

Tarea Integradora II



Daniela Olarte Borja - A00368359

Juan Jacobo Garcia Aristizabal - A00368502

Santiago Arévalo Valencia - A00368494

Alexander Sánchez Sánchez- A00368238

Programa de Ingeniería de Sistemas

Facultad de Ingeniería

Periodo II de 2021

Universidad Icesi

Cali, Colombia

INDEX

FUNCTIONAL REQUIREMENTS	3
NON FUNCTIONAL REQUIREMENTS	4
ENGINEERING METHOD	5
TEST DESIGN	11

FUNCTIONAL REQUIREMENTS

- R1.** Enter data, either in bulk or through an interface.
- R2.** Delete or modify data.
- R3.** Make player queries using the statistical categories included as search criteria
- R4.** Save the most relevant data of each of the basketball professionals on the planet like the following items: name, age, team, and 5 statistical items points per game, rebounds per game, assists per game, steals per game, blocks per game.
- R5.** Retrieve players according to the selected search category and the value given for it.
- R6.** Millions of data can be stored.
- R7.** The importance of ensuring rapid access to data should be highlighted.
- R8.** Should show the time it takes to make a query.
- R9.** Will allow the customer to search on two statistical criteria using ABB as the structure for index management.

NON FUNCTIONAL REQUIREMENTS

R1.Efficient search for categories of 4 statistical items, through indices.

R2. Two types of query must be with AVL trees and the other two with trees red and black. Where each tree will save the value of the index (attribute) and the position of this data on the disk.

R3. There must be two queries for unbalanced binary trees. This means that one query should be performed by AVL trees and by ABB and the other by trees red and black and ABB.

R4. The complexity cannot be linear, only in one search.

R5. Handling large software: The program must contain at least 200,000 data valid on players.

ENGINEERING METHOD

1. Identification of the problem

FIBA is the regulatory body for basketball worldwide. Given a large number of players, FIBA has decided to collect the most relevant data of each professional in order to identify patterns according to a criteria to get an idea of where the sport is currently heading.

2. The collection of the necessary information

Binary Search Tree (BST): Is based on the property that keys that are less than the parent are in the left subtree, and keys that are greater than the parent are in the right subtree.

Transversal In-order: The traversal in In-order consists of traversing the left subtree in Inorder, then the data of the root node is examined, and finally the right subtree in In-order is registered.

Source:

<http://profesores.elo.utfsm.cl/~agv/elo320/01and02/dataStructures/binarySearchTree.pdf>

Self-Balancing Binary Search Tree (AVL) : Also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

Following are the conditions for a height-balanced binary tree:

1. Difference between the left and the right subtree for any node is not more than one
2. The left subtree is balance
3. The right subtree is balanced

Source: <https://www.programiz.com/dsa/balanced-binary-tree>

AVL Rotations:

- **Left Rotation:** If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation .
- **Right Rotation:** AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.
- **Left-Right Rotation:** Double rotations are slightly complex versions of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.
- **Right- Left Rotation:** The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

Source: https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm

Red-Black Tree: Is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around $O(\log n)$ time, where n is the total number of elements in the tree.

Rules That Every Red-Black Tree Follows:

1. Every node has a colour either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.

Source: <https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/>

OpenCSV: It is a Java library that allows you to work with CSVs in a very simple way, both to read and to write them. Offer these classes:

- CSVReader: gives Operations to read the CSV file as a list of String arrays.
- CSVWriter: to write a CSV file.
- CsvToBean: reads the CSV and converts it to a Bean according to a MappingStrategy.

Source: [https://unpocodejava.com/2014/12/31/opencsv-trabajando-con-csv-de-forma-sencilla-conjava/#:~:text=OpenCSV%20\(http%3A%2F%2Fopencsv.sourceforge,lista%20de%20arrays%20de%20String.](https://unpocodejava.com/2014/12/31/opencsv-trabajando-con-csv-de-forma-sencilla-conjava/#:~:text=OpenCSV%20(http%3A%2F%2Fopencsv.sourceforge,lista%20de%20arrays%20de%20String.)

3. Search of creative solutions:

Alternatives:

1. We thought about how we could collect such a large amount of information through a csv file. To which we thought that the information of the players was separated by a character, we could do this with the bufferedReader and later with this information generate the players. These players can be saved in a linked list to later search a player by criteria like points, robberies, passes, etc.

2. We thought about looking for a tool (a library) that allows us to bring the information without so much process with different methods of the library we can get the information from the csv and then generate the players. We thought in implementation of Binary Search Tree, we can use the binary Search tree to save a determinate criteria and make the search more efficient. The node will have the key and value, the key will be the attribute and the

value can be the index of the player in the csv or a reference to this player. Also the implementation of AVL, With the Avl is the same idea of the binary search tree, this structure will help with saving a criteria and searching that criteria and get the search player, the difference is that when we insert a new player the whole tree will be ordered. And last the RBT in these trees will stay only the information that is not repeated in the trees, we can use an array of index to later get the players that share the same characteristic.

3. We thought about entering data into our program through an interface and saving them. We can create a User interface and the user of the program can Save data of the player, Modify Data or Delete his data. We can get the information of a csv file and with this information generate the players, these players can be saved in a linked list to later search a player by criteria like points, robberies, passes, etc. With LinkedList we can use binary search. The criteria that we are going to look for are all numeric, so the binary search could be implemented along the linked list and thus find that value, however the values can be repeated so it can be somewhat complex. In the LinkedList will stay all the information in the chosen data structures with the index of the player in the csv.

4. We first thought about how we could collect such a large amount of information through a csv file. To which we thought that the information of the players was separated by a character, we could do this with the bufferedReader and later with this information generate the players. These players can be saved in different Trees(AVL,BTS,RBT) so we insert all the information in determinate criteria in the trees with a reference to the player.

5. We thought about looking for a tool (a library) that allows us to bring the information without so much process with different methods of the library we can get the information from the csv and then generate the players. These players can be saved in a linked list to later search a player by criteria like points, robberies, passes, etc. Insert only the information that is not repeated in the LinkedList, we can use an array of index to later get the players that share the same characteristic.

6. We thought about looking for a tool (a library) that allows us to bring the information without so much process with different methods of the library we can get the information from the csv and then generate the players. We thought in implementation of Binary Search Tree, we could use the binary Search tree to save a determinate criteria and make the search more efficient. The node will have the key and value, the key will be the attribute and the value can be the index of the player in the csv or a reference to this player. Also the implementation of AVL, With the Avl is the same idea of the binary search tree, this structure will help with saving a criteria and searching that criteria and get the search player, the difference is that when we insert a new player the whole tree will be ordered. And last the RBT in these trees will stay all the players in the csv and insert all the information in the trees with a reference to the player.

4.Moving from ideas to preliminary designs:

Criteria:

- **Solution precision:** The solution is viable and meets the effective development of all the requirements of the problem

1. Imprecise
2. Almost Accurate
3. Accurate

- **Effectiveness:** Regardless of the complexity, the solution complies with a perfect effectiveness lacking any logical error

1. No Effective
2. Effective Medium
3. Effective

- **Usability:** The solution can be used because it meets all the proposed objectives

1. Not usable at all
2. Complex
3. Usable

- **Accessible:** The solution presents easy access to the data generated

1. Not accessible
2. Moderately accessible
3. Accessible

	Solution precision	Effectiveness	Usability	Accessible
Alternative 1	2	2	2	3
Alternative 2	2	3	3	3
Alternative 3	2	1	2	3

Alternative 1: 9 points

Alternative 2: 11 points

Alternative 3: 8 points

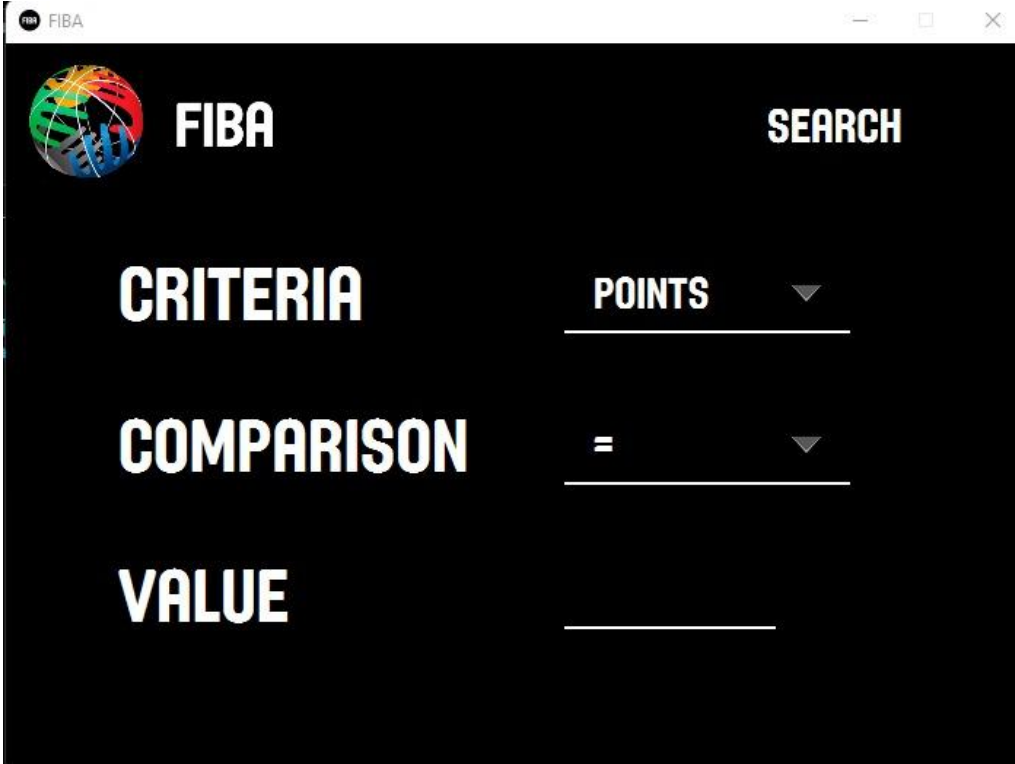
According to the previous evaluation, **Alternative 2** should be selected, since it obtained the highest score according to defined criteria.

5.The evaluation and selection of the preferred solution:

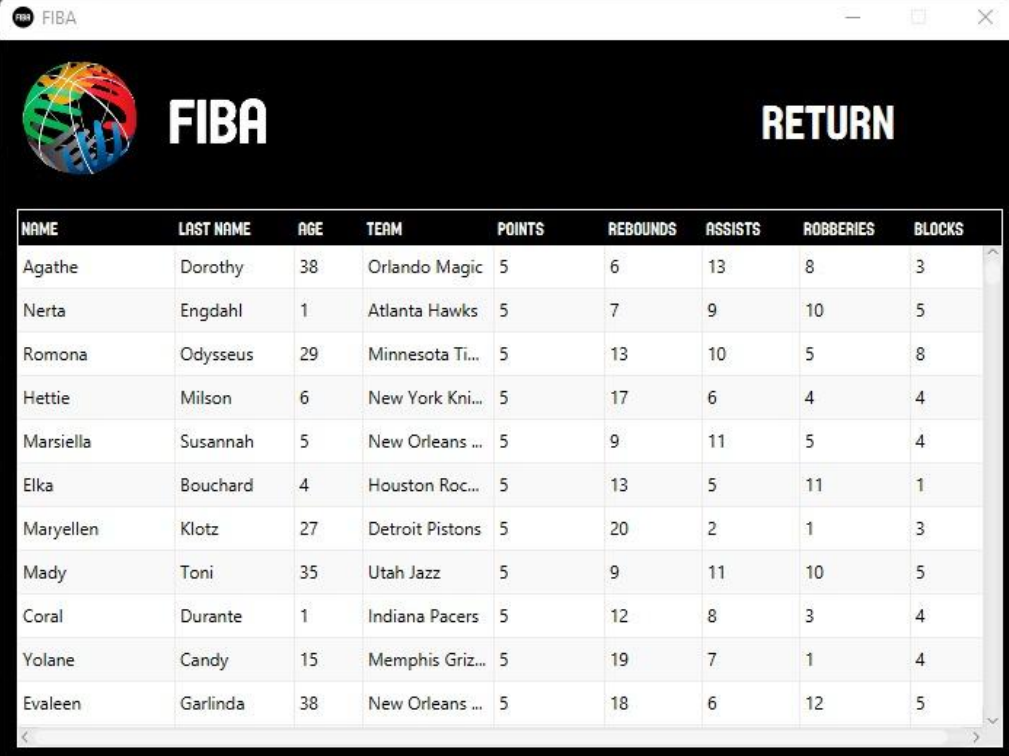
We choose the alternative 2 because by saving the information from a csv file, we are saving the information in secondary memory which will allow the program to work with large amounts of data, doing so with a library like OpenCsv will make data collection easier. We want the searches to be optimal, the most feasible way is to use the different trees and manage them by indexes so as not to create all the players. Having used other options such as handling it in a LinkedList or entering the data through an interface would take us more time than desired, for this reason the best option is alternative 2.

6.Design implementation:

First screen: in this screen the user is able to put the criteria (points, rebounds, assists, robberies, blocks, age), comparison (<, =, >) and value (positive integer number) in order to search for the player(s) that matches those requirements.

A screenshot of a web application window titled "FIBA". The interface has a dark background with white text. In the top left corner is the FIBA logo, a colorful globe. To its right is the word "FIBA" in large white capital letters. In the top right corner is the word "SEARCH" in white capital letters. Below this, there are three rows of input fields. The first row is labeled "CRITERIA" in large white capital letters, followed by a dropdown menu currently showing "POINTS". The second row is labeled "COMPARISON" in large white capital letters, followed by a dropdown menu currently showing "=". The third row is labeled "VALUE" in large white capital letters, followed by an empty text input field. Each input field has a horizontal line underneath it.

Second screen: in this screen the user is able see the player(s) that was searched by the criteria.



The screenshot shows a web application window with a black header. On the left is the FIBA logo, a colorful globe. To its right is the word 'FIBA' in white. Further right is the word 'RETURN' in white. Below the header is a table with 9 columns: NAME, LAST NAME, AGE, TEAM, POINTS, REBOUNDS, ASSISTS, ROBBERIES, and BLOCKS. The table contains 12 rows of player data. The window has standard OS controls (minimize, maximize, close) in the top right corner.

NAME	LAST NAME	AGE	TEAM	POINTS	REBOUNDS	ASSISTS	ROBBERIES	BLOCKS
Agathe	Dorothy	38	Orlando Magic	5	6	13	8	3
Nerta	Engdahl	1	Atlanta Hawks	5	7	9	10	5
Romona	Odysseus	29	Minnesota Ti...	5	13	10	5	8
Hettie	Milson	6	New York Kni...	5	17	6	4	4
Marsiella	Susannah	5	New Orleans ...	5	9	11	5	4
Elka	Bouchard	4	Houston Roc...	5	13	5	11	1
Maryellen	Klotz	27	Detroit Pistons	5	20	2	1	3
Mady	Toni	35	Utah Jazz	5	9	11	10	5
Coral	Durante	1	Indiana Pacers	5	12	8	3	4
Yolane	Candy	15	Memphis Griz...	5	19	7	1	4
Evaleen	Garlinda	38	New Orleans ...	5	18	6	12	5

TEST DESIGN

Stages

Name	Class	Stage
BSTStage1	BTS_Test	Create a BST
AVLStage1	AVL_Test	Create an AVL
RBTStage1	RBT_Test	Create a RBT
BSTStage2	BTS_Test	<p>Create a BST and insert 3 players</p> <p>Players:</p> <p>{“Jacob”, “xxx”, 4, “xxx”, 3, 3, 3, 3} {“Juan”, “xxx”, 3, “xx”, 3, 3, 3, 3} {“Sebastian”, “xxx”, 10, “xxx”, 3, 3, 3, 3}</p>
AVLStage2	AVL_Test	<p>Create a AVL and insert 3 players</p> <p>Players:</p> <p>{“Jacob”, “xxx”, 4, “xxx”, 8, 7, 3, 3, 3} {“Juan”, “xxx”, 3, “xx”, 6, 3, 3, 3, 3} {“Sebastian”, “xxx”, 10, “xxx”, 15, 9, 3, 3, 3}</p>
RBTStage2	RBT_Test	<p>Create a RBT and insert 3 players</p> <p>Players:</p> <p>{"Jacob", "xxx", 4, "xxx", 8, 7, 12, 3, 3} {"Juan", "xxx", 3, "xx", 6, 3, 10, 9, 3} {"Sebastian", "xx", 10, "xxx", 15, 9, 11, 12, 3} {"Santiago", "ss", 12, "sss", 10, 12, 5, 13, 3}</p>
FIBASage1	FIBA_Test	<p>Create a FIBA and insert 4 players</p> <p>Players:</p> <p>{"Jacob", "xxx", 4, "xxx", 8, 7, 12, 3, 3} {"Juan", "xxx", 3, "xx", 6, 3, 10, 9, 3} {"Sebastian", "xx", 10, "xxx", 15, 9, 11, 12, 3} {"Santiago", "ss", 12, "sss", 10, 12, 5, 13, 3} {"Mateo", “pp”, 23, “ppp”, 12, 4, 6, 5, 3}</p>

Test Cases

Test Objective: Validate the correct creation of a BST				
Class	Method	Stage	Input Values	Result
BST	BST(Constructor)	BSTStage1	Comparator<Player> weight = 0 root = null	BST created; constructor method works correctly

Test Objective: Validate the correct insertion of a node in BST				
Class	Method	Stage	Input Values	Result
BST	insert(Node<Player>)	BSTStage1	Node<Player> Name = "Jacobo" Last Name = "xxx" Age = 4 Team = "xxx" Points = 3 Rebounds = 3 Assists = 3 Robberies = 3 Blocks = 3	New node inserted in BST; the method insert works correctly
BST	insert(Node<Player>)	BSTStage1	Node<Player> Name = "Juan" Last Name = "xxx" Age = 3 Team = "xxx" Points = 3 Rebounds = 3 Assists = 3 Robberies = 3 Blocks = 3	New node inserted in BST; the method insert works correctly
BST	insert(Node<Player>)	BSTStage1	Node<Player> Name = "Sebastian" Last Name = "xx" Age = 10 Team = "xxx" Points = 3 Rebounds = 3 Assists = 3 Robberies = 3 Blocks = 3	New node inserted in BST; the method insert works correctly

Test Objective: Validate the correct search of a node in BST

Class	Method	Stage	Input Values	Result
BST	Search (Player)	BSTStage 2	Node<Player> Name = "xxx" Last Name = "xxx" Age = 11 Team = "sdfsdf" Points = 3 Rebounds = 3 Assists = 3 Robberies = 3 Blocks = 3	Node not founded, NullPointerException is thrown
BST	Search (Player)	BSTStage 2	Node<Player> Name = "Sebastian" Last Name = "xx" Age = 10 Team = "xxx" Points = 3 Rebounds = 3 Assists = 3 Robberies = 3 Blocks = 3	Node founded, return the value of the Node<Player>

Test Objective: Validate the correct remove of a node in BST

Class	Method	Stage	Input Values	Result
BST	Delete (Player)	BSTStage 2	Node<Player> Name = "Sebastian" Last Name = "xx" Age = 10 Team = "xxx" Points = 3 Rebounds = 3 Assists = 3 Robberies = 3 Blocks = 3	Node deleted; associations re-established. The new BST root is the node with value = 3

Test Objective: Validate the correct insertion of a node in AVL

Class	Method	Stage	Input Values	Result
AVL	insert Player)	AVLStage 1	Node<Player> Name = "Jacobo" Last Name = "xxx" Age = 4 Team = "xxx" Points = 8 Rebounds = 7 Assists = 3 Robberies = 3 Blocks = 3	New node inserted in AVL; Insert method, balancing and rotations work correctly.
AVL	insert Player)	AVLStage 1	Node<Player> Name = "Juan" Last Name = "xxx" Age = 3 Team = "xxx" Points = 6 Rebounds = 3 Assists = 3 Robberies = 3 Blocks = 3	New node inserted in AVL; Insert method, balancing and rotations work correctly.
AVL	insert Player)	AVLStage 1	Node<Player> Name = "Sebastian" Last Name = "xx" Age = 10 Team = "xxx" Points = 15 Rebounds = 9 Assists = 3 Robberies = 3 Blocks = 3	New node inserted in AVL; Insert method, balancing and rotations work correctly.
AVL	Insert Player)	AVLStage 2	Node<Player> Name = "Santiago" Last Name = "ss" Age = 10 Team = "sss" Points = 12 Rebounds = 10 Assists = 3 Robberies = 3 Blocks = 3	New node inserted in AVL; Insert method, balancing and rotations work correctly.

Test Objective: Validate the correct insertion of a node in RBT

Class	Method	Stage	Input Values	Result
RBT	insertNode (Player)	RBTStage 1	Node<Player> Name = "Jacobo" Last Name = "xxx" Age = 4 Team = "xxx" Points = 8 Rebounds = 7 Assists = 12 Robberies = 3 Blocks = 3	New node inserted in RBT; Insert method and rotations work correctly.
RBT	insertNode (Player)	RBTStage 1	Node<Player> Name = "Juan" Last Name = "xxx" Age = 3 Team = "xxx" Points = 6 Rebounds = 3 Assists = 10 Robberies = 9 Blocks = 3	New node inserted in RBT; Insert method and rotations work correctly.
RBT	insertNode (Player)	RBTStage 1	Node<Player> Name = "Sebastian" Last Name = "xx" Age = 10 Team = "xxx" Points = 15 Rebounds = 9 Assists = 11 Robberies = 12 Blocks = 3	New node inserted in RBT; Insert method and rotations work correctly.
RBT	insertNode (Player)	RBTStage 1	Node<Player> Name = "Santiago" Last Name = "ss" Age = 12 Team = "sss" Points = 10 Rebounds = 12 Assists = 5 Robberies = 13 Blocks = 3	New node inserted in RBT; Insert method and rotations work correctly.

RBT	insertNode (Player)	RBTStage 2	Node<Player> Name = "Mateo" Last Name = "pp" Age = 23 Team = "ppp" Points = 12 Rebounds = 4 Assists = 6 Robberies = 5 Blocks = 3	New node inserted in RBT; Insert method and rotations work correctly.
-----	---------------------	------------	---	---

Test Objective: Validate the correct search for a player according to the criteria selected in FIBA.				
Class	Method	Stage	Input Values	Result
FIBA	Search (Criteria, comparison, value)	FIBASTage1	Criteria = "Points" Comparison = ">" Value = 12	Player founded (Sebastian), return a List with the player in the position 0.
FIBA	Search (Criteria, comparison, value)	FIBASTage1	Criteria = "Robberies" Comparison = "=" Value = 13	Player founded (Santiago), return a List with the player in the position 0.