Data Structures & Algorithms 2

Mini-project Report
Football Tournament Management System (FTMS)

Abstract

This report intends to follow and explain the attached code and our approach for creating a Football Tournament Management System (FTMS) according to the directives stated in the project description document.It decomposes the problem at hand; namely the data management of a football tournament, describes the representation of the data chosen, clarifies how we tackled each aspect of the football tournament with its different relations and abstracted it to a fairly representative model, and compares the different approaches we ended up picking as the solution.

The report also goes over the main components of the C++ code non-rigorously, and it states clearly each team member's contribution in each part of the project from solution design, solution implementation, writing the report and other aspects.

Team Members:

- **Mohanned Derar (Team Leader)**
- **Arabet Abdelhakim**
- **Youcef Mhammdi Bouzina**
- **Mohamed Elamine Ali Zergaoui**

1.Introduction

With the constant increase of businesses and organizations relying ever more on data mining to obtain best performance at their respective fields, it is vital to maintain a robust and accessible way to retrieve and manipulate data however the end-user may require. It also should solve the set of all conceivable problems that may face it whether they are scaling issues, unsuitable data inputs and, most vitally, performance in a reasonable time.

The task handled in the included program is the management of a football tournament of the format where teams playing against each other in span of weeks and collect points depending on the outcome of each game, the team with highest points at the end of the tournament is considered the champion, with different stats we would like to access all of which are to be explained in the section 2.1 of the report. Although the dataset example provided in the program has a specific format, the management system is flexible and adapts tinkering with various parameters so that it can manage tournaments regardless of age categories, regions and male and female tournaments. It also accepts redefining the number of teams or weeks, and the start date of the season.

2.The Problem and the Data

2.1 Defining the Problem

In order to build the FTMS, one has to understand the problem and outline every requisite and limitation stated or implied by the project statement. And in this section, we aim to translate the problem description into a clear and well-defined set of coding goals that we aim to achieve thus forming the holistic management system at the end.

The system is mainly the relationship between its two components; the data which is the source that system extracts its information provided by the system manager, and the queries that perform operations and output the desired results. The data consists of first, *Teams* that form the football tournament, there are 20 teams for every season of the tournament, the same team may participate in different seasons or even all the seasons depending of its performance, each team consists of the staff member which are simplified to just include the president of the team holding an administrative role, and the coach responsible for the technical aspect of the team, in addition a team has 23 players which form the second part of our data. A *Player* is defined by his name and a number which corresponds to his position in the field, one player may be in different teams in different seasons. The third piece of information necessary for the management system is the *Games*. It keeps data about: the teams participating in it specifying the home and away teams, the referee, the date and the final score of the game, it also stores the game events, and we define a game event to be any of the following: each goal and the player scoring it at a specific game, warnings either yellow or red cards taken during the match with the associated player, the game duration which could vary depending on the added minutes but should never read below 90 minutes, the time that each player started at the game which would be zero if he was of the starting squad and the time he was replaced; the duration of the game if he was not, the distance run by each player in kilometers during that game. There are of course other events in the game, we choose only this set to be handled by the FTMS. Additionally, a player may join a team during or before the season and we would like to keep track about that. With this extra piece of information, our data structure is complete and will serve all of the operations performed by the management system.

The FTMS is expected to process the data and output any of the following desired results:

- Given an input of a date, a table displays the teams of the tournament of the matching season ordered by the points they acquired from the beginning of the season up until the specified date. The points are determined by the outcome of each game in the specified duration as previously explained. The table also includes other information about each team namely: the points count, both the number of goals scored and conceded and the arithmetic difference between them and the number of games played. The input could also be of the shape of a week number in which case the table displays all of the aforementioned information after all the games of the given week are done.
- A record of the goal scorers for a given number of weeks during a season listed from most to least number of goals, the player should score at least one goal to be a part of the list. This record also offers an option to display the cumulative curve of each player's number of goals for each week of the specified duration in the tournament.
- Given an input of a team and a game, a list of the distance run by each player of the team in kilometers during that game is displayed. In addition, the average distance of all games preceding a given date.
- A listing of the best scoring players of the entire tournament from the very first season to the latest one to date is displayed, it could be for one team or for all the competing teams. The players are ordered by their number of goals.

To store and manage all of this data, it is necessary to choose the most suitable set of combinations of abstract data types (ADTs) and data structures tailored specifically for our purposes to ensure the best performance out of the data we have. The FTMS provides two variations of data storage and access choices, *The Binary Search Tree* and *The AVL Tree* each of which would make use of the different ADTs we see appropriate. All of the design choices are to be discussed rigorously in the Solution Design section. The two versions of the system would be tested with various inputs on all the operations it is expected to perform, and a one-on-one comparison between their running time is presented in the Results section of the report.

2.2 The Data Chosen

The FTMS, as brought up earlier, could manage a lot of different tournaments with different structures. Of course, the general layout of a tournament played during a number of weeks and determined by games' outcomes should be maintained, but other aspects of it like the tournament's duration are amenable to modification and the system is left untroubled handling them. The dataset example for the FTMS we chose is based on the English football tournament the Premier League due to the good documentation of this tournament from various sources. The data instances concerning the teams and the players are from the team of *FBREF* belonging to the online database *Sports Reference*. The games are the documentation from the online schema *england.db.* We made sure both sources are public domain and are open to use for our purposes.

The data was selected so that it mirrors the problem definition and accomplishes its goals. Although the structure for each of the data chosen is to be discussed in the Solution and Results section of the report, it needs to be mentioned broadly in our attempt to explain the data instances. Our dataset keeps information about a generation of players stretching between the seasons 2000-2001 and 2019-2020, which we found to be a suitable size for simulating the general behavior of the FTMS. Each player has a name, an id and position on the field, other information such as the number of goals are added during running time for each player to avoid inconsistencies inside the program. The tournament has 20 teams where each team has an id, a name and a staff consisting of the team's coach and president, both of which were assumed to stay the same throughout the seasons for sake of simplicity although it is of course not the case in actual

tournaments. Each team also has players that change over the seasons. The average week of the tournament has ten games where each team plays either as an away or home team, but games could be postponed to another week for various reasons so weeks of nine or eleven games do exist. Along the home and away teams, each game also keeps track of the following information: the week where the game took place, the date, the start time of the match, the referee, the duration, the final score and the scorers for each team, the warnings taken during the game associated with the players , the substitutions and the distance run by each player in each team.

Some of the aforementioned data were not tracked in the dataset we used for this example so we relied on statistics in crafting programs to get as close of an accurate representation as we could have. For example, the running distance for the players was not a part of the dataset. In order to simulate it we created a program that randomly generates a distance in kilometers for each player where the stats for our simulated data match the actual numbers (the average distance, the dispersion parameters…), we respected of course each positions' specifics so a Goalkeeper does not run as much as a Centre Midfielder.

2.3 Data Format

Once the dataset has had its logical layout, we had to choose the data format to incorporate it into the FTMS later. The nlohmann json library was the pick for its various features in working with C++ programs, specifically the to json and from json functions which guarantee a quick copy from a json object to a C++ one in very few lines.

3. Solution and Results

After the problem being appropriately outlined in the previous section, a comprehensive solution has been constructed to address not only the pointed-out objectives but also additional improvements. With us ensuring the suggested solution is sustainable, scalable and free from any conceivable issue. In this section we will describe our solution approach to the problem as a C++ program along with results from various tests performed on the FTMS.

3.1 The Solution Design

The program translates all of the problem's aspects carefully deconstructed above into a program. First, the data is represented by making use of the OOP aspects of C++. The players, the teams and the games are instances of classes inside the program each of which with its appropriate attributes and member function to allow the manipulation of the attributes. Along with these three classes, we added the additional class of *Date* that represents the date attribute in the Game class is an object, this class with its various operations enables us to compare the games based on the date this game was played, the relevance of this comparison will become clear later in this section as we discuss the tree design of our solution. All of the relevant aspects of the classes mentioned above are discussed one by one.

3.1.1 Player Class:

The class is just really the implementation of the already explained, so we're keeping it short to free the report from repeating itself. The class has the name of the player, his id, number of goals and position on the field. It has its getters and setters to allow manipulation of these attributes and functions to convert the json formatted data to a class player instance. Additionally to all of this we figured to add an additional attribute *Stats* which is an array containing the number of goals, the team the player has played for in a given season, we found it helpful from a design standpoint and this choice would be explained later in the section when discussing queries. These stats and the number of goals are updated from the games, so that each time a game is inserted to the tree the number of goals of that season and the global one are increased accordingly, This is done rather than just another set of data to avoid inconsistencies inside the program.

3.1.2 Team Class:

The team class has the attributes id, name, coach and president along with an array of players for each season. This array tracks the players that played for the team during each season and what number each player wore in that season. It also possesses member functions for the various operations needed of that class including the conversion from json functions.

3.1.3 Game Class:

Football Management System's Game class, serves as a crucial component encapsulating all information and operations related to a football game. The class, emphasizing encapsulation, includes details like home and away teams, game date, start time, referee, and duration. Utilizing vectors, tuples, and pairs, we efficiently stored and manipulated game data, including scores, goal scorers, cards, substitutions, and player distances. Several member functions were implemented for data manipulation, covering actions like adding goals, issuing cards, and making substitutions. Furthermore, we implemented operator overloading for comparison (>, <, ==), facilitating easy sorting and equality checks of Game objects . Team management functions were added to the Game class, offering centralized and efficient handling of teams. These static functions allow the addition and retrieval of teams by ID or name, ensuring consistency across different games, independence from Game instances, and increased code efficiency. The class also includes the json conversion functions.

3.1.4 Date Class:

The Date class encapsulates a date with attributes for the year, month, and day, designed for encapsulation. It includes a parameterized constructor, copy constructor, and destructor. Getter and setter methods (GetYear, GetMonth, GetDay; SetYear, SetMonth, SetDay) provide access and modification. Operator overloading facilitates various operations, and utility methods like Print and GetMonthDay are implemented for human-readable display and day calculation, respectively.

The data now can be appropriately converted from the json format previously explained to an instance of its respective C++ class thanks to careful crafting of the classes and utility functions that allows for. The players and the teams are stored now on a static vector that allows access to each player or team via their id.

Now, for easy access to our games so that the FTMS displays its expected results, the most appropriate data structure is the binary search tree where the nodes of the tree are the games sorted according to the date of the game from the first to last date, here comes the date class with its overloaded operations to offer us an easy way to compare between the different games. All the queries emerge as just functions

traversing over the tree and performing a range of operations that differ from a query to the other. All the queries are going to have their logic explored.

3.1.5 Display The Standing:

The query displays the standing table of the tournament on a given point of time. The standings display the teams sorted according to their points gained from the season's beginning until the date that is hoped for, along with other stats. To do that, the query takes the season and the date of which the standings are wished to be displayed, then for every team in the given season a new struct is created containing the attributes that are to be displayed, namely: the number of games played, wins, losses, draws and goal difference, and all new structs are stored on a new temporary vectors. Then the BST is then traversed from the game corresponding to the first date of the season until the game played before or at the date input, and on each game traversed the following procedures are executed: the home team and away team are accessed in this new vector and extracts the data of the game to update their attributes; the number of games is increased by one, the number of wins, losses or draws is updated depending on the outcome of the game (with obvious logic where each case takes place), the goal difference is updated by adding the number of goals scored and subtract the number of goals conceded . And then the number of points is increased by 3 in the case of a win, by 1 in case of a draw and 0 in case of a loss. After the last game is visited, the temporary array is sorted from the team with most points to the one with least. Then a simple display of the vectors' content is carried out.

3.1.6 Display Best Scorers:

This query is made out of two segments. First it displays the best scorers in the tournament for a given season and date. Just like the first query it takes an input of the season and the date. It then traverses the BST from the first game of the season until the game played before or at the same date given, then for every player that scores in any game it creates an instance of a struct containing the player and the number of goals in that season, with each game he scores in the number of goals gets updated. At the end all of the players that scored get sorted by the number of goals they scored that season, and get displayed. For the second query, a graph of goals of a given player against the week of a season is displayed. The query takes an input of player and season and then traverses over the game tree plotting the goals scored in a certain week for that player as a straight line and updating it with each week the player scores accordingly. This query makes use of the library pbPlots.

3.1.7 Display Kilometers:

This query displays the number of kilometers run by the players of a given team in a given game. With a simple access to the game and the team that are given as inputs, each player is then displayed along with his distance run.

3.1.8 Display Best Scorers Historically :

This query is also of two parts, and since it is independent of time it does not make use of the games' tree. The first query displays the best historic scorers in the league for a given team, it takes an input of the team then creates a struct of attributes player and the goals scored for that team it, then traverses the vector containing all players, and within each player it goes over his stats until a season where he played for the targeted team is found. It creates a new instance of the mentioned above struct with the player, takes the number of goals scored in that season and assigns it to the number of goals corresponding to his instance of the struct, this new instance is then pushed into a vector containing the all players that played

for that time. If the player plays for that team for more than once, only the number of goals is updated and no new instance is inserted into the vector. After that the players are sorted inside the vector based on their number of goals with that team, and get displayed. The second query is simpler since it displays the best scorers of the entire tournament, and each player has his number of goals stored in one variable, with a simple sort according to the number of goals and a display the query is done.

This finishes the FTMS, the program includes some additional features included in here:

3.1.9 Bouncing Ball Simulation with ASCII Rendering in Terminal:

This code simulates a bouncing ball with ASCII rendering in the terminal. It utilizes the linear algebra library "la" from the GitHub repository . The libraries include assert.h, stdio.h, string.h, unistd.h, thread, iostream, conio.h, vector, and Windows.h, along with some constants and global variables like WIDTH, HEIGHT, and the display array. Key functions include fill(Pixel p), circle(V2f c, float r), show(), back(), reset(), and runthBall(). The simulation constants involve frames per second (FPS), ball radius (RADIUS), gravity (GRAVITY), time delta (DT), and damping factor (DAMPER). The main execution occurs in the runthBall() function, running the ball simulation in a separate thread, continuously updating position and velocity, checking for boundaries, and redrawing the display until stopThreads is true.

3.1.9.1 FTMS User Interface Overview:

This multithreaded application combines a soccer statistics menu with a bouncing ball simulation using C++ Standard Library's threading capabilities(references). Key libraries include iostream, thread, atomic, and conio.h. Global variables include a global atomic boolean, stopThreads, controlling thread execution. Functions such as displayMenu, processSelection, runQueries, and MainProgram::run drive the application, creating threads for menu and simulation, waiting for their completion.

3.2 Results

The FTMS is now ready to perform its functionalities. To evaluate both the BST and AVLT versions of the system, a series of tests has been carried out to assess every operation’s performance. We try to compare the theoretical complexity of our queries to the actual running time.

Both the BST and AVL versions of the FTMS take next to no time processing the data and displaying. Where the insertion of the games took slightly more time in the AVL due to its nature, the access time is also slightly less in the AVL compared to the BST. The BST averaged 2 ms in insertion and AVL a 1.5 ms.

4. Code Guidelines

4.1 Program Requirements:

- g++ Compiler Integration:
  Ensure g++ compiler is integrated for code translation.
- Library Download:
  Get the required libraries for the program to work properly .
- Configuration Setup:

Configure development environment for paths and variables.
- Version Compatibility:
Verify compatibility of g++ compiler and libraries with program version.

4.2 Running the Program:
There are two methods available for utilizing the football management system program:
1. Direct Execution from Sent Files:
- Run the program directly from the provided files. In case of compilation errors or issues, consider the second option.
2. Code Blocks Project Download:
- Utilize the Code Blocks project, accessible for download in the references, to ensure a smoother program execution, follow these steps:
1. Open the Code Blocks program.
2. Click on "Open an Existing Project" and select the specified file.
3. Choose the ".cbp" file and proceed to build the code.
4. Run the program after successful code compilation.

4.3 Program Interaction:

Navigating through the program is made simple and intuitive through the following interaction instructions:

1.Navigation:
- Utilize the up and down arrow keys to effortlessly move through the menu options. This allows for a seamless exploration of the various features and functionalities the program has to offer.
2.Selection:
- Press the Enter key to make selections and confirm your chosen options. This streamlined approach ensures a quick and efficient process when choosing preferences or initiating specific actions within the program.
3.Go Back:
- Should you wish to backtrack or cancel a particular action, simply press the left arrow key. This feature ensures flexibility in navigating the program, allowing users to reconsider and modify their choices easily.This is available until you reach the main menu.
4.Exit:
- When you're ready to conclude your interaction with the program, opt for the "Quit" option. This straightforward exit strategy ensures a clean and orderly closure, maintaining the overall user-friendly experience.

These interaction guidelines are designed to enhance user control and ease of use, promoting a positive and efficient experience while engaging with the program.

4.4 Inserting Date via Terminal:

1. Introduction: The `getHiddenInput` function securely captures date input without displaying characters. Uses `_getch()` for character capture.

2. Purpose: Ensures secure entry of "DD-MM-YYYY" format with visual feedback for each digit entered.

3. Input Handling:Allows only numeric input, ignores non-digits, and supports backspace for corrections.

4. Formatting Input: Inserts hyphens as digits are entered to create formatted date input.

5. Date Extraction: Extracts day, month, and year, converts to integers, and creates a `Date` object.

6. Return Type: Returns a `Date` object encapsulating the date information.

5. Work Distribution

Mohanned Derar:

- Design and implementation of the query number 1 explained in section 3.1.5
- Design and implementation of the query number 2 explained in section 3.1.6
- Design and implementation of the query number 3 explained in section 3.1.7
- Writing sections of the report: **3.1.5 / 3.1.6 / 3.1.7**
- Generating and crafting the data for the FTMS

Arabet Abdelhakim:

- Design and implementation of the class player.
- Design and implementation of the class team.
- Design and implementation of the query number 4 explained in section 3.1.8
- Writing sections of the report: **1 / 2.1 / 2.2 / 2.3 / 3.1.1 / 3.1.2 / 3.1.8**

Youcef Mhammdi Bouzina:

- Design and implementation of the class game.
- Design and implementation of the class date.
- Create and implement the interface, along with its associated functionalities:
  Arrow key moving .
  Insert the date using a technique that avoids echoing.
  The bouncing ball
  The coloring of the selected choices
- Writing sections of the report : **3.1.3 / 3.1.4 / 3.1.9 / 3.1.9.1 / 4.1 / 4.2 / 4.3 / 4.4**.

Mohamed Elamine Ali Zergaoui:

- Design and implementation of the Binary Search Tree and the AVL tree
- Integration of the json data into the C++ program by designing all the the json functions
- Writing section of the report: **3.2 Results**

Appendix A

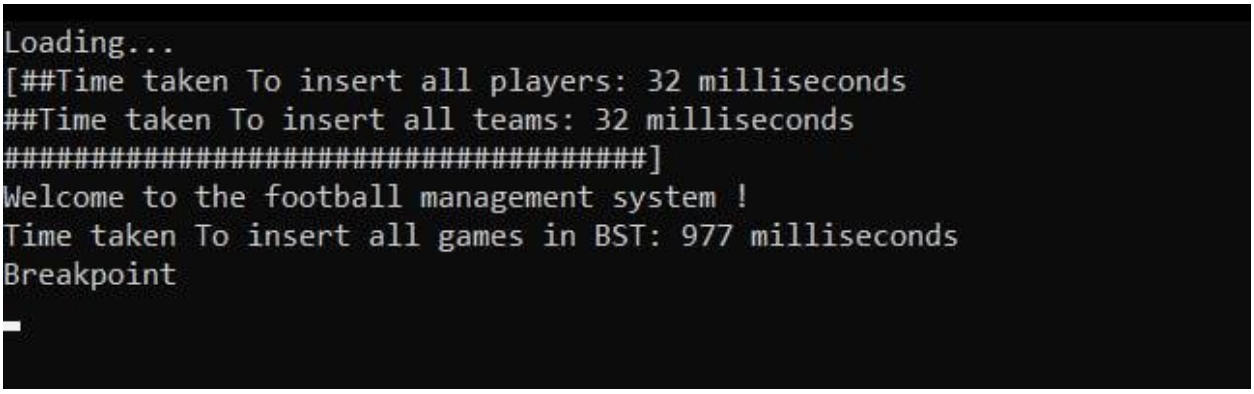

Figure 1. Insertion time in Binary Search Tree

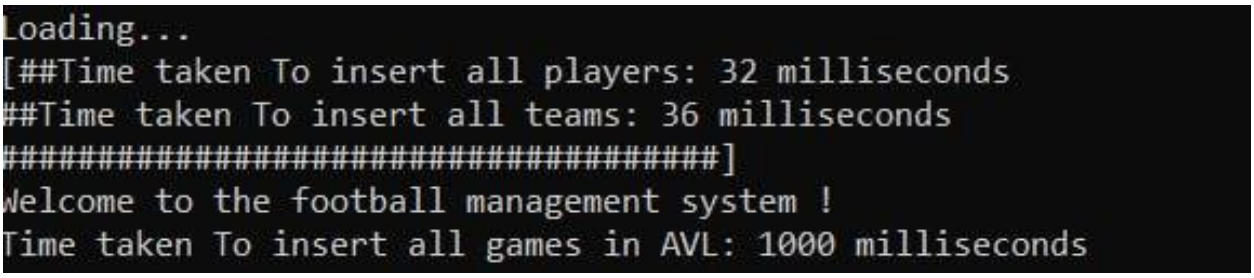

Figure 2. Insertion time in AVL

References

1. Sports Reference: an online database that collects data and statistics about various sports, FBREF is their branch documenting the statistics of football. Their website: https://fbref.com/en/?__hstc=213859787.e649ec443349b31a788dbd0b379ffc60.1703336610156.1703336610156.1703336610156.1&__hssc=213859787.3.1703336610156&__hsfp=1853282402

2.england.db: a github repository that is a work from several online contributors. The repository: https://github.com/openfootball/england/tree/master

3. nlohmann json library: their github repository; https://github.com/nlohmann/json

4. pbPlots: a plotting library available in many programming languages. The github repository: https://github.com/InductiveComputerScience/pbPlots/tree/v0.1.9.1

5.Linear algebra lib: https://github.com/tsoding/la

6.Thread in cpp : https://en.cppreference.com/w/cpp/thread/thread

7.the alternative codeblocks code : https://drive.google.com/file/d/1lIqXICDvniqqzfsCT1D8QuwgcdXlQutP/view?usp=sharing