CENG218 – Analysis and Design of Algorithms Homework 2

Last update: April 22, 2022

Note: Please **do not be overwhelmed** by the length of this document. It is easier to do these tasks than to explain it. This homework will be done in pairs.

We want to implement a web service that allows its clients to view football match statistics in real time, and to make some queries such as "player of the match" at the end of the match. For this, we need to process stream data coming from the football matches. The data includes detailed match events such as a player passing the ball accurately or receiving a yellow card. Other than this, we have a database that contains information such as names and shirt numbers of the players for each team.

Your team is responsible for developing the core of the software: implementing three data structures and adapting them to meet our needs. More specifically, you must

- keep a **hash table** (Exercise 3) that stores all the statistics as well as an overall score for each player in **real time**, and
- construct two binary trees, a **max heap** (Exercise 1) and a **binary search tree** (BST) (Exercise 2), **at the end of the match** using the player scores that are stored in the hash table.

The hash table will be used for querying the statistics for any player at any time, while the trees will be used at the end of the match for querying the best players of the match and querying the players in a particular score range.

For simplifying your task, we have "mocked" the data in CSV files. Therefore, you need to process data stored in CSV files line by line rather than processing stream data in real time.

We want to test (and evaluate) each of these three parts **independently**. Thus, while constructing the max heap and the BST you will depend on a CSV file rather than using the actual hash table you create and fill.

For reading the CSV files and parsing the data line by line, you **can** modify and use the code provided in *ceng218_02_Sorting.tar.gz*. We provide structs for storing the entries in trees and tables, and we provide functions for displaying the trees and the tables. You **must** use these structs (e.g. *Node*) and functions (e.g. *print_complete_binary_tree*).

Please, download ceng218_hw02.tar.gz and extract the contents. The extracted directory contains four directories: one for evaluation, and three for each of the exercises. You only need to edit a single file in each project: exercise1.cc, exercise2.cc, exercise3.cc. Make sure you do not change any other file. After you finish writing and testing the code, you will submit a single compressed file that only contains these three files you edited!

Exercise 1 Max Heap (30 pts)

Edit *exercise1.cc* to fulfill the requirements below.

Two positional command-line arguments will be given to the program:

1. Path of a scores file (e.g. *data/galatasaray_fenerbahce/scores.csv)*
2. A small positive integer *k* for indicating the number of best players we want to query (e.g. $k = 3$ indicates that we want to view the 3 best players in the match)

You can assume both *path* and *k* are valid (Such file exists and $1 \leq k \leq n$ such that n is the total number of players appeared in this match).

This program

1. creates an empty max heap by calling a function you define named ***create_heap*** (This heap is essentially an array of ***Node*** instances. This is a struct we provide and it contains extra information *is_home* and *shirt_no* as well as *score*. You can exploit the fact that this heap will contain at most $11 + 11 + 3 + 3 = 28$ entries considering the starting lineups and the maximum number of players that can be substituted in.),
2. reads the relevant *scores.csv* line by line and adds scores to the heap one by one by calling a function you define named ***push*** (For efficiency, as well as consistency with our solution, this function should swap two nodes only when needed, i.e. there is no swapping in case of equal scores.),
3. displays the max heap (which is a complete binary tree) **after each new entry** calling the function we provide named ***print_complete_binary_tree*** (In source code, you can find a simple example that calls this function.), and
4. displays *k* best players once all data is consumed by calling a function you define named ***pop*** *k* times and simply printing the results in this format: *H 10 200* (This means that the player of the home team that has the shirt number of *10* has a score of *200*. There is a space between each value. The function *pop* returns and removes the entry at the top of the heap. Simply extract the maximum value *k* times and do not worry about players with the same score.).

See **exercise1_example_run.pdf** for an example run. Your outputs must be **exactly** the same. You can create your own *scores.csv* files for testing.

Exercise 2 Binary Search Tree (BST) (30 pts)

This exercise requires you to construct another binary tree consuming the same data as in the previous exercise. However, there are some differences:

- You need to implement a BST rather than a max heap: assign entries that are less than **or equal** to the parent as the left child, and the entries that are greater than the parent as the right child. For creating an empty BST, you can implement *create_bst* which may simply return a null pointer.
- BSTs are typically not complete trees. Thus, we will not implement a BST as an array. Instead, we will link "children" to their "parents" (as we do in linked lists). Please see the struct named *Node* and the function named *print_binary_tree*.
- There are three command line arguments: *path*, *s_min*, *s_max*. Rather than displaying *k* best players, this time display the players that have a score in the range $[s\_min, s\_max]$. (Display them one by one in the ascending order of scores. Simply print them as you traverse the tree without aggregating the values and sorting them separately! The algorithm roughly corresponds to *logarithmic-time search* of a node that is in the given range, followed by *in-order traversal* of the subtree using this node as the root. This subtree may contain entries outside the range. You can simply check each entry before displaying and skip if not in the range.[1]). Instead of the destructive function *pop*, we need a function (maybe named *query*) that does not change the tree and does not return anything but only prints the entries which are in the given range.

You are free to choose names for your functions. See **exercise2_example_run.pdf** for an example run. Again, your outputs must be **exactly** the same and you can create your own *scores.csv* files for testing.

Exercise 3 Hash Table (40 pts)

In this part of the homework, you are required to read a CSV file which describes significant events of a match ordered by their occurrence time. Each line contains a *minute* for indicating the minute in which the event occurred, a *team* and a *shirt number* which collectively indicate the player associated with the event, and an *event code* which indicates the event occurred. Please see the example *events.csv* for reference.

Use a **hash table** to keep track of goals, assists and scores of the players. For example, scoring a goal is worth 20 points. Refer to table 1 for all event types.

[1]It is an optional challenge for you to find an efficient way to skip these entries that are outside the given range. You are recommended to do this but it will not be graded with extra points.

Table 1: Event Types

Event Code	Event Description	Action	Contribution to Score
S	Scoring a goal	Increment number of goals.	20
A	Making an assist	Increment number of assists.	10
T	Tackling	-	1
P	Accurate passing	-	1
SH	Shooting on target	-	2
SG	Saving a goal	-	3
IP	Inaccurate passing	-	-1
YC	Receiving a yellow card	-	-10
RC	Receiving a red card	-	-20
L	Losing a ball	-	-2
CG	Conceding a goal	-	-5
...	Everything else	-	0

Your hash table must have 28 slots, which is the maximum number of players who can appear in a match. (Thus the hash table must be implemented as an **array of 28 elements**.) Players are identified by their team and shirt numbers. Thus, use this pair as a **key**. We want to keep track of number of goals, number of assists and a score for each player. Thus, these three variables constitute a **value** for the hash table. See *exercise3_example_run.pdf* for a visualization of an example hash table.

Define and use the following hash function:

$$hash(team, shirt\_number) = (f(team) + shirt\_no)\ mod\ 28$$

where

$$f(team) = \begin{cases} 0, & \text{if } team = H \\ 99, & \text{otherwise} \end{cases}$$

Apply **quadratic probing**[2] in case of collisions.

An integer $1 \leq m \leq 90$ will be given to your program as a command line argument to indicate a target minute for which you are required to construct the hash table. Refer to *exercise3_example_run.pdf* for an example run. Your outputs must be **exactly** the same.

[2]Quadratic probing is very similar to linear probing that you can see on the course slides. Only difference is that you need to add i^2 to the hash rather than i.