FIBA Player Statistics Mock-up Application

Arturo Díaz, David Montaño, and Samuel Hernández

ITC Department, ICESI University

09687: Algorithms and Data Structures

Johnatan Garzón Montesdeoca

FIBA Player Statistics Mock-up Application: Engineering Method

Basketball is a worldwide known game, and it has a rich history of player, rules, and events. And even though the essence of the game itself has remained untouched, several rules and traditions have been added, modified, or removed from the game altogether. This evolutionary trend of the game has made necessary a broader reach in the data it produces, including more and more details into them, which is why a close follow-up of this data is a real necessity by institutions and organisms that regulate and promote the sport. Under this premise, we have been tasked with the development of a desktop app that is able to showcase the storage, management, and retrieval of this data, by the International Basketball Federation Association, or FIBA for short. Next up, an engineering approach to solve the problem using the Engineering Method.

Context of the Problem

FIBA requires a desktop application that can handle worldwide basketball players' statistics, including management, retrieval, and adding of said statistics and players. The search and storage of data must be fast and efficient.

Development of the Solution

Based on the description of the engineering method given in the book Introduction to Engineering by Paul Wright, the following flowchart was drawn, and will be followed according to the steps shown in it during the development of the solution.

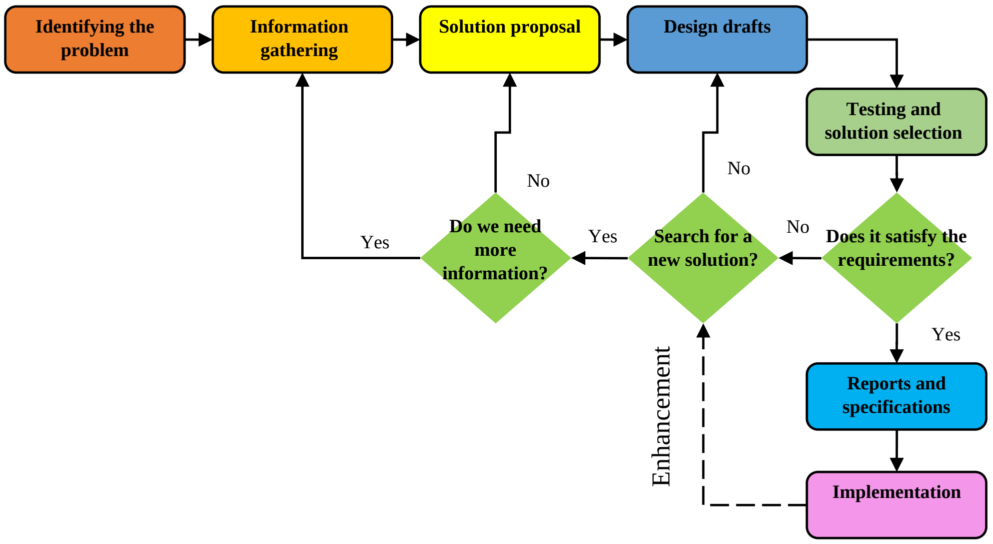

Identifying the problem
Information gathering
Solution proposal
Design drafts
Testing and solution selection
Does it satisfy the requirements?
Yes
No
Reports and specifications
Implementation
Enhancement
Search for a new solution?
No
Yes
Do we need more information?
No
Yes

Figure 1. Flowchart representing The Engineering Method proposed by Paul Wright.

The steps shown in figure 1 are elaborated in detail following up.

Identifying the Problem

Identifying symptoms and necessities

- The app should be able to show each player's most relevant information
- The app should be able to quickly fetch and store data from search queries
- The app should be able to store large amount of data in one action effectively timewise and memory-wise
- The app should be GUI based, but should also be able to take file input such as CSV files

Definition of the Problem

FIBA requires a desktop GUI based application to effectively store large amount of basketball player related statistics data, and allow its management, sorting, and searching; being able to take GUI and file input of large quantities.

Information Gathering

Given the technological context of the solutions to be proposed, the following terms must be defined prior to anything:

Software: Software is every application program and operative system that allow the computer can run smart tasks, directing the physical components or hardware with instructions and data through several different kinds of programs.

Simulation Software: A simulation software has the objective of facilitating or automating the modelling process for a real-world phenomenon, using mathematical formulas through programming. At its core, it is a program that allows the user to see what will happen after doing a specific action or set of actions, without having to do it in the real world.

Graphical User Interface (GUI): A graphical user interface (GUI by its English acronym) is a program or environment that manages the interaction with the user basing itself on visual relations such as icons, menus, or pointers.

Now, we need to define the terms relevant to the context of the problem: Basketball. Since the application requires to show the most relevant statistics for each player, we need to first understand and delimit which are these aforementioned stats. Apart from name, team, position, and age; project *Five Thirty Eight* breaks each player's stats into 5 major categories: vitals, corresponding to weight, height, and the draft position (on the year they were drafted); scoring, divided into True Shooting Percentage, Free Throw Percentage, and Usage Percentage; Tendencies, which correspond to the frequency of 3-point and free throws; ball handling, which are the assists and turnover percentages; and defence and rebounding, broken into the rebound, block, and steal percentages, and a defensive +/- number, corresponding to their 2019 RAPTOR model (FiveThirtyEight, 2021).

Additionally, the model they use to rate the players also provides a category for the player (the exact way the categories is rather loosely understood by us at the time of writing), divided from our understanding as such: *prospect* (good and OK), *starters* (good and average), and several special categories consisting in *all-star*, *Future all*-star, *MVP-candidate*, *key role player*, *up-and-comer*, and finally, *scrappy-veteran*. The metric they use to weight the averages in the previous paragraph depends on whether or not the player is a prospect, and therefore a rookie, in which case an "Adjusted College Statistics" measurement is used. Be the player not a rookie, then a weighted average is used taking the last 3 seasons into account (FiveThirtyEight, 2021).

Several other useful statistics are listed and defined in the website *Basketball References*, as well as their respective formulas and abbreviations (Sports Reference LLC, n.d.). With this in mind, the following terms are deemed as relevant to define for the problem:

Prospect: A prospect is a college basketball player that wishes to join the NBA league through an official event called NBA Draft. Every prospect is considered a rookie, and they can be either OK or good, according to their Draft position and other useful metrics.

Starter: In basketball, a starter is a player that usually plays first, i.e. is part of the initial roster of players. A starter can be either average or good, with the extra classification of key-role-player, a player fundamental to the synergy of the team and favourable development of the game.

Veteran: A veteran is a player that has signed at least one contract out with an NBA team, it generally takes 2 – 3 years playing to reach this state. A veteran can be an MVP-candidate, which stands for Most Valuable Player candidate, and is honoured to players who are great key role players and move their team forward and up the charts.

All-Star: An All-Star player is part of the top players of a team and has been the star in any other team on their career.

Solution Proposal

Proposal 1: A CLI (Command Line Interface) framework would work best since it's a pretty efficient way to take and receive user input, especially if it is a large amount of data, like in the case of the problem. The app would be sequential and navigable through different menus. Firstly, the app needs a way to take the players' and teams' data, and in this case a comma separated values (CSV) file with a standard format would be appropriate. A single line would represent a player's or team's data, and the app would break each line by a standardized separator (this would mean the separator couldn't be contained within the data itself, or else the process would break). Inputting data by CSV files is an optimal solution for large chunks of data that share a similar or equal organizational structure, however for the occasional requirement of taking small input, a menu would be better. This menu would ask for a line containing the player or team info – as well as specifying which of the two cases applies – and then process it in a similar fashion to the way it processes CSV files. There should also be options for deleting, editing, and searching for players and teams in the database, which can all be done through simple text menus accessible through the command line.

Proposal 2: A GUI framework would be the best fitted option for both large and short inputs of data. A GUI based app would also be optimal for displaying information, since it can be easily visualized with sortable tables, charts and graphs, and information snippets. A simple input interface for single players would include a field of some kind for each relevant statistic, basic information of the player, and an optional photo. This solution would also allow the input of mass data (with no image URLs compatibility) input and removal. The former, through CSV files with each row representing a new player or team, and the latter through multiple selection of sortable and filterable tables. The app would separate each of its functions into menus and submenus, allowing to separate creation, management, and visualization of player and team data, both of which will be stored in some variation of binary search trees to maximize search and removal efficiency.

Proposal 3: A GUI based application that allows batch and single inputs of new data all through its interface. The app would follow a specific flow of operations, so that the exact result is always obtained. It can create and store data of players and teams in hash tables to ease its input and management, and the GUI will include lists or tables to visualize and manage said data easily.

Solution Selection

Starting with the first proposal, a major flaw is that the requirements specify that it should be a GUI based application, which means that the idea of text-based command line interface is discarded. Even if this limitation did not exist, were we to implement graphing in the future as a feature of the app (a very reasonable upgrade given the app manages a lot of data), the solution becomes suboptimal and cumbersome to implement this, which is why we discard it.

The third solution solves this by being a GUI based solution but fails in excluding the input through CLI or CSV files, since that’s the easiest solution for large amounts of data. Plus, it was proposed that the data is stored in hash tables, and although this is a very fast data structure, there are structures which are not only faster but neater in organizing the data they contain, and with the amount of data the app is set to hold, collisions are expected to increment greatly in any sort of hash table. Lastly, since the solution is expected to follow a very specific flow, nothing can grant that omitting any given step of this flow won’t result in errors. A restrained app like this makes errors more prone to happen, since a lot of responsibility is given to the user.

This leaves us with a single solution, which takes advantage of both the ease of GUI apps and the capacity and speed of CLI apps to result in an app that is both easy to use and adapted to take in heaps of data effectively. This app would also be fast since it would use BSTs to store and sort data, also possibly with the help of heaps. This is the solution that will be further developed.

Design Drafts

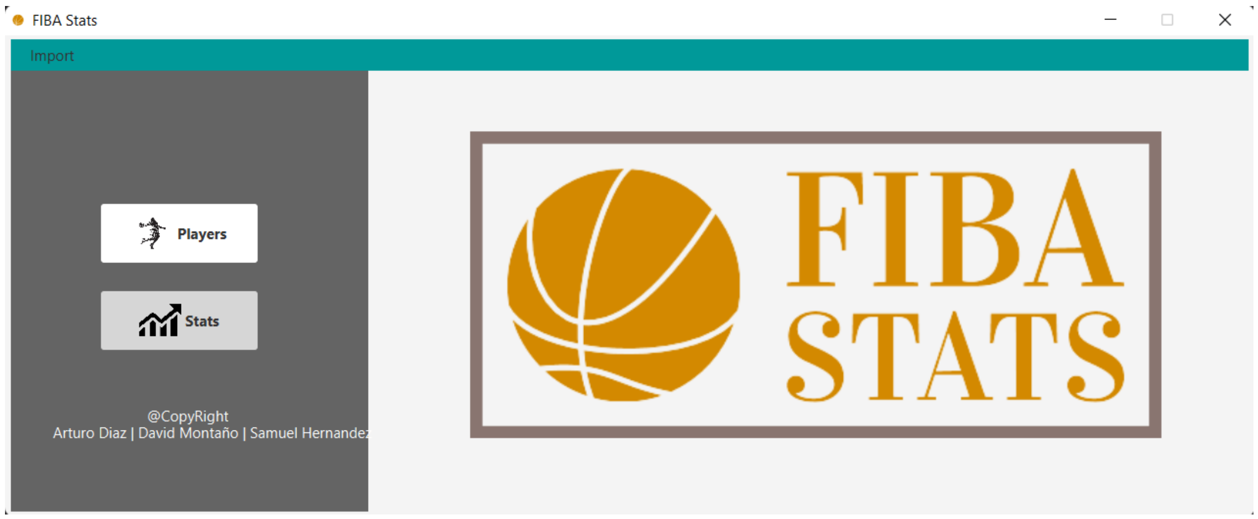


Figure 2. Main View of the software

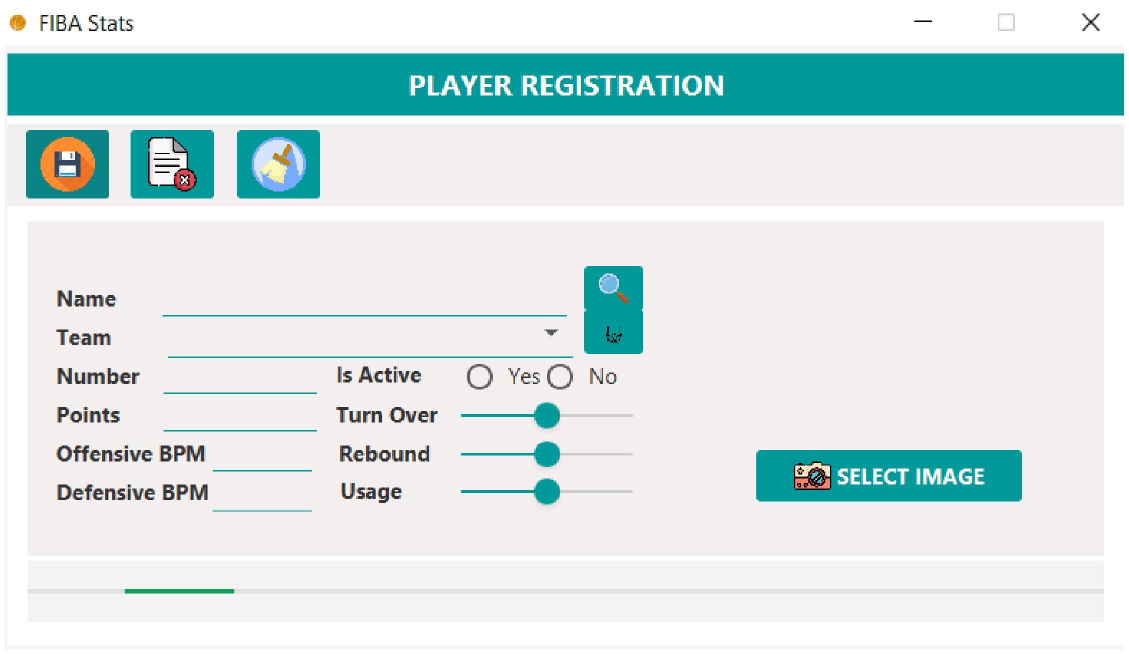
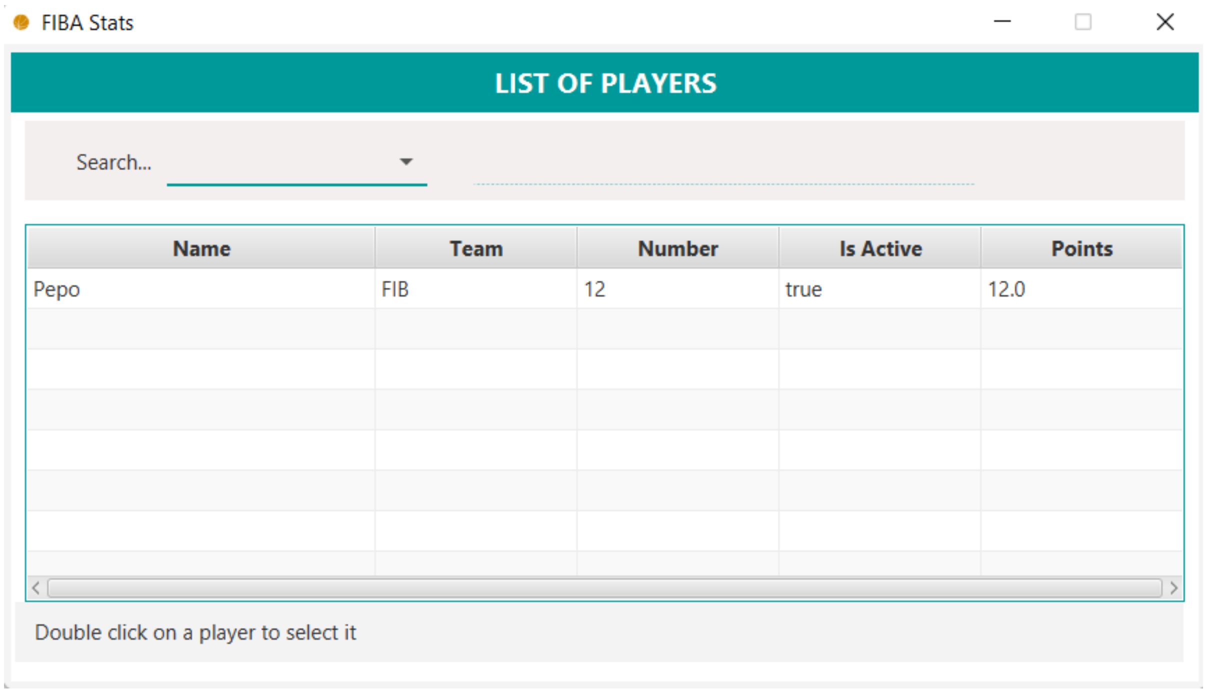
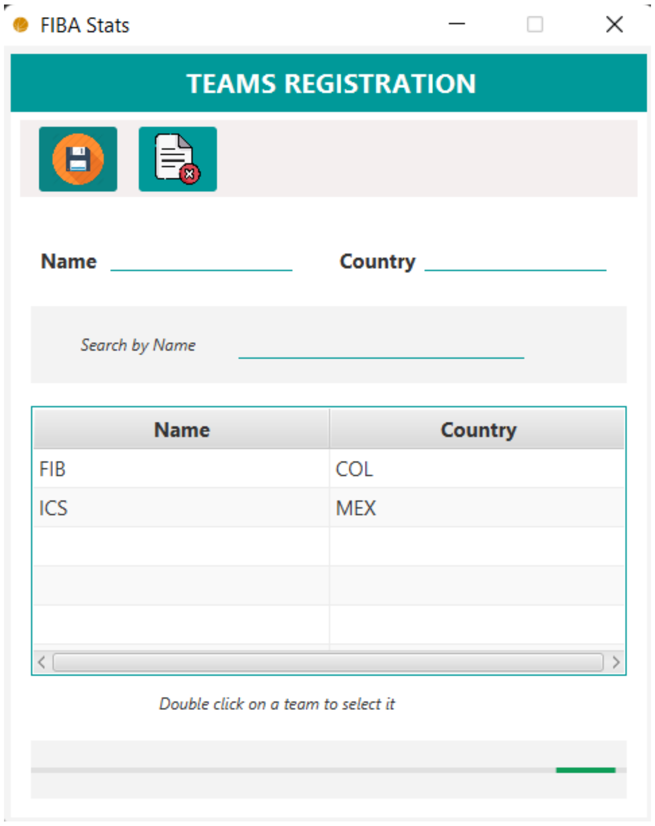


Figure 3. Player registration View



Name	Team	Number	Is Active	Points
Pepo	FIB	12	true	12.0

Figure 4. List of Players



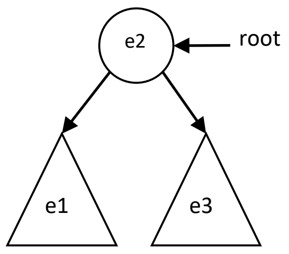
Name	Country
FIB	COL
ICS	MEX

Figure 5. Team registration view

Abstract Data Types

This project requires the usage of some abstract data structures to properly model the problem it attempts to solve. Namely, Binary Search Trees (BST), Self-Balancing Trees (AVL), and Red Black Trees (RBT), all of which will be shortly characterized below, taking into account AVLs and RBTs inherit from BSTs.

Binary Search Tree (BST) ADT



$$BST = \ll e_1, e_2, e_3, \dots, e_{n-1}, e_n, e_{n+1} \gg, root >$$

$\{inv\colon \forall n\ (e_{n-1} \wedge key[e_{n-1}] \leq key[e_n]) \vee (e_{n+1} \wedge key[e_{n+1}] \geq key[e_n])\}$

$BST \rightarrow BST$

$searh\colon\ BST\ x\ Key \rightarrow Element$

$insert\colon\ BST\ x\ Element \rightarrow BST$

$delete\colon BST\ x\ Element \rightarrow BST$

$min\colon BST \rightarrow Element$

$max\colon BST \rightarrow Element$

$successor\colon BST\ x\ Element \rightarrow Element$

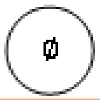
$predecessor\colon BST\ x\ Element \rightarrow Element$

BST

Builds an empty Tree

$\{pre\colon\ null\}$

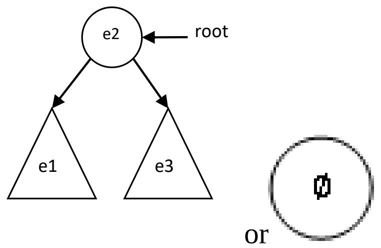
$\{post\colon\ BST\ t = \emptyset\}$



search

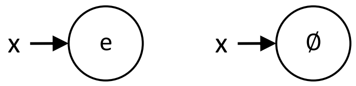
Retrieves element ***e*** from tree ***t***

$\{pre\colon\ (BST\ t = < e_1, e_2, e_3, \dots, e_{n-1}, e_n, e_{n+1} > \wedge\ e) \vee (t = \emptyset \wedge e)\}$



or

$\{\boldsymbol{post}: [(\boldsymbol{t} = \boldsymbol{t} \wedge \boldsymbol{x} = \boldsymbol{e}) \wedge (\boldsymbol{e} \ni \boldsymbol{t})] \vee (\boldsymbol{x} = \emptyset)\}$

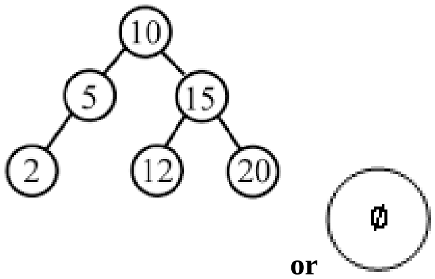


or

insert

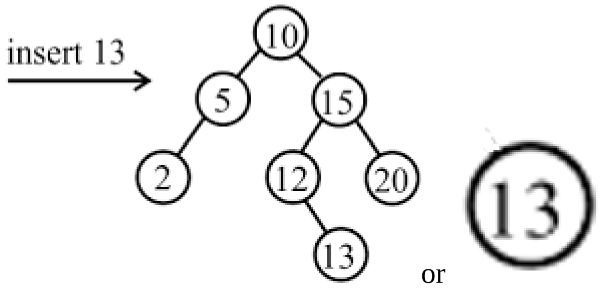
Adds element $\boldsymbol{e}$ to tree $\boldsymbol{t}$.

$\{\boldsymbol{pre}: \ (\boldsymbol{BST\ t} = < \boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \dots, \boldsymbol{e_{n-1}}, \boldsymbol{e_n}, \boldsymbol{e_{n+1}} > \wedge\ \boldsymbol{e}) \vee (\boldsymbol{t} = \emptyset \wedge \boldsymbol{e})\}$



or

$\{\boldsymbol{post}: \ \boldsymbol{BST\ t} = < \boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \dots, \boldsymbol{e_{n-1}}, \boldsymbol{e_n}, \boldsymbol{e_{n+1}}, \boldsymbol{e} > \vee\ \boldsymbol{t} =< \boldsymbol{e} >\}$

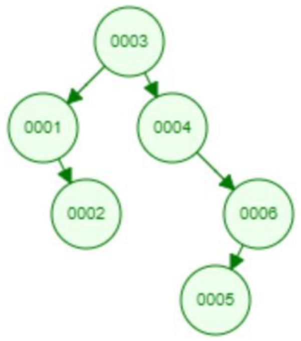


or

delete

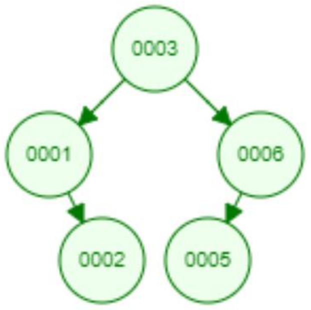
Removes element $\boldsymbol{e}$ from BST t.

$\{\boldsymbol{pre}: \ \boldsymbol{BST\ t} \neq \emptyset\ \boldsymbol{i.e.}\ \ \boldsymbol{t} = < \boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \dots, \boldsymbol{e_{n-1}}, \boldsymbol{e_n}, \boldsymbol{e_{n+1}} > \}$



$\{\boldsymbol{post}\text{: } \boldsymbol{BST\ t} = <\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \dots, \boldsymbol{e_{n-1}}, \boldsymbol{e_n}, \boldsymbol{e_{n+1}}>\}$

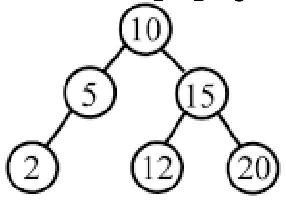
Delete 4



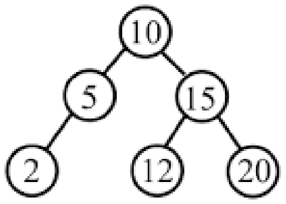
min

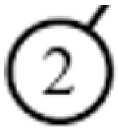
Fetches minimum element $\boldsymbol{e}$ from tree $\boldsymbol{t}$.

$\{\boldsymbol{pre}\text{: } \boldsymbol{BST\ t} \neq \emptyset\ \boldsymbol{i.e.}\ \ \boldsymbol{t} = <\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \dots, \boldsymbol{e_{n-1}}, \boldsymbol{e_n}, \boldsymbol{e_{n+1}}>\}$



$\{\boldsymbol{post}\text{: } \boldsymbol{BST\ t} = <\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}, \dots, \boldsymbol{e_{n-1}}, \boldsymbol{e_n}, \boldsymbol{e_{n+1}}> \wedge\ \boldsymbol{e}\}$

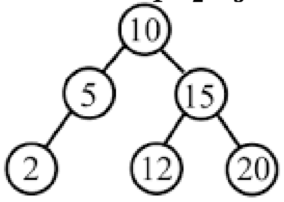


and 

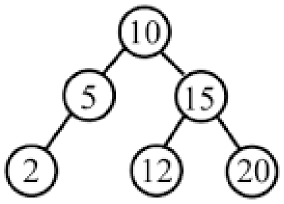
max

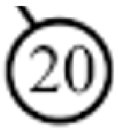
Fetches maximum element $\boldsymbol{e}$ from tree $\boldsymbol{t}$.

$\{\boldsymbol{pre}\text{: } \boldsymbol{BST\ t \neq \emptyset\ i.e.\ t = < e_1, e_2, e_3, \ldots, e_{n-1}, e_n, e_{n+1} >}\}$



$\{\boldsymbol{post}\text{: } \boldsymbol{BST\ t = < e_1, e_2, e_3, \ldots, e_{n-1}, e_n, e_{n+1} > \wedge\ e}\}$



and 

Bibliography

FIBA Central Board. (2021, March 26). *Document Library*. Retrieved September 29, 2021, from FIBA.basketball: http://www.fiba.basketball/documents

FiveThirtyEight. (2021, August 30). *2021-22 NBA Player Projections*. Retrieved from FiftyThirtyEight: https://projects.fivethirtyeight.com/2022-nba-player-projections/

Hagness, M. (2021). *The Most Important Stats To Track For Your Basketball Team*. Retrieved from Basketball Breaktrough: https://www.breakthroughbasketball.com/stats/how-we-use-stats-Hagness.html

Leadoff Digital. (2015). *NBA Dictionary*. Retrieved from Sporting Charts: https://www.sportingcharts.com/NBA/dictionary/

Sports Reference LLC. (n.d.). *Glossary*. Retrieved from Basketball-Reference.com - Basketball Statistics and History: https://www.basketball-reference.com/about/glossary.html