HW1 Data Structure

Ron Raphaeli and Roey Graif

December 8, 2022

Intro:

AVL tree:

We constructed an AVL tree as we saw in class. The AVL tree is constructed exactly like the algorithem we saw therefor it holds for the complexity of log(n). We did add a few more functions to our AVL tree to help us with the worldcup functions I will mention the ones that are not revial:

1) SwitchNodesLocation, this function is O(1) and it helps us replace the node pointers when we remove players from the AVL tree.

2)inOrder and storeInOrder are O(logn) as we saw in the lecture.

3)treesToArray, take two sorted AVL trees and puts them in a sorted array using storeInOrder therefor also O(logn).

4)addFromSortedArray, puts elements of a sorted array in the tree, we find the middle of the array and put it in the root then for the sons of the root (using recursion) we look at the left side of the array and the right side and so on. this is O(n), we saw this algorithem in class.

Node:

the node class is simply as we saw in class with 2 children that are also nodes it holds for O(1).

Player.h

There are 2 classes in player.h. playerInTeam and Player, Player is a class that holds for the specific instructions we were given and playerInTeam has only a node pointer to the Player. The order in which we use for this 2 classes is the id order of the player.

functions:

Consists of "get" functions, constructors destructors and compare functions all of O(1) and are trevial

playerStats.h

This class is similare to playerInTeam class the diffrence is that for playerStats class we are using the lex order to sort our objects in the tree the order is <goals,cars,id>.

functions:

Consists of constructors destructors and compare functions all of O(1) and are trevial

Team.h

This class has information about the team and 2 player trees inside aswel (one tree is in order of the id and another is in order of the lex order we defined above).

functions:

we update the trees in the team with the following functions:

Consists of constructors, compare and other trevial functions all of O(1).

worldcup23a1.h

Our data structure consists of 4 AVL trees. Two are for all players in tournament one in order Id_player and the other in order Lex. The other 2 trees are Team trees with the order of Id_team. one tree is for all the teams and one tree is only for valis teams (once a team is valid it is in both trees).

We store a few functions in the private section of this file for convenience purposses I will denote them:

1)updateClosestPlayers-this function takes a player and finds him his prediccessor and succedor and updates them into the data of the player. finding the prediccessor and succedor take O(logn) the rest is O(1).

2)recursiveKnockOut-help the knockout team function the recursion is log(n).

worldcup23a1.cpp

This section contains the worldcup functions. Denote n-number of players, k-number of teams:

world_cup_t -

Initilize the worldcup database in O(1).

~world_cup_t -

Delete the database first remove all the players in the players id tree and players O(2n) then we remove all the teams and for every team revoe the players inside it in total all the players in all the teams is n so removing all the teams is $O(k)+O(n_{team1}+...+n_{teamk})$ and we know that $n = n_{team1}+...+n_{teamk}$ and we do this twice for the players tree and stats tree and we get overall:

$$O(2n)+O(k)+O(n_{team1}+...+n_{teamk})+O(n_{team1}+...+n_{teamk}) = O(4n)+O(k) = O(n+k)$$

add_team -

Finds if the team exists and if not we add it to the teams treethis proccess takes O(2logn) to find the team (if exists) and then to insert $\Rightarrow O(logn)$.

remove_team-

Find the team and once we do we shall remove it if its empty. We search for the team using the operator< this takes as O(logn).

add_player -

We add the player to the two player and stats trees this is O(2logn) and then find the team which takes O(logk) we also save the pointer to the team in the player data. after we do that we enter the player in the 2 trees that the team hold this takes $O(logn_{team}) \leq O(logn)$ we do this for both trees so O(2logn) and in total $\underbrace{O(4logn)+O(logk) = O(logn+logk)}_{\text{we showed these equalitis in class, they are trevial}}$. When adding the player to the team we add the stats to the total team stats, this will help us later therefor it is important to do it in the add_player function where we are permited to use O(logn+logk). We also aquaire the player with the successor and preduccessor this will help us in the get_closest function since this takes O(logn) we do this using the operator< of the playerstats which is the lexographic order. $\underbrace{O(logn)}_{\text{closest player}} + \underbrace{O(logn+logk)}_{\text{adding the player}} = O(logn+logk)$. It is also important to mention that the games the player played is being saved as minus the the games the team has already played later when the number of games of the player is requested we will simpy add the games of the player plus the games of the team to get a correct answer.

remove_player -

We remove the player from the 2 player and player stats tree log(n) in the database but before we remove it we look in the data of the player, we will find a pointer to the team he belongs to, therefor we reach the team in O(1) once we do that we can remove the player from the trees in the team data this takes $O(2logn_{team}) \leq O(2logn) = O(logn)$ we also update the players that are the succedor and predeccessor of the player that is being removed. In total the complexety holds for O(logn).

update_player_stats -

First we need to find the player in the datapase of the players this takes O(logn) then we will take his team pointer from his data in O(1) we will remove the player from the stats tree in both stats trees (one is the main stats tree and the other is the stats tree of the team) in O(2logn) then we update the players stats according to the input we got then we put back the player in the stats trees and update his succedor and predeccessor in O(logn). in total:

$$O(logn) + O(2logn) + O(1) = O(logn)$$

play_match -

Since we updated the data of the team points cards and so every time we added a player to the team we have access to this data in O(1) (checking the validation of the team is also O(1)), so all we need to do is find the team in the teams tree which is simply O(logk) and calculate the teams score in O(1)$\Rightarrow O(logk) + O(1) = O(logk)$

get_num_played_games -

Finding the player in the tree is O(logn) and then we take the pointer to the team and look at the teams games played. We add the players games played (Note: it is a negative number I explained why in the add_player function) with the games the team played and return the result.

get_team_points -

This is done by finding the team O(logk) and then taking the data (points) from the team.

unite_teams -

First of all we must check that the validation of the input and making sure the team doesnt exist in log(k) after that we will save the data of both teams (points, cards...) and add it together this is O(1) now we have to combine the teams trees. since the trees are sorted already (each tree with its own special operato<) we can take 2 trees and merge the into one sorted array by doing an inorder search and saving the elements in an array then we combine these 2 arrays into one array this takes O(n+m) (n-players in team 1 and m-players in team 2). We then remove the 2 teams this is also O(n+m) and create a new team with the newId we then add the sorted array back to this new teams tree by using the addFromSortedArray function I explained in the begining which is

O(n+m) and then we put the data (points, cards...) back in the new team. In total we get:

$$O(logk) + O(m + n)$$

get_top_scorer -

There are 2 options:

*) Top scorrer from all the players will be save in the worldcup members and will be updated every time we add or remove a player (because we can us O(logn) when adding or removing a player) therefor we can access it in O(1).

**) Top scorer from the team. First of all we will find the team in O(logk) and then in the team data we will keep a pointer to the player that is the top scorrer in the team this pointer will be updated everytime we add or remove a player from the team, this way it is always correct.

get_all_players_count -

There are 2 options:

*)We keep a counter in the worlcup members section and update it everytime we add or rempve a player O(1).

**) Find the team in O(logk) and we keep a counter in the data of the team that is being updated everytime we add or rempve a player O(1).

get_all_players -

Finding the team is O(logk) then we do an inorder search on the stats tree which is $O(n_{team})$ and when we do this we save the players in the stats order $\Rightarrow O(logk + n_{team})$

get_closest_player -

First we find the team and then we find the player inside the team this is O(logk+n_{team}) once we have the player we know who are the succesor and thr preduccessor we have a function that finds out which one of the two is closer (in O(1)) to ower player and return him.

knockout_winner -

There are 2 options:

k≤n) The complexity is O(k+r), because we need to do an inorder search on the valid teams tree, the search is O(k) because $k_{\text{valid teams}} \leq k$ and then we do the function recursiveKnockOut which is working on only the teams in the range of the requested teamId's therefor it is O(r) ⇒in total we get $O(k_{\text{valid teams}} + r) \leq O(k + r)$.

n<k) The complexity is O(n+r), We know that in every valid team there are 11 players or more therefor $k_{\text{valid teams}} < n$ and from the same explanation above we get $O(k_{\text{valid teams}} + r) \leq O(n + r)$