86] vs Dara [49] -> Winner: Ben
Match 3 (Round 1): Ean [88] vs Faye [30] -> Winner: Ean
Match 4 (Round 1): Gita [88] vs Hout [30] -> Winner: Gita
Match 6 (Round 2): Ean [17] vs Gita [35] -> Winner: Gita
Match 7 (Round 3): Ben [93] vs Gita [53] -> Winner: Ben

Final Bracket:
[Match 7, Round 3] Winner: Ben
    [Match 5, Round 2] Winner: Ben
        [Match 1, Round 1] Winner: Ben
            [Leaf] Anna
            [Leaf] Ben
        [Match 2, Round 1] Winner: Dara
            [Leaf] Chou
            [Leaf] Dara
    [Match 6, Round 2] Winner: Gita
        [Match 3, Round 1] Winner: Ean
            [Leaf] Ean
            [Leaf] Faye
        [Match 4, Round 1] Winner: Gita
            [Leaf] Gita
            [Leaf] Hout

wouldMeet(Anna,Dara) => matchId=5, round=2

pathToFinal(Anna): 1 5 7

Champion: Ben
```