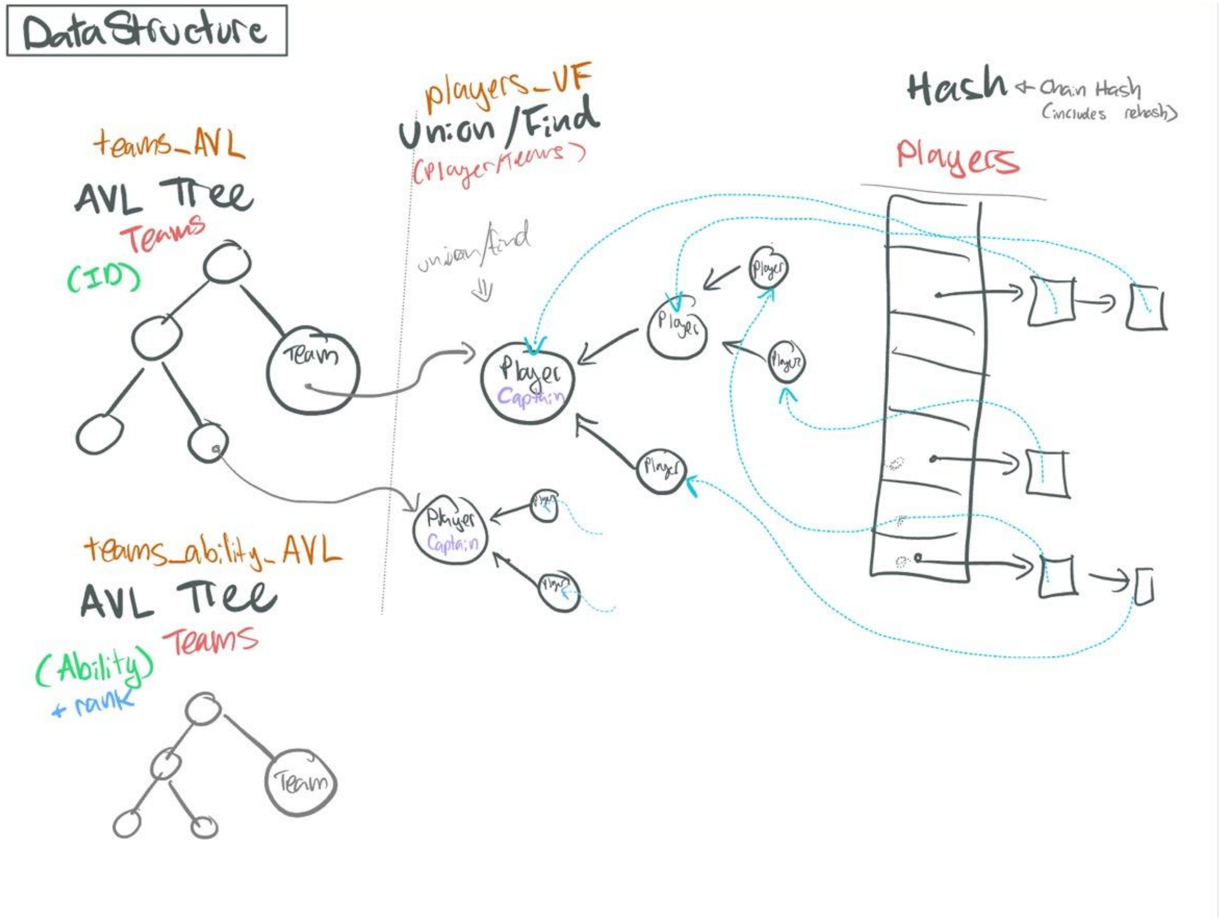
Dry

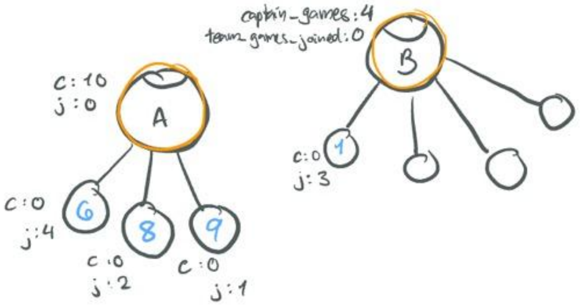


Num Played Games Algorithm Illustrated

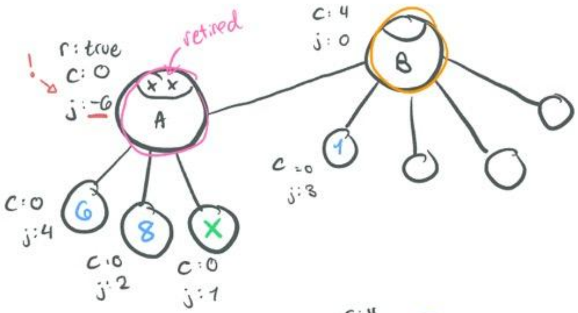
○ = captain

● games played = c of captain − j of all retired captains on way − j of X

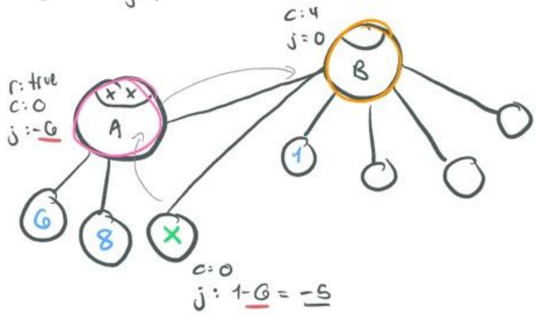
↑ captain_games ↑ team_games_joined ↑ team_games_joined ↑ games played of player X



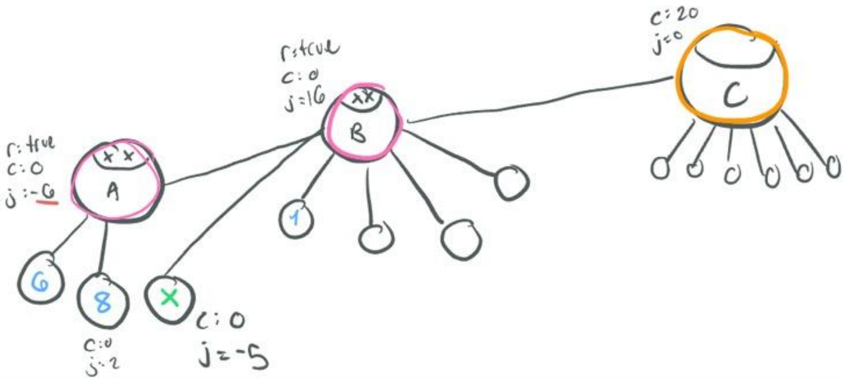
B buys A OR A buys B



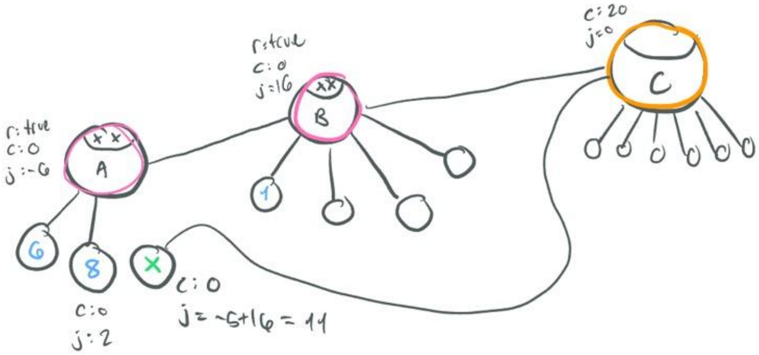
Find X game played



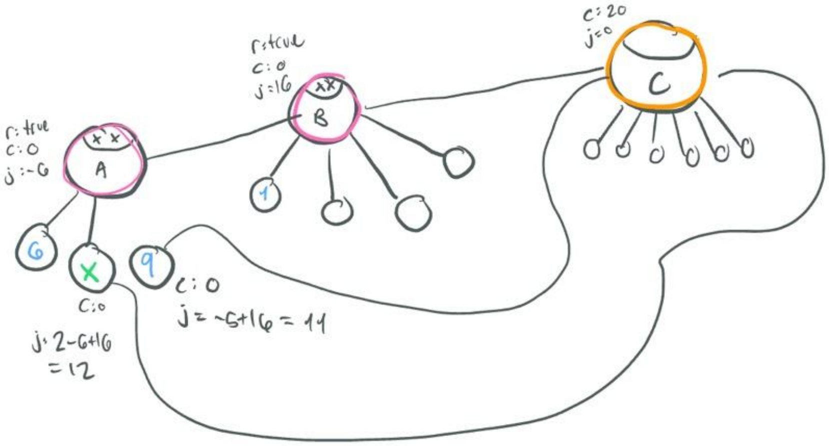
↑ update j to add all j of retired captains

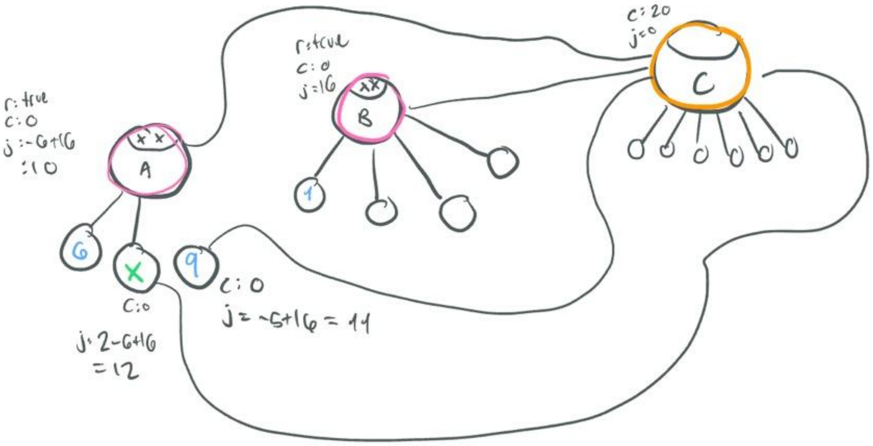


Find X game played



Find × game played





Function Implementations

world_cup_t()

- Create empty AVL trees one sorted by IDs the other by Ability: teams_AVL, teams_ability_AVL => O(1)
- Create empty UnionFind that uses a chain Hash, connects between teams_AVL and the players: players_UF=> O(1)
- O(1)

virtual ~world_cup_t()

- teams tree destructor ⇒ O(k)
- players UF destructor ⇒ O(n)

add_team(int teamId)

- Add team to both AVL trees => O(log(k))

remove_team(int teamId)

- Remove team from both AVLs => O(log(k))

add_player(int playerId, int teamId, const permutation_t &spirit, int gamesPlayed, int ability, int cards, bool goalKeeper)

- Find Team in teams_AVL => O(log(k))
- Add new player to players_UF => O(1)
    - If Team doesn't have a captain, set player as captain
    - Else, make it point to the team captain
- Temporarily remove Team from teams_ability_AVL => O(log(k))
- Update Team Stats => O(1)
- Re add Team to teams_ability_AVL => O(log(k))
- Player's games_of_captain_when_joined == captain_games of Captain node => O(1) [because Team has direct pointer to Captain]
- O(log(k))

play_match(int teamId1, int teamId2)

- Find both teams using teams_AVL => O(log(k))
- Compare Stats => O(1)
- Update points for directly to the Team class => O(1)
- Increment amount of games played to the Captain node of the team only => O(1)
- Return appropriate value depending on the previous comparison => O(1)
- O(log(k))

num_played_games_for_player(int playerId)

- [Use diagram as reference]
- Each player node in players_UF holds two value: captain_games (== 'c' in illustration) and games_of_captain_when_joined (=='j' in illustration) as well as a bool of isRetired and isCaptain
- When a Player joins a Team the games_of_captain_when_joined == captain_games of Captain node, and the captain_games (unless its a Captain) is set to 0 too
- Find player (uses Hash O(1), but then compresses the UF so log*n) => O(log*(n)) amortized with the rest of the UF actions.
    - Updates Player Node's games_of_captain_when_joined to be its own value minus the games_of_captain_when_joined of all the retired Captains (sum_retired) on the way => O(log*n))
        - While doing this, the nodes on its path which point to the new Captain/root will also have their games_of_captain_when_joined updated (whenever walking past a retired Captain update sum_retired accordingly)
        - This takes only log*n because there are two traversals, in one you find the newest Captain and the sum_retired, and in the second you change parent relationship for all nodes on path and update sum_retired
        - Meaning you get access to Captain node
- Return captain_games of Captain node minus the games_of_captain_when_joined and added to it the initial gamesPlayed of the Player when it was initialized => O(1)
- O(log*(n))

add_player_cards(int playerId, int cards)

- find() the set of the player in the UF ⇒ O(log*(n)) amortized with the rest of the UF actions.
- player.update_cards() ⇒ O(1)

get_player_cards(int playerId)

- get the player from the players_UF hash ⇒ O(1) on average
- return player.cards ⇒ O(1)

get_team_points(int teamId)

- Find Team in AVL teams_AVL => O(log(k))
- Return the team_points => O(1)
- O(log(k))

get_ith_pointless_ability(int i)

- Find Team in the ability sorted AVL teams_ability_AVL => O(log(k))
- (Note when added a new Player made sure to update the AVL)
- O(log(k))

get_partial_spirit(int playerId)

- set attributes: product and seniors_product
    - in addition to the normal UF attributes, and the normal Player attributes, every node (and specifically, every captain (set)) also has:
        - team_product: the product of all the players that are in the team. whether added or bought.
        - seniors_product: the product of all the players in the team that are senior to the player, including his own initial permutation.
            - the full permutation can only be obtained after multiplying a node by the seniors_products of all the old team captains (subset roots) up from its node, all the way up to the team's captain (the set's root).
- how the attributes are maintained:
    - buy_team(A,B)
        - update products: ⇒ O(1)
            - we will assume team A buys team B.
                - note that in terms of pointers, this works like a normal UF, i.e. the smaller set points at the larger set. no matter who bought who. this is why there are 2 different cases
            - if team A is bigger:
                - B.seniors_product = (A.seniors_product^-1) * A.team_product * B.seniors_product ⇒ O(1)
            - if team B is bigger:
                - B.seniors_product = A.team_product * B.seniors_product. ⇒ O(1)
                - A.seniors_product = B.seniors_product^-1 * A.seniors_product ⇒ O(1)
                    - (this effectively makes A ignore B when calculating partial_spirit.)
            - in the end, the root's team_product is updated to be the product of all the players in the team.

    - add_player()
- get_partial_spirit(player): O(log*n) amortized
    - we find the player through the UF array (implemented as a hash table) ⇒ O(1) amortized
    - find(player) ⇒ O(log*n) amortized
        - first we do a traversal to the root to get the product of all the old team captains on the way up, so that on the second traversal the update of the seniors_product is O(1) for every node.
        - on the second traversal to the root, we do union find path compression, and on every node on the way update its seniors_product to take into account all of its ancestors (besides the root)
    - return team_captain.seniors_product * player.seniors_product ⇒ O(1)
        - because the path was compressed, this takes into account all the old team captains on the way from the player to the captain (in the union-find)
    - O(log*n) amortized with the rest of the players_UF functions.

buy_team(int buyerId, int boughtId)

- find teams ⇒ O(log(k)
- update buyer team stats ⇒ O(1)
- unite() the players like a normal UF (but with the maintenance of permutations and games played (O(1))) ⇒ O(log*n) amortized with other players_UF actions
- remove bought team ⇒ O(log(k))
- The Captain which was the Captain of one of the old Teams but is not a Captain of the new Team becomes retired and its games_of_captain_when_joined becomes captain_games of new Captain node minus its retired captains captain_games. captain_games becomes 0. => O(1)
- remove and re-add the buyer team from the teams_ability_AVL ⇒ O(log(k))
- O(log(k) + log*(n)) amortized